# BI as an Assertion Language for Mutable Data Structures

Samin Ishtiaq Peter W. O'Hearn

Queen Mary & Westfield College, London

## ABSTRACT

Reynolds has developed a logic for reasoning about mutable data structures in which the pre- and postconditions are written in an intuitionistic logic enriched with a spatial form of conjunction. We investigate the approach from the point of view of the logic BI of bunched implications of O'Hearn and Pym. We begin by giving a model in which the law of the excluded middle holds, thus showing that the approach is compatible with classical logic. The relationship between the intuitionistic and classical versions of the system is established by a translation, analogous to a translation from intuitionistic logic into the modal logic S4. We also consider the question of completeness of the axioms. BI's spatial implication is used to express weakest preconditions for object-component assignments, and an axiom for allocating a cons cell is shown to be complete under an interpretation of triples that allows a command to be applied to states with dangling pointers. We make this latter a feature, by incorporating an operation, and axiom, for disposing of memory. Finally, we describe a local character enjoyed by specifications in the logic, and show how this enables a class of frame axioms, which say what parts of the heap don't change, to be inferred automatically.

## 1. INTRODUCTION

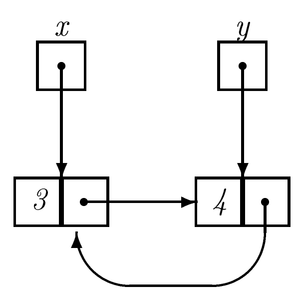
Pointers are an extremely powerful and flexible programming mechanism, useful for manipulating linked data structures and for providing structured access to data in memory. They are also extremely dangerous. Pointer-manipulating programs are notoriously difficult to get right, and even lead to runtime safety violations (such as from dereferencing `nil` or a disposed pointer) which lie beyond the range of conventional type systems. An effective program-proving formalism for dealing with pointers would be most welcome.

But pointers have also always been one of the thorny patches of program proving. The most immediate issue to face is that the Hoare substitution-oriented treatment of assignment

$$\{P[E/x]\}x := E\{P\}$$

does not cope with component assignments of the form $E.i := E'$ (or $E \to i = E'$ in C syntax) that alter the heap. Other issues are raised by operations for allocating and, especially, disposing of memory. A number of researchers have developed program-proving formalisms for pointers (e.g., [16, 30, 23, 17, 22, 3, 6]), but no definitive solution has emerged as of yet. Most importantly, lying behind technicalities with axioms for assignment and storage management is a deeper difficulty, the "complexity of pointer swing [15]" that results from *aliasing*: there can be more than one pointer to a cell that is altered, in which case assignment to the cell affects seemingly unrelated expressions. The real problem is to control, or understand, this complexity, rather than simply to axiomatize it.

A striking advance has been recently made by Reynolds [35], building on early work of Burstall [5]. The main novelty is the use of a spatial form of conjunction $P * Q$, that splits the heap into distinct portions that the different conjuncts talk about. In addition, there is a form of assertion, the points-to relation $\mapsto$, which is used to make statements about the contents of heap cells. For instance, the spatial conjunction $(x \mapsto 3, y) * (y \mapsto 4, x)$ says that $x$ and $y$ denote distinct locations, where the cdr of $x$ is a pointer to $y$, the cdr of $y$ is a pointer to $x$, and where the car's contain 3 and 4.




The combination of $*$ and $\mapsto$ leads to remarkably simple axioms. In particular, when an assertion of the form $P * (x \mapsto a, b)$ holds prior to a component assignment $x.1 := z$ we know that the assignment cannot affect $P$, and so $P * (x \mapsto z, b)$ will hold on conclusion. The logic of pointer swing is treated in a local way that mirrors the intuitive operational locality of assignment.

In this paper we investigate the approach from the point of view of the logic BI of O'Hearn and Pym [25]. The most

To appear in the Proceedings of the 28th ACM-SIGPLAN Symposium on Principles of Programming Languages, London, January 2001

distinctive feature of BI is its joint treatment of two implication connectives. One implication, $\Rightarrow$, is from standard intuitionistic or classical logic, while the other, $-\!\!*$, is the implication for a basic substructural logic. Reynolds's assertion language is already substructural: it adds a Contraction-free conjunction, where $P$ and $P*P$ are not generally equivalent, to intuitionistic logic: BI's $-\!\!*$ is related to $*$ by the deduction theorem, which states that a consequence $A \models B -\!\!* C$ holds iff $A * B \models C$ does. There is also a version of the deduction theorem which relates $\Rightarrow$ and the usual conjunction $\wedge$.

The basic idea of BI's semantics is to allow statements to be made about the world using familiar connectives such as $\Rightarrow$, $\neg$ and $\wedge$, and then to combine these statements in a modular way using $*$ and $-\!\!*$. The key to this is the resource interpretation of the connectives, where $*$ decomposes the current resource into pieces and $-\!\!*$ talks about new or fresh resource [25, 26]. Substructural logics may appear rather exotic beasts. But the pointer models we present provide a very concrete and, we believe, intuitive way of understanding the connectives, quite apart from overtly logical concerns.

With this as background, we now describe the main contributions of the paper. These fall under three headings: (i) Classical versus Intuitionistic; (ii) Completeness Issues; (iii) Local Reasoning.

*Classical or Intuitionistic?.* Reynolds used an intuitionistic interpretation of pre- and postconditions in his logic. This was presented in a possible worlds style, which treats negation and implication by quantifying over all extensions to the current heap [19], and gives rise to a monotonicity property where all propositions are invariant under heap extension. Although the intuitionistic semantics is intuitive, it seemed to us reasonable to ask: might the law of the excluded middle be compatible with the axioms for assignment and other statements? Reynolds gave an example which indicates that his axioms are unsound under a "straightforward" classical reading, but left open the question of whether a different classical semantics might be possible.

We answer by presenting a classical model, which is a possible worlds model of Boolean BI (where $\neg$ and $\Rightarrow$ are classical), and which validates all of the axioms for Hoare triples. In this model the worlds are heaps (collections of cons cells in storage), and the conjunction $P*Q$ is true just when the current heap can be split into two components, one of which makes $P$ true and the other of which makes $Q$ true. The implication $P -\!\!* Q$ talks about new or fresh pieces of heap, disjoint from the current heap. It says that, whenever we are given new heap that makes $P$ true, the combined new and current heap will make $Q$ true. The other connectives are interpreted pointwise in this model; for example $\neg P$ is true of a world just if $P$ is not true of that world.

Perhaps more significant than excluded middle is that the classical semantics is more expressive than the intuitionistic one. In particular, it allows the specification of exact properties of the heap, such as "the heap is empty", which cannot be expressed in the intuitionistic semantics because they are not invariant under heap extension.

We work with the classical semantics for most of the paper, but consider the relation to the intuitionistic model in Section 9. We describe a translation from intuitionistic to classical which is similar to a standard translation from intuitionistic logic to the modal logic S4, thereby showing that all intuitionistic properties can be expressed within the classical setup.

*Completeness Issues.* Although the treatment of assignment given by Reynolds, and by Burstall, is very elegant, at first sight its simplicity appears to come at the price of forcing assertions to be written in a stylized form, which would not allow certain programs to be verified. This would perhaps be a price worth paying, but we show that the weakest precondition can in fact be expressed. The crucial point for this is an interesting interaction between the $-\!\!*$ and $*$ connectives. For example, we will explain in Section 3.1 how

$$(x \mapsto 3,5) \, * \, ((x \mapsto 7,5) -\!\!* P)$$

says that $x$ points to a cell holding $\langle 3,5\rangle$, but also that if we update the car to 7 then $P$ will be true. We would expect this to be a valid precondition for a postcondition $P$ with assignment statement $x.1 := 7$, where the indicated assignment sets the first component of (the cons cell denoted by) $x$ to 7.

We show how the weakest precondition for each atomic statement can be expressed in the logic. The semantics used for this result is based on an interpretation of triples that allows commands to be applied to states with dangling pointers. Dangling pointers also play an essential role in the interpretations of $-\!\!*$ and $*$. We make this a feature by considering an operation that disposes of memory (thereby creating dangling pointers). Since disposing memory is such a devastatingly effective method of introducing programming errors, we were pleasantly surprised to find that the approach allows for a simple axiom which enables programs with disposal to be proven. The semantics of triples we use is one that supports the slogan *well-specified programs don't go wrong*, where going wrong could result from, say, dereferencing `nil` or a disposed pointer.

The classical semantics presented in this paper came after the intuitionistic semantics, and we must admit that it took some time to get used to. (The intuitionistic semantics was discovered independently by us, while we were working from an early version of [35].) Ultimately, the classical model seemed natural only after we had the courage to consider disposal, where it is essential to be able to specify memory utilization exactly. Reynolds has since been braver still, working with a generalization of the logic that encompasses pointer arithmetic [36].

*Local Reasoning.* Above we mentioned the local way that pointer swing is treated. We examine the sense in which local reasoning extends to larger-scale operations. In fact, one of the most promising suggestions in the approach of Reynolds and Burstall is that verifications might be done in a way that scales well, by localizing the effects of heap-altering commands to certain of the conjuncts in an assertion $P_1 * \cdots * P_n$.

We investigate this idea by formulating a rule for automatically inferring certain frame axioms, which describe invariants of the heap. Traditionally, an inordinate amount of effort needs to be spent specifying what a program doesn't change, so much so that these frame axioms distract from the main concern – what changes. In the absence of pointers what doesn't change can be succinctly summarized using modifies clauses [14], which list the program variables corresponding to locations that can be altered by a program. But for pointers, which may include links to cells not named by variables in the program, the problem is much more acute; we show how the conjunction $*$ can be used to derive such axioms. The point is that this allows specifications to be kept "small", where they describe only the area of the heap that a program actually acts on. Invariant properties for other areas of the heap come for free.

## 2. COMMANDS AND BASIC DOMAINS

The imperative language that Reynolds deals with is a simple command language, with Lisp-like expressions for accessing and creating cons cells. We will not give a full syntax of commands, as the treatment of conditionals and looping statements is standard. Instead, we will concentrate on assignment statements, which is where the main novelty of the approach lies.

The commands we consider are as follows.

$$\begin{array}{rcl} C & ::= & x := E \\ & | & x := E.i \\ & | & E.i := E' \\ & | & x := \mathtt{cons}(E_1, E_2) \\ & | & \vdots \\ i & ::= & 1 \mid 2 \end{array}$$

Here, each of the $E$'s is a pure expression; that is, $E$ does not contain a dot. In $E.i$ the $i$ is assumed to be one of the constants 1 or 2 (the extension to varying length records, or named alternatives, is straightforward). The second and third assignment statements read and update the heap, respectively. The fourth creates a new cons cell in the heap, and places a pointer to it in $x$.

Notice that these commands do not directly handle double-dereferencing, such as $x.1.2$, where one looks more than one-deep into the heap. One would have to break a use of such an expression, either on the left or right of :=, into several steps, possibly using auxiliary variables.[1]

An expression can denote an integer, an atom, or a cons cell.

$$\begin{array}{rcll} E & ::= & x & \text{Variable} \\ & | & 42 & \text{Integer} \\ & | & \mathtt{nil} & \text{nil} \\ & | & a & \text{atom} \\ & | & \cdots & \end{array}$$

We have not given a full expression syntax; the only constraint is that an expression can be interpreted in the semantic domain specified below.

We use the following semantic domains, which are as in [35] (except for our restriction to binary cells, which is not essential).

$$\begin{array}{rcl} Val & = & Int \cup Atoms \cup Loc \\ S & = & Var \rightharpoonup_{fin} Val \\ H & = & Loc \rightharpoonup_{fin} Val \times Val \end{array}$$

Here, $Loc = \{\ell, ...\}$ is an infinite set of locations, $Var = \{x, y, ...\}$ is a set of variables, $Atoms = \{\mathtt{nil}, a, ...\}$ is the set of atoms, and $\rightharpoonup_{fin}$ is for finite partial functions. We call an element $s \in S$ a stack, and $h \in H$ a heap. There is a deliberate distinction between the two: stack variables are maintained according to a stack discipline and are not allowed to alias one another; heap variables or pointers do not obey a stack discipline. [We will not include an explicit operation for allocating stack variables.]

We use $dom(h)$ to denote the domain of definition of a heap $h \in H$, and $dom(s)$ to denote the domain of a stack $s \in S$.

An expression is interpreted as a heap-independent value

$$[\![E]\!]s \quad \in \quad Val$$

where the $dom(s)$ includes the free variables of $E$.

The commands are interpreted using a relation $\rightsquigarrow$ on configurations, where the configurations include triples $C, s, h$ and terminal configurations $s, h$, for $s \in S$ and $h \in H$. We assume the semantics of expressions to specify $\rightsquigarrow$.

In the following rules we use $r$ to range over elements of $Val \times Val$, $\pi_i r$ for the first or second projection, and $(r \mid i \mapsto v)$ to indicate the pair like $r$ except that the $i$'th component is replaced with $v$.

$$\frac{[\![E]\!]s = v}{x := E, s, h \rightsquigarrow [s \mid x \mapsto v], h}$$

$$\frac{[\![E]\!]s = \ell \in Loc \quad h(\ell) = r}{x := E.i, s, h \rightsquigarrow [s \mid x \mapsto \pi_i r], h}$$

$$\frac{[\![E]\!]s = \ell \in Loc \quad h(\ell) = r \quad [\![E']\!]s = v'}{E.i := E', s, h \rightsquigarrow s, [h \mid \ell \mapsto (r \mid i \mapsto v')]}$$

$$\frac{\ell \in Loc \quad \ell \notin dom(h) \quad [\![E_1]\!]s = v_1\,,\ [\![E_2]\!]s = v_2}{x := \mathtt{cons}(E_1, E_2), s, h \rightsquigarrow [s \mid x \mapsto \ell], [h \mid \ell \mapsto \langle v_1, v_2 \rangle]}$$

The location $\ell$ in the fourth case is not specified uniquely, so a new location is chosen non-deterministically. We can also include typical rules for sequencing, looping, etc. The relation $\rightsquigarrow$ is a one-step semantics, and these other constructs would give rise to non-terminal configurations. We say that

- "$C, s, h$ is stuck" in case there is no configuration $K$ such that $C, s, h \rightsquigarrow K$, and
- "$C, s, h$ is safe" in case $C, s, h \rightsquigarrow^* K$ implies that $K$ is a terminal configuration $s', h'$ or is not stuck.

Being stuck is a kind of runtime error. For instance, a command can get stuck by an attempt to dereference nil or an integer. Note also that the semantics allows dangling references, as in the stack $[x \mapsto \ell]$ with empty heap $[]$. The assignment $x.1 := 2$ is stuck for this stack and heap.

This definition of safety is formulated with partial correctness in mind: with loops $C, s, h$ could fail to converge to a terminal configuration without becoming stuck.

## 3. A MODEL OF BOOLEAN BI

The pre- and postconditions for commands will be written using the following formulae.

$$\begin{array}{rcll} P, Q, R & ::= & \alpha & \text{Atomic Formulae} \\ & | & \mathtt{false} & \text{Falsity} \\ & | & P \Rightarrow Q & \text{Classical Implication} \\ & | & \mathtt{emp} & \text{Empty Heap} \\ & | & P * Q & \text{Spatial Conjunction} \\ & | & P \mathrel{-\!\!*} Q & \text{Spatial Implication} \\ & | & \exists x.P & \text{Existential Quantification} \end{array}$$

[1] This restriction is similar to the form of assignment statements sometimes used in intermediate languages for static analysis of pointer programs.

This syntax differs from that of Reynolds in three ways. First, we consider the substructural implication $-\!\!*$ and unit `emp` from BI. The unit was not needed in [35] because in the intuitionistic semantics the unit of $*$ is `true` (this is because Weakening for $*$ is present). Second, we use the BI symbol $*$ instead of & for the spatial conjunction. Finally, because we are in a Boolean situation we can define various other connectives as usual, rather than taking them as primitive: $\neg P \;=\; P \Rightarrow \texttt{false}$; $\texttt{true} \;=\; \neg(\texttt{false})$; $P \vee Q \;=\; (\neg P) \Rightarrow Q$; $P \wedge Q \;=\; \neg(\neg P \vee \neg Q)$; $\forall x.\, P \;=\; \neg\exists x.\,\neg P$.

The set $free(P)$ of free variables of a formula is defined as usual, as is the capture-avoiding substitution $P[E/x]$.

The atomic formulae include an equality relation and the points-to relation.

$$\begin{array}{rcll} \alpha & ::= & E = E' & \text{Equality} \\ & \mid & E \mapsto E_1, E_2 & \text{Points to} \\ & \mid & \cdots & \end{array}$$

In practice, one would also want atomic predicates describing inductive properties of the heap, or a recursive facility which allows such properties to be defined.

## 3.1 Semantic Clauses

The semantics of assertions is given by a forcing relation of the form

$$s, h \models P$$

which asserts that $P$ is true of stack $s \in S$ and heap $h \in H$. It is required that $dom(s) \supseteq free(P)$. The semantics is organized in a possible worlds style, where the heaps are the worlds. We use the following notation in formulating the semantics:

- $h \# h'$ indicates that the domains of heaps $h$ and $h'$ are disjoint;
- $h \cdot h'$ denotes the union of disjoint heaps (i.e., the union of functions with disjoint domains).

Here are the semantic clauses.

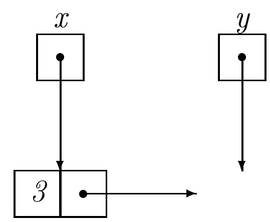
$$\begin{array}{lll}
s,h \models E = E' & \text{iff} & [\![E]\!]s = [\![E']\!]s \\[1ex]
s,h \models E \mapsto E_1, E_2 & \text{iff} & \{[\![E]\!]s\} = dom(h) \\
 & & \text{and } h([\![E]\!]s) = \langle [\![E_1]\!]s, [\![E_2]\!]s \rangle \\[1ex]
s,h \models \texttt{emp} & \text{iff} & h = [\,] \text{ is the empty heap} \\[1ex]
s,h \models P * Q & \text{iff} & \exists h_0, h_1.\ h_0 \# h_1,\ \ h_0 \cdot h_1 = h, \\
 & & \qquad s, h_0 \models P \text{ and } s, h_1 \models Q \\[1ex]
s,h \models P -\!\!* Q & \text{iff} & \forall h'.\, \text{if } h' \# h \text{ and } s, h' \models P \text{ then} \\
 & & \qquad s, h \cdot h' \models Q \\[1ex]
s,h \models \texttt{false} & & \text{never} \\[1ex]
s,h \models P \Rightarrow Q & \text{iff} & \text{if } s,h \models P \text{ then } s,h \models Q \\[1ex]
s,h \models \exists x.P & \text{iff} & \exists v \in Val.\, [s \mid x \mapsto v], h \models P
\end{array}$$

The points-to relation $E \mapsto E_1, E_2$ looks one-deep into the heap. In the classical semantics it is interpreted "exactly", by requiring that that $E$ denotes *the only* cell in the current heap. The semantics is flexible here, in allowing $E_i$ in $E \mapsto E_1, E_2$ to denote a location that is not in the domain of $h$. For example, in

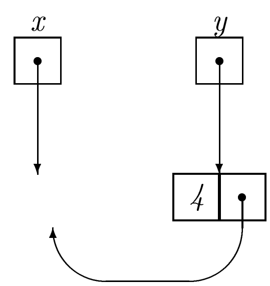
$$[x \mapsto \ell, y \mapsto \ell'], [\ell \mapsto \langle 2, \ell' \rangle] \models (x \mapsto 2, y)$$

the location $\ell'$ is dangling, which is to say that it is not in the domain of the heap.

The conjunction $P * Q$ is true just when the current heap can be decomposed into two constituents in a way that makes $P$ true of one constituent and $Q$ true of the other. With this definition, $(x \mapsto 3, y) * (y \mapsto 4, x)$ corresponds to the box-and-pointer diagram from the Introduction. Notice the importance of dangling pointers here: the store corresponding to the left conjunct is




while that for the right is




This model differs from the one in [35] in three ways. First, the implication $\Rightarrow$ is interpreted in a pointwise fashion, which results in a classical semantics. This is a semantics which uses the boolean algebra structure of the powerset of $H$, rather than the 2-element boolean algebra. Second, we include `emp` and $-\!\!*$. And third, the points-to relation is interpreted exactly, where $s, h \models E \mapsto E_1, E_2$ does not imply $s, h' \models E \mapsto E_1, E_2$ whenever $h'$ is a bigger heap than $h$ (bigger in the sense of inclusion of partial functions).

We can express an inexact variant of points-to as follows

$$E \hookrightarrow E_1, E_2 \;\;=\;\; (\texttt{true} * E \mapsto E_1, E_2).$$

Generally, $\texttt{true} * P$ says that $P$ is true of some heap contained in the current one. Conversely, if we were to take $\hookrightarrow$ as primitive then we could define $\mapsto$ in terms of it using the formula

$$E \hookrightarrow E_1, E_2 \;\wedge\; \neg\big((\neg\texttt{emp}) * (E \hookrightarrow E_1, E_2)\big).$$

The different way that the two conjunctions $*$ and $\wedge$ behave is illustrated by the following examples.

1. $(x \mapsto 1, 2) * (x \mapsto 1, 2)$ is never true, because, however the heap is split up, $x$ will be left dangling in one of the conjuncts.
2. $(x \mapsto 1, 2) \wedge (x \mapsto 1, 2)$ is equivalent to $x \mapsto 1, 2$, and so is true in the singleton heap where $x$ points to $\langle 1, 2 \rangle$.
3. $(x \mapsto 1, 2) * \neg(x \mapsto 1, 2)$ can be true when $x$ points to a cell holding $\langle 1, 2 \rangle$ in the current heap, because the heap can then be split into a singleton where $(x \mapsto 1, 2)$ and another heap where $x$ is dangling, thus making $\neg(x \mapsto 1, 2)$ true.

4. $(x \mapsto 1,2) \wedge \neg(x \mapsto 1,2)$ is never true.

The difference between $\hookrightarrow$ and $\mapsto$ shows up in the presence or absence of Weakening for $*$.

1. $P * (x \mapsto 1,2) \Rightarrow (x \mapsto 1,2)$ is not always true, for instance when the antecedent is true of a heap with more than one defined location.

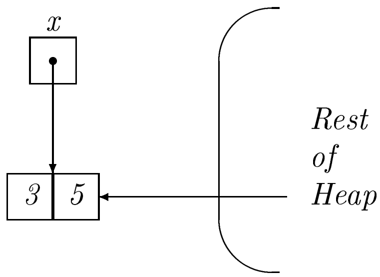
2. $P * (x \hookrightarrow 1,2) \Rightarrow (x \hookrightarrow 1,2)$ is always true.

A crucial ingredient in the semantics of $-\!*$ is the requirement $h' \# h$, which has the effect of ensuring that $h'$ is a new or fresh piece of heap. That is, its domain of definition must be disjoint from the domain of the current heap $h$.

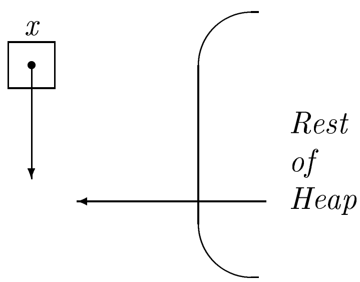
We can now explain the example

$$(x \mapsto 3,5) \; * \; \big((x \mapsto 7,5) -\!* P\big)$$

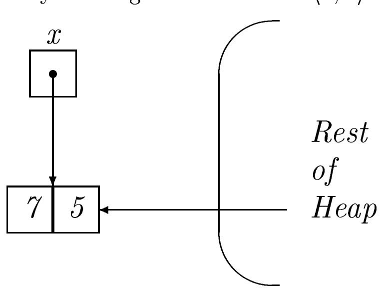
from the Introduction. We claim that this formula says that $x$ denotes a cell which holds $\langle 3,5 \rangle$ in the current heap, but also that if we update the car to 7 then $P$ will be true. To see why, first note that the semantics of $*$ splits the heap, say,




into two portions, one where $(x \mapsto 3,5)$ holds and a second heap where the location denoted by $x$ is dangling:




We have included a dangling pointer out of the rest of the heap here to emphasize that the location might be referenced from within a heap cell, as well as from $x$. Because the association $(x \mapsto 3,5)$ has been, in a sense, retracted by deleting the association from the heap in the right conjunct, this frees $-\!*$ to extend the second heap with a different contents for the location denoted by $x$. The semantics of $-\!*$ and $\mapsto$ then ensure that $P$ must be true when this second heap is extended by binding $x$'s location to $\langle 7,5 \rangle$.




So, the intuitive description in terms of updating follows from several steps in the semantics, which add up to "update as deletion followed by extension". (We stress that $x$ denotes the same location at each step in this narrative , even when that location is dangling; the update expressed is to the heap, not the stack.)

## 3.2 Properties

The semantic consequence relation $P \models Q$ between formulae is defined to hold iff for all $s, h$, if $s, h \models P$ then $s, h \models Q$. This assumes that $dom(s) \supseteq free(P) \cup free(Q)$.

PROPOSITION 1. *The usual rules of classical logic are sound for $\models$, along with*

$*$ *is commutative and associative, with unit* `emp`

$$\frac{P' \models P \quad Q' \models Q}{P' * Q' \models P * Q}$$

$$\frac{R * P \models Q}{R \models P -\!* Q} \qquad\qquad \frac{R \models P -\!* Q \quad R' \models P}{R * R' \models Q}$$

In particular, note that two versions of the deduction theorem hold at the same time:

$$R \models P -\!* Q \quad \text{iff} \quad R * P \models Q$$
$$R \models P \Rightarrow Q \quad \text{iff} \quad R \wedge P \models Q.$$

In [25], these properties were taken as the basis for a natural deduction presentation of BI, where contexts were *bunches*: trees built from two kinds of combining operator, one corresponding to $*$ and the other to $\wedge$. That presentation is (after we add reductio ad absurdum) equivalent, in terms of provability, to the bunch-free presentation stated in the proposition. The model in this section is a possible worlds model for Boolean BI [25, 26].

Because we do not have Weakening ($P * Q \models P$) or Contraction ($P \models P * P$) for $*$, we are in the territory of *substructural logic*. To see why Contraction fails, consider $x \hookrightarrow 2,3$. It can be satisfied in a heap with a cons cell whose contents is $\langle 2,3 \rangle$, but $(x \hookrightarrow 2,3) * (x \hookrightarrow 2,3)$ is false for every heap. To see why Weakening fails, consider $(x \mapsto 2,3) * (y \mapsto 4,5)$. For this to be true the current heap must have size two, and $(x \mapsto 2,3)$ cannot then hold because it requires the current heap to have size one.

The importance of restricting Contraction was brought to the fore by linear logic [12, 13]. But it is important to realize that BI takes a very different approach to the surrounding additive connectives. To see this, consider that $P \multimap Q \models P \Rightarrow Q$ always holds in linear logic, using the decomposition $P \Rightarrow Q \; = {!P} \multimap Q$ and the rule of Dereliction for !. But here,

$$(x \mapsto 1,2) -\!* \texttt{false} \not\models (x \mapsto 1,2) \Rightarrow \texttt{false}$$

because the antecedent can hold in a heap where $x \mapsto 1,2$ while the consequent cannot.

This shows that there can be no ! which decomposes $P \Rightarrow Q$ into ${!P} -\!* Q$ in this model; this highlights the difference between the joint treatment of $-\!*$ and $\Rightarrow$ in the model and the approach of linear logic. Furthermore, it is not unusual to use additive implications where $\mapsto$ appears to the left. An example is when specifying that any defined location in the heap is reachable; such a specification would be of the form $\forall x.\,(\exists ab.\, x \mapsto a,b) \Rightarrow \cdots)$ (where to fill in the $\cdots$ we could use an appropriate inductive definition).

Next, we consider the notion of purity.

> *Purity*. We say that an assertion is *syntactically pure* if it does not contain $\mapsto$ or $I$.

Recall that we do not have terms of the form $E.i$ for field selection within assertions: $\mapsto$ is the only way that an assertion might look into the heap.

PROPOSITION 2. *Any synactically pure assertion is independent of the heap: If $P$ is a pure assertion then*

$$s,h \models P \text{ iff } s,h' \models P.$$

*As a result, pure assertions are completely additive: if $P$ and $Q$ are pure, then*

> *$P * Q$ and $P \wedge Q$ are equivalent;*
>
> *$P \mathrel{-\!\!*} Q$ and $P \Rightarrow Q$ are equivalent.*

The first of these properties indicates a formal similarity between purity and "!" in linear logic [12]: we get Contraction $P \models P * P$ and Weakening $P * Q \models P$ for pure propositions. (This remark is independent of the issue of decomposing $\Rightarrow$ into $\mathrel{-\!\!*}$.) The second property shows a further similarity with passivity in syntactic control of interference [34], where additive and multiplicative function type constructs agree on passive types [28].

## 3.3 Interpretation of Triples

Hoare triples are of the form $\{P\}\,C\,\{Q\}$, where $P$ and $Q$ are assertions as above and $C$ is a command. We adopt an interpretation which ensures that well-specified commands do not get stuck.

> $\{P\}C\{Q\}$ is true just when
>
> > if $s,h \models P$ then $C,s,h$ is safe and if $C,s,h \rightsquigarrow^* s',h'$ then $s',h' \models Q$
>
> for all $s,h$ where $dom(s) \supseteq free(P) \cup free(Q)$.

This is a partial correctness interpretation; with looping, it would not guarantee termination. However, the safety requirement rules out certain runtime errors and, as a result, we do not have that $\{\mathtt{true}\}C\{\mathtt{true}\}$ holds for all commands. For example, $\{\mathtt{true}\}x := \mathtt{nil}; x.1 := 3\{\mathtt{true}\}$ fails. Generally, if we can establish $\{P\}C\{\mathtt{true}\}$ then we will know that $C$ is safe to execute in any state satisfying $P$.

# 4. THE REYNOLDS AXIOMS

We start with standard Hoare rules for sequencing, consequence and simple assignment.

*Sequencing*

$$\frac{\{P\}C\{Q\} \quad \{Q\}C'\{R\}}{\{P\}C;C'\{R\}}$$

*Consequence*

$$\frac{P \models P' \quad \{P'\}C\{R'\} \quad R' \models R}{\{P\}C\{R\}}$$

*Simple Assignment*

$$\{P[E/x]\}x := E\{P\}$$

In the Consequence rule, $\models$ refers to the semantic consequence relation for assertions. (Equivalently, we could replace $\models$ by $\Rightarrow$, and ask that the resulting implications hold in every state in which the stack component binds all variables in the involved formulae.)

The first heap-accessing command is the statement $x := E.i$, which can read from both the stack and the heap, but which only alters the stack. Here there are two things to keep in mind. First, $E$ will be a pure expression, which doesn't look into the heap. So, we will not consider an assignment statement like $x := y.1.2$, which would have to be broken into two steps. Second, we will expect $E.i$ to be determined by an assertion of the form $E \hookrightarrow E_1, E_2$, which lets us find its value.

*Object-component Lookup*

> Suppose that the variables $x_1, x_2$ are not free in $E$, and that $x_1$ does not occur free in $P$. Then
>
> $$\begin{array}{l}\{\exists x_1.\, P[x_1/x] \wedge \exists x_2.\, E \hookrightarrow x_1, x_2\}\\ x := E.1\\ \{P\}\end{array}$$

The substitution in $P[x_1/x]$ is saying that $P$ is true in the postcondition, similarly to the simple assignment axiom, but there is also additional information to make sure that $x_i$ is the proper value. The axiom for the second selection $E.2$ is obtained by rearranging $x_1$ and $x_2$ in the precondition.

We have used $\hookrightarrow$ in this axiom, where Reynolds used $\mapsto$. In the intuitionistic semantics described later the two versions of the axiom are equivalent, but in the classical semantics the version with $\hookrightarrow$ is preferable. If we had used $\mapsto$ instead of $\hookrightarrow$ in the classical case then the heap in the precondition would be forced to be a singleton. This would be sound, but not very useful.

Next,

*Object-component Assignment*

> Suppose that the variables $x_1, ..., x_m$ are not free in $E$ or $E'$. Then
>
> $$\begin{array}{l}\{\exists x_1,...,x_m.\,(E \mapsto E_1, E_2) * P\}\\ E.1 := E'\\ \{\exists x_1,...,x_m.\,(E \mapsto E', E_2) * P\}\end{array}$$

The simplicity of this axiom is remarkable, and is where the effect of $\mapsto$ and $*$ is coming through. The idea is that we can simply slot $E'$ into the heap in the appropriate place. The $E.2$ version has $(E \mapsto E_1, E')$ in the postcondition.

Finally,

*Cons*

> Suppose that the variables $x_1, ..., x_m$ are not free in $E_1, E_2$, that $x$ and $x'$ are distinct from each other and $x_1, ..., x_m$, that $x'$ is not free in $E_1, E_2$ or $P$. Let $X'$ denote the result of substituting $x'$ for $x$ in expression or assertion $X$. Then
>
> $$\begin{array}{l}\{\exists x_1,...,x_m.\,P\}\\ x := \mathtt{cons}(E_1, E_2)\\ \{\exists x', x_1,...,x_m.\,P' * (x \mapsto E_1', E_2')\}\end{array}$$

Here, a new cell is created and a pointer to it is placed into $x$; the newness of this cell is why it can be separated from $P'$ using $*$.

The following proof outline, for a piece of code for inserting a cell in the middle of a linked list, exemplifies the workings of the axioms for pointer swing and heap extension.

$$
\begin{array}{l}
\{(x \mapsto a, z) * (y \mapsto c, w)\} \\
t := \mathtt{cons}(b, y) \\
\{(x \mapsto a, z) * (y \mapsto c, w) * (t \mapsto b, y)\} \\
\{(x \mapsto a, z) * (t \mapsto b, y) * (y \mapsto c, w)\} \\
x.2 := t \\
\{(x \mapsto a, t) * (t \mapsto b, y) * (y \mapsto c, w)\}
\end{array}
$$

PROPOSITION 3. *The Reynolds axioms are true in the classical semantics.*

## 5. COMPLETENESS ISSUES

We begin by discussing the *Component Assignment* axiom. The way the axiom is formulated requires both the precondition and the postcondition to be of a special shape, and this raises the question: can the axiom be applied generally, or does it restrict our reasoning to situations where the assertions are of a specific form?

Before answering this question, we formulate a backwards axiom with the help of the form of update that can be expressed using $*$ and $-\!*$ .

*Backwards Component Assignment*

Suppose that variables $x$ and $y$ are distinct and not free in $E$, $E'$, or $P$. Then

$$
\begin{array}{l}
\{\exists xy.\, (E \mapsto x, y) * ((E \mapsto E', y) -\!* P)\} \\
E.1 := E' \\
\{P\}
\end{array}
$$

The $E.2$ version is similar. The backwards version can obviously be applied generally, since it works for any postcondition.

In a draft version of this paper (dated 10 March, 2000), we made the erroneous claim that the backwards axiom is strictly stronger than *Component Assignment*, because of the latter's seemingly restricted form. However, Reynolds has pointed out that the axioms are of equal strength, if we include the rule of consequence and consider an instance of *Component Assignment* with an occurrence of $-\!*$ to the right of $*$, using again the "update as deletion followed by extension" idea.

$$
\begin{array}{l}
\{\exists xy.\, (E \mapsto x, y) \; * \; ((E \mapsto E', y) -\!* P)\} \\
E.1 := E' \\
\{\exists xy.\, (E \mapsto E', y) \; * \; ((E \mapsto E', y) -\!* P)\} \\
\{\exists xy.\, P\} \\
\{P\}
\end{array}
$$

The second-last step uses the consequence $A * (A -\!* P) \models P$, and the fact that consequence is valid under $\exists$. The $\exists$ can be eliminated in the final step because $x$ and $y$ are not free in $P$. So, although the backwards form of the axiom expresses the weakest precondition directly, the two versions are interderivable.

We next discuss a curious point about the interpretation of triples. We have allowed commands to be applied to states with dangling pointers, which are states that mention locations not in the domain of the current heap. In contrast, in [35] commands are only applied to states in which there are no dangling pointers; dangling pointers arise only during the evaluation of assertions.

The difference between these interpretations of triples is significant in the case of `cons`. For example,

$$\{\mathtt{true}\}x := \mathtt{cons}(1, 2)\{\neg(x = y)\}$$

is true under a no-dangling interpretation of triples, but not under the interpretation we have adopted. The reason is that if there are no dangling pointers then the operational rule for `cons` allocates a location that is not the contents of any stack variable, but in the dangling case a location might be allocated that is already the contents of some stack variable.

This indicates that the *Cons* axiom is not complete under the no-dangling interpretation of triples. (This remark applies equally to the classical semantics and to the intuitionistic semantics presented later.) For, the example triple above is not derivable from the forwards *Cons* axiom, which simply gives us

$$\{\mathtt{true}\}x := \mathtt{cons}(1, 2)\{\mathtt{true} * x \mapsto 1, 2\}$$

the postcondition of which is equivalent to $x \hookrightarrow 1, 2$.

One way to react to this incompleteness is to say that since dangling pointers never arise during program execution (for the programs considered so far), we should interpret the rule of consequence as an implication which holds in states where there is no dangling. That is, rule out dangling pointers at the top level, so to speak, but allow them when delving into subformulae involving $*$ or $-\!*$ . Another reaction, which we follow up on here, is to see dangling pointers as a natural characteristic of languages which allow memory to be manipulated on a low level; we elaborate on this point in the next section.

To describe a backwards axiom for `cons`, suppose we are given an arbitrary postcondition $P$. In the precondition we would like to say that $P$ will be true if we extend the heap with a new location, which is initialized appropriately. We can express this using $\forall$ to quantify over locations, indicating that any one will do, together with $-\!*$ for guaranteeing newness.

*Backwards Cons*

Suppose that $x'$ is not free in $E_1, E_2$ or $P$. Then

$$
\begin{array}{l}
\{\forall x'.\, (x' \mapsto E_1, E_2) -\!* P[x'/x]\} \\
x := \mathtt{cons}(E_1, E_2) \\
\{P\}
\end{array}
$$

In case $x$ is not free in $E_1$ or $E_2$ we can simply quantify over $x$ in the above. For example, $\forall x.\, (x \mapsto 1, 2) -\!* P$ is the precondition for $x := cons(1, 2)$.

If $C$ is a command and $Q$ a formula, then the weakest precondition is defined as follows.

$s, h \in wp(C, Q)$ just when

$C, s, h$ is safe and if $C, s, h \rightsquigarrow^* s', h'$ then $s', h' \models Q$

We are not extending the syntax of formulae here, but are simply defining $wp(C, Q)$ as a set of stack-heap pairs. (With this definition we should perhaps speak of weakest *liberal* preconditions; but partial and total correctness coincide for the basic commands that we are considering.)

In the following result the "backwards axioms" are considered to be those from this section, along with *Simple Assignment* and *Object-component Lookup*.

THEOREM 4. *The weakest precondition for each atomic statement is expressed by the corresponding backwards axiom.*

For a sequence $C$ of assignment statements it follows that $\{P\}C\{Q\}$ is derivable from the basic axioms (in either the Reynolds or backwards forms), *Sequencing*, and *Consequence* exactly when it is true. (Extending this result to loops would get us into the issue of expressiveness [10], which is outside the scope of our concerns here.)

The following notation will be convenient: if $\ell \in dom(h)$ then let $h@\ell$ denote the singleton heap in which $\ell$ is mapped to $h(\ell)$; also, let $h - \ell$ denote the heap like $h$ except that it is undefined on $\ell$. It is evident that $h = (h@\ell) \cdot (h - \ell)$ when $\ell \in dom(h)$.

**Proof**. We only give the proofs for the heap-altering commands $E.i := E'$ and $x := \mathtt{cons}(E_1, E_2)$.

For soundness of *Backwards Component Assignment*, assume that $s, h$ satisfies the precondition. The precondition ensures $[\![E]\!]s = \ell \in dom(h)$ is a defined location, and so the assignment statement does not get stuck. By the semantics of $E.i := E'$ we need to show that $s, h' \models P$, where $h' = [h \mid \ell \mapsto \langle [\![E']\!]s, v_2\rangle]$ and $h(\ell) = \langle v_1, v_2\rangle$. From the assumption and the semantics of $\exists$ we get that

$$s', h \models (E \mapsto x, y) \;*\; ((E \mapsto E', y) -\!\!* \, P)$$

for the extension $s'$ of $s$ which binds $x$ to $v_1$ and $y$ to $v_2$. Then, from the definitions of $*$ and $\mapsto$, we get that

$$\begin{array}{l} s', h@\ell \models (E \mapsto x, y) \\ s', h - \ell \models (E \mapsto E', y) -\!\!* \, P. \end{array}$$

The semantics of $-\!\!*$ then implies that $s', (h - \ell) \cdot [\ell \mapsto \langle [\![E']\!]s, v_2\rangle] \models P$ and, since $h' = (h - \ell) \cdot [\ell \mapsto \langle [\![E']\!]s, v_2\rangle]$, we get $s', h' \models P$. The stack $s'$ can be replaced by $s$, because $x$ and $y$ are not free in $P$, and we are done.

For completeness, assume that $s, h \in wp(E.i := E', P)$. From the safety part of $wp$ we get that $[\![E]\!]s = \ell \in Loc$ for some $\ell \in dom(h)$. Suppose $h(\ell) = \langle v_1, v_2\rangle$. We claim that

$$[s \mid x \mapsto v_1, y \mapsto v_2], h \models (E \mapsto x, y) \;*\; ((E \mapsto E', y) -\!\!* \, P)$$

The singleton heap $h@\ell$ makes the left conjunct true. That $h - \ell$ satisfies the right conjunct follows from the $wp$ assumption, which implies that $P$ is true if we update the original heap $h$ by mapping the first component of $\ell$ to $[\![E']\!]s$. That is, the semantics of $-\!\!*$ and of the instance of $\mapsto$ to its left conspire to ensure that $h - \ell$ satisfies the right conjunct. The clauses for $\exists$ and $*$ imply that $s, h$ satisfies the precondition.

For soundness of *Backwards Cons*, assume that $s, h$ satisfies the precondition. By the operational rule for allocation we need to show $[s \mid x \mapsto \ell], [h \mid \ell \mapsto \langle v_1, v_2\rangle] \models P$ when $\ell \notin dom(h)$, $[\![E_1]\!]s = v_1$, and $[\![E_2]\!]s = v_2$. We know that $[s \mid x' \mapsto \ell], [h \mid \ell \mapsto \langle v_1, v_2\rangle]$ satisfies $P[x'/x]$, from the definitions of $-\!\!*$, $\mapsto$ and $\forall$. The result then follows using standard lemmas about renaming variables and removing from a state those not appearing freely in an expression.

For completeness, assume $s, h \in wp(x := \mathtt{cons}(E_1, E_2), P)$. From the operational rule for $\mathtt{cons}$, we obtain that

$$[s \mid x \mapsto \ell], [h \mid \ell \mapsto \langle [\![E_1]\!]s, [\![E_2]\!]s\rangle] \models P$$

for any location $\ell \notin dom(h)$ (non-determinism of $\rightsquigarrow$ is being used here). That $s, h$ satisfies the precondition then follows immediately from this and the definitions.

**End of Proof**

## 6. DISPOSE

All of the axioms we have considered so far are compatible with the presence of dangling pointers, and dangling pointers play an important role in the interpretations of $*$ and $-\!\!*$. We might as well push this further and consider a command $\mathtt{dispose}(E)$ which deallocates a location (thereby creates a dangling pointer).

The semantics of `dispose` is a slippery subject, and what happens on subsequent attempts to dereference a disposed location tends to be "undefined" by programming language definitions. Operationally, we take the position that `dispose` simply removes a location from the heap.

$$\frac{\ell \in Loc \quad \ell \in dom(h) \quad [\![E]\!]s = \ell}{\mathtt{dispose}(E), s, h \rightsquigarrow s, (h - \ell)}$$

Recall that $h - \ell$ is $h$ with $\ell$ removed.

We do not wish to enter into a controversy over how well this models "undefined". Indeed, there may be no definitive operational semantics of `dispose`, and it is perhaps better treated from an axiomatic perspective.

*Dispose*
Suppose that $a, b$ are not free in $E$. Then,

$$\begin{array}{l} \{P * \;\exists ab.\, (E \mapsto a, b)\} \\ \mathtt{dispose}(E) \\ \{P\} \end{array}$$

This axiom takes the view that you simply shouldn't depend on what contents the disposed location might or might not have in the postcondition.

Reasoning backwards from `true` we can find circumstances under which a program is safe to execute. For a double dispose we obtain `false` as the precondition as expected, indicating that the program is not safe to execute for any start state.

$$\begin{array}{l} \{\mathtt{false}\} \\ \{\mathtt{true} \;*\; \exists ab.\, (x \mapsto a, b) \;*\; \exists cd.\, (x \mapsto c, d)\} \\ dispose(x) \\ \{\mathtt{true} \;*\; \exists ab.\, (x \mapsto a, b)\} \\ dispose(x) \\ \{\mathtt{true}\} \end{array}$$

PROPOSITION 5. *The Dispose axiom expresses the weakest precondition.*

**Proof**. For soundness, assume the precondition holds for $s, h$. The precondition ensures $[\![E]\!]s = \ell \in dom(h)$ is a defined location, so the command does not get stuck. The result of the dispose statement is the pair $s, h - \ell$, and we need to show that $s, h - \ell \models P$. This follows using the definitions of $\exists$, $*$ and $\mapsto$,

For completeness, assume $s, h \in wp(\mathtt{dispose}(E), P)$. From the operational rule and the definition of $wp$, which requires safety, we obtain that $[\![E]\!]s = \ell \in dom(h)$ is a location that points to something, say $\langle v_1, v_2\rangle$, and that $s, h - \ell \models P$. It is clear that

$$[s \mid x \mapsto v_1, y \mapsto v_2], h@\ell \models E \mapsto x, y$$

so, by the semantics of $\exists$ and $*$, and the assumption that $x, y \notin free(P)$, we obtain that $s, h$ satisfies the precondition as required.

**End of Proof**

# 7. A SMALL EXAMPLE

We give a small example: a program for disposing a list. To formulate the precondition, we use an inductive definition of a predicate $\mathtt{rep}\ n\ E$, which says that $E$ represents a list of size $n$.

$$\begin{array}{lcl} \mathtt{rep}\ 0\ E & \stackrel{\Delta}{\Longleftrightarrow} & E = \mathtt{nil}\ \wedge\ \mathtt{emp} \\ \mathtt{rep}\ n+1\ E & \stackrel{\Delta}{\Longleftrightarrow} & \exists xy.\,(E \mapsto x, y)\ *\ \mathtt{rep}\, n\, y. \end{array}$$

Then $E$ points to a non-circular linked list when $\mathtt{rep}\ n\ E$ holds for some $n$, and we define

$$\mathtt{nclist}\, E \ \stackrel{\Delta}{\Longleftrightarrow}\ \exists n.\,\mathtt{rep}\ n\ E.$$

Note that this definition just says that $E$ points to a list, and ignores head links; variations are possible.[2]

The specification for the program says that, if $p$ points to a list to begin with, then the program will (assuming it terminates) delete all the cells, resulting in the empty heap. (The presence of emp in the base case of the inductive definition is necessary for this.)

$$\begin{array}{l} \{\mathtt{nclist}\, p\} \\ \ \mathtt{while}\ p \neq \mathtt{nil}\ \mathtt{do} \\ \qquad q := p;\ p := p.2;\ \mathtt{dispose}(q) \\ \{\mathtt{emp}\} \end{array}$$

Now, we use the usual Hoare partial-correctness rule for *while* loops, where we choose the precondition as the invariant. A proof outline for the body is

$$\begin{array}{l} \{p \neq \mathtt{nil}\ \wedge\ \mathtt{nclist}\, p\} \\ \{\exists p_0.\, \exists x.\,(p \mapsto x, p_0)\ *\ \mathtt{nclist}\, p_0\} \\ \{\exists p_0. \exists x.\,(p \hookrightarrow x, p_0) \wedge \big((\mathtt{nclist}\, p_0) * \exists ab.\,(p \mapsto a, b)\big)\} \\ \ q := p \\ \{\exists p_0. \exists x.\,(p \hookrightarrow x, p_0) \wedge \big((\mathtt{nclist}\, p_0) * \exists ab.\,(q \mapsto a, b)\big)\} \\ \ p := p.2 \\ \{(\mathtt{nclist}\, p)\ *\ \exists ab.\,(q \mapsto a, b)\} \\ \ \mathtt{dispose}(q) \\ \{\mathtt{nclist}\, p\} \end{array}$$

In the second line we have listed an intermediate step used in applying the rule of consequence.

To complete the proof, combining the negation of $p \neq \mathtt{nil}$ with the invariant we obtain

$$p = \mathtt{nil}\ \wedge\ \mathtt{nclist}\, p$$

as a valid postcondition for the whole program. This implies emp by the definition of rep and so, by the rule of consequence, we are done.

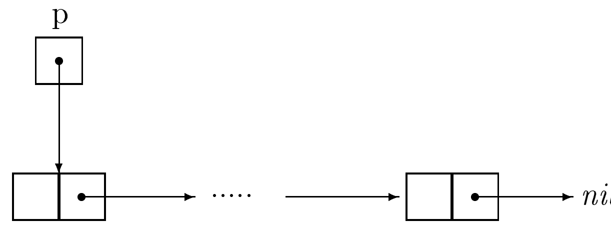
# 8. LOCALITY OF SPECIFICATIONS AND REASONING

Consider again the specification of the program to dispose a list.

[2]We have not included recursive definitions in the formal syntax, but the intent should be clear. In any case, we will be somewhat less formal here, and in particular use a $\exists n$ for quantifying over natural numbers only.

$$\{\mathtt{nclist}\, p\} \cdots \{\mathtt{emp}\}$$

The first thing to notice here is the exact nature of the precondition: if $\mathtt{nclist}\, p$ is true then there can be no cells in the current heap other than those in the list pointed to by $p$. That is, $\mathtt{nclist}\, p$ holds of a structure




but not of a heap with additional nodes not in the list. It is possible for one of the head nodes to contain a pointer, but that pointer must either be to one of the nodes in the list or be dangling.

This exact nature comes about because of the use of emp in the base case of rep, and also because of the exact nature of $\mapsto$. In fact, such an exact specification is necessary, because if there were "junk cells", cells in the heap but not in the list, then we could not conclude emp on termination. Here "junk" is relative: it just means cells that are not relevant to the correct operating of the program, not necessarily garbage cells.

The second thing to note is that these junk cells have been avoided without talking about them explicitly in the definition of $\mathtt{nclist}\, p$. Normally, one would have to include an auxiliary clause which says "for all cells, if that cell is in the heap it is in the list". But we did not need to.

However, there appears to be a problem with the specification: what if we want to run the program when there are extra cells around? The specification appears not to be strong enough. Intuitively, however, we have verified exactly the correct property: the precondition mentions only those cells which are accessed by the program during execution. Why should we have to mention others? This section explains why we don't have to.

The basis for our approach is a local property of specifications, which we state informally as follows.

> *If $\{P\}C\{Q\}$ holds, then execution of $C$ in a state satisfying $P$ can attempt to dereference* only *those heap cells guaranteed to exist by $P$.*

Conventionally, the assumption is that a pre/post specification makes a positive statement about alterations to the store that can be made, but additional changes are allowed: this leads to the need for explicit frame axioms, which say what doesn't change. The formalism here turns the situation around, by restricting the alterations (to the heap) that can be made to be those specifically mandated by the specifications. Explicit provision is then required to sanction changes, instead of to disallow them.

In this section we investigate these ideas by examining a rule, *Frame Axiom Introduction.*

## 8.1 Local/Global Interaction

The discussion above is concerned exclusively with the heap. For all we know, if $\{x \hookrightarrow 1, 2\}C\{x \hookrightarrow 3, 2\}$ holds then $C$ might change a stack variable $z$. For example, $z := 7; x.1 := 3$ satisfies the specification. So, in order to state the rule for frame axiom introduction, we need to keep track of stack variables altered by a program. We do this with a syntactic condition.

> Define *ModifiesOnly*$(C)$ to be the set of (free) variables appearing alone to the left of := in $C$.

The qualification "alone" means, for example, that the set *ModifiesOnly*$(x.i := E)$ is empty: *ModifiesOnly* is concerned with modifications to stack variables only here.

*Frame Axiom Introduction*

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}}\ \ \mathit{ModifiesOnly}(C) \cap \mathit{free}(R) = \emptyset$$

It is important to see that we cannot use $\wedge$ instead of $*$, as the resulting rule is unsound. More positively, using this rule we can perform an inference

$$\frac{\{(x \hookrightarrow 1,2)\}C\{(x \hookrightarrow 3,2)\}}{\{(x \hookrightarrow 1,2)\ *\ (z \hookrightarrow 7,11)\}C\{(x \hookrightarrow 3,2)\ *\ (z \hookrightarrow 7,11)\}}$$

as long as we know that $C$ doesn't modify the stack variable $z$. We use $*$ here to identify a portion of the heap that is not modified.

The soundness of *Frame Axiom Introduction* can be shown for assignment statements, sequencing, looping, and conditionals. A thorough theoretical account of this rule and its consequences will be presented in a future paper [27].

## 8.2 Framing Procedure Specifications

Frame axioms take on greater importance in the presence of procedures, where one wants to be able to specify a procedure without referring to its code [2]. We give a brief discussion of procedures in light of the above.

Let us regard the program for disposing a list as a procedure, parametric in $p$, and where the auxiliary variable $q$ is local. To specify *DisposeList* we should give not only the precondition and postcondition, but also a *ModifiesOnly* clause.

$$\{\texttt{nclist}\, p\}\ \mathit{DisposeList}(p)\ \{\texttt{emp}\}$$
$$\mathit{ModifiesOnly}(\mathit{DisposeList}(p)) \ = \ \{p\}$$

We claim that just using the local specification, which only mentions those heap cells touched by the program, we can infer properties of calls in wider contexts. A good example of this is when we chain two calls to *DisposeList*, to dispose of two different lists. Then, using *Frame Introduction* together with *Sequencing* and *Consequence*, we can infer that the two calls work properly, as long as the input lists don't overlap.

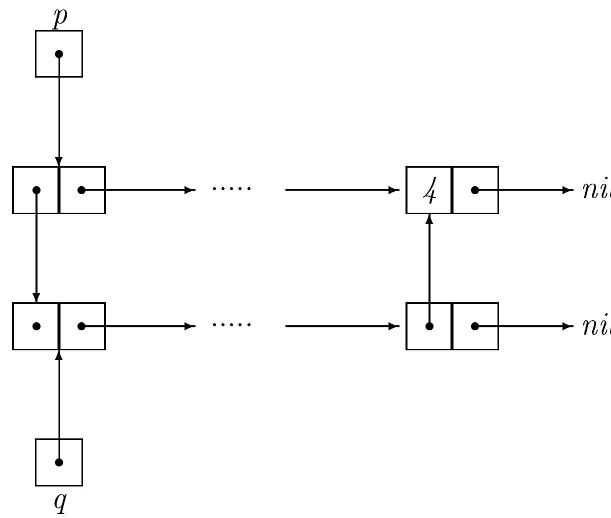
$$\frac{\dfrac{\{\texttt{nclist}\, p\}\mathit{DisposeList}(p)\{\texttt{emp}\}}{\{(\texttt{nclist}\, p) * (\texttt{nclist}\, q)\}\mathit{DisposeList}(p)\{\texttt{emp} * \texttt{nclist}\, q\}}}{\{(\texttt{nclist}\, p) * (\texttt{nclist}\, q)\}\mathit{DisposeList}(p)\{\texttt{nclist}\, q\}}$$

Then, the specification $\{\texttt{nclist}\, q\}\mathit{DisposeList}(q)\{\texttt{emp}\}$ together with the usual Hoare rule for sequencing gives us

$$\{(\texttt{nclist}\, p) * (\texttt{nclist}\, q)\}\mathit{DisposeList}(p); \mathit{DisposeList}(q)\{\texttt{emp}\}$$

as desired. Conventionally, an explicit frame axiom would be needed to sanction a conclusion of this sort, because otherwise we would have no way of knowing that *DisposeList*$(p)$ doesn't alter the list pointed to by $q$. (For instance, if the first call were to incorrectly dispose of one of the nodes in $q$'s list, then we would get a safety violation in the second.) The same principle works when we chain together calls to different procedures, such as procedures for inserting into, deleting from, or copying lists.

It is important to realize that the use of $*$ in the conjunction $(\texttt{nclist}\, p) * (\texttt{nclist}\, q)$ is not simply a reachability condition, which states, say, that the cells reachable from $p$ and $q$ are disjoint. For instance, $(\texttt{nclist}\, p) * (\texttt{nclist}\, q)$ holds of




Here, it is certainly possible to reach one list from the other, by following head links, but this does not cause a runtime error in *DisposeList*$(p)$; *DisposeList*$(q)$.

# 9. THE INTUITIONISTIC SEMANTICS

In this section we consider an intuitionistic semantics. All assertions will satisfy the

> *Monotonicity Condition*: If $s, h \models P$ and $h \sqsubseteq h'$ then $s, h' \models P$,

where $h \sqsubseteq h'$ indicates that the graph of $h$ is a subset of the graph of $h'$. Formally, the intuitionistic language is obtained by omitting emp, adding clauses for intuitionistic connectives that cannot be defined away

$$s, h \models P \wedge Q \quad \text{iff} \quad s, h \models P \text{ and } s, h \models Q$$

$$s, h \models P \vee Q \quad \text{iff} \quad s, h \models P \text{ or } s, h \models Q$$

$$s, h \models \forall x.P \quad \text{iff} \quad \forall v \in Val.\, [s \mid x \mapsto v], h \models P$$

and making two redefinitions:

$$s, h \models E \mapsto E_1, E_2 \quad \text{iff} \quad [\![E]\!]s \in dom(h) \text{ and } h([\![E]\!]s) = \langle [\![E_1]\!]s, [\![E_2]\!]s \rangle$$

$$s, h \models P \Rightarrow Q \quad \text{iff} \quad \forall h' \sqsupseteq h.\ \text{if } s, h' \models P \text{ then } s, h' \models Q.$$

The other semantic clauses are as in Section 3.1.[3] To see why the law of the excluded middle fails in this model, consider

[3] Intuitionistic $\forall$ usually quantifies over "future" possible worlds, but in a fixed-domain semantics (where the same individuals exist at each world) the pointwise definition remains adequate. Also, in the clause for $*$ one might have expected to see a condition $h_0 \cdot h_1 \sqsubseteq h$ instead of asking

$(x \mapsto 2, 2) \vee \neg(x \mapsto 2, 2)$, where $\neg P = P \Rightarrow \mathtt{false}$. If $s$ is a stack with $sx = \ell$ and $[]$ is the empty heap, then $s, [] \not\models x \mapsto 2, 2$. But we also have $s, [] \not\models \neg(x \mapsto 2, 2)$, since there is an extension $[\ell \mapsto 2, 2]$ of $[]$ where $s, [\ell \mapsto 2, 2] \models x \mapsto 2, 2$. So $s, [] \not\models (x \mapsto 2, 2) \vee \neg(x \mapsto 2, 2)$.

The semantic consequence relation and interpretation of triples are defined as before. Some of the basic properties of the logic are altered by the intuitionistic semantics.

PROPOSITION 6. *Propositions 1 and 2 go through for the intuitionistic semantic of this section, with the following changes:*

- *The semantics validates intuitionistic rather than classical logic, so that excluded middle fails generally;*
- $\mathtt{true}$ *is the unit of* $*$*;*
- *Weakening for* $*$ *holds:* $A * B \models A$*;*
- *Excluded middle holds for pure assertions;*
- $P * Q$ *and* $P \wedge Q$ *are equivalent if* $P$ *is pure, even when* $Q$ *is not.*

A useful observation is that the classical and intuitionistic interpretations behave similarly when $\mapsto$ appears as an immediate constituent of $*$. To formulate this, recall that if $\ell \in dom(h)$ then we use $h@\ell$ to denote the singleton heap in which $\ell$ is mapped to $h(\ell)$ .

LEMMA 7. [Exactness Lemma]

$$s, h \models (E \mapsto E_1, E_2) * P$$

*in the intuitionistic semantics iff there is some* $\ell \in dom(h)$ *such that*

$$s, h@\ell \models (E \mapsto E_1, E_2), \text{ and}$$
$$s, h - \ell \models P.$$

Thus, even though the intuitionistic semantics uses an inexact interpretation of $\mapsto$, we can get away with the exact interpretation when looking at one occurrence of $\mapsto$ in an argument to $*$. This explains why it is possible to use either of the intuitionistic or classical semantics for the same program-proving axioms.

THEOREM 8. *The weakest precondition results hold for the intuitionistic semantics.*

Of course, this result has a different import than the previous ones, because it refers exclusively to intuitionistic propositions, that are invariant under heap extension. The only alterations to the previous proofs involve an appeal to the Exactness Lemma in several places, and appeals to monotonicity in some situations where it was not needed in the argument for classical semantics (the completeness parts of *Backwards Cons* and *Backwards Object-component Assignment*).

We can compare the two semantics by noting that we can translate from the intuitionistic language into the classical one using a modal translation. We do not actually need to extend the classical language with an explicit modality to do this, because we can already express the necessity modality for heap extension. That is,

for equality: but the monotonicity condition, together with the fact (true of the particular model here) that $h_0 \cdot h_1 \sqsubseteq h$ when $h$ bounds each, implies that the two definitions are equivalent.

$$s, h \models \mathtt{true} \mathrel{-\!\!*} P \quad \text{iff } \forall h' \sqsupseteq h.\, s, h' \models P$$

holds in the classical semantics.

*The Modal Translation.* The translation $(\cdot)^\circ$ sends

$$\begin{array}{lll} E \mapsto E_1, E_2 & \text{to} & E \hookrightarrow E_1, E_2 \\ P \Rightarrow Q & \text{to} & \mathtt{true} \mathrel{-\!\!*} (P^\circ \Rightarrow Q^\circ) \end{array}$$

and everything else (inductively) to itself.[4]

PROPOSITION 9. $s, h \models P$ *in the intuitionistic semantics iff* $s, h \models P^\circ$ *in the classical semantics.*

So, the classical semantics is, in this sense, the more expressive of the two. More to the point, the intuitionistic semantics has an additional condition, monotonicity, and we should ask whether there are any properties of interest that do not satisfy it.

It turns out that many natural pre- and postconditions for pointer algorithms do satisfy monotonicity. Often, one makes a positive statement to the effect that a collection of cells in the heap represents some abstract data structure, and these cells continue to represent the structure when more cells are added. Still, there are some natural properties that do not satisfy monotonicity. An example is given by the `rep` and `nclist` predicates from Section 7. There, the use of `emp` in the base case of `rep` has the effect of limiting a heap satisfying `nclist` $E$ to exactly those cells reachable, by following tail links, from $E$; this was essential for showing that all of the cells were de-allocated. Other typical properties of this sort are that there is a unique pointer (in the heap) to cons cell $x$, or that the heap has exactly 4 cons cells. Generally, non-monotone properties are useful in situations where one is concerned with close control over memory usage, such as when ensuring that there are no space leaks.

We conclude this section by contrasting the two semantics using a subtle example from [35], the following instance of the *Cons* axiom:

$$\begin{array}{l} \{\neg\exists x.\, x \mapsto 1, 2\} \\ \quad y := \mathtt{cons}(1, 2) \\ \{(\neg\exists x.\, x \mapsto 1, 2) * (y \mapsto 1, 2)\}. \end{array}$$

At first sight it looks as if the triple should be false, because the postcondition appears to be inconsistent. The intuitionistic semantics saves the situation by making the precondition inconsistent as well. To see why, consider any $s, h$. We can extend $h$ with a location $\ell \notin dom(h)$, and obtain $[h \mid \ell \mapsto \langle 1, 2\rangle]$. Since this heap extends $h$, the intuitionistic negation quantifies over it. And in this extended heap, $\exists x.\, x \mapsto 1, 2$ is true.

The same triple holds as well in the classical semantics, but the reason now is not that the precondition is false, but rather that the postcondition is not inconsistent. That is, $\neg\exists x.\, x \mapsto 1, 2$ may be true of a small world but false at a bigger one, and the $*$ in the postcondition lets us pick this smaller world out without incurring falsity at the big world. For example, in the singleton heap where the a location denoted by $x$ has contents $\langle 1, 2\rangle$ the empty heap can be

[4]This translation uses the induced modality less often than one might have expected. Normally, one would use the modality with $\forall$ as well, and a backwards modality in the case of $*$. It is specific properties of the model (constant domain, bounding properties of $\cdot$) that justify the simpler translation.

selected for $\neg\exists x. x \mapsto 1, 2$ and the singleton heap itself for $x \mapsto 1, 2$.

The absence of Weakening in the classical semantics is significant here. For, if we had

$$(\neg\exists x. x \mapsto 1,2) * (y \mapsto 1,2) \models \neg\exists x. x \mapsto 1,2, \text{ and}$$
$$(\neg\exists x. x \mapsto 1,2) * (y \mapsto 1,2) \models y \mapsto 1,2$$

then we could obtain

$$(\neg\exists x. x \mapsto 1,2) * (y \mapsto 1,2) \models (\neg\exists x. x \mapsto 1,2) \wedge (y \mapsto 1,2),$$

the consequent of which is contradictory.

## 10. SUMMARY AND RELATED WORK

The most relevant related work is contained in the two main precursors, the papers of Burstall and Reynolds [5, 35]. To summarize our additions to [35], we have: (i) provided a classical model, and investigated the relation between classical and intuitionistic variants; (ii) added BI's spatial implication $-\!\!*$ to the assertion language, and used it to express weakest preconditions; (iii) given a treatment of `dispose`; and (iv) further explicated the form of local reasoning made possible by the spatial approach to pointer logic.

There have been a number of papers on program-proving for pointers ([16, 30, 23, 17, 22, 11, 3, 6] is a partial list). What sets the approach of Reynolds and Burstall apart is its local treatment of assignment. In other approaches assignment in the presence of aliasing tends to be dealt with using global store parameters, or several global parameters, or with axioms that involve major surgery on formulae. In contrast, in $\{P * (x \mapsto a, b)\}x.1 := z\{P * (x \mapsto z, b)\}$ the operationally local nature of assignment is mirrored beautifully in the logic.

There has been growing interest in using program logic for pointers in static analysis and related problems, and some excellent results have been obtained [18, 24, 37, 40]. The work here appears to be largely complementary. Indeed, although the devil is in the detail, it would be conceivable to combine one of these assertion languages with a substructural logic, in the style of BI. The main question is whether such a combination would give rise to local reasoning or specifications, in a way that does not interfere with the already successful properties of these languages.

We described the local character of specifications in the logic, and began an exploration of its consequences by consideration of the rule for introducing frame axioms. There are many vaguely related ideas in dozens of papers in the AI, modal and temporal logic of processes, and program specification literatures; we cannot do justice to these literatures in this short space (we mention only one from each strand: [33, 20, 1]). The main point, however, is the implicit and succinct way that behind-the-scenes dependencies, which arise from pointers that are not directly named by program variables, are dealt with using $*$. We are not aware of a previous approach that deals with these dependencies in a comparable manner. That being said, there is much more to be learnt about local reasoning; some further developments will be presented in a followup paper [27]. In addition, it would be interesting to attempt to apply these ideas in related situations where aliasing is prevalent, such as $\pi$-calculus or object calculi.

In the linear logic literature there have been numerous hints, suggesting that substructural logic can be used to specify and reason about actions locally (e.g. [13, 21]). While this proposal was tantalyzing, it has not subsequently been developed very far, certainly not as far as a program logic for pointers. (Encodings of the semantics of imperative languages, e.g. [9], are important and useful, but fall well short of program logic.) The results of this paper might be interpreted as offering fresh justification for those early hints, and in the demanding territory of pointers, albeit for a logic that is different from linear logic in key respects. A feature of BI is that it offers a simple-minded treatment of additive connectives (based on classical or intuitionistic logic) alongside substructural ones; there is no "!", and no need to stay within a constructive setup. This comparative simplicity, as illustrated by the pointer model, is a key to applications.

There are two other closely related pieces of work to report on. The first is work of Cardelli and Gordon on Ambient Logic [8], a logic for mobile ambients. Their logic can be seen as an extension of Boolean BI; on the common connectives, the semantic models of Ambient Logic that have been presented are instances of the possible worlds semantics of BI first presented in [25] and further developed in [26, 32]. Ambient logic also has a connective, the "ambient match", which interacts with $*$ in a way that leads to pleasantly compact and intuitive specifications of certain properties of mobile processes.

In an interesting further development, Cardelli and Ghelli have proposed a labelled tree model as a basis for a query language for semi-structured data [7]. The tree model is similar to the pointer model of BI, but for two main differences: the model here allows for circular structures as well as trees; and, the combining operation here is partial, where in the labelled tree model it is total. Partiality enables us to ensure that subheaps are disjoint, and this is essential for the soundness of the Hoare triple axioms. We speculate that the ideas in this paper, especially those involving the interaction between $*$ and $-\!\!*$, might be adapted to account for update or reconfiguration of semi-structured data.

The second closely related work is that of Smith, Walker, and Morrisett on Alias types [38, 39]. Alias types use type-theoretic cousins of the conjunction $*$ and points-to relation $\mapsto$ to state properties of data structures. The resulting typing rule for component assignment is very close to (a CPS version of) Reynolds's axiom, and their treatment of memory disposal is very near to that here. Of course, the benefit of a type system is that it is static, while conversely logic is more expressive. In any case, the remarkable convergence of ideas in spatial pointer logic and in Alias types might perhaps be taken as a positive indication, of the naturalness of the approach.


### Acknowledgements

We are grateful to David Pym, Uday Reddy and John Reynolds for advice and comments that helped to improve the material in this paper. This research was supported by a grant from the EPSRC.

## 11. REFERENCES


[1] Alur, R., and Grosu, R. Modular refinement of hierarchic reactive machines. In POPL [31].

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

[3] Bornat, R. Proving pointer programs in Hoare logic. In *Fifth Internationsl Conference on Mathematics of Program Construction*, LNCS 1837, Ponte de Lima, Portugal, 2000.

[4] Brookes, S., Main, M., Melton, A., and Mislove, M., Eds. *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference* (Tulane University, New Orleans, Louisiana, March 29–April 1 1995), vol. 1 of *Electronic Notes in Theoretical Computer Science*, Elsevier Science.

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

[6] Calcagno, C., Ishtiaq, S., and O'Hearn, P. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. Proceedings of the 2nd international ACM SIGPLAN conference on on Principles and practice of declarative programming, 2000.

[7] Cardelli, L., and Ghelli, G. A query language for semistructured data based on the ambient logic. Manuscript, 4 April 2000.

[8] Cardelli, L., and Gordon, A. D. Anytime, anywhere. modal logics for mobile ambients. In POPL [31].

[9] Cervesato, I., and Pfenning, F. A linear logical framework. In *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science — LICS'96* (27–30 July 1996), IEEE Computer Society Press, pp. 264–275.

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

[11] de Boer, F. A WP calculus for OO. In *Proceedings of FOSSACS'99* (1999).

[12] Girard, J.-Y. Linear logic. *Theoretical Computer Science* (1987), 1–102.

[13] Girard, J.-Y. Towards a geometry of interaction. In *Categories in Computer Science and Logic* (1989), American Mathematical Society, pp. 69–108. Contemporary Mathematics Volume 92.

[14] Guttag, J., Horning, J., and Wing, J. Larch in five easy pieces. TR 5, DEC Systems Research Center, 1985.

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

[16] Hoare, C. A. R., and Wirth, N. An axiomatic definition of the programming language Pascal. *Acta Informatica 2* (1973), 335–355.

[17] Honsell, F., Mason, I. A., Smith, S., and Talcott, C. A variable typed logic of effects. *Information and Computation 119*, 1 (may 1995), 55–90.

[18] Jenson, J., Jorgensen, M., Klarkund, N., and Schwartzback, M. Automatic verification of pointer programs using monadic second-order logic. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation* (1997), pp. 225–236. SIGPLAN Notices 32(5).

[19] Kripke, S. A. Semantical analysis of intuitionistic logic I. In *Formal Systems and Recursive Functions*, J. N. Crossley and M. A. E. Dummett, Eds. North-Holland, Amsterdam, 1965, pp. 92–130.

[20] Leino, K. *Toward Reliable Modular Programs.* Ph.D. thesis, California Institute of Technology, Pasadena, California, 1995.

[21] Miller, D. Observations about using logic as a specification language. In *GULP-PRODE'95 – Joint Conference on Declarative Programming* (Marina de Vietri, Salerno, Italy, September 1995).

[22] Moller, B. Calculating with pointer structures. In *Proceedings of Mathematics for Software Construction,* (1997), Chapman and Hall, pp. 24–48.

[23] Morris, J. A general axiom of assignment. Assignment and linked data structure. A proof of the Schorr-Waite algorithm. In *Theoretical Foundations of Programming Methodology* (1982), M. Broy and G. Schmidt, Eds., Reidel, pp. 25–51.

[24] Necula, G. Proof-carrying code. In *In Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)* (1997).

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

[26] O'Hearn, P., Pym, D., and Yang, H. Possible worlds and resources: The semantics of BI. Submitted, October 2000.

[27] O'Hearn, P., and Yang, H. Local reasoning about pointer programs using bunched implications. In Preparation, 2000.

[28] O'Hearn, P. W., Power, A. J., Takeyama, M., and Tennent, R. D. Syntactic control of interference revisited. *Theoretical Computer Science 228*, 1-2 (October 1999), 211–252. Preliminary version in [4] and in [29], vol 2.

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

[30] Oppen, D. C., and Cook, S. A. Proving assertions about programs that manipulate data structures. In *Conference Record of Seventh Annual ACM Symposium on Theory of Computation* (Albuquerque, New Mexico, 5–7 May 1975), pp. 107–116.

[31] *Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2000), ACM, New York.

[32] Pym, D. The semantics and proof theory of the logic of bunched implications. Monograph in Preparation, 2000. See http://www.dcs.qmw.ac.uk/ ˜ pym.

[33] Reiter, R. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.

[34] Reynolds, J. C. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages* (Tucson, Arizona, January 1978), ACM, New York, pp. 39–46. Also in [29], vol 1.

[35] Reynolds, J. C. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*, Palgrove, 2000.

[36] Reynolds, J. C. 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.

[37] Sagiv, M., Reps, T., and Wilhelm, R. Parametric shape analysis via 3valued logic. In *POPL'99.*

[38] Smith, F., Walker, D., and Morrisett, G. Alias types. Proceedings of ESOP'99.

[39] Walker, D., and Morrisett, G. Alias types for recursive data structures. Manuscript, April 2000.

[40] Xu, Z., Miller, B., and Reps, T. Safety checking of machine code. In *PLDI'00.*