This is a revised draft of a paper to appear in "Millennial Perspectives in Computer Science", the proceedings of the Oxford–Microsoft Symposium in Honour of Sir Tony Hoare, (held September 13–15, 1999), to be published by Palgrave. It supercedes the version (dated August 12, 1999) that was distributed at the meeting, which contained a serious error.

Intuitionistic Reasoning about Shared Mutable Data Structure*

John C. Reynolds
Department of Computer Science
Carnegie Mellon University

July 28, 2000

Abstract

Drawing upon early work by Burstall, we extend Hoare's approach to proving the correctness of imperative programs, to deal with programs that perform destructive updates to data structures containing more than one pointer to the same location. The key concept is an "independent conjunction" $P \;\&\; Q$ that holds only when P and Q are both true and depend upon distinct areas of storage. To make this concept precise we use an intuitionistic logic of assertions, with a Kripke semantics whose possible worlds are heaps (mapping locations into tuples of values).

The dichotomy between functional and imperative programming has obscured a variety of programming techniques that fit comfortably in neither approach. In fact, between these two paradigms there is a no-man's land inhabited by many useful and intuitively straightforward programs that have been poorly served by both type systems and program-proving methodologies.

*This research was sponsored by National Science Foundation Grant CCR-9804014.

Particularly important are programs where the data structure may contain more than one pointer to the same location, and the program destructively updates the contents of such locations. This class of programs includes many interesting and practically important algorithms; it is far more than a hacker's jungle. Yet there has been surprisingly little research on reasoning about such programs.

In 1972, Burstall [1] gave correctness proofs for imperative programs that alter data structures, by using a novel kind of assertion that he called a "distinct nonrepeating tree system"; this approach was extended by Kowaltowski [2]. In 1975, Cook and Oppen [3, 4] devised an more general approach by extending Hoare logic with extremely complicated inference rules. Then, in 1981, J. M. Morris [5] extended weakest-precondition logic by generalizing the notion of substitution.

In the late 80's Mason and Talcott [6, 7, 8] investigated reasoning about program equivalence for LISP-like functional languages where expression evaluation can alter data structures as a side effect; more recently they and others [9, 10] have extended this approach to a logic for reasoning about programs.

Also recently, Pitts and Stark [11] have studied operational reasoning about an ML-like language with data-altering expressions. (In this work, however, only simple values can be stored at locations; not structured values that themselves contain locations.) This research builds upon earlier work by Stark [12, 13, 14] on languages that create local names.

The present paper builds upon Burstall's ideas, which fit nicely into the framework for reasoning about imperative programs that was devised about the same time by Hoare [15, 16], as well as earlier work by Floyd [17] and Naur [18]. Burstall's "distinct nonrepeating tree system" was a sequence of assertions, written $P_1 \;\&\; \cdots \;\&\; P_n$ in the notation of this paper, where each P_i described a distinct region of storage, so that an assignment to a single location could change only one of the P_i. I believe that this idea of organizing assertions to localize the effect of a mutation may be the key to scalability in reasoning about shared mutable data structure.

The goal of this paper is to overcome two limitations of Burstall's work. In his formalism, each P_i, which he called a "triple", described a fragment of data structure with no internal sharing. (Technically, the triples were morphisms in categories called "free theories" by Lawvere [19].) Sharing only occurred among pointers from variables into fragments, or from one fragment to another, so that a particular assertion could describe only structures with a fixed finite bound on the number of shared locations.

A subtler limitation was the specific notion of composition of triples. Roughly speaking, one composed P with Q by identifying the pointers coming out of P with those going into Q.

In this paper we will use the doubly-linked list as a simple example that violates both limitations. In this structure, every location is shared, so that a description of a fragment representing an arbitrary sequence must permit unbounded internal sharing. Moreover, the natural way of composing fragments P and Q is to identify both a pointer coming out of P with one going into Q, *and* a pointer coming out of Q with one going into P.

The preliminary version of this paper was flawed by a serious error: The inference rule for the **cons** operation was unsound. In the present version (as discussed at the end of Section 3), we have repaired this flaw, and substantially simplified our development, by interpreting assertions intuitionistically, using a Kripke semantics [20] with heaps as possible worlds. A similar intuitionistic semantics has been discovered independently by Ishtiaq and O'Hearn [21], using the logic of bunched implications [22].

1 Syntax

The programming language we will use is the simple imperative language originally axiomatized by Hoare, with additional commands for the manipulation of list structures. These structures will be similar to those of LISP, restricted by the elimination of property lists for atoms, and extended by permitting any positive number of values to be "cons-ed" together. Specifically, the LISP constructor **cons** will be generalized to $\mathbf{cons}_1$, $\mathbf{cons}_2$, $\mathbf{cons}_3$, etc., and the selectors $\mathbf{car}(E)$ and $\mathbf{cdr}(E)$ to $E.1$, $E.2$, etc.

In contrast to LISP, however, these constructors and selectors will be permitted only in commands, not in expressions. The reason for this restriction is that the power of the kind of proof system advocated by Hoare (or Floyd or Dijkstra) depends on the ability to use any expression of the programming language in an assertion. In particular, substituting any expressions for the variables of a tautology should give a valid assertion.

Constructors cannot be expressions since they have side effects. For instance, if we could substitute $\mathbf{cons}_2(1, 2)$ for x in the tautology $\mathsf{x} = \mathsf{x}$, we would obtain $\mathbf{cons}_2(1, 2) = \mathbf{cons}_2(1, 2)$, which must not hold if we are going to distinguish different locations with the same contents.

Selectors cannot be expressions because of their interaction with the "independent conjunction" operator &. For instance, if we could substitute $\mathsf{z}.2$ for both x and y in the tautology $\mathsf{x} = \mathsf{x}\ \&\ \mathsf{y} = \mathsf{y}$, we would obtain $\mathsf{z}.2 = \mathsf{z}.2\ \&\ \mathsf{z}.2 = \mathsf{z}.2$, which is false since the two operands of & do not depend upon distinct regions of storage.

Instead of permitting constructors and selectors in expressions, we introduce three novel forms of commands, which use constructors to create new list structures, and selectors to evaluate and mutate such structures:

$$x := \mathbf{cons}_n(E_1, \ldots, E_n)$$
$$x := E.i$$
$$E.i := E'$$

(where the various E's denote expressions). We have used the assignment symbol := to give these commands a familiar appearance that should make their informal meaning obvious. Formally, however, they are not assignment commands; in particular they will not obey Hoare's axiom of assignment.

As in Hoare's work, assertions include boolean expressions, enriched with quantifiers. In addition, we introduce the form

$$E \to E_1, \ldots, E_n,$$

which holds if the value of E is a location at which is stored an n-tuple containing the values of E_1, ... E_n. Finally, we add the independent conjunction operator, so that an assertion can have the form

$$P\ \&\ P'$$

(where P and P' denote assertions). Roughly speaking, this form is true when P and P' are both true and depend upon distinct regions of storage.

We will define two forms of specification: the original partial-correctness triple introduced by Hoare, which we will write $\{P\}\ C\ \{Q\}$, and an analogous total-correctness specification, which we will write $[\,P\,]\ C\ [\,Q\,]$.

2 Semantics

To make the meaning of our language and its specifications precise, we define a *value* to be an integer, an *atom*, or a *location*, where integers, atoms, and locations comprise disjoint, countably infinite sets, and **nil** is a particular atom:

$$\mathrm{Values} = \mathrm{Integers} \cup \mathrm{Atoms} \cup \mathrm{Locations}$$

$$\mathbf{nil} \in \mathrm{Atoms}.$$

To formalize mutable list structures, we must complicate the usual notion of the state of a computation, which now consists of two parts: the *store*, which maps some finite set of variables into values, and the *heap*, which maps some finite set of locations into nonempty tuples of values. Thus

$$\begin{aligned}
\mathrm{Stores}_V &= (V \to \mathrm{Values}) \qquad \text{where } V \text{ is a finite set of variables} \\
\mathrm{Heaps}_L &= (L \to \mathrm{Values}^+) \qquad \text{where } L \text{ is a finite set of locations} \\
\mathrm{Heaps} &= \textstyle\bigcup_{L \overset{\text{fin}}{\subseteq} \mathrm{Locations}} \mathrm{Heaps}_L \\
\mathrm{States}_V &= \mathrm{Stores}_V \times \mathrm{Heaps}.
\end{aligned}$$

A state $\langle \eta, \sigma \rangle$ is said to be *complete* if the domain of its heap component includes all locations that occur anywhere within the state, i.e., if

$$\forall x \in \mathrm{dom}\,\eta.\ \eta\, x \in \mathrm{Integers} \cup \mathrm{Atoms} \cup \mathrm{dom}\,\sigma$$

$$\forall \ell \in \mathrm{dom}\,\sigma.\ \sigma\, \ell \in (\mathrm{Integers} \cup \mathrm{Atoms} \cup \mathrm{dom}\,\sigma)^+.$$

We define the set

$$\mathrm{CmplStates}_V = \{\, \langle \eta, \sigma \rangle \mid \langle \eta, \sigma \rangle \in \mathrm{States}_V \text{ and } \langle \eta, \sigma \rangle \text{ is complete} \,\}.$$

For each of the five classes of phrases used in programs or their specifications, there is a semantic function giving a different kind of meaning. Each of these semantic functions is indexed by a finite set of variables that must include the free variables of the argument of the semantic function. (We write OrdExp_V for the set of ordinary expressions whose free variables belong to V, and similarly for the other phrase classes.)

- Ordinary Expressions

$$[\![-]\!]^{\text{ordexp}}_V \in \text{OrdExp}_V \to \text{Stores}_V \to \text{Values},$$

- Boolean Expressions

$$[\![-]\!]^{\text{boolexp}}_V \in \text{BoolExp}_V \to \text{Stores}_V \to \{\mathbf{true}, \mathbf{false}\},$$

- Commands

$$[\![-]\!]^{\text{comm}}_V \in \text{Comm}_V \to \text{CmplStates}_V \to (\text{CmplStates}_V \cup \{\perp\}),$$

- Assertions

$$[\![-]\!]^{\text{assert}}_V \in \text{Assert}_V \to \text{Stores}_V \to \text{Heaps} \to \{\mathbf{true}, \mathbf{false}\},$$

- Specifications

$$[\![-]\!]^{\text{spec}}_V \in \text{Spec}_V \to \{\mathbf{true}, \mathbf{false}\}.$$

Notice that ordinary and boolean expressions do not depend upon the heap. (This reflects our decision not to permit constructors and selectors in expressions.) Assertions that do not depend upon the heap are called *pure*, as are commands that neither depend upon nor change the heap. (As usual in denotational semantics, $\perp$ is used to denote the nonterminating execution of a command.)

The meaning of expressions, commands, and specifications is standard. Note, however, that the implicit quantification of specifications extends over both the store and heap components of complete states:

$$\begin{aligned}[\![\{P\}\ C\ \{P'\}]\!]^{\text{spec}}_V = \forall \langle \eta, \sigma \rangle \in \text{CmplStates}_V.\ [\![P]\!]\eta\sigma \text{ and } [\![C]\!]\langle \eta, \sigma \rangle \neq \perp \\ \text{implies } [\![P']\!]\eta'\sigma' \text{ where } \langle \eta', \sigma' \rangle = [\![C]\!]\langle \eta, \sigma \rangle\end{aligned}$$

$$\begin{aligned}[\![[\ P\ ]\ C\ [\ P'\ ]]\!]^{\text{spec}}_V = \forall \langle \eta, \sigma \rangle \in \text{CmplStates}_V.\ [\![P]\!]\eta\sigma \\ \text{implies } [\![C]\!]\langle \eta, \sigma \rangle \neq \perp \text{ and } [\![P']\!]\eta'\sigma' \text{ where } \langle \eta', \sigma' \rangle = [\![C]\!]\langle \eta, \sigma \rangle.\end{aligned}$$

What is not standard is the meaning of assertions, which is defined for incomplete as well as complete states, by a Kripke semantics [20] in which the possible worlds are heaps, ordered by extension. Thus the logic of assertions is intuitionistic rather than classical.

When a boolean expression is used as an assertion, it is pure, i.e., its meaning is independent of the heap:

$$[\![B]\!]_V^{\text{assert}}\eta\sigma \text{ iff } [\![B]\!]_V^{\text{boolexp}}\eta.$$

The simplest impure assertion is $E \rightarrow E_1, \ldots, E_n$, which describes the value of the heap for a single location:

$$\begin{array}{l}[\![E \rightarrow E_1, \ldots, E_n]\!]_V^{\text{assert}}\eta\sigma \text{ iff} \\ \quad [\![E]\!]_V^{\text{ordexp}}\eta \in \operatorname{dom}\sigma \text{ and } \sigma([\![E]\!]_V^{\text{ordexp}}\eta) = \langle [\![E_1]\!]_V^{\text{ordexp}}\eta, \ldots, [\![E_n]\!]_V^{\text{ordexp}}\eta\rangle.\end{array}$$

The operations of conjunction, disjunction, and quantification are defined conventionally, with the heap being treated pointwise:

$$\begin{aligned}
[\![P_1 \wedge P_2]\!]_V^{\text{assert}}\eta\sigma &\text{ iff } [\![P_1]\!]_V^{\text{assert}}\eta\sigma \text{ and } [\![P_2]\!]_V^{\text{assert}}\eta\sigma \\
[\![P_1 \vee P_2]\!]_V^{\text{assert}}\eta\sigma &\text{ iff } [\![P_1]\!]_V^{\text{assert}}\eta\sigma \text{ or } [\![P_2]\!]_V^{\text{assert}}\eta\sigma \\
[\![\forall x.\ P]\!]_V^{\text{assert}}\eta\sigma &\text{ iff } \forall v \in \text{Values}.\ [\![P]\!]_V^{\text{assert}}[\,\eta \mid x\colon v\,]\sigma \\
[\![\exists x.\ P]\!]_V^{\text{assert}}\eta\sigma &\text{ iff } \exists v \in \text{Values}.\ [\![P]\!]_V^{\text{assert}}[\,\eta \mid x\colon v\,]\sigma.
\end{aligned}$$

Here $[\,\eta \mid x\colon v\,]$ denotes the store, with domain $\operatorname{dom}\eta \cup \{x\}$, that maps x into v and maps all other variables x' into $\eta\, x'$. (Analogous quantifiers that range over integers, atoms, or locations are left to the reader.)

On the other hand, the definitions of implication and negation involve an implicit universal quantification over extensions of the heap:

$$\begin{aligned}
[\![P_1 \Rightarrow P_2]\!]_V^{\text{assert}}\eta\sigma &\text{ iff } \forall\sigma' \supseteq \sigma.\ [\![P_1]\!]_V^{\text{assert}}\eta\sigma' \text{ implies } [\![P_2]\!]_V^{\text{assert}}\eta\sigma' \\
[\![\neg\, P]\!]_V^{\text{assert}}\eta\sigma &\text{ iff } \forall\sigma' \supseteq \sigma.\ \text{not } [\![P]\!]_V^{\text{assert}}\eta\sigma'.
\end{aligned}$$

Finally, we must define the independent conjunction operation &. The idea that P_1 and P_2 must depend upon distinct regions of the heap is captured by requiring them to hold for restrictions of the heap with disjoint domains:

$$\begin{array}{l}[\![P_1 \,\&\, P_2]\!]_V^{\text{assert}}\eta\sigma \text{ iff } \exists\sigma_1, \sigma_2. \\ \qquad \sigma_1 \subseteq \sigma \text{ and } \sigma_2 \subseteq \sigma \text{ and } \operatorname{dom}\sigma_1 \cap \operatorname{dom}\sigma_2 \text{ empty} \\ \qquad\qquad \text{and } [\![P_1]\!]_V^{\text{assert}}\eta\sigma_1 \text{ and } [\![P_2]\!]_V^{\text{assert}}\eta\sigma_2.\end{array}$$

We are able to define independent conjunction so simply because we have given meaning to assertions in all states, not just complete states, and because this meaning obeys a monotonicity law that is characteristic of Kripke semantics:

$$\text{If } [\![P]\!]_V^{\text{assert}}\eta\sigma \text{ and } \sigma \subseteq \sigma', \text{ then } [\![P]\!]_V^{\text{assert}}\eta\sigma'.$$

To obtain this property, however, we have sacrificed the law of the excluded middle. For instance, if the store η maps the variable x into a location that does not belong to the domain of the heap σ, then

$$[\![\mathsf{x} \rightarrow 7]\!]_V^{\text{assert}}\eta\sigma, \qquad [\![\neg\, \mathsf{x} \rightarrow 7]\!]_V^{\text{assert}}\eta\sigma, \qquad [\![\mathsf{x} \rightarrow 7 \vee \neg\, \mathsf{x} \rightarrow 7]\!]_V^{\text{assert}}\eta\sigma$$

are all false.

As one expects in an intuitionistic logic, one has axiom schemata such as

$$\begin{aligned} \forall x.\ \neg\, P &\Leftrightarrow \neg(\exists x.\ P) \\ \exists x.\ \neg\, P &\Rightarrow \neg(\forall x.\ P) \\ P &\Rightarrow \neg(\neg\, P), \end{aligned}$$

but not the converses of the second and third lines. (Pure assertions, however, which are those that do not contain $\rightarrow$, behave classically.)

Insight into the operation of independent conjunction is provided by a simple example: Suppose η is a store that maps x and y into distinct locations, and consider the heaps

$$\sigma_1 = \{\langle \eta\, \mathsf{x}, 1\rangle\} \qquad \text{and} \qquad \sigma_2 = \{\langle \eta\, \mathsf{y}, 2\rangle\},$$

which have disjoint domains. Then

If P is:	then $[\![P]\!]\eta\sigma$ is:
$\mathsf{x} \rightarrow 1$	$\sigma_1 \subseteq \sigma$
$\mathsf{y} \rightarrow 2$	$\sigma_2 \subseteq \sigma$
$\mathsf{x} \rightarrow 1 \;\&\; \mathsf{y} \rightarrow 2$	$\sigma_1 \cup \sigma_2 \subseteq \sigma$
$\mathsf{x} \rightarrow 1 \;\&\; (\mathsf{x} \rightarrow 1 \vee \mathsf{y} \rightarrow 2)$	$\sigma_1 \cup \sigma_2 \subseteq \sigma$
$(\mathsf{x} \rightarrow 1 \vee \mathsf{y} \rightarrow 2) \;\&\; (\mathsf{x} \rightarrow 1 \vee \mathsf{y} \rightarrow 2)$	$\sigma_1 \cup \sigma_2 \subseteq \sigma$
$\mathsf{x} \rightarrow 1 \;\&\; \mathsf{y} \rightarrow 2 \;\&\; (\mathsf{x} \rightarrow 1 \vee \mathsf{y} \rightarrow 2)$	**false**.

In general, independent conjunction is described by the axiom schemata

$$\begin{aligned}
P_1 \mathbin{\&} P_2 &\Rightarrow P_1 \wedge P_2 \\
P_1 \wedge P_2 &\Rightarrow P_1 \mathbin{\&} P_2 \quad \text{when } P_2 \text{ is pure} \\
P_1 \mathbin{\&} P_2 &\Leftrightarrow P_2 \mathbin{\&} P_1 \\
(P_1 \mathbin{\&} P_2) \mathbin{\&} P_3 &\Leftrightarrow P_1 \mathbin{\&} (P_2 \mathbin{\&} P_3) \\
(P_1 \mathbin{\&} P_3) \vee (P_2 \mathbin{\&} P_3) &\Leftrightarrow (P_1 \vee P_2) \mathbin{\&} P_3 \\
(P_1 \vee P_3) \mathbin{\&} (P_2 \vee P_3) &\Rightarrow (P_1 \mathbin{\&} P_2) \vee P_3 \\
(\exists x.\ P_1) \mathbin{\&} P_2 &\Leftrightarrow \exists x.\ (P_1 \mathbin{\&} P_2) \quad \text{when } x \text{ not free in } P_2,
\end{aligned}$$

and the inference rule

$$\frac{P_1 \Rightarrow P_2}{P_1 \mathbin{\&} P_3 \Rightarrow P_2 \mathbin{\&} P_3}.$$

Note, however, that the analogous schemata $(P_1 \Rightarrow P_2) \Rightarrow (P_1 \mathbin{\&} P_3 \Rightarrow P_2 \mathbin{\&} P_3)$ is not valid.

3 Inference Rules for Specifications

By prohibiting expressions from depending upon the heap, we insure that both the partial-correctness rules given by Hoare and the analogous total-correctness rules (see, for example, [23, Chapter 3]) remain valid, even for assertions containing &. Thus we need to introduce additional rules only for the new commands that depend upon or affect the heap. Since the new commands always terminate, these rules are identical for partial and total correctness.

First, we have the command $x := E.i$, which examines the tuple stored in the heap at the location that is the value of E, and makes the ith component of this tuple the value of x. This operation is similar enough to assignment that one might hope to extend Hoare's assignment axiom to describe it. But that axiom involves a substitution that would insert $E.i$ into expressions, violating their purity. (If such substitutions were allowed, they would not preserve &.) Instead, we must mimic the effect of such a substitution by using existential quantifiers:

Suppose that $x_1, \ldots, x_n$ are distinct variables that do not occur free in E, that $1 \leq i \leq n$, and that x_i does not occur free in P. Let $P^{(i)}$ denote the result of substituting x_i for x in P. Then

$$\{\exists x_i.\ (P^{(i)} \wedge \exists x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n.\ E \rightarrow x_1, \ldots, x_n)\}$$
$$x := E.i$$
$$\{P\}.$$

Next, we have the command $x := \mathbf{cons}_n(E_1, \ldots, E_n)$, which extends the heap with a new location mapped into the n-tuple of values of $E_1, \ldots, E_n$, and then makes the new location the value of x. Again, we cannot use Hoare's assignment axiom; instead we introduce an existential quantifier in the postcondition, in the style of Floyd [17]. The & operation is used to assert that the new location plays no role in any part of the postcondition that is inherited from the precondition:

Suppose that the variables x and x' are distinct, and that x' does not occur free in $E_1, \ldots E_n$, or P. Let X' denote the result of substituting x' for x in the expression or assertion X. Then

$$\{P\}$$
$$x := \mathbf{cons}_n(E_1, \ldots, E_n)$$
$$\{\exists x'.\ (P' \mathbin{\&} x \rightarrow E'_1, \ldots, E'_n)\}.$$

Happily, this rule can be simplified when the variable x does not occur in the precondition or the right side of the command being specified:

Suppose that the variables x does not occur free in $E_1, \ldots, E_n$ or P. Then

$$\{P\}$$
$$x := \mathbf{cons}_n(E_1, \ldots, E_n)$$
$$\{P \mathbin{\&} x \rightarrow E_1, \ldots, E_n\}.$$

Finally, we have a rule for the command $E.i := E'$, which alters the heap at the location that is the value of E by changing the ith component of the tuple at that location to the value of E'. Here the & operator in the precondition separates an assertion about the heap at the value of E from assertions about other parts of the heap, which are not affected by the command:

Suppose that the variables $x_1, \ldots, x_m$ do not occur free in the expression E or E', and that $1 \leq i \leq n$. Then

$$\begin{aligned}&\{\exists x_1, \ldots, x_m.\, (E \rightarrow E_1, \ldots, E_i, \ldots, E_n \,\&\, P)\}\\ &E.i := E'\\ &\{\exists x_1, \ldots, x_m.\, (E \rightarrow E_1, \ldots, E', \ldots, E_n \,\&\, P)\}.\end{aligned}$$

The importance of using an intuitionistic logic is illustrated by the following instance of the simplified **cons**-rule:

$$\begin{aligned}&\{\neg(\exists \mathsf{x}.\ \mathsf{x} \rightarrow 1, 2)\}\\ &\mathsf{y} := \mathbf{cons}_2(1, 2)\\ &\{(\neg(\exists \mathsf{x}.\ \mathsf{x} \rightarrow 1, 2)) \,\&\, \mathsf{y} \rightarrow 1, 2\}.\end{aligned}$$

Although its postcondition is always false, this instance is not unsound, because the precondition is always false in our intuitionistic logic — since it must hold for all extensions of the heap that was current immediately before execution of the **cons** operation, including the extension that is current immediately afterwards.

(This instance was unsound, however, in the classical logic used in the preliminary version of this paper.)

4 Inductive Definition of Predicates

To deal with real programs (even small ones), we must permit predicates to be defined by induction over sets of abstract data.

For instance, to prove the correctness of a program that uses some list representation, it is not enough to be able to assert that something is a list representation; one must say that it represents a particular sequence of values. Thus one must define "is representation of" as a function from the abstract data set "sequence of values" to predicates.

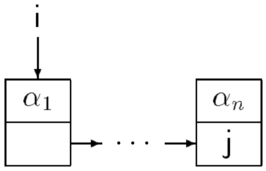
Specifically (though without being as formal as in previous sections), we will allow variables that occur in assertions but not in programs (often called logical or ghost variables) to range over inductively defined sets, and we will allow functions from such sets to predicates to be defined by induction (i.e., by primitive recursion).

As a first example, consider the representation of sequences by singly-linked lists. (This example is essentially similar to one of Burstall's.) We write ϵ to denote the empty sequence, use other Greek letters as variables ranging over sequences, write $\cdot$ to indicate concatenation, and assume that a value is a sequence of length one. Then

$$\mathsf{list}\ \epsilon\,(\mathsf{i},\mathsf{j}) \equiv \mathsf{i} = \mathsf{j}$$

$$\mathsf{list}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{k}) \equiv \exists \mathsf{j}.\ \mathsf{i} \rightarrow \mathsf{a},\mathsf{j}\ \&\ \mathsf{list}\ \alpha\,(\mathsf{j},\mathsf{k})$$

is an inductive definition of a function list, from sequences to predicates, such that $\mathsf{list}\ \alpha\,(\mathsf{i},\mathsf{j})$ asserts that the sequence α is represented by the list fragment from location i to j. More precisely, it asserts that there is a succession of $\mathbf{cons}_2$-cells beginning at location i, containing the successive elements of α as their first components, and linked together by their second components, with a final link containing j:



From this definition, it is obvious that

$$\mathsf{list}\ \mathsf{a}\,(\mathsf{i},\mathsf{j}) \Leftrightarrow \mathsf{i} \rightarrow \mathsf{a},\mathsf{j}.$$

Less trivially, one can show by induction on the length of α that

$$\mathsf{list}\ \alpha{\cdot}\beta\,(\mathsf{i},\mathsf{k}) \Leftrightarrow \exists \mathsf{j}.\ \mathsf{list}\ \alpha\,(\mathsf{i},\mathsf{j})\ \&\ \mathsf{list}\ \beta\,(\mathsf{j},\mathsf{k}),$$

and from this result one can obtain

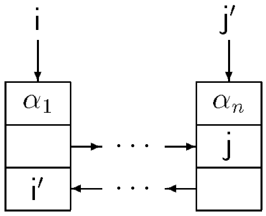
$$\mathsf{list}\ \alpha{\cdot}\mathsf{b}\,(\mathsf{i},\mathsf{k}) \Leftrightarrow \exists \mathsf{j}.\ \mathsf{list}\ \alpha\,(\mathsf{i},\mathsf{j})\ \&\ \mathsf{j} \rightarrow \mathsf{b},\mathsf{k}.$$

Because of the use of & in the definition of list, the assertion $\mathsf{list}\ \alpha\,(\mathsf{i},\mathsf{j})$ implies that the list fragment from i to j contains a distinct $\mathbf{cons}_2$-cell for each element of α. Nevertheless, j can be the location of one of these cells, in which case a cyclic structure is being described. For instance, when α is nonempty, $\mathsf{list}\ \alpha\,(\mathsf{i},\mathsf{i})$ describes a cycle whose length is the length of α.

However, when $\mathsf{i} = \mathbf{nil}$, the assertion $\mathsf{i} \rightarrow \mathsf{a},\mathsf{j}$ cannot hold, so that $\mathsf{list}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{k})$ cannot hold. In fact,

$$\mathsf{list}\ \alpha\,(\mathsf{i},\mathsf{j}) \Rightarrow \begin{pmatrix} & (\mathsf{i} = \mathbf{nil} \Rightarrow (\alpha = \epsilon \wedge \mathsf{j} = \mathbf{nil})) \\ \wedge & (\mathsf{i} \neq \mathsf{j} \Rightarrow \alpha \neq \epsilon). \end{pmatrix}$$

A more complex example, which goes beyond Burstall's framework, is the representation of sequences by doubly-linked lists. We write $\mathsf{dlist}\ \alpha\,(i, i', j, j')$ to assert the existence of a a collection of **cons**$_3$-cells whose first components give the elements of α, whose second components are the links of a forward list beginning at i with a final link containing j, and whose third components are the links of a backward list beginning at j' with a final link containing i':



This function can be defined inductively by

$$\mathsf{dlist}\ \epsilon\,(i, i', j, j') \equiv i = j\ \&\ i' = j'$$

$$\mathsf{dlist}\ a{\cdot}\alpha\,(i, i', k, k') \equiv \exists j.\ i \mapsto a, j, i'\ \&\ \mathsf{dlist}\ \alpha\,(j, i, k, k').$$

From this definition, it is obvious that

$$\mathsf{dlist}\ a\,(i, i', j, j') \Leftrightarrow i \mapsto a, j, i'\ \&\ i = j'.$$

Less trivially, one can show by induction on the length of α that

$$\mathsf{dlist}\ \alpha{\cdot}\beta\,(i, i', k, k') \Leftrightarrow \exists j, j'.\ \mathsf{dlist}\ \alpha\,(i, i', j, j')\ \&\ \mathsf{dlist}\ \beta\,(j, j', k, k'),$$

and from this result one can obtain

$$\mathsf{dlist}\ \alpha{\cdot}b\,(i, i', k, k') \Leftrightarrow \exists j'.\ \mathsf{dlist}\ \alpha\,(i, i', k', j')\ \&\ k' \mapsto b, k, j'.$$

Much as in the singly-linked case, the assertion $\mathsf{dlist}\ \alpha\,(i, i', j, j')$ describes a list fragment containing a distinct **cons**$_3$-cell for each element of α. The existence of unique back-links prevents a list from having a cyclic tail, but one can still have an isolated cyclic list, which would be described by $\mathsf{dlist}\ \alpha\,(i, i', i, i')$ when α is nonempty.

Finally, to distinguish empty from nonempty lists, one can derive

$$\mathsf{dlist}\ \alpha\,(i, i', j, j') \Rightarrow \left(\begin{array}{ll} & (i = \mathbf{nil} \Rightarrow (\alpha = \epsilon \wedge j = \mathbf{nil} \wedge i' = j')) \\ \wedge & (j' = \mathbf{nil} \Rightarrow (\alpha = \epsilon \wedge i' = \mathbf{nil} \wedge i = j)) \\ \wedge & (i \neq j \Rightarrow \alpha \neq \epsilon) \\ \wedge & (i' \neq j' \Rightarrow \alpha \neq \epsilon) \end{array}\right)$$

5 Annotated Specifications

For even a small program, a formal proof of its specification, in the sense of a sequence (or tree) of instances of inference rules, is usually too large to be readable. Fortunately, one can annotate a specification with intermediate assertions in such a way that its proof can be mechanically and straightforwardly derived from the annotations.

In the next section, we will give an illustrative proof in the form of an annotated specification. Before doing so, however, it is useful — independently of the main subject of this paper — to define the notion precisely (for the partial-correctness case).

The basic idea goes back at least to the "proof outlines" of concurrent computations by Owicki and Gries [24], which have been formalized by Schneider [25, Chapter 4]. Our formulation for sequential programs, however, permits the omission of many intermediate assertions that can be derived straightforwardly from their context.

We will call a specification *annotated* if it can be derived from the inference rules to be given below. All of these rules are easily derivable from Hoare's rules and the rules given in Section 3, except that intermediate assertions occur in the conclusions of certain rules. Because of the presence of these intermediate assertions, the rules are *determinate*, i.e., every specification can be the conclusion of an instance of at most one rule, and the premisses of this instance are determined by its conclusion. Thus it is straightforward to derive a proof of an annotated specification from the specification itself, and a Hoare-style proof (albeit using derived rules) can then be obtained by erasing the intermediate assertions. (Of course, this proof will contain verification conditions, i.e., unproved implications between assertions, that must be verified independently. Pragmatically, one must supply enough intermediate assertions to make this verification tractable.)

The first two rules deal with assignment. Their determinacy stems from the fact that, in Hoare's assignment axiom, the precondition $P/x \to E$ is determined by the postcondition P and the assignment command $x := E$:

$$\frac{P_0 \Rightarrow P/x \to E}{\{P_0\}\; x := E\; \{P\}} \qquad\qquad \frac{\{P_0\}\; C\; \{P/x \to E\}}{\{P_0\}\; C\, ;\, x := E\; \{P\}}.$$

A similar situation holds for the rule for the select-and-assign command $x := E.i$, where again the precondition is determined by the postcondition and the command:

Suppose that $x_1, \ldots, x_n$ are distinct variables that do not occur free in E, that $1 \leq i \leq n$, and that x_i does not occur free in P. Let $P^{(i)}$ denote the result of substituting x_i for x in P. Then

$$\frac{P_0 \Rightarrow \exists x_i.\ (P^{(i)} \wedge \exists x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n.\ E \rightarrow x_1, \ldots, x_n)}{\{P_0\}\ x := E.i\ \{P\}}$$

$$\frac{\{P_0\}\ C\ \{\exists x_i.\ (P^{(i)} \wedge \exists x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n.\ E \rightarrow x_1, \ldots, x_n)\}}{\{P_0\}\ C\ ;\ x := E.i\ \{P\}.}$$

In contrast, the remaining axioms for heap-dependent commands given in Section 3 have the property that their postcondition is determined by their precondition and the heap-dependent command. This insures the determinacy of:

Suppose that the variables x and x' are distinct, and that x' does not occur free in $E_1, \ldots E_n$, or P. Let X' denote the result of substituting x' for x in the expression or assertion X. Then

$$\frac{(\exists x'.\ (P' \ \&\ x \rightarrow E_1', \ldots, E_n')) \Rightarrow P''}{\{P\}\ x := \mathbf{cons}_n(E_1, \ldots, E_n)\ \{P''\}}$$

$$\frac{\{\exists x'.\ (P' \ \&\ x \rightarrow E_1', \ldots, E_n')\}\ C\ \{P''\}}{\{P\}\ x := \mathbf{cons}_n(E_1, \ldots, E_n)\ ;\ C\ \{P''\}.}$$

Suppose that the variables $x_1, \ldots, x_m$ do not occur free in the expression E or E', and that $1 \leq i \leq n$. Then

$$\frac{(\exists x_1, \ldots, x_m.\ (E \rightarrow E_1, \ldots, E', \ldots, E_n \ \&\ P)) \Rightarrow P''}{\{\exists x_1, \ldots, x_m.\ (E \rightarrow E_1, \ldots, E_i, \ldots, E_n \ \&\ P)\}\ E.i := E'\ \{P''\}}$$

$$\frac{\{\exists x_1, \ldots, x_m.\ (E \rightarrow E_1, \ldots, E', \ldots, E_n \ \&\ P)\}\ C\ \{P''\}}{\{\exists x_1, \ldots, x_m.\ (E \rightarrow E_1, \ldots, E_i, \ldots, E_n \ \&\ P)\}\ E.i := E'\ ;\ C\ \{P''\}.}$$

Next, we have rules for **skip**, **if**, and **while** commands:

$$\frac{P \Rightarrow P'}{\{P\}\ \mathbf{skip}\ \{P'\}} \qquad \frac{\{P \wedge B\}\ C_1\ \{P'\} \qquad \{P \wedge \neg B\}\ C_2\ \{P'\}}{\{P\}\ \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}\ \{P'\}}$$

$$\frac{\{P \wedge B\}\ C\ \{P\} \qquad (P \wedge \neg B) \Rightarrow P'}{\{P\}\ \mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}\ \{P'\}.}$$

Finally, there are rules corresponding to Hoare's rules for sequencing, strengthening preconditions, and weakening postconditions. It is in the conclusions of these rules that the intermediate assertions appear:

$$\frac{\{P\}\ C\ \{P'\} \qquad \{P'\}\ C'\ \{P''\}}{\{P\}\ C\,;\{P'\}\ C'\ \{P''\}}$$

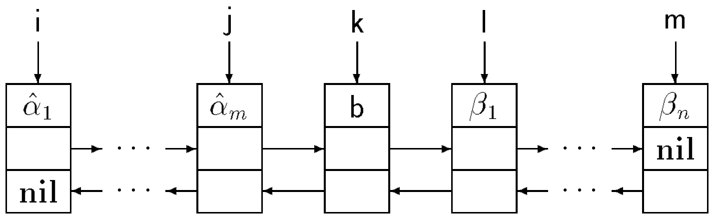
$$\frac{P \Rightarrow P' \qquad \{P'\}\ C\ \{P''\}}{\{P\}\ \{P'\}\ C\ \{P''\}} \qquad\qquad \frac{\{P\}\ C\ \{P'\} \qquad P' \Rightarrow P''}{\{P\}\ C\ \{P'\}\ \{P''\}.}$$

6 An Example

In conclusion, we illustrate our formalism with an example: an annotated partial-correctness specification of a program for deleting zero-valued elements from a doubly-linked list.

Throughout execution of the program, the forward linkage of the list begins at location i and the backward linkage begins at location m. There is a single loop which moves forward through the list. Within this loop, the deletion operation is symmetric about the cell to be deleted.

Just before the test b = 0, the variable k points to the cell to be tested and perhaps removed, the preceding list fragment has a forward linkage from i to k and a backward linkage from j to **nil**, and the following list fragment has a forward linkage from l to **nil** and a backward linkage from m to k:



If the current cell is to be removed, the program tests whether the preceding list fragment is empty, and resets either i or the last forward link to point to the following list fragment. Then it performs a symmetric operation on the following list fragment.

The notation $\alpha - 0$ denotes the sequence obtained from α by deleting all occurrences of zero.

$\{\mathsf{dlist}\ \gamma\ (\mathsf{i}, \mathbf{nil}, \mathbf{nil}, \mathsf{m})\}$

$\{\exists \alpha, \beta, \hat{\alpha}.\ \mathsf{dlist}\ \hat{\alpha}\ (\mathsf{i}, \mathbf{nil}, \mathsf{i}, \mathbf{nil})\ \&\ \mathsf{dlist}\ \beta\ (\mathsf{i}, \mathbf{nil}, \mathbf{nil}, \mathsf{m})$
$\quad \&\ \alpha{\cdot}\beta = \gamma\ \&\ \alpha - 0 = \hat{\alpha}\}$

$\mathsf{k} := \mathsf{i}\ ;\ \mathsf{j} := \mathbf{nil}\ ;$

$\{\exists \alpha, \beta, \hat{\alpha}.\ \mathsf{dlist}\ \hat{\alpha}\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ \&\ \mathsf{dlist}\ \beta\ (\mathsf{k}, \mathsf{j}, \mathbf{nil}, \mathsf{m})$
$\quad \&\ \alpha{\cdot}\beta = \gamma\ \&\ \alpha - 0 = \hat{\alpha}\}$

$\mathbf{while}\ \mathsf{k} \neq \mathbf{nil}\ \mathbf{do}$

$\quad \{\exists \alpha, \beta, \hat{\alpha}, \mathsf{b}, \mathsf{l}.\ \mathsf{dlist}\ \hat{\alpha}\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ \&\ \mathsf{k} \to \mathsf{b}, \mathsf{l}, \mathsf{j}\ \&\ \mathsf{dlist}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})$
$\qquad \&\ \alpha{\cdot}\mathsf{b}{\cdot}\beta = \gamma\ \&\ \alpha - 0 = \hat{\alpha}\}$

$\quad \mathsf{b} := \mathsf{k}.1\ ;\ \mathsf{l} := \mathsf{k}.2\ ;$

$\quad \{\exists \alpha, \beta, \hat{\alpha}.\ \mathsf{dlist}\ \hat{\alpha}\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ \&\ \mathsf{k} \to \mathsf{b}, \mathsf{l}, \mathsf{j}\ \&\ \mathsf{dlist}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})$
$\qquad \&\ \alpha{\cdot}\mathsf{b}{\cdot}\beta = \gamma\ \&\ \alpha - 0 = \hat{\alpha}\}$

$\quad \mathbf{if}\ \mathsf{b} = 0\ \mathbf{then}$

$\qquad \{\exists \alpha, \beta, \hat{\alpha}.\ \mathsf{dlist}\ \hat{\alpha}\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ \&\ \mathsf{dlist}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})$
$\qquad\quad \&\ \alpha{\cdot}0{\cdot}\beta = \gamma\ \&\ (\alpha{\cdot}0) - 0 = \hat{\alpha}\}$

$\qquad \{\exists \alpha, \beta, \hat{\alpha}.\ \mathsf{dlist}\ \hat{\alpha}\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ \&\ \mathsf{dlist}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})$
$\qquad\quad \&\ \alpha{\cdot}\beta = \gamma\ \&\ \alpha - 0 = \hat{\alpha}\}$

$\qquad \mathbf{if}\ \mathsf{j} = \mathbf{nil}\ \mathbf{then}$

$\qquad\quad \{\exists \alpha, \beta, \hat{\alpha}.\ \hat{\alpha} = \epsilon\ \&\ \mathsf{i} = \mathsf{k}\ \&\ \mathbf{nil} = \mathsf{j}\ \&\ \mathsf{dlist}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})$
$\qquad\qquad \&\ \alpha{\cdot}\beta = \gamma\ \&\ \alpha - 0 = \hat{\alpha}\}$

$\qquad\quad \mathsf{i} := \mathsf{l}$

$\qquad\quad \{\exists \alpha, \beta, \hat{\alpha}.\ \hat{\alpha} = \epsilon\ \&\ \mathsf{i} = \mathsf{l}\ \&\ \mathbf{nil} = \mathsf{j}\ \&\ \mathsf{dlist}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})$
$\qquad\qquad \&\ \alpha{\cdot}\beta = \gamma\ \&\ \alpha - 0 = \hat{\alpha}\}$

$\qquad \mathbf{else}$

$\qquad\quad \{\exists \alpha, \beta, \hat{\alpha}, \mathsf{a}, \mathsf{n}.\ \mathsf{dlist}\ \hat{\alpha}\ (\mathsf{i}, \mathbf{nil}, \mathsf{j}, \mathsf{n})\ \&\ \mathsf{j} \to \mathsf{a}, \mathsf{k}, \mathsf{n}\ \&\ \mathsf{dlist}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})$
$\qquad\qquad \&\ \alpha{\cdot}\beta = \gamma\ \&\ \alpha - 0 = \hat{\alpha}{\cdot}\mathsf{a}\}$

$\qquad\quad \mathsf{j}.2 := \mathsf{l}$

$\qquad\quad \{\exists \alpha, \beta, \hat{\alpha}, \mathsf{a}, \mathsf{n}.\ \mathsf{dlist}\ \hat{\alpha}\ (\mathsf{i}, \mathbf{nil}, \mathsf{j}, \mathsf{n})\ \&\ \mathsf{j} \to \mathsf{a}, \mathsf{l}, \mathsf{n}\ \&\ \mathsf{dlist}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})$
$\qquad\qquad \&\ \alpha{\cdot}\beta = \gamma\ \&\ \alpha - 0 = \hat{\alpha}{\cdot}\mathsf{a}\}$

$\qquad \mathbf{fi}\ ;$

$\{\exists\alpha,\beta,\hat{\alpha}.\ \mathsf{dlist}\ \hat{\alpha}\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\ \&\ \mathsf{dlist}\ \beta\,(\mathsf{l},\mathsf{k},\mathbf{nil},\mathsf{m})$
$\quad\&\ \alpha{\cdot}\beta=\gamma\ \&\ \alpha-0=\hat{\alpha}\}$

if $\mathsf{l}=\mathbf{nil}$ **then**

$\{\exists\alpha,\beta,\hat{\alpha}.\ \mathsf{dlist}\ \hat{\alpha}\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\ \&\ \beta=\epsilon\ \&\ \mathsf{l}=\mathbf{nil}\ \&\ \mathsf{k}=\mathsf{m}$
$\quad\&\ \alpha{\cdot}\beta=\gamma\ \&\ \alpha-0=\hat{\alpha}\}$

$\mathsf{m}:=\mathsf{j}$

$\{\exists\alpha,\beta,\hat{\alpha}.\ \mathsf{dlist}\ \hat{\alpha}\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\ \&\ \beta=\epsilon\ \&\ \mathsf{l}=\mathbf{nil}\ \&\ \mathsf{j}=\mathsf{m}$
$\quad\&\ \alpha{\cdot}\beta=\gamma\ \&\ \alpha-0=\hat{\alpha}\}$

else

$\{\exists\alpha,\beta,\hat{\alpha},\mathsf{a},\mathsf{n}.\ \mathsf{dlist}\ \hat{\alpha}\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\ \&\ \mathsf{l}\rightarrow\mathsf{a},\mathsf{n},\mathsf{k}\ \&\ \mathsf{dlist}\ \beta\,(\mathsf{n},\mathsf{l},\mathbf{nil},\mathsf{m})$
$\quad\&\ \alpha{\cdot}\mathsf{a}{\cdot}\beta=\gamma\ \&\ \alpha-0=\hat{\alpha}\}$

$\mathsf{l}.3:=\mathsf{j}$

$\{\exists\alpha,\beta,\hat{\alpha},\mathsf{a},\mathsf{n}.\ \mathsf{dlist}\ \hat{\alpha}\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\ \&\ \mathsf{l}\rightarrow\mathsf{a},\mathsf{n},\mathsf{j}\ \&\ \mathsf{dlist}\ \beta\,(\mathsf{n},\mathsf{l},\mathbf{nil},\mathsf{m})$
$\quad\&\ \alpha{\cdot}\mathsf{a}{\cdot}\beta=\gamma\ \&\ \alpha-0=\hat{\alpha}\}$

fi

else

$\{\exists\alpha,\beta,\hat{\alpha}.\ \mathsf{dlist}\ \hat{\alpha}{\cdot}\mathsf{b}\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{k})\ \&\ \mathsf{dlist}\ \beta\,(\mathsf{l},\mathsf{k},\mathbf{nil},\mathsf{m})$
$\quad\&\ \alpha{\cdot}\mathsf{b}{\cdot}\beta=\gamma\ \&\ (\alpha{\cdot}\mathsf{b})-0=\hat{\alpha}{\cdot}\mathsf{b}\}$

$\{\exists\alpha,\beta,\hat{\alpha}.\ \mathsf{dlist}\ \hat{\alpha}\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{k})\ \&\ \mathsf{dlist}\ \beta\,(\mathsf{l},\mathsf{k},\mathbf{nil},\mathsf{m})$
$\quad\&\ \alpha{\cdot}\beta=\gamma\ \&\ \alpha-0=\hat{\alpha}\}$

$\mathsf{j}:=\mathsf{k}$

fi ;

$\{\exists\alpha,\beta,\hat{\alpha}.\ \mathsf{dlist}\ \hat{\alpha}\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\ \&\ \mathsf{dlist}\ \beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})$
$\quad\&\ \alpha{\cdot}\beta=\gamma\ \&\ \alpha-0=\hat{\alpha}\}$

$\mathsf{k}:=\mathsf{l}$

od

$\{\mathsf{dlist}\ (\gamma-0)\,(\mathsf{i},\mathbf{nil},\mathbf{nil},\mathsf{m})\}$

7 Future Directions

The work described here is very preliminary, so that the most immediate need is to explore more examples. In particular, it is not clear that every abstract structure represented by mutable data structures can be defined inductively; for example, mutable data structures are often used to represent cyclic graphs.

It may be possible to extend our approach to dynamic logic [26], where commands can occur as modal operators within assertions. This would necessitate giving meaning to the execution of commands in incomplete states. When a command tries to examine a heap outside of its domain, it would raise a "heap fault" that would make the immediately enclosing assertion false.

It is easy to prove

$$\{\mathbf{true}\}$$
$$\mathsf{x} := \mathbf{cons}_2(1, 2)\ ;$$
$$\{\mathsf{x} \to 1, 2\}$$
$$\mathsf{x} := 3$$
$$\{(\exists \mathsf{x}.\ \mathsf{x} \to 1, 2) \wedge \mathsf{x} = 3\}.$$

Here the existentially quantified location is disconnected from the data structures accessible to the computation, and can be eliminated by garbage collection. Of course, one can view garbage collection as a program optimization with no effect on observable computations, so that this example is sound. Nevertheless the fact that $\exists \mathsf{x}.\ \mathsf{x} \to 1, 2$ is an "unobservable assertion" is worrisome.

It should be straightforward to impose upon the present development a type system based upon recursive data type declarations (similar to those in Standard ML but without the reference concept). More refined type systems might permit the specification of which fields are mutable or where pointers are required to be unique.

Finally, there is a need to move towards some form of higher-order formalism, akin to object-oriented programming, where the mutable shared data structures can contain closures. An obvious question is whether there is a connection with syntactic control of interference [27, 28, 29] or specification logic [30, 31, 32], both of which deal with interference in a higher-order but heap-free setting. Superficially this work seems quite different, but the recurring use of possible-world semantics is suggestive.

Acknowledgements

The author would like to thank Peter O'Hearn for his insights and helpful suggestions.

References

[1] Rodney M. Burstall. Some techniques for proving correctness of programs which alter data structures. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland, 1972.

[2] T. Kowaltowski. Correctness of programs manipulating data structures. Memorandum ERL-M404, University of California, Berkeley, California, September 1973.

[3] Stephen A. Cook and Derek C. Oppen. An assertion language for data structures. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, pages 160–166, New York, 1975. ACM.

[4] Derek C. Oppen and Stephen A. Cook. Proving assertions about programs that manipulate data structures. In *Proceedings of Seventh Annual ACM Symposium on Theory of Computing*, pages 107–116, New York, 1975. ACM.

[5] Joseph M. Morris. A general axiom of assignment; assignment and linked data structures; a proof of the Schorr-Waite algorithm. In Manfried Broy and Gunther Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 25–51. D. Reidel, Dordrecht, Holland, 1982.

[6] Ian A. Mason. *The Semantics of Destructive Lisp*. CSLI Lecture Notes Number 5. Center for the Study of Language and Information, Menlo Park, CA, 1986.

[7] Ian A. Mason. Verification of programs that destructively manipulate data. *Science of Computer Programming*, 10(2):177–210, April 1988.

[8] Ian A. Mason and Carolyn Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, July 1991.

[9] Furio Honsell, Ian A. Mason, Scott Smith, and Carolyn Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, May 15, 1995.

[10] Ian A. Mason. A first order logic of effects. *Theoretical Computer Science*, 185(2):277–318, October 20, 1997.

[11] Andrew M. Pitts and Ian D. B. Stark. Operational reasoning for functions with local state. In Andrew D. Gordon and Andrew M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Cambridge University Press, 1998.

[12] Ian D. B. Stark. *Names and Higher-Order Functions*. Ph. D. dissertation, University of Cambridge, Cambridge, England, December 1994.

[13] Ian D. B. Stark. Categorical models for local names. *Lisp and Symbolic Computation*, 9(1):77–107, February 1996.

[14] Ian D. B. Stark. Names, equations, relations: Practical ways to reason about *new*. *Fundamenta Informaticae*, 33:369–396, 1998.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969. Reprinted in [33, pages 89–100].

[16] C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14(1):39–45, January 1971. Reprinted in [33, pages 101–115].

[17] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[18] Peter Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.

[19] F. William Lawvere. Functorial semantics of algebraic theories. *Proceedings of the National Academy of Sciences of the United States of America*, 50(1):869–872, July 1963.

[20] Saul A. Kripke. Semantical analysis of intuitionistic logic i. In John N. Crossley and Michael A. E. Dummett, editors, *Formal Systems and Recursive Functions*, Studies in Logic and the Foundations of Mathematics, pages 92–130, Amsterdam, 1965. North-Holland.

[21] Peter W. O'Hearn. Private communication. dated January 17, 2000.

[22] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 1999.

[23] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, Cambridge, England, 1998.

[24] Susan Speer Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976. Reprinted in [33, pages 130–152].

[25] Fred B. Schneider. *On Concurrent Programming*. Springer-Verlag, New York, 1997.

[26] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *17th Annual Symposium on Foundations of Computer Science*, pages 109–121, Long Beach, California, 1976. IEEE Computer Society.

[27] John C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, New York, 1978. ACM. Reprinted in [34, vol. 1, pages 273–286].

[28] Peter W. O'Hearn, A. John Power, Makoto Takeyama, and Robert D. Tennent. Syntactic control of interference revisited. *Electronic Notes in Theoretical Computer Science*, 1, 1995. Reprinted in [34, vol. 2, pages 189–225].

[29] Uday S. Reddy. Global state considered unnecessary: An introduction to object-based semantics. *Lisp and Symbolic Computation*, 9(1):7–76, February 1996. Reprinted in [34, vol. 2, pages 227–295].

[30] John C. Reynolds. Idealized Algol and its specification logic. In Danielle Néel, editor, *Tools and Notions for Program Construction*, pages 121–161, Cambridge, England, 1982. Cambridge University Press. Reprinted in [34, vol. 1, pages 125–156].

[31] Robert D. Tennent. Semantical analysis of specification logic. *Information and Computation*, 85(2):135–162, April 1990. Reprinted in [34, vol. 2, pages 41–64].

[32] Peter W. O'Hearn and Robert D. Tennent. Semantical analysis of specification logic, 2. *Information and Computation*, 107(1):25–57, November 1993. Reprinted in [34, vol. 2, pages 65–93].

[33] David Gries, editor. *Programming Methodology*. Springer-Verlag, New York, 1978.

[34] Peter W. O'Hearn and Robert D. Tennent, editors. *ALGOL-like Languages*. Birkhäuser, Boston, Massachusetts, 1997. Two volumes.