Semantic Analysis of Pointer Aliasing, Allocation and Disposal in Hoare Logic

Cristiano Calcagno
Department of Computer Science
QMW College
and
DISI, University of Genova
ccris@dcs.qmw.ac.uk

Samin Ishtiaq
Department of Computer Science
Queen Mary & Westfield College
si@dcs.qmw.ac.uk

Peter W. O'Hearn
Department of Computer Science
Queen Mary & Westfield College
ohearn@dcs.qmw.ac.uk

ABSTRACT

Bornat has recently described an approach to reasoning about pointers, building on work of Morris. Here we describe a semantics that validates the approach, and use it to help devise axioms for operations that allocate and dispose of memory.

1. INTRODUCTION

It is widely acknowledged that pointers cause problems for program-proving formalisms (e.g. [8, 17, 13, 16, 9, 1, 14, 7]), but there is less agreement on precisely what the problems are. So, before describing our own work, we first discuss where we believe the difficulties lie.

The first issue that must be faced is *aliasing*, where distinct expressions can denote the same l-value. The problem here can be seen by reference to Hoare logic, where assignment is treated using substitution on the object-language level:

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

For this treatment of assignment to be sound it is necessary that different identifiers are not aliases. With pointers the problem is that aliasing is not an exceptional circumstance: for example, it will often be the case that distinct dereferencing expressions, references $x.a$ and $y.a$ to records in the heap, are aliases even if pointers x and y are not. As a result, an assignment to $x.a$ might alter the value of seemingly unrelated expressions.

There are a number of ways to deal with aliasing. One is by "dropping down a level," and including an explicit store parameter in assertions. Although this approach works technically, it is essentially compiling to another language, and as a result comes with a price: assertions become more complex, due to a proliferation of occurrences of store parameters.

To appear in the proceedings of the second International Conference on Principles and Practice of Declarative Programmming (PPDP 2000).

An interesting approach was taken by Morris in the early eighties [15]: it extends the notion of substitution to apply to object components as well as to simple variables. The idea is that the store is organized as records, whose components contain data as well as pointers to other records.

The intent is then that the substitution $E[E'/V.a]$ behaves as if the l-value corresponding to a record component $V.a$ is overwritten. The crucial case is when E is itself of the form $V'.a$, in which case the l-values of $V.a$ and $V'.a$ might be aliases. This is dealt, informally, by defining substitution as follows:

$$x.a[17/y.a] \ \mapsto\ \texttt{if}\ x == y\ \texttt{then}\ 17\ \texttt{else}\ x.a.$$

The idea here is that if $x.a$ and $y.a$ are aliases, then assigning 17 to $y.a$ changes the value of $x.a$ to be 17. However, if they are not aliases, then the value of $x.a$ will remain unchanged.

Uday Reddy has pointed out an equivalent way to present component substitution, which dates at least as far back as some early work of Burstall [4]. In this view, a component name a is considered as a global array, and $V.a$ as $a[V]$; Morris' treatment simply resolves some of the `if` clauses in the treatment of arrays as early as possible. The above definition is then written as

$$x.a[17/y.a] \ \mapsto\ x.(a \oplus y \mapsto 17)$$

This is read as follows: the global array a is as it was except that the y component is now assigned to 17. This component-as-array trick is theoretically simpler than Morris' because it removes the need to define a new notion of substitution: it arises by composing two known ideas, ordinary substitution and the Hoare logic treatment of arrays. The idea to treat components as arrays, semantically, is commonplace; the point, however, is that it fits particularly well with program logic, where it has a pleasant simplifying effect. The main value of component substitution, be it in the Morris or components-as-arrays style, is that it works on the same syntactic level as the programming language, thus obviating the need to carry around a store parameter in assertions to resolve aliasing. It deserves to be better known.

Comparing to a more recent work, this component-as-array view can also be thought of as a specialization of work of

Leino [12], where a pointer type is viewed as a global array. The specialization is also a simplification, made possible by a semantic choice, where records are objects in the heap that can only be accessed by pointers.

Although component substitution handles pointer aliasing properly (under some programming language restrictions, discussed in the conclusion), it does not, in and of itself, help with a more significant problem: the *local reasoning problem*. In fact, it actually draws attention to the problem. The mechanism works "globally", where assignment to $V.a$ induces substitutions for all occurrences of a in an assertion. Conceptually, this is at odds with computational intuition regarding pointer assignment: On a basic operational level, an assignment to $V.a$ changes only a single location, not a large array.

A similar criticism can be lodged of the standard Hoare logic treatment of arrays, but in pointer programs the problem is more acute. It is common to work with a number of data structures at one time in the heap, where the records used share component names, but where the data structures are distinct (or distinct in certain logical senses). For example, linked structures might be used to keep track of the files in several directories, and adding a new file to one does not require a global update of all the directories. The problem is to transfer this programming intuition to program logic, by allowing reasoning about programs that act on one area of storage, without having that reasoning affect assertions that describe other areas of storage.

This local reasoning problem has not been solved as of yet, but there have recently been two promising developments. One is in the work of Reynolds [18], extending early work of Burstall. The other is in the work of Bornat [3]. Both approaches rely on what Bornat calls "spatial separation." In the Reynolds approach, this is accomplished using a spatial form of conjunction, which splits the heap into components; in [10] it is shown how this enables certain frame axioms, which describe invariants of the heap, to be inferred automatically. In the Bornat approach, which builds on work of Morris, the idea is to use a stylized form of inductive definition for predicates that talk about data structures; the definitions are arranged so as to reveal the portion of storage relevant to the data structure in question, which gives rise to what Bornat calls spatial separation properties. This enables substitutions for components in formulas to be resolved early, without descending into a definition, when that formula is describing a data structure that is distinct from the one being updated.

With this as background, we now describe the contents of the present paper, which studies the semantics of the component-substitution approach. Our first purpose is modest: to give a model that shows its soundness with respect to a standard operational semantics that works with l-values. Though modest, a semantic analysis is called for, especially since substitution has been a notoriously delicate area. Also, a semantics makes the assumptions underlying the approach especially clear, and provides a basis for extension or modification of it. So, our first task will be to formulate an appropriate Substitution Lemma, and use it to validate an axiom for assignment statements. In doing this, we use a logic of partial functions to account for expressions (such as $x.a$ where x denotes nil) that cause run-time errors. The semantics of Hoare triples we adopt is one that adheres to the slogan *verified programs don't go wrong*.

Neither Bornat nor Morris treats definedness in detail; in the case of assignment this is perhaps reasonable, but we find that close attention to definedness (or run-time errors) actually helps in the consideration of storage allocation. Our second purpose, then, is to use the semantics to help devise axioms for operations $\texttt{new}(x)$ and $\texttt{dispose}(E)$, for allocating and disposing of memory; these have been almost as problematic as assignment. For `new` the difficulty is to find a way of talking about "unallocated locations" which do not have any contents in the precondition, but which will have contents in the postcondition. For `dispose` the problem is dual: in the precondition a location will have contents, but we must arrange that in the postcondition any attempt to dereference a disposed location is "undefined." For both we use component substitution in a crucial way, and in the case of disposal there is a surprising interaction between undefinedness and substitution.

Our axioms for `new` and `dispose` are both simpler than, and theoretically superior to, most versions that have appeared in the literature. Often, additional predicates are introduced to keep track of which locations have been allocated, but in our treatment these are not necessary. The reason is that pointers already necessitate a consideration of a form of partiality, or run-time error, in expressions, for example due to attempts to dereference nil. We find that, for our operational semantics, the same assertional device used to characterize definedness can also be used to account for allocation, thus reducing the number of logical concepts involved. Justification for this view is given by results showing that the axioms for `new` and `dispose` express weakest preconditions for our operational semantics, showing a sense in which they are correct and complete.

In early work, Oppen and Cook [17] gave an axiom for `new` (they did not consider `dispose`), which they proved expressed the weakest precondition. The formulation here is considerably simpler, owing mainly to our use of component substitution. When he introduced component substitution Morris did not consider allocation or deallocation, and neither does Bornat. Leino [12] did consider both, in the context of his array-like treatment of pointers, but he did not show that his axioms express weakest preconditions; indeed, they appear to be incomplete as stated.

In a series of papers, de Boer has investigated an approach to `new` (but not `dispose`) which works by defining a substitution form $E[\texttt{new}/u]$ (see [6]). Intuitively, this behaves as if each occurrence of u in E denotes the same new location. The definition is subtle, and the relation to the work here is not clear. One crucial point of difference is that de Boer treats quantifiers in a varying domain style (where we use a fixed domain of locations); again this is subtle, and it might be useful to study the approach for a pared-down language, stripped of object-oriented features.

Our final purpose will be to validate Bornat's stylized use of inductive definitions in one specific case: linked lists. We state non-interference and substitution lemmas, which show the soundness of the interaction between definitions and component substitution, and a spatial separation property. The treatment of lists is typical, and can be generalized to many other data structures; however, this work is somewhat preliminary. In particular, a systematic theory explaining definitions that reveal storage remains to be worked out.

2. EXPRESSIONS AND COMPONENT SUBSTITUTION

The language we consider distinguishes between the *stack*, which holds the values of local variables, and the *heap*, which contains data created and destroyed dynamically. The stack can be extended by declarations of local variables, and modified by assignments. We assume that the heap can contain only one type of data structure: records. For simplicity, records have a fixed number of components, indexed by a fixed finite set $Tags = \{a_1, ..., a_n\}$. These tags may include hd and tl for implementing lists, and generally will be indicated with the letters a, b.

The syntax of expressions is given by the following grammar:

$$\begin{array}{rcl} E & ::= & x \\ & | & N \\ & | & BE \\ & | & \texttt{if } BE \texttt{ then } E \texttt{ else } E \\ & | & \texttt{nil} \\ & | & E.\kappa \\ \\ N & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \cdots \\ \\ BE & ::= & \texttt{true} \mid \texttt{false} \\ & | & E == E \\ & | & \cdots \\ \\ \kappa & ::= & a \\ & | & \kappa \oplus V \mapsto E \end{array}$$

An expression can be a variable, natural number, boolean expression, conditional, null pointer, or access to a record component. A boolean expression can be a boolean constant or a comparison of expressions. The syntax $E.\kappa$ means "accessing the κ-component pointed to by E". κ can be an atomic tag, such as hd, in which case $E.hd$ is the expression obtained by dereferencing the pointer and accessing the record component. In a C-style syntax this would be written $(*E).a$. κ can also be a component extension of the form $\kappa \oplus V \mapsto E$. Under the component-as-array notation, this says that the specific index V has been recently updated to E.

When dealing with pointers one has to face the problem of aliasing: two pointers may point to the same element. The difficulty arises because modifying the element pointed to by one will affect what the other points to. For example in our language if two variables x and y point to the same record, then an assignment $x.a := 7$ will change also the value of $y.a$. We say then that $x.a$ and $y.a$ have the same l-value: essentially they are components of the same record in the heap. It is important to note that distinct variables x and y in our language cannot be aliased, because their values depend only on the stack, and not on the heap. In particular the l-values of x and y cannot be modified, because their l-values are essentially "x" and "y".

We identify a class of expressions that have l-values, and which appear on the left side of an assignment; they are defined by the following grammar:

$$\begin{array}{rcl} V & ::= & x \\ & | & V.a \end{array}$$

The substitution $E'[E/V]$ is defined for arbitrary l-value expressions V. If V is a variable x, then there cannot be aliasing and we can proceed with the usual substitution. But if V is of the form $V'.a$, it may be the case that $V'.a$ is an alias of a component of E', hence this possibility must be considered. Substitution is defined by a simultaneous induction on E' and V. Some of the inductive cases have subcases, indicated by side conditions; it is easy to see that there is no overlap between cases. In the following we use $\mapsto$ to mean "reduces to".

$$\begin{array}{rcl} x[E/V] & \mapsto & x \quad (\text{when } x \neq V) \\ x[E/x] & \mapsto & E \\ \texttt{k}[E/V] & \mapsto & \texttt{k} \quad (\texttt{k} \text{ a constant}) \\ (E_1 == E_2)[E/V] & \mapsto & E_1[E/V] == E_2[E/V] \\ (\texttt{if } BE \texttt{ then } E_1 \texttt{ else } E_2)[E/V] & \mapsto & \\ \multicolumn{3}{r}{\texttt{if } BE[E/V] \texttt{ then } E_1[E/V] \texttt{ else } E_2[E/V]} \\ (E'.\kappa)[E/V] & \mapsto & (E'[E/V]).(\kappa[E/V]) \\ a[E/x] & \mapsto & a \\ a[E/V.b] & \mapsto & a \quad (\text{when } a \neq b) \\ a[E/V.a] & \mapsto & a \oplus V \mapsto E \\ (\kappa \oplus V' \mapsto E')[E/V] & \mapsto & \kappa[E/V] \oplus V'[E/V] \mapsto E'[E/V] \end{array}$$

This is essentially Morris's definition in component-as-array form. We now describe a semantics that enables us to validate the definition.

We use the notation $X \to Y$ to indicate the set of total functions from X to Y, and $X \rightharpoonup_{fin} Y$ for the set of partial functions with finite domain. The domain of a partial function f is written $dom(f)$. We write $(f \mid c \mapsto d)$ to indicate the function defined like f but mapping c to d. Our semantics makes use of the following sets.

$$\begin{array}{rcl} \texttt{Nat} & \triangleq & \{0, 1, ..., 17, ...\} \\ \texttt{Bool} & \triangleq & \{false, true\} \\ \texttt{Tags} & \triangleq & \{hd, tl, a, b, ...\} \\ \texttt{Variables} & \triangleq & \{x, y, ...\} \\ \texttt{Locations} & \triangleq & \{\ell, ...\} \\ \texttt{Values} & \triangleq & \texttt{Nat} + \texttt{Bool} + \texttt{Locations} + \{nil\} \\ \texttt{Stacks} & \triangleq & \texttt{Variables} \rightharpoonup_{fin} \texttt{Values} \\ \texttt{Heaps} & \triangleq & \texttt{Locations} \rightharpoonup_{fin} (\texttt{Tags} \to \texttt{Values}) \\ \texttt{States} & \triangleq & \texttt{Stacks} \times \texttt{Heaps} \end{array}$$

A stack s is a partial function from variables to values, which can be numbers, booleans, locations or nil. The domain of s

simply represents the set of variables in the scope, and can be extended only with the declaration of a new variable. We assume that the variables in an expression E are always in the domain of the stack. A heap is a partial function from locations to records, and a record is a tuple of values indexed by `Tags`. The domain of a heap h is the set of locations that have been allocated so far, and this set is finite. We will use the notation $(h \mid \ell.a \mapsto v)$ as a short for $(h \mid \ell \mapsto (h(\ell) \mid a \mapsto v))$.

EXAMPLE 1. *Suppose that we have only two tags hd and tl, with* $dom(s) = \{x\}$, *and* $dom(h) = \{\ell_1, \ell_2, \ell_3\}$, *and* s, h *are as follows:*

$$\begin{array}{rcl} s(x) & = & \ell_1 \\ \\ h(\ell_1) & = & [hd : 4,\ tl : \ell_2] \\ h(\ell_2) & = & [hd : 7,\ tl : \ell_3] \\ h(\ell_3) & = & [hd : 5,\ tl : nil] \end{array}$$

The variable x holds a pointer to a record, which we can think of as the first element of the list $[4, 7, 5]$ *terminated by* nil.

In general the evaluation of an expression will not produce a defined result. For example, if x denotes `nil` then evaluation of $x.a$ would cause a run-time error. We represent these errors as "undefined," where the semantics of an expression is a partial function:

$$[\![E]\!] : \texttt{States} \rightharpoonup \texttt{Values}$$

The function $[\![E]\!]$ is defined by cases. We use the notation $\uparrow$ to indicate when a partial function is undefined.

$$\begin{array}{rcll} [\![x]\!]s,h & \triangleq & s(x) & \\ \\ [\![\mathtt{k}]\!]s,h & \triangleq & k & k \in \texttt{Nat} \cup \texttt{Bool} \\ \\ [\![E_1 == E_2]\!]s,h & \triangleq & \uparrow & \text{if } [\![E_1]\!]s,h = \uparrow \\ & & \uparrow & \text{if } [\![E_2]\!]s,h = \uparrow \\ & & true & \text{if } [\![E_1]\!]s,h = [\![E_2]\!]s,h \\ & & false & \text{if } [\![E_1]\!]s,h \neq [\![E_2]\!]s,h \end{array}$$

$$\begin{array}{rll} \multicolumn{3}{l}{[\![\texttt{if } BE \texttt{ then } E_1 \texttt{ else } E_2]\!]s,h \quad \triangleq} \\ & \uparrow & \text{if } [\![BE]\!]s,h \notin \texttt{Bool} \\ & [\![E_1]\!]s,h & \text{if } [\![BE]\!]s,h = true \\ & [\![E_2]\!]s,h & \text{if } [\![BE]\!]s,h = false \end{array}$$

$$\begin{array}{rcll} [\![\texttt{nil}]\!]s,h & \triangleq & nil & \\ \\ [\![E.a]\!]s,h & \triangleq & \uparrow & \text{if } [\![E]\!]s,h \notin dom(h) \\ & & h(\ell)(a) & \text{if } [\![E]\!]s,h = \ell \in dom(h) \end{array}$$

$$\begin{array}{rll} \multicolumn{3}{l}{[\![E.(\kappa \oplus V \mapsto E')]\!]s,h \quad \triangleq} \\ & \uparrow & \text{if } [\![E]\!]s,h \notin \texttt{Locations} \\ & \uparrow & \text{if } [\![V]\!]s,h \notin \texttt{Locations} \\ & [\![E']\!]s,h & \text{if } [\![E]\!]s,h = [\![V]\!]s,h \\ & [\![E.\kappa]\!]s,h & \text{if } [\![E]\!]s,h \neq [\![V]\!]s,h \end{array}$$

The semantics of a variable x is always defined: we assume that the domain of s is fixed, and the variables occurring in expressions range over that domain. There are three ways in which $[\![E.a]\!]s,h$ can be undefined:

- $[\![E]\!]s,h$ is undefined;
- $[\![E]\!]s,h$ is defined but is not a location;
- $[\![E]\!]s,h$ is a location which is not in the domain of the heap.

The substitution $E'[E/V]$ produces an expression which, semantically, speaks only of r-values. But, for this substitution to be operationally compatible with assignment, it is important to relate it to operations on l-values in the semantics. This is the purpose of the Substitution Lemma.

LEMMA 2. **Substitution.**

1. $[\![E'[E/x]]\!]s,h = [\![E']\!](s \mid x \mapsto v),h$; *when* $[\![E]\!]s,h = v \in \texttt{Values}$.
2. $[\![E'[E/V.a]]\!]s,h = [\![E']\!]s,(h \mid \ell.a \mapsto v)$; *when* $[\![E]\!]s,h = v \in \texttt{Values}$ *and* $[\![V]\!]s,h = \ell \in dom(h)$.

The separation of the Substitution Lemma into two parts reflects the existence of two kinds of substitution: substitution for a variable x does not involve aliasing and is equivalent to modifying the stack, while substitution for a component $V.a$ must deal with aliasing and affects the heap.

3. LOGIC OF ASSERTIONS

In this section we introduce a logic to reason about expressions of the language. The only subtlety is the treatment of undefinedness: because the semantics of expressions is partial, we need to use a logic of partial functions.

There is something of a controversy over the proper way to treat undefined expressions in program-proving formalisms [2, 19, 11]. The simple-minded approach is to use the usual two-valued logic. In this approach, quantifiers range over only defined values, and definedness assumptions are needed for substitution and quantifier laws. A more sophisticated approach is to consider a logic in which assertions can themselves be undefined. We choose the former here for theoretical simplicity; we want to study rules for pointers with a minimum of distraction. This choice does not constitute an ideological commitment, and we believe that the main points could be made in a many-valued logic approach as well.

The logic is divided into two parts: atomic predicates, which express assertions about expressions, and formulas of first order logic built on atomic predicates. The two basic issues related to expressions are partiality and absence of types. This leads to a logic for partial objects with predicates to test the type of objects. The syntax is given by the following grammar:

$$\begin{array}{rcl} A & ::= & E = E \\ & \mid & loc?(E) \\ & \mid & num?(E) \\ & \mid & bool?(E) \\ & \mid & \ldots \\ \\ P & ::= & true \\ & \mid & A \\ & \mid & P \wedge P \\ & \mid & \neg P \\ & \mid & \forall x.\, P \end{array}$$

Formulas P are those of first-order logic, with atoms A ranging over equality (two expressions are defined and equal), and predicates to test if an expression yields a location, a number or a boolean.

We give an interpretation of formulas in the model.

$$s, h \models P$$

means that formula P holds of stack s and heap h. The undefinedness of expressions is dealt with in the interpretation of atomic predicates A.

$$s, h \models E = E' \quad \overset{\Delta}{\iff} \quad [\![E]\!]s, h = [\![E']\!]s, h = v \in \texttt{Values}$$

$$s, h \models loc?(E) \quad \overset{\Delta}{\iff} \quad [\![E]\!]s, h \in \texttt{Locations}$$

$$s, h \models num?(E) \quad \overset{\Delta}{\iff} \quad [\![E]\!]s, h \in \texttt{Nat}$$

$$s, h \models bool?(E) \quad \overset{\Delta}{\iff} \quad [\![E]\!]s, h \in \texttt{Bool}$$

It is important to notice that, although the semantics of expressions is partial, the interpretation of an atomic predicate in a given stack and heap is either true or false. In particular $E = E'$ is true if both expressions are defined and yield the same values, false otherwise. We use the symbols $=$ and $==$ to distinguish $E = E'$, which is an atomic predicate, from $E == E'$, which is an expression, and as such can be undefined if either side is.

The interpretation of connectives and quantifiers is classical.

$$s, h \models true \qquad \text{always}$$

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

$$s, h \models \neg P \quad \overset{\Delta}{\iff} \quad s, h \not\models P$$

$$s, h \models \forall x. P \quad \overset{\Delta}{\iff} \quad \forall d \in \texttt{Values}.\, (s \mid x \mapsto d), h \models P$$

(We use $\rightarrow$ and $\vee$ for the induced implication and disjunction.)

The separation between atomic predicates and general formulas is important because it allows us to deal with specific features of the language with the first, and reason in a classical standard style with the second.

Definedness will play a central role, and we introduce a derived predicate for it:

$$def?(E) \overset{\Delta}{\equiv} E = E$$

Clearly $s, h \models \neg\, def?(E)$ if and only if $[\![E]\!]s, h = \uparrow$. We also use the notation

$$E_1 =_e E_2 \overset{\Delta}{\equiv} (E_1 = E_2) \vee (\neg\, def?(E_1) \wedge \neg\, def?(E_2))$$

Building on this, we consider some logical rules that will be useful in proving formulas involving our predicates. These rules are not meant to be complete in any sense, but are given to highlight what has to be added to the usual rules for first order logic.

$$\frac{E = E'}{def?(E) \ \wedge \ def?(E')} \qquad \frac{def?(E.a)}{loc?(E)}$$

$$\frac{p(E)}{def?(E) \wedge \neg\, p'(E)} \ p, p' \in \{loc?, num?, bool?\};\ p \neq p'$$

$$\frac{}{\neg\, p(nil)} \ p \in \{loc?, num?, bool?\} \qquad \frac{}{def?(x)}$$

$$\frac{}{def?(k)} \ k \text{ a constant} \qquad \frac{\forall x. P \quad def?(E)}{P[E/x]}$$

$$\frac{\neg\, loc?(E) \vee \neg\, loc?(V)}{\neg\, def?(E.(\kappa \oplus V \mapsto E'))} \qquad \frac{(E = V) \wedge loc?(E)}{E.(\kappa \oplus V \mapsto E') =_e E'}$$

$$\frac{\neg\,(E = V) \wedge loc?(E) \wedge loc?(V)}{E.(\kappa \oplus V \mapsto E') =_e E.\kappa}$$

The first two rules say that equality implies definedness, and that we can only dereference locations. Note that the reverse implication does not hold: it may be the case that E denotes a location but $E.a$ is undefined, when that location is undefined in the heap. The third rule says that locations, numbers and booleans are defined, and are mutually exclusive. The next two axioms say that nil is a special constant distinct from locations, numbers and booleans, and that variables are always defined (from the assumption that all the variables we use belong to the domain of the stack). The last three rules are about object-component extension: the first one says that an expression is undefined if either the object or the component is not a location; the second and third define the behaviour of component updates.

Consider the abbreviation $\Omega \overset{\Delta}{\equiv} nil.a$. It is easy to derive $\neg\, def?(\Omega)$ since $\neg\, loc?(nil)$ is an axiom and implies $\neg\, def?(nil.a)$ using the second rule. We will use Ω as canonical undefined expression.

To understand the need for $def?(E)$ in the $\forall$-elimination rule, consider the formula $\forall x. def?(x)$. Clearly it holds in every state, but of course we cannot derive $def?(\Omega)$.

3.1 Hoare Triples

In order to interpret Hoare triples we presume an operational semantics $\rightsquigarrow$ which specifies a transition between *configurations*, where a configuration K is either a triple (C, s, h), where C is a command, s a stack and h a heap; or a pair (s, h). We will refer to the latter as a *final configuration*. We will write $\rightsquigarrow^*$ for the reflexive and transitive closure of $\rightsquigarrow$.

We make the following definitions.

- A configuration K is **stuck** when K is a triple and there is no K' such that $C, s, h \rightsquigarrow K'$. In particular, a final configuration is never stuck.
- A configuration K is **safe** if, whenever $K \rightsquigarrow^* K'$, K' is not stuck.

Intuitively, a configuration C, s, h is stuck when C cannot proceed from state s, h, for instance because it tries to dereference *nil*. A safe configuration is one that will never get stuck. Notice that in general a computation starting from a safe configuration does not necessarily terminate, since it may be an infinite loop which never reaches a stuck configuration. This allows us to distinguish non termination from the occurrence of run-time errors.

With these presumed we can now set out the semantics of triples.

> We say that $\{P\}\ C\ \{Q\}$ is true just when
>
> If $s, h \models P$ then C, s, h is safe, and if $C, s, h \rightsquigarrow^* s', h'$ then $s', h' \models Q$.

Notice that this is a partial correctness interpretation; the total correctness variant is evident. Notice also that the interpretation adheres to the slogan *verified programs don't go wrong*. For example, $\{x = \texttt{nil}\}x.a := 17\{\texttt{true}\}$ is not a true triple.

In order to judge the completeness of axioms for the statements we use the notion of weakest precondition.

WEAKEST PRECONDITION.

$$wp(C, R) \triangleq \{(s,h) \mid C, s, h \text{ is safe, and } C, s, h \rightsquigarrow^* s', h' \text{ implies } s', h' \models R\}$$

[Strictly speaking, we should refer to this as the weakest *liberal* precondition, but for our main results involving assignment, `new` and `dispose` the two notions *wlp* and *wp* are equivalent.]

We recall briefly syntax, semantics and Hoare axioms for the basic constructs for commands. Specific commands of our language will be treated in the following section. The syntax is given by the following grammar:

$$\begin{array}{rcl} C & ::= & \texttt{skip} \\ & \mid & C_1\,;\,C_2 \\ & \mid & \texttt{while}\ BE\ \texttt{do}\ C\ \texttt{od} \\ & \mid & \ldots \end{array}$$

As usual `skip` is the do-nothing command, $C_1\,;\,C_2$ is the command that executes first C_1 and then C_2, and `while` is a loop executing repeatedly command C until condition E becomes false. The semantics is standard:

$$\frac{}{\texttt{skip}, s, h \rightsquigarrow s, h}$$

$$\frac{C_1 \rightsquigarrow C_1', s', h'}{(C_1; C_2) \rightsquigarrow (C_1'; C_2), s', h'} \qquad \frac{C_1 \rightsquigarrow s', h'}{(C_1; C_2) \rightsquigarrow C_2, s', h'}$$

$$\frac{[\![BE]\!]s, h = false}{\texttt{while}\ BE\ \texttt{do}\ C\ \texttt{od}, s, h \rightsquigarrow s, h}$$

$$\frac{[\![BE]\!]s, h = true \quad (C; \texttt{while}\ BE\ \texttt{do}\ C\ \texttt{od}), s, h \rightsquigarrow K}{\texttt{while}\ BE\ \texttt{do}\ C\ \texttt{od}, s, h \rightsquigarrow K}$$

We conclude this section with a standard set of axioms for partial correctness.

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{\{P\}\ C_1\ \{Q\} \quad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1; C_2\ \{R\}}$$

$$\frac{P \rightarrow def?(BE) \qquad \{P \wedge (BE = true)\}\ C\ \{P\}}{\{P\}\ \texttt{while}\ BE\ \texttt{do}\ C\ \texttt{od}\ \{P \wedge (BE = false)\}}$$

$$\frac{P \rightarrow P' \quad \{P'\}\ C\ \{Q'\} \quad Q' \rightarrow Q}{\{P\}\ C\ \{Q\}}$$

The first three rules are those for the three constructs for commands, and the fourth, the *consequence rule*, is needed to strengthen the precondition or weaken the postcondition of the triple.

(To extend consideration of completeness to loops would require consideration of expressiveness [5]; this, however, is outside the scope of our concerns here.)

4. ASSIGNMENT

The first command we consider is assignment

$$V := E.$$

To define the relation $\rightsquigarrow$ we need to distinguish between variable assignment $x := E$ and component assignment $V.a := E$. In the first case we update the stack and in the second the heap.

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

$$\frac{[\![E]\!]s, h = v \quad [\![V]\!]s, h = \ell \in dom(h)}{V.a := E, s, h \rightsquigarrow s, (h \mid \ell.a \mapsto v)}$$

The weakest precondition in both cases follows from this description:

> $(s, h) \in wp(x := E, R)$ iff
> $[\![E]\!]s, h = v$ and $(s \mid x \mapsto v), h \models R$
>
> $(s, h) \in wp(V.a := E, R)$ iff $[\![E]\!]s, h = v$ and
> $[\![V]\!]s, h = \ell \in dom(h)$ and $s, (h \mid \ell.a \mapsto v) \models R$

An assignment statement is "stuck" when the antecedent of the rule cannot be satisfied. For example, $x.a := 17$ is stuck when the domain of h is empty.

In order to express the weakest precondition with a formula, we have to extend the definition of substitution to formulas.

$$\begin{array}{rcl} true[E'/V] & \mapsto & true \\ (E_1 = E_2)[E'/V] & \mapsto & E_1[E'/V] = E_2[E'/V] \\ p(E)[E'/V] & \mapsto & p(E[E'/V]) \\ & & \text{where } p \in \{loc?, num?, bool?\} \\ (P \wedge Q)[E/V] & \mapsto & P[E/V] \wedge Q[E/V] \\ (\neg P)[E/V] & \mapsto & \neg(P[E/V]) \\ (\forall x.\, P)[E/V] & \mapsto & \forall x.\, (P[E/V]) \quad x \notin var(E, V) \end{array}$$

In the $\forall$ case we are using a freshness assumption; we could alternatively use the standard device of α-renaming.

LEMMA 3. *Substitution II*

1. $s, h \models P[E/x]$ *iff* $(s \mid x \mapsto v), h \models P$; *when* $[\![E]\!]s, h = v \in \mathtt{Values}$.
2. $s, h \models P[E/V.a]$ *iff* $s, (h \mid \ell.a \mapsto v) \models P$; *when* $[\![E]\!]s, h = v \in \mathtt{Values}$ *and* $[\![V]\!]s, h = \ell \in dom(h)$.

We now have a natural way to give Hoare-style axioms.

AXIOMS FOR ASSIGNMENT

$$\begin{array}{l} \{def?(E) \wedge R[E/x]\} \\ x := E \\ \{R\} \end{array}$$

$$\begin{array}{l} \{def?(V.a) \wedge def?(E) \wedge R[E/V.a]\} \\ V.a := E \\ \{R\} \end{array}$$

In the axiom for $V.a := E$ the precondition asks that $V.a$ be defined. Logically, this is a statement about an r-value but, in our semantics, it implies that the corresponding l-value exists.

PROPOSITION 4. *The axioms for variable and component assignment express the weakest preconditions.*

EXAMPLE 5. *Consider the program $x := nil;\ x.a := 42$. Intuitively in every state it will try to dereference nil and will crash. We can ask on which states it will terminate, i.e. what is the weakest precondition with respect to true:*

$$\begin{array}{l} wp(x := nil;\ x.a := 42,\ true) = \\ wp(x := nil,\ def?(x.a) \wedge def?(42) \wedge true[42/x.a]) = \\ wp(x := nil,\ def?(x.a)) = \\ def?(nil) \wedge def?(x.a)[nil/x] = def?(nil.a) \end{array}$$

Note that we have freely eliminated $def?(42)$ and $def?(nil)$ since they are axioms. The resulting formula $def?(nil.a)$ is always false, and matches the intuition that there is no state on which the program above can terminate.

5. NEW AND DISPOSE

In this section we consider two commands for allocating and de-allocating memory. The syntax is the following:

$$\mathtt{new}(x) \quad | \quad \mathtt{dispose}(E)$$

$\mathtt{new}(x)$ finds an unused location ℓ of the heap and assigns a new record to it, filling the components with any values. $\mathtt{dispose}(E)$ will, on the other hand, de-allocate the record pointed to by E. The domain of the heap represents the locations that are currently allocated, and `new` will non-deterministically choose a location outside the domain of the heap and will non-deterministically initialise the corresponding record. The effect of `dispose` will be to simply remove the location from the domain of the heap.

We recall the syntactic sugar used so far and extend it:

- $def?(E)$ is short for $E = E$.
- Ω is short for $nil.a$.
- $def?(E.*)$ is short for $def?(E.a_1) \wedge \cdots \wedge def?(E.a_n)$, with $a_1, ..., a_n$ a repetition-free enumeration of $Tags$.
- $\forall x_*$ is short for $\forall x_{a_1}, ..., \forall x_{a_n}$.
- $[E_*/x.*]$ is short for $[E_{a_1}/x.a_1, ..., E_{a_n}/x.a_n]$.
- $v_* \in X$ is short for $v_{a_1} \in X, \ldots, v_{a_n} \in X$.
- $(h \mid l.* \mapsto v_*)$ is short for $(h \mid l \mapsto [v_{a_1}, \ldots, v_{a_n}])$.

Here is the semantics of `new` .

$$\frac{\ell \notin dom(h) \quad \ell \in \mathtt{Locations} \quad v_* \in \mathtt{Values}}{\mathtt{new}(x), s, h \rightsquigarrow (s \mid x \mapsto \ell), (h \mid \ell.* \mapsto v_*)}$$

$$\begin{array}{l} (s, h) \in wp(\mathtt{new}(x), R) \quad \text{iff} \\ \quad \forall \ell \in \mathtt{Locations} - dom(h).\ \forall v_* \in \mathtt{Values}. \\ \qquad (s \mid x \mapsto \ell), (h \mid \ell.* \mapsto v_*) \models R \end{array}$$

$\mathtt{new}(x)$ is never stuck.

To formulate an axiom for `new`, the first point to consider is that we want to say that the location selected is not in the domain of the current heap. This can be mimicked in the logic by asking that the r-value of an expression $x.a$ be undefined.

But then we face a conundrum: if $x.a$ is undefined how can we speak of the values it might have in the postcondition? For example, in the postcondition we might want to say $def?(x.a)$, which would be used in the precondition in a subsequent assignment to $x.a$, but it seems hard to talk about this if $x.a$ is undefined. (Also, if we were to consider a version of `new` which initialized the components or took an initialization as an argument, then we would want to speak of specific values, as in $x.a = y$.)

The way to resolve this conundrum is to use component substitution: We use a substitution $R[x_*/v.*]$ in the precondition, which implicitly makes a statement about the components in the postcondition R: this neatly solves the problem of having $x.a$ be undefined in the precondition, while still being able to speak about values it takes in the postcondition.

AXIOM FOR NEW

$$\begin{array}{l} \{\forall x. \forall x_*. \neg\, def?(x.a) \wedge loc?(x) \rightarrow R[x_*/x.*]\} \\ \mathtt{new}(x) \\ \{R\} \end{array}$$

Formally, the above is actually a family of axioms parametric in a. The choice of a particular component a does not change the truth of $def?(x.a)$ since either all the components are defined or none is. So, throughout this section a may be considered to be some fixed component name (though it doesn't matter which one).

REMARK 6 (WARNING ON DEFINED NOTATION). *It is tempting to introduce a definition $A(x) \triangleq def?(x.a)$, and to formulate the axiom using this defined notation instead of $def?(x.a)$. However, $A(x)$ has different substitution* *behaviour than $def?(x.a)$ because a is not free in it: a component substitution for a has no effect on $A(x)$. Because of this, the reformulation of the* `new` *axiom using $A(x)$ is unsound; the notation $A(x)$ hides an access to the a component, hence aliases to x would not be captured by component substitution.*

There are two points to be made here. First, the $def?(x.a)$ notation is speaking directly about an r-value: the contents of $x.a$. Of course, it implies that the location denoted by x is in the domain of the current heap, but by phrasing this in terms of r-values and not l-values we are able to stay in a substitution-friendly setup, which is crucial for Hoare logic. Second, we stress that, when introducing definitions, one must be very careful to ensure that they behave correctly with respect to substitution; and this goes for component as well as ordinary substitution.

Notice that our use of the abbreviation $\Omega \stackrel{\Delta}{\equiv} nil.a$ is justified, by the fact that any substitution into $nil.a$ results in an expression $nil.\kappa$, which must be undefined.

An initializing version of `new` can be axiomatized by substituting specific values, (say, 0) for $x.*$, instead of quantifying over all possible values.

The further subtlety here is that the substitution $R[x_*/x.*]$ is a *simultaneous component substitution*, which substitutes for all the components of records. The definition is similar to that of normal substitution:

$$\begin{array}{rcl} x[E_*/V.*] & \mapsto & x \\ \mathtt{k}[E_*/V.*] & \mapsto & \mathtt{k} \quad (\mathtt{k}\text{ a constant}) \\ (E_1 == E_2)[E_*/V.*] & \mapsto & E_1[E_*/V.*] == E_2[E_*/V.*] \end{array}$$

$$\begin{array}{l} (\mathtt{if}\ BE\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2)[E_*/V.*] \quad \mapsto \\ \quad \mathtt{if}\ BE[E_*/V.*]\ \mathtt{then}\ E_1[E_*/V.*]\ \mathtt{else}\ E_2[E_*/V.*] \end{array}$$

$$E'.a[E_*/V.*] \quad \mapsto \quad E'[E_*/V.*].(a \oplus V \mapsto E_a)$$

$$\begin{array}{l} (E''.(\kappa' \oplus V' \mapsto E'))[E_*/V.*] \quad \mapsto \\ \quad E''[E_*/V.*].(\kappa'[E_*/V.*] \oplus V'[E_*/V.*] \mapsto E'[E_*/V.*]) \end{array}$$

The extension to formulas is straightforward.

LEMMA 7. *Simultaneous Component Substitution.*

1. $[\![E'[E_*/V.*]]\!]s,h = [\![E']\!]s,(h \mid \ell.* \mapsto v_*)$; *if* $[\![V]\!]s,h = \ell \in$ `Locations` *and* $[\![E_*]\!]s,h = v_* \in$ `Values`.
2. $s,h \models P[E_*/V.*]$ *iff* $s,(h \mid \ell.* \mapsto v_*) \models P$; *when* $[\![V]\!]s,h = \ell \in$ `Locations` *and* $[\![E_*]\!]s,h = v_* \in$ `Values`.

PROPOSITION 8. *The axiom for* `new` *expresses the weakest precondition.*

EXAMPLE 9. *As a first example of the use of* `new`*, we want to show that genuinely new locations are generated. To express this, we may ask what are the conditions that guarantee $x \neq y$ after the command* `new`*(x). We can use the logic to derive the weakest precondition.*

$$\begin{array}{l} wp(\mathtt{new}(x),\ x \neq y) = \\ \forall x.\forall x_*.\, \neg\, def?(x.a) \wedge loc?(x) \rightarrow (x \neq y)[x_*/x.*] = \\ \forall x.\forall x_*.\, \neg\, def?(x.a) \wedge loc?(x) \rightarrow (x \neq y) = \\ \forall x.\, \neg\, def?(x.a) \wedge loc?(x) \rightarrow (x \neq y) = \\ \forall x.\, (x = y) \rightarrow (def?(x.a) \vee \neg\, loc?(x)) = \\ def?(y.a) \vee \neg\, loc?(y) \end{array}$$

The weakest precondition says that if y denotes a location ℓ, we can be sure that `new`*(x) will generate a different location only if ℓ is defined in the heap. This is because ℓ may have been disposed, and in that case we want to be able to reuse it.*

We now do a similar analysis for `dispose`:

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

$$\begin{array}{l} (s,h) \in wp(\mathtt{dispose}(E),R) \quad \text{iff} \\ [\![E]\!]s,h = \ell \in dom(h) \text{ and } s,h-\ell \models R \end{array}$$

Here, $h - \ell$ is the heap like h except that it is undefined on ℓ. `dispose`(E) is stuck when E is not a location, or when it is a location but is not in the domain of h.

To represent the weakest precondition we should now say that the location disposed *is* in the domain of the current heap. This aspect is in a sense inverse to `new`, and we simply have to require $def?(E.a)$. A more subtle problem is that we want to guarantee that subsequent attempts to dereference the disposed location will be undefined. This can be modelled as a component substitution once again, where we substitute "undefined" for each of the components of the disposed record. Note that this does not replace the location itself by undefined, but only the (r-values of the) components.

AXIOM FOR DISPOSE

$$\begin{array}{l} \{def?(E.a) \wedge R[\Omega/E.*]\} \\ \mathtt{dispose}(E) \\ \{R\} \end{array}$$

This substitution of an undefined expression is, at first sight, worrying. For, in partial function logic rules for substitution usually require that the substituted expression is defined. However, the definition of component substitution, as opposed to ordinary substitution, is entirely happy with substituting undefined. This, for us, came as a surprise, and is what allows the treatment of `dispose` to work.

As an example, consider substituting Ω for $x.a$ in $y.a$:

$$\begin{array}{rcl} y.a[\Omega/x.a] & \mapsto & y[\Omega/x.a].(a \oplus x \mapsto \Omega) \\ & = & y.(a \oplus x \mapsto \Omega) \end{array}$$

This does not result in undefined if x and y are unequal. And further, the Ω is entirely bypassed when substituting for different components:

$$x.b[\Omega/x.a] \ \mapsto \ x.b.$$

These examples give the flavour of how component substitution interacts with undefinedness: the following lemma sums up the properties needed for analysis of the `dispose` axiom.

LEMMA 10. *Component Substitution and Undefinedness.*

1. $[\![E[\Omega/V.*]]\!]s,h = [\![E]\!]s,(h-\ell)$; *when* $[\![V]\!]s,h = \ell \in dom(h)$.

2. $s,h \models P[\Omega/V.*]$ *iff* $s,(h-\ell) \models P$; *when* $[\![V]\!]s,h = \ell \in dom(h)$.

PROPOSITION 11. *The axiom for* `dispose` *expresses the weakest precondition.*

EXAMPLE 12. *We have seen that* `dispose` *has the effect of making a location undefined in the heap. But what happens if we try, by mistake, to dispose the same location twice? We can derive it!*

$$\begin{array}{l}
wp(\mathtt{dispose}(x);\mathtt{dispose}(x),\ true) = \\
wp(\mathtt{dispose}(x),\ def?(x.a) \wedge true[\Omega/x.*]) = \\
wp(\mathtt{dispose}(x),\ def?(x.a)) = \\
def?(x.a) \wedge def?(x.a)[\Omega/x.*] = \\
def?(x.a) \wedge def?(x[\Omega/x.*].(a \oplus x \mapsto \Omega)) = \\
def?(x.a) \wedge def?(x.(a \oplus x \mapsto \Omega)) = \\
def?(x.a) \wedge def?(\Omega)
\end{array}$$

Since $\neg\, def?(\Omega)$ *is true, the weakest precondition is* $false$*. This says that, whatever the initial state, dereferencing a location twice is not safe.*

6. INDUCTIVE DEFINITIONS THAT REVEAL STORAGE

When reasoning about a data structure in memory, one usually wants to make high-level statements and proofs using these definitions and some lemmas, without having to perform a new inductive proof for every use of a data structure. In particular, inductive definitions are useful for reasoning about lists, trees, and the like.

In the approach of Morris and Bornat there are a few surprises in the way that definitions are treated, owing to their interaction with component substitution. We explain this by going through a few putative definitions of properties involving lists, finally arriving at an example of the stylized form of definition that Morris and Bornat rely on. We then show two lemmas, a Substitution Lemma which verifies correctness, and a non-interference lemma which, to some extent, enables the global nature of component substitution to be controlled.

We confine our attention to a single kind of data structure here: linked lists. Much of the material can apply to other data structures, but we do not yet have a general theory to account for inductive data structures.

6.1 Definitions for Lists

We first define a predicate which says "E points to a non-circular linked list in storage," which means that we get to nil by following a finite number of tail links. We are not concerned with the contents of other components just now, only that the list exists. (From now on we will drop the qualification "non-circular.")

First attempt

$$\begin{array}{rcl}
list(0,E) & \triangleq & E = nil \\
list(n+1,E) & \triangleq & loc?(E) \wedge list(n, E.tl)
\end{array}$$

We will not be completely formal about inductive definitions, but the intent should be clear. (In particular, we regard E and n as schematic metavariables, which are substituted for on the meta-level.)

Using this definition, we can say that E points to a list by quantifying over numbers: $\exists n.\, list(n,E)$. This definition is fine semantically, as it says just what we want. But it is incorrect for the program logic we have been using, for two reasons.

The first problem is that the definition interacts badly with component substitution. We cannot substitute just into the parameters, since an unrolling of the list predicate may hide aliases. Consider the following example.

$$list(1,p)[7/q.tl]$$

If we apply the component substitution now, we obtain $list(1,p)$, but unrolling the definition first, we have

$$(loc?(p) \wedge list?(0, p.tl))[7/q.tl]$$

which expands to

$$(loc?(p) \wedge list?(0, (p.(tl \oplus q \mapsto 7))).$$

This second formula is equivalent to the first only if $p \neq q$. The reason for this unsoundness is that the expansion of the $list$ predicates reveals an access to the tl component. The definition needs to allow this substitution to be captured.

There are a couple of ways around this. One is to introduce a notion of substitutions which block on the predicate $list(n,E)$ and then use induction to unroll the list predicate n times, to prove properties about substitution. The semantics of the predicate in this case is given, essentially, by an infinite unrolling of the definition. This can be made to work but, due to the need to define new syntactic categories and substitution mechanisms, is perhaps technically inelegant. Another solution, which we use here, is to make the component name tl a parameter of the definition, and to substitute for that. The definition of the list predicate is then both stack-variable and object-component closed.

Second attempt

$$\begin{array}{rcl}
list(0,E,\kappa) & \triangleq & E = nil \\
list(n+1,E,\kappa) & \triangleq & loc?(E) \wedge list(n, E.\kappa, \kappa)
\end{array}$$

We can instantiate κ with tl to obtain the desired predicate. Now a substitution

$$list(1,p,tl)[7/q.tl]$$

results in

$$list(1,p,(tl \oplus q \mapsto 7)).$$

We are now left with some work to resolve the updated component, but the problem with substitution above has been successfully dealt with. (We will verify this claim below.)

Although making the component a parameter allows us to treat substitution properly, there is a further problem: the definition hides all of the cells in the data structure (even though it does not hide component names). To see what we mean, suppose that we are specifying a program that disposes any node in a list pointed to by q that appears in a list pointed to by p. For correctness of the evident algorithm, we might require that the lists associated with p and q do not overlap in the store, as deleting from q's list would alter p. How, then, can we state the precondition?

$$\exists n.\, list(n, p, tl) \ \wedge\ \exists m.\, list(m, q, tl) \ \wedge\ \textit{there is no overlap.}$$

This idea of non-overlapping is typical. For example, if we were to specify a program that disposes of all nodes in a list p whose head components are equal to those in a list q, then we might want a similar non-overlapping precondition: otherwise the dispose of an element in p might destroy the "listness" of q, and lead to a run-time error.

To describe the appropriate precondition we might try to introduce another inductively-defined predicate, which says that two lists do not overlap. But if we were to adopt this solution, then we would have to introduce similar predicates for every pair of data structure definitions we were working with: for non-overlap between two trees, between a tree and a list, and so on.

Morris and Bornat instead use inductive definitions that explicitly reveal the cells in a data structure. To describe this for $list$, we use a metavariable ls to range over sequences of locations.

Third attempt

$$\begin{array}{rcl} list([], E, \kappa) & \triangleq & E = nil \\ list(\ell : ls, E, \kappa) & \triangleq & E = \ell \wedge list(ls, E.\kappa, \kappa) \end{array}$$

Then, the desired precondition for the append program is

$$\exists ls, ls'.\, list(ls, p, tl) \ \wedge\ list(ls', q, tl) \ \wedge\ ls \cap ls' = \emptyset$$

where the predicate $ls \cap ls' = \emptyset$ states that sequences ls and ls' do not have any elements in common.

This third definition is not the first one that comes to mind, so it is worth pausing to recount how we have arrived at it. The most immediate definition, which said "p points to a list," was logically correct, but interacted badly with component substitution. This led us to add a component parameter. But then the second attempt did not allow us to state, directly, when two lists overlap in the store, and we altered the definition to make the cells encountered in the linked list a parameter of the defined predicate.

This is perhaps the most interesting feature of the Morris/Bornat approach: they use inductive definitions that describe logical properties of a data structure, as one would expect, but that also *reveal the portion of storage relevant to the data structure*. Note that this is not the same as looking at all reachable elements, as there is pruning: only those reachable cells relevant to the property of interest are revealed. Further, the case of lists is not special here. Bornat proposes that for any sort of data structure – doubly-linked lists, trees, etc. – there will be an associated formula that describes the cells involved in it, and the suggestion is to arrange logical definitions to be used in proofs in a stylized manner, that purposely reveals these cells.

There is one other important point about these stylized definitions: they allow the global nature of component substitution to be short-circuited, in some instances. For example, if we have a post-condition that says that two lists do not overlap, and we know that a cell E is in the first list,

$$list(ls, p, tl) \ \wedge\ list(ls', q, tl) \ \wedge\ ls \cap ls' = \emptyset \ \wedge\ E \in ls$$

then when generating the substitution $[E'/E.tl]$ associated with an assignment $E.tl := E'$ there is no need to substitute into the tl component of $list(ls', q, tl)$. Of course, we still must do some work in p's list to resolve the component substitution in $list(ls, p, (tl \oplus E \mapsto E'))$, but our work is effectively localized to the list pointed to by p.

6.2 Lemmas

Now we state two lemmas about the $list$ predicate, to validate some of the previous discussion.

Substitution for the list predicate is as expected:

$$list(ls, E, \kappa)[E'/V.a] \quad \mapsto \quad list(ls, E[E'/V.a], \kappa[E'/V.a])$$

though as E will, in practice, be a pointer variable, the real effect of substitution is on the component parameter. The following lemma shows that substituting for the parameters of $list$ is sound.

LEMMA 13. *Let P be the formula obtained expanding the definition of $list(ls, E, \kappa)$, V an l-value expression and E' an expression. Then the following are equivalent:*

1. $P[E'/V]$
2. $list(ls, E[E'/V], \kappa[E'/V])$.

The usual induction principle for lists is encoded in the definition of the $list$ predicate. While this principle is useful for proving general properties about lists, in many typical cases — for instance, in the example proofs that we will consider in the following section — we can use spatial properties generated by the revealed cells to simplify the proofs. The sense in which the revealed cells are correct is stated in the following non-interference lemma.

LEMMA 14. *If $(loc?(V) \wedge V \notin ls)$ holds, $V \notin [l_1, ..., l_n]$ being short for $\bigwedge_{i=1}^{n} V \neq l_i$, then the following are equivalent for any expression E and component expression κ:*

1. $list(ls, E, \kappa)$
2. $list(ls, E, \kappa \oplus V \mapsto E)$

A variant of this non-interference lemma would, it appears, have to be proven for each inductive definition one uses, because it is easy to make a definition where the result is false.

7. A WORKED EXAMPLE

In this section we consider a complete example: a program which disposes a list. A detailed proof is given, using the material of the previous section. Other examples, involving the use of *new*, can be developed in a similar way.

We introduce some new notation: $E \neq E'$ is short for $def?(E) \wedge def?(E') \wedge \neg(E = E')$; $[l_1, ..., l_n]$ is the obvious syntactic sugar for a sequence of locations of length n, and $E \notin [l_1, ..., l_n]$ is short for $\bigwedge_{i=1}^{n} E \neq l_i$. The following lemma shows that lists are repetition-free.

LEMMA 15. *If* $list([l_1, ..., l_n], E, \kappa)$ *holds and* $i \neq j$ *then* $l_i \neq l_j$.

Consider the following definition of the program, together with the specification in the form of a Hoare triple:

$\{list([l_1, \ldots, l_n], p, tl)\}$
while $p \neq$ **nil do**
  $q := p$;
  $p := p.tl$;
  dispose(q);
od
$\{\bigwedge_{k=1}^{n} \neg def?(l_i.tl)\}$

The precondition says that at the beginning p points to a list, given by the sequence $[l_1, \ldots, l_n]$. The postcondition says that at the end all the locations have been disposed.

The invariant INV of the loop is

$$\exists i. (list([l_{i+1}, \ldots, l_n], p, tl) \wedge \bigwedge_{k=1}^{i} \neg def?(l_k.tl))$$

where $\bigwedge$ is just a defined operator for a sequence of conjunctions. We will do the proof backwards: many of the intermediate steps are easy consequences of the definitions.

$$\{ \; def?(p.tl) \wedge \exists j. list([l_{j+1}, \ldots, l_n], p.tl, tl \oplus p \mapsto \Omega) \wedge \bigwedge_{k=1}^{j} \neg def?(l_k.(tl \oplus p \mapsto \Omega)) \; \}$$

$\mathtt{q} := \mathtt{p}$;

$$\{ \; def?(p.tl) \wedge def?(q.tl) \wedge \exists j. list([l_{j+1}, \ldots, l_n], p.tl, tl \oplus q \mapsto \Omega) \wedge \bigwedge_{k=1}^{j} \neg def?(l_k.(tl \oplus q \mapsto \Omega)) \; \}$$

$\mathtt{p} := \mathtt{p.tl}$;

$$\{ \; def?(q.tl) \wedge \exists j. list([l_{j+1}, \ldots, l_n], p, tl \oplus q \mapsto \Omega) \wedge \bigwedge_{k=1}^{j} \neg def?(l_k.(tl \oplus q \mapsto \Omega)) \; \}$$

$$\{ \; def?(q.tl) \wedge (\exists j. list([l_{j+1}, \ldots, l_n], p, tl) \wedge \bigwedge_{k=1}^{j} \neg def?(l_k.tl))[\Omega/q.tl] \; \}$$

dispose$(\mathtt{q})$;

$$\{ \; \exists j. list([l_{j+1}, \ldots, l_n], p, tl) \wedge \bigwedge_{k=1}^{j} \neg def?(l_k.tl) \; \}$$

We have to prove that $INV \wedge p \neq nil$ implies the calculated weakest precondition (the top line in the above outline). We can use $p \neq nil$ to unroll the list once, and derive the following.

$$\{ \; p \neq nil \wedge \exists i. list([l_{i+1}, \ldots, l_n], p, tl) \wedge \bigwedge_{k=1}^{i} \neg def?(l_k.tl) \; \}$$

by Lemma 15

$$\{ \; p \neq nil \wedge \exists i. p = l_{i+1} \wedge l_{i+1} \notin [l_{i+2}, \ldots, l_n] \; list([l_{i+2}, \ldots, l_n], p.tl, tl) \wedge \bigwedge_{k=1}^{i} \neg def?(l_k.tl) \; \}$$

by Lemma 14

$$\{ \; p \neq nil \wedge \exists i. p = l_{i+1} \wedge l_{i+1} \notin [l_{i+2}, \ldots, l_n] \; list([l_{i+2}, \ldots, l_n], p.tl, tl \oplus p \mapsto \Omega) \wedge \bigwedge_{k=1}^{i} \neg def?(l_k.tl) \; \}$$

Here, by listing the assertions in sequence we are saying that the first implies the second, and the second implies the third. Notice how we have used Lemma 15 in the first step, when unrolling the list once, to derive the spatial separation $l_{i+1} \notin [l_{i+2}, \ldots, l_n]$, and used it for applying Lemma 14 in the second step to resolve the component substitution into a *list* assertion, without having to unroll the definition further.

Now we need to show that the third formula implies the weakest precondition calculated for the body. To do this we can take $j = i + 1$. Clearly $def?(p.tl)$ holds since it appears inside the *list* predicate. We are left with the proof of $\bigwedge_{k=1}^{i+1} \neg def?(l_k.(tl \oplus p \mapsto \Omega))$. When $k < i$, $l_k.(tl \oplus p \mapsto \Omega)$ reduces to $l_k.tl$ since $p \neq l_k$, and we know $\neg def?(l_k.tl)$. When $k = i + 1$, we have to show $\neg def?(\Omega)$ since $p = l_{i+1}$, and it always holds since it is an axiom. This completes the proof of preservation of the invariant.

To complete the proof of the program, we first note that the precondition implies INV: it is enough to take $i = 0$. Secondly, $INV \wedge p = nil$ implies $i = n$, hence the postcondition.

So far we have seen how to prove that correct programs meet the specification. The other side of this coin is showing that incorrect programs do not meet the specification — and highlighting where the error occurs. For instance, a typical error for the list disposal program would be to dispose the current, rather than old, value of p.

while $p \neq$ **nil do**
  dispose(p);
  $p := p.tl$;
od

Backwards reasoning in this case will help in finding the mistake:

$$\{ \quad def?(p.tl) \wedge def?(\Omega) \wedge \exists j.\, list([l_{j+1},\ldots,l_n], p.tl, tl \oplus p \mapsto \Omega) \wedge \bigwedge_{k=1}^{j} \neg\, def?(l_k.(tl \oplus p \mapsto \Omega)) \quad \}$$

$$\{ \quad def?(p.tl) \wedge def?(p.(tl \oplus p \mapsto \Omega)) \wedge \exists j.\, list([l_{j+1},\ldots,l_n], p.tl, tl \oplus p \mapsto \Omega) \wedge \bigwedge_{k=1}^{j} \neg\, def?(l_k.(tl \oplus p \mapsto \Omega)) \quad \}$$

$$\{ \quad def?(p.tl) \wedge (def?(p.tl) \wedge \exists j.\, list([l_{j+1},\ldots,l_n], p.tl, tl) \wedge \bigwedge_{k=1}^{j} \neg\, def?(l_k.tl))[\Omega/p.tl] \quad \}$$

```
dispose(p);
```

$$\{ \quad def?(p.tl) \wedge \exists j.\, list([l_{j+1},\ldots,l_n], p.tl, tl) \wedge \bigwedge_{k=1}^{j} \neg\, def?(l_k.tl) \quad \}$$

```
p := p.tl;
```

$$\{ \quad \exists j.\, list([l_{j+1},\ldots,l_n], p, tl) \wedge \bigwedge_{k=1}^{j} \neg\, def?(l_k.tl) \quad \}$$

The presence of $def?(\Omega)$ indicates that the weakest precondition is false, hence there is a mistake in the program at this point.

8. CONCLUSION

There are two main limitations to the component substitution approach to reasoning about pointers. The first concerns the assumptions required for the approach to work. At present these include:

- distinct records in the heap must not overlap;
- the l-value of a component of a record cannot be extracted and manipulated as an r-value; and
- there is no parameter aliasing, which is to say aliasing between stack variables, which may hold pointers to records.

The main question is whether, or the extent to which, these restrictions are *necessary*.

The second limitation concerns the global nature of the mechanism, where each component is essentially treated as a large array. As discussed in Section 6, some progress has been made on this issue. Reynolds has also made good progress [18] (also, [10]), using a completely different approach: a spatial form of conjunction is used, instead of the method of revealing storage in inductive definitions. Both cases are, however, first steps, and it appears that there is much more to be learnt about local reasoning.

ACKNOWLEDGEMENTS. We are grateful to Richard Bornat, especially for his contributions to the development of the axioms for `dispose` and `new`. This research was supported by EPSRC grant GR/L54578.

9. REFERENCES

[1] AMERICA, P. AND DE BOER, F. Reasoning about dyamically evolving process structures. *Formal Aspects of Computing 6* (1994), 269–316.

[2] BARRINGER, H., CHENG, J. H., AND JONES, C. B. A logic covering undefinedness in program proofs. *Acta Informatica 21* (1984), 251–269.

[3] BORNAT, R. Proving pointer programs in Hoare logic. To appear in *Mathematics of Program Construction*, 2000.

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

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

[6] DE BOER, F. A WP calculus for OO. In *Proceedings of FOSSACS'99* (1999).

[7] 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.

[8] HOARE, C. A. R. AND WIRTH, N. An axiomatic definition of the programming language pascal. *Acta Informatica 2* (1973), 335–355.

[9] 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.

[10] ISHTIAQ, S. AND O'HEARN, P. BI as an assertion language for mutable data structures. Manuscript, March 2000.

[11] JONES, C. Partial functions and logics: a warning. *Inf. Proc. Letters 54* (1995), 65–67.

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

[13] LONDON, R. E. A. Proof rules for the programming language Euclid. *Acta Informatica 10* (1995), 1–26.

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

[15] 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.

[16] NELSON, G. Verifying reachability invariants of linked structures. In *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages* (1983), pp. 38–47.

[17] 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.

[18] REYNOLDS, J. Intuitionistic reasoning about shared mutable data structure. To appear in the *Proceedings of the Symposium in Celebration of the Work of C.A.R. Hoare*, 2000.

[19] TENNENT, R. D. A note on undefined expression values in programming logics. *Inf. Proc. Letters 24* (1987), 331–333.