A trace model for pointers and objects

C.A.R. Hoare[1] and He Jifeng[2]

[1] *Oxford University Computing Laboratory,*
Wolfson Building, Parks Road, Oxford, OX1 3QD
tony.hoare@comlab.ox.ac.uk

[2] *United Nations University*
International Institute for Software Technology
P.O. Box 3058, Macau
jifeng@iist.unu.edu

Abstract: Object-oriented programs [Dahl, Goldberg, Meyer] are notoriously prone to the following kinds of error, which could lead to increasingly severe problems in the presence of tasking

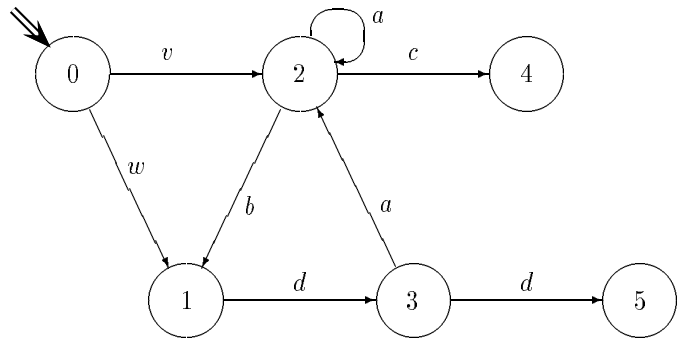
1. Following a null pointer
2. Deletion of an accessible object
3. Failure to delete an inaccessible object
4. Interference due to equality of pointers
5. Inhibition of optimisation due to fear of (4)

Type disciplines and object classes are a great help in avoiding these errors. Stronger protection may be obtainable with the help of assertions, particularly invariants, which are intended to be true before and after each call of a method that updates the structure of the heap. This note introduces a mathematical model and language for the formulation of assertions about objects and pointers, and suggests that a graphical calculus [Curtis, Lowe] may help in reasoning about program correctness. It deals with both garbage-collected heaps and the other kind. The theory is based on a trace model of graphs, using ideas from process algebra; and our development seeks to exploit this analogy as a unifying principle.

1 Introduction: the graph model

Figure 1.0 shows a rooted edge-labelled graph. Its **nodes** are represented by circles and its **edges** by arrows from one node to another. The letter drawn next to each arrow is its **label**. The set of allowed labels is called the **alphabet** of the graph. A double-shafted arrow singles out a particular node as the **root** of the graph.

Figure 1.0 (Rooted edge-labelled graph)



□

Such a graph can be defined less graphically as a tuple

$$G \quad = \quad (A_G, N_G, E_G, \mathrm{root}_G),$$

where	A_G	is the alphabet of labels
	N_G	is the set of nodes
	E_G	is the set of edges with their labels, i.e., a subset of $N_G \times A_G \times N_G$
	root_G	is the node selected as the root

We use variables G, G' to stand for graphs, l, m, n to stand for nodes, x, y, z to stand for general labels and s, t, u to stand for sequences of labels (traces). We write $l \xrightarrow[G]{x} m$ to mean $(l, x, m) \in E_G$. Where only one graph is in question, we omit the subscript G. The smallest graph (called 0_A) with given alphabet A consists only of the root node, with no edges, i.e. $(A, \{\mathrm{root}\}, \{\}, \mathrm{root})$. Another small but interesting graph is $1_A =_{df} \{(A, \{\mathrm{root}\}, (\{\mathrm{root}\} \times A \times \{\mathrm{root}\}), \mathrm{root})$.

Example 1.1 (Tuple from graph) The graph of Figure 1.0 is coded as the mathematical structure defined by the following equations

$$\begin{array}{lcl} A & = & \{v, w, a, b, c, d\} \\ N & = & \{0, 1, 2, 3, 4, 5\} \\ E & = & \{(0, v, 2), (0, w, 1), (2, a, 2), (2, b, 1), (1, d, 3), (2, c, 4), (3, a, 2), (3, d, 5)\} \\ \mathrm{root} & = & 0 \end{array}$$

□

The awkward feature of this encoding is the arbitrary selection of the first six natural numbers to serve as the nodes. Any other six distinct values would have done just as well. We are only interested in properties of graphs that are preserved by one-one transformations (isomorphisms) of the node-set. The use of isomorphism in place of mathematical equality is an inconvenience. We aim to avoid it by constructing a canonical representation for the nodes of a graph. For this, we will have to restrict the theory to graphs satisfying certain healthiness conditions.

Rooted edge-labelled graphs are useful in the study of many branches of computing science, of which data diagrams and heap storage are relevant to object-oriented programming.

Example 1.2 (Automata theory) A graph defines the behaviour of an automaton. The nodes stand for states, with the root as the initial state. The labels stand for events, and the presence in E of an edge $l \xrightarrow{x} m$ means that event x happens as the automaton passes from state l to state m. □

Example 1.3 (Data diagrams) In a data diagram, a node stands for a set of values, e.g., a type or a class of objects. The labels stand for functions, and the presence of an edge $l \xrightarrow{x} m$ means that x maps values of type l to results of type m. The root is somewhat artificial: the labels on arrows leading from the root can be regarded as the names of the types that they point to. □

Example 1.4 (Control flow) In a control flow graph, the nodes represent basic blocks, i.e sections of program code with no internal label. The edge $l \xrightarrow{x} m$ represents the presence in block l of a jump to a label x which is placed at the beginning of block m. The root is the main block of the program. The same analysis applies when the jumps are procedure calls and the nodes are procedure bodies. □

Example 1.5 (Heap storage) A graph can describe the instantaneous content of the entire heap at a particular point in the execution of an object-oriented program. The nodes stand for the objects, and the labels are the names for the attributes. An edge $l \xrightarrow{x} m$ means that m is the value of the x-attribute of the object l. □

When used to model objects and heaps, the labelled graph is both simple and general, in that it allows more complex concerns to be treated separately. For example,

1. Simple values (e.g., like 5, which is printable) can be treated in the usual way as sinks of the graph, i.e. as nodes from which no pointer can ever point. A method local to an object can be similarly represented as a value of one of its attributes.

2. The labels on pointers from the unique root represent the directly accessible program variables. There is no restriction on pointing from the heap into declared program workspace; such pointers are often used in legacy code for cyclic representations of chains, even if their use is deprecated or forbidden in higher level languages.

3. Absence of a pointer from an object in which space has been allocated for it is often represented by filling the space with a **nil** value. The model allows this; another representation permitted by our model is to introduce a special **nil** object, with special properties, e.g. all arrows from it lead back to itself.

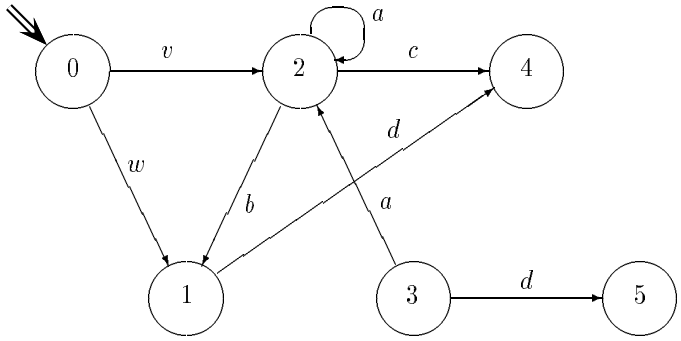
4. The model describes the statics and dynamics of object storage, and is quite independent of the class declarations and inheritance structure of the source language in which a program has been written. In fact, the relationship between the run-time heap and a data diagram is a special case of an invariant assertion, that remains true throughout the execution of the program. The invariant is elegantly formalised with the aid of graph homomorphisms, as described in Definition 1.10.

The main operation for updating the value of the heap is written $l \rightarrowtail a := m$. It causes the a-labelled arrow whose tail rests at node l to point to node m, instead of what it pointed to before. The operation changes only the edges of the graph, leaving the nodes, the alphabet, and the root unchanged.

Definition 1.6 (Pointer swing)

$$(l \rightarrowtail a := m) \quad :=_{df} \quad (E := (E - \{l\} \times \{a\} \times N) \cup \{(l, a, m)\}), \text{ where } l, m \in N$$ □

Example 1.7 (Pointer swing) After execution of $1 \rightarrow d := 4$, the graph of Figure 1.0 would appear, as follows



□

Further operations are needed for deleting an edge and for creating a new node. Node creation introduces an arbitrary new object into the node-set and swings a pointer to point to it. Deletion of a node is more problematic, and will be treated later.

Definition 1.8 (Edge deletion, Node creation)

$$(l \rightarrow a \;:=\; \textbf{nil}) \;=_{df}\; E := E - \{l\} \times \{a\} \times N$$
$$(l \rightarrow a \;:=\; \textbf{new}) \;=_{df}\; N := N + \{m\};\; l \rightarrow a := m,\ \text{where } m \;\tilde{\in}\; N \qquad \square$$

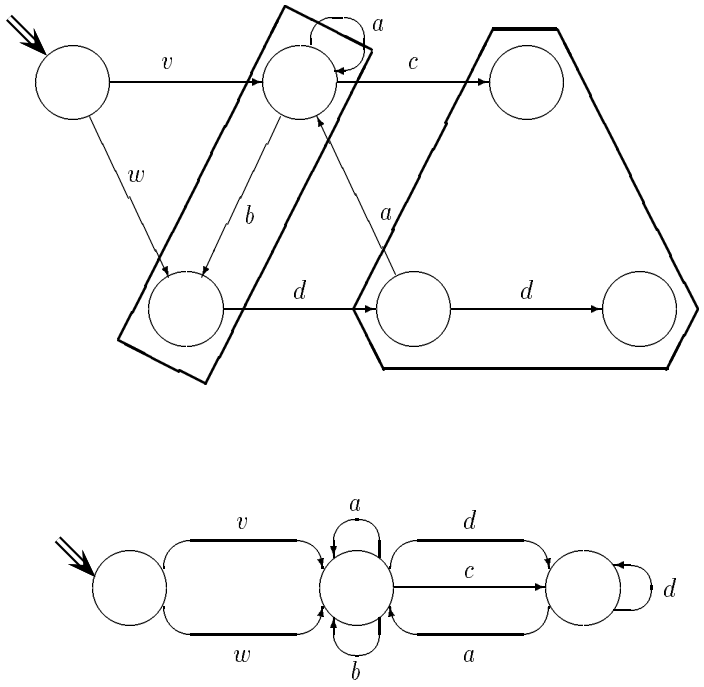
There are two problems with the above definitions of operations on the heap. The first is that an object-oriented program has no means of directly naming the objects l and m. These references have to be made indirectly by quoting the sequence of labels on a path which leads from the root to the desired object. Thus the assignment in Example 1.7 might have been written

$$w \rightarrow d \;:=\; w \rightarrow d \rightarrow a \rightarrow c$$

The second problem is that, after the assignment, two of the nodes (3 and 5) have become inaccessible: the program will never again be able to refer to those nodes by any path. In a garbage-collected heap, such nodes are subject to disappearance at any time. In a non-collected heap they could represent a storage leak. Our trace model of object-orientation will solve all these problems, with the help of a canonical representation of the graph.

When a graph is used as a data diagram it specifies the classes of object to which each variable and attribute is allowed to point. A compiler can therefore allocate to each object only just enough store to hold all its permitted attributes. The compiler will also check all the operations of a program to ensure that all the rules have been observed. As a result, at all times during execution it is possible to ascribe each object in the heap to a node in the data diagram representing the object class to which it belongs. This can be pictured by drawing a polygon around all nodes belonging to the same particular class. Each polygon is then contracted to a single node, dragging the heads and tails of the arrows with it. The result will be a data diagram, which will match the intended structure of class declarations.

Figure 1.9 (Object classes)



□

The informal description of the transformation of a heap structure to a class diagram is formalised in the mathematical definition of a homomorphism. This is a function from the nodes of one graph to the nodes of another that preserves the root and the labels on the edges.

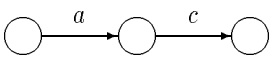
Definition 1.10 (Homomorphism) Let $G = (A, N, E, \text{root})$ and $G' = (A', N', E', \text{root}')$. Let f be a total function from N to N'. The triple $f : G \to G'$ is called a homomorphism if $A \subseteq A'$, and for all x in A

1. $f(\text{root}) = \text{root}'$
2. $m \xrightarrow{x}_{G} n$ implies $f(m) \xrightarrow{x}_{G'} f(n)$, for all x in A. □

Examples 1.11 (Homomorphisms) From every graph G with alphabet A, there is just one homomorphism to 1_A; from 0_A to G, there are as many homomorphisms as nodes in G. □

Homomorphisms can also be used to define the relationship between a subclass and its parent class in a class hierarchy [Cardelli, Cook]. For this, we will later introduce a method of reducing the alphabet of labels to match that of the target of the homomorphism. Multiple inheritance is simply modelled by asserting the existence of more than one homomorphism from the heap to several different data diagrams. Different languages enforce differing conventions and rules, to ensure that the invariance of such assertions at run time is checkable by compiler. Our theory is claimed to be sufficiently expressive to describe all such checkable rules in any language. It can also formulate much more general assertions, whose truth cannot be checked at compile time, but only at run time or by proof.

Another important role for a homomorphism is to select from a large graph a smaller subgraph for detailed consideration. The shape of the subgraph is specified by the source of the homomorphism, and the target specifies which particular subgraph of that shape is selected. For example, consider the graph



A subgraph of this shape occurs just twice in Figure 1.0; there is only one injective homomorphism from it into the Figure, and one that is non-injective. This kind of subgraph homomorphism has to be redefined to allow for absence of a root.

The remaining role of the homomorphism is to define the concept of an isomorphism of graphs, and so specify what it means for two graphs with different node sets to be essentially the same.

Definition 1.12 (Isomorphism) Let $f : G \to G'$ be a homomorphism. This is said to be an isomorphism if f is invertible, and $f^{-1} : G' \to G$ is also a homomorphism. G and G' are isomorphic if there is an isomorphism from one to the other. □

This rather indirect definition represents the very simple intuitive idea of laying one graph on top of another, and ensuring that it has nodes and edges and labels in all the same places. Like congruent triangles in geometry, they are just two copies of the same graph!

2 The trace model

The problem of inaccessible objects is the same as that of inaccessible states in automata theory; and the solution that we adopt is the same: calculate the language of traces that are generated by the graph. A trace of an automaton is a sequence of consecutive events that can occur during its evolution. A trace can be read from the graph by starting at node l and following a path of consecutive edges leading from each node to the next, along a path of directed edges. The trace is extracted as the sequence of labels encountered on the path up to its last node m. The existence of such a trace s is denoted $l \xrightarrow{s} m$. A formal definition uses recursion on the length of the trace.

Definition 2.0 (Traces)

$$
\begin{array}{lll}
l \xrightarrow{<>} m & \textbf{iff} & l = m \\
l \xrightarrow{<a>} m & \textbf{iff} & (l, a, m) \in E \\
l \xrightarrow{s \,\widehat{}\, t} m & \textbf{iff} & \exists n \bullet l \xrightarrow{s} n \ \wedge \ n \xrightarrow{t} m \\
l \xrightarrow{*} m & =_{df} & \{s \mid l \xrightarrow{s} m\} \\
\text{traces } (l) & =_{df} & \text{root} \xrightarrow{*} l
\end{array}
$$
□

Example 2.1 (Figure 1.0) From the graph of Figure 1.0 the sets of traces of each of the six nodes are given by the following six regular expressions

$$
\begin{array}{lcl}
n_0 & = & \varepsilon \\
n_1 & = & w + n_2\, b \\
n_2 & = & v + n_2\, a + n_3\, a \\
n_3 & = & n_1 d \\
n_4 & = & n_2 c \\
n_5 & = & n_3 d
\end{array}
$$
□

In the canonical trace model of a graph, each node l is represented by the set traces(l), containing all traces on paths to it from the root. The set of nodes is therefore a family N of sets of traces ($N \subseteq \mathbf{P} A^*$). The labelled edges and the root of the graph can be defined in terms of this family.

Definition 2.2 (Canonical representation)

$$
\begin{array}{llll}
\text{Let} & G & = & (A, N, E, r) \\
 & \widehat{N} & =_{df} & \{\text{traces}(n) \mid n \in N\} \\
 & \widehat{E} & =_{df} & \{l \xrightarrow{x} m \mid l, m \in \widehat{N} \wedge \ l \,\widehat{}\, < x > \subseteq m\} \\
 & \widehat{r} & =_{df} & \text{the unique } n \text{ in } \widehat{N} \text{ containing } <>. \\
 & \widehat{G} & =_{df} & (A, \widehat{N}, \widehat{E}, \widehat{r})
\end{array}
$$
□

Theorem 2.3

For all l, m, $n \in \widehat{N}$ and $X \subseteq A^*$

(1) $(l \frown X) \subseteq m$ **iff** $X \subseteq (l \xrightarrow[\widehat{G}]{*} m)$

(2) $(l \xrightarrow[\widehat{G}]{*} m) \frown (m \xrightarrow[\widehat{G}]{*} n) \subseteq (l \xrightarrow[\widehat{G}]{*} n)$

(3) $(\widehat{r} \xrightarrow[\widehat{G}]{*} m) = m$

Proof: (1) From the fact that for all $s \in A^*$

$$(l \frown s) \subseteq m \quad \textbf{iff} \quad l \xrightarrow[\widehat{G}]{s} m$$

(2) From the associativity of the catenation operator and the Galois connection (1)

(3)

		$X \subseteq LHS$
$\equiv$	{(1)}	$\widehat{r} \frown X \subseteq m$
$\Rightarrow$	{$<> \in \widehat{r}$}	$X \subseteq RHS$
$\equiv$	{let $m = \text{traces}(n)$}	$\forall s \in X \bullet (\text{root} \xrightarrow[G]{s} n)$
$\Rightarrow$	$\{\forall t \in \widehat{r} \bullet (\text{root} \xrightarrow[G]{t} \text{root})\}$	$\forall t \in \widehat{r}, s \in X \bullet (\text{root} \xrightarrow[G]{\widehat{t\ s}} n)$
$\equiv$	{def of traces}	$(\widehat{r} \frown X) \subseteq \text{traces}(n) = m$
$\equiv$	{(1)}	$X \subseteq LHS$

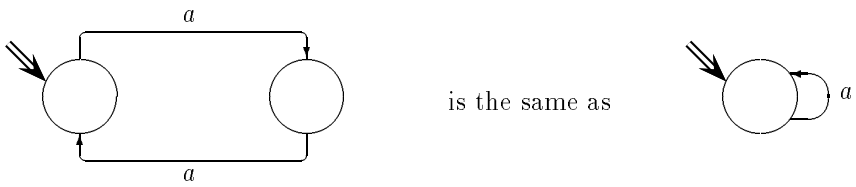
□

In the theory of deterministic automata, the language generated by the automaton is just the union of the set of traces of all its states

$$\text{language } (G) = \bigcup \{\text{traces}(l) \mid l \in N_G\}$$

The great advantage of this is that an inaccessible state has no traces at all, and so makes no contribution to the language. Two automata are therefore decreed to be identical if they have the same language.

Example 2.4 (Identical automata)



because in both cases the language is $\{a\}^*$ □

For automata, the purpose of this identification is to allow automatic minimisation of the number of states needed to generate or recognise a specified language. But in object-oriented programming, such identification of objects would be wholly inappropriate. The reason is that the pointer swinging operation (not considered by automata theory) distinguishes graphs which automata theory says should be the same.

Example 2.5 (after swing) After the assignment $a \rightarrow a := \mathbf{nil}$, the two graphs of example 2.4 now look like

□

Even in automata theory, these two graphs are distinct. For this reason, we cannot model a heap simple as a set of traces, and we have to go up one level in complexity to model it as a **set** of sets of traces, as shown in the definition of $\widehat{N}$.

Nevertheless, there are many interesting analogies between our trace model the process algebra of non-deterministic automata. For example, as in CSP [Hoare], the entire set of valid traces is prefix-closed.

Theorem 2.6 (Prefix closure) $\bigcup \widehat{N}$ is non-empty; and if it contains $s \frown t$, it also contains s. □

An important property of the $\widehat{\ }$ operator, transforming a graph to its canonical representation, is that it leaves unchanged an argument that is already canonical.

Theorem 2.7 (Idempotence)

$$\widehat{\widehat{G}} = \widehat{G}$$

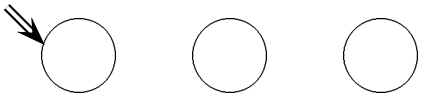
Proof

$$\begin{aligned} & \text{traces}(\text{traces}(n)) \\ = \; & \{\text{def of traces}\} \\ & \text{traces}(\text{root}) \xrightarrow[\widehat{G}]{*} (\text{traces}(n)) \\ = \; & \{\text{Theorem 2.3(3)}\} \\ & \text{traces}(n) \end{aligned}$$

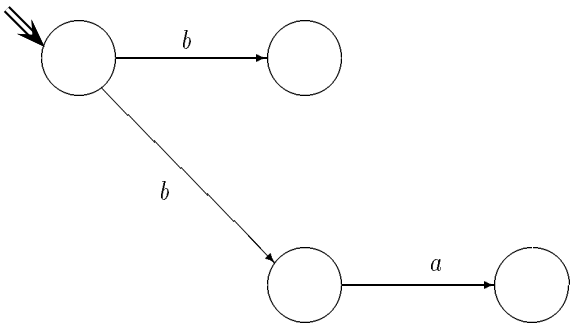
□

Note that this is an equality, not just an isomorphism. But the claim that the result is canonical for all graphs is not justified: there are G such that $\widehat{G}$ is not even isomorphic to G.

Counterexamples 2.8 (Disappearing nodes)



$\widehat{N} = \{\{<>\}, \{\}\}$



$\widehat{N} = \{\{<>\}, \{< b >\}, \{< b, a >\}\}$

□

In choosing to study the canonical representation, we exclude such counterexamples from consideration. The remainder of this section will define and justify the exclusion.

An important property in object-oriented programming is that each name should uniquely denote a single object. This is assured if the graph is deterministic.

Definition 2.9 (Determinism) A graph is deterministic if for each l and x there is at most one m such that $l \xrightarrow{x} m$. This is necessary for determinism of the corresponding automaton. In a data diagram, determinism permits automatic resolution of the polymorphic use of the same label to denote different functions on different data types. In object-oriented programming, it states the obvious fact that each attribute of each object can have only one value. □

If the original graph is deterministic, its canonical node-set $\widehat{N}$ satisfies an additional property familiar from process algebra — it is a bisimulation [Milner].

Definition 2.10 (Bisimulation) A family N of sets of traces is a bisimulation if it is a partial equivalence which is respected by trace extension. More formally, for all p, q in N

$$\begin{array}{rcl} p = q & \vee & p \cap q = \{\} \\ s,t \in p & \wedge & t \frown u \in q \;\Rightarrow\; s \frown u \in q \end{array}$$ □

Determinism ensures that any two distinct objects will have distinct trace sets, except in the extreme case that both have empty trace sets. Such objects can never be accessed by a program, so they might as well not exist.

Definition 2.11 (Accessibility) A node n is accessible if traces(n) is non-empty. A graph is accessible if all its nodes are, i.e. $\{\} \stackrel{\sim}{\in} \widehat{N}$. For automata, inaccessible nodes represent unreachable states, which can and should be ignored in any comparison between them. In a heap, they represent unusable storage, which can and should be garbage-collected (or otherwise explicitly deleted). □

At last we can show that we can model all the graphs that we are interested in by simply considering canonical graphs; furthermore, we can assume that N is always a prefix-closed bisimulation.

Theorem 2.12 (Representability) If G is deterministic and accessible, it is isomorphic to $\widehat{G}$

Proof Let $f(n) =_{df}$ traces(n) for all $n \in N$. Because G is deterministic

$$(n \neq m) \;\Rightarrow\; (\text{traces}(n) \neq \text{traces}(m))$$

Futhermore we have

$$\begin{array}{rl} & f(n) \xrightarrow[\widehat{G}]{x} f(m) \\ \equiv & \{\text{def of } \widehat{E}\} \\ & \text{traces}(n)^\frown < x > \subseteq \text{traces}(m) \\ \equiv & \{\text{traces}(n) \neq \{\} \text{ and } G \text{ is deterministic}\} \\ & n \xrightarrow[G]{x} m \end{array}$$ □

We can now solve the problems of graph representation left open in the previous section: objects will be named by traces, and inaccessible objects will disappear. We will assume that the heap G is at all times held in its canonical representation; and redefine each operation of object-oriented programming as an operation on the trace sets N.

Edge deletion $t \rightarrow x :=$ **nil** now has to remove not only the single edge x, but also all traces that include this edge. Every such trace must begin with a trace of the object t itself, i.e. a trace which is

equivalent to t by the equivalence N. The trace to be removed must of course contain an occurrence of $< x >$, the edge to be removed. It ends with a trace leading to some other node n in N. The traces removed from n are therefore exactly defined by the set

$$[t]^\frown < x > ^\frown (t^\frown < x > \xrightarrow{*} n)$$

We use the usual square bracket notation $[t]_N$ that contains t to denote the equivalence class (or more simply just $[t]$). Of course, x may occur more than once in the trace, either before or after the occurrence shown explicitly above. In the following definition, the removal of the edge is followed by removal of any set that becomes empty – a simple mathematical implementation of garbage-collection.

Re-definition 2.13 (Edge deletion, Node creation)

$$\begin{array}{lllll} (t \rightarrow x & := & \textbf{nil}) & =_{df} & N := \{n - [t]^\frown < x > ^\frown (t^\frown < x > \xrightarrow{*} n) \mid n \in N\} - \{\{\}\} \\ (t \rightarrow x & := & \textbf{new}) & =_{df} & t \rightarrow x := \textbf{nil};\ N := N + \{[t]^\frown < x >\} \end{array}$$ □

Unfortunately, pointer swing is even more complicated than this. We consider first the effect of $t^\frown < y > := s$, in the case where y is a **new** label, occurring nowhere else in the graph. The question now is, what are all the new traces introduced as a result of insertion of this new and freshly labelled edge? As before, every such trace must start with a trace from $[t]$, followed by the first occurrence of y. But now we must consider explicitly the possibility that the new edge occurs many times in a loop. The trace that completes the loop from the head of y back to its tail must be a path leading from s to t in the original graph, i.e. a member of $(s \xrightarrow{*} t)$. After any number of repetitions of $((s \xrightarrow{*} t)^\frown < y >)$, the new trace concludes with a path from s to some node n. The traces added to an arbitrary equivalence class n are exactly defined by the set

$$[t]^\frown < y > ^\frown ((s \xrightarrow{*} t)^\frown < y >)^* ^\frown (s \xrightarrow{*} n)$$

Note that in many cases $(s \xrightarrow{*} n)$ will be empty, because there is no path from s to n. Then by definition of$^\frown$between sets, the whole of the set described above is empty, and no new traces are added to n.

After inserting these new traces, it is permissible and necessary to remove the original edge x from the original graph and from the newly added traces too. Finally, the freshly named new edge y can be safely renamed as x.

Re-definition 2.14 (Pointer swing)

$$\begin{array}{lll} (t \rightarrow x := s) & =_{df} & \text{let } y \text{ be a fresh label in} \\ & & \quad N := \{n + [t]^\frown < y > ^\frown ((s \xrightarrow{*} t)^\frown < y >)^* ^\frown (s \xrightarrow{*} n) \mid n \in N\}; \\ & & \quad t \rightarrow x := \textbf{nil};\ \text{rename } y \text{ to } x \end{array}$$ □

Note that it is **not** permissible to delete the edge x before adding the new edge: this could make inaccessible some of the objects that need to be retained because they are accessible through s. The problem is clearly revealed in the simplest case: $t \rightarrow x := t \rightarrow x$. The necessary complexity of the pointer swing is a serious, perhaps a crippling disadvantage of the trace model of pointers and objects.

3 Applications

The purpose of our investigations is not just to contribute towards a fully abstract denotational semantics for an object-oriented programming language. We also wish to provide assistance in reasoning about the correctness of such programs, and to clarify the conditions under which they can be validly optimised. Both objectives can be met with the aid of assertions, which describe useful properties of the values of variables at appropriate times in the execution of the program. To formulate clear assertions (unclear ones do not help), we need an expressive language; and to prove the resulting verification conditions, we need a toolkit of powerful theorems. This section makes a start on satisfying

both these needs.

Two important properties of an individual node are defined as follows. It is acyclic if the only path leading from itself to itself is the empty path.

$$n \text{ is acyclic } =_{df} (n \xrightarrow{*} n) = \{<>\}$$

A graph is acyclic if all its nodes are. A node is a sink if there is no node accessible from it except itself

$$n \text{ is a sink } =_{df} \forall m \bullet n \xrightarrow{*} m \subseteq \{<>\}$$

These definitions can be qualified by a subset B of the alphabet, e.g.

$$n \text{ is a } B\text{-sink } =_{df} \forall m \bullet (n \xrightarrow{*} m) \cap B^* \subseteq \{<>\}$$

Two important relationships between nodes are connection and dominance. Connection is defined by the existence of a path between the nodes; and this path may be required to use only labels from B

$$m \overset{B}{\Longrightarrow} n \ =_{df} \ (n \xrightarrow{*} m) \cap B^* \neq \{\}$$

$\overset{B}{\Longrightarrow}$ is clearly a pre-order, i.e., transitive, and reflexive, but it is antisymmetric only in acyclic graphs. The root is the bottom of any accessible graph. The superscript B is omitted when it is the whole alphabet of the graph under discussion. The relation of dominance between objects is stronger than connection. One object l in a graph dominates an object m if every path to m leads through l

$$l \sqsubseteq m \ =_{df} \ l\hat{}(l \xrightarrow{*} m) = m$$

Deletion of a dominating object makes a dominated object inaccessible. So this relationship is very important in proving that a graph remains accessible and/or acyclic after a pointer swing. Its properties are similar to those of the prefix ordering over simple traces.

Theorem 3.0 (Dominance ordering) $\sqsubseteq$ is a partial order with the root as a bottom and the empty set as its top. The dominators of any node are totally ordered, i.e.

$$\text{if } l \sqsubseteq n \text{ and } m \sqsubseteq n \text{ then } l \sqsubseteq m \text{ or } m \sqsubseteq l$$

Proof see appendix. □

For non-empty nodes, dominance implies connection. If a node has only one trace, then every node that connects to it will dominate it. If all nodes have this property, the graph is called a divergent tree — divergent because all its pointers point away from the root and towards its sinks (i.e. the leaves).

In a language without garbage-collection, there is a grave danger that a pointer swing will leave an object that has no other pointer pointing to it. Such an object can never again be accessed by the program, and the storage space that it occupies will never be reused. This phenomenon is known as a space leak. In order to prevent it, the programmer must accept the obligation to ensure that a certain precondition is satisfied before each pointer swing $s \rightarrow x := t$. The relevant precondition is expressed as non-dominance

$$\neg s \sqsubseteq s\hat{}<x>$$

In a language without garbage-collection, the only way in which heap storage can be recovered for reuse is by an explicit command in the program, declaring that a particular object will never be accessed again. We will treat the simplest form of atomic deletion, as for example the delete command in PASCAL. This command must be given at the same time that the last pointer to the object

is deleted by $s \rightarrow x := \mathbf{nil}$. The precondition of such a deletion is the opposite of that for an assignment

$$s \sqsubseteq s^\frown < x >$$

In fact, a stronger precondition is necessary. All the objects accessible through $s \ ^\frown < x >$ must be accessible through some other object as well (otherwise their space would leak anyway). The full precondition for deletion is

$$\forall y \bullet (s^\frown < x > \sqsubseteq s^\frown < x >^\frown y \ \mathbf{iff}\ y = <>).$$

The complexity of these preconditions may explain why control of space leaks is a difficult problem in practice.

A heap as represented in the store of a computer must be described as a single variable, even though its value is of great size and complexity. Any pointer in the heap can at any time be swung to point to any other object whatsoever. To control this complexity, a programmer usually constrains the use of a heap in a highly disciplined way. The heap is understood to be split into a number of component subgraphs, satisfying invariant properties that limit the connections within and between the components. A component of a graph can readily be selected in two ways: by restricting the alphabet, or by concentration on a single branch.

Definition 3.1 (Subgraphs)

$$N \upharpoonright B =_{df} \{n \cap B^* \mid n \in N\} - \{\{\}\}$$

$N \upharpoonright B$ is a canonical graph with alphabet B, containing just those objects nameable by chains of labels drawn wholly from B.

$$N/n = \{n \xrightarrow{*} m \mid m \in N\} - \{\{\}\}, \text{ where } n \in N$$

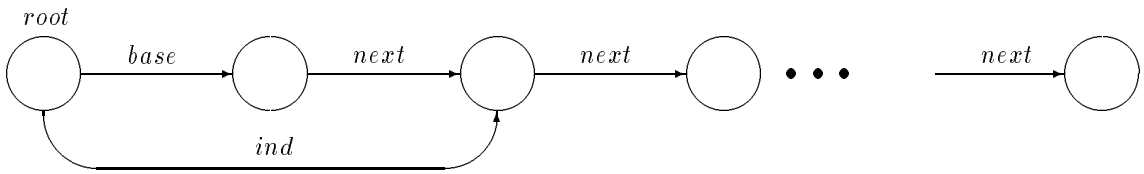
N/n is a canonical graph, isomorphic to the subgraph of all nodes accessible from n. It consists of just that part of the heap that is seen by a method local to n. □

These subgraph operations obey laws identical to those found in process algebra

$$\begin{array}{lcl} N \upharpoonright A & = & N, \ \text{if } A \text{ is the alphabet of the graph} \\ (N \upharpoonright B) \upharpoonright C & = & N \upharpoonright (B \cap C) \\ N \upharpoonright \{\} & = & 0_{\{\}} \\ N/<> & = & N \\ (N/s)/t & = & N/(s^\frown t) \end{array}$$

The purpose of this paper has been to provide a conceptual framework for formalisation of invariant assertions about data structures represented as objects in a heap. Class and type declarations serve the same purpose; they are also carefully designed to enjoy the additional advantage that their validity can be checked by compiler. Assertions have more expressive power, but they can be tested only at run time, and they can be validated only by proof. In the remainder of this section we explore the power of the trace model in the formulation of assertions, and suggest that a diagram may be helpful in visualising them.

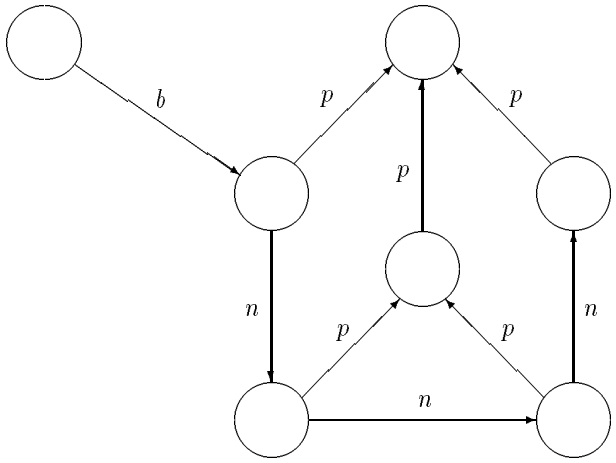
Definition 3.2 (Chain) Consider a pair of labels $B = \{base, next\}$. This defines a chain if $C = (N/base) \upharpoonright \{next\}$ is invariantly acyclic.



A variable ind is an index into this chain if invariantly $base \stackrel{next}{\Longrightarrow} ind$. A last-pointer is an index that always points to a $next$-sink. A final segment of the chain, chopped off at a given index, is C/ind. □

A chain is often used to scan a set of objects of interest. A good example is a convergent tree — convergent because the pointers point away from the leaves towards the root (a sink). Without a chain through them, the leaves (and indeed the whole tree) would be inaccessible.

Figure 3.3 (Convergent tree)



The attribute p points from an offspring to its parent in the tree; the attribute n constructs the leaf chain starting at a declared base variable b. □

We wish to formalise the properties shared by all such trees, without restriction on size or shape, and without defining which other attributes besides p and n may be pointing to or from the nodes. Let us first confine attention to the subgraph T of interest

$$T =_{df} (N/b) \upharpoonright \{p, n\}$$

The aim is to formulate the desired invariant properties of T as a conjunction of simple conditions that can be checked separately, and reused in different combinations for different purposes. The first condition has already been given a formal definition

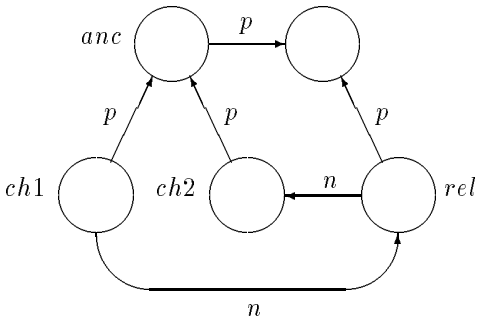
1. T is acyclic

2. Every object on the chain has a parent: if $j \stackrel{n}{\longrightarrow} k$ then $(\exists l \bullet k \stackrel{p}{\rightarrow} l) \ \wedge \ (\exists l \bullet j \stackrel{p}{\rightarrow} l)$

3. No parent is an object on the chain: if $l \stackrel{p}{\longrightarrow} m$ then $(\neg\, k \stackrel{n}{\rightarrow} m) \wedge (\neg\, m \stackrel{n}{\rightarrow} k)$

4. Any two nodes on the tree share a common ancestor: $\forall j,k\ \exists l \bullet j \stackrel{p}{\Rightarrow} l \wedge k \stackrel{p}{\Rightarrow} l$

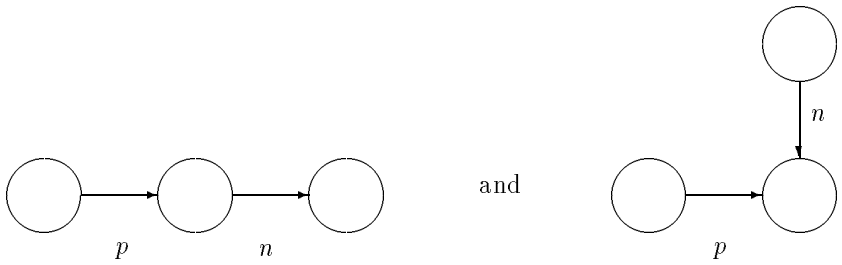
There is one more property that is usually desired of a leaf-chain: it should visit the leaves in some reasonable order, for example, close relatives should appear close in the chain. In particular, the following picture should **not** appear in the graph.



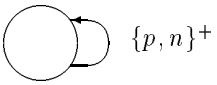
Note that *ch1* and *ch2* are more closely related to each other than to *rel*, which therefore should not separate them in the chain. The requirement is formalised

5. If $ch1 \stackrel{p}{\Rightarrow} anc \wedge ch2 \stackrel{p}{\Rightarrow} anc \wedge ch1 \stackrel{n}{\Rightarrow} rel \wedge rel \stackrel{n}{\Rightarrow} ch2$ then $rel \stackrel{p}{\Rightarrow} anc$.

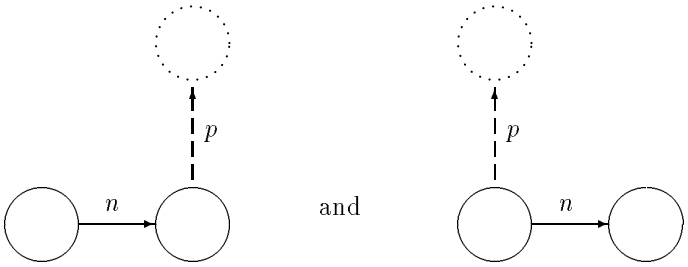
These invariants are expressed in the predicate calculus, using variables that have either implicit quantification over all traces in $\bigcup T$, or explicit existential quantification. The invariants can also be conveniently represented pictorially in the graphical calculus [Curtis and Lowe]. The simpler invariants directly prohibit occurrence in the heap of any subgraph of a certain shape. For example, condition (2) prohibits any occurrence of the two shapes



More formally, there is no homomorphism from either of these graphs into the heap. The acyclic condition (1) can be pictured by using a single arrow to represent a complete (non-empty) trace drawn from the specified alphabet; the following is prohibited

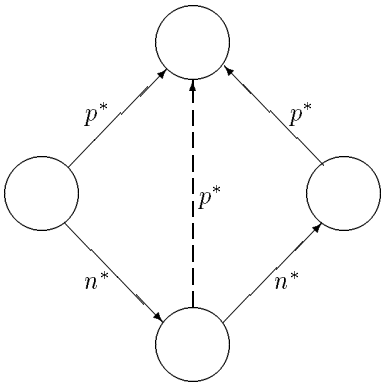


The more complicated invariants take the form of an implication, whose consequent has existentially quantified variables. These are drawn as dotted lines rather than the solid lines that represent variables universally quantified over the whole formula. So the condition (2) would be drawn



The condition (4) combines this convention with the path convention

The fifth condition is the most elaborate



The meaning of the dotted line convention illustrated above can be formalised, again in terms of graph homomorphisms. The picture states that every homomorphism from the graph drawn as solid lines and nodes can be extended to a homomorphism from the whole picture, including dotted lines and nodes. The extension must not change the mapping of any of the solid components. Even more formally, the diagram defines an obvious injective homomorphism $j : solid \rightarrow diagram$ from its solid components to the whole diagram. It states that for all $h : solid \rightarrow T$ there exists an $h' : diag \rightarrow T$ such that $h = j; h'$ (h factors through j). In plainer words, perhaps the programmer's instinct to draw pictures when manipulating pointers can be justified by appeal to higher mathematics.

4 Conclusion

The ideas reported in this paper have not been pursued to any conclusion. Perhaps, in view of the difficulties described at the end of section 2, they never will be. Their interest is mainly as an example of the construction of a generic mathematical model to help in formalisation of assertions about interesting and useful data structures. Such assertions can be helpful in designing and maintaining complicated class libraries, and in testing the results of changes, even if they are never used for explicit program proof.

Other published approaches to reasoning about pointer structures have been much better worked out. An early definition of a tree-structured machine with an equivalence relation for sharing was given in [Landin]. The closest in spirit to the trace model is described in [Morris] and applied to the proof of an ingenious graph marking algorithm. A similar approach using nice algebraic laws was taken in [Nelson]. Another promising approach [Möller] exploits the proof obligation as a driver for the design

of the algorithm in a functional style. It models each label as a function from addresses to values or other addresses contained in the addressed location. Other authors too have been deterred by the complexity of sharing structures introduced by pointer swing [Suzuki].

Acknowledgements For useful comments on earlier drafts we thank Frances Page, Bernhard Möller, Manfred Broy, Jay Misra, Paul Rudin, Ralph Steinbrüggen, Zhou Yu Qian.

References

[1] M. Abadi and L. Cardelli. A theory of objects. Springer (1998).

[2] L. Cardelli. A semantics of multiple inheritance. Information and Computation 76: 138–164 (1988).

[3] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. Information and Computation 114(2), 329–350 (1994).

[4] S. Curtis and G. Lowe, A graphical calculus. In B. Möller (ed) Mathematics of Program Construction LNCS 947 Springer (1995)

[5] O. Dahl and K. Nygaard. Simula, an Algol-based simulation language. Communications of the ACM 9(9) 671–678 (1966).

[6] A. Goldberg and D. Robson. Smalltalk-80. The language and its implementation. Addison-Wesley (1983).

[7] C.A.R. Hoare, Communicating Sequential Processes. Prentice-Hall (1985).

[8] S.N. Kamin and U.S. Reddy. Two semantic models of object-oriented languages. In C.A. Gunter and J.C. Mitchell (eds): Theoretical Aspects of Object-Oriented Programming, 463–495, MIT Press, (1994).

[9] P.J. Landin, A correspondence between ALGOL 60 and Church's lambda-notation Part 1. Communications ACM 8.2 (1965) 89-101

[10] B. Meyer. Object-oriented Software Construction, Prentice-Hall second edition (1997).

[11] R. Milner. Communication and Concurrency, Prentice Hall (1987)

[12] B. Möller, Towards pointer algebra. Science of Computer Programming 21 (1993), 57-90.

[13] B. Möller, Calculating with pointer structures. Proceedings of Mathematics for Software Construction, Chapman and Hall (1997), 24-48.

[14] J.M.Morris, A general axiom of assignment, Assignment and linked data structure, A proof of the Schorr-Waite algorithm. In M Broy and G. Schmidt (eds.) Theoretical Foundations of Programming Methodology, 25-51, Reidel 1982 (Proceedings of the 1981 Marktoberdorf Summer School).

[15] G. Nelson, Verifying reachability invariants of linked structures. Proceedings of POPL (1983), ACM Press, 38-47.

[16] N. Suzuki, Analysis of pointer rotation. Communications ACM vol 25 No 5, May (1982), 330-335.

Appendix

Proof of Theorem 3.0

First we are going to show that dominance is a partial order.

(reflexive)
$$\begin{array}{lll} & & l \\ & = & \{s^\frown <> \ = \ s\} \\ & & l^\frown\{<>\} \\ & \subseteq & \{<> \in (l \xrightarrow{*} l)\} \\ & & l^\frown(l \xrightarrow{*} l) \\ & \subseteq & \{\text{Theorem 2.3.(1)}\} \\ & & l \end{array}$$

(antisymmetric) Assume that $l \sqsubseteq m$ and $m \sqsubseteq l$. If $l = \{\}$ then

$$m \ = \ l^\frown(l \xrightarrow{*} m) \ = \ \{\}^\frown(l \xrightarrow{*} m) \ = \ \{\} \ = \ l$$

Assume that $l \neq \{\}$. From the fact that

$$l \;=\; l\hat{}(l \xrightarrow{*} m)\hat{}(m \xrightarrow{*} l)$$

and $l \neq \{\}$ we conclude that

$$\begin{array}{ll} & <> \in (l \xrightarrow{*} m)\hat{}(m \xrightarrow{*} l) \\ \equiv & \{s\hat{}t = <> \;\; \textbf{iff} \;\; s = t = <>\} \\ & <> \in (l \xrightarrow{*} m) \cap (m \xrightarrow{*} l) \\ \Rightarrow & \{l \;=\; m\hat{}(m \xrightarrow{*} l) \text{ and } m \;=\; l\hat{}(l \xrightarrow{*} m)\} \\ & (l \subseteq m) \;\wedge\; (m \subseteq l) \\ \equiv & l = m \end{array}$$

(transitive) Assume that $l \sqsubseteq m$ and $m \sqsubseteq n$.

$$\begin{array}{ll} & n \\ \supseteq & \{\text{Theorem 2.3(1)}\} \\ & l\hat{}(l \xrightarrow{*} n) \\ \supseteq & \{\text{Theorem 2.3(2)}\} \\ & l\hat{}(l \xrightarrow{*} m)\hat{}(m \xrightarrow{*} n) \\ = & \{l \sqsubseteq m \text{ and } m \;\sqsubseteq\; n\} \\ & n \end{array}$$

Let n be a non-empty node. Assume that

$$l \sqsubseteq n \text{ and } m \sqsubseteq n$$

For any subset X of A^* we define $sht(X)$ as the set of shortest traces of X

$$sht(X) \;=_{df}\; \{s \in X \mid \forall t \in X \bullet t \leq s \Rightarrow t = s\}$$

From the fact that $n \neq \{\}$ we conclude that neither $sht(l)$ nor $sht(m)$ is empty. Consider the following cases:

(1) $sht(l) \cap sht(m) \neq \{\}$: From the determinacy it follows that

$$l = m$$

(2) $sht(l) \cap sht(m) \;=\; \{\}$: From the assumption that $l \sqsubseteq n$ and $m \sqsubseteq n$ it follows that

$$\forall u \in sht(l) \exists v \in sht(m) \bullet (u \leq v \;\vee\; v \leq u)$$

and

$$\forall v \in sht(m) \exists u \in sht(l) \bullet (u \leq v \;\vee\; v \leq u)$$

(2a) $\forall u \in sht(l) \exists v \in sht(m) \bullet v \leq u$: From the bisimulation property it follows that

$$m \sqsubseteq l$$

(2b) $\forall v \in sht(m) \exists u \in sht(l) \bullet u \leq v$: In this case we have

$$l \sqsubseteq m$$

(2c) There exist u, $\hat{u} \in sht(l)$ and v, $\hat{v} \in sht(m)$ such that

$$u < v \text{ and } \hat{v} < \hat{u}$$

Let

$$\begin{array}{lll} j & =_{df} & min\{length(s) \mid s \in sht(l \xrightarrow{*} n)\} \\ k & =_{df} & min\{length(t) \mid t \in sht(m \xrightarrow{*} n)\} \end{array}$$

From $u \leq v$ we conclude that $j > k$, and from $\hat{v} < \hat{u}$ we have $j < k$, which leads to contradiction.

From the above case analysis we conclude that

$$l \sqsubseteq m \;\; \text{or} \;\; m \sqsubseteq l$$

□