A Theory of Pointers for the UTP

Will Harwood[1], Ana Cavalcanti[2], and Jim Woodcock[2]

[1] Citrix Systems (R & D) Ltd, Venture House, Cambourne Business Park
Cambourne, Cambs, UK
[2] University of York, Department of Computer Science
York, UK

Abstract. Hoare and He's unifying theories of programming (UTP) provide a collection of relational models that can be used to study and compare several programming paradigms. In this paper, we add to the UTP a theory of pointers and records that provides a model for objects and sharing in languages like Java and C++. Our work is based on the hierarchical addressing scheme used to refer to record fields (or object attributes) in conventional languages, rather than explicit notions of location. More importantly, we support reasoning about the structure and sharing of data, as well as their, possibly infinite, values. We also provide a general account of UTP theories characterised by conjunctive healthiness conditions, of which our theory is an example.

Keywords. semantics, refinement, relations, object models.

1 Introduction

Interest in reasoning about pointers is not recent [4], and has been renewed by the importance of sharing in object-oriented languages [1, 16]. The classic model of pointers [17] uses two functions: one associates memory addresses to values, and the other associates names to memory addresses. Similarly, most models use indexes to represent memory locations or embed a heap [13, 24]. Modern object-oriented languages, however, do not directly support manipulation of addresses.

The unifying theories of programming (UTP) [11] provide a modelling framework for several programming paradigms. It is distinctive in that its uniform underlying alphabetised relational model allows the combination of constructs from different theories, and supports comparison and combination of notations and techniques. Currently, there are UTP theories for imperative languages, functional languages, CSP, timed notations, and so on.

In this paper, we present a UTP theory for pointers based on the hierarchical addressing created by data types defined by recursive records. For example, for a variable l, we have the address l itself, and possibly addresses like $l.label$ and $l.next.item$, if the object value of l contains attributes *label* and *next*, and the value of $l.next$ is another object with an attribute *item*.

We have three components in our model: a set of (hierarchical) addresses, a function that associates the addresses of attributes that are not object valued to their primitive values, and a sharing relation. Our addresses are particular

to each variable, and we abstract from the notion of specific locations. This simplifies our theory, as compared, for example, to our own previous work [5].

We assume that all values have a location, including primitive values. Variables are names of locations, as are attribute access expressions like $l.next$, which define composed names of locations. These names, however, also have a value. A distinguishing feature of our model is that it represents sharing in a program as an updatable automaton that also associates (some) addresses to values. This allows us to define a simple de-referencing operator to tie the values in the automaton to those of the programming variables and of attribute accesses. In this way, we can, to a large degree, separate specifications into pure logical expressions on values, and update expressions involving pointers.

We do not assume strong typing, and so can cater for the use of pointers in languages like Lisp and, to some extent, C, although we do not cover pointer arithmetic. Infinite values are constructible by loops in pointer references; they are explicitly definable by fixed points and are generally represented by partial functions. In our model, de-referencing a pointer yields a partial function that can represent an infinite value. Finally, we do not model unreachable locations: we have automatic garbage collection. The only way of naming a location is by a path (composed name) that leads (refers) to it; if there is no path to a location, it does not exist. This does mean that we cannot reason about issues like memory leakage, but have a simpler model to reason about the functionality of a program.

In this paper, we also define a few constructs related to creation and assignment of records. We give their predicate model, and establish the adequacy of the definitions by proving that the predicates are healthy. The reasoning technique encouraged by the UTP is based on algebraic laws of refinement; our theory, however, can also be used to justify the soundness of techniques based on Hoare triples, for example. An account of Hoare logic in the UTP is available in [11].

Our long-term goal is to provide a pointer semantics for an object-oriented language for refinement that supports the development of state-rich, concurrent programs. In particular, we are interested in *OhCircus* [6]; it combines Z and CSP, with object-oriented constructs in the style of Java, and, as such, it has in the UTP an appropriate choice of semantic model. Following the UTP style, we are considering individual aspects of the *OhCircus* semantics separately. The theory presented here will be integrated with the copy semantics of *OhCircus*.

The UTP relational models are defined in a predicative style similar to that adopted in Z or VDM, for example. They comprise a collection of theories that model specifications, designs, and programs; a theory is characterised by a set of alphabetised predicates that satisfy some healthiness conditions.

There is a subtle difficulty in defining a pointer theory that can be easily combined with the existing UTP theories: the UTP is a logical language where variables range over values. To illustrate the issue, we consider a variable l that holds a value of a type $List ::= (label : \mathbb{Z};\ next : List)$ of recursive records with fields *label* and *next*. After the assignments $l.label := 1$ and $l.next := l$, the value of l is an infinite list, but it is constructed as a pointer structure. Pointers are used for two distinct purposes: to construct infinite values by self-reference,

and to share storage. We need a model that records the sharing, but allows appropriate reasoning about values, to simplify specification.

A simple solution is to model storage as a graph and to use fixed points to handle self-references when constructing the denotation of a value. In this case, however, the update operations explicitly use fixed points (for the values) and reasoning is cumbersome. Instead, we create a model where updates act directly on the values associated with the graph as well as on its structure.

A large number of healthiness conditions used to characterise UTP theories are defined by a conjunction. In this case, a function $\mathbf{H}$ from predicates to predicates is defined as $\mathbf{H}(P) = P \wedge \psi$, for some predicate ψ; the healthy predicates are the fixed points of $\mathbf{H}$: those for which $\mathbf{H}(P) = P$. A number of properties are satisfied by these predicates, independently of the particular definition of ψ. We present and prove some of these results; they simplify proof in our theory.

In [5], we also present a theory of pointers and records for the UTP. It is based on the model of entity groups in [21] to formalise rules of a refinement calculus for Eiffel [15]. In that work, the complications of an explicit model of the memory are also avoided; each entity (variable) is associated with the set of variables that share its location (entity group). Here, we use a binary relation to model sharing, and record the set of valid names and values explicitly to simplify definitions and proof. This is in addition to the simplification that arises from the separate treatment of healthiness conditions defined by conjunction.

In the next section, we describe the UTP. Section 3 describes our model informally. In Section 4, we present general results about theories characterised by healthiness conditions defined by conjunctions, before we formalise our theory in Section 5. A model for usual programming constructs is presented in Section 6. Finally, in Section 7 we consider some related and future work.

2 Unifying theories of programming

In the unifying theories of programming, relations are defined by predicates over an alphabet (set) of observational variables that record information about the behaviour of a program. In the simplest theory of general relations, these include the programming variables v, and their dashed counterparts v', with v used to refer to an initial observation of the value of v, and v' to a later observation. In the sequel, we use v to stand for the list of all programming variables, and v' to the corresponding list of dashed variables. The set of undecorated (unprimed) variables in the alphabet αP of a predicate P is called its input alphabet $in\alpha P$, and the set of dashed variables is its output alphabet $out\alpha P$. A condition is a predicate whose alphabet includes only input variables.

Theories are characterised by an alphabet and by healthiness conditions defined by monotonic idempotent functions from predicates to predicates. The predicates of a theory with an alphabet A are all the predicates on A which are fixed points of the healthiness conditions. As an example, we consider designs.

The general theory of relations does not distinguish between terminating and non-terminating programs. This distinction is made in the UTP in a theory

of designs, which includes two extra boolean observational variables to record the start and the termination of a program: ok and ok'. All designs can be split into precondition/postcondition pairs, making them similar to specification statements of a refinement calculus. The monotonic idempotents used to define the healthiness conditions for designs can be defined as follows, where P is a relation (predicate) with alphabet $\{ok, ok', v, v'\}$.

$$\mathbf{H1}(P) \ \widehat{=} \ ok \Rightarrow P \qquad \mathbf{H2}(P) \ \widehat{=} \ P \ ; J, \ \textbf{where} \ J \ \widehat{=} \ (ok \Rightarrow ok') \wedge v' = v$$

If P is **H1**-healthy, then it makes no restrictions on the final value of variables before it starts. If P is **H2**-healthy, then termination must be a possible outcome from every initial state. The functional composition of **H1** and **H2** is named **H**. Our definition of **H2** is different from that in [11], but it is equivalent [7]; it uses the sequence operator that we define below.

Typically, a theory defines a number of programming operators of interest. Common operators like assignment, sequence, and conditional, are defined for general relations. Sequence is relational composition.

$$P; \ Q \mathrel{\widehat{=}} \exists \, w_0 \bullet P[w_0/w'] \wedge Q[w_0/w], \textbf{ where } out\alpha(Q) = in\alpha(Q)' = w'$$

The relation $P; \ Q$ is defined by a quantification that relates the intermediate values of the variables. It is required that $out\alpha(P)$ is equal to $in\alpha(Q)'$, which is named w'. The sets w, w', and w_0 are used as lists that enumerate the variables of w and the corresponding decorated variables in the same order.

A conditional is written as $P \lhd b \rhd Q$; its behaviour is (described by) P if the condition b holds, else it is defined by Q.

$$P \lhd b \rhd Q \mathrel{\widehat{=}} (b \wedge P) \vee (\neg b \wedge Q), \textbf{ where } \alpha(b) \subseteq \alpha(P) = \alpha(Q).$$

A central concern of the UTP is refinement. A program P is refined by a program Q, which is written $P \sqsubseteq Q$, if, and only if, $P \Leftarrow Q$, for all possible values of the variables of the alphabet. The set of alphabetised predicates form a complete lattice with this ordering. Recursion is modelled by weakest fixed points $\mu \, X \bullet F(X)$, where F is a monotonic function from predicates to predicates.

The programming operators of a theory need to be closed: they need to take healthy predicates to healthy predicates. In Section 4, we provide some general results for healthiness conditions defined by conjunctions.

3 A model for pointers

In this section we introduce a straightforward and intuitive model of a storage graph that we call a pointer machine. Afterwards, in Section 3.2 we introduce an alternative representation for a pointer machine that is easier to model in the UTP. The new UTP theory itself is presented in Section 5.

3.1 The pointer machine

A simple model for storage is a labelled graph, in which the labels are names of attributes, and terminal nodes hold values. It is useful to think of this graph as a particular kind of automaton, a *pointer machine* that accepts addresses (attribute by attribute) and produces values when you reach a terminal node.

Intuitively, arcs represent pointers, internal nodes represent storage locations for pointers, and terminal nodes, storage locations for values. A mapping assigns attribute names to arcs for selecting pointers and values, and another gives the values stored at a terminal node. For example, Figure 1 gives a pointer machine for the variable l defined in Section 1. There is only one internal node X, the initial node, and one terminal node, Y. The arcs are $\{ X \mapsto X, X \mapsto Y \}$; we write $a \mapsto b$ to describe the pair (a, b). The set of labels is $\{l, \mathit{label}, \mathit{next}\}$, and the labelling functions are $\{ (X \mapsto X) \mapsto \mathit{next}, (X \mapsto Y) \mapsto \mathit{label} \}$ for the arcs, and just $\{ Y \mapsto 1 \}$ for the terminal node.

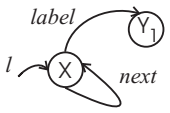

Fig. 1. A labelled graph representation for a pointer machine

In fact, a pointer machine naturally represents a structure in which there is a single entry point.Conceptually, we regard the start node as a fictional root of all memory, whose arcs correspond to simple variable names.

3.2 A simpler model

We represent pointer machines by a triple $\langle A, V, S \rangle$, where A is the set of addresses that the machine accepts, V is a partial function mapping addresses to primitive values, and S is an equivalence relation on addresses, recording that two addresses lead to the same node. The addressing map V defines which addresses yield values: the difference between A and dom V is the set of addresses that are accepted by the machine, but do not yield a primitive value. The set A defines the valid addresses. The storage map S records the sharing. For the pointer machine in Figure 1, A contains l, all the addresses formed only by accesses to *next* ($l.\mathit{next}$, $l.\mathit{next}.\mathit{next}$, and so on), and those that end with an access to *label*, possibly after (repeated) accesses to *next* ($l.\mathit{label}$, $l.\mathit{next}.\mathit{label}$, $l.\mathit{next}.\mathit{next}.\mathit{label}$, and so on). The V function maps all accesses to *label* to 1. Finally, S relates l to all (repeated) accesses to *next*, all these addresses to each other, and all accesses to *label* to each other as well.

The set of finite addresses $\mathit{FAd} \mathrel{\widehat{=}} (\mathrm{seq}\, \mathit{Label}) \setminus \{ \langle\rangle \}$ contains the non-empty sequences of labels. An infinite address is an element of $\mathit{seq}_\infty \mathrel{\widehat{=}} \mathbb{N}_1 \to \mathit{Label}$, that is, a total function from the positive natural numbers to *Label*. Finally, an address in $\mathit{Ad} \mathrel{\widehat{=}} \mathit{FAd} \cup \mathit{seq}_\infty$ can be finite or infinite.

Equality Two equalities are definable: *value equality*, written $=_v$, and *pointer equality*, which is written $=_p$. The former holds for pointers that have the same value. It establishes that, if you follow the pointers, and use the nodes you arrive at as the start nodes of two pointer machines, say $\langle A_1, V_1, S_1 \rangle$ and $\langle A_2, V_2, S_2 \rangle$, then $A_1 = A_2$ and $V_1 = V_2$, that is, the domains of definition and the addressing map of the machines are the same. This means that two pointers are equal if further addressing off these pointers leads to the same values (and the same failures). Pointer equality holds for pointers that point to exactly the same location.

For a finite p, we define the p-projection $\langle A.p, V.p \rangle$ of the machine with $A.p \mathrel{\widehat{=}} \{ q : Ad \mid p.q \in A \}$ and $V.p \mathrel{\widehat{=}} \{ q : Ad \mid p.q \in \mathrm{dom}\, V \bullet q \mapsto V(p.q) \}$. We use the dot operator to combine addresses, as well as to append an attribute name to the end of an address. The value associated with the object pointed by p is the tree coded by $\langle A.p, V.p \rangle$. Value equality is defined by equality of pointer projections, and pointer equality is defined by the storage map.

$$p =_v q \mathrel{\widehat{=}} (A.p = A.q \land V.p = V.q) \qquad p =_p q \mathrel{\widehat{=}} (p, q) \in S$$

For our simple example, we have that $l.next$ is both value and pointer equal to l. The l-projection of the machine described above is formed by stripping off the leading l in all addresses in A and dom V.

4 Conjunctive healthiness conditions

We refer to a healthiness condition that is, or can be, defined in terms of conjunction as a conjunctive healthiness condition. In this section, we consider an arbitrary conjunctive healthiness condition $\mathbf{CH}(P) = P \land \psi$, for some predicate ψ. All the healthiness conditions of our theory are conjunctive.

Conjunction, disjunction, and conditional are closed with respect to **CH**.

Theorem 1. *If P and Q are* **CH***-healthy predicates, then $P \land Q$, $P \lor Q$, and $P \lhd c \rhd Q$ are* **CH***-healthy.*

The proof of this and every other result in this paper can be found in [10].

To establish closedness for sequence, we consider a specific kind of conjunctive healthiness condition: those in which ψ is itself the conjunction of conditions ψ_i and ψ'_i over the input and output alphabets, respectively. In this case, **CH** imposes similar restrictions on the input and output alphabets. (As expected, the predicate P' is that obtained by dashing all occurrences of the observational variables in P.) With these results, we can prove closedness of sequence.

Theorem 2. *If P and Q are* **CH***-healthy, where $\mathbf{CH}(P) = P \land \psi \land \psi'$, for some condition ψ on input variables, then P; Q is* **CH***-healthy as well.*

The set of **CH**-healthy predicates is a complete lattice, since it is the image of a monotonic idempotent healthiness condition [11]. So, recursion can still be defined using weakest fixed points; closedness is established by the next theorem.

Theorem 3. *If F is a monotonic function from* **CH***-healthy predicates to* **CH***-healthy predicates, then $\mu_{ch}\, X \bullet F(X) = \mathbf{CH}(\mu\, X \bullet F(X))$, where $\mu_{ch}\, X \bullet F(X)$ is the least fixed point of F in the lattice of* **CH***-healthy predicates.*

This states that a recursion is a **CH**-healthy predicate, if, for instance, its body is built out of **CH**-healthy predicates itself using closed constructors.

Designs In general, a theory of **CH**-healthy predicates is disjoint from the theory of designs: on abortion, a design provides no guarantees, but a **CH**-healthy predicate requires ψ to hold. Of course, if ψ is *true*, in which case CH is the identity, we do not have a problem, but for interesting **CH**, there is a difficulty. We follow the UTP approach used to combine the theory of reactive processes and designs to combine the theory of designs with a theory of **CH**-healthy predicates. We take **CH** as a link that maps a design to a **CH**-healthy predicate.

What we have is an approximate relationship between the two theories: for a **CH**-healthy relation P, $\mathbf{CH} \circ \mathbf{H1}(P) \sqsubseteq P$. This is a property of a Galois connection that translates between the theories. The healthiness condition **H2** is not a problem: it commutes with **CH**, provided ψ does not refer to ok'.

Theorem 4. $\mathbf{CH} \circ \mathbf{H2} = \mathbf{H2} \circ \mathbf{CH}$*, provided ok' is not free in ψ.*

Galois connection Our proof of the existence of a Galois connection between the theories of **CH**-healthy predicates and designs relies on two simple lemmas about **H1**, **CH**, and refinement. In fact, instead of considering **H1** in particular, we consider an arbitrary implicational healthiness condition $\mathbf{IH}(P) = \phi \Rightarrow P$.

Lemma 1 (IH-refinement). $\mathbf{IH}(P) \sqsubseteq \mathbf{IH}(Q)$ *if, and only if,* $\mathbf{IH}(P) \sqsubseteq Q$.

This lemma lets us cancel an application of **IH** on the right-hand side of the refinement. This works because $\mathbf{IH}(P)$ is a disjunction, and the cancellation strengthens the implementation. Something similar can be done with **CH**, but since **CH** is a conjunction, the cancellation takes place on the specification side.

Lemma 2 (CH-refinement). $P \sqsubseteq \mathbf{CH}(Q)$ *if, and only if,* $\mathbf{CH}(P) \sqsubseteq \mathbf{CH}(Q)$.

Applications of the above lemmas justify the main result for a combination of the theories of **IH**-healthy and **CH**-healthy predicates.

Theorem 5. *There is a Galois connection between* **IH***-healthy and* **CH***-healthy predicates, where* **CH** *is the right adjoint and* **IH** *is the left one.*

$$P \sqsubseteq \mathbf{IH}(Q) \quad \textit{if, and only if,} \quad \mathbf{CH}(P) \sqsubseteq Q.$$

Here, P is **IH***-healthy, and Q is* **CH***-healthy.*

For designs, more specifically, we have the result below.

Theorem 6. **CH** *and* **H** *form a Galois connection between designs and* **CH***-healthy predicates.*

$$D \sqsubseteq \mathbf{H}(P) \quad \textit{if, and only if,} \quad \mathbf{CH}(D) \sqsubseteq P.$$

Here, D is a design, and P is **CH***-healthy.*

Proof of closedness of the operators in the combined theories is simple.

5 Pointers and records in the UTP

The alphabet of our theory includes three new observational variables A, V, and S that record separately the components of the pointer machine.

The first healthiness condition, named **HP1**, guarantees that A is prefix closed. We write $x < y$ when x is a (finite) strict prefix of the address y.

HP1 $\quad P = P \wedge \forall\, a_1 : A;\; a_2 : FAd \mid a_2 < a_1 \bullet a_2 \in A$

This means that if $x.y.z$, for instance, is a valid address, then $x.y$ and x must be as well. As already said, the healthiness conditions are characterised by functions, so in accordance with the UTP style, we use the name of the healthiness condition as the name of the corresponding function. In the case of **HP1**, for example, we have a function $\mathbf{HP1}(P) \mathrel{\widehat=} P \wedge \forall\, a_1 : A;\; a_2 : FAd \mid a_2 < a_1 \bullet a_2 \in A$.

To formalise **HP2**, we define the subset $term(A)$ of addresses of terminal nodes. In general, $\mathrm{term}(X) \mathrel{\widehat=} \{x : X \cap FAd \mid \neg\; \exists\, y : X \bullet x < y\}$, that is, a terminal address is finite and has no valid extensions. In **HP2**, we connect $\mathrm{dom}\, V$ and A by requiring each terminal in A to have a value defined by V.

HP2 $\quad P = P \wedge \mathrm{dom}\, V = \mathrm{term}(A)$

The third healthiness condition **HP3** connects the programming variables in the alphabet to pointers in the pointer machine. For every variable x, we use $'x$ to refer to its name. We require in **HP3** that $'x$ is a variable in the pointer machine, and that the value of x is consistent with that assigned by V. The variables of the pointer machine are the first elements of the addresses: for a set of addresses X, the variables are $\mathrm{vars}(X) \mathrel{\widehat=} \{x : X \bullet x(1)\}$. If $'x$ is a terminal, then x must have value $V('x)$. If $'x$ is not a terminal, then the value of x is a partial function that maps addresses to values defined by the projection of V at $'x$, that is, $V.('x)$. For every x in $in\alpha(P) \setminus \{A, V, S\}$, we define the de-referencing operator $!x$ as follows: $!x \mathrel{\widehat=} \mathbf{if}\ x \in \mathrm{term}(A)\ \mathbf{then}\ V(x)\ \mathbf{else}\ V.x$. For a dashed variable, the definition is $!(x') \mathrel{\widehat=} \mathbf{if}\ x \in \mathrm{term}(A')\ \mathbf{then}\ V'(x)\ \mathbf{else}\ V'.x$. In **HP3** we use this operator to constrain the values of the input variables.

HP3 $\quad P = P \wedge\; v_1 = \,!'v_1 \wedge \ldots \wedge v_n = \,!'v_n \wedge \{'v_1, \ldots, 'v_n\} = \mathrm{vars}(A)$
$\quad\quad \mathbf{where}\ \{'v_1, \ldots, 'v_n\} = in\alpha(P) \setminus \{A, V, S\}$

The remaining healthiness conditions are related to S. It should involve only addresses in A and should be an equivalence relation. We use R^* to describe the reflexive, symmetric, and transitive closure of R.

HP4 $\quad P = P \wedge S \in (A \leftrightarrow A) \wedge S = S^*$

Also, if two addresses are equivalent under S, then any extension by the same address should be equivalent. We define an equivalence relation E between addresses to be *forward closed* with respect to a set of addresses A to mean that once two addresses are equivalent, then their common extensions are equivalent, that

is $\mathrm{fclos}_A\, E \mathrel{\widehat{=}} \forall\, x, y, a : Ad \mid (x, y) \in E \wedge (x.a \in A \vee y.a \in A) \bullet (x.a, y.a) \in E$.

HP5 $\quad P = P \wedge \mathrm{fclos}_A\, S$

Finally, if two terminals share a location, then they have the same value.

HP6 $\quad P = P \wedge \forall\, a, b : Ad \mid (a, b) \in S \wedge a \in \mathrm{dom}\, V \bullet b \in \mathrm{dom}\, V \wedge V(a) = V(b)$

We also have extra healthiness conditions **HP7**-**HP12** that impose the same restrictions on the dashed variables. Our theory is characterised by the healthiness condition **HP**, the functional composition of all these healthiness conditions.

All our healthiness conditions are conjunctive; consequently, **HP** is conjunctive, and moreover, it imposes the same restrictions on S, V, and A, and on S', V' and A', as studied in Section 4. So, we can conclude, based on Theorems 1, 2, and 3, that the usual specification and programming constructs are closed with respect to **HP**. In addition, **HP** and **H** are adjuncts of a Galois connection that defines a theory of pointers for terminating programs. We only need to be careful with the definition of **HP2**. In the theory of pointer designs, ok and ok' are not programming variables, and just like A, V, and S, they are not to have space allocated in the pointer machine. So, ok is not to be included in the vector v of variables considered in **HP2**, and ok' is not to be included in v' in **HP8**.

In the sequel, we define some programming constructs; for that, it is useful to define $\mathbf{HPI} \mathrel{\widehat{=}} \mathbf{HP1} \circ \mathbf{HP2} \circ \mathbf{HP3} \circ \mathbf{HP4} \circ \mathbf{HP5} \circ \mathbf{HP6}$. It imposes restrictions only on the input variables. These definitions illustrate the use of the healthiness conditions also to simplify definitions; in particular **HP9** is very useful, as it relates changes in the machine to changes in values of variables. Most of the healthiness conditions are restrictions that characterise the automata that model pointer machines. In the case of **HP3**, and the corresponding **HP9**, however, we have healthiness conditions that unify the structural and logical views of variables. They are the basis for a reasoning technique that copes in a natural way with (infinite) values whose storage structure is also of interest.

6 Programming constructs

We can update the pointer machine using a value assignment, which we write $x := e$ for a finite address x in A, or a pointer assignment $x :\!- y$, where both x and y are finite addresses in A. Both types of assignment may change an internal or a terminal node, and consequently alter A, V, and S.

6.1 Value assignment

We define, for an address x, the set $\mathrm{share}(x) \mathrel{\widehat{=}} S(\!|\, \{x\} \,|\!)$ of addresses that share its location; $S(\!|\, \{x\} \,|\!)$ is the relational image of $\{x\}$ through S: all elements related to x in S. A value assignment to a terminal address is defined as follows.

$$x := e \mathrel{\widehat{=}} \mathbf{HPI} \circ \mathbf{HP9}(A' = A \wedge V' = V \oplus \{a : \mathrm{share}_S(x) \bullet a \mapsto e\}) \wedge S' = S)$$

provided $x \in \mathrm{dom}\, V$

The symbol $\oplus$ is used for the functional overriding operator. In the new value

of V, x and all the addresses that share its location are associated with the value e. The application of the healthiness condition **HPI** ensures that the input variables are healthy; **HP9** ensures that the values of the programming variables are updated in accordance with the changes to V.

For an internal address, the definition of assignment is a generalisation of that above. Before we present it, we define the set $\mathrm{ext}(x) \mathrel{\widehat{=}} \mathrm{share}_S(x)^{\uparrow}$ of all addresses that extend x or any of the other addresses that share the location of x. The set $X^{\uparrow} \mathrel{\widehat{=}} \bigcup\{x : X \cap FAd \bullet x\infty\}$ contains all addresses that extend those in a set X, and $x\infty \mathrel{\widehat{=}} \{\, a : Ad \bullet x.a \,\}$ contains all the extensions of the finite address x. All the addresses in $\mathrm{ext}(x)$ become invalid if x is assigned a value; they are removed from A, from the domain of V, and from the domain and range of S. The operators for domain and range subtraction are $\lhd$ and $\rhd$.

$$x := e \mathrel{\widehat{=}}$$

$$\mathbf{HPI} \circ \mathbf{HP9} \left(\begin{array}{l} A' = A \setminus \mathrm{ext}_S(x) \wedge \\ V' = (\mathrm{ext}_S(x) \lhd V) \cup \{a : (\mathrm{share}_S(x) \setminus \mathrm{ext}_S(x)) \bullet a \mapsto e\} \wedge \\ S' = \mathrm{ext}_S(x) \lhd S \rhd \mathrm{ext}_S(x) \end{array} \right)$$

provided $x \in A$ and $x \notin \mathrm{dom}\, V$.

For a terminal address x, the set $x\infty \cap A$ is empty, and so is $\mathrm{ext}_S(x) \cap A$. So, in the definition of assignment to a terminal address, we do not change A and S. In V, we include the addresses that share a location with x according to the new storage map S', that is, $\mathrm{share}_S(x) \setminus \mathrm{ext}_S(x)$.

As an example, we consider again the variable l in Figure 1; after the assignment $l.next := 3$, all extensions of $l.next$ become invalid. The set $\mathrm{share}_S(l.next)$ contains l and all accesses to *next*; so, $\mathrm{ext}(l.next)$ contains all addresses in A, except l. So, after $l.next := 3$, A contains only l, S only associates l to itself, and finally, the domain of V is wiped out, and l is added: it is mapped to 3.

The proofs of **HP**-healthiness [10] provide validation for our definitions.

6.2 Pointer Assignment

We present here just the definition of pointer assignment to an internal address. As a motivating example, we consider the variables l, m, and n depicted in Figure 2(a). After the assignment $l :\!- m.link$, the value and sharing properties of l and its extensions are completely changed, but no other variable is affected: l now points to the same location as $m.link$, but n, for example, does not change. The address l is still valid, but its extensions, like $l.next$, $l.label$, and so on, cease to exist. Instead, all addresses formed by concatenating a suffix of $m.link$ to l are now valid. For example, $l.value$, $l.link$, $l.link.link$ and so on become valid.

Accordingly, in the definition of $x :\!- y$, we remove $x\infty$ from A, and add the set $\{a : A.y \bullet x.a\}$ of new addresses. In the case of V, we remove $x\infty$ from its domain, and give x and its new extensions the values defined by y, if any. In our example, after the assignment $l :\!- m.value$, since $m.value$ is a terminal, with value 3, then l also becomes a terminal with the same value. On the other hand,

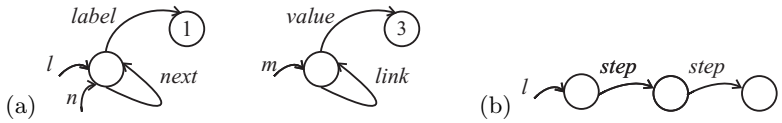

Fig. 2. Two pointer machines

after $l :- m.link$, the new terminal locations correspond to those of $m.link$. Namely, we have new terminals $l.value$, $l.link.value$ and so on, all with value 3.

If x and y share the same location, the pointer assignment $x :- y$ has no effect and, in particular, S does not change. If, however, they point to different locations, the sharing information for x and all elements of $x\infty$ change. To simplify the definition of S', we define the set $\mathrm{bsh}(x) \mathrel{\widehat{=}} \{x\} \cup x\infty$ of addresses whose sharing is broken. The existing sharing information about these addresses is eliminated, and their new sharing with y and its extensions is recorded.

Conditional expressions are used below: the value of $e_1 \lhd b \rhd e_2$ is e_1 if the condition b holds, otherwise its value is e_2.

$$x :- y \mathrel{\widehat{=}} \mathbf{HPI} \circ \mathbf{HP9} \left(\begin{array}{l} A' = A \setminus x\infty \cup \{ a : A.y \bullet x.a \} \wedge \\ V' = \left(\begin{array}{l} (x\infty \lhd V) \cup \\ (\{x \mapsto V(y)\} \lhd y \in \mathrm{dom}\ V \rhd \emptyset) \cup \\ \{ a : \mathrm{dom}\ V.y \bullet x.a \mapsto V(y.a) \} \end{array} \right) \wedge \\ S' = \left(\begin{array}{l} S \lhd (x, y) \in S \rhd \\ \left(\begin{array}{l} (\mathrm{bsh}(x) \lhd S \rhd \mathrm{bsh}(x)) \cup \\ (\{x\} \times ((S(\!|\ \{y\}\ |\!) \setminus x\infty) \cup \{x\})) \cup \\ \bigcup \{ a : A.y \bullet \{x.a\} \times ((S(\!|\ \{y.a\}\ |\!) \setminus \mathrm{bsh}(x)) \cup \{x.a\}) \} \end{array} \right)^* \end{array} \right) \end{array} \right)$$

provided $x \notin \mathrm{dom}\ V$ and x is a simple name.

The address x now shares its location with y and all the addresses that already share a location with y: those in $S(\!|\ \{y\}\ |\!)$. It may be the case, however, that y shares a location with an extension of x, and in this case that address does not exist anymore, and needs to be eliminated. In our example, after $l :- n.label$, l should be associated with $n.label$, $n.next.label$, $n.next.next.label$ and so on; however $S(\!|\ \{n.label\}\ |\!)$ also includes $l.label$, $l.next.label$, and all other accesses to $label$ via l; these need to be excluded. The same comment applies to the sharing information related to the new extensions $l.a$ of x; they should be related to $S(\!|\ \{y.a\}\ |\!)$, but the extensions of x should be eliminated. If present, x should also be eliminated, as information about it in S is no longer valid. In the case of $S(\!|\ \{y\}\ |\!)$, we know that x does not belong to this set, since $(x, y) \notin S$.

Finally, we need to consider the cases in which $S(\!|\ \{y\}\ |\!)$ is contained in $x\infty$, so that $S(\!|\ \{y\}\ |\!) \setminus x\infty$ is empty, or $S(\!|\ \{y.a\}\ |\!)$ is contained in $\mathrm{bsh}(x)$, so that $S(\!|\ \{y.a\}\ |\!) \setminus \mathrm{bsh}(x)$ is empty. The machine in Figure 2(b) gives us an example: after $l :- l.step$, the valid addresses are l and $l.step$, and S' should be the identity. When we eliminate l and all its extensions from S, however, we get the empty relation; furthermore, $S(\!|\ \{l.step\}\ |\!)$ contains only $l.step$, and

similarly, $S(\!|\ \{l.step.step\}\ |\!)$ is $\{l.step.step\}$. To guarantee that S' includes all valid addresses, we explicitly associate x and each new $x.a$ to themselves.

A value assignment $x := y$ affects the value of all addresses that share the location of x. In the case of a pointer assignment, however, not all of them are affected. For example, as we discussed above, even if the variables x and z share a location, $x :\!- y$ does not affect z. On the other hand, $x.a :\!- y$ affects both $x.a$ and $z.a$, since $x.a$ is an attribute of both x and z. The definition of $x.a :\!- y$ is similar to that of $x :\!- y$, but it takes into account the fact that other addresses, and not only $x.a$ and $x.a\infty$ are affected. Details are in [10].

6.3 Object creation

New structures are created in programming languages by allocating storage. The effect in the pointer machine is to make new addresses available.

In our untyped theory, we define that the attributes of newly created objects have an unspecified value. We introduce a construct $x : new(a)$, which allocates new storage for an object that becomes accessible from x; here a is a list of the attribute names. We use the notation $\{x.a_i\}$ to refer to the set of addresses formed by appending an attribute a_i in a to x. Similarly, we write $\{x.a_i \mapsto v_i\}$ to denote the set of pairs that associate each $x.a_i$ to the corresponding element of a list v of values; similarly $\{x.a_i \mapsto x.a_i\}$ associates each address $x.a_i$ to itself. The definition of $x : new(a)$ is much like that of a pointer assignment to x, and we consider below the case in which x is a simple name.

$$x : new(a) \mathrel{\widehat{=}} \mathbf{HPI} \circ \mathbf{HP9} \begin{pmatrix} A' = (A \setminus x\infty) \cup \{x.a_i\} \wedge \\ \exists\, v \bullet V' = ((x\infty \cup \{x\}) \lhd V) \cup \{x.a_i \mapsto v_i\} \wedge \\ S' = \begin{pmatrix} ((x\infty \cup \{x\}) \lhd S \rhd (x\infty \cup \{x\})) \cup \\ \{x \mapsto x\} \cup \{x.a_i \mapsto x.a_i\} \end{pmatrix} \end{pmatrix}$$

provided $x \in A$ and x is a simple name.

Its proof of healthiness is similar to that for assignment.

6.4 Variable declaration and undeclaration

Pointer variables can be introduced by the operator $begin(x)$ and removed by $end(x)$.

$$begin(x) \mathrel{\widehat{=}} \mathbf{HPI} \circ \mathbf{HP9} \begin{pmatrix} A' = A \cup \{\, x \,\} \wedge \\ \exists\, v \bullet V' = V \oplus \{\, x \mapsto v \,\} \wedge \\ S' = S \cup \{x \mapsto x\} \end{pmatrix} \quad \textbf{provided}\ x \notin A$$

In this case **HP9** guarantees that the output alphabet includes x', corresponding to the new variable x, which is now in A.

$$end(x) \mathrel{\widehat{=}} \mathbf{HPI} \circ \mathbf{HP9} \begin{pmatrix} A' = A \setminus (x\infty \cup \{\, x \,\}) \wedge \\ V' = (\, x\infty \cup \{\, x \,\}) \lhd V) \wedge \\ S' = (\, x\infty \cup \{\, x \,\}) \lhd S \rhd (x\infty \cup \{\, x \,\}) \end{pmatrix}$$

These constructs correspond to the (**var** x) and (**end** x) operators of the UTP; they do not create a variable block.

7 Conclusions

We have presented a UTP theory of programs with variables whose object values and their attributes may share locations. We capture an abstract memory model of a modern object-oriented language based on (mutually) recursive records.

These have also been considered by Naumann in the context of higher-order imperative programs and a weakest precondition semantics [19]. In that work, many of the concerns are related to record types, and the possibility of their extension, as achieved by class inheritance in object-oriented languages. Here, we are only concerned with record values. We propose to handle the issue of inheritance separately, in a theory of classes with a copy semantics [25].

The idea of avoiding the use of explicit locations was first considered in [3] for an Algol-like language. The motivation was the definition of a fully abstract semantics, which does not distinguish programs that allocate variables to different positions in memory. In that work, sharing is recorded by a function that maps each variable to the set of variables that share its location; a healthiness condition ensures that variables in the same location have the same value. A stack of functions is used to handle nested variable blocks and redeclaration. We do not consider redeclarations, but we handle the presence of record variables, and sharing between record components, not only variables.

Hoare & He present in [12] a theory of pointers and objects using an analogy with process algebras. They draw attention to the similarities between pointer structures, automata and processes, and use of trace semantics and bisimulation in discussing pointers. They use a graph model based on a trace semantics: a set of sets of traces, each set of traces describing the paths that may be used to access a particular object. This work, however, stops short of providing a specification or refinement framework for pointer programs. We take the view that pointer structures are not just like automata, they are automata; this leads naturally to the view that updatable pointer structures are updatable automata. We handle the correspondence between the values of object variables and attribute accesses, and the sharing structure of these variables and their components in the unified context of a programming theory. To manage complexity, we use healthiness conditions to factor out basic properties from definitions.

Work on separation logic [23] and local reasoning [20] also provided inspiration for our work. These approaches establish a system of proof rules to address the frame problem. When a change is made to a data structure, variables not affected by the change maintain their values; standard approaches to reasoning require explicit invariants for every variable that does not change its value, with a large overhead in specification and reasoning. Any effective theory must address this problem in some way. Our work builds a semantic view of pointers which we believe supports derived rules of inference that mirror those used in local reasoning. At the moment, we are concerned with local reasoning, rather than separation logic, because we want to work with classical logic to simplify connection with other UTP theories.

Chen and Sanders' work [8] lifts and extends combinators of separation logic to handle modularisation and abstraction at the levels of specification and design.

This work is based on the model in [12], and as such it also does not consider the relationship between the pointer structure and the values of programming variables and attribute accesses. On the other hand, they present a number of operators and laws to support reasoning about the graph structures.

Möller [18] uses relations to represent pointer graphs; the extension of this work to Kleene algebra [9] provides a natural formalism to capture self-referential structures. Since we adopt the automaton view of pointers, there is inevitably a correspondence with Kleene algebras: all our constructions could be expressed in terms of an updatable Kleene algebra. The difference is one of perspective: if we were concerned with data structures, our work would provide results similar to Möller's, although expressed in terms of automata. We aim, however, at a theory for specification and refinement, so our model is directly linked to the UTP framework via the healthiness conditions that connect the updatable automaton to the program variables.

Bakewell, Plump, and Runciman [2] suggest the explicit use of a graph model to reason about pointers. With this perspective, it is natural to talk about invariants of the graph and about pointer structure manipulations in terms of invariant preservation. This is at the heart of the work in [2], in which a set of invariants are defined for pointer graphs and program safety is defined in terms of preservation of suitable sets of these invariants. The technique would be directly applicable to automaton models, including ours.

The refinement calculus for object systems (rCOS) [14] and TCOZ [22], an object-oriented language that combines Object-Z [26], CSP, and timing constructs, have been given a UTP semantics. In both works, objects have identities which are abstract records of their location in memory. Object identities refer explicitly to storage and, as already discussed, prevent full abstraction.

In the short term, we plan to investigate refinement laws of our theory, and explore its power to reason about pointer programs in general, and data structures and algorithms typically used in object-oriented languages in particular. After that, we want to go a step further in our combination of theories and consider a theory of reactive designs with pointers.

References

1. R. J. Back, X. Fan, and V. Preoteasa. Reasoning about Pointers in Refinement Calculus. In *APSEC 2003*, page 425. IEEE Computer Society, 2003.
2. A. Bakewell, D. Plump, and C. Runciman. Specifying Pointer Structures by Graph Reduction. In *Applications of Graph Transformations with Industrial Relevance*, v. 3062 of *LNCS*, pages 30 – 44, 2006.
3. S. D. Brookes. A Fully Abstract Semantics and a Proof System for an Algol-like Language with Sharing. In A. Melton, editor, *Mathematical Foundations of Programming Semantics*, v. 239 of *LNCS*, pages 59 – 100. Springer, 1985.
4. R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23 – 50, 1972.
5. A. L. C. Cavalcanti, W. Harwood, and J. C. P. Woodcock. Pointers and Records in the Unifying Theories of Programming. In *UTP'06*, v. 4010 of *LNCS*, pages 200 – 216. Springer, 2006.

6. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Unifying Classes and Processes. *SoSyM*, 4(3):277 – 296, 2005.
7. A. L. C. Cavalcanti and J. C. P. Woodcock. A Tutorial Introduction to CSP in Unifying Theories of Programming. In *Refinement Techniques in Software Engineering*, v. 3167 of *LNCS*, pages 220 – 268. Springer, 2006.
8. Y. Chen and J. Sanders. Compositional Reasoning for Pointer Structures. In *MPC*, v. 4014 of *LNCS*, pages 115 – 139. Springer, 2006.
9. J. Desharnais, B. Möller, and G. Struth. Modal Kleene Algebra and applications a survey. *Methods in Computer Science*, (1):93 – 131, 2004.
10. W. Harwood, A. L. C. Cavalcanti, and J. C. P. Woodcock. A Model of Pointers for the Unifying Theories of Programming – Extended Version. Technical report, University of York, Department of Computer Science, UK, 2008. www-users.cs.york.ac.uk/˜alcc/publications/HCW08.pdf.
11. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
12. C. A. R. Hoare and He Jifeng. A trace model for pointers and objects. *Programming methodology*, pages 223 – 245, 2003.
13. Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*. ACM Press, 2001.
14. Z. Liu, J. He, and X. Li. rCOS: Refinement of Component and Object Systems. In *Formal Methods for Components and Objects*, v. 3657 of *LNCS*. Springer Verlag, 1994.
15. B. Meyer. *Eiffel: the language*. Prentice-Hall, 1992.
16. B. Meyer. Towards practical proofs of class correctness. In *ZB 2003*, v. 2651 of *LNCS*, pages 359 – 387. Springer, 2003.
17. R. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, 1976.
18. B. Möller. Calculating with pointer structures. In *IFIP TC 2 WG 2.1 International Workshop on Algorithmic Languages and Calculi*, pages 24 – 48. Chapman & Hall, Ltd., 1997.
19. D. A. Naumann. Predicate Transformer Semantics of a Higher Order Imperative Language with Record Subtypes. *SCP*, 41(1):1 – 51, 2001.
20. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, v. 2142 of *LNCS*, pages 1 – 19. Springer, 2001.
21. R. F. Paige and J. S. Ostroff. ERC – An object-oriented refinement calculus for Eiffel. *Formal Aspects of Computing*, 16(1):5, 2004.
22. S. Qin, J. S. Dong, and W. N. Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In *FME2003*, v. 2805 of *LNCS*, pages 321 – 340. Springer, 2003.
23. J. Reynolds. Separation logic: a logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*, pages 55 – 74. IEEE Press, 2002.
24. J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*. Palgrave, 2001.
25. T. L. V. L. Santos, A. L. C. Cavalcanti, and A. C. A. Sampaio. Object Orientation in the UTP. In *UTP'06*, v. 4010 of *LNCS*, pages 18 – 37. Springer, 2006.
26. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 1999.