# Pointers and Records in the Unifying Theories of Programming

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

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

**Abstract.** We present a theory of pointers and records that provides a representation for objects and sharing in languages like Java and C++. Our approach to pointers is based on Paige's entity groups, which give an abstract view of storage as equivalence classes of variables that share the same memory location. We first define our theory as a restriction of the general theory of relations, and, as a consequence, it does not distinguish between terminating and non-terminating programs. Therefore, we link it with the theory of designs, providing a foundation for reasoning about total correctness of pointer-based sequential programs. Our work is a step towards the semantics of an object-oriented language that also integrates constructs for specifying state-rich and concurrent systems.

**Keywords.** semantics, refinement, relations, object models.

## 1 Introduction

Interest in reasoning about pointer programs is not recent [3], and has been renewed by the importance of sharing in object-oriented languages [1, 11]. Most semantic models of pointers use indexes to represent memory locations or embed a heap [8, 15]. Modern object-oriented languages, however, do not encourage or directly support manipulation of the memory.

In this paper, we present a theory for pointers based on the model of entity groups presented in [13] to formalise rules of a refinement calculus for Eiffel [10]. In that work, the complications of an explicit model of the memory are avoided; instead, each entity (variable) is associated with the set of variables that share its location (entity group). Using this model, the Eiffel semantics for object creation, reference assignment, and call is formalised.

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 the language *OhCircus* presented

[*] Also, a Visiting Research Fellow at the University of Kent Computing Laboratory, Canterbury Kent, UK.

in [4]; it is a combination of Z [20] and CSP [16], with object-oriented constructs in the style of Java, including inheritance, subtyping, and null values. Since *OhCircus* combines constructs from several programming theories, the UTP is a very appropriate choice for its semantic model.

Following the UTP style, we are concentrating on the individual aspects of the *OhCircus* semantics separately. The theory that we present here provides a reference semantics for a language with variables whose values are objects: recursive records. It will be integrated to the copy semantics of *OhCircus*.

The program in Figure 1 illustrates the sort of concepts in which we are interested. This program compacts a list $l$, by sharing references to equal values. The type $List$ of $l$ can be defined as: $List ::= (label : \mathbb{Z};\ next : List)$. This is a recursive labelled record with two fields: $label$ and $next$. The assignments in Figure 1 are pointer assignments, and the equalities are value equalities. In this example we use a reasonably standard programming notation involving **while** and **if** commands, but in our theory we use the notation adopted in the UTP.

```
var $p \bullet p := l$;
  while $p \neq$ null do
    var $q \bullet q := p.next$;
      while $q \neq$ null do
        if $q.label = p.label$ then $q.label := p.label$ fi;
        $q := q.next$
      od;
      $p := p.next$
  od
```

**Fig. 1.** Compacting a list $l$

We assume that all values, including primitive values, have a location; variables are names of locations. We are not interested in the particular locations of variables and values, but on whether two (or more) variables are different names for the same location or not. A healthiness condition guarantees that variables that denote the same location have the same value.

In the next section we present our theory: its alphabet and its healthiness conditions. Section 3 revisits the semantics of assignment and variable blocks, and establishes the closedness of our theory. In Section 4 we explore the link to the theory of designs; the combined theory supports reasoning about total correctness of pointer programs. Finally, in Section 5 we summarise our results, and consider some related and future work.

## 2 Relational pointer theory

In our work, we consider recursive data types $d_i$ defined by a set of recursive equations of the form $d_i = \langle\!\langle f_1 : d_1, ...., f_n : d_n \rangle\!\rangle \mid null$ or $d_i = s$, where $s$ is a simple set and the $f_j$'s are field names. We define the predicate $\mathsf{field}(f, d_i)$ to mean that $f$ is a field of the data type $d_i$. In our example, the definition of *List* is a shorthand for $List = \langle\!\langle label : Z;\ next : List \rangle\!\rangle \mid null$. These recursive records are enough to model object values in a language like Java.

As with the general theory of relations of the UTP, the alphabet of our theory of pointer relations includes the programming variables and their dashed counterparts. Their values, however, are elements of recursive data types.

If the value of a variable $x$ is a record with a field called $y$, we can use the name $x.y$ to refer to the value of this field: the dot notation is a field selector. If $x.y$ is again a record, we can refer to its $z$ field as $x.y.z$, and so on. We refer to both simple names (of programming variables) and such compound names formed using the field selector, as paths; the set *Path* contains all paths.

Our theory also includes two extra variables $pg$ and $pg'$; they are path groups: sets of groups (sets) of paths. Two paths that share the same location are in the same group. Path groups correspond to the entity groups in [13].

In the next section, we introduce additional notation related to paths. Later on, in Section 2.2, we define the healthiness conditions of our theory.

## 2.1 Paths

Given an observational variable $x$, we use $'x$ to refer to its name. References to $x$ itself are interpreted to stand for the value of $x$, as usual in the UTP.

We use meta variables $p$ and $q$ to refer to paths; we use subscripts if we need extra variables. Given a path $p$, its root is $p$ itself, if $p$ is a simple name, or $'x$, if it is of the form $'x.q$. In this latter case, $q$ is called the extension of $p$. We refer to these as $\mathsf{root}(p)$ and $\mathsf{ext}(p)$. The extension of a variable is empty.

In general, for paths $p$ and $q$, we call $p.q$ an extension of $p$ by $q$. The path $p.q$ is said to be a descendant of $p$. For any two paths $p$ and $q$, we write $p \prec q$ when $p$ is a descendant of $q$. Given a set of paths $\pi$ we define the set of its descendants as follows.

$$\mathsf{desc}(\pi) \mathrel{\widehat=} \{\, p \mid \exists\, q : \pi \bullet p \prec q \,\}$$

We introduce two meta functions: $\theta$ and $\delta$. The function $\theta$ is inspired by the Z $\theta$-notation. Given an alphabet $A$, and a path $p$, $\theta_A(p)$ gives the value of $p$, if its root $'x$ is in $A$ and $p$ is an appropriate reference to a field of $x$.

$$\theta_A('x) = x,\ \textbf{provided}\ 'x \in A$$

$$\theta_A(p.f) = v.f,\ \textbf{provided}\ \theta_A(p) = v \wedge v \in d_i \wedge \mathsf{field}(f, d_i)$$

We also introduce decorated versions of $\theta$. For example $\theta'_A$ is defined as follows.

$$\theta'_A('x) = x',\ \textbf{provided}\ 'x \in A$$

$$\theta_A(p.f) = v.f,\ \textbf{provided}\ \theta'_A(p) = v \wedge v \in d_i \wedge \mathsf{field}(f, d_i)$$

Other decorations can also be used. The important point is that the domain of $\theta$ is always a set of undecorated variable names, along with some of their descendants, whether $\theta$ is decorated or not. If we decorate $\theta$, however, these paths are associated to the value of the similarly decorated path. Of course, the decoration of a path is reflected in its root; for example $(x.y)' = x'.y$.

The set $\delta_{A,i}(p)$ includes all paths for $p$ with $i$ extra field selectors. A path for another path $p$ is either $p$ itself or a descendant of $p$, and, most importantly, it has a value, as defined by $\theta$.

$$\delta_{A,0}(p) = \{\, p \,\},\ \textbf{provided}\ p \in \operatorname{dom}\theta.$$

$$\delta_{A,n+1}(p) = \{\, q.f \mid q \in \delta_{A,n}(p) \land \theta_A(q) \in d_i \land \mathsf{field}(f, d_i) \,\}$$

In general, $\delta_A(x)$ is the set of all paths for $x$.

$$\delta_A(x) = \bigcup_i \delta_{A,i}(x)$$

Given a set $\pi$ of paths, $\mathit{\Delta}(\pi)$ is the set of paths for the paths in $\pi$.

$$\mathit{\Delta}_A(\pi) = \{\, p \mid \exists\, x : \pi \bullet p \in \delta_A(x) \,\}$$

In summary, the descendants of a variable $x$ are all path names that can be built using $'x$ as a root. The paths for $x$, on the other hand, are $x$ itself, and all descendants that can be meaningfully used to access a component of the record value of $x$, if any. Both notions generalise to paths in general.

Generally, we will drop the alphabet subscript from the above functions when they can be inferred from context.

## 2.2 Healthiness conditions

We need healthiness conditions to establish the relationship between the values of the variables and the path groups in $pg$ and $pg'$. First of all, we have a healthiness condition **HP1** to guarantee that the path group $pg$ partitions all paths of the variables of the program.

$$\textbf{HP1}\quad P = P \land pg\ \mathsf{partition}\ \mathit{\Delta}(var\alpha P)$$

In the UTP, the set $in\alpha P$ includes all the undashed variables in the alphabet of $P$. We define $var\alpha P = in\alpha P \setminus \{\, pg \,\}$ to include all the undecorated programming variables in the alphabet of $P$.

We use **HP1** to name the function $\textbf{HP1}(P) \mathrel{\widehat=} P \land pg\ \mathsf{partition}\ \mathit{\Delta}(var\alpha P)$ as well. The **HP1**-healthy relations are the fixed points of **HP1**. As usual in the UTP, we adopt the same sort of convention in relation to the definitions of the other healthiness conditions in the sequel.

The second property we require is that the path group is well structured, so that if any group contains a pair of paths $p_1$ and $p_2$, then if these paths are extended in the same way, there is a group containing both extensions.

$$\textsf{\textbf{HP2}} \quad P = P \land \forall\, g_1 : pg;\; p_1, p_2 : g_1;\; p : Path \mid \{\, p_1.p, p_2.p \,\} \subseteq \Delta(var\alpha P) \bullet (\exists\, g_2 : pg \bullet \{\, p_1.p, p_2.p \,\} \subseteq g_2)$$

This reflects the fact that if $p_1$ and $p_2$ are different names for the same location, then accesses to their components are also accesses to the same location.

Finally, all paths in the same group must have the same value.

$$\textsf{\textbf{HP3}} \quad P = P \land \forall\, g : pg;\; p_1, p_2 : g \bullet \theta_{var\alpha P}\ p_1 = \theta_{var\alpha P}\ p_2$$

We use the $\theta$ function to determine the values of the paths $p_1$ and $p_2$. The $\theta$ function is partial: it is only defined for valid applications of the field selector operator. For example, $\theta(x.f)$ is not defined if the value of $x$ is *null*. Therefore, by requiring that $p_1$ and $p_2$ have the same image under $\theta$, we not only require that they have the same value, but also that they are valid paths.

The healthiness conditions HP1, HP2 and HP3 impose conditions on the input path group *pg*; HP4, HP5, and HP6 below impose the same conditions on the output path group $pg'$.

$$\textsf{\textbf{HP4}} \quad P = P \land pg'\ \textsf{partition}\ \Delta(var\alpha P)$$

It is a consequence of **HP4** that $pg'$, in the same way as *pg*, includes only undecorated variable names. This is important to avoid the need to change the definition of a sequence $P;\ Q$ to match the value of $pg'$ defined by $P$ to the value of *pg* used by $Q$.

$$\textsf{\textbf{HP5}} \quad P = P \land \forall\, g_1 : pg';\; p_1, p_2 : g_1;\; p : Path \mid \{\, p_1.p, p_2.p \,\} \subseteq \Delta(var\alpha P) \bullet (\exists\, g_2 : pg' \bullet \{\, p_1.p, p_2.p \,\} \subseteq g_2)$$

$$\textsf{\textbf{HP6}} \quad P = P \land \forall\, g : pg';\; p_1, p_2 : g \bullet \theta'_{\alpha P}\ p_1 = \theta'_{\alpha P}\ p_2$$

In the definition of **HP6**, we use a decorated version of $\theta$. The paths in $pg'$ are not decorated, but $\theta'$ gives the values of the primed variables.

The set of healthiness conditions can be simplified by noting that conditions **HP3-6** can be replaced by the condition below.

$$\textsf{\textbf{HP7}} \quad P;\ I\!I_p = P$$

The program $I\!I_p$ is the **HP1-3**-healthy identity relation, which we denote by $I\!I_r$ to avoid confusion.

$$I\!I_p \mathrel{\widehat=} \textsf{\textbf{HP1}} \circ \textsf{\textbf{HP2}} \circ \textsf{\textbf{HP3}}(I\!I_r)$$

The theorems below establish that the two sets of healthiness conditions are indeed interchangeable.

**Theorem 1.** *Every relation $R$ that is* **HP1**-**3**-*healthy and* **HP7**-*healthy is also* **HP4**-**6**-*healthy.*

**Theorem 2.** *Every relation $R$ that is* **HP1**-**6**-*healthy is also* **HP7**-*healthy.*

A yet more concise way of characterising the healthy pointer relations is justified by the following theorem. It establishes that we can use just the healthiness condition below.

**HP8** $P = I\!I_p;\ P;\ I\!I_p$

**Theorem 3.** *A pointer relation $R$ is healthy if, and only if, it is* **HP8**-*healthy.*

This result is a consequence of the fact that our healthiness conditions are restrictions on the initial and after state of a relation, but not on the transitions that they describe.

This also allows us to prove a further useful theorem.

**Theorem 4.** *For any pointer relation $P$,* **HP8**$(P)$ *is the weakest healthy pointer relation characterised by $P$: $P \sqsubseteq I\!I_p;\ P;\ I\!I_p$, and for every healthy $Q$ such that $P \sqsubseteq Q$, we have $I\!I_p;\ P;\ I\!I_p \sqsubseteq Q$.*

*Proof.*

$$\begin{array}{ll} P \sqsubseteq Q & \textit{monotonicity of sequence} \\ \Rightarrow I\!I_p;\ P;\ I\!I_p \sqsubseteq I\!I_p;\ Q;\ I\!I_p & \textit{healthiness of } Q \\ = I\!I_p;\ P;\ I\!I_p \sqsubseteq Q & \square \end{array}$$

This justifies the specification of pointer relations by defining unhealthy relations and using **HP8** to make it healthy.

## 3 Programming constructs

In this section, we revisit the semantics of (value) assignment already in the UTP, and introduce a new form of assignment: pointer assignment. For each form of assignment, there is a corresponding notion of equality.

### 3.1 Equality

Our two notions of equality are standard equality $\_ =_p \_$ and pointer equality $\_ ==\_ \_$. Standard equality equates values and pointer equality equates storage locations.

Value equality is defined in terms of the $\theta$ function.

$$p_1 =_p p_2 \mathrel{\widehat{=}} \theta(p_1) = \theta(p_2)$$

The paths $p_1$ and $p_2$ are required to be valid, that is, in the domain of $\theta$, and have the same value.

Pointer equality is defined with respect to the path group which models storage.

$$p_1 ==_{pg} p_2 \mathrel{\widehat{=}} \exists\, g : pg \bullet \{\, p_1, p_2 \,\} \subseteq g$$

These two equalities reflect the same distinction found in Lisp, where EQUAL compares values and EQ compares pointers. On the other hand, this is slightly in contrast with Java, where $\_ == \_$ compare values, but the values of objects are locations. In our language, every value has a location, and we assume that literal values have fresh locations. To write the Java expression $e_1 == e_2$ in our language, we have to determine the type(s) of $e_1$ (and $e_2$). If they have primitive types, we write $e_1 = e_2$; if not, we keep the $\_ == \_$.

In our theory, if either of $x$ or $y$ is a primed name, then $x == y$ is going to be false, whether the extra parameter is $pg$ or $pg'$. This is because, as already mentioned, they only hold undashed names.

## 3.2 Assignment

The first form of assignment $p_1 := p_2$ that we consider is that already available in the UTP, which assigns the value of $p_2$ to $p_1$, and, consequently, to all other paths in its group. The second is $p_1 :== p_2$, which makes $p_1$ to become another name for the location of $p_2$; in our context, this assignment alters the storage model by merging the path groups containing $p_1$ and $p_2$.

Both assignments are alphabetised; they take a set $A$ of programming variables as a parameter.

$$\alpha(p_1 :=_A p_2) = \alpha(p_1 :=_A p_2) = A \cup A' \cup \{\, pg, pg' \,\}$$

Alphabets are left implicit whenever convenient.

*Value assignment* As already said, the value assignment $p_1 := p_2$ has the side effect of altering the value of all paths that share the storage location of $p_1$. As a consequence, the value of all their descendants are also changed. No other paths have their value changed.

In terms of memory usage, there are two issues. Firstly, if a component $x.f$ of $x$ shares location with a path $p$, and we assign a new value $p_2$ to $x$, then $x.f$ takes on a new value as well, that of $p_2.f$, if this is well defined. Therefore, $x.f$ and $p$ cease to share their location. This means that all the descendants of $x$ have to be eliminated from the path groups in which they are.

Secondly, a value assignment duplicates a value and potentially requires extra storage. For example, the assignment $x.f := y$ makes the value of $x.f$, and of all the paths that share its location, to become that of $y$; the locations of $x.f$ and $y$, however, are not changed. Moreover, if the value of $y$ is itself a record, with a field $g$, then $x.f.g$ and $y.g$ have the same value, but different locations. If the location $x.f.g$ did not exist before, because, for example, $x.f$ had value *null* before the assignment, a new location is created.

We define this behavior by defining a general notion of assignment and then making it healthy using **HP8**.

$$\begin{array}{l}
\mathsf{assignV}(p_1, p_2, A) \mathrel{\widehat=} \\
\quad \forall\, q : \mathsf{group}(p_1, pg) \bullet \mathsf{update}(\mathsf{root}(q), \mathsf{root}(q)', \mathsf{ext}(q), p_2) \land \\
\quad \forall\, n : A \mid (\neg\, \exists\, p : \mathsf{group}(p_1, pg) \bullet n =_n \mathsf{root}(p)) \bullet n' = n \land \\
\quad pg' = \mathsf{remove}(pg, \mathsf{group}(p_1, pg)) \cup \\
\qquad\qquad \{\, q_1 : \Delta(p_2);\ q_2 : Path \mid q_1 =_n p_2.q_2 \bullet \{\, q_3 : \mathsf{group}(p_1, pg) \bullet q_3.q_2 \,\}\,\}
\end{array}$$

The function $\mathsf{group}(x, pg)$ selects the path group of $pg$ that contains $x$.

$$\mathsf{group}(x, pg) \mathrel{\widehat=} \iota\ g : pg \mid x \in g$$

The $\iota$ expression $\iota\ o : S \mid p(o)$ gives a definite description of an object $o$ of a set $S$ that satisfies a constraint $p(o)$; it is defined only when $o$ exists and is unique. It is identical to the Z $\mu$ operator; we do not use $\mu$ to avoid confusion with the least fixed point operator of the UTP.

The first conjunct in the definition of $\mathsf{assignV}(p_1, p_2, A)$ defines the new value of all the paths in the group of $p_1$. For each of them, including $p_1$ itself, we change the value of the variable at its root. This is because, changing the value of $x.f$ really corresponds to changing the value of $x$: its $f$ field takes on a new value, and all the others keep the same value.

The operator $\mathsf{update}(x, y, p, q)$ defines the value of $y$ as the result of updating the value of $x$ to change the value of its component $x.p$ to be that of the path $q$. All the other components of $y$ have the same value of the corresponding component of $x$.

$$\begin{array}{l}
\mathsf{update}(x, y, p, q) \mathrel{\widehat=} \\
\quad \forall\, p_1 : \Delta(x) \bullet (p_1 =_n x.p \Rightarrow y.p =_p q) \land \\
\qquad\qquad\qquad ((p_1 \neq_n x.p \land \neg\, (p_1 \prec x.p)) \Rightarrow y.\mathsf{ext}(p_1) =_p p_1)
\end{array}$$

To define $\mathsf{update}(x, y, p, q)$, we consider each of the descendant paths $p_1$ of $x$. For the descendant $x.p$, the corresponding value of $y.p$ is that of $q$. For the other descendants $p_1$, if they are not descendants of $x.p$, the value of the corresponding component of $y$ is that of $p_1$ itself, which is a component of $x$. If they are a descendant of $x.p$, by defining $y.p$, we have already defined its value.

The equality operator $\_ =_n \_$ compares paths for syntactic equality. In the case of simple names, it compares the names of the variables, instead of their values.

The second conjunct in the definition of $\mathsf{assignV}(p_1, p_2, A)$ defines the value of the variables that are not affected by the assignment: those that are not roots of paths in the group of $p_1$. As already said, the value of the paths in the group of $p_1$ is defined by the $\mathsf{update}$ function. In doing so, we also determine the value of all the descendants of the roots of those paths, as explained above.

The third conjunct in the definition of $\mathsf{assignV}(p_1, p_2, A)$ defines the value of $pg'$. The function $\mathsf{remove}(pg, \pi)$ defines the set of path groups obtained by removing all descendants of the paths in $\pi$ from the groups in $pg$. If a group of $pg$ contains only descendants of $\pi$, it becomes empty, and should be excluded.

$$\mathsf{remove}(pg, \pi) \mathrel{\widehat=} \{\, g : pg \mid \neg\, g \subseteq \mathsf{desc}(\pi) \bullet g \setminus \mathsf{desc}(\pi) \,\}$$

The use of $\mathsf{remove}(pg, \mathsf{group}(p_1, pg))$ accounts for the first issue discussed above in relation to memory usage in the behaviour of the assignment $p_1 := p_2$; namely, the sharing information about all the descendants of the assigned path changes. The duplication of the assigned value is taken into account by requiring that $pg'$ includes new path groups $\{\, q_3 : \mathsf{group}(p_1, pg) \bullet q_3.q_2 \,\}$, for each extension $q_2$ of the descendants of $p_2$.

Finally, the definition of assignment is as follows.

$$p_1 :=_A p_2 \mathrel{\widehat=} \mathbf{HP8}(\mathit{assignV}(p_1, p_2, A))$$

An interesting observation is that we only need to compose $I\!I_p$ on the left of $\mathit{assignV}(p_1, p_2, A)$ to make it healthy.

**Theorem 5.** $p_1 :=_A p_2 = I\!I_p;\ \mathit{assignV}(p_1, p_2, A)$

This is because the path group $pg'$ defined by $\mathsf{assignV}$ satisfies the requirements of our healthiness conditions. What it does not enforce is that $pg$ is suitable.

*Pointer assignment* The second form of assignment, $p_1 :==_A p_2$, makes $p_1$ to share the location of $p_2$. As a consequence, the value of $p_1$ is also changed to that of $p_2$. Moreover, by changing the location of $p_1$ to that of $p_2$, we implicitly change the location of all descendants of $p_1$, and their values. In our model, we remove them all from their current path groups, and, for each well defined descendant of $p_2$, we insert a corresponding descendant of $p_1$ in its group. We use the same style of construction as for value assignment, using **HP8** to ensure healthiness.

$$\begin{aligned} p_1 :==_A p_2 \mathrel{\widehat=} \mathbf{H8}(&p_1' =_p p_2 \land (\forall\, p : \textstyle\bigcup pg' \mid pg \notin \Delta(p_1) \bullet p' =_p p) \land \\ &pg' = \mathsf{add}(p_1, p_2, \mathsf{purge}(p_1, pg))) \end{aligned}$$

The first conjunct of this definition determines the new value of $p_1$, and implicitly that of all its descendants. It also establishes that the value of all other paths are not changed.

The second conjunct of the above definition determines the new value of $pg$. We use a strengthened $\mathsf{remove}$ operator to state that both $p_1$ and all its descendants need to be removed from the original path groups.

$$\mathsf{purge}(p, pg) \mathrel{\widehat=} \{\, g : pg \mid \neg\, g \subseteq (\mathsf{desc}(p) \cup \{\, p \,\}) \bullet g \setminus (\mathsf{desc}(p) \cup \{\, p \,\}) \,\}$$

Next, we use a function $\mathsf{add}$ to define that $p_1$ itself and its descendants need to be inserted back into the corresponding groups of $p_2$ and its descendants.

$$\begin{aligned} &\mathsf{add}(p_1, p_2, pg) \mathrel{\widehat=} \\ &\quad \{\, g : pg \bullet g \cup \{\, p : g;\ q : \mathit{Path} \mid p =_n p_2.q \bullet p_1.q \,\} \cup \{\, p : g \mid p =_n p_2 \bullet p_1 \,\} \,\} \end{aligned}$$

Again, our use of **HP8** in the definition of $p_1 :==_A p_2$ is required only to enforce that assignments are only defined for healthy path groups $pg$.

## 3.3 Variable blocks

The declaration of a variable requires its inclusion in the set of path groups: new singleton groups containing the new variable and its descendants should be defined. Also, ending the scope of a variable entails in removing it and its descendants from the path groups. Therefore, we redefine **var** $x$ and **end** $x$.

$$\mathbf{var}_A\ x \mathrel{\widehat{=}} \mathsf{\mathbf{HP8}}(\forall\, n : A \bullet n' =_p n \wedge pg' = pg \cup \{\, p : \varDelta('x) \bullet \{\, p \,\}\,\})$$

The alphabet of the variable declaration includes the new variable.

$$\alpha(\mathbf{var}_A\ x) = A \cup A' \cup \{\, x' \,\} \cup \{\, pg, pg' \,\}$$

This is just as in the UTP definition for the alphabet of **var**, except for the extra observational variables $pg$ and $pg'$.

To define **end** $x$, we use the function purge introduced in the previous section.

$$\mathbf{end}_A\ x \mathrel{\widehat{=}} \mathsf{\mathbf{HP8}}(\forall\, n : A \bullet n' =_p n \wedge pg' = \mathsf{purge}(x, pg))$$

The alphabet definition is similar to that of **var** $x$.

$$\alpha(\mathbf{end}_A\ x) = A \cup \{\, x \,\} \cup A' \cup \{\, pg, pg' \,\}$$

The proof of laws is in our agenda for future work.

## 3.4 Closure properties

In this section, we prove that the programming operators are closed. In other words, when applied to healthy relations, they result in healthy relations.

**Theorem 6.** *If the relations $P$ and $Q$ are healthy, then so is $P;\ Q$.*

*Proof.*

$$\begin{array}{lr}
\mathit{II}_p;\ P;\ Q;\ \mathit{II}_p & \textit{P and Q are healthy} \\
= \mathit{II}_p;\ \mathit{II}_p;\ P;\ \mathit{II}_p;\ \mathit{II}_p;\ Q;\ \mathit{II}_p;\ \mathit{II}_p & \mathit{II}_p;\ \mathit{II}_p = \mathit{II}_p \\
= P;\ Q & \square
\end{array}$$

**Theorem 7.** *If the relations $P$ and $Q$ are healthy, then so is $P \vee Q$.*

*Proof.*

$$\begin{array}{lr}
\mathit{II}_p;\ (P \vee Q);\ \mathit{II}_p & \textit{property of sequence and } \vee \\
= \mathit{II}_p;\ P;\ \mathit{II}_p \vee \mathit{II}_p;\ Q;\ \mathit{II}_p & \textit{P and Q are healthy} \\
= P \vee Q & \square
\end{array}$$

**Theorem 8.** *If the relations $P$ and $Q$ are healthy, then so is $P \wedge Q$.*

*Proof.*

$$
\begin{array}{ll}
\mathit{II}_p;\ (P \wedge Q);\ \mathit{II}_p & \textsf{\textit{property of }} \mathit{II}_p \\
= \mathit{II}_p;\ P;\ \mathit{II}_p \wedge \mathit{II}_p;\ Q;\ \mathit{II}_p & P \textsf{\textit{ and }} Q \textsf{\textit{ are healthy}} \\
= P \wedge Q & \square
\end{array}
$$

**Theorem 9.** *If relations $P$ and $Q$ are healthy, then so is $P \lhd b \rhd Q$.*

*Proof.*

Essentially the same. □

The set of healthy pointer relations is a complete lattice, since it is the image of monotonic and idempotent healthiness conditions [7].

**Theorem 10.** *If $F$ is built out of conjunctions, disjunctions, and sequences applied to healthy pointer relations, then*

$$\mu_p X \bullet F(X) = \mathbf{HP8}(\mu X \bullet F(X))$$

*where $\mu_p X \bullet F(X)$ is the least fixed point of $F$ in the lattice of healthy pointer relations.*

*Proof.* Follows from the closure theorems above, and from the fact that **HP8** is a monotonic idempotent that semi-commutes with the programming constructors [7]. □

This result states that a recursion is a healthy pointer relation, if its body is built out of pointer relations itself.

## 4 Pointer Designs

The theory of pointer relations does not distinguish between terminating and non-terminating programs. This distinction is made in the UTP by defining *designs*, a subclass of relations that satisfy two healthiness conditions (**H1** and **H2**). All design relations can be split into precondition/postcondition pairs, making them similar to specification statements in the refinement calculus.

In this section, we combine the theories of designs and pointers, thereby providing a foundation for a theory of total correctness for pointer-based sequential programs. First, we reproduce the definitions of the design theory that we need, then we define a Galois connection between our theory and designs. Finally, we introduce an extra healthiness condition of the combined theory.

## 4.1 Designs

The theory of designs include two extra boolean observational variables to record the start and the termination of a program: $ok$ and $ok'$. The monotonic idempotents used to define the healthiness conditions for designs can be defined as follows, where $P$ is a relation with alphabet $\{ok, ok', v, v'\}$.

$$\mathbf{H1}(P) \mathrel{\widehat{=}} ok \Rightarrow P$$

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

The variable $ok$ records the observation that the program has been started; the variable $ok'$ records the observation that the program terminated. 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 composition of **H1** and **H2** is named **H**.

The above formulation of **H2** is different from that in [7], but in [19], we prove that it is equivalent.

## 4.2 Pointer relations and designs

The theory of pointer relations is stronger than the theory of designs. This is because on abortion, a design provides no guarantees; however, a pointer relation still requires the properties of $pg$ to hold. This seems to be compatible with the reality of pointer programs: the information held in $pg$ (and $pg'$) is related to the physical constraints over variables that share locations, and these constraints are not suspended when the program aborts. In this case, the final values of the variables are arbitrary, but those that share the same location will still have the same value, for instance.

Therefore, to combine the theories of pointers and designs, we follow the approach used to combine the theory of reactive processes and designs. We take **HP8** as a link that maps a design to a pointer relation; the range of **HP8** characterises a subset of pointer relations: *pointer designs*. This is our proposed theory for total correctness of pointer programs.

First of all, for insight, we consider **HP8**$(\neg\, ok)$; this program is strictly stronger than $\neg\, ok$, which is the top of the lattice of designs. This property prevents **H1** from commuting exactly with **HP8**. In general, we have the following result.

$$\begin{array}{lr}
\quad \mathbf{HP8} \circ \mathbf{H1}(P) & \mathbf{H1} \\
= \mathbf{HP8}(ok \Rightarrow P) & \textit{propositional calculus, } \mathbf{HP8} \textit{ disjunctive} \\
= \mathbf{HP8}(\neg\, ok) \lor \mathbf{HP8}(P) & \mathbf{HP8}(\neg\, ok) \neq \neg\, ok \\
\neq \neg\, ok \lor \mathbf{HP8}(P) & \mathbf{H1} \\
= \mathbf{H1} \circ \mathbf{HP8}(P) & \Box
\end{array}$$

For this reason, the theory of pointer relations is disjoint from the theory of designs: a pointer relation cannot be a design, and *vice versa*. Instead, there is an approximate relationship between the two theories:

$$\textsf{\textbf{HP8}} \circ \textsf{\textbf{H}}(P) \sqsubseteq P$$

for pointer relation $P$. This relationship is a property of a Galois connection that translates between the two theories. In particular, it allows us to embed the theory of designs and its refinement calculus in the world of pointers.

*Galois connection* Let **S** and **T** both be partial orders; let $L$ be a function from **S** to **T**; and let $R$ be a function from **T** to **S**. The pair $(L, R)$ is a *Galois connection* if, for all $X \in \mathbf{S}$ and $Y \in \mathbf{T}$

$$Y \sqsubseteq L(X) \quad \textbf{iff} \quad R(Y) \sqsubseteq X$$

$L$ and $R$ are known as the left and right adjoints, respectively.

Our proof of the existence of a Galois connection relies on two simple lemmas about our healthiness conditions and refinement. First, a lemma concerning **H1**.

**Lemma 1 (H1-refinement).** *For any two relations $P$ and $Q$ with ok and ok′ in their alphabets,*

$$\textsf{\textbf{H1}}(P) \sqsubseteq \textsf{\textbf{H1}}(Q) \quad \textit{\textbf{iff}} \quad \textsf{\textbf{H1}}(P) \sqsubseteq Q$$

*Proof.*

$$\begin{array}{ll}
\;\;\textsf{\textbf{H1}}(P) \sqsubseteq \textsf{\textbf{H1}}(Q) & \textsf{\textit{refinement}} \\
= [\, \textsf{\textbf{H1}}(Q) \Rightarrow \textsf{\textbf{H1}}(P) \,] & \textsf{\textbf{H1}} \\
= [\, (ok \Rightarrow Q) \Rightarrow (ok \Rightarrow P) \,] & \textsf{\textit{propositional calculus}} \\
= [\, Q \Rightarrow (ok \Rightarrow P) \,] & \textsf{\textbf{H1}} \\
= [\, Q \Rightarrow \textsf{\textbf{H1}}(P) \,] & \textsf{\textit{refinement}} \\
= \textsf{\textbf{H1}}(P) \sqsubseteq Q & \square
\end{array}$$

This lemma lets us cancel an application of **H1** on the right-hand side of the refinement relation. This works because **H1**$(P)$ is a disjunction, and the cancelation strengths the implementation. Something similar can be done with **HP8**, but since **HP8**$(P)$ is a conjunction, the cancelation takes place on the specification side.

**Lemma 2 (HP8-refinement).** *For any two relations $P$ and $Q$ with pg and pg′ in their alphabets,*

$$P \sqsubseteq \textsf{\textbf{HP8}}(Q) \quad \textit{\textbf{iff}} \quad \textsf{\textbf{HP8}}(P) \sqsubseteq \textsf{\textbf{HP8}}(Q)$$

*Proof.*

$$\begin{array}{ll}
\;\;P \sqsubseteq \textsf{\textbf{HP8}}(Q) & \textsf{\textit{refinement}} \\
= [\, \textsf{\textbf{HP8}}(Q) \Rightarrow P \,] & \textsf{\textbf{HP8}}
\end{array}$$

$$\begin{array}{ll}
= [\,\mathit{I\!I}_P \;;\; Q \;;\; \mathit{I\!I}_P \Rightarrow P\,] & \textit{sequence} \\
= [\,(\exists\, v_0, v_1 \bullet \mathit{I\!I}_P[v_0/v'] \land Q[v_0, v_1/v, v'] \land \mathit{I\!I}_P[v_1/v]) \Rightarrow P\,] & \\
 & \textit{predicate calculus} \\
= [\,\mathit{I\!I}_P[v_0/v'] \land Q[v_0, v_1/v, v'] \land \mathit{I\!I}_P[v_1/v] \Rightarrow P\,] & \textit{predicate calculus} \\
= [\,\mathit{I\!I}_P[v_0/v'] \land Q[v_0, v_1/v, v'] \land \mathit{I\!I}_P[v_1/v] & \mathit{I\!I}_P\textit{, Leibnitz} \\
\qquad \Rightarrow \mathit{I\!I}_P[v_0/v'] \land P \land \mathit{I\!I}_P[v_1/v]\,] & \\
= [\,\mathit{I\!I}_P[v_0/v'] \land Q[v_0, v_1/v, v'] \land \mathit{I\!I}_P[v_1/v] & \textit{sequence} \\
\qquad \Rightarrow \mathit{I\!I}_P[v_0/v'] \land P[v_0, v_1/v, v'] \land \mathit{I\!I}_P[v_1/v]\,] & \\
= [\,\mathit{I\!I}_P \;;\; Q \;;\; \mathit{I\!I}_P \Rightarrow \mathit{I\!I}_P \;;\; P \;;\; \mathit{I\!I}_P\,] & \mathbf{HP8} \\
= [\,\mathbf{HP8}(Q) \Rightarrow \mathbf{HP8}(P)\,] & \textit{refinement} \\
= \mathbf{HP8}(P) \sqsubseteq \mathbf{HP8}(Q) & \Box
\end{array}$$

Applications of the above lemmas justify the main result of this section.

**Theorem 11.** *There is a Galois connection between designs and pointer relations, where* **HP8** *is the right adjoint and* **H** *is the left one.*

$$D \sqsubseteq \mathbf{H}(P) \quad \textit{iff} \quad \mathbf{HP8}(D) \sqsubseteq P$$

*Here, $D$ is a design whose alphabet contains $pg$ and $pg'$; and $P$ is a pointer relation whose alphabet contains $ok$ and $ok'$.*

*Proof*

$$\begin{array}{ll}
\quad D \sqsubseteq \mathbf{H1}(P) & \mathbf{H1}\textit{-refinement} \\
= D \sqsubseteq P & P \textit{ is } \mathbf{HP8} \\
= D \sqsubseteq \mathbf{HP8}(P) & \mathbf{HP8}\textit{-refinement} \\
= \mathbf{HP8}(D) \sqsubseteq \mathbf{HP8}(P) & P \textit{ is } \mathbf{HP8} \\
= \mathbf{HP8}(D) \sqsubseteq P & \Box
\end{array}$$

Proof of closedness of the programming operators in this new theory is simple.

## 4.3 Healthy pointer designs

The variables $ok$ and $ok'$ describe observations about initiation and termination of designs; they are certainly not program variables, and so must never be mentioned in program texts. In order to avoid confusion, a pointer design should isolate $ok$ in its own partition in $pg$ and $pg'$. This is a healthiness condition of our combined theory.

$$\mathbf{HD} \quad P = P \land \#group(ok, pg) = 1 \land \#group(ok, pg') = 1$$

Further exploration of the laws of this theory is left as future work.

## 5 Conclusions

We have presented a UTP theory for programs involving variables whose record values and their components may share locations. With this theory, we capture an abstract memory model of a modern object-oriented language.

In this work, we do not consider, for instance, the issues of classes and visibility in object-oriented languages, because our aim is the isolation of programming concepts. On the other hand, we do not have an explicit memory model that allows the definition of allocation and deallocation operations, because these are not needed to reason about object-oriented programs.

In order to reason about total correctness, we have investigated the theory that combines pointer relations and UTP designs. We established a formal link to translate between the two theories.

Recursive records have also been considered by Naumann in the context of higher-order imperative programs and a weakest precondition semantics [12]. 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 [17].

Hoare & He present a theory of pointers and objects using an analogy with process algebras [6]. They use a model of graphs based on a traces semantics [5], where a graph describes a snapshot of the entire heap during the execution of an object-oriented program. The heap is represented by a set of sets of traces: each set of traces describing the paths that may be used to access a particular object; this corresponds to our path groups. The main operator for updating the heap is known as *pointer swing*, and it updates the target of a pointer; this corresponds to our pointer assignment. In our work, we consider a model of pointers in the unified context of programming language models. We also handle the correspondence between the values of record variables and the sharing structure of these variables and their components. To manage complexity, we use healthiness conditions to factor out basic properties from definitions.

The idea of avoiding the use of locations to model pointers and sharing was first considered in [2] 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, groups are represented by functions in which each variable is associated with the set of variables that share its location. A healthiness condition ensures that variables in the same location have the same value: this corresponds to our **HP3**. A stack of functions is used in [2] to handle nested variable blocks and redeclaration. We do not consider the scope issues of redeclaration, but we handle the presence of record variables, and sharing between record components, not only variables.

The refinement calculus for object systems (rCOS) [9] is based on a UTP semantics for a relational object-oriented programming language that contains sub-typing, type casting, visibility, inheritance, dynamic binding, and polymorphism. Values in the language are drawn from primitive types or an infinite set of object references, augmented by information essential to the resolution of dynamic typing. By using object identities, the model refers explicitly to storage in an implementation-oriented way, and as a result is not fully abstract.

A UTP reference semantics for an object-oriented language has also been considered in [14]. In this case, we have a language that combines Object-Z [18], CSP, and timing constructs. Again, object values have identities which are abstract records of their location in memory.

For the kind of language in which we are interested, we believe that these identities are not needed, and the simpler model of the theory of path groups is enough. As already mentioned, our long-term goal is the definition of a reference semantics for *OhCircus*: an object-oriented language that also combines Z and CSP. Our approach, however, is based on the combination of models of isolated features of this rather rich language.

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.

## Acknowledgments

The authors are grateful to Richard Paige for detailed comments on a draft version of this paper; anonymous referees have also contributed with important suggestions. The work of Ana Cavalcanti is partially supported by the Royal Society and QinetiQ. Jim Woodcock also acknowledges QinetiQ's support.

## References

1. R. J. Back, X. Fan, and V. Preoteasa. Reasoning about Pointers in Refinement Calculus. In *10th Asia-Pacific Software Engineering Conference (APSEC 2003)*, page 425. IEEE Computer Society, 2003.
2. 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*, volume 239 of *Lecture Notes in Computer Science*, pages 59 – 100. Springer-Verlag, 1985.
3. R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
4. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Unifying Classes and Processes. *Software and System Modelling*, 4(3):277 – 296, 2005.
5. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
6. C. A. R. Hoare and H. Jifeng. A trace model for pointers and objects. pages 223–245, 2003.
7. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
8. Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*. ACM Press, 2001.

9. Z. Liu, J. He, and X. Li. rCOS: Refinement of Component and Object Systems. In F. .S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proceedings of Formal Methods for Components and Objects: FMCO 2004*, volume 3657 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.
10. B. Meyer. *Eiffel: the language*. Prentice-Hall, 1992.
11. B. Meyer. Towards practical proofs of class correctness. In *ZB 2003: Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, pages 359 – 387. Springer-Verlag, 2003.
12. D. A. Naumann. Predicate Transformer Semantics of a Higher Order Imperative Language with Record Subtypes. *Science of Computer Programming*, 41(1):1–51, 2001.
13. R. F. Paige and J. S. Ostroff. ERC – An object-oriented refinement calculus for Eiffel. *Formal Aspects of Computing*, 16(1):5, 2004.
14. S. Qin, J. S. Dong, and W. N. Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 321 – 340, 2003.
15. John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*. Palgrave, 2001.
16. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
17. T. L. V. L. Santos, A. L. C. Cavalcanti, and A. C. A. Sampaio. Object-orientation in the UTP. In *UTP'06*, Lecture Notes in Computer Science. Springer-Verlag, 2006. To appear.
18. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 1999.
19. J. C. P. Woodcock and A. L. C. Cavalcanti. A Tutorial Introduction to CSP in Unifying Theories of Programming. In *Pernambuco Summer School on Software Engineering 2004: Refinement*, Lecture Notes in Computer Science. Springer-Verlag, 2005. To appear.
20. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.