Theory of Reference Types

Pavel Naumov
Computer Science Department
Cornell University, Ithaca, NY 14853
pavel@cs.cornell.edu

October 20, 1998

Abstract
Type Theory language is extended by a new constructor to deal with types, representing circular data structures. Although this constructor was designed to model Java reference types, it is general enough to represent self-referring data in many other programming languages. Informal introduction of the new reference type constructor is followed by a set of inference rules and a proof of their consistency.

key words: Type Theory, data structure, reference, semantics, Java

1 Introduction
This work was done to provide mathematical foundations for the formal semantics of a Java fragment in Nuprl1 that has been defined in [11]. Most commonly, recursive data structures are modeled by some kind of memory table that changes its content during program execution. This approach does not consider such structures as an independent mathematical objects and models them instead as type-less web of pointers on the memory. In my work I wanted to find an adequate mathematical model of circular data structures in Type Theory language. In this model the value of every variable at every moment would be some element of an appropriate type.

Two widely studied ([1], [10], [2], [9], [4], [7], [14], [12], [13]) kinds of recursive types: inductive and co-inductive types do not provide suitable representation of circular data structures. Although such structures can be mapped into co-inductive streams, this representation is not unique. For example, two different structures on Figure 1 correspond to the same stream \((5, (5, (5, \ldots)))\).

Klarlund and Schwartzbach [6] introduced graph types to formalize circular structures. They have not specified any inference rules for graph types, but have proved that some formalization of the graph type theory is decidable. This paper describes reference type constructor, which is similar to graph types, but it closely follows Java class type specification. As a result, the language is different from the one used in [6].

---

*This work was support by ONR N00014-92-J-1764 and DARPA f30602-95-1-0047 grants.
1Nuprl [3] is a proof development system based on Martin-Löf-like Constructive Type Theory ([8]).

---

![Figure 1: Data structures indistinguishable by co-inductive types.](image)
The first part of this paper studies one particular example of reference types - type *RecPair*. It gives informal description of this type and specifies inference rules. In the second part general theory of reference types is presented. The third part sketches consistency proof.

2 Type *RecPair*

2.1 Java Class *RecPair*

Cartesian product type $Z \times Z$ is an obvious Type Theory equivalent of the Java class

```
Pair class {int a; int b;}
```

The projectors $pr_1$ and $pr_2$ on elements of the type $Z \times Z$ correspond to Java expressions $p.a$ and $p.b$. Java constructor

```
Pair(int x; int y){a = x; b = y;}
```

corresponds, under the same interpretation, to Type Theory constructor $pair$ that for any two integers $x$ and $y$ returns $\langle x, y \rangle$. Even the fact that Java expressions

```
(new Pair(x,y)).a  (new Pair(x,y)).b
```

return the same value as expressions $x$ and $y$ can be related to the Type Theory equalities

```
pr_1(x, y) = x    pr_2(x, y) = y.
```

But, as we are about to see, in spite of this similarity a slight modification of the definition (1) creates a Java type unknown in the Type Theory world.

Indeed, because class components are implemented in the Java Virtual Machine as pointers, Java allows the name of the class to appear as the type of one or more of its components. For example, we can define class

```
RecPair class {int z; RecPair r;}
```

At first glance, class *RecPair* also looks similar to the Cartesian product. It has constructor

```
RecPair(int x; RecPair y){z = x; r = y;}
```

and destructors $p.a$ and $p.r$ that satisfy equations (2). But this similarity ends as soon as we realize that there are objects in class *RecPair* that have themselves as their second components. For example, for any element $x$ of type int Java constructor

```
RecPair(int x){z = x; r = this;}
```

returns such an object.

This example illustrates that class *RecPair* is indeed different from the Cartesian product type because it easily can be shown that in either Type Theory or in Set Theory a pair can never have itself as one of its components. The same example shows the second major difference between class *RecPair* and Cartesian product - any element of Cartesian product type can be constructed using the *pair* constructor, but not every element of *RecPair* type can be built by constructor (4). In fact, no objects of class *RecPair* can be created using only constructor (4) because it requires an object $y$ of the class *RecPair* to exist before the constructor is applied for the first time.

There are two ways to present class *RecPair* in Type Theory. The *low-level*, or *indirect*, approach would be to imitate the pointer structure used by the Java Virtual Machine. Using this approach we would model class *RecPair* by a function:

```
F : N \rightarrow Z \times N
```

and objects of the class *RecPair* would be represented by natural numbers. Such an approach assumes that function $F$ is given *a priori* as a part of the type definition. It makes impossible to construct any "new"
elements of this type. Hence, we need to have a very sophisticated function $F$ that includes all possible “objects” that we may need in the future, or we must use the so called object state model where function $F$ has an extra argument known as time or machine state. In this case we can not think about an object as just an element of a type, but need to think about functions on time or machine states. Either way we end up with a much more complicated structure than Cartesian product type that corresponds to class Pair.

The second, high-level, or direct, approach to presenting class RecPair in Type Theory is to add a new primitive type constructor to Type Theory that satisfies all the properties of class RecPair that have been mentioned above. The major advantage of this approach is its simplicity. It will just map Java classes into appropriate types in Type Theory without any extra functions, time, or states. The second advantage of this approach is that it develops a new constructor for Type Theory that reflects the important concept of typed reference in programming languages.

Of course, there is a price to pay for choosing the second approach. Once a new primitive abstraction is added to the Type Theory we will also need to add new axioms or inference rules. The biggest challenge will be to prove that new inference rules are consistent with existing Type Theory.

In this work I apply the second approach in a much more general case than just Java class RecPair, but before doing so I want to illustrate the main ideas on the sample case of RecPair class. I will do it by adding to Type Theory a new primitive type constructor RecPair that will model Java class RecPair.

### 2.2 The Graph Model

Before proceeding with the formal description of operations on elements of type RecPair and inference rules that axiomatize these operations, I want to explain the intuitive model behind these axioms. Type RecPair is viewed as an infinite directed graph in which there is exactly one edge starting at each vertex. Each vertex is labeled by an element of type $Z$. Different vertices may be labeled by the same element. In this model, every vertex $v$ represents an element of type RecPair, the label on vertex $v$ is its $z$-component and the unique edge starting at vertex $v$ points to element’s $r$-component. Figure 2 illustrates a fragment of the graph for type RecPair. It is important to note that there is only one graph representing type RecPair and this graph has infinitely many vertices and edges. Hence, Figure 2 displays only a finite fragment of the graph.

Obviously, there are many graphs that satisfy the description above. Not all of them can be used to model RecPair type. Later I will formulate axioms for elements of RecPair type and discuss what kind of restrictions on the graph they impose.

---

2I use **typewriter** style when referring to Java constructors and expressions and *italic* for their Type Theory counterparts.
2.3 Naïve Approach

In order to add a new type constructor, we have to specify elements of this type and their relation to elements of other types by selecting the set of primitive operations on the elements and formulating axioms for these operations. The naïve way to do so would be to try to apply the scheme already used for the Cartesian product type: one type constructor and two type destructors that return the components. Let us call such constructor \textit{recpair} and destructors \textit{pr}_z and \textit{pr}_r. We want constructor \textit{recpair} to return an element \textit{recpair}(x, y) of type \textit{RecPair} for each integer \( x \) and each element \( y \) of type \textit{RecPair}. For any element \( p \) of type \textit{RecPair}, projectors \textit{pr}_z(p) and \textit{pr}_r(p) should return elements of types \( \mathbb{Z} \) and \textit{RecPair} correspondingly. In addition, we would like to assume that the following version of Cartesian product properties (2) hold for type \textit{RecPair}:

\[
\text{pr}_z(\text{recpair}(x, y)) = x \quad \text{pr}_r(\text{recpair}(x, y)) = y
\]

Unfortunately, from our discussion on page 2 it follows that this naïve approach fails because constructor \textit{recpair} can not build self-referencing elements of type \textit{RecPair} and it needs to have at least one of them \textit{a priori}.

Below we will choose another approach. Instead of the universal constructor \textit{recpair}(x, y) that was expected to be able to produce any element of type \textit{RecPair}, we will introduce just a single constant \( \rho \) of the type \textit{RecPair}. All other elements of this type will be the results of operations on this type that will be defined later.

2.4 Constant \( \rho \)

We will assume that there is at least one element of type \textit{RecPair} that has 0 as its \( z \)-component and itself as its \( r \)-component. We call this self-referential element \( \rho \). In our graph model \( \rho \) corresponds to a vertex labeled by integer 0 which is an initial and final point of a loop edge. This situation is illustrated on Figure 3. If a graph has several self-referential vertices labeled by 0 any of them can be chosen to be \( \rho \). We add an inference rule to Type Theory \(^3\) that postulates the existence of the element \( \rho \):

\textbf{Inference Rule I}

\[
\rho \in \text{RecPair}
\]

As one can see, constant \( \rho \) is just a Type Theory version of the Java constructor

\[
\text{RecPair}()\{z = 0; r = \text{this};\}
\]

We will be able to formulate the other properties of the constant \( \rho \) only after extending the Type Theory language with two destructors of type \textit{RecPair} elements.

2.5 Destructors \( \text{pr}_z(p) \) and \( \text{pr}_r(p) \)

Like Cartesian product projectors \( \text{pr}_1 \) and \( \text{pr}_2 \), \textit{RecPair} type projectors \( \text{pr}_z \) and \( \text{pr}_r \) return corresponding components of the type element. These components have types \( \mathbb{Z} \) and \textit{RecPair}:

\(^3\) Although in [11] I formalized reference types in Nuprl Type Theory, in this paper I prefer to formulate “generic” rules that can be adopted to a wide range of Type Theory formalizations.
Inference Rule II
\[ p \in \text{RecPair} \]
\[ \text{pr}_z(p) \in \mathbb{Z} \]

Inference Rule III
\[ p \in \text{RecPair} \]
\[ \text{pr}_r(p) \in \text{RecPair} \]

As assumed above, constant \( \rho \) is a self-referencing element with \( z \)-component 0:

Inference Rule IV
\[ \text{pr}_z(\rho) = 0 \]

Inference Rule V
\[ \text{pr}_r(\rho) = \rho \]

It is important to keep in mind that unlike elements of the Cartesian product type, elements of \( \text{RecPair} \) type are not uniquely determined by their components. Figure 4 presents two different vertices labeled by the same number and their edges point to the same elements. Obviously, such elements of the type \( \text{RecPair} \) must be different because one of them is self-referential and the other is not. Although we will consider distinct vertices in our graph model to represent different elements of type \( \text{RecPair} \), discussion of equality on type \( \text{RecPair} \) definitely should be postponed – at this moment we do not even have sufficient inference rules to prove that there is an element of type \( \text{RecPair} \) that can be represented by the left vertex of Figure 4.

We will be able to prove this after the introduction of the operations on type \( \text{RecPair} \). These operations are Type Theory equivalents of assignment statements in Java.

2.6 Java Assignment Statements

We intend to define operations on type \( \text{RecPair} \) that correspond to assignments of a new object component in Java. But before doing so let us consider the effect that is created by a simple assignment of a new value to an integer variable. For example, Java code

\[ n = 5; \quad (7) \]

assigns integer value 5 to variable \( n \). A function that maps variable names to elements of appropriate types is called \emph{environment}. Therefore, we can say that assignment (7) changes the environment from \( \text{env} \) to

\[ \text{env}' = \lambda v. \begin{cases} 5 & \text{if } v = n \\ \text{env}(v) & \text{otherwise} \end{cases} \]

Now let us assume that Java variable \( p \) has type \( \text{RecPair} \) and \( k \) is a Java constant or variable\(^4\) of type \( \text{int} \). How does assignment:

\[ p.z = k; \quad (8) \]

\(^4\)If \( k \) is a more complicated expression then its evaluation, if considered as a part of assignment (8) evaluation, may arbitrarily change the environment as a side effect.
change the environment \( env \)? Obviously, this assignment changes the value of the variable \( p \) because its \( z \)-component is changed. In the Type Theory terms it means that the statement (8) changes the element of \( \text{RecPair} \) assigned to variable \( p \) by the environment.

It may be less obvious that the assignment (8) also produces other changes in the environment. To show this, assume that the value of variable \( p \) is Java object \( x \) and that there is another variable \( q \) the value \( y \) of which has \( x \) as its second component:

\[
\begin{align*}
env(p) &= x \\
env(q) &= y \\
pr_r(y) &= x
\end{align*}
\]

The environment created after evaluation of assignment (8), let us call this environment \( env' \), maps the variable \( p \) into a Java object \( x' \). We already know that object \( x' \) is different from object \( x \) because these two objects have different \( z \)-components. We also expect that assignment (8) should not change the fact that the value of variable \( p \) is the \( r \)-component of the value of variable \( q \):

\[
pr_r(env'(q)) = env'(p)
\]

Hence, we can conclude that environments \( env \) and \( env' \) should assign different values to variable \( q \) because

\[
pr_r(env'(q)) = env'(p) = x' \neq x = pr_r(y) = pr_r(env(q))
\]

Therefore, assignment (8) will change the environment function not only on variable \( p \), but also on some other variables of type \( \text{RecPair} \).

As it will turn out, function \( env' \) can be decomposed into a function

\[
h : \text{RecPair} \rightarrow \text{RecPair}
\]

and function \( env \):

\[
env' = h \circ env
\]

In other words, we will be able to “prove” that there is a function \( h \) that makes diagram on Figure 5 commutative.

In order to establish the existence of such a function \( h \) we only need to prove that for any two variables \( p \) and \( q \), if \( env(p) = env(q) \) then \( env'(p) = env'(q) \). But we know that statement (8) does not re-assign any variables. Hence, if two variables pointed to the same object before the evaluation of the statement, then they will point to the same object after the evaluation.

Therefore, we have proved that there is a function \( h \) from the type \( \text{RecPair} \) into itself that satisfies equation (10). We will call function \( h \) the update function because it basically “updates” environment \( env \) to environment \( env' \).

A similar result can be proved for assignment

\[
p.r = r0;
\]

where \( r0 \) is a Java constant or variable of type \( \text{RecPair} \).
2.7 Operation $update_z(x, b, y)$

In the previous section we introduced function $update$ to represent assignment (8) in Type Theory. This assignment has three parameters: variable $p$, class component name $z$, and constant expression $k$. Note that if $p$ and $q$ are two different variables that have the same value, then the two assignments

$p.z = k;$ \hspace{1cm} $q.z = k;$

will have exactly the same effect on the environment. This means that $update$ is a function of the variable $p$’s value, not its name. For the same reason, the second argument of $update$ function is the value of expression $k$, not the expression itself.

Also, because type $RecPair$ has only two components, instead of using component name as a function $update$ parameter, I prefer to deal with two different update functions $update_z$ and $update_r$. We will start with the function $update_z$, leaving $update_r$ for the next section.

If element $x$ of type $RecPair$ represents in Type Theory the value of variable $p$ and the integer $b$ represents the value of expression $k$, then $update_z(x, b)$ is a function

$\text{update}_z(x, b) : \text{RecPair} \rightarrow \text{RecPair}$

It is an empirical observation that one seldom uses functions without applying them to an argument\footnote{Ironically, this section will be an exception.}. This is why, as with most other Type Theory operators, it is more convenient to consider the three-place operator $\text{update}_z(x, b, y)$ that combines two-place operator $\text{update}_z(x, b)$ and function application:

$\text{update}_z(x, b, y) := \text{update}_z(x, b) \; y$

Now we are ready to add $update$ as a new primitive to Type Theory. For any two elements $x$ and $y$ of type $RecPair$ and for any integer $b$, expression $\text{update}_z(x, b, y)$ has type $RecPair$. This can be formalized as the following inference rule:

**Inference Rule VI**

\[
\frac{x, y \in \text{RecPair} \quad b \in \mathbb{Z}}{\text{update}_z(x, b, y) \in \text{RecPair}}
\]

After a new primitive abstraction is introduced, we need to add axioms that relate it to other abstractions in the theory. In our case, we want to specify components of element $\text{update}_z(x, b, y)$. Because we want type $RecPair$ to model Java class $\text{RecPair}$ we again turn to informal arguments about Java in order to find the axioms.

If $env$ is the environment before assignment (8) is evaluated then

$\text{env}' = \lambda v.\text{update}_z(x, b, \text{env}(v))$

is the environment after this assignment is evaluated. We know that after the evaluation the $z$-component of the object, corresponding to variable $p$, is equal to $b$:

$b = \text{pr}_z(\text{env}'(p)) = \text{pr}_z(\lambda v.\text{update}_z(x, b, \text{env}(v)) \; p) = \text{pr}_z(\text{update}_z(x, b, x))$

This can be formalized as an inference rule

**Inference Rule VII**

\[
\frac{x \in \text{RecPair} \quad b \in \mathbb{Z}}{\text{pr}_z(\text{update}_z(x, b, x)) = b}
\]

Although we have established this rule only for the case when there is a variable that has value $x$ and there is a variable or constant that has value $b$, I formulated this rule for any $x$ of type $RecPair$ and any $b$ of type $\mathbb{Z}$, because of the fundamental assumption that a Java object not named by a variable has exactly
the same properties as objects that are denoted by variables. I will use the same assumption for deriving the other inference rules for type RecPair\(^6\).

Similarly, if variable \(q\) has value \(y\) before the evaluation and objects \(x\) and \(y\) are different, then we know that \(z\)-component of variable \(q\) value will not be changed by assignment (8):

\[
pr_z(y) = pr_z(env(q)) = pr_z(env'(q)) =
pr_z(\lambda.v.update_z(env(x), b, env(v)) q) = pr_z(update_z(x, b, y))
\]

This justifies the following inference rule:

**Inference Rule VIII**

\[
x, y \in \text{RecPair} \quad x \neq y \quad b \in \mathbb{Z} \quad pr_z(update_z(x, b, y)) = pr_z(y)
\]

Situation with the \(r\)-component of \(update(x, b, y)\) is more complicated since it is an element of type RecPair and, as we know, assignment (8) can change value of RecPair variable even if this value is different from the one of variable \(p\).

Fortunately, we can still say that if before the evaluation variables \(p, q,\) and \(t\) had values \(x, y,\) and \(z\) and object \(z\) was \(r\)-component of object \(y\), then after the evaluation the new value of variable \(t\) is still the \(r\)-component of variable \(q\) value:

\[
pr_r(env'(q)) = env'(t)
\]

Using (11) the last equation can be reduced to

\[
pr_r(update_z(x, b, y)) = update_z(x, b, z)
\]

Taking into account that \(z = pr_r(y)\), the following inference rule can be stated:

**Inference Rule IX**

\[
x, y \in \text{RecPair} \quad b \in \mathbb{Z} \quad pr_r(update_z(x, b, y)) = update_z(x, b, pr_r(y))
\]

We also know that because assignment (8) does not re-assign variables, for any two Java variables \(p\) and \(q\), \(env'(p) = env'(q)\) implies that \(env(p) = env(q)\). This means that function \(\lambda y.\text{update}_z(x, b, y)\) is a 1-1 mapping:

**Inference Rule X**

\[
update_z(x, b, y_1) = update_z(x, b, y_2) \quad y_1 = y_2
\]

Finally, let us formulate what kind of restriction on our graph model we should impose to guarantee that inference rules VII, VIII, IX, and X are satisfied. As one can see, these rules state that function

\[
\lambda y.\text{update}(x, b, y)
\]

is a 1-1 endomorphism of the type RecPair graph that preserves all edges and labels at all vertices except for the label of vertex \(x\). The image of vertex \(x\) is labeled by \(b\). In order for a graph to be a model of the type RecPair there should exist a mapping from the graph into itself that satisfies this condition.

\(^6\) As with any other “fundamental principle”, this assumption is true only in some simplified abstract model of the real world. In the Java Virtual Machine objects that are not named directly by variables or indirectly via expressions may be subject to destruction by the garbage collector.
2.8 Operation \( update_r(x, w, y) \)

In this section I introduce one more operator on the type \( 	ext{RecPair} \). This operator corresponds to Java environment update, generated by Java assignment

\[
p.r = r_0; \tag{12}
\]

Similarly to the case of assignment (8), changes in the environment generated by this assignment can be represented by a function \( update_r(x, w, y) \):

\[
env' = \lambda v. update_r(x, w, env(v)) \tag{13}
\]

where \( x \) is the value of variable \( p \) and \( w \) is the value of variable \( r_0 \). Formal inference rule that specifies the type of \( update_r(x, w, y) \) operator is

**Inference Rule XI**

\[
\begin{array}{c}
x, w, y \in \text{RecPair} \\
update_r(x, w, y) \in \text{RecPair}
\end{array}
\]

Just as in the case of \( update_r(x, b, y) \) operator, we will formulate the inference rules for the operator based on our informal intuition about corresponding Java assignment.

We know that for any variable \( q \) of type \( 	ext{RecPair} \), the \( z \)-component of the value of the variable \( q \) should be the same before and after the evaluation:

\[
pr_z(env(q)) = pr_z(env'(q))
\]

Using (13) we can reduce this equation to

\[
pr_z(env(q)) = pr_z(update_r(x, w, env(q)))
\]

If \( y \) is the value of variable \( q \) before the evaluation, then

\[
pr_z(y) = pr_z(update_r(x, w, env(y)))
\]

We have established this fact only for objects \( x \) and \( y \) that were the values of some variables of type \( 	ext{RecPair} \) before the evaluation took place, but in Type Theory we assume this property for any objects \( x \) and \( y \) of type \( 	ext{RecPair} \):

**Inference Rule XII**

\[
\begin{array}{c}
x, w, y \in \text{RecPair} \\
pr_z(update_r(x, w, y)) = pr_z(y)
\end{array}
\]

We also know that \( r \)-component of the value of the variable \( p \) after the evaluation should be equal to the value of variable \( r_0 \) after the evaluation:

\[
pr_r(env'(p)) = env'(r_0)
\]

With the help of (13) this equation can be reduced to:

\[
pr_r(update_r(x, w, x)) = update_r(x, w, w)
\]

Again, we have established the last equality only for objects that are denoted by some variables, but in Type Theory we will assume this for any objects:

**Inference Rule XIII**

\[
\begin{array}{c}
x, w \in \text{RecPair} \\
pr_r(update_r(x, w, x)) = update(x, w, w)
\end{array}
\]
Finally, if a variable \( q \) of the type \( \text{RecPair} \) had value \( y \), different from \( x \), before the evaluation (12) took place, and \( r \)-component of object \( y \) was equal to the value of variable \( t \), then after the evaluation \( r \)-component of new value of variable \( q \) is also equal to the new value of variable \( t \):

\[
pr_r(env'(q)) = env'(t)
\]

From this equation, using equality (13) we can derive that

\[
pr_r(update_r(x, w, y)) = update_r(x, w, pr_r(y))
\]

Assuming that this equation holds in Type Theory for any objects \( x \neq y \), and \( w \), not only for named by some variables, we can get the following rule:

**Inference Rule XIV**

\[
x, w, y \in \text{RecPair} \quad x \neq y \quad pr_r(update_r(x, w, y)) = update_r(x, w, pr_r(y))
\]

Just as in case of assignment (8), we know that assignment (12) satisfies the property

\[
env'(p) = env'(q) \Rightarrow env(p) = env(q)
\]

Therefore, \( \lambda y. update_r(x, w, y) \) is a 1-1 mapping:

**Inference Rule XV**

\[
update_r(x, w, y_1) = update_r(x, w, y_2) \quad y_1 = y_2
\]

The inference rules XII, XIII, XIV, and XV translated into the language of our graph model for type \( \text{RecPair} \), state that

\[
\lambda y. update_r(x, w, y)
\]

is a 1-1 endomorphism of type \( \text{RecPair} \) that preserves vertex labels and all edges except for the edge that starts at vertex \( x \). This edge is mapped into the edge that goes from the image of vertex \( x \) to the image of vertex \( w \).

### 2.9 On Modeling Java new Constructor

It is common to associate Java `new RecPair()` constructor with some kind of magic hat from which one can take a new object each time when an assignment like

\[
p = \text{new RecPair}();
\]

is evaluated. But there is no magic in Type Theory – each element of every type exists at any time.

To model assignment (14) in Type Theory I will use a very different idea. I assume that there is a 1-1 endomorphism \( h \) from type \( \text{RecPair} \) into itself that preserves operations \( pr_z, pr_r, update_z, \) and \( update_r \). For any such mapping \( h \) and for any environment \( env \), environment

\[
env' = h \circ env
\]

would be equivalent to the environment \( env \) at least in the sense that they intuitively represent the same state of Java Virtual Machine. I also assume that function \( h \) is not a surjection. This means that there is such an element \( r_0 \) in type \( \text{RecPair} \) which is not a member of \( h(\text{RecPair}) \). See Figure 6 for an illustration.

If such endomorphism \( h \) exists, then Java assignment (14) can be represented in Type Theory by a function \( update \) that maps an environment \( env \) into environment

\[
env' = \lambda v. \begin{cases} r_0 & \text{if } v = p \\ h \circ env(v) & \text{otherwise} \end{cases}
\]

As one can see, I model Java assignment (14) similarly to assignments (8) and (12). The important difference is that this time we do not need to add a new primitive abstraction, because

\[
h := \lambda y. update_r(p, p, y) \\
r_0 := p
\]

satisfy all the properties of the function \( h \) and of the element \( r_0 \) stated above.
2.10 Canonical Elements of Type \textit{RecPair}

After the introduction of \(\rho, pr_z, pr_r, update_z,\) and \(update_r\), there are plenty of elements in the type \textit{RecPair} that we can define. Basically, any well-typed expression that uses \(\rho, pr_r, update_z,\) and \(update_r\) returns an element of type \textit{RecPair}. We will call such elements of type \textit{RecPair} definable elements. There are two questions that probably almost everyone would ask

1. Can two different expressions of the form described above define the same element?

2. Are there any other elements in type \textit{RecPair} except for definable?

The answer to the first question is simple. Already the inference rule V gives us an example of two different expressions that are equal as \textit{RecPair} type elements:

\[ pr_r(\rho) = \rho \]

In fact, from inference rules V, IX, XIII, and XIV it follows that any expression that is built from \(\rho, pr_z, update_z,\) and \(update_r\) can be reduced to an expression that uses only \(\rho, update_z,\) and \(update_r\). An expression that uses only elements of type \(Z\) and operators \(\rho, update_z,\) and \(update_r\) we will call canonical expressions. The values of canonical expressions will be called canonical elements of type \textit{RecPair}. As we just have seen,

\textbf{Theorem 1} Any definable element is canonical. \(\Box\)

In this paper I will assume that all canonical elements are different\footnote{exclude \(pr_z\) from this list because it has type \(Z\)}\footnote{The question whether any two canonical elements are different cannot be solved using existing rules. In fact, various people may be inclined to answer this question differently depending on their intuition and goals. I have found that for Java semantics it is enough to assume that all canonical elements are different. But many participants of PRL Seminar at Cornell argued that this assumption results in the fact that the following two intuitively equivalent Java pieces of code correspond to different changes in the environment:

\begin{itemize}
  \item \(p.z = 5;\)
  \item \(p.z = 7;\)
\end{itemize}

Although there is nothing wrong with the fact that these two programs will be represented in Type Theory by different functions, I would agree that the existence of a stronger but still adequate for our purposes equality on type \textit{RecPair} is an interesting problem for which I do not have a good solution at the present time.}. This fact is formalized by the following five rules:
Inference Rule XVI

\[ \text{update}_z(x_1, b_1, y_1) = \text{update}_z(x_2, b_2, y_2) \]

\[ (x_1 = x_2) \& (b_1 = b_2) \]

Inference Rule XVII

\[ \text{update}_z(x_1, w_1, y_1) = \text{update}_z(x_2, w_2, y_2) \]

\[ (x_1 = x_2) \& (w_1 = w_2) \]

Inference Rule XVIII

\[ b \in \mathbb{Z}, \ x, y \in \text{RecPair} \]

\[ \rho \neq \text{update}_z(x, b, y) \]

Inference Rule XIX

\[ x, w, y \in \text{RecPair} \]

\[ \rho \neq \text{update}_z(x, w, y) \]

Inference Rule XX

\[ b \in \mathbb{Z}, \ x_1, y_1, x_2, w_2 \in \text{RecPair} \]

\[ \text{update}_z(x_1, b_1, y_1) \neq \text{update}_z(x_2, w_2, y_2) \]

The second question is simpler. Since any Java program can be decomposed into a sequence of assignments of different values to object component, it seems reasonable to assume that canonical elements are the only ones that we will need for Java semantics. My implementation of Java semantics in Nuprl supports this thesis.

Therefore, we assume that there are no other elements in RecPair type besides canonical ones. This fact can be formalized as an induction principle for type RecPair:

Inference Rule XXI

\[ H; \ J[\rho/\rho] \vdash P[\rho/\rho] \]

\[ H; \ x_1, y_1 : \text{RecPair}; \ b_1 : \mathbb{Z}; \ J[\text{update}_z(x_1, b_1, y_1)/\rho] \vdash P[\text{update}_z(x_1, b_1, y_1)/\rho] \]

\[ H; \ x_1, y_1 : \text{RecPair}; \ J[\text{update}_z(x_1, w_1, y_1)/\rho] \vdash P[\text{update}_z(x_1, w_1, y_1)/\rho] \]

\[ H; \ x, w, y : \text{RecPair}; \ J[\rho/\rho] \vdash P \]

3 General Theory of Reference Types

3.1 Reference Type Signature

The presented above axiomatization of the type RecPair is just an example of Java classes formalization in Type Theory. In this section we will look at how the same technique can be applied in a more general case. We will use the term reference type for referring to new types that we will add to Type Theory to represent Java classes.

Although objects of Java class RecPair have only two components, objects of other Java classes can have different number of components depending on which class they belong to. All components of any Java class can be divided into two categories: primitive and reference according to their type\(^9\). For instance, class RecPair has one primitive component and one reference component: \(z\)-component and \(r\)-component correspondingly. In Type Theory we will use the same terms “primitive” and “reference” for appropriate class type components. From inference rules for RecPair type we already know that primitive and reference components will have different properties.

One of the possible ways to generalize type RecPair is to consider reference types that have several primitive and several reference components. But it turns out that reference type with several primitive components can be modeled by a reference type the unique primitive component of which is a Cartesian product of appropriate types.

Since the same construction cannot be applied to reference components, we will assume that reference types may have several reference components, but only one primitive component. Each reference type will

\(^9\)Java specification \cite{5} defines types int, float, char, and boolean as primitive and types class and array as reference.
have Index type associated with it. The Index type is the type of names for reference components. We will call the unique primitive component of a reference type element a core of this element. Each reference type has some type reserved for cores of its elements. We will call this type Core. In the case of RecPair reference type, Index is any type with only one element and Core type is type Z.

An important Java feature is the ability to define simultaneously several classes that can use each other as component types. For example,

\[
\begin{align*}
X & \text{ class } \{ \text{int } a; \text{ y; } \} \\
Y & \text{ class } \{ \text{boolean } b; \text{ x; } \}
\end{align*}
\]

(15)

To deal with such situations, we will talk about parametrized reference types. Parametrized reference type is a function from some type, that we call Name type, into type universe:

\[
f : \text{Name } \rightarrow \text{U}
\]

(16)

For instance, the definition (15) can be modeled by a function \( f : \text{Z}_2 \rightarrow \text{U} \) such that \( f(0) \) represents class \( X \) and \( f(1) \) represents class \( Y \).

Since different reference types can have different number of reference components and different Core types, we will assume that Index and Core are also functions of the type Name \( \rightarrow \text{U} \).

For the reason that will be explained later, we want to have function \( c \in n : \text{Name } \rightarrow \text{Core}(n) \) that returns a representative element \( c(n) \) of the type Core\((n)\) for each reference type name \( n \in \text{Name} \). In particular, the existence of such a function implies that Core\((n)\) is not empty for any \( n \in \text{Name} \).

Finally, every reference component of every parametrized reference type has some type. This type can be specified by a function that for any \( n \in \text{Name} \) and any \( i \in \text{Index}(n) \) returns the name of the type of its type \( i \)-th component: Field\((n,i)\) \( \in \text{Name} \).

 Quintuple

\[
S = (\text{Name,Core,c,Index,Field})
\]

of the type

\[
(\text{Name} : \text{U}) \times (\text{Core} : \text{Name } \rightarrow \text{U}) \times (n : \text{Name } \rightarrow \text{Core}(n)) \times
\]

\[
(\text{Index} : \text{Name } \rightarrow \text{U}) \times (n : \text{Name } \rightarrow \text{Index}(n) \rightarrow \text{Name})
\]

that defines reference type will be called reference type signature.

### 3.2 The Graph Model

Our graph model of RecPair type can be easily adopted to a more general situation of parametrized reference type. We will think about disjoint union of parametrized types (16) as about one big “super graph”. Each vertex of this super graph is labeled by reference type name \( n \in \text{Name} \) and a core value of type Core\((n)\). There will be one edge starting at the vertex for each element of Index\((n)\). This edge will be labeled by the corresponding element of type Index\((n)\) and it will point to appropriate reference component of the element. We will assume that the edge marked by \( i \in \text{Index}(n) \) points to a vertex with reference name Field\((n,i)\). For example, Figure 7 shows a fragment of a “super graph” for Java classes (15).
3.3 Type $\Upsilon(S@n)$

In this section we will introduce inference rules for a parametrized reference type defined by a signature:

$$S = \langle Name, Core, c, Index, Field \rangle$$

As I have mentioned in the previous section, parametrized reference type is a function from type $Name$ into type universe $U$. For any element $n$ of type $Name$ we will denote the value of this function on the argument $n$ as $\Upsilon(S@n)$.

We assume that reference type $\Upsilon(S@n)$ exists for any signature $S$:

**Inference Rule I**

$$\begin{align*}
& \text{Name} \in U \\
& \text{Core} \in \text{Name} \to U \\
& c \in n : \text{Name} \to \text{Core}(n) \\
& \text{Index} \in \text{Name} \to U \\
& n \in \text{Name} \\
& \text{Field} \in n : \text{Name} \to \text{Index}(n) \to \text{Name}
\end{align*}$$

$$\Upsilon(S@n) \in U$$

Until the end of this chapter we will deal with reference types defined by a prespecified reference signature. Hence, it will be possible to use more compact notation $\Upsilon(n)$ for the type $\Upsilon(S@n)$. Below I will introduce destructors, constructor, and operations on the elements of the type $\Upsilon(n)$. I start with destructors since constructor $\rho$ will require slight modification before it can be adopted to parametrized reference types.

3.4 Destructors core and ref

In the case of RecPair type we had two element destructors: projector $pr_z$ returned the core of the element and projector $pr_r$ returned reference component. Since any element of type $\Upsilon(n)$ has one core component and many reference components, the only change that we will need to make is to add reference component name to the second projector. We will call projectors for reference type core and ref. The formation rules for them are:

**Inference Rule II**

$$\frac{x \in \Upsilon(n)}{\text{core}(x) \in \text{Core}(n)}$$

**Inference Rule III**

$$\frac{x \in \Upsilon(n) \quad i \in \text{Index}(n)}{\text{ref}(x, i) \in \Upsilon(\text{Field}(n, i))}$$

3.5 Constants $\rho(n)$

We will need a separate constant $\rho(n)$ for each reference type name $n \in \text{Name}$.

**Inference Rule IV**

$$\frac{n \in \text{Name}}{\rho(n) \in \Upsilon(n)}$$

Also, as one can see, we will no longer be able to assume that $\rho(n)$ has itself as its reference components since the $i$-th reference component has type $\Upsilon(\text{Field}(n, i))$ that may be different from $\Upsilon(n)$. Instead, we will assume that $i$-th reference component of $\rho(n)$ is $\rho(\text{Field}(n, i))$

**Inference Rule V**

$$\frac{n \in \text{Name} \quad i \in \text{Index}(n)}{\text{ref}(\rho(n), i) = \rho(\text{Field}(n, i))}$$

---

10 Careful reader may notice that I have not formulated similar formation rule for RecPair type. It was only because RecPair type inference rules were solely meant to be an example and I tried not to divert reader's attention from other, much less trivial, inference rules.
We will need to select some element of type Core\(n\) that will be the core value of the element \(\rho(n)\). This choice is absolutely unimportant because an operation, similar to \(update_o\), will be able to create elements with other core values. We will use signature component \(c\) to select such an element of type Core\(n\):

**Inference Rule VI**

\[
\frac{n \in Name}{\text{core}(\rho(n)) = c(n)}
\]

### 3.6 Operation \(update_c(x, b, y)\)

Operation \(update_c(x, b, y)\) on type \(\Upsilon(n)\) is almost identical to \(update_c(x, b, y)\) on the type RecPair except that now \(x\) and \(y\) can have different types.

**Inference Rule VII**

\[
\frac{x \in \Upsilon(n) \quad b \in \text{Core}(n) \quad y \in \Upsilon(m)}{update_c(x, b, y) \in \Upsilon(m)}
\]

**Inference Rule VIII**

\[
\frac{x \in \Upsilon(n) \quad update_c(x, b, y) \in \Upsilon(m) \quad \langle n, x \rangle \neq \langle m, y \rangle}{\text{core}(update_c(x, b, y)) = \text{core}(y)}
\]

**Inference Rule IX**

\[
\frac{x \in \Upsilon(n) \quad b \in \text{Core}(n)}{\text{core}(update_c(x, b, x)) = b}
\]

**Inference Rule X**

\[
\frac{update_c(x, b, y) \in \Upsilon(n) \quad i \in \text{Index}(n)}{\text{ref}(update_c(x, b, y), i) = update_c(x, b, \text{ref}(y, i))}
\]

### 3.7 Operation \(update_r(x, i, z, y)\)

Compared to \(update_o\) on RecPair type, reference component update for elements of \(\Upsilon(n)\) type has one extra parameter that specifies the name of the reference component being updated:

**Inference Rule XI**

\[
\frac{x \in \Upsilon(n) \quad i \in \text{Index}(n) \quad z \in \Upsilon(\text{Field}(n, i)) \quad y \in \Upsilon(m)}{update_r(x, i, z, y) \in \Upsilon(m)}
\]

**Inference Rule XII**

\[
\frac{update_r(x, i, z, y) \in \Upsilon(n)}{\text{core}(update_r(x, i, z, y)) = \text{core}(y)}
\]

**Inference Rule XIII**

\[
\frac{x \in \Upsilon(n) \quad update_r(x, i, z, y) \in \Upsilon(m) \quad j \in \text{Index}(m) \quad \langle n, x, i \rangle \neq \langle m, y, j \rangle}{\text{ref}(update_r(x, i, z, y), j) = update_r(x, i, z, \text{ref}(y, j))}
\]

**Inference Rule XIV**

\[
\frac{update_r(x, i, z, x) \in \Upsilon(n)}{\text{ref}(update_r(x, i, z, x), i) = update_r(x, i, z, i)}
\]

15
3.8 On Modeling new constructor

When modeling Java new constructor, the procedure described in section on RecPair can be applied in a more general case of reference types. Namely, for any reference name \( n \in Name \), the statement

\[
\text{p} = \text{new n();}
\]

corresponds to the the environment update

\[
\text{env'} = \lambda v. \left\{ \begin{array}{ll}
r_0 & \text{if } v = \text{p} \\
h \circ \text{env}(v) & \text{otherwise}
\end{array} \right.
\]

where

\[
r_0 := \rho(n) \\
h := \lambda y. \text{update}_c(\rho(n), c(n), y)
\]

3.9 Canonical Elements

By canonical elements of the type \( \Upsilon(n) \) I will mean the minimal subtype \( C(n) \) of the type \( \Upsilon(n) \) which satisfies the following three conditions:

1. \( \rho(n) \in C(n) \) for all \( n \in Name \)
2. if \( x \in C(n), \ b \in \text{Core}(n), \) and \( y \in C(m) \) then \( \text{update}_c(x, b, y) \in C(m) \)
3. if \( x \in C(n), \ i \in \text{Index}(n), \ z \in \Upsilon(\text{Field}(n, i)), \) and \( y \in C(m) \) then \( \text{update}_c(x, b, y) \in C(m) \)

All properties of canonical elements formulated for type RecPair will be assumed to hold in the more general case of the reference type. Below I just formulate these properties, since we already have discussed them above.

**Syntactically different canonical elements are not equal.** This fact can be formalized with the following inference rules

Inference Rule XV

\[
\frac{n \in Name \quad \text{update}_c(x, b, y) \in \Upsilon(n)}{\rho(n) \neq \text{update}_c(x, b, y)}
\]

Inference Rule XVI

\[
\frac{n \in Name \quad \text{update}_c(x, i, z, y) \in \Upsilon(n)}{\rho(n) \neq \text{update}_c(x, i, z, y)}
\]

Inference Rule XVII

\[
\frac{\text{update}_c(x_1, b_1, y_1) \in \Upsilon(n) \quad \text{update}_c(x_2, i, z, y_2) \in \Upsilon(n)}{\text{update}_c(x_1, b_1, y_1) \neq \text{update}_c(x_2, i, z, y_2)}
\]

Inference Rule XVIII

\[
\frac{\text{update}_c(x_1, b_1, y_1) = \text{update}_c(x_2, b_2, y_2)}{(x_1 = x_2) \& (b_1 = b_2) \& (y_1 = y_2)}
\]

Inference Rule XIX

\[
\frac{\text{update}_c(x_1, i_1, z_1, y_1) = \text{update}_c(x_2, i_2, z_2, y_2)}{(x_1 = x_2) \& (i_1 = i_2) \& (z_1 = z_2) \& (y_1 = y_2)}
\]

Any element of type \( \Upsilon(n) \) is canonical. This fact can be stated as induction principle for type \( \Upsilon(n) \)

Inference Rule XX

\[
H; \ J[\rho(n)/p] \vdash P[\rho(n)/p] \\
H; \ m : \text{Name}; \ x : \Upsilon(m); \ b : \text{Core}(m) \ y : \Upsilon(n); \\
J[\text{update}_c(x, b, y)/p] \vdash P[\text{update}_c(x, b, y)/p] \\
H; \ m : \text{Name}; \ x : \Upsilon(m); \ i : \text{Index}(m); \ z : \Upsilon(\text{Field}(m, i)); \\
y : \Upsilon(n); \ J[\text{update}_c(x, i, z, y)/p] \vdash P[\text{update}_c(x, i, z, y)/p] \\
H; \ p : \Upsilon(n) \; J \vdash P
\]
4 Consistency

In order to add the class type constructor to Type Theory we should guarantee the consistency of the new inference rules with the existing ones. The most straightforward way to do it would be to take some model of the existing Type Theory and to extend it by a class type constructor in such a way that inference rules for class type hold in extended model. This procedure is a standard technique in the part of Mathematical Logic known as Model Theory because it can be applied to a wide range of different theories. Unfortunately, this approach is also rather complicated, because it requires detailed knowledge of an existing model for Type Theory and the understanding of the possible ways of its expansion.

Another way of extending a theory language by new abstractions is more common in theories with expressively rich languages such as Set Theory. In such theories new notions could be defined using existing constructions. For example, such important objects as pair, function, natural number, real number, or group can be expressed in Set Theory in terms of sets. Of course, the ability to define all objects in Set Theory using only sets does not depreciate the value of such mathematical theories as Arithmetics, Real Number Theory, or Abstract Algebra. Most mathematicians would agree that although real numbers can be defined in terms of sets, these definitions are only good for providing foundations for Real Number Theory. Available axiomatization of the real numbers makes much more sense for everybody who is studying them.

This work uses the second approach. I will define class type using existing constructors in Nuprl Type Theory. These definitions would guarantee the consistency of the extended theory and show expressive power of the existing Type Theory. But just as in the case of Set Theory, these definitions should not be considered to be replacement for class type axiomatization presented in the previous chapter.

4.1 RecPair Type Definition in Nuprl

In this section I will show how RecPair type can be defined in Nuprl Type Theory using inductive types. Let RecPair be Nuprl type

$$\mu(X.\text{Unit} + (X \times Z \times X) + (X \times X \times X))$$

or, in other words, RecPair is the minimal solution of the following type equation

$$\text{RecPair} = \text{Unit} + (\text{RecPair} \times Z \times \text{RecPair}) + (\text{RecPair} \times \text{RecPair} \times \text{RecPair})$$

(17)

As one can see from the above definition, there are three ways to construct elements of the type RecPair that correspond to three different components of disjoint union:

$$\rho(a) = \text{inl}(\cdot)$$

$$\text{update}_a(x, b, y) = \text{inr}(\text{inl}((x, b, y)))$$

$$\text{update}_r(x, b, y) = \text{inr}(\text{inr}((x, z, y)))$$

The fact that any element of RecPair type can be constructed from the above defined operations $\rho$, $\text{update}_a$, and $\text{update}_r$ justifies the induction principle for type RecPair. Also, from (17) and basic properties of Cartesian product and disjoint union follows that all canonical elements of the type RecPair are different.

Therefore, we only need to define projectors $pr_x$ and $pr_r$ that would satisfy the inference rules. It can be done if projector $pr_x(p)$ is defined recursively as

1. If $p = \rho$ then return $a$.
2. If $p = \text{update}_a(x, b, y)$ and $x \neq y$ then return $pr_x(y)$.
3. If $p = \text{update}_a(x, b, y)$ and $x = y$ then return $b$.
4. If $p = \text{update}_r(x, z, y)$ then return $pr_r(y)$.

and projector $pr_r(p)$ is defined as

17
1. If \( p = \rho \) then return \( p \).
2. If \( p = \text{update}_a(x, b, y) \) then return \( \text{update}_a(x, b, \text{pr}_r(y)) \).
3. If \( p = \text{update}_r(x, z, y) \) and \( x \neq y \) then return \( \text{update}_r(x, b, \text{pr}_r(y)) \).
4. If \( p = \text{update}_r(x, z, y) \) and \( x = y \) then return \( \text{update}_r(x, b, b) \).

**Theorem 2** Defined above type \( \text{RecPair} \) satisfies inference rules I - XXI of Section 2.

**Proof** Routine induction over co-inductive type \( \text{RecPair} \) using definitions of functions \( \text{pr}_r \) and \( \text{pr}_r \). \( \square \)

An arbitrary family of reference types \( \Upsilon(S@n) \) can be defined similarly using parametrized inductive types from [10] and [9].

**Theorem 3** For any reference type signature \( S \) there exists a family of type \( \Upsilon(S@n) \) that satisfies rules I - XX of Section 3 \( \square \)


5 Conclusion

The paper introduced reference type constructor to represent circular data structures in Type Theory. Each reference type includes a constant \( \rho \), destructor \( \text{pr} \) that returns components of type elements and operation \( \text{update} \) that assigns a new value to an element component. It has been shown that as a result of side effects, \( \text{update} \) changes not only the element to which it has been applied, but the whole universe of referencing elements. As a result, \( \text{update} \) is viewed as an endomorphism on reference types.

Reference types can be used to provide simple, more elegant, Type Theory based, semantics of programming languages. As an example, author has defined such semantics for a small Java fragment in [11].

The work also states that any reference type element is equal to a canonical element. An element is canonical if it can be represented by a term that includes only constant \( \rho \) and operation \( \text{update} \). Under the assumption that all canonical elements are different, it means that the reference type can be described as a free algebra, generated from constant \( \rho \) by the operation \( \text{update} \).

The question whether a stronger equality on reference type elements can be used is left open. Another possible extension of this work is defining a subtyping relation on reference types.

6 Acknowledgments

Author would like to thank Robert L. Constable, Tim Teitelbaum, Gregory Morrisett, Richard Shore, and participants of PRL Seminar at Cornell for useful comments on this work.

References


