Object-Orientation in the UTP

Thiago Santos[1], Ana Cavalcanti[2], and Augusto Sampaio[1]

[1] Centre of Informatics, Federal University of Pernambuco
P.O. Box 7851, 50732-970 Recife-PE, Brazil
[2] Department of Computer Science, University of York,
Heslington York, YO10 5DD, United Kingdom

Abstract. In this paper, we study object-oriented programming concepts present in languages like Java and C++ in the framework of the Unifying Theories of Programming (UTP). This work shows how subtyping, data inheritance, (mutually) recursive methods, and dynamic binding can be described in the UTP by combining and extending the theories of designs and higher-order procedures. A distinguishing feature of our approach is modularity: following the style of the UTP, we deal with each concept in isolation; this makes our theory convenient to model integrated languages that include constructs from several paradigms.

1 Introduction

Since object-oriented languages have been widely used to develop software for different domains of application, there has been a strong need to understand and describe the meaning of object-oriented programs. Approaches like operational [1, 2], denotational [3], and algebraic semantics [4, 5] have been used to describe languages and how their concepts are related.

In the Unifying Theories of Programming (UTP) [6], Hoare and He establish a framework to allow reasoning about different programming paradigms using a relational calculus. In this paper, we describe in the UTP a subset of the object-oriented programming concepts found in languages like Java and C++. Our theory is an extension of the theories of designs and higher-order procedures.

In [7], we can find a description in the UTP of an object-oriented (OO) language that handles pointers and visibility mechanisms, among other OO features. The authors also present a set of rules related to refinement. However, (mutually) recursive methods are not described explicitly. Another example of an OO language described in UTP is presented in [8], where the semantics of TCOZ [9, 10], a language that combines processes, classes and time, is defined.

We target general object-oriented concepts, rather than any specific language. We introduce concepts of OO languages progressively and in isolation. We cover subtyping, single inheritance, dynamic binding, and (mutual) recursion, assuming a copy semantics. By introducing these features independently we provide a general theory of object-orientation that can be combined with other UTP theories in the usual way. In particular, our long-term goal is to define a combined theory for reactive, object-oriented designs, and use it to give a semantics to

OhCircus [11]. This is an object-oriented extension of *Circus* [12], a combination of Z [13] and CSP [14] whose semantics is based on the UTP.

In our theory, a class declaration is not a single block, as usual in object-oriented languages. We have separated constructs to declare a class and its immediate superclass, to declare an attribute, and to declare a method.

Example 1. Consider a simple banking system; we define a class *Account*, and its attributes and methods as follows:

class *Account*;
att *Account id* : $\mathbb{Z}$, *balance* : $\mathbb{Z}$;
meth *Account credit* = (**val** $x : \mathbb{Z} \bullet$ **self**.*balance* := **self**.*balance* + x)

The declarations of the attributes and methods are independent, and combined in sequence. In particular, the declarations of the attributes and methods have to indicate their classes. We show that this approach simplifies the semantics, and makes the treatment of (mutual) recursion straightforward, as it should be.

It is well-known that, in the semantics of an object-oriented language, the types of the variables play a central role due to subtyping and dynamic binding [15]. In our theory, we have a collection of observational variables that are used to model declarations. They record important typing information and are used in the semantics of commands. We also drop the assumption that expressions are total; this is not realistic for object-oriented languages due to the possibility of attempts to access attributes and methods of a "**null** object" (that is, "**null** pointer exceptions"). As a consequence, we have to characterize well-defined expressions, and extend the semantics of assignments and conditionals.

Method names are also part of the alphabet of our theory. Their values are parametrised programs [16]. Their treatment follows the approach originally proposed in [17], and adopted in [15] to handle methods. It is also the approach followed in the UTP for higher-order procedures.

Dynamic binding is reflected in the value of a method variable. It is a conditional that checks the type of the target object and determines the right program that defines the behaviour of the method in each case. In this way, we capture dynamic binding in isolation. This follows the style adopted in an algebraic semantics for object-orientation [5].

This paper is organized as follows. In Section 2, we introduce the alphabet of our theory: observational variables related to the OO concepts of subtyping, inheritance and dynamic binding. In Section 3, we define class, attribute and method declaration. In Section 4, we review the concept of variables, to include type information explicitly. In Section 5, we describe well-definedness rules for expressions and the meaning of object creation, type test, type cast and attribute access. In Section 6, we review the semantics of commands emphasizing method call. Finally, in Section 7, we discuss related and future work.

2 Observational Variables

In addition to the programming variables and their dashed counterparts, and to ok and ok' from the theory of designs, our theory includes two new observational variables: one to record the subclass relation; and another to record the types of attributes associated to a given class. For classes, we introduce:

$$\Gamma_{cls} : name \mapsto name$$

This is a mapping from class names to the corresponding name of their immediate superclasses. This observational variable allows us to introduce new types other than the primitive ones: booleans ($\mathbb{B}$) and integers ($\mathbb{Z}$).

Our second observational variable holds information about the attributes of each class and their types:

$$\Gamma_{att} : name \mapsto \{name \mapsto type\}$$

This is a mapping from a class name to a description of its attributes, which maps each attribute name to its type; *type* stands for any primitive type, or any name in $\operatorname{dom} \Gamma_{cls}$.

The method names are also part of the alphabet of our theory. Their values are parametrised programs ($pds \bullet p$), where *pds* is a list of parameter declarations, and p is a program: the body of the parametrised program, which uses the parameters. Value (**val**), result (**res**), and value-result (**valres**) parameters are allowed. The notation *pds* stands for any parameter declaration list, possibly including the three parameter passing mechanisms. For example, **val** $x : X$; **res** $y : Y$; **valres** $z : Z$, is a valid instance of *pds*, where x, y, and z are variable names and X, Y, and Z are types. The function *types* applied to a list of parameter declarations returns the parameter types as a set. For example, *types* applied to the previous example yields $\{X, Y, Z\}$.

In bodies of the values of the observational variables named after methods nested conditionals with each branch representing the meaning of a method redefinition. For instance, considering that C is a subclass of B, which itself is a subclass of A, and that m is a parameterless method defined in A (with body ma), and redefined in both B and C (with bodies mb and mc), the m value is:

$$\begin{array}{l}\textbf{valres self} : \textbf{\textit{Object}} \bullet \\ \quad mc \lhd \textbf{self is } C \rhd (mb \lhd \textbf{self is } B \rhd (ma \lhd \textbf{self is } A \rhd \bot))\end{array}$$

Based on the type of the current object (**self**) the nested conditional allows selection of the more specialized version of m. When m is not defined for a given class, then the behaviour of a call to m with an object of this class as a target is unpredictable ($\bot$). The condition **self is** N, for a class name N, checks whether the value of **self** is an object of class N, or one of its subclasses. This is why the type of the object held by **self** is tested from the more specialized subclass to the less specialized one in the class hierarchy.

Finally, for each programming variable x, besides x itself, and x', we include in the alphabet two more observational variables (xt and xt') to record the

declared type of x. This is potentially different from the actual (runtime) type of the value of x, which can be an object of a subclass of the type recorded in xt, when this is a class.

Object-oriented features such as attribute overriding, variable shading, and the use of **super** or related notations (to refer to elements of a superclass) are not considered here because they are only syntactic abbreviations that can be easily eliminated by preprocessing. We also consider that the names of classes, attributes, methods (except for method overriding), local variables and parameters are different. This allows us to write simpler predicates while not imposing any relevant practical limitation.

3 Declarations

In this section we provide the meaning, as designs, for class, attribute and method declarations.

3.1 Classes

As mentioned before, our aim is to add each feature of object-orientation in isolation. In this direction, a class declaration introduces just a new type, without any attribute or method. We use the notation of designs in the UTP to define each feature. The declaration **class** A, explained in the sequel, stands for the design:

$$\textbf{class } A =_{df} \begin{pmatrix} A \neq \textbf{\textit{Object}} \wedge \\ A \notin \operatorname{dom} \Gamma_{cls} \end{pmatrix} \vdash \begin{pmatrix} \Gamma'_{cls} = \Gamma_{cls} \cup \{A \mapsto \textbf{\textit{Object}}\} \wedge \\ w' = w \end{pmatrix}$$

$$\textbf{where } w = in\alpha(\textbf{class } A) \setminus \{\Gamma_{cls}\}.$$

By default, every class has as parent a special class named ***Object***, which has no attributes or methods. It cannot be redeclared, so the precondition of the design above requires A to be different from ***Object***. It also requires A to be a new class name: not in the domain of Γ_{cls}. The postcondition of the design specifies that the declaration includes A in Γ_{cls} with ***Object*** recorded as its immediate superclass. It also specifies that no other observational variable w is modified. In the UTP, $in\alpha(\textbf{class } A)$ is the input alphabet of the program **class** A, which includes all undashed observational variables of its alphabet. For the declaration **class** A **extends** B, we have:

$$\textbf{class } A \textbf{ extends } B =_{df} \begin{pmatrix} A \neq \textbf{\textit{Object}} \wedge \\ A \notin \operatorname{dom} \Gamma_{cls} \wedge \\ B \in \operatorname{dom} \Gamma_{cls} \end{pmatrix} \vdash \begin{pmatrix} \Gamma'_{cls} = \Gamma_{cls} \cup \{A \mapsto B\} \wedge \\ w' = w \end{pmatrix}$$

$$\textbf{where } w = in\alpha(\textbf{class } A \textbf{ extends } B) \setminus \{\Gamma_{cls}\}.$$

This introduces a record of class A with B as immediate superclass in Γ_{cls}. The class B needs to have been previously declared.

Using Γ_{cls}, we can define the subtyping relation $A \preceq B$, which holds if, and only if, both types are defined in Γ_{cls} and A is associated to B in the reflexive and transitive closure of Γ_{cls}, or if both types are equal and primitive. The inclusion of primitive types into the subtyping relation allows us to simplify definitions.

$$A \preceq B \equiv (A \in \operatorname{dom} \Gamma_{cls} \land (A, B) \in \Gamma_{cls}^{*}(\!| \{A\} |\!)) \lor (A \in \{\mathbb{B}, \mathbb{Z}\} \land A = B)$$

Example 2. Consider again a simple banking application, with classes *Account*, which depicts an account of a bank, *BAccount*, an extension of *Account* to hold bonus information, *Contact*, to hold traditional contact information, and *EContact*, an extension of *Contact* to hold electronic contact information. The meaning of the sequence of declarations of these classes is the design below.

class *Account*;
class *BAccount* **extends** *Account*;
class *Contact*;
class *EContact* **extends** *Contact*

$\equiv$

$$\begin{array}{l}
Account \neq \textbf{\textit{Object}} \land Account \notin \operatorname{dom} \Gamma_{cls} \vdash \\
\quad \Gamma'_{cls} = \Gamma_{cls} \cup \{Account \mapsto \textbf{\textit{Object}}\}; \\
BAccount \neq \textbf{\textit{Object}} \land BAccount \notin \operatorname{dom} \Gamma_{cls} \land Account \in \operatorname{dom} \Gamma_{cls} \vdash \\
\quad \Gamma'_{cls} = \Gamma_{cls} \cup \{BAccount \mapsto Account\}; \\
Contact \neq \textbf{\textit{Object}} \land Contact \notin \operatorname{dom} \Gamma_{cls} \vdash \\
\quad \Gamma'_{cls} = \Gamma_{cls} \cup \{Contact \mapsto \textbf{\textit{Object}}\}; \\
EContact \neq \textbf{\textit{Object}} \land EContact \notin \operatorname{dom} \Gamma_{cls} \land Contact \in \operatorname{dom} \Gamma_{cls} \vdash \\
\quad \Gamma'_{cls} = \Gamma_{cls} \cup \{EContact \mapsto Contact\}
\end{array}$$

The meaning of sequence in our theory is the same as that in the UTP.

3.2 Attributes

We can introduce attributes in Γ_{att} for those classes already in Γ_{cls}. All attributes are public. To introduce an attribute x of type T in class A we use the design:

$$\begin{array}{l}
\textbf{att}\ A\ x : T =_{df} \\
\left(\begin{array}{l} A \in \operatorname{dom} \Gamma_{cls} \land \\ x \notin \operatorname{dom} \bigcup \{\Gamma_{att}(N) \mid N \in \operatorname{dom} \Gamma_{att}\} \land \\ T \in \{\mathbb{B}, \mathbb{Z}\} \cup \operatorname{dom} \Gamma_{cls} \end{array}\right) \vdash \\
\quad \left(\begin{array}{l} \left(\begin{array}{l} A \notin \operatorname{dom} \Gamma_{att} \land \\ \Gamma'_{att} = \Gamma_{att} \cup \{A \mapsto \{x \mapsto T\}\} \end{array}\right) \lor \\ \left(\begin{array}{l} A \in \operatorname{dom} \Gamma_{att} \land \\ \Gamma'_{att} = \Gamma_{att} \oplus \{A \mapsto (\Gamma_{att}(A) \cup \{x \mapsto T\})\} \end{array}\right) \end{array}\right) \land\ w' = w \\
\textbf{where}\ w = in\alpha(\textbf{att}\ A\ x : T) \setminus \{\Gamma_{att}\}.
\end{array}$$

If we try to declare an attribute of a class that has not been declared previously, with a name that was already used, or of a type that is not primitive or present in $\operatorname{dom} \Gamma_{cls}$, the declaration fails.

We can declare several attributes simultaneously, with the obvious meaning.

$$\textbf{att}\ A\ x : T, y : U, \ldots \equiv \textbf{att}\ A\ x : T;\ \textbf{att}\ A\ y : U;\ \ldots$$
$$\textbf{att}\ A\ x : T, B\ y : U, \ldots \equiv \textbf{att}\ A\ x : T;\ \textbf{att}\ B\ y : U;\ \ldots$$

Our notation allows interleaving concerning the order of class, attribute and method declaration. For example, the sequence below is allowed.

$$\textbf{class}\ A;\ \textbf{att}\ A\ x : \mathbb{Z};\ \textbf{class}\ B\ \textbf{extends}\ A;\ \textbf{att}\ A\ y : \mathbb{B};\ \textbf{att}\ B\ z : A$$

In this case, the attribute y of the class A is declared after the declaration of the class B. In fact, if we have recursive classes, the required order of the declaration is different from that adopted in languages where classes are blocks. For example, if a class A has an attribute x whose type is a subclass B of A, then the following order of declaration is required.

$$\textbf{class}\ A;\ \textbf{class}\ B\ \textbf{extends}\ A;\ \textbf{att}\ A\ x : B$$

Transforming the class-based declarations of an object-oriented language into an appropriate sequence of class and attribute declarations is a simple task. For methods, similar considerations apply; mutual recursion, however, is further discussed in the Section 6.4.

Example 3. This example adds some attributes to the classes of Example 2.

$$\begin{array}{l}
\textbf{att}\ \mathit{Account}\ \mathit{id} : \mathbb{Z}, \mathit{balance} : \mathbb{Z}, \mathit{contact} : C; \\
\textbf{att}\ \mathit{BAccount}\ \mathit{bonus} : \mathbb{Z}; \\
\textbf{att}\ \mathit{Contact}\ \mathit{phone} : \mathbb{Z}; \\
\textbf{att}\ \mathit{EContact}\ \mathit{icq} : \mathbb{Z}
\end{array}$$

$\equiv$

$$\begin{array}{l}
\mathit{Account} \in \operatorname{dom} \Gamma_{cls} \land \mathit{id} \notin \operatorname{dom} \bigcup \{\Gamma_{att}(N) \mid N \in \operatorname{dom} \Gamma_{att}\} \land \\
\quad \mathbb{Z} \in \{\mathbb{B}, \mathbb{Z}\} \cup \operatorname{dom} \Gamma_{cls} \vdash \\
\qquad \left(\begin{array}{l} \left(\begin{array}{l} \mathit{Account} \notin \operatorname{dom} \Gamma_{att} \land \\ \Gamma'_{att} = \Gamma_{att} \cup \{\mathit{Account} \mapsto \{\mathit{id} \mapsto \mathbb{Z}\}\} \end{array} \right) \lor \\ \left(\begin{array}{l} \mathit{Account} \in \operatorname{dom} \Gamma_{att} \land \\ \Gamma'_{att} = \Gamma_{att} \oplus \{\mathit{Account} \mapsto (\Gamma_{att}(\mathit{Account}) \cup \{\mathit{id} \mapsto \mathbb{Z}\})\} \end{array} \right) \end{array} \right) \\
;\ \mathit{Account} \in \operatorname{dom} \Gamma_{cls} \land \mathit{balance} \notin \operatorname{dom} \bigcup \{\Gamma_{att}(N) \mid N \in \operatorname{dom} \Gamma_{att}\} \land \ldots
\end{array}$$

We apply the design definition of attribute declaration to each element of the sequence, starting with the attribute *id*, and ending with *icq*.

For a given class N we define $\mathcal{C}(N)$ to be a mapping that records all the attributes of N, including those declared in its superclasses. We define $\mathcal{C}(N)$ in terms of Γ_{cls}, and Γ_{att}.

$$\mathcal{C}(N) = \bigcup \Gamma_{att}(\!|\ \{\Gamma^{+}_{cls}(\!|\ \{N\}\ |\!) \cup \{N\}\} \setminus \{\textbf{\textit{Object}}\}\ |\!)$$

In words, $\mathcal{C}(N)$ contains all the attribute definitions of all classes related to N by the closure of the superclass relation, and N itself.

3.3 Methods

For a method declaration to succeed, the class to which it is associated must have been introduced before, and all formal parameters, passed as value (**val**), result (**res**) or value-result (**valres**), must have types introduced in Γ_{cls} or primitive ones. In any case, the meaning depends on whether the method is being declared for the first time or not. If it is ($m \notin \alpha(\textbf{meth}\ A\ m = (pds \bullet p))$), then the definition below applies. The new name m is introduced in the alphabet using a variable declaration. The design defines the value of m.

$$
\begin{array}{l}
\textbf{meth}\ A\ m = (pds \bullet p) =_{df} \\
\quad \textbf{var}\ m\ ; \\
\qquad \begin{pmatrix} A \in \operatorname{dom} \Gamma_{cls} \wedge \\ \forall\, t \in types(pds) \bullet t \in \{\mathbb{B}, \mathbb{Z}\} \cup \operatorname{dom} \Gamma_{cls} \end{pmatrix} \vdash \begin{pmatrix} m' = program \\ \wedge w' = w \end{pmatrix}
\end{array}
$$

$$
\begin{array}{l}
\textbf{provided}\ m \notin \alpha(\textbf{meth}\ A\ m = (pds \bullet p)) \\
\textbf{where}\ program = \textbf{valres self} : \textbf{\textit{Object}};\ pds \bullet (p \lhd \textbf{self is}\ A \rhd \bot) \\
\textbf{and}\ w = in\alpha(\textbf{meth}\ A\ m = (pds \bullet p)) \setminus \{m\}.
\end{array}
$$

The value of m is a parametrised program. Methods are higher-order, predicate-valued variables as in the theory of higher-order procedures and parameters of the UTP. The parameters of m are those in pds and an extra parameter **self** to represent the target of a call; its type is ***Object***. Just as in **var** x, where we introduce in the alphabet new variables x and x', with **meth** $A\ m$, we introduce in the alphabet the variables m and m'. At the same time, we use a design to define the value of m'.

For the case of a redefinition of a method m ($m \in \alpha(\textbf{meth}\ A\ m = (pds \bullet p))$), we have the definition below.

$$
\begin{array}{l}
\textbf{meth}\ A\ m = (pds \bullet p) =_{df} \\
\quad \left(\begin{array}{l} A \in \operatorname{dom} \Gamma_{cls} \wedge \\ \forall\, t \in types(pds) \bullet t \in \{\mathbb{B}, \mathbb{Z}\} \cup \operatorname{dom} \Gamma_{cls} \wedge \\ \exists\, q \bullet m = \textbf{valres self} : \textbf{\textit{Object}};\ pds \bullet q \end{array} \right) \vdash \\
\qquad \left(\begin{array}{l} \exists\, q \bullet m = (\textbf{valres self} : \textbf{\textit{Object}};\ pds \bullet q) \\ \wedge m' = \textbf{valres self} : \textbf{\textit{Object}};\ pds \bullet join(A, p, q) \\ \wedge w' = w \end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
\textbf{provided}\ m \in \alpha(\textbf{meth}\ A\ m = (pds \bullet p)) \\
\textbf{where}\ w = in\alpha(\textbf{meth}\ A\ m = (pds \bullet p)) \setminus \{m\}, \\
\textbf{and} \\
join(A, a, \bot) = a \lhd \textbf{self is}\ A \rhd \bot \\
join(A, a, b_l \lhd \textbf{self is}\ B \rhd b_r) = \\
\quad \begin{cases} a \lhd \textbf{self is}\ A \rhd (b_l \lhd \textbf{self is}\ B \rhd b_r), if\ A \preceq B \wedge A \neq B \\ b_l \lhd \textbf{self is}\ B \rhd join(A, a, b_r) \qquad , otherwise \end{cases}
\end{array}
$$

It is worth emphasizing that the definition of *join* deals with redefinition of m both in superclasses and in subclasses A of the class where the original definition

is placed. The use of *join* allows us to introduce the method values, expressed as (parametrised) programs [16], in a form where dynamic binding is already resolved, as in algebraic methods [18, 5], and in the weakest precondition approach [15]. The special variable **self** denotes the instance of the target of the method call. All references to attributes on method bodies must be prefixed with **self**; variables without this prefix are formal parameters or local variables.

If the method is a redefinition, the method signatures must be exactly the same, and a new conditional is built to take into account the class hierarchy. Finally, if we try to make a call to m with an object of an inappropriate type as a target, the result is $\bot$ as well. Thus, a program with invalid method calls has unpredictable behavior.

We give the meaning of a parametrised program as a function from a value or a variable name to a program (or predicate). We consider each of the mechanisms of parameter passing individually; the definitions reflect the standard way of implementing them.

For a value parameter, the semantics is a higher-order function that takes the value of the argument and gives the program that declares the formal parameter as a local variable and initializes it with the argument.

$$(\mathbf{val}\ v : T \bullet p) = (\lambda\, w : T \bullet (\mathbf{var}\ v : T;\ v := w;\ p;\ \mathbf{end}\ v))$$

A function that models a parametrised program with a parameter passed by result takes as argument the name of a variable: an element of the syntactic category $\mathcal{N}$. This is the argument in a method call.

$$(\mathbf{res}\ v : T \bullet p) = (\lambda\, w : \mathcal{N} \bullet (\mathbf{var}\ v : T;\ p;\ w := v;\ \mathbf{end}\ v))$$

In this case, the local variable corresponding to the formal parameter is not initialized; its value is assigned to the argument.

For a value-result parameter, the definition is as expected: the local variable is initialized and then assigned to the argument in the end.

$$(\mathbf{valres}\ v : T \bullet p) = (\lambda\, w : \mathcal{N} \bullet (\mathbf{var}\ v : T;\ v := w;\ p;\ w := v;\ \mathbf{end}\ v))$$

The parameter of the function is again a program variable. This is an abstraction over three arguments: a variable, its dashed counterpart, and the type variable.

$$(\lambda\, x : \mathcal{N} \bullet p)(y) = p[y, y', yt/x, x', xt]$$

In this case, lambda-reduction is extended to cope with variable parameters: elements of the syntactic category $\mathcal{N}$. This semantics for methods was presented in [11].

Example 4. In this example we show the semantics of method declarations, considering that Γ_{cls} is the one defined in Example 2 and Γ_{att} that defined in Example 3. There is a method *credit* for *Account* and we redefine it for class

BAccount to increase the value of a bonus variable before executing the *credit* behaviour.

$$\begin{array}{l}\textbf{meth}\ \mathit{Account}\ \mathit{credit} = (\textbf{val}\ x : \mathbb{Z} \bullet \\ \quad\quad \textbf{self}.\mathit{balance} := \textbf{self}.\mathit{balance} + x); \\ \textbf{meth}\ \mathit{BAccount}\ \mathit{credit} = (\textbf{val}\ x : \mathbb{Z} \bullet \\ \quad\quad \textbf{self}.\mathit{bonus} := \textbf{self}.\mathit{bonus} + 1;\ \textbf{self}.\mathit{balance} := \textbf{self}.\mathit{balance} + x)\end{array}$$

We observe that, in the body of the redefinition of *credit* for *BAccount* we have a repetition of the code in the body of *credit* as defined for *Account*. In a programming language, this is likely to be written as $\textbf{super}.\mathit{credit}(x)$ or using some other similar notation that avoids code repetition. As we explained in Section 2, however, semantically, these constructs can be removed using a copy rule. For this reason, do not consider such issue here. The meaning for the two method declarations is given by the sequence:

$$\begin{array}{l}\textbf{var}\ \mathit{credit}\ ; \\ \left(\mathit{Account} \in \operatorname{dom} \Gamma_{cls} \wedge \forall\, t \in \mathit{types}(\textbf{val}\ x : \mathbb{Z}) \bullet t \in \{\mathbb{B}, \mathbb{Z}\} \cup \operatorname{dom} \Gamma_{cls}\right) \\ \vdash \\ \left(\mathit{credit}' = \left(\begin{array}{l}\textbf{valres self} : \textbf{\textit{Object}};\ \textbf{val}\ x : \mathbb{Z} \bullet \\ \quad \textbf{self}.\mathit{balance} \ldots \lhd \textbf{self is}\ \mathit{Account} \rhd \bot\end{array}\right)\right) \\ ; \\ \left(\begin{array}{l}\mathit{BAccount} \in \operatorname{dom} \Gamma_{cls} \wedge \forall\, t \in \mathit{types}(\textbf{val}\ x : \mathbb{Z}) \bullet t \in \{\mathbb{B}, \mathbb{Z}\} \cup \operatorname{dom} \Gamma_{cls} \wedge \\ \exists\, q \bullet m = (\textbf{valres self} : \textbf{\textit{Object}};\ \textbf{val}\ x : \mathbb{Z} \bullet q)\end{array}\right) \\ \vdash \\ \left(\begin{array}{l}\mathit{credit}' = \textbf{valres self} : \textbf{\textit{Object}};\ \textbf{val}\ x : \mathbb{Z} \bullet \\ \quad\quad \mathit{join}\left(\begin{array}{l}\mathit{BAccount}, \\ (\textbf{self}.\mathit{bonus} := \textbf{self}.\mathit{bonus} + 1;\ \ldots), \\ \left(\textbf{self}.\mathit{balance} \ldots \lhd \textbf{self is}\ \mathit{Account} \rhd \bot\right)\end{array}\right)\end{array}\right)\end{array}$$

The value associated to *credit* after the second design is of the following form:

$$\begin{array}{l}\textbf{valres self} : \textbf{\textit{Object}};\ \textbf{val}\ x : \mathbb{Z} \bullet \\ \quad \textbf{self}.\mathit{bonus} \ldots \lhd \textbf{self is}\ \mathit{BAccount} \rhd \\ \quad\quad (\textbf{self}.\mathit{balance} \ldots \lhd \textbf{self is}\ \mathit{Account} \rhd \bot)\end{array}$$

The conditional type test created by *join* selects the appropriate command.

4 Variables

In [6], type information is not explicitly recorded for the variables. In an object-oriented language, where types play a central role, this is not appropriate. In our theory, the values of the variables are pairs, whose first element is the (runtime) type of the current value of the variable and the second is the value itself.

We give semantics to the construct **var** $x : T$, where T is the static (declared) type of the variable x. The new definition for a **var** that declares the types of the variables that it introduces is as follows:

$$\begin{array}{l}\textbf{var}\ x : T =_{df} \\ \quad \textbf{var}\ x, xt;\ T \in \{\mathbb{B}, \mathbb{Z}, \textbf{\textit{Object}}\} \cup \operatorname{dom} \Gamma_{cls} \vdash xt' = T \wedge x' \in T \wedge w' = w\end{array}$$

$$\textbf{where}\ w = in\alpha(\textbf{var}\ x : T) \setminus \{x, xt\}.$$

We use the existing **var** construct to introduce both x and xt in the alphabet. In the design, we check that T is a valid type. In this case, the type of x is defined to be T, and an arbitrary element of T is chosen as its initial value. All the other variables are not changed. In assignments to x, the pair (e_t, e_v) which denotes the value may change, but xt does not.

To complete this definition, we need to define the set of elements of a class type C. These are pairs in which the first element is C, and the second element is either the special value **null** or a mapping (record) that associates a value to the name of each of the attributes of C, and the values of the types determined by the subclasses of C. A formal definition is a function that takes Γ_{cls} and Γ_{att} as parameters; a similar function is specified in [15].

As within the UTP, **var** $x : T$ is a non-homogeneous relation: the alphabet of **var** $x : T$ does not include x or xt. The definition of **end** $x : T$ (the construct used to finalize the scope of x) is similar to that in the UTP. There are no concerns about type at the end of the scope of a variable, but we need to close the scope of both x and xt.

This discussion about the structure of values is extremely important to guide our concepts of what is an object value and how we can guarantee the correctness of assignments, and method requests, in an OO context. This interpretation of variables and values is not against the principles of the UTP; we have just made explicit representation of values in order to handle the concepts of OO.

5 Expressions

In this section we specify well-definedness rules for expressions, and the semantics of object creation, type test, type cast and attribute accesses.

5.1 Well-definedness

Our theory includes new forms of expression e characterized by the following BNF-like definition.

$$\begin{array}{l}e ::= v \mid le \mid \textbf{new}\ N \mid e\ \textbf{is}\ N \mid (N)e \mid f(e) \mid \textbf{null} \\ le ::= x \mid \textbf{self} \mid le.x\end{array}$$

Here v is a primitive or object value. The expressions le, named left expressions, can be a variable, the special variable named **self**, or a sequence of dot-separated

names. The expression **new** N stands for object creation, e **is** N for type test, and $(N)e$ for type cast. There is also a group of built-in operations over expressions, like, for instance, arithmetic and relational operators denoted by $f(e)$.

For an expression e, we write e_t to denote the first element of the value of e, and e_v to denote the second element. In other words, e_t is the type of the value of e, and e_v is the value itself forming a pair (e_t, e_v). The construct **null** actually stands for a family of values, one for each class. The type held by e_t in this case is inferred from the context. For instance, in an assignment $x := \mathbf{null}$, $e_t = xt$. Which means that the runtime type of **null** is the declared type of x.

The well-definedness of expressions is specified by a function named $\mathcal{D}$. If an expression has a primitive value, it is well-defined if the value belong to the set of possible values of the type. For objects, we must check if the type belongs to $\operatorname{dom} \Gamma_{cls}$, and if the value belongs to the type.

Primitive Values	**Objects**
$\mathcal{D}((\mathbb{B}, v)) \equiv v \in \mathbb{B}$	$\mathcal{D}((T, \mathbf{null})) \equiv T \in \operatorname{dom} \Gamma_{cls}$
$\mathcal{D}((\mathbb{Z}, v)) \equiv v \in \mathbb{Z}$	$\mathcal{D}((T, v)) \equiv T \in \operatorname{dom} \Gamma_{cls} \land v \in T$

Variables are well-defined if their types are either primitive or present in the in $\operatorname{dom} \Gamma_{cls}$. If a variable has the special name **self**, it cannot be of a primitive type.

Variables

$$\mathcal{D}(x) \equiv xt \in \{\mathbb{B}, \mathbb{Z}, \textbf{\textit{Object}}\} \cup \operatorname{dom} \Gamma_{cls}$$
$$\mathcal{D}(\mathbf{self}) \equiv \mathbf{self}t \in \{\textbf{\textit{Object}}\} \cup \operatorname{dom} \Gamma_{cls}$$

An attribute access $le.x$ is valid only if le is well-defined, the value of le is different from **null** and x is in the domain of le.

Attribute Accesses

$$\mathcal{D}(le.x) \equiv \mathcal{D}(le) \land le_v \neq \mathbf{null} \land x \in \operatorname{dom} le_v$$

A **new** N declaration is valid only if the class N is recorded in $\operatorname{dom} \Gamma_{cls}$. The type test and casting can be done only if e is a well-defined expression and N belongs to $\operatorname{dom} \Gamma_{cls}$.

Typing

$$\mathcal{D}(\mathbf{new}\ N) \equiv N \in \operatorname{dom} \Gamma_{cls}$$
$$\mathcal{D}(e\ \mathbf{is}\ N) \equiv \mathcal{D}(e) \land N \in \operatorname{dom} \Gamma_{cls} \land e_t \preceq N$$
$$\mathcal{D}((N)e) \equiv \mathcal{D}(e) \land N \in \operatorname{dom} \Gamma_{cls} \land e_t \preceq N$$

The well-definedness restrictions for built-in operations for primitive types, $f(e)$, are defined individually and are very similar. We show the example of the remainder of a division operator, usually written '%' in programming languages:

Remainder

$$\mathcal{D}(x\%y) \equiv \mathcal{D}(x) \land \mathcal{D}(y) \land xt = \mathbb{Z} \land yt = \mathbb{Z} \land y \neq 0$$

In Section 6.1, we use the function $\mathcal{D}$ on expressions to define well-definedness rules for commands.

5.2 Object Creation

An object value is a pair $(type, value)$: the *type* is a class name and the *value* is a mapping from names to attribute values. Using Γ_{cls} and Γ_{att} to recover attributes and inheritance information, we provide a definition for **new** as:

$$\textbf{new } N \equiv \left(N, \left\{ \begin{array}{l} x : \operatorname{dom} \Gamma_{cls}; \\ t : \{\mathbb{B}, \mathbb{Z}\} \cup \operatorname{dom} \Gamma_{cls}; \\ v : \mathbb{B} \cup \mathbb{Z} \cup \{\, T : \operatorname{dom} \Gamma_{cls};\ i : T \bullet i \,\} \\ | \\ (\mathcal{C}(N)(x) = \mathbb{B} \land t = \mathbb{B} \land v = \textbf{false}) \lor \\ (\mathcal{C}(N)(x) = \mathbb{Z} \land t = \mathbb{Z} \land v = 0) \lor \\ (\exists\, T : \operatorname{dom} \Gamma_{cls} \bullet \mathcal{C}(N)(x) = T \land t = T \land v = \textbf{null}) \\ \bullet\ x \mapsto (t, v) \end{array} \right\} \right)$$

This definition says that the value of a newly created object is a mapping from attribute names to values that associates all boolean attributes to **false**, all integer attributes to 0, and all class-typed attributes to **null**. For example, the value of **new** *BAccount* is:

$$(BAccount, \{id \mapsto (\mathbb{Z}, 0), balance \mapsto (\mathbb{Z}, 0), contact \mapsto (Contact, \textbf{null})\})$$

5.3 Type Test

The expression e **is** N is a boolean that indicates whether the value of e belongs to the class N or one of its subclasses.

$$e \textbf{ is } N \equiv (\mathbb{B}, e_t \preceq N)$$

For example:

$$\begin{array}{rl} (\textbf{new} BAccount) \textbf{ is } Account & \equiv (BAccount, \{\ldots\}) \textbf{ is } Account \\ & \equiv (\mathbb{B}, BAccount \preceq Account) \\ & \equiv (\mathbb{B}, \textbf{true}) \end{array}$$

This is justified by the definitions of **new**, type test, and $\preceq$, if we assume that Γ_{cls} is as defined in Example 2.

5.4 Type Cast

The result of a casting $(N)e$ is the expression e itself, if the casting is well defined. Since we are only defining the meaning of well-defined expressions, our specification is surprisingly trivial.

$$(N)e \ \equiv e$$

For example, provided that $BAccount \preceq Account$:

$$\begin{array}{rl} (Account) \textbf{ new } BAccount & \equiv (Account)(BAccount, \{\ldots\}) \\ & \equiv (BAccount, \{\ldots\}) \end{array}$$

In the semantics of assignments and conditionals, we guarantee that well-definedness is checked.

5.5 Attribute Access

An attribute access $le.x$ recovers from the object value mapping (le_v) the attribute named x.

$$le.x \equiv le_v(x)$$

Again, we have a very simple definition, because we are only considering well-defined attribute accesses.

6 Commands

In addition to the commands in the theory of designs, our theory includes assignments $le := e$ of a value e to a left expression le, and method calls $le.m(a)$ with target le and list of arguments a. Moreover, since expressions have changed, we need to consider well-definedness for some commands. We also consider mutual recursion. The other commands such as sequential composition ($P;\ Q$) remain unchanged.

6.1 Well-definedness

In this section, we specify well-definedness for assignments, conditionals and method calls. We consider two cases of assignments: assignments to variables, and assignments to object attributes. An assignment of an expression e to a variable x is considered well-defined if x is well-defined, e is well-defined and the type of e is a subtype of x.

Assignment to variables
$\mathcal{D}(x := e) \equiv \mathcal{D}(x) \land \mathcal{D}(e) \land e_t \preceq xt$

For an assignment of an expression e to an attribute x of le to be well-defined, the expression $le.x$ must be well-defined, e must be well-defined and the type of the expression e must be a subtype of the type of the attribute x in the class le_t ($\mathcal{C}(le_t)(x)$).

Assignment to attributes
$\mathcal{D}(le.x := e) \equiv \mathcal{D}(le.x) \land \mathcal{D}(e) \land e_t \preceq \mathcal{C}(le_t)(x)$

For a conditional to be well-defined, the conditional expression must be well-defined and yield a boolean value.

Conditional
$\mathcal{D}(P \lhd e \rhd Q) \equiv \mathcal{D}(e) \land e_t = \mathbb{B} \land \mathcal{D}(P) \land \mathcal{D}(Q)$

The well-definedness for method calls is the most extensive rule. A method call in the form $le.m(a)$ is valid if:

- le is well-defined;

- the method m is defined for the type of le;
- the value of le is different from **null**;
- to avoid aliasing, le is not passed as an argument and is not involved in any argument, or as part of a variable in these parameters. For further details about this restriction see [15];
- the types of the arguments in the list a must be compatible with the formal parameter list of m.

We present well-definedness rules according to the parameter mechanism. Starting with value parameters, we have:

$$\mathcal{D}(le.m(e)) \equiv \mathcal{D}(le) \land \mathit{compatible}(le, m) \land le_v \neq \mathbf{null} \land e_t \preceq T$$

provided $\exists\, m, p \bullet m = (\mathbf{val}\ x : T \bullet p)$,
where $\mathit{compatible}(le, m)$ is the predicate:
$\exists\, pds, p \bullet m = (pds \bullet p) \land le_t \in \mathit{scan}(p)$
and
$\mathit{scan}(\bot) = \{\}$
$\mathit{scan}(a_l \lhd \mathbf{self\ is}\ A \rhd a_r) = \{B : \mathrm{dom}\, \Gamma_{cls} \mid B \preceq A\} \cup \mathit{scan}(a_r)$

The *scan* function yields the set of class names for which the method m can have a definition different from abort. For result and value-result parameters we use the function *disjoint* described in [15], which verifies if le is involved in any of the arguments.

$$\mathcal{D}(le.m(y)) \equiv \mathcal{D}(le) \land \mathit{compatible}(le, m) \land le_v \neq \mathbf{null} \land \mathit{disjoint}(le, y) \land T \preceq y_t$$
provided $\exists\, m, p \bullet m = (\mathbf{res}\ x : T \bullet p)$

$$\mathcal{D}(le.m(z)) \equiv \mathcal{D}(le) \land \mathit{compatible}(le, m) \land le_v \neq \mathbf{null} \land \mathit{disjoint}(le, z) \land T = z_t$$
provided $\exists\, m, p \bullet m = (\mathbf{valres}\ x : T \bullet p)$

A method call with multiple arguments can be checked using combinations of these definitions.

6.2 Assignments

Now we give the semantics for assignments to variables, and assignments to attributes of object variables. In our theory, for assignments, we observe that modifying the value of method variables, the type variable xt, or Γ_{cls} and Γ_{att} is not allowed, in much the same way that assignments to ok are not allowed in the theory of designs.

If we establish the well-definedness of an assignment, we can update the value of the variable with that of the expression on the right side.

$$x := e =_{df} \mathcal{D}(x := e) \vdash x' = e \land w' = w$$

where $w = in\alpha(x := e) \setminus \{x\}$.

For example, given a variable x of type *Account* ($xt = Account$), we can calculate the meaning of the assignment $x := \mathbf{new} BAccount$ as follows, provided that y is the list of undashed variables in the alphabet, other than x, and that Γ_{cls} is as in Example 2.

$$
\begin{array}{l}
\mathcal{D}(x := (BAccount, \{\ldots\})) \vdash \\
\quad x' = (BAccount, \{\ldots\}) \wedge y' = y \\
\equiv \mathcal{D}(x) \wedge \mathcal{D}((BAccount, \{\ldots\})) \wedge BAccount \preceq xt \vdash \\
\quad x' = (BAccount, \{\ldots\}) \wedge y' = y \\
\equiv xt \in \{\mathbb{B}, \mathbb{Z}, \textbf{\textit{Object}}\} \cup \mathrm{dom}\, \Gamma_{cls} \wedge BAccount \in \mathrm{dom}\, \Gamma_{cls} \wedge \mathbf{true} \vdash \\
\quad x' = (BAccount, \{\ldots\}) \wedge y' = y \\
\equiv \mathbf{true} \vdash \\
\quad x' = (BAccount, \{\ldots\}) \wedge y' = y
\end{array}
$$

When we have to update an attribute of an object-valued expression, we must check the well-definedness of the assignment, and if it is valid, then we update the mapping that records the attribute value, maintaining the left expression type unchanged.

$$le.x := e =_{df} \mathcal{D}(le.x := e) \vdash le' = (le_t, le_v \oplus \{x \mapsto e\}) \wedge w' = w$$

$$\textbf{where } w = in\alpha(le.x := e) \setminus \alpha(le).$$

We use $\alpha(le)$ to denote a variable in the alphabet whose value is being inspected by the left-expression le. If le is a variable, then $\alpha(le)$ is the variable itself. For $x.y$ and $x.y.z$, the result is x. The equality $le' = (le_t, le_v \oplus \{\, x \mapsto e \,\})$ for the case in which le is itself an attribute access $y.z$ is an abbreviation of the equality $y' = (y_t, y_v \oplus \{\, z \mapsto y.z \oplus \{\, x \mapsto e \,\}\,\})$.

For example, given a variable x of type *Account* ($xt = Account$), which has been initialized with **new** *BAccount* ($x = (BAccount, \{id \mapsto (\mathbb{Z}, 0), \ldots\})$), we can calculate the attribute update $x.id := 1$ as follows, provided that y is the list of undashed variables in the alphabet, other than x, and that Γ_{cls} is as in Example 2.

$$
\begin{array}{l}
x.id := 1 \\
\equiv \mathcal{D}((BAccount, \{id \mapsto (\mathbb{Z}, 0), \ldots\}).id := (\mathbb{Z}, 1)) \vdash \\
\quad x' = (BAccount, \{id \mapsto (\mathbb{Z}, 0), \ldots\} \oplus \{id \mapsto (\mathbb{Z}, 1)\}) \wedge y' = y \\
\equiv \mathcal{D}((BAccount, \{id \mapsto (\mathbb{Z}, 0), \ldots\}).id) \wedge \mathcal{D}((\mathbb{Z}, 1)) \wedge \mathbb{Z} \preceq \mathcal{C}(xt)(id) \vdash \\
\quad x' = (BAccount, \{id \mapsto (\mathbb{Z}, 1), \ldots\}) \wedge y' = y \\
\equiv \mathcal{D}((BAccount, \{id \mapsto (\mathbb{Z}, 0), \ldots\})) \wedge \{id \mapsto (\mathbb{Z}, 0), \ldots\} \neq \mathbf{null} \wedge \\
id \in \mathrm{dom}\{id \mapsto (\mathbb{Z}, 0), \ldots\} \wedge \mathbf{true} \wedge \mathbb{Z} \preceq \mathbb{Z} \vdash \\
\quad x' = (BAccount, \{id \mapsto (\mathbb{Z}, 1), \ldots\}) \wedge y' = y \\
\equiv BAccount \in \{\mathbb{B}, \mathbb{Z}, \textbf{\textit{Object}}\} \cup \mathrm{dom}\, \Gamma_{cls} \wedge \mathbf{true} \wedge \mathbf{true} \wedge \mathbf{true} \wedge \mathbf{true} \vdash \\
\quad x' = (BAccount, \{id \mapsto (\mathbb{Z}, 1), \ldots\}) \wedge y' = y \\
\equiv \mathbf{true} \vdash \\
\quad x' = (BAccount, \{id \mapsto (\mathbb{Z}, 1), \ldots\}) \wedge y' = y
\end{array}
$$

Notice that if we had not initialized the variable x, the assignment would not be well-defined and would abort. The same behaviour would occur if we had tried

to access the attribute *bonus* of the *BAccount* instance: since the variable has type *Account*, we cannot access variables from its subclass instance.

6.3 Conditional

We need to redefine the conditional to consider the well-definedness of the condition.

$$P \lhd e \rhd Q =_{df} \mathcal{D}(P \lhd e \rhd Q) \land ((e_v \land P) \lor (\neg e_v \land Q))$$

For example, suppose we have that $\mathbf{self} = (BAccount, \{\ldots\})$, the type of **self** is a class, Γ_{cls} is that provided by Example 2, and both P and Q are well-defined ($\mathcal{D}(P) \land \mathcal{D}(Q) = \mathbf{true}$). The conditional $P \lhd \mathbf{self\ is}\ BAccount \rhd Q$ leads to the execution of P, as shown below.

$$\begin{array}{l} P \lhd \mathbf{self\ is}\ BAccount \rhd Q \\ \equiv \mathcal{D}(P \lhd \mathbf{self\ is}\ BAccount \rhd Q) \land \\ \quad ((\mathbb{B}, \mathbf{self}_t \preceq BAccount)_v \land P) \lor (\neg(\mathbb{B}, \mathbf{self}_t \preceq BAccount)_v \land Q)) \\ \equiv \mathcal{D}(\mathbf{self\ is}\ BAccount) \land (\mathbb{B}, \mathbf{self}_t \preceq BAccount)_t = \mathbb{B} \land \mathcal{D}(P) \land \mathcal{D}(Q) \land \\ \quad ((\mathbf{true} \land P) \lor (\mathbf{false} \land Q)) \\ \equiv \mathcal{D}(\mathbf{self}) \land BAccount \in \operatorname{dom} \Gamma_{cls} \land P \\ \equiv \mathcal{D}(\mathbf{self}) \land P \\ \equiv \mathbf{self}t \in \{\boldsymbol{Object}\} \cup \operatorname{dom} \Gamma_{cls} \land P \\ \equiv P \end{array}$$

If the type test were **false**, the branch selected would be Q. Moreover, according to the well-definedness rules for the variable **self**, it cannot be an instance of a primitive type. If this were the case, the meaning of the conditional would be abort.

6.4 Recursion

Basically, the meaning of recursion is as in the UTP: defined in terms of least fixed point. Our complete lattice is that of parametrised programs, with refinement as the partial order. The general form of a recursive method m of class A is the following.

$$\mathbf{meth}\ A\ m = \mu\, X \bullet \big(\, pds \bullet F(X)\,\big)$$

For example, the factorial function could be added to A as:

$$\mathbf{meth}\ A\ m = \mu\, X \bullet \left(\begin{array}{l} \mathbf{val}\ n : \mathbb{Z};\ \mathbf{res}\ r : \mathbb{Z} \bullet \\ \quad r := 1 \lhd n \leq 0 \rhd r := n * X(n-1, r) \end{array} \right)$$

We observe that this is not in conflict with the expected form of a method declaration, $\mathbf{meth}\ A\ m = (pds \bullet p)$, since, of course, the least fixed point operator results in a parametrised program. In particular, the parameters are

the same as those in the body of the recursion. As a matter of fact, for each parameter declaration, we take the fixed point in the lattice of parametrised programs with those parameters.

Mutual recursion is easily addressed in our theory. It can be defined as:

$$\textbf{meth}\ A\ m, B\ n = \mu\, X, Y \bullet \big(\, pds_m \bullet F(X, Y), pds_n \bullet G(X, Y)\,\big)$$

In this case, since m and n are mutually recursive, they are defined together, even though they are methods of different classes. This follows the standard approach to the definition of mutually recursive procedures. The vector of programs m, n is defined as the least fixed point of the function from vectors of programs to vectors of programs defined by the bodies of m and n: $pds_m \bullet F(X, Y)$ and $pds_n \bullet G(X, Y)$. As an example, calling the methods m or n defined below and a variable a as arguments results in the assignment of 0 to a.

$$\textbf{meth}\ A\ m, B\ n = \mu\, X, Y \bullet \begin{pmatrix} \textbf{val}\ x : \mathbb{Z};\ \textbf{res}\ i : \mathbb{Z} \bullet i := x \lhd x = 0 \rhd Y(-x, i), \\ \textbf{val}\ y : \mathbb{Z};\ \textbf{res}\ j : \mathbb{Z} \bullet X(y-1, j) \lhd x > 0 \rhd X(y+1, j) \end{pmatrix}$$

In many theories of object-orientation, mutual recursion is a difficulty. The complication is really attached to the fact that the mutually recursive methods may be declared in an independent way in separate classes. By splitting the block structure of a class into its basic semantic blocks, we trivially overcome this difficulty.

6.5 Method Call

The most interesting feature of this work is the resolution of a method call. Since we have already solved dynamic binding when dealing with the semantics of method declaration (Section 3.3), the semantics of method call is just a call to the value of the method. In other words, we have isolated the several aspects involved in a method call, so that dynamic binding is captured in the definition of the value of the method variable, which holds a parametrised program, and a method call is just a simple call to a higher-order procedure. Thus, we can defined the method call as:

$$le.m(args) =_{df} \mathcal{D}(le.m(args)) \wedge \neg(m(le, args)[\textbf{false}/okay']) \vdash m(le, args)$$

The condition $\neg(m(args)[\textbf{false}/okay'])$ is the precondition of the design that characterises the method call.

Suppose we start with $\Gamma_{cls} = \{\}$ and $\Gamma_{att} = \{\}$, and execute the declaration of classes, attributes and methods in the Examples 2 and 3. Then consider the program fragment below.

$$\begin{array}{l} \textbf{var}\ a : Account; \\ a := \textbf{new}\ BAccount; \\ a.credit(10) \end{array}$$

Due to dynamic binding, $a.credit(10)$ must execute the body of the method *credit* defined for the subclass *BAccount*. As described in Section 3, we have solved this problem using a conditional test over the special variable named **self**. Below, we show how the method call is expanded and how the program associated to variable *credit* resolves the dynamic binding. Due to lack of space, we omit the precondition of $a.credit(10)$, and calculate only $credit(a, 10)$.

$$
\begin{array}{l}
credit(a, 10) \\
\equiv\{ \text{ method expansion } \} \\
\left(\begin{array}{l} \textbf{valres self} : \textbf{\textit{Object}};\ \textbf{val}\ x : \mathbb{Z} \bullet \\ \quad \textbf{self}.bonus \ldots \lhd \textbf{self is}\ BAccount \rhd (\ldots \lhd \textbf{self is}\ Account \rhd \bot) \end{array} \right)(a, 10)
\end{array}
$$

$$
\begin{array}{l}
\equiv\{ \text{ semantics of } \textbf{valres} \} \\
\textbf{var self} : \textbf{\textit{Object}}; \\
\quad \textbf{self} := a; \\
\quad \left(\begin{array}{l} \textbf{val}\ x : \mathbb{Z} \bullet \\ \quad \textbf{self}.bonus \ldots \lhd \textbf{self is}\ BAccount \rhd (\ldots \lhd \textbf{self is}\ Account \rhd \bot) \end{array} \right)(10); \\
\quad a := \textbf{self}; \\
\textbf{end self}
\end{array}
$$

$$
\begin{array}{l}
\equiv\{ \text{ semantics of } \textbf{val} \} \\
\textbf{var self} : \textbf{\textit{Object}}; \\
\quad \textbf{self} := a; \\
\qquad \textbf{var}\ x : \mathbb{Z}; \\
\qquad\quad x := 10; \\
\qquad\quad \textbf{self}.bonus \ldots \lhd \textbf{self is}\ BAccount \rhd (\ldots \lhd \textbf{self is}\ Account \rhd \bot); \\
\qquad \textbf{end}\ x; \\
\quad a := \textbf{self}; \\
\textbf{end self}
\end{array}
$$

$$
\begin{array}{l}
\equiv\{ \text{ the conditional reduces to its left branch } \} \\
\textbf{var self} : \textbf{\textit{Object}}; \\
\quad \textbf{self} := a; \\
\qquad \textbf{var}\ x : \mathbb{Z}; \\
\qquad\quad x := 10; \\
\qquad\quad \textbf{self}.bonus := \textbf{self}.bonus + 1; \\
\qquad\quad \textbf{self}.balance := \textbf{self}.balance + x; \\
\qquad \textbf{end}\ x; \\
\quad a := \textbf{self}; \\
\textbf{end self}
\end{array}
$$

This can be expanded to a predicate that establishes the final value of a to be its initial value with attributes updated by assignments. The expansion of this sequential composition is exactly the expected meaning of the method call.

7 Conclusions

We have demonstrated that object-orientation with subtyping, data inheritance and dynamic binding can be defined in the UTP, using a theory that combines designs and higher-order procedures. In particular, we have introduced two observational variables to capture information about class declarations, extra variables xt and xt', for each programming variable x, to capture the type of the variables, and, finally, variables m and m' to capture the meaning (parameters and body) of each method named m. In our theory, recursion and mutual recursion are handled in a very simple way.

The concept of variable in the object-orientation context requires explicit typing information to allow the specification of well-definedness rules for expressions and commands, and to provide the correct semantics of object-oriented expressions and commands such as assignments, conditional and method calls. We have a strong type system where all operations, and commands, over variables, values and expressions must be checked to be considered correct. We have seen that invalid declarations and commands associated to OO elements lead to $\bot$; in other words, the meaning of a badly-typed program is $\bot$, which has the unpredictable behavior that we would expect.

In contrast to [7], we do not use a runtime environment; we adopt a copy semantics, as in [15]. In the future, we intend to introduce the concept of object sharing; we plan to include extra information about variables, and review well-definedness, expressions and commands. With object sharing, the view of the target of a method call as a value-result parameter, whose value is updated to reflect changes carried out by the method, becomes unnecessary since changes are reflected directly in the objects, not in a copy. Other features that we will explore in the future are visibility mechanisms and exception handling.

The work reported in [19] presents a method for defining object specifications and refinement in a predicative style [20]. The idea is to decouple the concepts associated with general OO features, like, for instance, inheritance and class specification. This results in very general specification constructs, of which those usually found in object-oriented languages are a special case. Here, we also pursue modularity and decoupling, but we only consider object-oriented constructs.

This work was our first step towards the definition of a semantics for *OhCircus*, our object-oriented combination of Z and CSP. Our next concern is with the proposal and proof of refinement laws. Afterwards, we plan to combine our theory with that of CSP processes.

Acknowledgements: The work of Thiago Santos and Augusto Sampaio are funded by the Brazilian Research Council (CNPq grants 141301/2004-0 and 521039/95-9). The work of Ana Cavalcanti is partially funded by the Royal Society and QinetiQ. A preliminary approach to the semantics of methods was previously studied in conjunction with Jim Woodcock; we benefitted from several discussions with him. We also thank to the reviewers for their very detailed and relevant comments and the symposium participants for their challenger questions.

References

1. Plotkin, G.: A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University (1981)
2. Drossopoulou, S., Eisenbach, S. In: Towards an Operational Semantics and Proof of Type Soundness for Java. Springer-Verlag (1998)
3. Schmdit, D.A.: Denotational Semantics. A Methodology for Language Development. Allyn and Bacon,Inc (1986)
4. Hoare, C.A.R., Hayes, I.J., Jifeng, H., Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sorensen, I.H., Spivey, J.M., Sufrin, B.A.: Laws of programming. Commun. ACM **30** (1987) 672–686
5. Borba, P.H.M., Sampaio, A.C.A., Cavalcanti, A.L.C., Cornélio, M.L.: Algebraic Reasoning for Object-Oriented Programming. Science of Computer Programming **52** (2004) 53–100
6. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall (1998)
7. Jifeng, H., Li, X., Liu, Z.: A Refinement Calculus for Object Systems. Technical report 322, UNU-IIST, P.O.Box 3058, Macau (2005)
8. Qin, S.C., Dong, J.S., Chin, W.N.: A Semantic Foundation of TCOZ in Unifying Theory of Programming. In: FM'03. Lecture Notes in Computer Science, Pisa, Italy, Springer-Verlag (2003) 321–340
9. Mahony, B., Dong, J.: Blending Object-Z and Timed CSP: An introduction to TCOZ. In: Proceedings of the 20th International Conference on Software Engineering (ICSE'98), Kyoto, Japan, IEEE Computer Society Press (1998) 95–104
10. Mahony, B.P., Dong, J.S.: Timed Communicating Object Z. IEEE Transactions on Software Engineering **26** (2000) 150–177
11. Cavalcanti, A.L.C., Sampaio, A.C.A., Woodcock, J.C.P.: Unifying Classes and Processes. Software and System Modelling **4** (2005) 277–296
12. Woodcock, J.C.P., Cavalcanti, A.L.C.: The Semantics of *Circus*. In Bert, D., Bowen, J.P., Henson, M.C., Robinson, K., eds.: ZB 2002: Formal Specification and Development in Z and B. Volume 2272 of Lecture Notes in Computer Science., Springer-Verlag (2002) 184–203
13. Woodcock, J.C.P., Davies, J.: Using Z-Specification, Refinement, and Proof. Prentice-Hall (1996)
14. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall Series in Computer Science. Prentice-Hall (1998)
15. Cavalcanti, A.L.C., Naumann, D.A.: A Weakest Precondition Semantics for Refinement of Object-oriented Programs. IEEE Transactions on Software Engineering **26** (2000) 713–728
16. Back, R.J.R.: Procedural Abstraction in the Refinement Calculus. Technical report, Department of Computer Science, Åbo, Finland (1987) Ser. A No. 55.
17. Naumann, D.A.: Predicate transformers and higher-order programs. Theor. Comput. Sci. **150** (1995) 111–159
18. Borba, P.H.M., Sampaio, A.C.A.: Basic Laws of ROOL: an object-oriented language. In: 3rd Workshop on Formal Methods, Brazil (2000) 33–44
19. Kassios, I.T.: Decoupling in Object Orientation. In Fitzgerald, J., Tarlecki, A., Hayes, I., eds.: FME 2005: Formal Methods. Volume 3582 of Lecture Notes in Computer Science., Springer-Verlag (2005) 43–58
20. Hehner, E.: A Practical Theory of Programming, the second edition. Springer-Verlag, New York (2004)