Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS

CML Definition 3 – Object Orientation

Deliverable Number: D23.4c

Version: 0.3

Date: 30 September 2013

Public Document

Contributors:

Ana Cavalcanti, York
Augusto Sampaio, UFPE
Thiago Santos, UFPE
Jim Woodcock, York

Editors:

Jim Woodcock, York

Reviewers:

Joey Coleman, Aarhus
Uwe Schulze, Bremen
Zoe Andrews, Newcastle

Document History

Ver	Date	Author	Description
0.1	01-08-2013	Jim Woodcock	Initial document version
0.2	03-09-2013	Jim Woodcock	Updated following internal review
0.3	30-09-2013	Jim Woodcock	Updated following internal reviews

Contents

Object Orientation in CML

This deliverable is a revised version of that contained in D23.3. The link between the semantics for object orientation and the syntax used in CML has been clarified; we have also taken the opportunity to make some other minor amendments and corrections.

In this document, we describe a UTP theory for object orientation that closely follows the work of Thiago Santos, Augusto Sampaio, and Ana Cavalcanti (see Santos's PhD thesis [8]). Our theory forms the basic treatment of the object-oriented features of CML, which were originally inherited from those in VDM++. The state part of CML is object oriented: it allows the definition of classes with state components and operations, inheritance, and dynamic binding. The behavioural part of the language, which is based on CSP, is not object oriented. Like other features of CML, object orientation is treated orthogonally. In the final deliverable (M30), Galois connections will establish the links between the object-oriented theory and the rest of the language semantics; this topic is initially addressed elsewhere in this deliverable.

Section 1 introduces the approach we take to defining the semantics of object orientation in UTP. It also describes the connection between the syntax used in CML and the notation used in the document; essentially, CML's syntax needs to be flattened by a preprocessor to produce a series of definitions that are given meaning in UTP. Section 2 introduces the alphabet for the theory of object orientation in UTP, describing the observational variables used to capture subtyping, inheritance, and dynamic binding. Section 3 gives the healthiness conditions for the theory and proves some laws. Section 4 defines class, state component, and operation declarations. Section 5 considers how to capture type information explicitly for variables. Section 6 describes well-definedness rules for expressions and the meaning of object creation, type test, type cast, and state component access. Section 7 reviews the semantics of commands emphasising operation calls. Finally, Section 8 concludes the report.

1 Introduction

In this report we show how subtyping, inheritance, mutually recursive operations, and dynamic binding can be described in UTP by combining and extending the theories of designs and higher-order procedures. Our approach is modular, with each language feature being described in isolation. A copy semantics for an object-oriented language treats objects as denoting values; a reference semantics treats objects as denoting references to values. In our current semantics, we separate the concerns of object-oriented constructs and pointers and consider only copy semantics for objects; reference semantics will be addressed in the final CML semantics Deliverable D23.5 in M30.

In CML, classes are specified using the following syntax, which is taken from Deliverable D31.2c [7]:

```
class declaration =
  "class", identifier, [ "extends", identifier ], "=",
  "begin", { class paragraph }, "end" ;

class paragraph =
    type declarations
  | value declarations
  | function declarations
  | operation declarations
  | state declarations
  | "initial", operation definition ;
```

A class is declared as introducing a body containing declarations of different kinds, and possibly extending previous class definitions. Here we are describing only syntax; the meaning of a class declaration is the topic of the entire document.

Example 1.1 (Syntax Example (CML)) Consider a model in CML of a simple banking system; we define a class `Account`, and its state components and operations as follows.

```
class Account =
  begin
    state
      accno: int
      balance: int

    operations
      Credit : int ==> ()
      Credit(x) ==
          self.balance := self.balance + x
      ...
  end
```

This simple class introduces a state with two components: the account number (`accno`) and the account balance (`balance`). Only the first in a series of operations is described: the operation to `Credit` the account with a particular value; the other operations are elided. □

The point of the example is to show the characteristic encapsulation of all the relevant `Account` declarations into a single class declaration.

The main task of the semantics for an object-orientated notation is to show how to extract the relationship between objects from the declaration of classes. We assume that there is a preprocessor for the concrete syntax for CML that flattens this structure and extracts each individual declaration. In the UTP theory we present, we have separate constructs to declare classes and their immediate superclasses, state components, and operations. This follows the approach presented in [10].

Example 1.2 (Syntax Example (UTP)) Reconsider the simple banking system presented in Example 1.1. In our UTP semantics, we define a class *Account*, and its state

components and operations as follows.

$$\begin{array}{l}\textbf{\textit{class}}\ \mathit{Account};\\ \textbf{\textit{att}}\ \mathit{Account}\ \mathit{accno} : \mathbb{Z}, \mathit{balance} : \mathbb{Z};\\ \textbf{\textit{meth}}\ \mathit{Account}\ \mathit{Credit} = (\textbf{\textit{val}}\ x : \mathbb{Z} \bullet \textbf{\textit{self}}.\mathit{balance} := \textbf{\textit{self}}.\mathit{balance} + 1);\\ \ldots\end{array}$$

The declarations of the state components and operations are independent and are combined in sequence; in particular, notice how the declarations indicate the classes of the state components and operations that are introduced. This approach simplifies the semantics, and makes the treatment of (mutual) recursion straightforward. □

Types play a central role in the semantics of an object-oriented language due to subtyping and dynamic binding [5]. In our theory, we have a collection of observational variables that are used to model declarations. They record important typing information and are used to define the semantics of commands. We do not assume that expressions are total, due to the possibility of attempting to access state components and operations of a ***null*** object. As a consequence, we have to characterise well-defined expressions, and extend the semantics of assignments and conditionals accordingly.

Operation names are part of the alphabet of our theory: their values are parametrised programs in the sense of Back [1]. Their treatment follows the approach originally proposed in [11], and adopted in [5] to handle operations. It is also the approach followed in UTP for higher-order procedures.

Dynamic binding is reflected in the value of an operation variable: it is conditional on the type of the target object (***self***) and determines the right program that defines the behaviour of the operation in each case. In this way, we capture dynamic binding in isolation. This follows the style adopted in the algebraic semantics for object-orientation presented in [4].

1.1 Assumptions

Object-oriented features such as state component overriding, variable shading, and the use of ***super*** or related notations (to refer to elements of a superclass) are not considered here. They are only syntactic abbreviations that can be easily eliminated by preprocessing.

We consider that the names of classes, state components, operations (except for operation overriding), local variables, and parameters belong to different name-spaces. This allows us to write simpler predicates, while not imposing any relevant practical limitation, either in denotational semantics or in algebraic laws.

2 Observational Variables

In addition to the programming variables, their dashed counterparts, and the design theory observational variables ok and ok', our theory includes several extra observational variables. We introduce a variable cls to record class names; a variable sc to record the subclass relation; and a variable $atts$ to record the names and types of the state components of every class.

Definition 2.1 (Classes) The set of classes is recorded in the observational variable $cls : \mathbb{P}\, Name$.

This observational variable allows us to introduce new types in addition to the primitive ones. An example is given below:

$$cls_0 = \{\textbf{\textit{Object}}\}$$

Object is a valid class name.

Definition 2.2 (Subclasses) The subclass relation is recorded in $sc : Name \pfun Name$.

This is a mapping that maps a class name to the name of its immediate superclass. An example is

$$sc_0 = \emptyset$$

Object does not have a parent, and so cannot be present in the domain of sc. This is the subject of the healthiness condition ***OO2*** presented in the next section.

Using sc, we define the subtyping relation $A \preceq B$ that holds if A is related to B in the reflexive, transitive closure sc^*. The inclusion of primitive types (such as $\mathbb{B}$ (Booleans) and $\mathbb{Z}$ (integers)) allows us to simplify definitions.

$$\_ \preceq \_ \mathrel{\widehat{=}} sc^*$$

The subtyping relation is important in an object-oriented context to establish the well-definedness of assignments and state component accesses, as we explain in Sections 6.1 and 7.1. The strict subtyping relation is denoted by $\prec$, and is defined by

$$A \prec B \mathrel{\widehat{=}} (A \preceq B) \land (A \neq B)$$

Definition 2.3 (State Components) The state component information is recorded in $atts : Name \pfun (Name \pfun Type)$.

This is a mapping from a class name to a description of its state components that maps each state component name to its type. Here $Type$ stands for any primitive type or any name in cls, that is,

$$Type \mathrel{\widehat{=}} \{\mathbb{B}, \mathbb{Z}\} \cup cls$$

Once again, because of ***Object***'s special characteristics, we define a special value for this mapping to be used in examples. In this case, we have a mapping which says that the set of state components for ***Object*** is empty.

$$atts_0 = \{\textbf{\textit{Object}} \mapsto \emptyset\}$$

Notice, however, that using the normalisation strategy presented in [4], this set of state components associated with ***Object*** can be extended.

Definition 2.4 (Operations) Operation names are part of the alphabet of the theory. Their values are texts of parametrised programs `(pds @ p)`, where `pds` is a list of parameter declarations and `p` is a program: the body of the parametrised program that uses the parameters.

We write `(pds @ p)` rather than $(pds \bullet p)$ to indicate that the values are texts `(pds @ p)` of parametrised programs, rather than their meanings $(pds \bullet p)$. Value (***val***), result (***res***), and value-result (***vres***) parameters are allowed. The notation `pds` stands for any parameter declaration list, possibly including the three parameter passing mechanisms. For example,

$$\textbf{\textit{val}}\ x : X\ ;\ \textbf{\textit{res}}\ y : Y\ ;\ \textbf{\textit{vres}}\ z : Z$$

is a valid instance of `pds`, where x, y, and z are variable names and X, Y, and Z are their types. The function *types* applied to a list of parameter declarations yields the parameter types as a set. For instance, *types* applied to the example above yields $\{X, Y, Z\}$.

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

```
vres self: Object @
    mc ◁ self is C ▷ (mb ◁ self is B ▷ (ma ◁ self is A ▷ abort))
```

Based on the type of the current object ***self***, the nested conditional allows selection of the most specialised 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 (the bottom predicate of our theory $\bot_{OO}$, defined in the next section). The type test ***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, in the type tests for ***self***, the more specialised classes are considered first. In UTP, programs (and specifications) are predicates; there is no notation to distinguish the text of a program from its semantics. Here, just like in [9, Sect. 10] we introduce the distinction. The values of operation observational variables have to be texts to allow the use of a syntactic function to capture dynamic binding (see Section 4.3).

As usual, for each programming variable x, the alphabet contains x and x'; but we also include two more observational variables x_t and x'_t 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 x_t, when this is a class.

3 Healthiness Conditions

The following healthiness conditions characterise our theory of object-oriented relations. Healthiness condition ***OO1*** says that ***Object*** is a valid type.

$$\textbf{\textit{OO1}} \qquad P \ = \ P \wedge \textbf{\textit{Object}} \in cls$$

Our theory relies on a superclass of all classes, represented by the ***Object*** type. An example of a *cls* instance that satisfies this predicate is cls_0 (defined on page 8). Healthiness condition ***OO2*** says that every class has a parent, except ***Object***.

$$\textbf{\textit{OO2}} \qquad P \ = \ P \wedge (\mathrm{dom}\, sc = cls \setminus \{\textbf{\textit{Object}}\})$$

The top-most superclass for all classes is ***Object***, therefore cyclic references are not allowed. Healthiness condition ***OO3*** says that a parent class in sc is necessarily present in cls.

$$\textbf{OO3} \quad P = P \land \forall C : \operatorname{dom} sc \bullet (C, \textbf{Object}) \in sc^+$$

Healthiness condition ***OO4*** says that, for all classes present in cls, there is a corresponding mapping in $atts$ that records the state component names and types.

$$\textbf{OO4} \quad P = P \land (\operatorname{dom} atts = cls)$$

Healthiness condition ***OO5*** says that the names of state components are different for all classes. This is something that can be achieved by a preprocessor that labels identifiers in some appropriate way so as to distinguish apparently iedntical names.

$$\textbf{OO5} \quad P = P \land \forall C_1, C_2 : \operatorname{dom} atts \bullet (C_1 \neq C_2) \Rightarrow (\operatorname{dom}(atts(C_1)) \cap \operatorname{dom}(atts(C_2)) = \emptyset)$$

Healthiness condition ***OO6*** says that all state components must have valid types, primitives or defined by cls.

$$\textbf{OO6} \quad P = P \land (\operatorname{ran}(\bigcup \operatorname{ran} atts) \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls)$$

We name the composition of these conditions

$$\textbf{OOI} \mathrel{\widehat{=}} \textbf{OO1} \circ \cdots \circ \textbf{OO6}$$

These healthiness conditions constrain the initial values of the variables. Predicates in our theory must preserve these properties for the final values of these observational variables; thus we have a similar set of healthiness conditions for the output variables.

$$\begin{array}{ll}
\textbf{OO7} & P = P \land \textbf{Object} \in cls' \\
\textbf{OO8} & P = P \land (\operatorname{dom} sc' = cls' \setminus \{\textbf{Object}\}) \\
\textbf{OO9} & P = P \land \forall C : \operatorname{dom} sc' \bullet (C, \textbf{Object}) \in sc'^+ \\
\textbf{OO10} & P = P \land (\operatorname{dom} atts' = cls') \\
\textbf{OO11} & P = P \land \forall C_1, C_2 : \operatorname{dom} atts' \bullet (C_1 \neq C_2) \Rightarrow (\operatorname{dom}(atts'(C_1)) \cap \operatorname{dom}(atts'(C_2)) = \emptyset) \\
\textbf{OO12} & P = P \land (\operatorname{ran}(\bigcup \operatorname{ran} atts') \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls')
\end{array}$$

The healthiness conditions ***OO7***–***OO12*** can be replaced by the following condition expressed in terms of the identity of our theory.

$$\textbf{OO13} \quad P = P \,;\, \mathit{II}_{OO}$$

The set of predicates that satisfy all these healthiness conditions is our theory of object-orientation. It is a complete lattice, and its bottom element is

$$\bot_{OO} \mathrel{\widehat{=}} \textbf{OO}(\bot)$$

where ***OO*** is the functional composition of ***OO1*** $\circ \cdots \circ$ ***OO12***. The identity of our theory, denoted by II_{OO}, is the result of the application of ***OOI*** to the relational identity II.

$$\mathit{II}_{OO} \mathrel{\widehat{=}} \textbf{OOI}(\mathit{II})$$

Our healthiness conditions are idempotent, and the UTP constructs are closed under these conditions. Below, we present some valid laws that confirm this result; similar laws hold for the remaining rules.

provided P, Q, and $F(X)$ are $\mathbf{OO1}$

$$\mathbf{OO1} \circ \mathbf{OO1} = \mathbf{OO1} \quad [\mathbf{OO1}\text{-idempotent}]$$

$$P \wedge Q \;=\; \mathbf{OO1}(P \wedge Q) \quad [\mathbf{OO1}\text{-}\wedge\text{-closure}]$$

$$P \vee Q \;=\; \mathbf{OO1}(P \vee Q) \quad [\mathbf{OO1}\text{-}\vee\text{-closure}]$$

$$P \lhd b \rhd Q \;=\; \mathbf{OO1}(P \lhd b \rhd Q) \quad [\mathbf{OO1}\text{-}\_ \lhd \_ \rhd \_\text{-closure}]$$

$$P \,;\, Q \;=\; \mathbf{OO1}(P \,;\, Q) \quad [\mathbf{OO1}\text{-};\text{-closure}]$$

$$\mu X \bullet F(X) \;=\; \mathbf{OO1}(\mu X \bullet F(X)) \quad [\mathbf{OO1}\text{-}\mu\text{-closure}]$$

Similar results are proved for the other healthiness conditions in much the same way. The order of application of $\mathbf{OO1}$ and $\mathbf{OO2}$ is irrelevant. This result, the forthcoming laws related to commutativity, and similar results for $\mathbf{OO7}$–$\mathbf{OO12}$ prove that there is a subset of relations involving *cls*, *sc*, and *atts* where all healthiness conditions $\mathbf{OO1}$ and $\mathbf{OO12}$ are satisfied. As already said, this subset is our theory of object-orientation, and we have already shown that the application of conjunction, disjunction, sequence and recursion are closed in this subset.

$$\mathbf{OO1} \circ \mathbf{OO2} \;=\; \mathbf{OO2} \circ \mathbf{OO1} \quad [\mathbf{OO1}\text{-}\mathbf{OO2}\text{-commutativity}]$$

$$\mathbf{OO1} \circ \mathbf{OO3} \;=\; \mathbf{OO3} \circ \mathbf{OO1} \quad [\mathbf{OO1}\text{-}\mathbf{OO3}\text{-commutativity}]$$

$$\mathbf{OO1} \circ \mathbf{OO4} \;=\; \mathbf{OO4} \circ \mathbf{OO1} \quad [\mathbf{OO1}\text{-}\mathbf{OO4}\text{-commutativity}]$$

$$\mathbf{OO1} \circ \mathbf{OO5} \;=\; \mathbf{OO5} \circ \mathbf{OO1} \quad [\mathbf{OO1}\text{-}\mathbf{OO5}\text{-commutativity}]$$

$$\mathbf{OO1} \circ \mathbf{OO6} \;=\; \mathbf{OO6} \circ \mathbf{OO1} \quad [\mathbf{OO1}\text{-}\mathbf{OO6}\text{-commutativity}]$$

$$\mathbf{OO2} \circ \mathbf{OO3} \;=\; \mathbf{OO3} \circ \mathbf{OO2} \quad [\mathbf{OO2}\text{-}\mathbf{OO3}\text{-commutativity}]$$

$$\mathbf{OO2} \circ \mathbf{OO4} \;=\; \mathbf{OO4} \circ \mathbf{OO2} \quad [\mathbf{OO2}\text{-}\mathbf{OO4}\text{-commutativity}]$$

The composition of $\mathbf{OO7}$–$\mathbf{OO12}$ can be replaced by the application of $\mathbf{OO13}$.

$$\mathbf{OO13} \;=\; \mathbf{OO7} \circ \cdots \circ \mathbf{OO12} \quad [\mathbf{OO7}\text{–}\mathbf{12}\text{-}\mathbf{OO13}\text{-equivalence}]$$

The laws have been proved correct in [8]; several others remain to be proved.

4 Declarations

In this section we provide the meaning for class, state component, and operation declarations with some examples. The general form of the declarations is shown in Table 1. To each design that we define, we apply the composition of the healthiness conditions $\mathbf{OO}$; that is, $\mathbf{OO1} \circ \cdots \circ \mathbf{OO12}$, to guarantee that the initial values of the variables are valid and the restrictions over cls', sc', and $atts'$ hold.

4.1 Classes

Our aim is to add each feature of object-orientation in isolation. In this direction, a class declaration introduces a new type, with an empty set of state components and operations.

Construct	Description
class A ***extends*** B	introduces a new class A, subclass of B.
att $A\ x : T$	introduces a state component x of type T in the class named A.
meth $A\ m = (pds \bullet p)$	introduces an operation m with formal parameters pds and body p in the class named A.

Table 1: Object-oriented declarations.

Definition 4.1 (Class introduction) The declaration of a class is defined as shown below.

$$\textbf{\textit{class}}\ A\ \textbf{\textit{extends}}\ B\ \widehat{=}\ \textbf{\textit{OO}}\left(\begin{array}{l} A \notin \mathit{Type} \land B \in \mathit{cls} \\ \vdash \\ (\mathit{cls}' = \mathit{cls} \cup \{A\}) \\ \land\ (\mathit{sc}' = \mathit{sc} \cup \{A \mapsto B\}) \\ \land\ (\mathit{atts}' = \mathit{atts} \cup \{A \mapsto \emptyset\}) \\ \land\ (w' = w) \end{array}\right)$$
$$\textbf{\textit{where}}\ (w = in\alpha(\textbf{\textit{class}}\ A\ \textbf{\textit{extends}} B) \setminus \{\mathit{cls}, \mathit{sc}, \mathit{atts}\})$$

The design introduces a record of class A in cls and maps it in the relation sc to B as its immediate superclass. Only new names are allowed ($A \notin \mathit{Type}$), and class B needs to have been previously declared ($B \in \mathit{cls}$). An entry for A in $atts$ is mapped to the empty set. No other observational variable w is modified. As explained before, in UTP, $in\alpha$(***class*** A ***extends*** B) is the input alphabet of the program ***class*** A ***extends*** B.

The postcondition of the design establishes new final values for the observational variables of our theory; these values satisfy the properties required by the healthiness conditions ***OO7***–***OO12***. More specifically, we do not remove ***Object*** from cls' (***OO7*** is satisfied); the domain of sc is extended with a new class (A) in cls associated with B, and the precondition guarantees that B is already in cls' (***OO8*** and ***OO9*** are satisfied); the domain of $atts$ is extended to include the new class A introduced in cls (***OO10*** is satisfied); and the state component name is new and has a valid type (***OO11*** and ***OO12*** are satisfied). For a simple declaration ***class*** A, we have the obvious meaning.

$$\textbf{\textit{class}}\ A\ =\ \textbf{\textit{class}}\ A\ \textbf{\textit{extends Object}}$$

Example 4.1 (Class declaration) For our simple banking application, we declare the 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 sequence below.

$$
\begin{array}{l}
\quad \textbf{class}\ Account \\
\quad \textbf{class}\ BAccount\ \textbf{extends}\ Account \\
\quad \textbf{class}\ Contact \\
\quad \textbf{class}\ EContact\ \textbf{extends}\ Contact \\
= \\
\quad \textbf{OO} \left(\begin{array}{l} Account \notin \{\mathbb{B}, \mathbb{Z}\} \cup cls \\ \land\ \textbf{Object} \in cls \\ \vdash\ (cls' = cls \cup \{Account\}) \\ \land\ (sc' = sc \cup \{Account \mapsto \textbf{Object}\}) \\ \land\ (atts' = atts \cup \{Account \mapsto \emptyset\}) \end{array} \right) ; \\
\quad \textbf{OO} \left(\begin{array}{l} BAccount \notin \{\mathbb{B}, \mathbb{Z}\} \cup cls \\ \land\ Account \in cls \\ \vdash\ (cls' = cls \cup \{BAccount\}) \\ \land\ (sc' = sc \cup \{BAccount \mapsto Account\}) \\ \land\ (atts' = atts \cup \{BAccount \mapsto \emptyset\}) \end{array} \right) ; \\
\quad \textbf{OO} \left(\begin{array}{l} Contact \notin \{\mathbb{B}, \mathbb{Z}\} \cup cls \\ \land\ \textbf{Object} \in cls \\ \vdash\ (cls' = cls \cup \{Contact\}) \\ \land\ (sc' = sc \cup \{Contact \mapsto \textbf{Object}\}) \\ \land\ (atts' = atts \cup \{Contact \mapsto \emptyset\}) \end{array} \right) ; \\
\quad \textbf{OO} \left(\begin{array}{l} EContact \notin \{\mathbb{B}, \mathbb{Z}\} \cup cls \\ \land\ Contact \in cls \\ \vdash\ (cls' = cls \cup \{EContact\}) \\ \land\ (sc' = sc \cup \{EContact \mapsto Contact\}) \\ \land\ (atts' = atts \cup \{EContact \mapsto \emptyset\}) \end{array} \right) ;
\end{array}
$$

The meaning of sequence in our theory is the same as that in UTP. If we take cls, sc, and $atts$ to be cls_0, sc_0, and $atts_0$, respectively, the sequence above specifies the following values for cls', sc' and $atts'$.

$$
\begin{array}{l}
cls' = \left\{ \begin{array}{l} \textbf{Object}, \\ Account, \\ BAccount, \\ Contact, \\ EContact \end{array} \right\} \\
sc' = \left\{ \begin{array}{l} Account \mapsto \textbf{Object}, \\ BAccount \mapsto Account, \\ Contact \mapsto \textbf{Object}, \\ EContact \mapsto Contact \end{array} \right\} \\
atts' = \left\{ \begin{array}{l} \textbf{Object} \mapsto \emptyset, \\ Account \mapsto \emptyset, \\ BAccount \mapsto \emptyset, \\ Contact \mapsto \emptyset, \\ EContact \mapsto \emptyset \end{array} \right\}
\end{array}
$$

□

4.2 State Components

We introduce state components in *atts* for those classes already in *cls*.

Definition 4.2 (State Component introduction) To introduce a state component x of type T in class A we can use the construct defined below.

$$
\begin{array}{l}
\textbf{\textit{att}}\ A\ x : T\ \widehat{=} \\
\quad \textbf{\textit{OO}} \left(\begin{array}{l}
A \in cls \\
\land\ x \notin \operatorname{dom} \mathcal{C}(atts, cls) \\
\land\ T \in Type \\
\vdash \\
(atts' = atts \oplus \{A \mapsto (atts(A) \cup \{x \mapsto T\})\}) \\
\land\ (w' = w)
\end{array} \right) \\
\textbf{\textit{where}}\ w = in\alpha(\textbf{\textit{att}}\ A\ x : T) \setminus \{atts\} \\
\textbf{\textit{and}}\ (\mathcal{C}(amap, cset) = \bigcup\{\, N : cset \bullet amap(N) \,\}) \\
\textbf{\textit{and}}\ amap \text{ is a state component mapping, and } cset \text{ is class set}
\end{array}
$$

If we try to declare a state component 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 *cls*, then the declaration aborts. The set $\mathcal{C}$ (defined above as $(\mathcal{C}(amap, cset) = \bigcup\{\, N : cset \bullet amap(N) \,\})$ is a useful abbreviation for a mapping of all state components of any class to their corresponding types, calculated from a state component mapping as defined for *atts*, and a class set as *cls*. Our healthiness conditions ***OO7***–***OO12*** are guaranteed by the design not changing the variables *cls* and *sc* nor the domain of *atts*.

We can declare several state components simultaneously, with the obvious meaning.

$$
\begin{array}{l}
\textbf{\textit{att}}\ A\ x1 : T1\ ;\ x2 : T2, \ldots\ =\ \textbf{\textit{att}}\ A\ x1 : T1\ ;\ \textbf{\textit{att}}\ A\ x2 : T2, \ldots \\
\textbf{\textit{att}}\ A\ x1 : T1\ ;\ B\ x2 : T2, \ldots\ =\ \textbf{\textit{att}}\ A\ x1 : T1\ ;\ \textbf{\textit{att}}\ B\ x2 : T2, \ldots
\end{array}
$$

Our notation allows interleaving concerning with the order of class, state component, and operation declaration. For example, the sequence below is allowed.

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

In this case, the state component y of class A is declared after the declaration of 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 a state component x whose type is a subclass B of A, then the following order of declaration is required.

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

Transforming the class-based declarations of an object-oriented language into an appropriate sequence of class and state component declarations is a simple task. For operations, similar considerations apply; mutual recursion, however, is further discussed in Section 7.4.

Example 4.2 (State Component declaration) This example adds some state components to the classes of Example 4.1.

$$
\begin{array}{l}
\textbf{\textit{att}}\ \mathit{Account}\ \mathit{number} : \mathbb{Z}, \mathit{balance} : \mathbb{Z}, \mathit{contact} : \mathit{Contact} \\
\textbf{\textit{att}}\ \mathit{BAccount}\ \mathit{bonus} : \mathbb{Z} \\
\textbf{\textit{att}}\ \mathit{Contact}\ \mathit{phone} : \mathbb{Z} \\
\textbf{\textit{att}}\ \mathit{EContact}\ \mathit{icq} : \mathbb{Z} \\
\textbf{\textit{oo}} \left(\begin{array}{l} \mathit{Account} \in \mathit{cls} \\ \land\ \mathit{number} \notin \mathrm{dom}(\mathbb{C}(\mathit{atts}, \mathit{cls})) \\ \land\ \mathbb{Z} \in \{\mathbb{B}, \mathbb{Z}\} \cup \mathit{cls} \\ \vdash (\mathit{atts}' = \mathit{atts} \oplus \{\mathit{Account} \mapsto (\mathit{atts}(\mathit{Account}) \cup \{\mathit{number} \mapsto \mathbb{Z}\})\}) \end{array} \right) ; \\
\textbf{\textit{oo}} \left(\begin{array}{l} \mathit{Account} \in \mathit{cls} \\ \land\ \mathit{balance} \notin \mathrm{dom}(\mathbb{C}(\mathit{atts}, \mathit{cls})) \\ \land\ \mathbb{Z} \in \{\mathbb{B}, \mathbb{Z}\} \cup \mathit{cls} \\ \vdash (\mathit{atts}' = \mathit{atts} \oplus \{\mathit{Account} \mapsto (\mathit{atts}(\mathit{Account}) \cup \{\mathit{balance} \mapsto \mathbb{Z}\})\}) \end{array} \right) ; \\
\textbf{\textit{oo}} \left(\begin{array}{l} \mathit{Account} \in \mathit{cls} \\ \land\ \mathit{contact} \notin \mathrm{dom}(\mathbb{C}(\mathit{atts}, \mathit{cls})) \\ \land\ \mathit{Contact} \in \{\mathbb{B}, \mathbb{Z}\} \cup \mathit{cls} \\ \vdash (\mathit{atts}' = \mathit{atts} \oplus \{\mathit{Account} \mapsto (\mathit{atts}(\mathit{Account}) \cup \{\mathit{contact} \mapsto \mathit{Contact}\})\}) \end{array} \right) ; \\
\textbf{\textit{oo}} \left(\begin{array}{l} \mathit{BAccount} \in \mathit{cls} \\ \land\ \mathit{bonus} \notin \mathrm{dom}(\mathbb{C}(\mathit{atts}, \mathit{cls})) \\ \land\ \mathbb{Z} \in \{\mathbb{B}, \mathbb{Z}\} \cup \mathit{cls} \\ \vdash (\mathit{atts}' = \mathit{atts} \oplus \{\mathit{BAccount} \mapsto (\mathit{atts}(\mathit{BAccount}) \cup \{\mathit{bonus} \mapsto \mathbb{Z}\})\}) \end{array} \right) ; \\
\textbf{\textit{oo}} \left(\begin{array}{l} \mathit{Contact} \in \mathit{cls} \\ \land\ \mathit{phone} \notin \mathrm{dom}(\mathbb{C}(\mathit{atts}, \mathit{cls})) \\ \land\ \mathbb{Z} \in \{\mathbb{B}, \mathbb{Z}\} \cup \mathit{cls} \\ \vdash (\mathit{atts}' = \mathit{atts} \oplus \{\mathit{Contact} \mapsto (\mathit{atts}(\mathit{Contact}) \cup \{\mathit{phone} \mapsto \mathbb{Z}\})\}) \end{array} \right) ; \\
\textbf{\textit{oo}} \left(\begin{array}{l} \mathit{EContact} \in \mathit{cls} \\ \land\ \mathit{icq} \notin \mathrm{dom}(\mathbb{C}(\mathit{atts}, \mathit{cls})) \\ \land\ \mathbb{Z} \in \{\mathbb{B}, \mathbb{Z}\} \cup \mathit{cls} \\ \vdash (\mathit{atts}' = \mathit{atts} \oplus \{\mathit{EContact} \mapsto (\mathit{atts}(\mathit{EContact}) \cup \{\mathit{icq} \mapsto \mathbb{Z}\})\}) \end{array} \right)
\end{array}
$$

We use the definition of state component declaration for each element of the sequence, starting with the state component *number*, and ending with *icq*. If we suppose that the declaration above comes after the class declarations of Example 4.1, the expected final value of *atts* is as follows.

$$
\mathit{atts}' = \left\{ \begin{array}{l} \textbf{\textit{Object}} \mapsto \emptyset, \\ \mathit{Account} \mapsto \{\mathit{number} \mapsto \mathbb{Z}, \mathit{balance} \mapsto \mathbb{Z}, \mathit{contact} \mapsto \mathit{Contact}\}, \\ \mathit{BAccount} \mapsto \{\mathit{bonus} \mapsto \mathbb{Z}\}, \\ \mathit{Contact} \mapsto \{\mathit{phone} \mapsto \mathbb{Z}\}, \\ \mathit{EContact} \mapsto \{\mathit{icq} \mapsto \mathbb{Z}\} \end{array} \right\}
$$

□

For a given class N, we define $\mathcal{U}(\mathit{amap}, \mathit{smap}, N)$ to be a mapping that records all the state components of N, including those declared in its superclasses, considering a state component mapping *amap*, and a subclass relation *smap* defined with the same types of atts, and sc, respectively. We define this closure as:

$$
\mathcal{U}(\mathit{amap}, \mathit{smap}, N) \ \widehat{=}\ \bigcup(\mathit{amap}(\!|\ \mathit{smap}^*(\!|\ \{N\}\ |\!)\ |\!))
$$

In words, $\mathcal{U}(amap, smap, N)$ contains all the state component declarations of all classes related to N by the reflexive and transitive closure of the superclass relation, considering the current state components in *atts* and subclass relation *sc*. This function is useful to define object creation and also to check if an instance of an object is well-defined.

4.3 Operations

For an operation declaration to succeed, the class to which it is associated must have been introduced before, and all formal parameters, passed by value (***val***), result (***res***) or value-result (***vres***), must have primitive types or those introduced in *cls*. The result depends on whether the operation is being declared for the first time or not. If it is $(m \notin \alpha(\textbf{\textit{meth}}\ A\ m\ = pds \bullet p))$, then the definition below applies. The new name m is introduced in the alphabet using a variable declaration.

Definition 4.3 (New operation introduction) For new operations, the declaration is defined as follows.

$$
\begin{array}{l}
\textbf{\textit{meth}}\ A\ m\ (pds\ p)\ \widehat{=} \\
\quad \textbf{\textit{OO}} \left(\begin{array}{l}
\textbf{\textit{var}}\ m\ ; \\
\left(\begin{array}{l}
A \in cls \\
\land\ \forall\, t \in types(pds) \bullet t \in Type \\
\vdash \\
(m' = \textbf{vres self: } Object\ ;\ \mathtt{pds}\ @\ (\mathtt{p} \lhd \textbf{self is } \mathtt{A} \rhd \mathtt{abort}) \\
\land\ (w' = w)
\end{array} \right)
\end{array} \right) \\
\textbf{\textit{provided}}\ m \notin \alpha(\textbf{\textit{meth}}\ A\ m = (pds \bullet p)) \\
\textbf{\textit{where}}\ (w = in\alpha(\textbf{\textit{meth}}\ A\ m = (pds \bullet p)))
\end{array}
$$

The value of m' is the text of a parametrised program. Operations are higher-order, parametrised program-valued variables, much in the same way as in the theory of higher-order procedures and parameters of 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 with ***var*** x, which introduces the new alphabetic variables x and x', for ***meth*** $A\ m$, we introduce the new alphabetic variables m and m', and use a design to define the value of m'.

For the case of a redefinition of an operation m $(m \in \alpha(\textbf{\textit{meth}}\ A\ m\ =\ pds \bullet p))$ we have another definition.

Definition 4.4 (Operation redefinition) If the operation name declared is not new, the corresponding definition is the following.

$$
\begin{array}{l}
\textbf{\textit{meth}}\ A\ m = (pds\ @\ p)\ \widehat{=} \\
\quad \textbf{\textit{OO}} \left(\begin{array}{l}
A \in cls \\
\land\ \exists\, \mathtt{q} \bullet m = \mathtt{pds}_e \bullet \mathtt{q} \\
\vdash\ \exists\, \mathtt{q} \bullet \\
\qquad (m = \mathtt{pds}_e \bullet \mathtt{q}) \\
\qquad \land\ (m' = \mathtt{pds}_e \bullet join(A, \mathtt{p}, \mathtt{q})) \\
\qquad \land\ (w' = w)
\end{array} \right) \\
\textbf{\textit{provided}}\ m \in \alpha(\textbf{\textit{meth}}\ A\ m = (pds \bullet p)) \\
\textbf{\textit{where}}\ \mathtt{pds}_e = \textbf{vres self: } \mathtt{Object}\ ;\ \mathtt{pds}\ @\ \mathtt{p}
\end{array}
$$

$$\textbf{\textit{and}}\ (w = in\alpha(\textbf{\textit{meth}}\ \mathtt{A}\ \mathtt{m} = (pds \bullet p)) \setminus \{m\})$$

$$\textbf{\textit{and}}\ join(A, \mathtt{p}, \bot_{OO}) = \textbf{if self is}\ \mathtt{A}\ \textbf{then}\ \mathtt{p}\ \textbf{else}\ abort$$

$$\textbf{\textit{and}}\ join(A, \mathtt{p}, \textbf{if self is}\ \mathtt{B}\ \textbf{then}\ \mathtt{q}_l\ \textbf{else}\ \mathtt{q}_r) = \begin{cases} \mathtt{p} \lhd \textbf{self is}\ \mathtt{A} \rhd (\mathtt{q}_l \lhd \textbf{self is}\ \mathtt{B} \rhd \mathtt{q}_r), & \text{if } A \prec B \\ \mathtt{q}_l \lhd \textbf{self is}\ \mathtt{B} \rhd join(A, \mathtt{p}, \mathtt{q}_r) & \text{, otherwise} \end{cases}$$

$$\textbf{\textit{and}}\ join(A, \mathtt{p}, \mathtt{q}) = \bot_{OO}, \text{ for programs } \mathtt{q} \text{ of every other form}$$

If the operation declaration is a redefinition, the operation signature must be exactly the same as that of the existing operation, and a new conditional is built to take into account the class hierarchy. The definition of the syntactic function *join* deals with redefinition of m both in superclasses and in subclasses of the class where the original definition is placed. The use of *join* allows us to introduce the operation values as (parametrised) programs in a form where dynamic binding is already resolved, as in algebraic operations [3, 4] and in the weakest precondition approach [5]. As already said, the special variable ***self*** denotes the target of the operation call. All references to state components in operation bodies must be prefixed with ***self***; variables without this prefix are formal parameters or local variables.

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.

$$(\textbf{\textit{val}}\ v : T \bullet p) \mathrel{\widehat{=}} (\lambda\, w : T \bullet (\textbf{\textit{var}}\ v : T\ ;\ v := w\ ;\ p\ ;\ \textbf{\textit{end}}\ v))$$

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 an operation call.

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

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.

$$(\textbf{\textit{vres}}\ v : T \bullet p) \mathrel{\widehat{=}} (\lambda\, w : \mathcal{N} \bullet (\textbf{\textit{var}}\ v : T\ ;\ v := w\ ;\ p\ ;\ w := v\ ;\ \textbf{\textit{end}}\ v))$$

The parameter of the function is again a program variable.

Alpha-reduction is extended to cope with variable parameters: elements of the syntactic category $\mathcal{N}$. It is an abstraction over four arguments: a variable, the corresponding type variable, and their dashed counterparts. A similar semantics for parametrisation was presented in [6].

$$(\lambda\, x : \mathcal{N} \bullet p)(y) \mathrel{\widehat{=}} p[y, y', y_t, y_t'/x, x', x_t, x_t']$$

Example 4.3 (Operation declaration) In this example we show the semantics of operation declarations, given that *cls* is the one defined in Example 4.1, and *atts* is that defined in Example 4.2. We introduce an operation *credit* for *Account*, and we redefine it for class *BAccount* to also increase the value of a bonus variable.

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

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 **super** `.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, we do not consider this issue here. The meaning of the two operation declarations is given by the sequence below.

$$\begin{array}{l}\textbf{\textit{OO}}\left(\begin{array}{l}\textbf{\textit{var}}\ \mathit{credit}\ ; \\ \left(\begin{array}{l}\mathit{Account} \in \mathit{cls} \vdash \\ \left(\begin{array}{l}\mathit{credit}' = \left(\begin{array}{l}\textbf{vres self:}\ \texttt{Object}\ \textbf{; val}\ \texttt{x:}\ \textbf{int @} \\ \quad \textbf{self.}\texttt{balance} := \cdots \\ \quad \lhd\ \textbf{self is}\ \texttt{Account}\ \rhd \\ \quad\quad \texttt{abort}\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)\ ; \\ \textbf{\textit{OO}}\left(\begin{array}{l}\mathit{BAccount} \in \mathit{cls} \\ \exists\, \texttt{q} \bullet (\mathit{credit} = (\textbf{vres self:}\ \texttt{Object}\ \textbf{; val}\ \texttt{x:}\ \textbf{int @}\ \texttt{q})) \\ \vdash \\ \exists\, \texttt{q} \bullet \\ \quad (\mathit{credit} = (\textbf{vres self:}\ \texttt{Object}\ \textbf{; val}\ \texttt{x:}\ \textbf{int @}\ \texttt{q})) \\ \quad (\mathit{credit}' = (\textbf{vres self:}\ \texttt{Object}\ \textbf{; val}\ \texttt{x:}\ \textbf{int @} \\ \quad\quad \mathit{join}(\mathit{BAccount}, (\textbf{self.}\texttt{bonus} := \cdots), \texttt{q})\end{array}\right)\end{array}$$

If we eliminate the sequential composition, the value of the variable `q` existentially quantified in the second design is determined to be the body of *credit* as defined in the first design. With that, we can calculate the result of *join*. The final value of *credit* is of the following form.

$$\begin{array}{l}\textbf{vres self:}\ \texttt{Object}\ \textbf{; val}\ \texttt{x:}\ \textbf{int @} \\ \quad (\textbf{self.}\texttt{bonus} := \textbf{self.}\texttt{bonus} + 1\ \textbf{;}\ \ldots) \\ \quad\quad \lhd\ \textbf{self is}\ \texttt{BAccount}\ \rhd \\ \quad\quad\quad \left(\begin{array}{l}(\textbf{self.}\texttt{balance} := \textbf{self.}\texttt{balance} + \texttt{x}) \\ \quad \lhd\ \textbf{self is}\ \texttt{Account}\ \rhd \\ \quad\quad \texttt{abort}\end{array}\right)\end{array}$$

The conditional generated by *join* selects the appropriate command depending on the type of ***self***. This is the expected behaviour in the presence of dynamic binding. □

5 Variables

In [9], type information is not explicitly recorded for 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 variable, and the second is the value itself. We define the construct $\textbf{\textit{var}}\ x : T$ for typed declaration of variables, where T is the static type of the variable x.

Definition 5.1 (Variable declaration)

$$\begin{array}{l}\textbf{\textit{var}}\ x : T\ \widehat{=} \\ \quad \textbf{\textit{OO}}\,(\{T \in Type\}_\perp\ ;\ \textbf{\textit{var}}\ x_t, x\ ;\ (\textbf{\textit{true}} \vdash (x_t' = T) \land x' \in \mathcal{V}(T) \land (w' = w))) \\ \textbf{\textit{provided}}\ x \notin in\alpha(\textbf{\textit{var}}\ x : T)\ \textbf{\textit{and}}\ (w = in\alpha(\textbf{\textit{var}}\ x : T))\end{array}$$

We use the existing ***var*** construct to introduce both x and x_t in the alphabet. In the definition, we assert that T is a valid type, if it is invalid the sequential composition aborts. Otherwise, the type x_t of x is defined to be T, and an arbitrary element of T is chosen as its initial value. The other variables remain unchanged. In assignments to x, its value, which is a pair (x_t, x_v), may change, but x_t does not.

To complete this definition, we need to define the set of elements $\mathcal{V}$ of a type C. These are pairs in which the first element is a subclass A of C, possibly C itself, and the second element is either the special value ***null*** or a mapping from each of the state components of A to a value, in the case of classes. For primitives types the second value is a primitive value, such as 1, for integer, or *true* for Booleans. A formal definition is a function that takes *sc* and *atts* as parameters; a similar function is specified in [5]. As with $\textbf{\textit{var}}\ x$, our typed declaration is a non-homogeneous relation: the alphabet of $\textbf{\textit{var}}\ x : T$ does not include x or x_t.

The definition of $\textbf{\textit{end}}\ x : T$ (the construct used to finalise the scope of x) is similar to that in UTP for $\textbf{\textit{end}}\ x$. There are no concerns with type at the end of the scope of a variable, but we need to close the scope of both x and x_t.

Definition 5.2 (Variable removal)

$$\textbf{\textit{end}}\ x : T\ \widehat{=}\ \textbf{\textit{OO}}(\textbf{\textit{end}}\ x, x_t)$$

The discussion about the structure of values is extremely important to the definition of value of an object and the correctness of assignments and operation calls. We have made explicit the representation of values in order to handle the concepts of object-orientation.

6 Expressions

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

6.1 Well-definedness

Our theory includes new forms of expression e characterised by the BNF-like definition in Table 2. In this syntax, v is a primitive or object value. The expressions le, left expressions, are ordinary variable names or the special variable named ***self***, followed by a (possibly empty) sequence of dot-separated names. The expression ***new*** N is an object

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

Table 2: BNF for object-oriented expressions.

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

An expression e is represented by a pairs: (e_t, e_v), where the first element of the pair e_t is the type of e and the second element e_v is its value. The construct ***null*** 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 $x := \textbf{\textit{null}}$, we have that $e_t = x_t$; this means that the runtime type ***null*** is the declared type of x.

The well-definedness of expressions, and commands, is specified by a function named $\mathcal{D}$. If an expression has a primitive value, it is well-defined if the value belongs to the set of possible values of the type. For objects, we must check if the type belongs to cls, and if the value belongs to the set of values of type T, $\mathcal{V}(T)$. For primitive types the test is simpler.

Primitive values
$$\mathcal{D}(\mathbb{B}, v) = v \in \mathbb{B}$$
$$\mathcal{D}(\mathbb{Z}, v) = v \in \mathbb{Z}$$
Objects
$$\mathcal{D}(T, \textbf{\textit{null}}) = T \in cls$$
$$\mathcal{D}(T, v) = T \in cls \land v \in \mathcal{V}(T)$$

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

Variables
$$\mathcal{D}(x) = x_t \in Type$$
$$\mathcal{D}(\textbf{\textit{self}}) = \textbf{\textit{self}}_t \in cls$$

A state component access $le.x$ is valid only if le is well-defined, the type of le is not primitive, the value of le is different from ***null***, and x is in the domain of the value of le.

State Component Accesses
$$\mathcal{D}(le.x) = \mathcal{D}(le) \land le_t \in cls \land (le_v \neq \textbf{\textit{null}}) \land x \in \mathrm{dom}\ le_v$$

A ***new*** N declaration is valid only if the class N is recorded in cls. A type test e ***is*** N or casting $(N)\ e$ can be done only if e is a well-defined expression and N is not primitive. For a type cast, the expression has to be of a valid subtype of N.

Typing
$$\mathcal{D}(\textbf{\textit{new}}\ N) = N \in cls$$
$$\mathcal{D}(e\ \textbf{\textit{is}}\ N) = \mathcal{D}(e) \land N \in cls$$
$$\mathcal{D}((N)e) = \mathcal{D}(e) \land N \in cls \land e_t \preceq N$$

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, "mod".

Remainder
$$\mathcal{D}(x \bmod y) \;=\; \mathcal{D}(x) \land \mathcal{D}(y) \land x_t = \mathbb{Z} \land y_t = \mathbb{Z} \land y_v \neq 0$$

In Section 7.1, we use the function $\mathcal{D}$ on expressions to define well-definedness rules for commands.

6.2 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 state component values. Using *sc* and *atts* to recover state components and inheritance information, we provide a definition for ***new*** as follows.

$$\begin{array}{l}\textbf{\textit{new}}\ N \;\widehat{=}\\ \quad \left(N, \left\{ \begin{array}{l} x : \mathrm{dom}\ map;\ t : Type;\ v : \{\, T : Type;\ i : T \bullet i \,\} \mid \\ ((map(x) = \mathbb{B}) \land (t = \mathbb{B}) \land (v = false)) \\ \lor\ ((map(x) = \mathbb{Z}) \land (t = \mathbb{Z}) \land (v = 0)) \\ \lor\ (\exists\, T : cls \bullet (map(x) = T) \land (t = T) \land (v = \textbf{\textit{null}})) \bullet \\ x \mapsto (t, v) \end{array} \right\} \right) \\ \textbf{\textit{where}}\ map = \mathcal{U}(atts, sc, N)\end{array}$$

This definition says that the value of a newly created object is a mapping from state component names x to values (t, v) that associate all Boolean state components to *false*, all integer state components to 0, and all class-typed state components to ***null***. For example, the value of ***new*** *BAccount*, considering the values of *sc* and *atts* obtained after the declarations of Examples 4.1 and 4.2, is

$$\left(\begin{array}{l} BAccount, \\ \{number \mapsto (\mathbb{Z}, 0), balance \mapsto (\mathbb{Z}, 0), contact \mapsto (Contact, \textbf{\textit{null}}), bonus \mapsto (\mathbb{Z}, 0)\} \end{array} \right)$$

In this example, all state components from class *Account* (*number*, *balance*, *contact*), as well as those from *BAccount* (*bonus*), are included.

6.3 Type Test

The expression e ***is*** N is a Boolean that indicates whether the value of e belongs to the class N or to one of its subclasses. The result yielded by such an expression is

$$e\ \textbf{\textit{is}}\ N \;\widehat{=}\; (\mathbb{B}, e_t \preceq N)$$

For example,

$$\begin{array}{l}(\textbf{\textit{new}}\ BAccount)\ \textbf{\textit{is}}\ Account \\ = (BAccount, \{\ldots\})\ \textbf{\textit{is}}\ Account \\ = (\mathbb{B}, BAccount \preceq Account) \\ = (\mathbb{B}, true)\end{array}$$

This is justified by the definitions of ***new***, type test, and $\preceq$, if we assume that *cls* and *sc* are as defined in Example 4.1.

6.4 Type Cast

The result of a cast $(N)e$ 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.

$$(N)e \;\widehat{=}\; e$$

For example, provided that $BAccount \preceq Account$

$$\begin{aligned}&(Account)\ \textbf{\textit{new}}\ BAccount\\ &= (Account)(BAccount, \{\ldots\})\\ &= (BAccount, \{\ldots\})\end{aligned}$$

Well-definedness is checked in the semantics of assignments and conditionals.

6.5 State Component Access

a state component access $le.x$ recovers from the object value mapping (le_v) the state component named x.

$$le.x \;\widehat{=}\; le_v(x)$$

Again, we have a very simple definition, because we are considering only well-defined state component accesses. For example, suppose that we have an instance of $BAccount$ such as in the following program.

$$\begin{aligned}&\textbf{\textit{var}}\ x : BAccount\ ;\\ &x := \textbf{\textit{new}}\ BAccount\ ;\end{aligned}$$

The result of the expression $x.bonus$ is given by

$$\begin{aligned}&x.bonus\\ &= (x_t, x_v).bonus\\ &= (BAccount, \{\ldots, bonus \mapsto (\mathbb{Z}, 0)\}).bonus\\ &= \{\ldots, bonus \mapsto (\mathbb{Z}, 0)\}(bonus)\\ &= (\mathbb{Z}, 0)\end{aligned}$$

As expected, we select the value associated to $bonus$ in the mapping of state component values for x. If we have a composite name like $le.x.y$, we successively apply the lookup $(le_v(x)).y$ to select the expected value.

7 Commands

Our theory so far includes assignments $le := e$ of a value e to a left expression le, and operation calls $le.m(a)$ with target le and list of arguments a. Now we move to commands, using the syntax described in Table 3.

$$c ::= le := e \mid \pi \mid \textbf{\textit{var}}\ x : T \mid \textbf{\textit{end}}\ x : T \mid c_1 \lhd e \rhd c_2 \mid c_1 \,;\, c_2 \mid \mu\, X \bullet F(X) \mid le.m(e)$$

Table 3: BNF for object-oriented commands.

7.1 Well-definedness

In this section, we specify well-definedness for assignments, conditionals, and operation calls. We consider two forms of assignment: assignments to variables, and assignments to object state components. 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 the type x_t of x.

Assignment to variables
$$\mathcal{D}(x := e) \;=\; \mathcal{D}(x) \wedge \mathcal{D}(e) \wedge e_t \preceq x_t$$

For an assignment of an expression e to a state component x of le to be well-defined, the expression $le.x$ must be well-defined, e must be well-defined, and the type of e must be a subtype of $\mathcal{U}(atts, sc, le_t)(x)$, the type of the state component x in the class le_t (defined in Section 4.2).

Assignment to state components
$$\mathcal{D}(le.x := e) \;=\; \mathcal{D}(le.x) \wedge \mathcal{D}(e) \wedge e_t \preceq \mathcal{U}(atts, sc, le_t)(x)$$

For a conditional to be well-defined, the condition must be well-defined and yield a Boolean value.

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

The definition of well-definedness for operation calls is the most extensive. An operation call of the form $le.m(a)$ is valid if:

- The left-hand expression le is well-defined.
- The value of le is not ***null***.
- The operation m is defined for the type of le.
- To avoid aliasing, le is not passed as an argument and is not involved in any argument. For further details about this restriction, see [5].
- The types of the arguments in the list a must be compatible with the formal parameters list of m.

We present well-definedness definitions according to the parameter passing mechanism. Starting with value parameters, we have the definition below.

Operation call
$$\mathcal{D}(le.m(e)) \;\widehat{=}\; \mathcal{D}(le) \wedge (le_v \neq \textbf{\textit{null}}) \wedge \mathit{compatible}(le, m) \wedge e_t \preceq T$$
provided $\exists\, \mathtt{p} \bullet m = (\textbf{val}\ \mathtt{x}\texttt{:}\ \mathtt{T}\ \texttt{@}\ \mathtt{p})$
where $\mathit{compatible}(le, m) \;=\; \exists \mathtt{pds}, \mathtt{p} \bullet m = (\mathtt{pds}\ \texttt{@}\ \mathtt{p}) \wedge le_t \in \mathit{scan}(\mathtt{p})$
with $\mathit{scan}(\bot_{OO}) = \emptyset$
and $\mathit{scan}(\mathtt{p}_l \lhd \textbf{self is}\ \mathtt{A} \rhd \mathtt{p}_r) \;=\; \{\, B : \mathit{cls} \mid B \preceq A \,\} \cup \mathit{scan}(\mathtt{p}_r)$

The *scan* function yields the set of class names for which the operation m may have a definition different from $\bot_{OO}$. For result and value-result parameters, we use the function sdisjoint [5], which verifies if le is involved in any of the arguments.

$$\begin{array}{l}\mathcal{D}(le.m(y)) \mathrel{\widehat{=}} \mathcal{D}(le) \land (le_v \neq \textbf{\textit{null}}) \land \mathit{compatible}(le, m) \land \mathrm{sdisjoint}(le, y) \land T \preceq y_t \\ \textbf{\textit{provided}}\ \exists\, \mathtt{p} \bullet m = (\textbf{\texttt{res}}\ \mathtt{x}\texttt{:}\ \mathtt{T}\ \texttt{@}\ \mathtt{p}) \\ \mathcal{D}(le.m(z)) \mathrel{\widehat{=}} \mathcal{D}(le) \land (le_v \neq \textbf{\textit{null}}) \land \mathit{compatible}(le, m) \land \mathrm{sdisjoint}(le, z) \land (T = z_t) \\ \textbf{\textit{provided}}\ \exists\, \mathtt{p} \bullet m = (\textbf{\texttt{vres}}\ \mathtt{c}\texttt{:}\ \mathtt{T})\end{array}$$

The definition for an operation call with multiple arguments is a straightforward extension of these definitions.

7.2 Assignments

Now we define assignments to variables, and assignments to state components of object variables. In our theory, we observe that modifying the value of operation variables, type variables x_t, cls, sc, $atts$, or ok is not allowed, in much the same way that assignments to ok are not allowed in the theory of designs as well.

If we establish the well-definedness of an assignment, we can update the value of the variable to be that of the expression.

$$\begin{array}{l}x := e \mathrel{\widehat{=}} \textbf{\textit{OO}}(\mathcal{D}(x := e) \vdash (x' = e) \land (w' = w)) \\ \textbf{\textit{where}}\ (w = in\alpha(x := e) \setminus \{x\})\end{array}$$

For example, given a variable x of type $Account$ $((x_t = Account))$, suppose that y is the list of undashed variables in the alphabet, other than x, and that cls, sc, and $atts$ are as in Examples 4.1 and 4.2. We can calculate the meaning of the assignment $x :=$ ***new*** $BAccount$:

$$\begin{array}{l}x := \textbf{\textit{new}}\ BAccount \\ = \{\ \text{assignment}\ \} \\ \textbf{\textit{OO}} \left(\begin{array}{l} \mathcal{D}(x := (BAccount, \{number \mapsto \ldots\})) \\ \vdash \\ (x' = (BAccount, \{number \mapsto \ldots\})) \land (y' = y) \end{array} \right) \\ = \{\ \mathcal{D}\ \text{for variable assignments}\ \} \\ \textbf{\textit{OO}} \left(\begin{array}{l} \mathcal{D}(x) \land \mathcal{D}(BAccount, \{number \mapsto \ldots\}) \land BAccount \preceq Account \\ \vdash \\ (x' = (BAccount, \{number \mapsto \ldots\})) \land (y' = y) \end{array} \right) \\ = \{\ \mathcal{D}\ \text{for variables and instances, assumptions on}\ cls\ \text{and subtyping}\ \} \\ \textbf{\textit{OO}} \left(\begin{array}{l} Account \in \{\mathbb{B}, \mathbb{Z}\} \\ \land\ BAccount \in cls \\ \land\ (BAccount, \{number \mapsto \ldots\}) \in \mathcal{V}(BAccount) \\ \vdash \\ (x' = (BAccount, \{number \mapsto \ldots\})) \land (y' = y) \end{array} \right) \\ = \{\ \text{assumptions on}\ cls\text{, definition of}\ \mathcal{V}\ \} \\ \textbf{\textit{OO}}\,(\textbf{\textit{true}} \vdash (x' = (BAccount, \{number \mapsto \ldots\})) \land (y' = y))\end{array}$$

To update a state component of an object-valued expression, we check the well-definedness of the assignment, and if it is valid, then we update the mapping that records the state component value, maintaining the left expression type unchanged.

$$
\begin{array}{l}
le.x := e \;\widehat{=}\; \mathbf{OO}(\mathcal{D}(le.x := e) \vdash (le' = (le_t, le_v \oplus \{x \mapsto e\})) \wedge (w' = w)) \\
\textbf{where } (w = in\alpha(le.x := e) \setminus \alpha(le))
\end{array}
$$

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. On the other hand, for $x.y$ and $x.y.z$, for example, the variable is x. The equality $(le' = (le_t, le_v \oplus \{x \mapsto e\}))$ for the case in which le is itself a state component access $y.z$, for instance, is an abbreviation for the equality $(y' = (y_t, y_v \oplus \{z \mapsto y.z \oplus \{x \mapsto e\}))$.

For example, given a variable x of type $Account$ $(x_t = Account)$, which has been initialised with ***new*** $BAccount$ $(x = (BAccount, \{number \mapsto (\mathbb{Z}, 0), \ldots\}))$, we can describe the state component update $x.number := 1$ as follows, provided that y is the list of undashed variables in the alphabet, other than x, and cls, sc, atts are as in Examples 4.1 and 4.2.

$$
\begin{array}{l}
x.number := 1 \\
= \{\ \text{assignment to state components}\ \} \\
\mathbf{OO}\left(\begin{array}{l}
\mathcal{D}((BAccount, \{number \mapsto (\mathbb{Z}, 0), \ldots\}).number := (\mathbb{Z}, 1)) \\
\vdash \\
(x' = (BAccount, \{number \mapsto (\mathbb{Z}, 0), \ldots\} \oplus \{number \mapsto (\mathbb{Z}, 1)\})) \wedge (y' = y)
\end{array}\right) \\
= \{\ \mathcal{D} \text{ for state component assignments, mapping replacement}\ \} \\
\mathbf{OO}\left(\begin{array}{l}
\mathcal{D}((BAccount, \{number \mapsto (\mathbb{Z}, 0), \ldots\}).number) \\
\wedge\ \mathcal{D}(\mathbb{Z}, 1) \\
\wedge\ \mathbb{Z} \preceq \mathcal{U}(atts, sc, Account)(number) \vdash \\
(x' = (BAccount, \{number \mapsto (\mathbb{Z}, 1), \ldots\})) \wedge (y' = y)
\end{array}\right) \\
= \{\ \mathcal{D} \text{ for state component accesses and primitive instances, definition of } \mathcal{U}\ \} \\
\mathbf{OO}\left(\begin{array}{l}
\mathcal{D}((BAccount, \{number \mapsto (\mathbb{Z}, 0), \ldots\})) \\
\wedge\ BAccount \in cls \\
\wedge\ (\{number \mapsto (\mathbb{Z}, 0), \ldots\} \neq \textbf{null}) \\
\wedge\ number \in \mathrm{dom}\{number \mapsto (\mathbb{Z}, 0), \ldots\} \\
\wedge\ \mathbb{Z} \preceq \mathbb{Z} \\
\vdash \\
(x' = (BAccount, \{number \mapsto (\mathbb{Z}, 1), \ldots\})) \wedge (y' = y)
\end{array}\right) \\
= \{\ \mathcal{D} \text{ for object instance, set properties, assumptions on } cls, \text{ subtyping}\ \} \\
\mathbf{OO}\left(\begin{array}{l}
BAccount \in cls \wedge (BAccount, \{number \mapsto \ldots\}) \in \mathcal{V}(BAccount) \\
\vdash \\
(x' = (BAccount, \{number \mapsto (\mathbb{Z}, 1), \ldots\})) \wedge (y' = y)
\end{array}\right) \\
= \{\ \text{definition of } \mathcal{V}, \text{ set properties}\ \} \\
\mathbf{OO}(\textbf{true} \vdash (x' = (BAccount, \{number \mapsto (\mathbb{Z}, 1), \ldots\})) \wedge (y' = y))
\end{array}
$$

If we had not initialised the variable x, the assignment would not be well-defined and would abort.

7.3 Conditional

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

$$P \lhd e \rhd Q \;\widehat{=}\; \mathbf{OO}\,(\mathcal{D}(P \lhd e \rhd Q) \land ((e_v \land P) \lor (\neg e_v \land Q)))$$

For example, suppose we have the variables cls, sc, and $atts$ as in the Examples 4.1 and 4.2. If we declare a variable $\mathbf{var}\ \mathbf{self} : Account$, and initialise it as $\mathbf{self} := \mathbf{new}\ BAccount$, the result of the conditional $P \lhd \mathbf{self}\ \mathbf{is}\ BAccount \rhd Q$, with arbitrary P and Q, is P, as shown below.

$$\begin{array}{l}
P \lhd \mathbf{self}\ \mathbf{is}\ BAccount \rhd Q \\
= \{\ \text{conditional, type test}\ \} \\
\mathcal{D}(P \lhd \mathbf{self}\ \mathbf{is}\ BAccount \rhd Q) \\
\land \left(\begin{array}{l} ((\mathbb{B}, (BAccount, \{number \mapsto \ldots\})_t \preceq BAccount)_v \land P) \\ \lor\ (\neg(\mathbb{B}, (BAccount, \{number \mapsto \ldots\})_t \preceq BAccount)_v \land Q) \end{array} \right) \\
= \{\ \text{type selection}\ \} \\
\mathcal{D}(P \lhd \mathbf{self}\ \mathbf{is}\ BAccount \rhd Q) \land \left(\begin{array}{l} ((\mathbb{B}, BAccount \preceq BAccount)_v \land P) \\ \lor\ (\neg(\mathbb{B}, BAccount \preceq BAccount)_v \land Q) \end{array} \right) \\
= \{\ \mathcal{D}\ \text{for conditional, value selection, assumptions on}\ cls\text{, subtyping}\ \} \\
\mathcal{D}(\mathbf{self}\ \mathbf{is}\ BAccount) \land ((\mathbb{B}, (BAccount, \{number \mapsto \ldots\})_t \preceq BAccount)_t = \mathbb{B}) \\
\land\ ((\mathbf{true} \land P) \lor (\mathbf{false} \land Q)) \\
= \{\ \mathcal{D}\ \text{for type test, type selection}\ \} \\
\mathcal{D}(\mathbf{self}) \land BAccount \in cls \land (\mathbb{B} = \mathbb{B}) \land P \\
= \{\ \mathcal{D}\ \text{for}\ \mathbf{self}\text{, assumptions on}\ cls\ \} \\
\mathbf{self}_t \in cls \land P \\
= \{\ \text{assumptions on}\ cls\ \} \\
P
\end{array}$$

If the type test were false, then 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 to abort. It may be the case that P is not well-defined; in this case, abortion arises from the definition of P.

7.4 Recursion

Recursion is as the least fixed-point in the complete lattice of parametrised programs partially ordered by refinement. For each parameter declaration, the set of programs with those parameters is a complete lattice; refinement is defined pointwise [2].

Definition 7.1 (Recursive Operation) The general form for the declaration of a recursive operation m of class A is the following.

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

For example, an operation to calculate factorials could be added to a class A as follows

$$\textbf{meth}\ A\ m\ =\ \mu\, X \bullet (\textbf{val}\ n : \mathbb{Z}\ ;\ \textbf{res}\ r : \mathbb{Z} \bullet r := 1 \lhd n \leq 0 \rhd r := n * X(n-1, r))$$

This defines a recursive method using the least fixed-point operator μ. The body of the recursion has a value parameter n and a result parameter r. The body itself is a conditional that checks if the parameter n is non-positive. If it is, then the recursion terminates with the result parameter being set to 1; otherwise, the result is set to $n * X(n-1, r)$.

This is not in conflict with the expected form of an operation declaration $\textbf{meth}\ A\ m\ = (pds \bullet p)$, 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. For each parameter declaration, we take the fixed point in the lattice of parametrised programs with those parameters.

Definition 7.2 (Mutually Recursive Operations) The general form for the declaration of mutually recursive operations m of class A, and n of class B is the following.

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

Mutual recursion is easily addressed in our theory. In this case, since m and n are mutually recursive, they are defined together, even though they are operations 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 predicates to vectors of predicates 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 operations m or n defined below, with a variable a as the result parameter, leads to the assignment of 0 to a.

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

Once the recursion is resolved and the fixed-point operators are eliminated, the description of a multiple operation declaration like $\textbf{meth}\ A\ m, B\ n\ = ((pds_m \bullet mb), (pds_n \bullet nb))$ is a trivial extension of the definition of simple operation declarations presented in Section 4. In many theories of object-orientation, mutual recursion is a difficulty. The complication is really attached to the fact that the mutually recursive operations 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.

7.5 Operation Call

Since we have already solved the problem of dynamic binding when dealing with the semantics of operation declaration in Section 4.3, the semantics of operation call is just a call to the value of the operation. In other words, we have isolated the various aspects involved in an operation call, so that dynamic binding is captured in the definition of the value of the operation variable, which holds a parametrised program, and an operation call is given mainly by the copy rule.

$$\begin{array}{l}le.m(args)\ \widehat{=}\ \textbf{OO}\,((\mathcal{D}(le.m(args)))_\perp\ ;\ (pds_e \bullet p)(le, args)) \\ \textbf{where}\ (m = (\mathtt{pds}_e\ \mathtt{@}\ \mathtt{p}))\end{array}$$

Example 7.1 Suppose that $(sc = sc_0)$, $(cls = cls_0)$, and $(atts = atts_0)$, and after the declaration of classes, state components and operations in the Examples 4.1 and 4.2, we have the program fragment below.

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

Due to dynamic binding, $a.credit(10)$ must execute the body of the operation *credit* defined for the subclass *BAccount*. As described in Section 4, the value of *credit* is a conditional over the special variable named ***self***. Below, we show how the program associated to the variable *credit* resolves the dynamic binding. The meaning of $a.credit(10)$ is defined in terms of $credit(a, 10)$, which we consider below.

$$
\begin{array}{l}
credit(a, 10) \\
= \{\ \text{operation expansion}\ \} \\
\left(\begin{array}{l} \textbf{\textit{vres self}} : \textbf{\textit{Object}}\ ;\ \textbf{\textit{val}}\ x : \mathbb{Z} \bullet \\ \textbf{\textit{self}}.bonus \ldots \lhd \textbf{\textit{self is}}\ BAccount \rhd (\ldots \lhd \textbf{\textit{self is}}\ Account \rhd \bot_{OO}) \end{array} \right) (a, 10) \\
= \{\ \text{semantics of } \textbf{\textit{vres}}\ \} \\
\textbf{\textit{var self}} : \textbf{\textit{Object}}\ ; \\
\quad \textbf{\textit{self}} := a\ ; \\
\quad \left(\begin{array}{l} \textbf{\textit{val}}\ x : \mathbb{Z} \bullet \\ \quad \textbf{\textit{self}}.bonus \ldots \lhd \textbf{\textit{self is}}\ BAccount \rhd (\ldots \lhd \textbf{\textit{self is}}\ Account \rhd \bot_{OO}) \end{array} \right) (10)\ ; \\
\quad a := \textbf{\textit{self}}\ ; \\
\textbf{\textit{end self}} : \textbf{\textit{Object}} \\
= \{\ \text{semantics of } \textbf{\textit{val}}\ \} \\
\textbf{\textit{var self}} : \textbf{\textit{Object}}\ ; \\
\quad \textbf{\textit{self}} := a\ ; \\
\quad \textbf{\textit{var}}\ x : \mathbb{Z}\ ; \\
\quad\quad x := 10\ ; \\
\quad\quad \textbf{\textit{self}}.bonus \ldots \lhd \textbf{\textit{self is}}\ BAccount \rhd (\ldots \lhd \textbf{\textit{self is}}\ Account \rhd \bot_{OO})\ ; \\
\quad \textbf{\textit{end}}\ x : \mathbb{Z}\ ; \\
\quad a := \textbf{\textit{self}}\ ; \\
\textbf{\textit{end self}} : \textbf{\textit{Object}} \\
= \{\ \textbf{\textit{self}}_t \text{ is } BAccount\ \} \\
\textbf{\textit{var self}} : \textbf{\textit{Object}}\ ; \\
\quad \textbf{\textit{self}} := a\ ; \\
\quad \textbf{\textit{var}}\ x : \mathbb{Z}\ ; \\
\quad\quad x := 10\ ; \\
\quad\quad \textbf{\textit{self}}.bonus := \textbf{\textit{self}}.bonus + 1\ ; \quad \textbf{\textit{end}}\ x : \mathbb{Z}\ ; \\
\quad\quad \textbf{\textit{self}}.balance := \textbf{\textit{self}}.balance + x\ ; \\
\quad a := \textbf{\textit{self}}\ ; \\
\textbf{\textit{end self}} : \textbf{\textit{Object}}
\end{array}
$$

This can be expanded to a predicate that establishes the final value of a to be its initial value with state components updated by assignments. □

8 Conclusions

We have presented a stepwise introduction to object-oriented concepts in the Unifying Theories of Programming and used it to give a denotational semantics to the object-oriented features present in CML. We started with the definition of observational variables and healthiness conditions, that restrict the values of these variables. Closure properties of the healthiness conditions are needed to show, for example, that any two programs (or specifications) that are independently valid for a given healthiness condition can be combined by conjunction, disjunction or sequential composition, and the resulting program is still part of the lattice of predicates.

The declarations of classes, state components and operations are defined in terms of the theory of designs. This theory itself filters the subset of terminating programs, combined with higher-order programming. This allows variables to record the behaviour associated to operations (abstractions), including dynamic binding resolution. We saw that each of these declarations preserves the defined healthiness conditions.

Type checking is an important issue in object-oriented languages. To record typing information, special variables were introduced and well-definedness conditions were specified for the new object-oriented commands and expressions. Those forms of commands and expressions already introduced by Hoare and He for a sequential programming language, however, had to be revised to cope with typing information, including operation calls.

The term *variance* refers to how subtyping between complex types relates to subtyping between their components. Depending on the variance of the type constructor, the subtyping relation may be either preserved, reversed, or ignored. For example, suppose that we have that S is s subtype of T and that $\mathcal{C}(X)$ is a complex type expression in terms of the type parameter X. If $\mathcal{C}(S)$ is a subtype of $\mathcal{C}(T)$, then the type constructor $\mathcal{C}$ is *covariant*. If, on the other hand, $\mathcal{C}(S)$ is a supertype of $\mathcal{C}(T)$, then the type constructor $\mathcal{C}$ is *contravariant*. Finally, if $\mathcal{C}(S)$ is neither a subtype nor a supertype of $\mathcal{C}(T)$, then the type constructor $\mathcal{C}$ is *invariant*. By making type constructors covariant or contravariant instead of invariant, more programs will be accepted as well-typed.

We allow operations covariance and contravariance of arguments and return types.

We have seen also that the separation of declarations in different blocks has allowed the definition of (mutually) recursive operations in a straightforward manner. Furthermore, this has allowed a compositional approach which focuses on each feature in isolation. Another facility is specially related to dynamic binding resolution. When processing an operation declaration, the observational variable responsible for that operation is updated to reflect the new operation's meaning. This might introduce a new observational variable for the operation (when processing the first definition of an operation), or update the value of an existing variable to take into account the dynamic binding resulting from an operation redefinition.

Our theory of object-orientation handles inheritance, recursive data types, dynamic binding, polymorphism, and mutually recursive operations. Some considerations about verification are discussed in the following section.

8.1 Verification

The concept of refinement for UTP is universal reverse implication, which leads to a straightforward approach to verification. As an example, suppose we have the following specification for the operation *debit*: informally, we cannot perform a *debit* without sufficient funds.

$$\textbf{\textit{pre self}}.balance > value$$
$$\textbf{\textit{post self}}.balance' = \textbf{\textit{self}}.balance - value$$

To check this specification against its operation definition, at any operation call $le.m(args)$ the semantics could be extended to

$$\textbf{\textit{OO}}\,((\mathcal{D}(le.m(args)) \wedge \mathbf{pre}(args_c))_\perp \,;\, (pds_e \bullet p)(le, args) \,;\, \textbf{\textit{post}}(args_r))$$

where the precondition is extended with the copy-by-value variables (***val*** and ***vres***) and the postcondition is extended with the result variables (***res*** and ***vres***). Notice that if the precondition is violated, the operation call behaves like abort (***true*** ; P = ***true***, from designs), and if the postcondition is violated, it aborts the remaining program.

Moreover, there is a simpler alternative. Remember that in the UTP programs and specifications are interchangeable; the verifications thus would be part of the operation body itself, and in this case the operation call semantics remains the same. The expression

$$\textbf{\textit{OO}}\,((\mathcal{D}(le.m(args)))_\perp \,;\, (pds_e \bullet \textbf{\textit{pre}} \,;\, p \,;\, \textbf{\textit{post}})(le, args)))$$

is the same as declaring a lengthier operation body with pre- and post-conditions.

Bibliography

[1] R. J. R. Back. Procedural Abstraction in the Refinement Calculus. Technical report, Department of Computer Science, Åbo, Finland, 1987. Ser. A No. 55.

[2] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science.

[3] P. H. M. Borba and A. C. A. Sampaio. Basic Laws of ROOL: an object-oriented language. In *3rd Workshop on Formal Methods*, pages 33–44, Brazil, 2000.

[4] P. H. M. Borba, A. C. A. Sampaio, A. L. C. Cavalcanti, and M. L. Cornélio. Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, 52:53–100, 2004.

[5] A. L. C. Cavalcanti and D. A. Naumann. A Weakest Precondition Semantics for Refinement of Object-oriented Programs. *IEEE Transactions on Software Engineering*, 26(8):713–728, 2000.

[6] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Unifying Classes and Processes. *Software and System Modelling*, 4(3):277–296, 2005.

[7] Joey Coleman and Stefan Hallerstede. *Second release of the COMPASS Tool Tool Grammar Reference, version 1.2*, d31.2c edition, January 2013.

[8] T. L. V. de Lima Santos. *A Unifying Theory of Object-Orientation*. Tese de doutorado, Centro de Informática, Universidade Federal de Pernambuco, 2010.

[9] C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[10] Ioannis T. Kassios. Decoupling in Object Orientation. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2005.

[11] David A. Naumann. Predicate transformers and higher-order programs. *Theor. Comput. Sci.*, 150(1):111–159, 1995.