Higher-Order UTP for a Theory of Methods

Frank Zeyda and Ana Cavalcanti

University of York, Deramore Lane, York, YO10 5GH, UK
{frank.zeyda,ana.cavalcanti}@york.ac.uk

Abstract. Higher-order programming admits the view of programs as values and has been shown useful to give a semantics to object-oriented languages. In building a UTP theory for object-orientation, one faces four major challenges: consistency of the program model, redefinition of methods in subclasses, recursion and mutual recursion, and simplicity. In this paper, we discuss how the UTP treatment of higher-order programs impacts on these issues and propose solutions to emerging problems. Our solutions give rise to a novel UTP theory of methods.

Keywords: object-orientation, semantics, recursion, consistency

1 Introduction

Higher-order programming is a paradigm that admits programs as values. Using the notation $\{\!|p|\!\}$ for the program value p, $\mathbf{var}\, m \bullet m := \{\!|x := x + 1|\!\}\ ;\ \mathbf{call}\, m$, for instance, is equivalent to the program $x := x + 1$. Thus, the local variable m holds a program value whereas x is an integer variable. Generally, we have that $\mathbf{call}\, \{\!|p|\!\}$ is equivalent to p, hence $\mathbf{call}\ \_$ can be regarded as the inverse of $\{\!|\_|\!\}$.

Higher-order programming has many useful applications and is prevalent in guise in many modern programming languages. An extensive semantic account based on predicate transformers is given in [9,10]. Our motivation is to reason about object-oriented programs: we take method definitions as assignments to program-valued variables that can be updated by method redefinitions.

There are a number of challenges in defining a comprehensive semantics of an object-oriented language and reasoning about object-oriented programs. Many of these have been addressed, for example, in [8,1,6,14]. Our objective in this paper is, however, not to present a comprehensive model for object-orientation but complement existing and on-going research efforts by presenting practical solutions to issues arising from the modelling and redefinition of class methods.

In [7], Hoare and He examine the integration of higher-order predicates into the Unifying Theories of Programming (UTP) framework. This includes the treatment of programs as values as well as procedures with parameters. A program value is said "to range over predicates, or rather some subset of predicates (or programs)". The word 'program' is used in the UTP sense here, thus referring to a predicate that is constructed from a syntactic (program) expression.

To build on such a theory to reason about (object-oriented) programs, we face four major challenges. The first is a consistent account of the notion of program value. Program values are used to specify the behaviour of methods.

As mentioned above, we do not handle a complete object-oriented theory; our focus is on methods. Nonetheless, due to the modular nature of the UTP, our work can be combined with existing UTP theories that address complementary features such as modelling of classes and inheritance, as well as object references.

To reason about higher-order programs written in a particular language, we require a method by which program values can be constructed. This usually induces a particular model for program values in terms of predicates and begs the question if there are constraints that such a model has to satisfy.

For instance, does the model have to be an encoding of some form of syntax or can we equate program values directly with predicates of a suitable UTP theory? The latter would be tempting as it is in harmony with the philosophy of the UTP, which is agnostic to syntactic issues and focuses on the semantic properties of objects. However, there are potential pitfalls. To illustrate this, we assume that program values are higher-order predicates themselves. The $\{\!|\_|\!\}$ operator then becomes a type constructor that turns a higher-order predicate into a (program) value. Because the set of predicates with a single variable whose value ranges over some type is equipotent to the subsets of such values, the domain of the $\{\!|\_|\!\}$ function would have a higher cardinality than its range. Therefore, its use as a type constructor is unsound (it fails to be injective). This is a well known issue in recursive data type definitions, and is, for example, illustrated in [15]. We can remedy the situation by confining ourselves to finitely expressible predicates. For an arbitrary UTP theory, however, we usually do not assume that all elements of the underlying predicate lattice are finitely expressible; this would already amount to narrowing the discourse to more specific families of theories.

The above hence shows that we cannot admit just any predicate as a program value. The account in [7] does in fact restrict the admissible higher-order predicates indirectly by constraining the type of higher-order variables. This effectively excludes recursions like $p := \{\!|x := x + 1 \,;\ \mathbf{call}\, p|\!\}$.

A second challenge that we face is method redefinition. In a view of methods as program variables, a method redefinition is an update to an existing program value. In the UTP theory of object-orientation presented in [14], for instance, this is handled by relying on the fact that program values are syntactic elements of a particular form that reflects the hierarchy of classes where the method is (re)defined. This is a simple elegant solution that enables the use of the copy rule to give semantics to method calls. On the other hand, it ties the theory to a specific syntax of programs, which is against the UTP philosophy.

Recursion, and, in particular, the extensive use of mutual recursion in object-oriented programs impose a third challenge. As already explained, the theory in [7] does not permit the use of a program variable itself in its value. This means that recursion has to be treated in the context defined by the particular notion of programs. In [14], this is achieved by taking fixed points in the UTP theory that is used to give semantics to the syntactic elements taken as program values. To treat mutual recursion, it is therefore necessary to give semantics to all method definitions (and their redefinitions) together. This is illustrated by the following example, where we have two program variables m_1 and m_2 that

represent methods that mutually call each other (they calculate $\mid x - y \mid$). Prior to encoding the methods as a predicate, the recursions have to be eliminated.

$$m_1, m_2 := \{\!|\, \mu\, X, Y \bullet \left\langle \begin{array}{l} (x := x - 1\,;\ \ Y) \lhd x > 0 \rhd \mathbf{II}, \\ (y := y - 1\,;\ \ X) \lhd y > 0 \rhd \mathbf{II} \end{array} \right\rangle |\!\}$$

As shown, this gives rise to fixed-point constructions in the program values. Another issue arises if m_1 is redefined later on. Such a redefinition does not merely affect the value of m_1, but also m_2 as the fixed point needs to be calculated afresh. Though this approach is feasible, it forfeits compositionality.

A final challenge is simplicity. Higher-order programs are just one of the many aspects of an object-oriented program. We strive for simplicity, although this can be fully appreciated only once we combine our theory with other UTP theories (to cater to concurrency, time, sharing, and so on).

Our contribution in this paper is to examine solutions to all these challenges to provide a UTP theory that can be used in the context of a theory of object-orientation like that in [14]. We first illustrate the construction of a sound semantic model for higher-order predicates. Importantly, our program model does not encode programs as syntax, but directly in terms of their semantics as predicates. We show how, in spite of that, we can still cope with method redefinition by using a combination of syntax and semantics in the program model; this also turns out to be useful for the semantic encoding of procedures with parameters.

Finally, we provide a sound solution for the (mutual) recursion problem. This does not affect the underlying semantic model of higher-order predicates and hence does not compromise consistency.

The structure of the paper is as follows. In Section 2 we review the UTP in its higher-order version. Section 3 describes a consistent model of higher-order predicates that is based on predicates rather than a fixed syntax. In Section 4 we propose a UTP theory of methods that overcomes the restriction on the use of recursion in [7]. Section 5 includes some discussions and revisits the initial problem presented above, and in Section 6 we report on related and future work.

2 Preliminaries

In this section, we discuss specific features of higher-order UTP, assuming the reader is familiar with standard UTP. We also give some brief background on the theory of object-orientation in [14], which motivated the work in this paper.

2.1 Higher-Order UTP

The Unifying Theories of Programming [7] is a mathematical framework that provides means for defining the semantics of a variety of programming languages and modelling notations. The primal extension in higher-order UTP is the inclusion of procedure variables. Procedure variables are declared and used just like standard variables. For example, $\mathbf{var}\, p : \mathit{proc}_{\{x,x'\}}\ ;\ p := \{\!|x := x + 1|\!\}$ introduces a (local) procedure variable p that holds predicates whose alphabet is $\{x, x'\}$, and assigns to it the program $x := x + 1$. Procedure values are directly

identified with predicates of some theory of designs or programs. The purpose of $\{\!|\_|\!\}$ is merely "to distinguish what is to be stored from what is to be executed" as stated in [7]. Otherwise, the brackets have no semantic significance and are simply omitted in a procedure call.

Procedure variables can be written as executable statements in a predicate. For example, $\mathbf{var}\, p\ ;\ p := \{\!|x := x + 1|\!\}\ ;\ p\ ;\ \mathbf{end}\, p$ is equivalent to $x := x + 1$. The type of p has been omitted in the declaration, but we note that all variables, whether they are standard variables or procedure variables, need to have a type. The notion of a procedure type is explained in more detail in the next section. For clarity, we hereafter make the invocation of a procedure variable explicit by writing $\mathbf{call}\, p$ rather than just p as in [7].

A fundamental law about procedure calls is recaptured below.

$$(p := \{\!|Q|\!\}\ ;\ \mathbf{call}\, p)\ =\ (p := \{\!|Q|\!\}\ ;\ Q)$$

It entitles us to replace the invocation of a procedure by its definition and can be regarded as a manifestation of the copy rule.

An intricacy in higher-order UTP arises from the desire that procedure assignment ought to be monotonic with respect to refinement. Formally,

$$P \sqsubseteq Q\ \Rightarrow\ (p := \{\!|P|\!\}) \sqsubseteq (p := \{\!|Q|\!\})$$

This cannot be true if assignment has its standard meaning of equating the primed variable with the assigned expression. This issue and a new definition of refinement were first discussed in [9]. Hence, in higher-order UTP, the meaning of assignment is modified. Here, the semantics of $p := \{\!|Q|\!\}$ is a non-determinism that constrains the value of p' to be any refinement of Q.

$$p := \{\!|Q|\!\}\ \widehat{=}\ (\mathbf{true} \vdash (Q \sqsubseteq p')) \wedge (v \sqsubseteq v')$$

where $\alpha(p := \{\!|Q|\!\}) = \{p, p', v, v'\}$. The new definition implies that we require a notion of refinement of values. For standard values, this is just a flat order, and for program values it is the underlying refinement order on predicates.

Procedures with parameters are supported through functions that map values or variables to (higher-order) predicates. This is essentially the approach that is described in [2]. Permitted are both value and result parameters, and their semantics is expressed in terms of a more general construct $\{\!|\, \lambda\, x : var(T) \bullet P|\!\}$ which corresponds to a procedure that takes a variable of type T as a parameter.

In [7] further aspects of the theory are discussed related to functions and declarative programming. They are not relevant for the material in this paper though. In terms of terminology, we shall use the word 'program' from here on in preference of 'procedure' and reserve the later for programs with parameters. We next give a brief summary of Santos' theory of object-orientation.

2.2 A theory of object-orientation

The theory in [14] builds on an integration of the theory of UTP designs and higher-order programs. The theory introduces observational variables that determine declared classes, their attributes, as well as the subclass order. Methods

are encoded via higher-order program variables, and only one variable is used for all redefinitions (overridings) of a method in subclasses.

The theory supports declarations of classes, attributes and methods, and hence entails the possibility to reason about class and method definitions, as well as particular object-oriented programs. Dynamic binding is supported by imposing a certain syntactic structure on method definitions that resolves method binding as part of the method invocation. Namely each value of a method variable has a fixed syntactic structure illustrated below.

$$(p_1 \lhd \mathit{self} \textbf{ is } C_1 \rhd (p_2 \lhd \mathit{self} \textbf{ is } C_2 \rhd (\ldots (p_n \lhd \mathit{self} \textbf{ is } C_n \rhd \bot_{oo}) \ldots)))$$

Above, *self* is an auxiliary variable that determines the target of a method invocation. The p_i are basically specifications of the same method, albeit defined in different subclasses C_1, C_2, ..., C_n. The cascade of tests is used to resolve dynamic binding when the method is called on an object, with tests against more concrete types being carried out before tests against more abstract types.

Method redefinition in a class C has to inject a new test $(p \lhd \mathit{self} \textbf{ is } C \rhd ...)$ at the right place into this cascade, depending on where C fits into the subclass hierarchy. Redefinition of methods is therefore a syntactic transformation of the top-level cascade of tests; this is made possible in [14] by the fact that programs are uniformly treated as syntax.

3 A program model

Our first challenge is to provide a consistent account of a program model. As already explained, our goal is an account that does not assume a fixed syntax for program values but identifies them directly with the predicates of a UTP theory. This enables us to consider a generic theory of object-orientation, independent of the syntax in which we write, for instance, the body of a method.

On the other hand, to take advantage of the approach in [14] to method redefinition and dynamic binding, we do not exclude syntax entirely. In Section 3.1, we first prove soundness of treating program values directly as predicates of an arbitrary UTP theory. This is a useful insight for any work that uses higher-order UTP. Our motivation, as hinted above, is to eradicate any constraints on the underlying theory in which we express the computational effect of methods when instantiating a generic theory of object-orientation. We then extend this argument (Section 3.2) by making a case for the safe combination of syntax and semantics to support method redefinition as in [14]. This provides us with full flexibility on the one hand to remain in the realm of semantics but escape into syntax where this is beneficial to the model and operator definitions.

3.1 Consistency of higher-order programs

As already pointed out, in a sound program model, program values cannot range over arbitrary predicates. The treatment in [7] rules this out by restrictions on alphabets that effectively prohibit recursion. More precisely, this is done by

introducing a notion of variable type for higher-order predicates that does not admit circularity. The corresponding BNF-like encoding is reproduced below.

$$<\mathit{type}> ::= <\mathit{program\ type}> \mid <\mathit{base\ type}>$$

$$<\mathit{program\ type}> ::= \mathit{ProcType}(<\mathit{alphabet}>)$$

$$<\mathit{alphabet}> ::= \text{list of } (<\mathit{variable}> : <\mathit{type}>)$$

$$<\mathit{base\ type}> ::= \mathit{BaseType}(\mathit{int}) \mid \mathit{BaseType}(\mathit{bool}) \mid \ldots$$

The dots indicate that we might have further type constructors for base values, for instance, to create composite values like pairs or (finite) sets. As long as those constructors are sound and only recursive into $<\mathit{base\ type}>$, this is not an issue and does not invalidate any of the subsequent reasoning.

As briefly discussed in the introduction, with the restriction in [7] to predicates whose variable types are finite terms constructed by the above rules, recursion is effectively excluded. To illustrate this, we consider the invalid predicate

$$p := \{\!|\, x := x + 1 \,;\ \mathbf{call}\, p \,|\!\}$$

In this example, it is already clear though that to define the type of the variable p, we would need to refer to that type itself, and this circularity is not allowed. Mutual recursion gives rise to similar situations. We use $\{\!|\_|\!\}$ only informally here since we have not formally established its existence and semantics yet.

In the sequel we argue that the finitary nature of types is sufficient to ensure consistency. This is a result left implicit in [7]. The argument that we present clarifies important issues related to the treatment of higher-order programs. It can also be used as a basis for a formal treatment of the UTP theory of higher-order programs and its embedding in a theorem prover. Our argument is based on the inductive construction of a model. For this, we first define the notion of the rank of a type inductively over the type structure.

$$\mathit{rank}(\mathit{BaseType}(t)) = 0 \quad \text{and}$$

$$\mathit{rank}(\mathit{ProcType}(\text{list of } [v_1 : t_1, v_2 : t_2, \ldots])) = \mathit{max}\,\{\mathit{rank}(t_1), \mathit{rank}(t_2), \ldots\} + 1$$

Since types are finite by construction, the above recursion properly defines the rank of any given type. We define the rank of a variable to be the rank of its type. The rank of an alphabet is defined as the maximum rank of its variables, and the rank of a predicate is defined just as the rank of its alphabet.

Intuitively, the rank determines the maximal nesting level of program abstractions in a predicate. For instance, the predicates of rank 0 are just the standard predicates; predicates of rank 1 include program variables whose values are standard predicates; predicates of rank 2 moreover admit program values being rank 1 predicates, and so on. Thus, $x := 1$ is a rank 0 predicate, $m_1 := \{\!|\, x := 1 |\!\}$ is a rank 1 predicate, and $m_2 := \{\!|\, x := 1 \,;\ \mathbf{call}\, m_1 |\!\}$ is a rank 2 predicate.

The motivation for introducing a notion of rank is twofold: first we observe that it allows us to partition all higher-order predicates into an enumerable succession of higher-order predicate subsets since every valid predicate must have a finite rank. Secondly, we shall see that the concept of ranks is also central in a theory of methods, which we propose and discuss in Section 4.

We next give a constructive definition of a function $pred(n)$ that yields the predicates of a given rank; our motivation is to subsequently use it to construct the predicates of arbitrary ranks, and as mentioned in the last paragraph, these encompass all valid higher-order predicates. We name *StdPred* the standard (non-higher-order) predicates and define, again inductively,

$$pred(0) = StdPred \quad \text{and} \quad pred(n+1) = lift(pred(n), pred(n))$$

This definition rests on the existence of a lifting function $lift\,(ps, vs)$, which takes a set of predicates ps and lifts them into a set of predicates that introduce program variables that range over the values in vs, which are predicates themselves. By way of an example, we have that $pred(1) = lift(StdPred, StdPred)$. These are the standard predicates augmented with variables whose values can range over standard predicates. We can convince ourselves that in general the application of $lift\,(ps, ps)$ admits predicates one rank higher than those in ps.

A precise constructive definition of *lift* can only be given with respect to a core semantic encoding of predicates, like the one in [12], which characterises them in terms of binding sets. Rather than defining *lift* for a specific model, we instead present an abstract axiomatic characterisation that relies on four operators, $\boxed{\alpha}\,p$, $m\ \boxed{=}\ v$, $\boxed{\sqcap}\ ps$ and $\boxed{\sqcup}\ ps$. The value of $lift(ps, vs)$ is equated with the smallest set of predicates hps that satisfies the following five properties.

A1 $ps \subseteq hps$

A2 $\forall\, m : ProcType(l) \bullet \forall\, v : vs \mid SetOf(l) = \boxed{\alpha}\, v \bullet m\ \boxed{=}\ v \in hps$

A3 $\forall\, ps \subseteq hps \bullet \boxed{\sqcap}\ ps \in hps$

A4 $\forall\, ps \subseteq hps \bullet \boxed{\sqcup}\ ps \in hps$

A5 $\boxed{\sqcap}$ and $\boxed{\sqcup}$ are the meet and join of a complete lattice $\boxed{\sqsubseteq}$

The axioms capture elemental correctness properties of the lifting that ensure completeness of the lifted model and that we retain the property of a complete lattice. The boxed operators have to be provided by the core predicate model. Here, $\boxed{\alpha}\,p$ determines the alphabet (set of variables) of a predicate p, $m\ \boxed{=}\ v$ constructs a simple equality between a variable m and a value v, and $\boxed{\sqcap}\ ps$ and $\boxed{\sqcup}\ ps$ are the greatest lower bound and least upper bound of a set of predicates with respect to an ordering that serves as refinement. The latter two operators are moreover used to define disjunction and conjunction of predicates in the lifted model. This is by virtue of $p_1\ \boxed{\vee}\ p_2 = \boxed{\sqcap}\ \{p_1, p_2\}$ and $p_1\ \boxed{\wedge}\ p_2 = \boxed{\sqcup}\ \{p_1, p_2\}$.

The first property **A1** establishes monotonicity, namely that each lift extends the previous predicate rank. From it we can prove, by induction over the rank, that $\forall\, n \leq m \bullet pred(n) \subseteq pred(m)$. The second property **A2** is a family of axioms for each alphabet given by the list l. The alphabet encoded by a list simply corresponds to the elements in the list, and we use the function *SetOf* to obtain the list elements as a set. **A2** introduces new predicates into the lifted model; they are just simple equalities over (new) program variables. We note that generally, the predicates in vs have a variety of types.

A3 and **A4** are closure properties that enable us to construct arbitrary predi-

cates over the added program variables and values. We note that no closure axiom for negation is needed because $\boxed{\neg}\, m = v$, for instance, can be constructed by $\boxed{\sqcap}\, \{w \mid w \neq v \bullet m \,\boxed{=}\, w\}$, the disjunction of all predicates $m = w$ where $w \neq v$.

We can think of **A2** as providing the building blocks for constructing predicates over program variables of the successor rank. If we consider, for example, the lifting of rank 0 predicates, $m = \{\!|x := 1|\!\}$ and $m = \{\!|x := 2|\!\}$ are admitted by **A2** and $m = \{\!|x := 1|\!\} \vee m = \{\!|x := 2|\!\}$ is admitted by **A3**. In this way, the complete lattice of successor rank predicates is constructible. A refinement ordering $\boxed{\sqsubseteq}$ on predicates exists by **A5**. The top and bottom of the lattice are obtained by the meet and join over empty sets: $\boxed{\top} \mathrel{\widehat{=}} \boxed{\sqcap}\, \{\}$ and $\boxed{\bot} \mathrel{\widehat{=}} \boxed{\sqcup}\, \{\}$.

The question of the semantics of *lift* has now been pushed into the definition of $\boxed{\alpha}\, p$, $m \,\boxed{=}\, p$, $\boxed{\sqcap}\, ps$ and $\boxed{\sqcup}\, ps$ in a core predicate model. For their interpretation in that model, we require that the operators obey the algebraic laws that are presented in [7]. This validates the soundness of the operator definitions in the lifted predicate model. We next define the set *pred* as follows.

$$pred = \bigcup \{n \in \mathbb{N} \bullet pred(n)\}$$

It contains all predicates of any rank. We claim that if *StdPred* are the standard predicates, and the boxed operators are soundly defined, in the above sense, *pred* is also a model for precisely the higher-order predicates considered in [7]. The axioms **A1** to **A5** are sufficient to establish this. The purpose and motivation for the *lift* function now becomes clear as being primarily a utility for constructing the entire set of admissible higher-order predicates.

To conclude the consistency argument, we observe that $\{\!|\_|\!\}$ only has to be injective on the predicates that are well-formed, thus having non-circular types as introduced above. We trivially define it as follows.

$$\{\!|\_|\!\} =_{\mathrm{df}} (\lambda\, p : pred \bullet p) \quad \text{where} \quad \mathrm{dom}\, \{\!|\_|\!\} = pred$$

It is simply the identity on *pred*. Clearly, $\{\!|\_|\!\}$ is injective on *pred*, so it serves as a sound type constructor for program values. We have thus shown that it is safe to treat higher-order UTP predicates as semantics just like the standard ones, and in doing so also illustrated the layered construction of a predicate model. The cardinality of values from $<$*base type*$>$ is moreover irrelevant. Namely, the carrier sets of base value types may be infinite, even uncountably so.

3.2 Syntax and semantics in program values

We have now established the use of predicates directly as program values. On the other hand, in order to support the approach in [14] for redefinition of methods in subclasses, it turns out that part of the program value in fact has to be kept as syntax as explained in Section 2.2. Our treatment views them as predicates and, despite the discussed benefits, this invalidates the transformational approach. Our solution is to alter the iterative definition of $pred(n)$ as follows.

$$pred(n+1) = lift(pred(n), embed(pred(n)))$$

The only modification is the application of a function *embed* to the set of predicates that determines the values of programs at the next rank. This function

realises the syntactic embedding of the semantic entities. The definition of *lift* remains fundamentally the same. The only implication is that the $\boxed{\alpha}$ function in **A2** now has to extract the alphabet of a predicate that is embedded in a segment of syntax. This is not a problem: we can define the extraction function inductively over the data type that encodes the syntactic structure.

In the above example, the syntax is specified by the following generic data type that represents a method in [14]. (We use the Z notation [16].)

$$METH[PRED] ::= CondSytx\ \langle\langle METH \times CVALUE \times METH \rangle\rangle \mid BotSytx \mid Body\ \langle\langle PRED \rangle\rangle$$

This is a Z definition of a new data type $METH$, which is generic ($PRED$ is a type parameter). As usual, the bar is used to separate the definition of type-constructor functions and between $\langle\langle \ldots \rangle\rangle$ brackets, we specify the types of those functions. The type constructor $CondSytx$ encodes the syntax $\underline{c}_1 \lhd self$ **is** $C \rhd \underline{c}_2$, where the underlined elements may themselves be pieces of syntax. $BotSytx$ encodes the syntax of $\bot_{oo}$, the bottom element in the theory of [14]. The constructor $Body$ is non-recursive and injects the semantics of a method body as a predicate, supplied by an element of the generic type $PRED$, into the syntactic domain defined by $METH$. Hence we have $embed(ps) = METH[ps]$.

We note that despite the presence of the *embed* function in the lifting, the result of the lifting is still a predicate set. On the other hand, the call operation has to be adjusted when identifying $METH[PRED]$ with program values. We require an additional layer of denotation in the definition of **call** m that turns a value from $METH[PRED]$ into a value of $PRED$. This can be achieved by interpreting the conditional and bottom with their usual definitions in the UTP. The denotation is inductively defined over $METH[PRED]$. It also serves as a basis for defining refinement on the syntactic program values.

Our conclusion in this section is that we have a certain leeway to mix syntax and semantics, as long as we can provide a way of embedding the semantics into the syntax and provide a denotation in terms of the embedded predicate model. Having established the soundness of a suitable program model for our purposes, in the next section we examine issues that emerge from method redefinition.

4 A theory of methods

In this section, we illustrate a fundamental challenge posed by method definition and redefinition in theories of object-orientation. This motivates us to propose a novel UTP theory of methods that overcomes the problem. It exploits the notion of programs as predicates, as established in the previous section, and is applicable and useful in any context where higher-order variables are used to record method behaviour. Importantly, it restores the simplicity of the treatment in [14] in the view of the issues raised and thereby paves the way for a compositional semantics.

In Section 4.1 we illustrate a fundamental problem with method redefinition in theories of object-orientation, and in Section 4.2 we present our solution. As

mentioned, the primary motivation is to solve issues of compositionality when defining methods, but also to unify the treatment of method (re)definition.

4.1 Method definition revisited

As an example, we consider the following higher-order predicate.

$$S_1 \;\widehat{=}\; m_1 := \{\!|x := x + 1|\!\} \;;\; m_2 := \{\!|x := x + 2 \;;\; \mathbf{call}\, m_1|\!\}$$

It captures the definition of two methods, recorded by the program variables m_1 and m_2. We observe that m_1 is a rank 1 variable whereas m_2 is a rank 2 variable. Hence, the predicate S_1 is a rank 2 predicate.

We first observe that, in general, to encode programs by way of method variables, we cannot restrict ourselves to predicates of a rank lower than 2. This begs the question whether rank 2 is enough to encode all possible object-oriented programs? Unfortunately, the answer is 'no'. For instance, assume we compose the predicate S_1 with the definition of another method m_3.

$$S_2 \;\widehat{=}\; S_1 \;;\; m_3 := \{\!|x := x + 3 \;;\; \mathbf{call}\, m_2|\!\}$$

Clearly, the rank of variable m_3 has to be one greater than the one of variable m_2. This renders S_2 a rank 3 predicate. The issue is subtle because it depends on the careful accounting for types in program variables. This has important implications. In deciding the type of m_3, we need to have knowledge of the type of m_2 in S_1 — its name is not enough. This is because the alphabet of m_3 does not merely include standard variables for the inputs and outputs of the method, but also program variables for methods that are called by the method; and clearly, the type of m_3 depends on the type(s) of those variables too.

This means that in general, we cannot give a compositional account of method definition in cases where a method calls other methods, unless we make the type of the called method(s) a parameter of that definition. In that case, the definition of S_2 would not be a predicate but a function that, if applied to a type, yields a predicate, and further mechanisms would have to be put into place to instantiate this type parameter. This kind of treatment is not unsound, in particular with our result of admitting predicates of any rank, but it considerably complicates the theory of object-orientation and its application.

Method redefinition further complicates matters because it can result in the type of a method variable having to *change*. We consider the scenario where we introduce another method m_4 and then redefine m_1 to call it.

$$S_3 \;\widehat{=}\; S_2 \;;\; m_4 := \{\!|x := x + 4|\!\} \;;\; m_1 := \{\!|\, \mathbf{call}\, m_4|\!\}$$

The variable m_1 above cannot possibly be the same m_1 as in the definition of S_1 because there its rank is 1 whereas here its rank has to be at least 2. We can envisage a solution in which we know in advance that m_1 would subsequently be redefined in terms of a program with a higher rank, and already use that higher rank in typing m_1 in S_1. Such knowledge, however, is doubtful in practice and certainly not available in a compositional treatment. In consequence, we have

to redefine m_1 together with all previous method definitions that depend on its value. This gives rise to

$$S_{3b} \;\widehat{=}\; \begin{pmatrix} m_1 := \{\!|\, \mathbf{call}\, m_4 \,|\!\}\ ;\ \ m_2 := \{\!|x := x+2\ ;\ \ \mathbf{call}\, m_1|\!\}; \\ m_3 := \{\!|x := x+3\ ;\ \ \mathbf{call}\, m_2|\!\}\ ;\ \ m_4 := \{\!|x := x+4|\!\} \end{pmatrix}$$

We thereby tame the impact of the type change of m_1 by adjusting the types and definitions of all method variables that directly or indirectly call m_1.

A similar problem arises even when redefinition does not involve a predicate of a higher rank. To illustrate this, instead of m_1 we redefine m_2 in S_2.

$$S_4 \;\widehat{=}\; S_2\ ;\ \ m_4 := \{\!|x := x+4|\!\}\ ;\ \ m_2 := \{\!|\, \mathbf{call}\, m_4|\!\}$$

The rank of the new program value of m_2 is the same as before, so this is not an issue. However, originally the variable m_4 was not in the alphabet of m_2. Introducing it during redefinition is again problematic since this changes the type of m_2, giving rise to exactly the same issues as illustrated before (since m_3 calls m_2). We could try and include all other method variables in the alphabet of any method variable we introduce. But then, what rank(s) should those other method variables have? This decision again imposes *a priori* restrictions on what calls between methods are permissible at a future point; this is not practical.

As a note, the finite nature of alphabets prohibits inclusion of all method variables, but we can get around this in practice by using a finite but large enough repository of method variables. Although this style of modelling is somewhat against the philosophy of the UTP, where alphabets are used in meaningful ways, it is difficult to avoid even in the solution we propose in the sequel.

Motivated by the above observations, we next present a treatment in which the rank of any method variable is not greater than 2. The rank of a predicate encoding an object-oriented program is thus not greater than 2 either.

4.2 A UTP theory of methods

We present our theory in the usual UTP style. The observational variables of the theory are program variables that represent methods. We only include program variables at rank 1 and rank 2 and call them method variables hereafter. The rationale for this is that all method definitions we introduce shall constrain rank 2 variables, while all calls within those definitions will be to rank 1 variables.

In the sequel we use overbars to highlight the rank of a method variable. Thus $\overline{m}$ is a rank 1 method variable and $\overline{\overline{m}}$ is a rank 2 method variable. No overbar indicates a standard program variable (rank 0). We note that the overbars are mere annotations that highlight the type of the variable.

To illustrate the main idea, below we encode the predicate S_1 presented earlier on in Section 4.1. We name it T, rather than S, to emphasise that this predicate belongs to the theory of methods we develop here.

$$T_1 \;\widehat{=}\; \overline{\overline{m}}_1 := \{\!|x := x+1|\!\}\ ;\ \ \overline{\overline{m}}_2 := \{\!|x := x+2\ ;\ \ \mathbf{call}\, \overline{m}_1|\!\}$$

Close inspection reveals an important difference: the call is to $\overline{m}_1$ rather than to $\overline{\overline{m}}_1$ as it was the case in S_1. Method assignment is uniformly carried out to

rank 2 variables, highlighted by two overbars in the assigned method variables $\overline{\overline{m}}_1$ and $\overline{\overline{m}}_2$. Although, in principle, the first assignment could be to a rank 1 variable, our approach puts uniformity above such *ad hoc* optimisations.

Next, we sequence T_1 with a predicate that introduces another method that calls m_2, as we did in S_2. This now yields

$$T_2 \;\widehat{=}\; T_1 \,;\; \overline{\overline{m}}_3 := \{\!|x := x + 3 \,;\; \mathbf{call}\, \overline{m}_2|\!\}$$

Once again, the call is to $\overline{m}_2$ rather than $\overline{\overline{m}}_2$. This shows that the rank of method variables does not increase with subsequent definitions of methods, and neither does it increase upon method redefinition. However, $\overline{m}_x$ and $\overline{\overline{m}}_x$ are clearly different variables, and our theory hence has to create a link between them.

This is achieved by a single healthiness condition. It establishes a connection between rank 1 and rank 2 method variables of the same name. To formulate it, we require a way to refer to the name of a variable rather than its identity, which includes its type. To facilitate notation, we shall assume that $\overline{m}$ and $\overline{\overline{m}}$ have the same name, and moreover that a quantification $\forall\, \overline{m}\ \overline{\overline{m}} \bullet P[\overline{m}, \overline{\overline{m}}]$ is over variables that have rank 1 and rank 2 and the same name. In this way, we do not have to talk about names and types explicitly.

The healthiness condition **HM** is defined as follows.

$$\mathbf{HM}(P) \;=\; P \wedge (\forall\, \overline{m}\ \overline{\overline{m}} \mid \{\overline{m}, \overline{\overline{m}}\} \subseteq \alpha\, P \bullet [\mathbf{call}\, \overline{m} \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}]_0)$$

It states that two method variables in P of the same name, but at different ranks, have to be consistent in terms of the constraints they impose on program variables ($[\_]_0$ is the closure operator over standard (program) variables).

We can think of **HM**, together with the constraints imposed on rank 2 method variables by a predicate of the theory, as defining a family of equations that constrain the value of rank 1 method variables and thereby yield an interpretation of methods purely in terms of standard predicates. This interpretation falls out when we quantify over the rank 2 method variables in a healthy predicate and observe the corresponding rank 1 method variables. It corresponds to an encoding of methods in terms of weakest fixed points of a recursive equation that uses recursive parameters instead of method variables. For instance, the predicate $(\exists\, \overline{\overline{m}}_1\ \overline{\overline{m}}_2 \bullet T_1)$ is equivalent to the concurrent assignment

$$\overline{m}_1, \overline{m}_2 := \{\!|\, \mu\, X, Y \bullet \langle x := x + 1, (x := x + 2 \,;\; X)\rangle|\!\}$$

where **call** statements in T_1 have been eliminated by virtue of a multi-variable recursion over standard predicates. For two variables, this takes the general form $\mu X, Y \bullet \langle F(X, Y), G(X, Y)\rangle$. We note that above only G recurses (into X) whereas F depends on neither X nor Y. Such a transformation was already used in [14] to deal with (mutual) recursions. Our claim is that both interpretations are mathematically equivalent. To support this conjecture, we first quote Hoare and He in [7]: "The inclusion of high order variables does not increase the power of the language". Secondly, in the particular example, we can use fixed-point laws to show that X is equivalent to $x := x + 1$ and Y is equivalent to $x := x + 3$. A

formal proof is presented at the end of the section that this is exactly the value of $\overline{m}_1$ and $\overline{m}_2$ in T_1. Proving the general case is still future work and requires a precise definition of how to transform one representation into the other.

We next present some essential properties and laws of our theory.

Closure of operators The notion of a conjunctive healthiness condition is formulated in [5] and means that the healthiness condition can be expressed in the form $\mathbf{CH}(P) = P \wedge \gamma$ for some constant predicate γ. In our case, that predicate is not constant though, as it depends on the alphabet of P. In particular, we have $\mathbf{HM}(P) = P \wedge \gamma_{\mathrm{HM}}\,(\alpha\, P)$ where

$$\gamma_{\mathrm{HM}}\,(a) \;=\; (\forall\, \overline{m}\; \overline{\overline{m}} \mid \{\overline{m}, \overline{\overline{m}}\} \subseteq a \bullet [\mathbf{call}\, \overline{m} \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}]_0)$$

Despite this, we can recover essential closure properties that hold for conjunctive healthiness conditions. They are, however, subject to additional caveats. To formulate them, we first require a notion of compatibility of alphabets.

Definition 1. *Two alphabets a_1 and a_2 are compatible if, and only if,*

$$\forall\, \overline{m}\; \overline{\overline{m}} \bullet (\overline{m} \in a_1 \wedge \overline{\overline{m}} \in a_2) \Leftrightarrow (\overline{\overline{m}} \in a_1 \wedge \overline{m} \in a_2)$$

Intuitively, compatibility implies that if alphabets share a method variable with the same name but at different ranks, each alphabet has to include both instances of that variable. By way of illustration, the alphabet pairs $(\{\overline{m}_1, \overline{\overline{m}}_1\}, \{\overline{m}_1, \overline{\overline{m}}_1\})$, $(\{\overline{m}_1, \overline{\overline{m}}_1\}, \{\overline{m}_2, \overline{\overline{m}}_2\})$ and $(\{\overline{m}_1\}, \{\overline{\overline{m}}_2\})$ are compatible but $(\{\overline{m}_1\}, \{\overline{\overline{m}}_1\})$ is not.

It is easy to show that compatibility is reflexive and symmetric, however, it is not transitive. The latter we illustrate by observing that $(\{\overline{m}_1\}, \{\overline{m}_2\})$ and $(\{\overline{m}_2\}, \{\overline{\overline{m}}_1\})$ are compatible alphabet pairs, but $(\{\overline{m}_1\}, \{\overline{\overline{m}}_1\})$ is not.

Compatibility of alphabets enjoys closure properties with respect to set operations like union, intersection and difference. The following law specifies them.

Law 1. *Let (a_1, a_2) and (a_1, a_3) be compatible alphabets. Then,*

$$(a_1, a_2 \cup a_3)\;,\; (a_1, a_2 \cap a_3) \;\text{ and }\; (a_1, a_2 \setminus a_3) \;\text{ are compatible alphabets.}$$

An important property of γ_{HM} is formulated by the following lemma.

Lemma 1. *Let a_1 and a_2 be compatible alphabets. Then we have*

$$\gamma_{\mathrm{HM}}(a_1 \cup a_2) = \gamma_{\mathrm{HM}}(a_1) \wedge \gamma_{\mathrm{HM}}(a_2)$$

The law is proved by splitting the universal quantification in γ_{HM} into a conjunction of two parts in which $\overline{m}$ and $\overline{\overline{m}}$ range over a_1 and a_2, respectively; this succeeds because of the compatibility property. A mechanised theory in Isabelle HOL that proofs the above law and lemma is available [17]. The lemma enables us to prove closure under conjunction of **HM**-healthy predicates.

Law 2. *Let P and Q be* **HM***-healthy predicates with compatible alphabets. Then,*

$$P \wedge Q \;\text{ is a }\; \mathbf{HM}\text{-healthy predicate.}$$

Proof. We show that $P \wedge Q$ is a fixed point of **HM**.

$$
\begin{aligned}
& P \wedge Q \\
\equiv\ & \text{“}P \text{ and } Q \text{ are } \mathbf{HM}\text{-healthy”} \\
& \mathbf{HM}(P) \wedge \mathbf{HM}(Q) \\
\equiv\ & \text{“unfolding definition of } \mathbf{HM}\text{”} \\
& (P \wedge \gamma_{\mathrm{HM}}(\alpha\, P)) \wedge (Q \wedge \gamma_{\mathrm{HM}}(\alpha\, Q)) \\
\equiv\ & \text{“reordering conjuncts”} \\
& (P \wedge Q) \wedge (\gamma_{\mathrm{HM}}(\alpha\, P) \wedge \gamma_{\mathrm{HM}}(\alpha\, Q)) \\
\equiv\ & \text{“Lemma 1”} \\
& (P \wedge Q) \wedge \gamma_{\mathrm{HM}}((\alpha\, P) \cup (\alpha\, Q)) \\
\equiv\ & \text{“rewriting } (\alpha\, P) \cup (\alpha\, Q) \text{ into } \alpha\,(P \wedge Q)\text{”} \\
& (P \wedge Q) \wedge \gamma_{\mathrm{HM}}(\alpha\,(P \wedge Q)) \\
\equiv\ & \text{“folding definition of } \mathbf{HM}\text{”} \\
& \mathbf{HM}(P \wedge Q)
\end{aligned}
$$

Unfortunately, compatibility of alphabets is insufficient for closure under disjunction. There, we require the stronger proviso of the alphabets being equal.

Law 3. *Let P and Q be* **HM***-healthy predicates with equal alphabets. Then,*

$P \vee Q$ *is a* **HM***-healthy predicate.*

In general, if we restrict ourselves to predicates over the same alphabet, all theorems for conjunctive healthiness conditions proved in [5] continue to hold. This is because in that case, we can treat $\gamma_{\mathrm{HM}}(\alpha\, P)$ as a constant. We thus have closure under sequential composition, too, proved by factoring $\gamma_{\mathrm{HM}}(a)$ into orthogonal constraints on undashed and dashed variables: $\gamma_{\mathrm{HM}}(\mathit{in}\, a) \wedge \gamma_{\mathrm{HM}}(\mathit{out}\, a)$. Requiring equal alphabets may nevertheless be a strong caveat, for instance, in the presence of local variable blocks that incur alphabet changes. The motivation for alphabet compatibility can also be understood as an attempt to weaken the assumptions of closure laws in our theory. Exploiting it further in order to discover laws with weaker assumptions is on-going research.

Lastly, also following from [5], the set of **HM**-healthy predicates over a fixed alphabet is a complete lattice, as it is the image of a monotonic and idempotent healthiness function [7]. Further properties detailed in [5] consider the interaction with designs; they also transfer to our work. We next examine how we use the theory to reason about programs.

Application example We first introduce a utility law that allows us to extract properties of specific method variables from an **HM**-healthy predicate. This law facilitates reasoning about methods and will also be used later on. To express it concisely, we extend the use of the α operator to apply to method variables also, where it yields the alphabet of the underlying procedure type.

Law 4. *Assume P is* **HM***-healthy and we have $\{\overline{m}_x, \overline{\overline{m}}_x\} \subseteq \alpha\, P$, $\alpha\, \overline{m}_x \subseteq \alpha\, P$, and $\alpha\, \overline{\overline{m}}_x \subseteq \alpha\, P$. Then, $P = P \wedge (\mathbf{call}\, \overline{m}_x \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}_x)$.*

Proof.

$P \equiv$ *"P is* **HM***-healthy"*

$\quad \mathbf{HM}(P)$

$\equiv$ *"unfolding definition of* **HM***"*

$\quad P \wedge (\forall\, \overline{m}\ \overline{\overline{m}} \mid \{\overline{m}, \overline{\overline{m}}\} \subseteq \alpha\, P \bullet [\mathbf{call}\, \overline{m} \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}]_0)$

$\equiv$ *"specialisation of quantification with $\overline{m}_x$ and $\overline{\overline{m}}_x$"*

$\quad P \wedge \ldots \wedge [\mathbf{call}\, \overline{m}_x \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}_x]_0$

$\equiv$ *"specialisation of quantification (universal closure)"*

$\quad P \wedge \ldots \wedge (\mathbf{call}\, \overline{m}_x \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}_x)$

$\equiv$ *"logic and folding definition of* **HM***"*

$\quad \mathbf{HM}(P) \wedge (\mathbf{call}\, \overline{m}_x \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}_x)$

$\equiv$ *"P is* **HM***-healthy"*

$\quad P \wedge (\mathbf{call}\, \overline{m}_x \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}_x)$

Another useful law is a predicative version of the substitution rule.

Law 5. *$P[Q_1] \wedge (Q_1 \Leftrightarrow Q_2) = P[Q_2] \wedge (Q_1 \Leftrightarrow Q_2)$ where the notation $P[Q]$ expresses that the predicate Q occurs in another predicate P.*

Let us revisit S_1. We encode it in our theory as illustrated below.

$$T_1 \ \widehat{=}\ \mathbf{HM}(\overline{\overline{m}}_1 := \{\!|x := x + 1|\!\}\ ;\ \overline{\overline{m}}_2 := \{\!|x := x + 2\ ;\ \mathbf{call}\, \overline{m}_1|\!\})$$

We note that the assignments above are relational assignments rather than generalised higher-order assignments. The simple technical reason for this is to take advantage of the one-point rule; it is not a limitation of our theory.

The transformation below exemplifies how we reason about T_1.

$T_1 \equiv$ "unfolding definition of **HM**, let $\gamma^*_{\mathrm{HM}} \mathrel{\widehat{=}} \gamma_{\mathrm{HM}}(\{\overline{m}_1, \overline{m}_2, \overline{\overline{m}}_1, \overline{\overline{m}}_2\})$"

$\quad (\overline{\overline{m}}_1 := \{\!|x := x + 1|\!\}\ ;\ \overline{\overline{m}}_2 := \{\!|x := x + 2\ ;\ \mathbf{call}\, \overline{m}_1|\!\}) \wedge \gamma^*_{\mathrm{HM}}$

$\equiv$ "unfolding sequential compositions and assignments, one-point rule"

$\quad (\overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2\ ;\ \mathbf{call}\, \overline{m}'_1|\!\}) \wedge \gamma^*_{\mathrm{HM}}$

$\equiv$ "Law 4 with $(\overline{m}'_1, \overline{\overline{m}}'_1)$ and $(\overline{m}'_2, \overline{\overline{m}}'_2)$, predicate is **HM**-healthy"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2\ ;\ \mathbf{call}\, \overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\, \overline{m}'_1 \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}'_1) \wedge (\mathbf{call}\, \overline{m}'_2 \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}'_2) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "one-point rule using $\overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\}$ and $\overline{\overline{m}}'_2 = \{\!|x := x + 2\ ;\ \mathbf{call}\, \overline{m}'_1|\!\}$"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2\ ;\ \mathbf{call}\, \overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\, \overline{m}'_1 \Leftrightarrow \mathbf{call}\, \{\!|x := x + 1|\!\}) \wedge \\ (\mathbf{call}\, \overline{m}'_2 \Leftrightarrow \mathbf{call}\, \{\!|x := x + 2\ ;\ \mathbf{call}\, \overline{m}'_1|\!\}) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "cancellation law: $\mathbf{call}\,\{\!|p|\!\} = p$"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x+1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x+2\,;\ \mathbf{call}\,\overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1) \wedge \\ (\mathbf{call}\,\overline{m}'_2 \Leftrightarrow (x := x+2\,;\ \mathbf{call}\,\overline{m}'_1)) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "Law 5 using $\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1$ "

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x+1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x+2\,;\ \mathbf{call}\,\overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1) \wedge \\ (\mathbf{call}\,\overline{m}'_2 \Leftrightarrow (x := x+2\,;\ x := x+1)) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "simplification of sequence: $(x := x+2\,;\ x := x+1) = (x := x+3)$"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x+1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x+2\,;\ \mathbf{call}\,\overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1) \wedge \\ (\mathbf{call}\,\overline{m}'_2 \Leftrightarrow x := x+3) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

The last step makes precise the effect of calling the methods $\overline{m}_1$ and $\overline{m}_2$. It agrees with our intuition and moreover shows the validity of the copy rule.

To summarise, in this section we have presented a novel theory of methods that deals with the issues raised when using higher-order UTP to model object-oriented software. For instance, it allows us to redefine $\overline{\overline{m}}_1$ in T_1 as follows.

$$T_2 \ \widehat{=}\ T_1\,;\ \mathbf{HM}(\overline{\overline{m}}_1 := \{\!|(x := x+1\,;\ \mathbf{call}\,\overline{m}_1) \triangleleft x < 10 \triangleright \mathbf{II}|\!\})$$

where $\mathbf{II}$ has a suitable alphabet. This introduces a call into the method body. The types of $\overline{\overline{m}}_1$ is exactly as before, assuming that $\overline{\overline{m}}_1$ and $\overline{\overline{m}}_2$ *a priori* have $\overline{m}_1$ and $\overline{m}_2$ in their alphabets; this ceases to be a problem in our theory as it does not constrain calls. Thus compositionality of method (re)definition is restored.

5 Discussion

In this section we first discuss the treatment of procedures with parameters and secondly tackle the problem of (mutual) recursion in our theory of methods.

5.1 Procedures with Parameters

In our treatment so far, we have ignored the possibility of program values being procedures — that is having parameters. A standard approach to support procedures is to encode them as functions [2] whose domains correspond to the kind of objects being passed to the procedure (a variable or value) and whose range is the underlying semantic model of the procedure body. Higher-order UTP adopts a similar approach to realise parameter passing. For instance, we can define a procedure p_1 with a name parameter n as follows.

$$p_1 := \{\!|\, \lambda\, n : var(\mathbb{Z}) \bullet n := n + x|\!\}$$

It takes an integer variable as its argument and adds the value of x to it. It is a function from variables to predicates. This procedure is besides 'polymorphic'

in the sense that the alphabet of the predicate that results from applying p_1 is determined by the argument. For instance, we have that the alphabet of the predicate resulting from the call $p_1(x)$ includes $\{x, x', y, y'\}$ whereas the call $p_1(z)$ gives rise to a predicate whose alphabet is $\{y, y', z, z'\}$.

The polymorphic nature of procedures is difficult to reconcile with the model construction in Section 3.1. This is because the notion of type that is used there and taken from [7] is not appropriate anymore, precisely because no *a priori* knowledge of the alphabet of a procedure's predicate is possible. Because of this, we confine ourselves to procedures that are non-polymorphic. They are the procedures that only admit value parameters. An example is given below.

$$p_2 := \{\!|\, \lambda\, v : val(\mathbb{Z}) \bullet x := x + v \,|\!\}$$

We note though that the absence of result parameters does not prohibit or constrain the use of object references (pointers) [3]. Java, for instance, only includes value parameters. To integrate these kinds of procedures into our higher-order program model, we can, in essence, use the same technique as in Section 3.2. This is by introducing additional syntax that corresponds to the declaration of formal procedure parameters. Once again, a data type is used for this purpose.

$$PROC[BODY] ::= ValArg\ \langle\!\langle TYPE \times PROC[BODY] \rangle\!\rangle \mid Body\ \langle\!\langle BODY \rangle\!\rangle$$

Above, $TYPE$ encodes the type of a parameter; we assume this is $<$*base type*$>$. The recursion in *ValArg* enables us to support procedures with arbitrary numbers of parameters, as an object of a unified type $PROC[BODY]$ where $BODY$ provides the semantic model of the procedure body. We note that in the theory of object-orientation in [14], $BODY$ is itself syntax, which is not a problem.

Importantly, a new definition of **call**, refinement and at least assignment (to support refinement of procedure values) have to be provided for $PROC[BODY]$. These definitions take advantage of a function *apply* that applies a procedure to a list of arguments; its signature is illustrated below.

$$apply : PROC[BODY] \rightarrow \mathrm{seq}(VALUE) \rightarrow BODY$$

The *apply* function has a simple inductive definition which we omit. Refinement is defined as a pointwise extension of $\sqsubseteq_{body}$, the refinement of objects of type $BODY$. It only considers argument sequences of the correct length and type, which is determined by an auxiliary function *valid*.

$$p_1 \sqsubseteq_{proc} p_2 = \forall\, args \mid valid(p_1, p_2, args) \bullet (apply\ p_1\ args) \sqsubseteq_{body} (apply\ p_2\ args)$$

The new call operation **pcall** is defined as $\mathbf{pcall}\, p(args) \mathrel{\widehat{=}} \mathbf{call}(apply\ p\ args)$ where **call** provides the semantics of calls on entities of type $BODY$, which we assume already exists. If needed, other operators can be provided via pointwise lifting too, using the same approach as in [2].

The above shows how we can integrate limited support for parametrised procedures into our model without compromising soundness. Its primary limitation is that it excludes result parameters. Result parameters are, it seems, needed in order to support methods with return values. This is an open issue that we are currently investigating and planning to report on in follow-up work.

5.2 Mutual Recursion

We return now to the problem in the introduction of encoding

$$S \;\widehat{=}\; m_1, m_2 := \{\!|\, \mu\, X,\, Y \bullet \left\langle \begin{array}{l} (x := x - 1\,;\;\; Y) \lhd x > 0 \rhd \mathbf{II}, \\ (y := y - 1\,;\;\; X) \lhd y > 0 \rhd \mathbf{II} \end{array} \right\rangle |\!\}$$

In our theory of methods, this can now be written as

$$T_1 \;\widehat{=}\; \mathbf{HM}(\overline{\overline{m}}_1 :=_A \{\!|(x := x - 1\,;\;\; \mathbf{call}\, \overline{m}_2) \lhd x > 0 \rhd \mathbf{II}|\!\}) \quad \text{and}$$

$$T_2 \;\widehat{=}\; \mathbf{HM}(\overline{\overline{m}}_2 :=_A \{\!|(y := y - 1\,;\;\; \mathbf{call}\, \overline{m}_1) \lhd y > 0 \rhd \mathbf{II}|\!\}) \quad \text{and}$$

$$T \;\widehat{=}\; T_1\,;\;\; T_2$$

where $A \mathrel{\widehat{=}} \{\overline{m}_1, \overline{m}'_1, \overline{m}_2, \overline{m}'_2, \overline{\overline{m}}_1, \overline{\overline{m}}'_1, \overline{\overline{m}}_2, \overline{\overline{m}}'_2\}$ and

$$\alpha\, \overline{\overline{m}}_1 = \alpha\, \overline{\overline{m}}'_1 = \alpha\, \overline{\overline{m}}_2 = \alpha\, \overline{\overline{m}}'_2 = \{\overline{m}_1, \overline{m}'_1, \overline{\overline{m}}_2, \overline{\overline{m}}'_2\}$$

We observe that T_1 introduces the method definition for m_1 and T_2 introduces the method definition for m_2. Neither of them relies on a fixed-point construction, and compositionality is illustrated by the combined definition T that composes the individual method definitions in sequence. For composability, the alphabets of the assignments have to be suitably extended with A.

Although this is not proved here, we claim that

$$S \Leftrightarrow (\exists\, \overline{\overline{m}}_1, \overline{\overline{m}}'_1, \overline{\overline{m}}_2, \overline{\overline{m}}'_2 \bullet T)$$

A proof of this conjecture requires special laws that permit one to move between formulations in terms of recursive calls to rank 1 method variables and fixed points; we are currently examining those laws. It seems that in order to reason about particular programs, the form in S may have practical advantages. However, to reason about features of object-orientation, the form in T is superior because there we profit from compositional method (re)definition.

Above we introduced an alphabet A that contains all method variables under consideration. In practice, it is necessary to fix such an alphabet since otherwise, we still run into the problems discussed in Section 4.1 regarding the types of method variables. We recapture that in the theory of methods, there is, however, no problem in fixing this alphabet as this *per se* does not restrict calls. The fixing of alphabets in general involves the provision of a predefined repository of method variables in which the rank 2 variables have all rank 1 variables in their alphabets; we believe that this is largely a technical (and tractable) issue.

Finally, it is even possible to redefine recursive methods individually. For instance, we may redefine $\overline{\overline{m}}_1$ in T as follows.

$$T\,;\;\; \mathbf{HM}(\overline{\overline{m}}_1 :=_A \{\!|\, \mathbf{call}\, \overline{m}_2 |\!\})$$

Importantly, this redefinition implicitly also alters the behaviour of $\overline{\overline{m}}_2$, which now leaves x unaffected and sets y to 0. It appears that the theory of methods solves the problem of redefinition gracefully also in the context of (mutual) recursion. This may be at the cost of a possibly more complicated strategy for reasoning about the properties of methods, such as proving that the above specification implies that $[\mathbf{call}\, \overline{m}_2 \Leftrightarrow y := 0]$. We are currently investigating this.

6 Conclusion

We have examined the ramifications of higher-order UTP in theories of object-orientation and presented solutions to four major challenges: consistency of the program model, redefinition of methods in subclasses, the treatment of recursion and mutual recursion, and simplicity. We briefly comment on each of them.

Consistency is achieved by the inductive construction of a program model that caters for our needs to combine syntax and semantics, as well as procedures with parameters. We thereby proved a result that was left implicit in [7], namely that arbitrary theories can be used in place of the program model. The construction also provides guidance for mechanisation in a theorem prover. There are still open issues with regards to supporting result parameters; it seems that in order to do so, we have to elaborate the notion of variable type to reflect the signature of polymorphic procedures. This is on-going research work.

A number of issues that arise from method (re)definition have been discussed and we have presented a novel solution in terms of a UTP theory. The important contribution of the theory is to restore compositionality. Almost as a side effect, it also gracefully handles recursive definitions in a compositional manner. Notably, this is useful for the theory of object-orientation in [14], as it eliminates the need to rewrite recursive methods into multi-variable fixed-point terms.

Simplicity is achieved as our theory of methods provides a uniform treatment of types: method definitions are assignments to rank 2 variables while method calls are to rank 1 variables. The only complication that persists is that we have to introduce an *a priori* repository of method variables which determines the minimal alphabet of all rank 2 method variables (the set A in Section 5.2).

As related work, we first note Naumann's foundational work on the semantics of higher-order imperative programming [9,10]. It is based on predicate transformers and tackles features of object-oriented programs, such as inheritance and dynamic binding through the use of record subtyping. In [6], Jifeng et al. introduce rCOS, a UTP-based refinement calculus for object systems. It is based on a fixed syntax and defines the semantics of an object-oriented program by way of a denotation function; this seems to side-step the explicit use of procedure variables, although the treatment of recursion is not discussed.

Recent work by Chin et al. [4] proposes a modular verification technique for object-oriented programs based on separation logic. Their approach seems efficient and pragmatic, but is tied to a design-based view of method specifications. Our aim is to create a framework that can be integrated with arbitrary theories of programming. Lessons may be learned from [4] in terms of modular reasoning.

There are two main strands for future work. The first one is to formulate and prove more laws and properties of the theory of methods, and show how they are used in practice to reason about object-oriented programs. We expect there exist further interesting laws waiting for discovery, in particular in conjunction with fixed points and the theory of designs.

A second strand is the mechanisation of higher-order UTP as well as the theory in [14]. We already have preliminary but promising results on such a mechanisation in the Isabelle HOL prover [11]; it extends the semantic model of

alphabetised predicates that was used in [13] and [18] to incorporate program values. A delicate open issue is that presently we rely on custom axioms for the type morphism $\{\!|\_|\!\}$ and its inverse; future work will aim to remove those axioms.

Acknowledgements We thank the anonymous reviewers for their useful comments. This work was funded by the EPSRC grant EP/H017461/1.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
2. R.-J. Back and V. Preoteasa. Reasoning About Recursive Procedures with Parameters. In *Proceedings of the 2003 ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*. ACM, August 2003.
3. A. Cavalcanti, A. Wellings, and J. Woodcock. The Safety-Critical Java Memory Model: A Formal Account. In *FM 2011: Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 246–261. Springer, June 2011.
4. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Enhancing Modular OO Verification with Separation Logic. *ACM SIGPLAN Not.*, 43(1):87–99, January 2008.
5. W. Harwood, A. Cavalcanti, and J. Woodcock. A Theory of Pointers for the UTP. In *Theoretical Aspects of Computing - ICTAC 2008*, volume 5160 of *Lecture Notes in Computer Science*, pages 141–155. Springer, September 2008.
6. Jifeng He, Xiaoshan Li, and Zhiming Liu. rCOS: A refinement calculus for object systems. *Theoretical Computer Science*, 365(1-2):109–142, November 2006.
7. C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice Hall Series in Computer Science. Prentice Hall, February 1998.
8. I. Kassios. Decoupling in Object Orientation. In *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 632–632. Springer, July 2005.
9. D. Naumann. Predicate Transformer Semantics of an Oberon-Like Language. In *PROCOMET '94, Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi*, pages 467–487, 1994.
10. D. Naumann. Predicate transformers and higher-order programs. *Theoretical Computer Science*, 150(1):111–159, October 1995.
11. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
12. G. Nuka and J. Woodcock. Mechanising a Unifying Theory. In *Unifying Theories of Programming, First International Symposium*, volume 4010 of *Lecture Notes in Computer Science*, pages 217–235. Springer, February 2006.
13. M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying Theories in ProofPower-Z. In *Unifying Theories of Programming, First International Symposium*, volume 4010 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2006.
14. T. Santos, A. Cavalcanti, and A. Sampaio. Object-Orientation in the UTP. In *Unifying Theories of Programming*, volume 4010 of *Lecture Notes in Computer Science*, pages 18–37. Springer, February 2006.
15. M. Spivey. The Consistency Theorem for Free Type Definitions in Z. *Formal Aspects of Computing*, 8:369–375, May 1996.
16. J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*. International Series in Computer Science. Prentice Hall, July 1996.
17. F. Zeyda. A Theory of Methods: Validation of Laws. Technical report, July 2012. Available from `http://www.cs.york.ac.uk/circus/hijac/publication.html`.
18. F. Zeyda and A. Cavalcanti. Mechanical reasoning about families of UTP theories. *Science of Computer Programming*, 77(4):444–479, April 2012.