An Algebraic Framework for Higher-Order Modules

Rosa Jiménez and Fernando Orejas

Dept. Leng. Sist. Inf., Univ. Polit. Catalunya
Barcelona, SPAIN

Abstract. This paper presents a new framework for dealing with higher-order parameterization allowing the use of arbitrary fitting morphisms for parameter passing. In particular, we define a category of higher-order parameterized or module specifications and, then, following the approach started in the ASL specification language, we define a typed λ -calculus, as a formalism for dealing with these specifications, where arbitrary fitting morphisms are allowed. In addition, the approach presented is quite general since all the work is independent of the kind of basic specifications considered and, also, of the kind of operations used for building basic specifications, provided that some conditions hold. In this sense we are not especially bound to any set of basic specification-building operations. We call our parameterized units *modules* to make clear the distinction between the basic specification level that is not fixed a priori and the parameterized units level that is studied in the paper. The kind of calculus presented can be seen as a variation/extension of the simply typed λ -calculus, which means that we do not allow dependent types. This would have been interesting, but it is not possible with the semantics proposed. The main result of the paper shows the adequacy of β -reduction with respect to the semantics given.

1 Introduction

There are two standard approaches for dealing with genericity in specification or programming languages. On the one hand, parametric polymorphism is used, especially, in functional languages like ML or Miranda. On the other, different forms of generic units or modules are used in specification languages like Clear or Act One or programming languages like OBJ or Ada. Each of these approaches has a number advantages and disadvantages of different kind. Actually, some languages like ML provide constructions for both kind of approaches. Among the advantages in favor of generic modules is the additional power provided by allowing modules to have “complex” formal parameters, i.e. parameters need not to be just sorts but may be arbitrary interfaces. In addition, module instantiation is usually defined in a flexible way by means of an arbitrary fitting morphism identifying the “elements” (sorts, functions, predicates,...) in the interface with corresponding elements in the actual parameter. Among the advantages in favour of parametric polymorphism is the additional power provided by allowing to deal with higher-order objects.

There have been two approaches to provide a higher-order extension of the basic constructions of some specification languages. One approach, consists in extending the logic underlying a given specification language with higher-order sorts (for instance extending first-order equational logic to higher-order equational logic). In this way, the structuring constructions of a specification language remain the same, only the underlying formalism changes. This approach has some disadvantages. The main one is that having a more complex underlying formalism implies a difficulty for building deductive tools. For instance, when going from first-order to higher-order equational logic, unification becomes undecidable. The other approach consists in (without changing the underlying formalism) allowing the interfaces of generic units to be generic or parameterized themselves. A simple way of doing this, which has its origins in the specification language ASL [13], is based on considering that a generic specification is any parameterized λ -expression built over some specification variables. Then, instantiating a parameterization consists in substituting these variables by any (adequate) specification. This can be seen as a form of β -reduction. One of the advantages of this approach, besides its generality, is the simplicity for defining higher-order parameterizations. In particular, given the parameterization:

$$MSP = \lambda X : SP.E(X)$$

where $E(X)$ denotes any expression defined in terms of the specification building operations considered, over a “specification variable” X of “type” (formal parameter specification) SP . If we allow SP or $E(X)$ to be arbitrary λ -expressions then MSP would be a higher-order specification. This kind of typed λ -notation may look a bit awkward since types and objects seem to denote the same thing (specifications). However, the type of a variable and the result of an expression denote different things. Being specific, when declaring $X : SP$, it is considered that the type denoted by SP is the class (or category) of all the specifications that are “more concrete” than SP , i.e. all the admissible actual parameters for SP . Let us see an example of a higher-order parameterization using this approach. Given the specifications:

$ELEM = \text{sorts } elem$

$PSEQ = \lambda X : ELEM. \text{enrich } X \text{ by}$

sorts seq

opns $\epsilon : \rightarrow seq$

$app : seq \ elem \rightarrow seq$

defining a parameterized specification of “linear” structures of a given type of elements, the following specifications :

$ELEM1 = \text{enrich } BOOL \text{ by}$

sorts $elem$

opns $p : elem \rightarrow bool$

$PFILTER = \lambda X : ELEM1.$

$\lambda Y : PSEQ.enrich\ Y[X]_{id}\ \mathbf{by}$
 $\quad \mathbf{opns}\ filter : seq \rightarrow seq$
 $\quad \mathbf{vars}\ S : seq; V : val$
 $\quad \mathbf{axms}\ filter(\epsilon) = \epsilon$
 $\quad p(V) = true \Rightarrow filter(app(S, V)) = app(filter(S), V)$
 $\quad p(V) = false \Rightarrow filter(app(S, V)) = filter(S)$

define a specification describing how to "filter" the elements of a linear structure satisfying a given property p . Previous work following this approach can be found in [12, 4]. However, in both works this extension is done without allowing the use of arbitrary fitting morphisms in module instantiation. One of the problems with using arbitrary fitting morphisms in this setting is that parameter passing cannot just work as standard β -reduction. The reason can be seen in the following example. Suppose that we have the specification:

$\lambda X : SP.rename\ X\ \mathbf{by}\ h$

This specification is considered correct if h is a morphism from the signature of the specification SP into another signature. Now suppose that we apply this parameterization to a specification $SP1$. The result would be:

$rename\ SP1\ \mathbf{by}\ h$

but, if we allow the use of arbitrary fitting morphisms for instantiation, it may happen that the expression is now incorrect because the signatures of $SP1$ and $SP1$ can be different. A second problem is that, if one wants to allow instantiations through arbitrary morphisms one would have to define the adequate notion of morphism between higher-order specifications (and, consequently, the right category of higher-order specifications). In this sense, in [12] actual and formal parameter specifications must share the same signature, which is quite restrictive. Some more flexibility is given in [4], where inclusions of signatures are allowed. However this is done using a quite more complex framework.

The work presented in this paper provides a solution to this problem. Using as a basis [12] and previous work of the authors for the first-order case [10], we define a category of higher-order parameterized or module specifications and, then, we define a typed λ -calculus, as a formalism for dealing with these specifications, where arbitrary fitting morphisms are allowed. In particular, in [12], a detailed study of different semantic issues concerning parameterized units is presented showing, essentially, that two possible meanings can be assigned to parameterizations depending on whether we consider them as parameterized specifications or as specifications of parameterized programs. In this sense, in this paper we consider just the former meaning, although the results could be adapted to deal also with the latter case.

The approach presented is quite general since all the work is independent of the kind of basic specifications considered and, also, of the kind of operations

used for building basic specifications, provided that some conditions hold. In this sense we are not especially bound to any set of basic specification-building operations. We call our parameterized units *modules* to make clear the distinction between the basic specification level that is not fixed a priori and the parameterized units level that is studied in the paper. The kind of calculus presented can be seen as a variation/extension of the simply typed λ -calculus, which means that we do not allow dependent types [11]. This would have been interesting, but it is not possible with the semantics proposed. The main result of the paper shows the adequacy of β -reduction with respect to the semantics given. However, it must be pointed out that this is not done for any arbitrary λ -expression but only for expressions in η -long normal form. We do not think that this is a serious limitation, since similar restrictions can be found in related frameworks.

The paper is organized as follows: in section 2, we describe the requirements assumed on the basic specification language. Section 3 defines the category of higher-order modules and a parameter passing operation for the objects in this category. Finally, in section 4 we present the language of λ -expressions, we define β -reduction and we obtain the main results.

2 Preliminaries

In this section we briefly present the abstract framework that we will use along the paper to present all constructions and results. We assume on the reader certain knowledge on the basic algebraic techniques used in the semantics of specification languages. For details see, for instance, [1, 7, 15]. Also, we assume some very basic knowledge on the λ -calculus. For details, see for example [2]. We do not assume much about how basic specifications look like. We just assume that specifications form a category, that we call *Spec*, satisfying certain properties and that we have a specification language to build them. This means that specifications may consist of a signature and a set of axioms [7], or of a signature and a set of models [13, 14] or they may be of any other form. In particular, we do not need to bind them to an arbitrary institution or specification frame [8, 9, 10]. The reason is that, in this work, we are not especially interested with semantical aspects of basic specifications. The only assumption about *Spec* is that it has multiple pushouts. Multiple pushouts were defined in [10] to define parameter passing for arbitrary parameterized expressions, as a slight generalization of the use of pushouts when dealing with parameterizations defined in terms of generic enrichments [3, 5, 6]. The difference with standard pushouts is that two of the arrows of a standard pushout are here a family of morphisms (which we call a multiple morphism). In the following section we will see a couple of examples that show the need of multiple morphisms in this setting.

Definition 1. *A multiple morphism in Spec F from SP_0 to SP_1 , denoted $F : SP_0 \Rightarrow SP_1$, is a family of morphisms in Spec, $F = \langle f_1, \dots, f_n : SP_0 \rightarrow SP_1, (n \geq 0) \rangle$. Multiple morphisms can be composed with standard morphisms: If F is defined as above and $f : SP_1 \rightarrow SP_2$ then $f \circ F : SP_0 \Rightarrow SP_2$ is the*

family $\langle f \circ f_1, \dots, f \circ f_n : SP0 \rightarrow SP1, (n \geq 0) \rangle$. Similarly, if $f : SP2 \rightarrow SP0$ then $F \circ f = \langle f_1 \circ f, \dots, f_n \circ f : SP0 \rightarrow SP1, (n \geq 0) \rangle$. Two additional operations that one can define on multiple morphisms are composition and addition. If $F' = \langle f_1', \dots, f_m' : SP1 \rightarrow SP2 \rangle$ then

$$F \circ F' = \langle f_1 \circ f_1', \dots, f_1 \circ f_m', \dots, f_n \circ f_m' : SP0 \rightarrow SP2 \rangle$$

and if $F' = \langle f_1', \dots, f_m' : SP0 \rightarrow SP1 \rangle$ then

$$F' + F = \langle f_1, \dots, f_n, f_1', \dots, f_m' : SP0 \rightarrow SP1 \rangle$$

Fig. 1. Multiple pushout

Given a multiple morphism $F : SP0 \Rightarrow SP1$ as above and given $f : SP0 \rightarrow SP2$ in Spec, the diagram in Figure 1 is called a multiple pushout of F and f if we have:

1. (Graded Commutativity): $g \circ f_i = g_i \circ f$ (for every $i \leq n$)
2. (Universal Property): For each object $SP3'$ and morphisms $g', g_1, \dots, g_n$ with $g' \circ f_i = g_i' \circ f$ (for every $i \leq n$) there is a unique morphism h such that $h \circ g = g'$ and $h \circ g_i = g_i'$ (for every $i \leq n$)

Remarks 2

1. In the case $n = 1$ a multiple pushout is a pushout. In the case $n = 0$ $SP3$ is equal (up to isomorphism) to $SP1$ with $g = 1_{SP1}$.
2. Multiple pushouts can be defined in terms of pushouts and (finite) coproducts or, more generally, finite colimits.

Example 3. Consider the following specifications and morphisms:

$$\begin{array}{ll} SP0 = \mathbf{sorts} \ s0 & SP1 = \mathbf{sorts} \ s1, s2 \\ & \mathbf{opns} \ f1 : s1 \rightarrow s1 \\ & \quad f2 : s2 \rightarrow s2 \end{array}$$

$$\begin{array}{ll} h1 : SP0 \rightarrow SP1 & h2 : SP0 \rightarrow SP1 \\ \text{with } h1(s0) = s1 & \text{with } h2(s0) = s2 \end{array}$$

Fig. 2. Universal property of multiple pushouts

Now, if we define:

```

SP2 = sorts s0
      opns a : → s0
           c : s0 → s0
    
```

together with the inclusion $i0 : SP0 \rightarrow SP2$, we can "compute" the multiple pushout shown in Figure 3, where $SP3$ would be (up to isomorphism) the specification:

Fig. 3. Multiple pushout

```

SP3 = sorts s1, s2
      opns f1 : s1 → s1
           f2 : s2 → s2
           a1 : → s1
           c1 : s1 → s1
           a2 : → s2
           c2 : s2 → s2
    
```

and the two morphisms, $h1'$ and $h2'$, would map, respectively, s, a and c to $s1, a1$ and $c1$ and to $s2, a2$ and $c2$.

In addition, we assume that we have a language for building specifications. We may think of this language as a many-sorted signature including, at least, a sort of specifications $SPEC$ and a sort of morphisms $MORPH$ and, as constants of these sorts, all objects and morphisms in $\underline{Spec}$, respectively (or just some distinguished subsets). We will call this signature $\Sigma SPEC$. For example, an operation in this signature may be:

$$combine : SPEC \times SPEC \times SPEC \times MORPH \times MORPH \rightarrow SPEC$$

that given specifications $SP0, SP1$ and $SP2$ and morphisms $h1 : SP0 \rightarrow SP1$ and $h2 : SP0 \rightarrow SP2$ is intended to combine the specifications $SP1$ and $SP2$, having a common subspecification $SP0$, where $h1$ and $h2$ make explicit how $SP0$ is included in $SP1$ and $SP2$, respectively. Typically, this combination would be done by means of a pushout.

Hence, $T_{\Sigma SPEC}$ is the class of specification expressions (syntactically) valid in the given specification language. It may be noted that not every term in $T_{\Sigma SPEC}$ may denote a semantically valid construction and, therefore, have a meaning. For instance, $combine(SP0, SP1, SP2, h1, h2)$ would be semantically incorrect if, for instance, $h1$ is not a morphism from $SP0$ into $SP1$. Anyhow, we assume given a semantics for the specification language as a (partial) function, called Sem , mapping (valid) terms in $T_{\Sigma SPEC}$ of sort $SPEC$ (respectively $MORPH$) into objects (respectively morphisms) in $\underline{Spec}$.

We assume that operations in $\Sigma SPEC$ are incremental, in the sense that given a variable X of sort $SPEC$ and given a specification term t in $T_{\Sigma SPEC}(\{X\})$ then for any specification SP such that $t[SP/X]$ is valid (i.e. $Sem(t[SP/X])$ is defined) we consider that there is a multiple specification morphism in $\underline{Spec}$ associated to t and SP , denoted $t_{SP} : SP \Rightarrow t([SP/X])$, such that if $h : SP \rightarrow SP'$ and $t([SP'/X])$ is valid then the diagram in Figure 4 is a multiple pushout.

$$\begin{array}{ccc}
 SP & \xrightarrow{t_{SP}} & Sem(t[SP/X]) \\
 \downarrow h & \parallel & \downarrow \\
 SP' & \xrightarrow{t_{SP'}} & Sem(t[SP'/X])
 \end{array}$$

Fig. 4. Incrementality

This means that we are assuming that for every operation op in $\Sigma SPEC$ there is also a morphism between any possible argument for op of sort $SPEC$

and the resulting specification. As a consequence, operations such as *derive* would not be allowed to be in $\Sigma SPEC$. The reason for this, besides some technical questions, is that we consider that hiding should be considered at the module level. Anyhow, this is not considered in this paper. It could have been done by defining modules with an additional export interface. We believe that the main results would still hold, but at the price of an additional complication.

We also consider that there exists an operation *assign* that, given t in $T_{\Sigma SPEC}(\{X\} \cup V)$ of sort $SPEC$, where V is a (finite) set of variables, two specifications $SP1$ and $SP2$ and a morphism $h : SP1 \rightarrow SP2$, it returns a new term, denoted $assign(t, h)$ such that for every assignment $\sigma : V \rightarrow \underline{Spec}$ such that $\sigma^*(t[SP1/X])$ is valid and the diagram in Figure 5 is a multiple pushout, where $\sigma^*(t)$ denotes the term obtained after substituting in t all its variables by the values assigned by σ . Moreover, if $t[SP2/X]$ is valid then $t[SP2/X] = assign(t, h)$.

$$\begin{array}{ccc}
 SP1 & \xrightarrow{\sigma^*(t)_{SP1}} & Sem(\sigma^*(t[SP1/X])) \\
 \downarrow h & & \downarrow \\
 SP2 & \xrightarrow{\quad\quad\quad} & Sem(\sigma^*(assign(t, h)))
 \end{array}$$

Fig. 5. The operation *assign*

3 The Category of Module Specifications $\underline{MSpec}$

In this section, we introduce $\underline{MSpec}$ a category of module specifications that will be used for giving semantics to module expressions. In particular, this category includes, as a full subcategory, the category of basic specifications, $\underline{Spec}$.

The objects in $\underline{MSpec}$ are either basic specifications, i.e. objects in $\underline{Spec}$, or triples $\langle IMP, RES, F \rangle$ denoting parameterized objects. The intuition is that IMP and RES , which are objects in $\underline{MSpec}$, are, respectively, the imported and the result specification of a module and F is a kind of mapping binding the “elements” (e.g. sorts and operations) in IMP with corresponding elements in RES . For instance, a standard case is when IMP and RES are basic specifications and RES is an enrichment of IMP , i.e. F would be a specification morphism in $\underline{Spec}$. Similarly, when considering modules defining first-order parameterized specifications defined by arbitrary specification expressions, as in [10], F is a multiple specification morphism, $F : IMP \Rightarrow RES$. This is needed, for instance, when dealing with expressions such as:

$$\lambda X : SP.X \sqcup X$$

where $SP1 \sqcup SP2$ denotes the coproduct of $SP1$ and $SP2$. In this case, the relation between the formal parameter and the result specification can be denoted by two morphisms binding the formal parameter SP with the two copies included in $SP \sqcup SP$, as shown in Figure 6. Also, given the parameterization $\lambda X : SP.SP'$,

$$\begin{array}{ccc} SP & \xrightarrow{\quad h1 \quad} & SP \sqcup SP \\ & \xrightarrow{\quad h2 \quad} & \end{array}$$

Fig. 6. “Double” morphism

where SP' is a constant specification, then the relation between the formal parameter and the result specification can be denoted by the empty family of morphisms. In the general case, when IMP or RES may be higher-order, the relation between their elements can be established by a multiple morphism between their “ground versions”. For instance, consider the module specification $PFILTER$ given in the introduction, its import specification $ELEM1$ is not first-order but its result is the specification denoted by the expression

```

λY : PSEQ.enrich Y[X]id by
  opns filter : seq → seq
  vars  S : seq; V : val
  axms filter(ε) = ε
        p(V) = true ⇒ filter(app(S, V)) = app(filter(S), V)
        p(V) = false ⇒ filter(app(S, V)) = filter(S)

```

Now, in order to bind the sorts and operations defined in $ELEM1$ with the sorts and operations in the result specification we may, first, define its associated ground specification as the specification denoted by the expression:

```

enrich PSEQ[ELEM1]id by
  opns filter : seq → seq
  vars  S : seq; V : val
  axms filter(ε) = ε
        p(V) = true ⇒ filter(app(S, V)) = app(filter(S), V)
        p(V) = false ⇒ filter(app(S, V)) = filter(S)

```

and, then, establish this binding in terms of a multiple specification morphism (in this case a standard morphism is enough).

To be more precise, we will define a functor $Body : \underline{MSpec} \rightarrow \underline{Spec}$ that associates to every module specification its ground version. Then F is a multi-

ple morphism between $Body(IMP)$ and $Body(RES)$, i.e. $F : Body(IMP) \Rightarrow Body(RES)$. This leads to the following inductive definition:

Definition 4. *The category $\underline{MSpec}$ and the functor $Body : \underline{MSpec} \rightarrow \underline{Spec}$ are defined by simultaneous induction as follows:*

- if SP is in $\underline{Spec}$ then SP is in $\underline{MSpec}$ and $Body(SP) = SP$
- if $h : SP1 \rightarrow SP2$ is in $\underline{Spec}$ then $h : SP1 \rightarrow SP2$ is in $\underline{MSpec}$ and $Body(h) = h$
- if $MSP1, MSP2 \in \underline{MSpec}$, and F is a multiple morphism in $\underline{Spec}$, $F : Body(MSP1) \Rightarrow Body(MSP2)$, then $MSP = \langle MSP1, MSP2, F \rangle$ is an object in $\underline{MSpec}$ and $Body(MSP) = Body(MSP2)$.
- if $h1 : MSP21 \rightarrow MSP11$ and $h2 : MSP12 \rightarrow MSP22$ are in $\underline{MSpec}$ and the diagram in Figure 7 commutes (has graded commutativity) in $\underline{MSpec}$:

$$\begin{array}{ccc}
 Body(MSP11) & \xRightarrow{F1} & Body(MSP12) \\
 \uparrow Body(h1) & & \downarrow Body(h2) \\
 Body(MSP21) & \xRightarrow{F2} & Body(MSP22)
 \end{array}$$

Fig. 7. Morphism in $\underline{MSpec}$

then $h = \langle h1, h2 \rangle : \langle MSP11, MSP12, F1 \rangle \rightarrow \langle MSP21, MSP22, F2 \rangle$ is a morphism in $\underline{MSpec}$ and $Body(h) = Body(h2)$.

The morphisms in $\underline{MSpec}$ are defined with the aim of using them as *parameter passing* or *fitting* morphisms. In this sense, morphisms between basic specifications are, as usual, standard specification morphisms. In the case of morphisms between parameterized objects, $\langle h1, h2 \rangle : MSP1 \rightarrow MSP2$, where $MSP1 = \langle IMP1, RES1, F1 \rangle$, $MSP2 = \langle IMP2, RES2, F2 \rangle$, the idea is, as usual, that $MSP2$ should be “less general” than $MSP1$. This means that the result of $MSP2$, $RES2$, should be less general than $RES1$ and that the formal parameter of $MSP2$, $IMP2$, should be more general than $IMP1$. In addition, $h1$ and $h2$ should be coherent with respect to the binding between the elements of $MSP1$ and $MSP2$. This is expressed by the commuting diagram above.

Remarks 5

1. $\underline{MSpec}$ and $Body$ can be proved to be a category and a functor by defining morphism composition in the most obvious way.
2. It may be noted that $\underline{MSpec}$ can not be shown to be a Cartesian Closed Category (CCC) because of the choice of morphisms in its definition. The

problem is that, as explained above, morphisms in $\underline{MSpec}$ have been defined with the aim of defining instantiation. A different choice would have been defining morphisms in $\underline{MSpec}$ as multiple morphisms in $\underline{Spec}$, i.e. if $F : \text{Body}(\text{IMP}) \Rightarrow \text{Body}(\text{RES})$ then $F : \text{MSP1} \rightarrow \text{MSP2}$ is a morphism in $\underline{MSpec}$. This notion of morphism would have allowed the identification between morphisms and functional objects which is the basis for the definition of a CCC. Anyhow, we would have still needed a notion of fitting morphism for parameter passing.

To end this section, we will show how parameter passing or module instantiation can be defined in $\underline{MSpec}$. Suppose that we have specifications $\text{MSP} = \langle \text{MSP1}, \text{MSP2}, F \rangle$ and MSP_{act} in $\underline{MSpec}$ and a morphism $h : \text{MSP1} \rightarrow \text{MSP}_{act}$ in $\text{Mor}(\underline{MSpec})$, then the result of *instantiating* MSP over MSP_{act} via h should be the specification obtained by “substituting” MSP1 by MSP_{act} in MSP2 , with respect to the bindings defined by h and F . Now, in general, MSP2 will have the form $\langle \text{MSP2}_1, \langle \text{MSP2}_2, \dots \langle \text{MSP2}_n, \text{MSP2}_{n+1}, F_n \rangle, F_{n-1} \rangle, \dots, F_1 \rangle$, where $\text{MSP2}_{n+1} = \text{Body}(\text{MSP2})$. Then, if we consider that the parameters $\text{MSP2}_1, \text{MSP2}_2, \dots, \text{MSP2}_n$, are independent of MSP1 , the result of the instantiation should be a specification

$$\text{MSP}_{res} = \langle \text{MSP2}_1, \langle \text{MSP2}_2, \dots \langle \text{MSP2}_n, \text{RES}, F'_n \rangle, F'_{n-1} \rangle, \dots, F'_1 \rangle$$

where RES is the result of substituting $\text{Body}(\text{MSP1})$ by $\text{Body}(\text{MSP}_{act})$ in MSP2_{n+1} , with respect to the bindings defined by $\text{Body}(h)$ and F . This means that RES can be defined in terms of the multiple pushout shown in Figure 8.

$$\begin{array}{ccc} \text{Body}(\text{MSP1}) & \xRightarrow{F} & \text{Body}(\text{MSP2}) \\ \downarrow \text{Body}(h) & & \downarrow f \\ \text{Body}(\text{MSP}_{act}) & \xRightarrow{F'} & \text{Body}(\text{MSP}_{res}) \end{array}$$

Fig. 8. Multiple Pushout

On the other hand, $F'_1, \dots, F'_n$ should bind the elements in $\text{MSP2}_1, \dots, \text{MSP2}_n$, respectively, to the corresponding elements in RES , i.e. $F'_i = f \circ F_i$. It may be noted that f can be extended into a specification morphism $h' : \text{MSP2} \rightarrow \text{RES}$ by just defining $h' = \langle \text{Id}_1, \langle \text{Id}_2, \dots \langle \text{Id}_n, f \rangle \dots \rangle \rangle$, where Id_i denotes the identity morphism, $\text{Id}_i : \text{MSP2}_i \rightarrow \text{MSP2}_i$. Actually, in what follows we consider h' as part of the result of the operation of instantiation. This means, more precisely:

Definition 6. Given specifications $\text{MSP} = \langle \text{MSP1}, \text{MSP2}, F \rangle$ and MSP_{act} in $\underline{MSpec}$ and a morphism $h : \text{MSP1} \rightarrow \text{MSP}_{act}$ in $\text{Mor}(\underline{MSpec})$, the re-

sult of the instantiation of MSP by MSP_{act} via h is a parameterized specification, denoted $MSP_{res} = MSP(MSP_{act})_h$, and a specification morphism $h' : MSP2 \rightarrow MSP_{res}$ defined inductively:

- If $MSP2$ is in Spec then MSP_{res} and h' are defined in terms of the multiple pushout shown in Figure 9.

$$\begin{array}{ccc}
 \text{Body}(MSP1) & \xRightarrow{F} & \text{Body}(MSP2) \\
 \downarrow \text{Body}(h) & & \downarrow \text{Body}(h') \\
 \text{Body}(MSP_{act}) & \xRightarrow{F'} & \text{Body}(MSP_{res})
 \end{array}$$

Fig. 9. Multiple Pushout

- If $MSP2 = \langle MSP2_1, MSP2_2, F2 \rangle$ then $h' = \langle Id : MSP2_1 \rightarrow MSP2_1, h'' \rangle$ and $MSP_{res} = \langle MSP2_1, MSP_{res}', \text{Body}(h'') \circ F2 \rangle$, where MSP_{res}' and h'' are the result of the instantiation of $\langle MSP1, MSP2_2, F \rangle$ by MSP_{act} via h .

This parameter passing construction has the following universal property, which means that h' is unique up to isomorphism:

Proposition 7. *Given $MSP, MSP_{act}, MSP_{res}, h$ and h' as in the previous definition and given a specification $MSP3$, a morphism $h3 : MSP2 \rightarrow MSP3$ and a multiple morphism $F3 : \text{Body}(MSP_{act}) \Rightarrow \text{Body}(MSP3)$ such that the diagram in Figure 10 commutes, there exist a unique $h3' : MSP_{res} \rightarrow MSP3$ such that $h3' \circ h' = h3$ and $\text{Body}(h3') \circ F' = F3$.*

$$\begin{array}{ccc}
 \text{Body}(MSP1) & \xRightarrow{F} & \text{Body}(MSP2) \\
 \downarrow \text{Body}(h) & & \downarrow \text{Body}(h3) \\
 \text{Body}(MSP_{act}) & \xRightarrow{F3} & \text{Body}(MSP3)
 \end{array}$$

Fig. 10. Multiple diagram

proof sketch The proposition is a consequence of the universal property of multiple pushouts and of the fact that all the component specifications involved as parameters in $MSP2$ and MSP_{res} coincide. $\square$

4 Parameterized Specification Expressions

In this section we define a concrete syntax for module specifications in terms of a language of λ -expressions and we define its semantics in terms of the category defined in the previous section. The class of all specification expressions will be called $\mathcal{SE}$. Since a λ -expression may involve specification morphisms we provide a specific syntax for them in terms of a class of morphism expressions, denoted $\mathcal{ME}$. Finally, we define and study a β -reduction relation between λ -expressions.

Definition 8. *The classes of specification expressions $\mathcal{SE}$ and morphism expressions $\mathcal{ME}$ are defined inductively:*

1. Variable : *any variable X is in $\mathcal{SE}$.*
2. Application : *$E1(E2)_M$ is in $\mathcal{SE}$ if $E1$ and $E2$ are in $\mathcal{SE}$ and M is in $\mathcal{ME}$.*
3. λ -abstraction : *$\lambda X : E1.E2[X]$ is in $\mathcal{SE}$ if $E2[X]$ is an expression over X in $\mathcal{SE}$ and $E1$ is an expression in $\mathcal{SE}$ not including any free variable.*
4. Basic specification expressions : *Any expression $E \in T_{\Sigma SPEC}(\langle \mathcal{SE}, \mathcal{ME} \rangle)$ of sort $SPEC$ is in $\mathcal{SE}$.*
5. Identities : *id_E is in $\mathcal{ME}$ if E is in $\mathcal{SE}$.*
6. Composition : *$M1 \circ M2$ is in $\mathcal{ME}$ if $M1$ and $M2$ are in $\mathcal{ME}$.*
7. Basic morphism expressions : *Any morphism expression $M \in T_{\Sigma SPEC}(\langle \mathcal{SE}, \mathcal{ME} \rangle)$ of sort $MORPH$ is in $\mathcal{ME}$.*
8. Higher-order morphisms : *$\langle M1, M2 \rangle$ is in $\mathcal{ME}$ if $M1$ and $M2$ are in $\mathcal{ME}$.*
9. Subcomponents : *$fst(M)$ and $snd(M)$ are in $\mathcal{ME}$ if M is in $\mathcal{ME}$.*
10. Substitution morphism : *$[E1/E2]_M$ is in $\mathcal{ME}$ if $E1$ and $E2$ are in $\mathcal{SE}$ and M is in $\mathcal{ME}$.*

Remarks 9

1. *The former definition describes the syntactically valid module expressions. However, as before, not every syntactically valid expression can be considered semantically well-formed. Instead, we could have given a definition, in terms of formation rules in order to ensure syntactic and semantic validity.*
2. *Rules 1,2 and 3 are standard for the simply typed λ -calculus. Rule 3 allows the use of expressions such as $\lambda X : E1.enrich X(E2)_M$ by..., where an operation in $\Sigma SPEC$ is used (e.g. an enrichment) over an arbitrary specification expression. As we will see below, this is considered semantically valid only if the parameters of such operation denote basic specifications.*
3. *Rules 5-10 define the morphism expressions considered. In particular, rules 5, 6, 7 and 8 define the most direct ways of denoting morphisms in $MSpec$ i.e. using morphisms from $Spec$ (rule 7), using a notation for the identities and for composition or building higher-order morphisms as pairs of lower-order ones. Rule 9 allows one to denote the components of a higher-order morphism. Finally, rule 10 defines a morphism that is defined when performing an operation of substitution and is needed for defining β -reduction.*

In order to cope with expressions that may include free variables we will define semantics with respect to a given environment, where an environment is, as usual, a set of declarations of the form $X : E$ where E is any closed specification expression. The semantics will only be defined for semantically valid expressions.

Definition 10. *Given a specification expression E and an environment Γ , the semantics of E in the environment Γ , denoted $\Gamma \vdash \llbracket E \rrbracket$ (if no specific environment is assumed, e.g. if E is closed, we will just write $\llbracket E \rrbracket$) is defined inductively:*

1. If $\Gamma \vdash \llbracket E \rrbracket = MSP$ then $\Gamma \cup \{X : E\} \vdash \llbracket X \rrbracket = MSP$
2. If $\Gamma \vdash \llbracket E1 \rrbracket = (IMP1, RES1, F1)$, $\Gamma \vdash \llbracket E2 \rrbracket = MSP2$ and $\Gamma \vdash \llbracket M \rrbracket = f$ is a morphism from $IMP1$ to $MSP2$ then

$$\Gamma \vdash \llbracket E1(E2)_M \rrbracket = (IMP1, RES1, F1)(MSP2)_f$$

3. If $\Gamma \vdash \llbracket E1 \rrbracket = MSP1$ and $\Gamma \cup \{X : E1\} \vdash \llbracket E2[X] \rrbracket = MSP2$ then

$$\Gamma \vdash \llbracket \lambda X : E1. E2[X] \rrbracket = \langle MSP1, MSP2, F : Body(MSP1) \Rightarrow Body(MSP2) \rangle$$

where F depends on the form of $E2[X]$. In particular, F is defined as follows:

- (a) If $E2[X] = X$, then $F = \langle Id_{Body(MSP1)} \rangle$.
- (b) If $E2[X] = E1'(E2')_M$ then $F = (F1'' \circ F2) + (g \circ F1)$, where $\Gamma \vdash \llbracket M \rrbracket = h$, $\Gamma \cup \{X : E1\} \vdash \llbracket E1' \rrbracket = MSP1' = \langle MSP11', MSP12', F1' \rangle$, $\Gamma \cup \{X : E1\} \vdash \llbracket E2' \rrbracket = MSP2'$, $\Gamma \vdash \llbracket \lambda X : E1. E1'[X] \rrbracket = \langle MSP1, MSP1', F1 \rangle$, $\Gamma \vdash \llbracket \lambda X : E1. E2'[X] \rrbracket = \langle MSP1, MSP2', F2 \rangle$ and $MSP2$ is the "result" of the instantiation defined by the multiple pushout in Figure 11.

$$\begin{array}{ccc}
 Body(MSP11') & \xRightarrow{F1'} & Body(MSP12') \\
 \downarrow Body(h) & & \downarrow g \\
 Body(MSP2') & \xRightarrow{F1''} & Body(MSP2)
 \end{array}$$

Fig. 11. Multiple Pushout

- (c) If $E2[X] = \lambda Y : E1'. E2'$ then F is the multiple morphism defined in $\Gamma \vdash \llbracket \lambda X : E1. E2'[E1'/Y] \rrbracket = \langle MSP1, MSP2', F \rangle$.
- (d) If $E2[X] = t[E1'/X1, \dots, En'/Xn, M1/Y1, \dots, Mm/Ym][X]$, where $t \in T_{SPEC}(\{X1, \dots, Xn, Y1, \dots, Ym\})$, let us assume that, for each i , $\Gamma \vdash \llbracket \lambda X : E1. Ei' \rrbracket = (MSP1, MSPi', Fi')$ and, for each j , $\Gamma \vdash \llbracket Mj \rrbracket = hj$. In addition, we know (see section 2) that for each i the multiple morphism

$$ti_{MSPi} : MSPi' \Rightarrow t[MSP1'/X1, \dots, MSPn'/Xn, h1/Y1, \dots, hm/Ym]$$

where

$$ti = t[MSP1'/X1, ..., MSPi-1'/Xi-1, MSPi+1'/Xi+1, ..., Ym/hm]$$

Then

$$F = t1_{MSP1} \circ F1' + ... + tn_{MSPn} \circ Fn'$$

4. If, for every i, j ($1 \leq i \leq n$ $1 \leq j \leq m$), $\Gamma \vdash \llbracket Ei \rrbracket = SPi$, $\Gamma \vdash \llbracket Mj \rrbracket = hj$, with SPi, hj in Spec and $t \in T_{\Sigma SPEC}(\langle \{X1, ..., Xn\}, \{Y1, ..., Ym\} \rangle)$ of sort SPEC then

$$\Gamma \vdash \llbracket t[E1/X1, ..., Ym/Mm] \rrbracket = Sem(t[SP1/X1, ..., hm/Ym])$$

5. If $\Gamma \vdash \llbracket E \rrbracket = MSP$ then $\Gamma \vdash \llbracket id_E \rrbracket = 1_{MSP}$.
 6. If $\Gamma \vdash \llbracket M1 \rrbracket = h1 : MSP1 \rightarrow MSP2$ and $\Gamma \vdash \llbracket M2 \rrbracket = h2 : MSP2 \rightarrow MSP3$ then $\Gamma \vdash \llbracket M2 \circ M1 \rrbracket = h2 \circ h1$.
 7. If, for every i, j ($1 \leq i \leq n$ $1 \leq j \leq m$), $\Gamma \vdash \llbracket Ei \rrbracket = SPi$, $\Gamma \vdash \llbracket Mj \rrbracket = hj$, with SPi, hj in Spec and $t \in T_{\Sigma SPEC}(\langle \{X1, ..., Xn\}, \{Y1, ..., Ym\} \rangle)$ of sort MORPH then

$$\Gamma \vdash \llbracket t[E1/X1, ..., Mm/Ym] \rrbracket = Sem(t[SP1/X1, ..., hm/Ym])$$

8. If $\Gamma \vdash \llbracket M1 \rrbracket = h1 : MSP1' \rightarrow MSP1$, $\Gamma \vdash \llbracket M2 \rrbracket = h2 : MSP2 \rightarrow MSP2'$ and $F : Body(MSP1) \Rightarrow Body(MSP2)$, $F' : Body(MSP1') \Rightarrow Body(MSP2')$ are such that the diagram in Figure 12 commutes in Spec, then $\Gamma \vdash \llbracket \langle M1, M2 \rangle \rrbracket = \langle h1, h2 \rangle$.

$$\begin{array}{ccc}
 Body(MSP1) & \xRightarrow{F} & Body(MSP2) \\
 \downarrow Body(h1) & & \downarrow Body(h2) \\
 Body(MSP1') & \xRightarrow{F'} & Body(MSP2')
 \end{array}$$

Fig. 12. Semantics of higher-order morphism expressions

9. If $\Gamma \vdash \llbracket M \rrbracket = \langle h1, h2 \rangle$ then $\Gamma \vdash \llbracket fst(M) \rrbracket = h1$ and $\Gamma \vdash \llbracket snd(M) \rrbracket = h2$
 10. If $\Gamma \cup \{X : E\} \vdash \llbracket E1 \rrbracket = MSP1$, $\Gamma \vdash \llbracket E \rrbracket = MSP$, $\Gamma \vdash \llbracket E2 \rrbracket = MSP2$ and $\Gamma \vdash \llbracket M \rrbracket = h : MSP \rightarrow MSP2$ then $\Gamma \vdash \llbracket [E1/E2]_M \rrbracket = h' : MSP1 \rightarrow MSP3$, where $MSP3 = \Gamma \vdash \llbracket (\lambda X : E. E1)(E2)_M \rrbracket$ and h' is the morphism defined by the instantiation operation between $MSP2$ and the result of the instantiation (see def. 6).

Remark 11. In general, the idea underlying this semantics when dealing with an expression including free variables is that, given an environment, the meaning of the expression is defined as if the variables were substituted by the expressions denoting their type. This is similar to what we explained about obtaining the ground version of a specification in the example in section 3.

Now, once defined the syntax and semantics of module expressions the next step is the definition of a β -reduction relation. Unfortunately, the most obvious candidate does not work, as the following example shows:

Example 12. Consider the following semantically valid module expressions:

$$E = \lambda X : (\lambda Y : SP1.E1).\lambda Z : SP1.X(Z)_{id}$$

$$E' = \lambda Y : SP2.E2$$

where id is the identity morphism, $id : SP1 \rightarrow SP1$, $SP1$ and $SP2$ denote two different basic specifications and $E1[SP1/Y]$ and $E2[SP2/Y]$ are semantically valid expressions. Then E and E' are semantically valid and moreover:

$$\llbracket E \rrbracket = ((SP1, MSP1, F1), (SP1, MSP1, F1), id : Body(MSP1) \rightarrow Body(MSP1))$$

$$\llbracket E' \rrbracket = (SP2, MSP2, F2)$$

for some multiple morphisms $Fi : SPi \Rightarrow Body(MSPi)$ ($i = 1, 2$), where $\{Y : SP1\} \vdash \llbracket E1 \rrbracket = MSP1$ and $\{Y : SP1\} \vdash \llbracket E2 \rrbracket = MSP2$. Now, suppose that $\langle h1, h2 \rangle$ is a morphism from $(SP1, MSP1, F1)$ to $(SP2, MSP2, F2)$. Then, the expression $E(E')_{\langle h1, h2 \rangle}$ is semantically valid. Actually, $\llbracket E(E')_{\langle h1, h2 \rangle} \rrbracket = (SP1, MSP2, h2 \circ F1)$. However, if we β -reduce $E(E')_h$ by substituting in $\lambda Z : SP1.X(Z)_{id}$ all occurrences of X by E' the result is not valid anymore:

$$E(E')_{\langle h1, h2 \rangle} \rightarrow_{\beta} \lambda Z : SP1.(\lambda Y : SP2.E2[Y])(Z)_{id}$$

The problem is that we have assumed that $SP1$ and $SP2$ are different and, as a consequence, we cannot use id in the application $(\lambda Y : SP2.E2[Y])(Z)_{id}$ $\square$

The problem shown by the example above can be avoided by defining β -reduction (or, more precisely, the substitution operation which is the basis of β -reduction) in a slightly more complex way. In particular we need also to change adequately the morphisms involved. In the example above, the right β -reduction would be:

$$E(E')_{\langle h1, h2 \rangle} \rightarrow_{\beta} \lambda Z : SP1.(\lambda Y : SP2.E2[Y])(Z)_{h1}$$

Unfortunately, there is still another, more subtle, problem in this approach as the following example shows:

Example 13. Consider the following semantically valid module expressions:

$$E = \lambda X : (\lambda Y : SP1.E1).X$$

$$E' = \lambda Y : SP2.E2$$

where, as before, $SP1$ and $SP2$ denote two different basic specifications and $E1[SP1/Y]$ and $E2[SP2/Y]$ are semantically valid expressions. Then E and E' are semantically valid and moreover:

$$\llbracket E \rrbracket = ((SP1, MSP1, F1), (SP1, MSP1, F1), id : Body(MSP1) \rightarrow Body(MSP1))$$

$$\llbracket E' \rrbracket = (SP2, MSP2, F2)$$

for some multiple morphisms $F1 : SP1 \Rightarrow Body(MSP1)$ and $F2 : SP2 \Rightarrow Body(MSP2)$, where $MSP1 = \{Y : SP1\} \vdash \llbracket E1 \rrbracket$ and $MSP2 = \{Y : SP1\} \vdash \llbracket E2 \rrbracket$. Now, suppose as in the previous example that $\langle h1, h2 \rangle$ is a morphism from $(SP1, MSP1, F1)$ to $(SP2, MSP2, F2)$. Then, the expression $E(E')_{\langle h1, h2 \rangle}$ is semantically valid. Actually,

$$\llbracket E(E')_{\langle h1, h2 \rangle} \rrbracket = (SP1, MSP2, h2 \circ F1)$$

But, if we β -reduce $E(E')_h$ in the only possible way, by substituting X by E' :

$$E(E')_{\langle h1, h2 \rangle} \rightarrow_{\beta} E'$$

we have two not very satisfactory consequences. The first one is that β -reduction does not preserve the semantics since $\llbracket E(E')_{\langle h1, h2 \rangle} \rrbracket \neq \llbracket E' \rrbracket$. The second one is that the most reasonable definition of β -reduction still does not work. The problem is similar to the one shown in the example above but the reasons are slightly different. In particular, consider now the expression:

$$(E(E')_{\langle h1, h2 \rangle})(SP1)_{id}$$

This expression is semantically valid since $\llbracket E(E')_{\langle h1, h2 \rangle} \rrbracket = (SP1, MSP2, F2)$ and, thus, it can be applied to $SP1$ via the identity. However, if we consider a standard rule defining β -reduction, if $E \rightarrow_{\beta} E'$ then $C[E] \rightarrow_{\beta} C[E']$ for any context C , then we would have that $(E(E')_{\langle h1, h2 \rangle})(SP1)_{id} \rightarrow_{\beta} E'(SP1)_{id}$. Unfortunately, again, $E'(SP1)_{id}$ is not semantically valid. $\square$

One may find the solution to the problem shown in the previous example by making the observation that if we substitute the expression E from example 13 by the expression E of example 12 then the problem disappears, although their meaning is the same (they are η -convertible). That is, if we take the expression

$$E = \lambda X : (\lambda Y : SP1.E1).\lambda Z : SP1.X(Z)_{id}$$

from example 12 and we perform the β -reduction step on the subexpression $E(E')_{\langle h1, h2 \rangle}$ as explained above, we have:

$$(E(E')_{\langle h1, h2 \rangle})(SP1)_{id} \rightarrow_{\beta} (\lambda Z : SP1.E'(Z)_{h1})(SP1)_{id}$$

which is a semantically valid expression. Being specific the problem with the expression E from example 13 is that the information about the "type" is only implicit, i.e. we know that the result of an expression such as $\lambda X : (\lambda Y :$

$SP1.E1).X$ is not a basic specification only by looking to the variable declarations and making some "type inference" (in this case quite trivial). This causes that if we β -reduce the expression and substitute some formal parameters by actual parameters of slightly different type then the type of the result may also change and, as a consequence, it may produce some type mismatches with respect to the context of the expression. This is not the case when considering the expression E in example 12, where the type information about the result $\lambda Z : SP1.X(Z)_{id}$ is included in the result itself. Hence, substituting some external formal parameters by actual parameters of slightly different type will not change the type of the result and will not produce type mismatches. The consequence is that, if we want to avoid this problem, we need to work only with expressions including explicitly all the information necessary about their type. This means that expressions must be "sufficiently" η -expanded or, to say it in other words, the expressions must be in η -long normal form, as it also happens in some theorem proving tools based on some typed λ -calculus such as Isabelle.

As a consequence, in order to define adequately the β -reduction relation, we first have to define a substitution operation avoiding the problem shown in example 12 and, then, restrict ourselves to η -long λ -expressions. As usual, in the substitution operation we assume that there is no problem with clashes of variable names, using α -conversions when needed.

Definition 14. Given $E1, E2[X]$ and $E3$, such that $\Gamma \vdash \llbracket E1 \rrbracket = MSP1, \Gamma \cup \{X : E1\} \vdash \llbracket E2[X] \rrbracket = MSP2$ and $\Gamma \vdash \llbracket E3 \rrbracket = MSP3$ and given a morphism expression M , with $\Gamma \vdash \llbracket M \rrbracket = h : \llbracket E1 \rrbracket \rightarrow \llbracket E3 \rrbracket$, the substitution of the expression $E3$ through M in $E2[X]$, written $Subst(E2[X], E3, M)$, is the expression:

1. If $E2[X] = X$, then $Subst(E2[X], E3, M) = E3$.
2. If $E2[X] = E1'[X](E2'[X])_{M1}$, $E5 = Subst(E1'[X], E3, M)$ and $E6 = Subst(E2'[X], E3, M)$, then $Subst(E2[X], E3, M) = E5(E6)_{M1'}$, where $M1' = M6 \circ M1 \circ fst(M5)$ with $M6 = [E2'[X]/E3]M$ and $M5 = [E1'[X]/E3]M$.
3. If $E2[X] = \lambda Y : E1'.E2'$, then $Subst(E2[X], E3, M) = \lambda Y : E1'.E5$ where $E5 = Subst(E2', E3, M)$.
4. If $E2[X] = t[E1'/X1, \dots, En'/Xn]$, where $t \in T_{SPEC}(\{X1, \dots, Xn\})$ then

$$Subst(E2[X], E3, M) = assign(\dots assign(t[X1, \dots, Xn], M1), \dots Mn)$$

where for every i $Mi = [Ei'/E3]M$.

Proposition 15. Given expressions $E1, E2[X]$ and $E3$ and a morphism expression M , as in the previous definition, if $Subst(E2[X], E3, M) = E4$ then $\Gamma \vdash \llbracket E4 \rrbracket = MSP4$, for some $MSP4$.

proof sketch Cases 1, 3 and 4 are very simple. With respect to case 2, we have to see that if $\Gamma \cup \{X : E1\} \vdash \llbracket E5 \rrbracket = (IMP5, RES5, F)$ and $\Gamma \cup \{X : E1\} \vdash \llbracket E6 \rrbracket = MSP6$ then $\Gamma \cup \{X : E1\} \vdash \llbracket M1' \rrbracket : IMP5 \rightarrow MSP6$. We know that $M1' = M6 \circ M1 \circ fst(M5)$, but we also know that: $\Gamma \cup \{X : E1\} \vdash \llbracket M6 \rrbracket :$

$MSP2' \rightarrow MSP6$, where $MSP2' = \Gamma \cup \{X : E1\} \vdash \llbracket E2'[X] \rrbracket$, $\Gamma \cup \{X : E1\} \vdash \llbracket M1 \rrbracket : IMP1' \rightarrow MSP2'$, where $(IMP1', RES1', F1') = \Gamma \cup \{X : E1\} \vdash \llbracket E1'[X] \rrbracket$, $\Gamma \cup \{X : E1\} \vdash \llbracket M5 \rrbracket : (IMP1', RES1', F1') \rightarrow (IMP5, RES5, F)$, implying $\Gamma \cup \{X : E1\} \vdash \llbracket fst(M5) \rrbracket : IMP5 \rightarrow IMP1' \square$

With respect to the second problem described above, we must restrict ourselves to expressions which are "sufficiently" η -expanded, where η -expansion is the inverse transformation to η -reduction:

$$C[(\lambda X : E1.E2(X)_{id})] \rightarrow_{\eta} C[E2]$$

where C is any arbitrary context. It is clear that, in general, one can η -expand a term infinitely many times. For instance:

$$C[E2]_{\eta} \leftarrow C[(\lambda X : E1.E2(X)_{id})_{\eta} \leftarrow C[(\lambda X : E1.(\lambda X : E1.E2(X)_{id})(X)_{id})_{\eta} \leftarrow \dots$$

However, given an arbitrary term it is sufficient to expand it finitely many times to make explicit all the necessary information. In particular this leads to the following notion of η -long normal form:

Definition 16. *Given an expression E and a position p in E we say that E is η -expanded at p if either the term at p is a λ -abstraction or a basic specification expression or if the term at position q is an application and $p = q \cdot 1$. An expression E is in η -long normal form if E is η -expanded at every position.*

As usual, we may see λ -expressions as trees and positions in a term as sequences of positive integers where the position of the root is the empty string and if p is a position in E then $p \cdot 1$ is its leftmost son, $p \cdot 2$ is its second son, etc. For instance, the expression $\lambda X : (\lambda Y : SP1.E1).\lambda Z : SP1.X(Z)_{id}$ is in η -long normal form, but the expression $\lambda X : (\lambda Y : SP1.E1).X$ is not. The following proposition shows that if we use the substitution operation defined in 14 for defining β -reduction the semantics of terms in η -long normal form is preserved.

Proposition 17. *If $E2$ is in η -long normal form, $\Gamma \vdash \llbracket (\lambda X : E1.E2[X])(E3)_M \rrbracket = MSP$ and $Subst(E2[X], E3, M) = \langle E4, h4 \rangle$ then $\Gamma \vdash \llbracket E4 \rrbracket = MSP$.*

proof sketch By induction on the order of $E2$: If MSP is a basic specification in *Spec* the result holds by construction since $Body(MSP)$ and $Body(\Gamma \vdash \llbracket E4 \rrbracket)$ always coincide, even if $E2$ is not in η -long normal form. If MSP is higher-order (and hence $E2$) then $E2$ must be a λ -abstraction. Thus, applying induction, one would trivially prove the claim. $\square$

Now, we can define the β -reduction relation (almost) as usual:

Definition 18. *β -reduction is the least reflexive and transitive relation satisfying:*

1. $(\lambda X : E1.E2[X])(E3)_M \rightarrow_{\beta} Subst(E2[X], E3, M)$.
2. If $E2[X] \rightarrow_{\beta} E2'[X]$ then $\lambda X : E1.E2[X] \rightarrow_{\beta} \lambda X : E1.E2'[X]$.
3. If $E1 \rightarrow_{\beta} E1'$ and $E2 \rightarrow_{\beta} E2'$ then $E1(E2)_M \rightarrow_{\beta} E1'(E2')_M$.
4. If $E \rightarrow_{\beta} E'$ and $t[E] \in T_{\Sigma SPEC}(\mathcal{SE})$ then $t[E] \rightarrow_{\beta} t[E']$

The following theorem is a direct consequence of proposition 17 and states that β -reduction preserves the semantics of λ -expressions and, in particular, semantic validity. Also, this theorem may be seen as a proof that this calculus satisfies the so-called subject reduction property for expressions in η -long normal form.

Theorem 19. *Let $E1$ and $E1'$ be specification expressions in η -long normal form, if $E1 \rightarrow_\beta E1'$ then $\Gamma \vdash \llbracket E1 \rrbracket = \Gamma \vdash \llbracket E1' \rrbracket$ for every Γ such that $\Gamma \vdash \llbracket E1 \rrbracket$ is defined.*

Acknowledgements This work initially started in cooperation with Hartmut Ehrig. Although he gave it up after some time, afterwards we still had some useful discussions. This work has been partially supported by Spanish CICYT projects COSMOS (ref. TIC95-1016-C02-01) and HEMOSS (ref. TIC98-0949-C02-01).

References

- [1] E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner (eds.) *Algebraic Foundations of System Specification*, Springer-Verlag, To appear.
- [2] H. Barendregt: Typed λ -calculi, in *Handbook of Logic in Computer Science* (S. Abramski et. al, eds.), Oxford Univ. Press 1993.
- [3] R.M. Burstall, J.A. Goguen. The semantics of Clear, a specification language, *Proc. Copenhagen Winter School on Abstract Software Specification*, LNCS 86, 292–332, Springer 1980
- [4] M.V. Cengarle. *Formal Specification with higher order parameterization*, Ph. D. Thesis, Ludwig-Maximilians Universitaet, Muenchen, 1994
- [5] H.-D. Ehrig. On the theory of specification, implementation and parameterization of abstract data types. *J. of the ACM* 29, 209–277, (1982)
- [6] H. Ehrig, H.-J. Kreowski, J.W. Thatcher, E. Wagner, J. Wright. Parameter passing in algebraic specification languages. *Theor. Comp. Science* 28, 45–81 (1984)
- [7] H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specifications 1*, Springer 1985
- [8] J.A. Goguen, R.M. Burstall. Introducing institutions. *Proc. Logics of Programming Workshop*, Carnegie-Mellon. LNCS 164, 221–256, Springer 1984
- [9] J.A. Goguen, R.M. Burstall. Institutions: Abstract model theory for specification and programming, *J. of the ACM* 39(1), 95–146, (1992)
- [10] R. Jiménez, F. Orejas, H. Ehrig. Compositionality and compatibility of parameterization and parameter passing in specification languages, *Math. Structures in Computer Science* 5(2), 283–313 (1995)
- [11] B. Krieg-Brückner, D. Sannella: Structuring specifications in-the-large and in-the-small: higher-order functions, dependent types and inheritance in SPECTRAL, in *TAPSOFT'91 Vol 2*, S. Abramski and T.S.E. Maibaum (eds.), Springer LNCS 494 (1991) pp. 313–336
- [12] D. Sannella, S. Sokolowski, A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica* 29, 689–736 (1992)
- [13] D.T. Sannella, M. Wirsing. A kernel language for algebraic specification and implementation, *Proc. FCT-83*, LNCS 158, 413–427, Springer 1983
- [14] M. Wirsing. Structured algebraic specifications: a kernel language. *Theor. Comp. Sc.* 42,123–249 (1986)
- [15] M. Wirsing. Algebraic Specification. *Handbook of Theoretical Computer Science, Vol 2: Formal Models and Semantics*, pp. 675 – 788, Elsevier 1991