Morphisms and Semantics for Higher Order Parameterized Programming

Kai Lin and Joseph Goguen
Dept. Computer Science & Engineering
University of California at San Diego
9500 Gilman Drive, La Jolla, CA 92093-0114, USA
{klin, goguen}@cs.ucsd.edu
Phone: (858) 534-4197. Fax: (858) 534-2079

Abstract. Parameterized programming is extended to higher order modules, by extending views, which fit actual parameters to formal parameters in a flexible way, to morphisms, with higher order module expressions to compose modules into systems. A category theoretic semantics is outlined, and examples in BOBJ show the power of morphisms.

1 Introduction

Modularization is key to controlling the complexity of large systems, by composing them from parts, so as to ease program construction and maintenance. First order parameterized programming [10, 11] significantly enhances the flexibility and reusability of modularization, by providing *parameterized modules* and *views*, to flexibly fit syntax of formal parameters to actual parameters, including defaults when there is an obvious choice. Here we generalize views to *morphisms*, which are used for both instantiating and refining higher order modules.

Our examples use the BOBJ [14] extension of the OBJ3 specification and functional programming language [16] to higher order parameterized programming, but the approach is not limited to specification and functional languages, as shown by the Lileanna [23, 15] extension of Ada. Although Lileanna modules are first order, the same techniques can implement higher order module systems for compiled functional and imperative languages. Module bodies are translated to the compiler's intermediate language (Dianna for Ada), which is manipulated for instantiation and refinement of modules. Once fully instantiated, the intermediate code is passed to the compiler's backend for optimization. This supports separate compilation, and experiments found that compiled Lileanna is often more efficient than Ada, because larger blocks of intermediate code are optimized. For an interpreted language like BOBJ, it suffices to transform signatures and equations, though the transformations are still rather complex.

OBJ began around 1974 as a notation for algebraic specification, and was soon implemented as a term rewriting interpreter. Early designs [9] had a first order module system like that of Clear [1], later formalized as *parameterized programming* [10], which added default views and simplified syntax. Parameterized programming was first fully implemented in OBJ3 [16], extending earlier versions of OBJ, and it appears in all members of the OBJ family, including CafeOBJ

[5], Maude [3], BOBJ [14], and the European languages CASL [4] and ACTTWO. Semantics follows that of Clear [1], using a category of theories, with theory morphisms for views, and colimit for module composition, based on ideas from an early category theoretic general systems theory [8]. This semantics works over any logical system, using the formalism of institutions [13].

Other languages influenced by parameterized programming include Ada, ML, C++, and Modula, none of which has views, so that actual parameter syntax must contain formal parameter syntax; only ML has higher order modules, and that only in recent releases. Ada generics do not even allow instantiated parameterized modules as actual parameters. C++ has higher order parameterized templates, but these are little more than macro expansions with type checking. Larch [7] has a parameter passing mechanism that can simulate some uses of first order views, but does not have views as reusable first class citizens. Recent SML/NJ releases of ML include higher order parameterized modules, based on higher order parameterized signatures, but without views. The only implemented higher order module system with views is in BOBJ.

We believe views are not just a syntactic convenience, but are key to unlocking the full potential of higher order parameterization. We speculate that the lack of views explains the confusing multiplicity of semantics given for ML higher order "functors," as well as ML's awkward treatment of functor sharing.

An early operational semantics for higher order ML modules due to MacQueen and Tofte [20] is expressed with rules of deduction, using higher order parameterized signatures and "stamps" (i.e., compile-time generated unique identifiers) to handle sharing. As in ML, it requires actual parameter signatures to contain formal parameter signatures, which is a special case of default viewing. Cengarle [2] extends the ASL approach [22] to higher order modules, with module denotations sets of models, functions between sets of models, etc., expressing parameterization with simply typed lambda calculus. Since this cannot distinguish legal from illegal specification expressions, "requirement conditions" are used for "signature terms" and "specification terms" (i.e., module expressions), with a complex deductive system to weed out ill-formed expressions; in addition, the signatures of actual and formal parameters are required to be equal. This strong restriction and much of the complexity in [2] can be avoided with views.

Jiminénez and Orejas [18] work over a category having "multiple pushouts," and use a lambda calculus formalism inspired by ASL to define higher order modules and views over this category; multiple pushouts are used for instantiation, subject to some technical conditions that seem very restrictive. We believe these problems arise from the way that the lambda calculus is used for parameterization. There is also a type theoretic literature on higher order modules, largely associated with ML, but because this approach is remote from ours, we do not survey it here; see [6] and its references.

Space precludes many details, including proofs, some definitions, and further examples; see the full technical report [19]. Parts of the paper assume familiarity with basics of category theory and algebraic specification (e.g. [17, 21]).

2 Examples

This section introduces parameterized programming through examples, beginning with the first order case, and then higher order examples.

Example 1 The simplest non-trivial module has only a single sort:

```
th TRIV is sort Elt .  end
```

The keyword "`th`" introduces "theories," which consist of declarations for sorts, operations, and variables, plus equations over these declarations. `TRIV` has just one declaration, for the sort `Elt`. The name of the module, `TRIV`, follows the keyword `th`. Next, we define a parameterized module `SET` for finite sets:

```
th SET[X :: TRIV] is sort Set .
   op empty : -> Set .
   op insert : Elt Set -> Set .
   op _in_ : Elt Set -> Bool .
   vars E E1 E2 : Elt . var S : Set .
   eq E in empty = false .
   eq E in insert(E,S) = true .
   eq E1 in insert(E2,S) = E1 in S if E1 =/= E2 .
end
```

Since this module has `TRIV` as its interface, it can be instantiated with any module having at least a sort. A new sort `Set` is declared, followed by three operations: `empty` is a constant for the empty set; `insert` adds an element to a set; and `in` checks if an element is in a set. The module `BOOL` of booleans is imported into every module (unless explicitly excluded), and operations declared in the form used here have "parenthesis-with-commas" syntax. □

Example 2 We can instantiate `SET` with a simple view, which picks out a particular sort in a module, such as the following

```
view V1 from TRIV to BOOL is sort Elt to Bool . end
view V2 from TRIV to INT  is sort Elt to Int .  end
```

Here `INT` is BOBJ's built-in module for integers. Instantiation using a named view may be done in the following forms:

```
make SET-OF-BOOL is SET[V1] end
open SET[V2] .
  red 2 in insert(1, empty) .
close
```

where the first creates a new named module, and the second "opens" a temporary environment. The command "`red`" calls for "reducing" the term that follows, in the sense of term rewriting, applying equations as rules left to right until no more apply. In both cases, since there is just one reasonable view, BOBJ can compute it from just the target, and we can write `SET[INT]` instead of `SET[V1]`, and `SET[BOOL]` instead of `SET[V2]`. These are default views, and their use can greatly simplify complex examples. □

In algebraic specification, a first order parameterized module is an inclusion $F\colon I \to M$, where I is the interface and M is the body; recall also that a view is a theory morphism, that is, a signature morphism that maps equations to logical consequences of equations in the target theory. Moreover, the instantiation of a parameterized module is the pushout of the inclusion with a view $f\colon I \to P$, denoted $F[f]$ or perhaps $M[f]$; the term **fitting morphism** is used for a view in an instantiation.

Definition 1 A **pushout** of $f\colon A \to B$, $g\colon A \to C$ in a category $\mathbf{C}$ is a pair $p\colon B \to D$, $q\colon C \to D$ such that $p \circ f = q \circ g$, and for any E and morphisms $r\colon B \to E$ and $s\colon C \to E$ such that $r \circ f = s \circ g$, there is a unique $t\colon D \to E$ such that $t \circ q = r$ and $t \circ p = s$; D is the **pushout object**. □

Intuitively, pushouts combine two objects, identifying parts of one with parts of another; this gives an elegant semantics for instantiating parameterized modules. (For more detail, see [12]; all this actually works for any logical system, via the formalism of institutions [13]. For example, the institution for Lileanna [23, 15] has Anna for specifications and Ada programs for models.) For the special case of Example 1, the construction is as follows: Given a fitting morphism $f\colon$ `TRIV` $\to P$, let $E = f($`Elt`$)$. Then the signature of `SET`$[f]$ is $\Sigma^{\texttt{SET}}(\texttt{Elt} \leftarrow E) \cup \Sigma^{P}$, the result of substituting E for `Elt` in the signature of `SET`, and then combining that with the signature of P (assuming there are no name clashes). Operations and equations are treated similarly.

Example 3 It is common to restrict the interface of a module to a more specialized module; e.g., `WEIGHT` below specializes `TRIV` by requiring actual modules to have (at least) a sort with a natural number valued operation on that sort,

```
th WEIGHT is sort Thing .  pr NAT .
   op weight : Thing -> Nat .
end
```

and we can restrict `SET` from Example 1 to the interface `WEIGHT` as follows:

```
th SETW[X :: WEIGHT] is pr SET[X] .  end
```

There is an obvious view from `TRIV` to `WEIGHT`, which just maps sort `Elt` to `Thing`; it is a default view, already constructed and used by BOBJ in evaluating `SET[X]` in `SETW` above.

```
view D from TRIV to WEIGHT is sort Elt to Thing .  end
```

However, the relation between `SET` and `SETW` is more subtle, and cannot be captured by just a view. The intuition is that `SETW` can be used wherever `SET` is called for. This more general notion is defined below. □

Definition 2 Given first order parameterized algebraic specifications (which could be modules over any institution [13, 12]), $F : I \to M$ and $F' : I' \to M'$, a **first order morphism** from F to F' is a view (i.e., a ground morphism) $d\colon I' \to I$ and a view $r\colon M \to M'[d]$ such that the diagram below commutes,

$$\begin{tikzcd}
M' \arrow[d, "d'"'] & I' \arrow[l, "F'"'] \arrow[d, "d"] & \\
M'[d] & I \arrow[l, "i"'] \arrow[r, "F"] & M \arrow[ll, bend left, "r"]
\end{tikzcd}$$

where the left square is the pushout of F' and d. □

This definition improves on the literature; e.g., [18] is less general in lacking the pushout. Intuitively, $M'[d]$ is a "re-parameterization" or "specialization" of M', to interface I instead of I', and r is a view from M to this re-parameterization. We apply $M'[d]$ to an actual parameter with a fitting view

$f\colon I \to P$ as usual by pushout, obtaining $M'[d][f] = M'[f \circ d]$. The first order parameterized module $I \to M'[d]$ is the **result** of (d, r). If $I = I'$ and $d = 1_I$, then $M'[d] = M'$ and (d, r) amounts to an ordinary view $r\colon M \to M'$. Thus we may refer to views as morphisms, and or even conversely.

Example 4 Define parameterized lists as follows:

```
th LIST[X :: TRIV] is sort List .
   op nil : -> List .      op cons : Elt List -> List .
   op hd : List -> Elt .   op tl : List -> List .
   op contains : List Elt -> Bool .
   var L : List . var S S' : Elt .
   eq hd(cons(S, L)) = S .  eq tl(cons(S, L)) = L .
   eq contains(nil, S) = false .
   eq contains(cons(S', L), S) = contains(L, S) or S == S' .
end
```

Here `nil` is the empty list, `cons` adds an element at the head of a list, `hd` and `tl` return the head and tail of a list, while `contains` checks if an element is in a list. A morphism from `SETW` to `LIST` is defined as follows:

```
morph SETW-TO-LIST from SETW to LIST is
    [view to WEIGHT is sort Elt to Thing . end]
    sort Set to List .  op empty to nil .  op insert to cons .
    var T : Thing . var S : Set .
    op T in S to contains(S,T) .
end
```

Note that all this is default, except the last; the abstract module morphism that it determines is given by the construction in Proposition 3 below. Use of this code requires a second order setting, for which see Example 7 in Section 4. □

Example 5 There is a morphism from `SETW` to `SET`, since we can use `SET` wherever `SETW` is needed:

```
morph SETW-TO-SET from SETW to SET is
   [view to WEIGHT is sort Elt to Thing . end]
end
```

This morphism has two components: a view d, from the target interface to the source interface, and a mapping from the body of the source to the body of the target via `D`. An interface view can be omitted if it is a default. For example, the above morphism can be written in any of the following forms:

```
morph SETW-TO-SET from SETW to SET is [view to WEIGHT is end] end
morph SETW-TO-SET from SETW to SET is end
morph SETW-TO-SET from SETW to SET is [D] end
```

since `D` is already defined. □

Example 6 The usual reason for restricting an interface is to support adding some new functionality to the body of its module. For example, we might want to calculate the total weight of things in a set of things that each have weight, as in the following:

```
th SETWT[X :: WEIGHT] is pr SETW[X] .
   op total : Set -> Nat .
   vars E : Thing . var S : Set .
   eq total(empty) = 0 .
   eq total(insert(E, S)) = weight(E) + total(S) if not E in S .
   eq total(insert(E, S)) = total(S) if E in S .
end
```

Notice that there is *no* module morphism between SET and SETWT, because the view and the natural transformation involved go in the *same* direction, rather than in opposite directions. This corresponds to the fact that we cannot use either of these where the other would be needed. However, there is a morphism from SETW to SETWT, and we can use SETWT wherever SETW was needed:

```
morph SETW-TO-SETWT from SETW to SETWT is [view to WEIGHT is end] end
```

(Note that both components of this are defaults.) □

Of course, higher order modules can do much more than this.

3 Abstract Parameterized Modules

If $\mathbf{C}$ is a category, then $|\mathbf{C}|$ denotes the class of objects of $\mathbf{C}$, and $1_{\mathbf{C}}$ denotes the identity functor on $\mathbf{C}$. Also, if A is an object in $\mathbf{C}$, then $\ulcorner A \urcorner \colon \mathbf{1} \to \mathbf{C}$ denotes the functor that sends the unique object in $\mathbf{1}$ to A, where $\mathbf{1}$ is the one morphism category, and 1_A denotes the identity morphism on A.

Definition 3 Given functors $F\colon \mathbf{A} \to \mathbf{C}$, $G\colon \mathbf{B} \to \mathbf{C}$, the **comma category** (F/G) has: triples (A, B, c) as objects, with $A \in |\mathbf{A}|$, $B \in |\mathbf{B}|$ and $c\colon F(A) \to G(B)$ in $\mathbf{C}$; pairs (f, g) as morphisms $(A_1, B_1, c_1) \to (A_2, B_2, c_2)$, where $f\colon A_1 \to A_2$, $g\colon B_1 \to B_2$, such that $c_1 \circ G(g) = F(f) \circ c_2$; the pair $(1_{F(A)}, 1_{G(B)})$ as identity on (A, B, c); and composition of $(f_1, g_1)\colon (A_1, B_1, c_1) \to (A_2, B_2, c_2)$ and $(f_2, g_2)\colon (A_2, B_2, c_2) \to (A_3, B_3, c_3)$ is $(f_2 \circ f_1, g_2 \circ g_1)$.

If $\mathbf{C}$ is a category and $A \in |\mathbf{C}|$, then $(A/\mathbf{C})$ denotes the $(\ulcorner A \urcorner/\mathbf{1_C})$, and is called a **slice category**, with A its **top object**. Also, let $(\mathbf{C}/\mathbf{C})$ denote $(\mathbf{1}_C/\mathbf{1}_C)$. If F is a functor on $(A/\mathbf{C})$ and $f\colon A \to C$, then f could be an object or a morphism in $(A/\mathbf{C})$; to avoid this confusion, we may write $|F|(f)$ for the application of F to f as an object.

Given $v\colon B \to A$ in $\mathbf{C}$, a functor $(v/\mathbf{C})\colon (A/\mathbf{C}) \to (B/\mathbf{C})$ has $(v/\mathbf{C})(f) = f \circ v$, for $f\colon A \to X$ in $|(A/\mathbf{C})|$; and $(v/\mathbf{C})(h) = h$, for $h\colon f_1 \to f_2$ in $(A/\mathbf{C})$, which works because h is a morphism $f_1 \circ v \to f_2 \circ v$ in $(B/\mathbf{C})$. □

Although parameterized modules are intuitively functions having modules as input and output, they differ from ordinary functions in significant ways:

1. The conventions of parameterized programming [10] build on those for procedures in ordinary programming. $M[X :: I]$ indicates a first order parameterized module with **body** M, **formal parameter** X, and **interface** I, a ground module "type" for actual modules allowed to instantiate X in M, since not all actual modules give the desired behavior to the instantiation.
2. After category theory, ground modules have *morphisms*, with composition and identities; these correspond to relations of refinement.

3. Often there are many ways to use a given module P as an actual parameter of a module M with interface I, corresponding to morphisms $I \to P$, i.e., to refinements of I by P, where P provides what I requires; such morphisms are called **fitting morphisms**. Thus parameterized modules are functions on fitting morphisms rather than on modules. Let $M[f]$ denote the result of instantiating M with P using $f\colon I \to P$.
4. Parameterized module instantiation should preserve actual module refinement, i.e., it should be *functorial.* Therefore given M with interface I, and P, P' actual modules with fitting morphisms f, f', if $h\colon P \to P'$ such that $f \circ h = f'$, then there should exist $M[h]\colon M[f] \to M[f']$.
5. If possible actual modules form a category $\mathbf{C}$, and possible instantiation results form a category $\mathbf{D}$, then a parameterized module M should be a functor from the slice category $(I/\mathbf{C})$ to $\mathbf{D}$, where I in $\mathbf{C}$ is the interface.
6. A key step in our higher order semantics is defining parameterized module morphisms, so that parameterization can be iterates from order n to order $n+1$. Since modules are functors, their morphisms should be natural transformations. But since their sources vary with the top object of the slice category (i.e., the interface), we also need functors to relate these. Intuitively, an abstract module morphism from F to G allows using G wherever F is needed, with G having a more general (less restrictive) interface and a more refined body (more functionality).

Definition 4 Given categories $\mathbf{C}$, $\mathbf{D}$, an **abstract** (**parameterized**) **module** is a functor $(A/\mathbf{C}) \to \mathbf{D}$ for some $A \in |\mathbf{C}|$, and a **morphism** from $F\colon (A/\mathbf{C}) \to \mathbf{D}$ to $G\colon (B/\mathbf{C}) \to \mathbf{D}$ is a pair (d, η) with $d\colon B \to A$ in $\mathbf{D}$ and η is a natural transformation $F \Rightarrow G \circ (d/\mathbf{C})$. The **identity** 1_F on $F\colon (A/\mathbf{C}) \to \mathbf{D}$ is $(1_A, \mathbf{1}_F)$, where $\mathbf{1}_F$ is the identity natural transformation on F, with $\mathbf{1}_F(f) = 1_{F(f)}$ for $f\colon A \to P$ in $\mathbf{C}$.

Given $F_i\colon (A_i/\mathbf{C}) \to \mathbf{D}$ for $i = 1, 2, 3$ and morphisms $(d_i, \eta_i)\colon F_i \to F_{i+1}$ for $i = 1, 2$, the **composition** $(d_2, \eta_2) \circ (d_1, \eta_1)$ is $(d_1 \circ d_2, \eta)$ where η is the natural transformation $F_1 \Rightarrow F_3 \circ (d_1 \circ d_2/\mathbf{C})$ defined by $\eta(f) = \eta_2(f \circ d_1) \circ \eta_1(f)$, for $f \in |(A_1/\mathbf{C})|$; this makes sense because $\eta_1(f)\colon F_1(f) \to F_2(f \circ d_1)$ and $\eta_2(f \circ d_1)\colon F_2(f \circ d_1) \to F_3(f \circ d_1 \circ d_2)$ are morphisms in $\mathbf{D}$.

$$\begin{tikzcd}
(A_1/\mathbf{C}) \arrow[r, "(d_1/\mathbf{C})"] \arrow[dr, bend right, "F_1"'] & (A_2/\mathbf{C}) \arrow[r, "(d_2/\mathbf{C})"] \arrow[d, "F_2"'] & (A_3/\mathbf{C}) \arrow[dl, bend left, "F_3"] \\
{} \arrow[ur, Rightarrow, "\eta_1"'] & D \arrow[ur, Rightarrow, "\eta_2"'] &
\end{tikzcd}$$

We say A is the **interface** of $M\colon (A/\mathbf{C}) \to \mathbf{D}$, and we let $\mathit{Inf}(M) = A$. □

Proposition 1 The structure with abstract modules as objects and with their morphisms as defined above, is a category, which we denote $(\mathbf{C}/\!/\mathbf{D})$. Moreover, $\mathit{Inf}\colon (\mathbf{C}/\!/\mathbf{D}) \to \mathbf{C}$ is a contravariant functor, sending (d, η) to d. □

For the cogniscenti, the categories $(\mathbf{C}/\!/\mathbf{D})$ are a special "lax comma category," replacing equality in the commutative squares that are morphisms by natural transformations.

To start a hierarchy of higher order modules, we assume a category $\mathbf{M}_0$ of zeroth order, i.e., of unparameterized or ground, modules. We then construct $\mathbf{M}_1$, the first order modules, as $(\mathbf{M}_0//\mathbf{M}_0)$; Section 4 iterates this to obtain modules of all orders. We illustrate the approach with $M_0 = \mathbf{Th}$, the category of unparameterized algebraic theories with subsorts and initiality constraints; since this is a sublogic of BOBJ [14] (excluding its powerful hidden algebra features), we can use BOBJ for executable examples.

The simple idea that a parameterized module is an inclusion extends to higher order algebraic specifications, but not to abstract modules, which include many other examples, such as higher order functions (see the end of Section 4).

Proposition 2 A first order algebraic specification $F\colon I \to M$ determines an abstract module $(I/\mathbf{Th}) \to \mathbf{Th}$, and thus an object in $(\mathbf{Th}//\mathbf{Th})$, which we denote $\overline{F}$, or just F or even $\overline{M}$ or M. □

Proposition 3 Given first order algebraic specifications F, F' with interfaces I, I' and bodies M, M', and given a morphism $(d, r)\colon F \to F'$, there is a natural transformation $\eta\colon \overline{F} \Rightarrow \overline{F'} \circ (d/\mathbf{Th})$ such that (d, η) is an abstract module morphism $\overline{F} \to \overline{F'}$ in $(\mathbf{Th}//\mathbf{Th})$. □

4 Higher Order Modules

Example 7 The ground module `WTHING` below provides an infinite collection of "things," each with a distinct weight, while the following second order module `SETWTHING` instantiates its parameterized formal parameter with `WTHING`, and then extends the result with an operation `total` that gives the total weight in a set of these things:

```
obj WTHING is sort Thing .
  pr NAT .
  op t : Nat -> Thing .    op weight : Thing -> Nat .
  var N : Nat .
  eq weight(t(N)) = 2 * N + 1 .
end
th SETWTHING[X :: SETW] is pr X[WTHING] .
  op total : Set -> Nat .
  vars E : Thing . var S : Set .
  eq total(empty) = 0 .
  eq total(insert(E, S)) = weight(E) + total(S) if not E in S .
  eq total(insert(E, S)) = total(S) if E in S .
end
```

The specification `SETWTHING` determines a second order abstract module, an object in $((\mathbf{Th}//\mathbf{Th})//\mathbf{Th})$, i.e., a functor $(\texttt{SETW}/(\mathbf{Th}//\mathbf{Th})) \to \mathbf{Th}$, by the construction in Example 8 below. We instantiate `SETWTHING` with the abstract module morphism `SETW-TO-SET`:

```
open SETWTHING[SETW-TO-SET] .
  red t(2) in insert(t(1), empty) .
  red total(insert(t(1), empty)).
close
```

We can also instantiate and test the code of Example 3:

```
obj APPLYSW[S :: SETW][W :: WEIGHT] is pr S[W].  end
open APPLYSW[SETW-TO-SET][WTHING] .
  red t(2) in insert(t(1), empty) .
close
```

Both of these give the expected results, and we can similarly test Example 4,

```
obj APPLYSWT[S :: SETW][W :: WEIGHT] is pr S[W].
  op total : Set -> Nat .
  var S : Set .  var T : Thing .  var N : Nat .
  eq total(empty) = 0 .
  eq total(insert(T, S)) = weight(T) + total(S) if not T in S .
  eq total(insert(T, S)) = total(S) if T in S .
end
open APPLYSWT[SETW-TO-LIST][WTHING] .
  red contains(cons(t(1), nil), t(2)) .
  red total(cons(t(1), nil)).
close
```

which again works as expected. □

Higher order modules often include a module expression that uses the formal parameters of the module. The line `pr S[W]` in `APPLYSWT` above illustrates this, since both `S` and `W` are formal parameters, with `S` first order and `W` ground. Such module expressions do not in general denote concrete modules, and can only do so when their formal parameters are fully instantiated. This implies that views (and morphisms) in such expressions are also formal when they contain formal parameters, i.e., they do not denote concrete views (or morphisms). For example, if `M[view to N is ...end]` where `M` and `N` contain formal parameters, then the expression and its inline view are formal. For such an expression to be well formed, the view should have source $Inf(M)$ and target N, instantiating to a concrete view from the interface of the result of instantiating `M` to that of instantiating `N`. "Telescopes" are an important special case; the term was introduced by de Bruijn in type theory, and is also been used for similar situations in algebraic semantics [12].

Definition 5 An nth **order telescope** is a sequence of theory inclusions, $I_n \to ... \to I_1 \to M$, where M is the **body**, and for each $j > 0$, the telescope $I_n \to ... \to I_j$ is the **interface** of I_{j-1} (with $I_0 = M$). □

Example 8 There can be many ways to instantiate a telescope. For example, $F = J \to I \to M$ can be first instantiated J with $f\colon\ J \to P$ to get a map from second order theories to ground theories. The result of instanting F by f is as below, with F_0 the first inclusion and F_1 the second, with both squares pushouts,

$$\begin{tikzcd}
J \arrow[r, "F_0"] \arrow[d, "f"'] & I \arrow[r, "F_1"] \arrow[d, "f'"'] & M \arrow[d, "f''"'] \\
P \arrow[r] & F_0[f] \arrow[r] & F_1[f']
\end{tikzcd}$$

where the diagonal $I \to F_1[f']$ of the second square is the result of instantiating J with f; denoting it $i\colon I \to M[f]$, we can instantiate i with $n\colon K \to N$ using a morphism (d, r) with $d\colon K \to J$ and $r\colon I \to N$, as shown below,

$$\begin{tikzcd}
 & N \arrow[dr] & & (M[f'])[r] & \\
K \arrow[ur, "n"] \arrow[dr, "d"'] & & N[d] \arrow[ur] & & M[f'] \arrow[ul] \\
 & J \arrow[ur] & & I \arrow[ul, "r"] \arrow[ur, "i"'] &
\end{tikzcd}$$

where the two squares are pushouts, and $(M[f'])[r]$ is the instantiation result. One can show that this determines an abstract module, and that instantiating the two formal parameters in the reverse order gives the same result. □

We can generalize from a single higher order parameter to multiple parameters of various orders, by having multiple telescopes of various lengths, $I^i_{n_i} \to \ldots \to I^i_1 \to M$ for various i; this amounts to having a tree of theory inclusions with root M. It may help to think of this as a collection of interconnected lenses that give a range of views under various magnifications. Modules that contain more than one instantiation of one (or more) of their higher order formal parameters can be given semantics using the "multiple morphisms" of [18].

Generalizing to acyclic graphs of inclusions is more challenging, because these allow "dependent interfaces," that contain formal parameters, analoguously to dependent types [12]. OBJ3 and CafeOBJ allow such interfaces; this works because these languages only have first order modules. Formal parameters can play two roles in module expressions: to instantiate modules, and to be instantiated. The latter is only possible for second or higher order modules. Dependent interfaces work for first order modules, because it is possible to check (dynamically) if a view is valid for an instantiation, but this fails for higher orders. For example, if M and V in $M[V]$ contain formal parameters, then when actuals are supplied, V gives a morphism $v\colon f_1 \to f_2$ where f_1 is declared in the code, but f_2 must be computed at run time. For this to work, M must be functorial. Thus BOBJ allows first order dependent interfaces, but not higher order, because these are not in general functorial under instantiation, unlike the special case of telescopes.

The iterative construction of higher order abstract parameterized modules, starting from a category $\mathbf{M}_0$ of ground or zeroth order modules, is straightforward. For $n > 0$, let $\mathbf{M}_{n+1} = (\mathbf{M}_n /\!/ \mathbf{M}_n)$. Every possible higher order abstract module is embedded somewhere in one of these, but for specific concrete modules (e.g., in BOBJ), it is more convenient to work in a more specific category, such as $\mathbf{M}_{i,j} = (\mathbf{M}_i /\!/ \mathbf{M}_j)$, $\mathbf{M}_{(i,j),k} = (\mathbf{M}_{i,j} /\!/ \mathbf{M}_k)$, $\mathbf{M}_{(i,j),(k,l)} = (\mathbf{M}_{i,j} /\!/ \mathbf{M}_{k,l})$ and so on, over what amount to binary trees of natural numbers (see Definition 6). In the most specific instances, all indices are zero. We can form even more specific semantic domains by using slice categories, in forms such as $(I/\mathbf{M}_j)$, $((I/\mathbf{M}_j)/\mathbf{M}_k)$, and so on, as well as functor categories like $[(I/\mathbf{M}_i) \to \mathbf{M}_j]$, etc., which are contained in the categories $(\mathbf{M}_i /\!/ \mathbf{M}_j)$.

We can see how other domains embed in the $\mathbf{M}_n$ through the injections $i_n\colon \mathbf{M}_n \to \mathbf{M}_{n+1}$ defined below, which represent a parameterized module of order n as a constant module of order $n+1$. We assume $\mathbf{M}_0$ has intial and final objects, $\perp$ and $\top$, and we let these symbols also denote initial and final objects in any $\mathbf{M}_n$.

- Given an object F in $\mathbf{M}_0$, let $i_0(F)$ in $\mathbf{M}_1$ be $\hat{F}\colon (\perp/\mathbf{M}_0) \to \mathbf{M}_0$ defined by $\hat{F}(\perp\to T) = F$ for any $\perp\to T$ in $(\perp/\mathbf{M}_0)$, and for f any morphism from $\perp\to T$ to $\perp\to T'$ in $(\perp/\mathbf{M}_0)$, let $\hat{F}(f)$ be the identity on F.
- Given a morphism $h\colon F \to G$ in $\mathbf{M}_0$, let $i_0(h)\colon \hat{F} \to \hat{G}$ in $\mathbf{M}_1$ be $(1_\perp, \eta)$ where $\eta(f) = h$ for any $f\colon T \to T'$ in $(\perp/\mathbf{M}_0)$.
- Given an object F in $\mathbf{M}_n$ for $n > 0$, say $F\colon (A/\mathbf{M}_{n-1}) \to \mathbf{M}_{n-1}$, let $i_n(F)$ be $\hat{F}\colon (\perp/\mathbf{M}_n) \to \mathbf{M}_n$ sending each object $(\perp\to T)$ in $(\perp/\mathbf{M}_n)$ to F, and sending each morphism in $(\perp/\mathbf{M}_n)$ to the identity on F, which is $(1_\perp, \mathbf{1}_F)$.
- Given a morphism $(d,\eta)\colon F \to G$ in $\mathbf{M}_n$ for $n > 0$, let $i_n(d,\eta)\colon \hat{F} \to \hat{G}$ in $\mathbf{M}_{n+1}$ be (d', η'), where $d' = 1_\perp$, and for f a morphism from $\perp\to T$ to $\perp\to T'$, let $\eta'(f) = (d,\eta)$.
- Now we define the initial objects $\perp_n$ in $\mathbf{M}_n$ for $n > 0$; for this, we also define final objects $\top_n$, where the subscripts are temporary for added clarity. Let $\top_n = \widehat{\top}_{n-1}$, and let $\perp_n$ be the functor $(\top_{n-1}/\mathbf{M}_{n-1}) \to \mathbf{M}_{n-1}$ sending each object $\top_{n-1} \to T$ in $(\top_{n-1}/\mathbf{M}_{n-1})$ to $\perp_{n-1}$, and sending each morphism in $(\top_{n-1}/\mathbf{M}_{n-1})$ to the identity on $\perp_{n-1}$, which is $(1_{\top_{n-1}}, \mathbf{1}_{\perp_{n-1}})$. Notice that the definitions of $\perp$ and $\top$ are mutually recursive.

It is easy to check that each i_n is injective, and that $\perp_n$ and $\top_n$ are respectively initial and final objects in $\mathbf{M}_n$. From this and the fact that an injection $\mathbf{N} \to \mathbf{N}'$ induces an injection $(\mathbf{M}//\mathbf{N}) \to (\mathbf{M}//\mathbf{N}')$, it follows for example that $(\mathbf{M}_0//(\mathbf{M}_0//M_0))$ is embedded in $\mathbf{M}_2$. In addition, any inclusion $i\colon \mathbf{M} \to \mathbf{M}'$ induces a functor $i^*\colon (\mathbf{M}'//\mathbf{N}) \to (\mathbf{M}//\mathbf{N})$, as follows: First, given an object M in $\mathbf{M}$, i induces an inclusion $(M/i)\colon (M/\mathbf{M}) \to (M/\mathbf{M}')$, which in turn induces a functor $[(M/i) \to \mathbf{N}]\colon [(M/\mathbf{M}') \to \mathbf{N}] \to [(M/\mathbf{M}) \to \mathbf{N}]$ that sends $F'\colon (M/\mathbf{M}') \to \mathbf{N}$ to $F' \circ (M/i)$. This then extends to the functor $i^*\colon (\mathbf{M}'//\mathbf{N}) \to (\mathbf{M}//\mathbf{N})$. Finally, it can be shown that the image of i^* includes all the functors that never take the value $\perp$. To prove this, extend $F\colon (M/\mathbf{M}) \to \mathbf{N}$ to $F'\colon (M/\mathbf{M}') \to \mathbf{N}$ by filling in the missing values with $\top$; then $i^*(F') = F$. This implies that the reasonable abstract modules in $((\mathbf{M}_0//\mathbf{M}_0)//\mathbf{M}_0)$ also appear in $\mathbf{M}_2$, and so on for higher orders. If $\mathbf{M}_0$ is the theories over some institution, then $\perp$ is the empty theory, which is never useful in specification or computation. We can construct the sequence of injections i_n and take its colimit,

$$\begin{tikzcd}
\mathbf{M}_0 \arrow[r, "i_0"] & \mathbf{M}_1 \arrow[r, "i_1"] & \mathbf{M}_2 \arrow[r, "i_2"] & \cdots \arrow[r, "i_{n-1}"] & \mathbf{M}_n \arrow[r, "i_n"] & \mathbf{M}_{n+1} \arrow[r, "i_{n+1}"] & \cdots
\end{tikzcd}$$

which we denote $\mathbf{M}_\infty$; it is a "universal domain," including all possible higher order abstract modules. But any particular program can work within a more specific category as described above, since a program has only finitely many modules, of finite order. $\mathbf{M}_\infty$ does not have initial or final objects, but we can get them by adding maps $k_n\colon \mathbf{1} \to \mathbf{M}_n$ to the above diagram, sending the unique object in $\mathbf{1}$ to $\perp$, so that the initial objects in the various $\mathbf{M}_n$ are identified in the colimit.

Example 9 If $\mathbf{M}_0 = \mathbf{Th}$, the category of equational (order sorted with initiality constraints [13]) theories, then $\mathbf{M}_\infty$ includes all possible higher order modules that can be written in BOBJ (though hidden algebra should be added to reflect the full power of BOBJ). □

Example 10 Although it would be tedious to give a complete formal definition, the reader can easily imagine a category $\mathbf{C}$ of all C++ template classes, with morphisms given by inheritence. Then with $\mathbf{M}_0 = \mathbf{C}$, the above construction gives a semantic space for higher order templates for C++. □

Given a poset D, let $\overrightarrow{D}$ be the category with objects the elements of D, and with a morphism from a to b iff $a \leq b$. If $\mathbf{N}$ is the poset of natural numbers plus a new minimum $\perp$, with $\perp \leq n$ the only non-trivial orderings, and if $\mathbf{M}_0 = \overrightarrow{\mathbf{N}}$, then the material below shows that $\mathbf{M}_\infty$ contains representations for all partial higher order functions over natural numbers. Let $[C \to D]$ denote the poset of all monotone functions from poset C to poset D.

Definition 6 Given a poset D with minimum element $\perp$, the set $\mathcal{B}$ of **binary types** is as follows: $\bullet$ is in $\mathcal{B}$; and if $t, t' \in \mathcal{B}$, then $t \to t'$ is in $\mathcal{B}$. Given $t \in \mathcal{B}$ and a poset D, let D^t denote the poset of functions of type t, defined as follows: $D^\bullet = D$; and $D^{t \to t'} = [D^t \to D^{t'}]$. □

Definition 7 For any $t \in \mathcal{B}$, define $\perp_t$ in $|\mathbf{M}^t|$ as follows: if $t = \bullet$, let $\perp_t = \perp_D$; and if $t = t_1 \to t_2$, let $\perp_t\colon (\perp_{t_1}/\mathbf{M}^{t_1}) \to \mathbf{M}^{t_2}$ as follows: For $g\colon \perp_{t_1} \to x$ in $\mathbf{M}^{t_1}$, let $\perp_t(g) = \perp_{t_2}$; and for $h\colon g_1 \to g_2$ in $(\perp_{t_1}/\mathbf{M}^{t_1})$, let $\perp_t(h) = \mathbf{1}_{\perp_{t_2}}$. □

Although $\perp_t$ is a kind of totally undefined function, it is not initial in $\mathbf{M}^t$ for types $t \neq \bullet$. Functors $i_t\colon D^t \to \mathbf{M}^t$ and $j_t\colon \mathbf{M}^t \to D^t$, for $t \in \mathcal{B}$, are defined in [19]; they have the properties below; the last one says i_t faithfully represents the functions of D^t within $\mathbf{M}^t$.

Proposition 4 $j_t \circ i_t = 1_{D^t}$, for every $t \in \mathcal{B}$. For any $t \in \mathcal{B}$ and $a \in D^t$, there is a unique morphism from $\perp_t$ to $i_t(a)$. Given f in $D^{t_1 \to t_2}$, a in D^{t_1} and b in D^{t_2}, then $f(a) = b$ iff $i_t(f)(g) = i_{t_2}(b)$, where g is the unique morphism from $\perp_{t_1}$ to $i_{t_1}(a)$. □

5 Semantic Module and Morphism Expressions

This section gives a language of "semantic expressions" for higher order modules over any category of basic modules; the semantics itself is in [19]. The language is analoguous to intermediate compiled code, or assembly code, a convenient intermediary between user level code and actual meaning. Some of these expressions are not meaningful; the semantics in [19] says which ones are.

Definition 8 Let $\mathbf{M}$ be a category, with objects the ground (zeroth order) modules, and let *Var* be a set of variable symbols. Then the semantic expressions for modules and for morphisms, denoted **Modx** and **Morphx** respectively, with their sets of free variables, are as follows:

1. **Module Expressions**:
   (a) $|\mathbf{M}| \subseteq$ **Modx**.
   (b) **Morphx** $\subseteq$ **Modx**.
   (c) Given $X \in \mathit{Var}$ and $E_i \in$ **Modx** for $i = 1, 2$, if E_1 has no free variables (i.e., $\mathit{Var}(E_1) = \emptyset$, as defined below), then $E_2[\,X :: E_1\,] \in$ **Modx**.

(d) Given $E \in$ **Modx** and $E \notin$ **Morphx**, then $E \bullet V \in$ **Modx**.

2. **Morphism Expressions**:
   (a) If $v\colon M \to N$ in $\mathbf{M}$, then $v \in$ **Morphx**.
   (b) $\mathit{Var} \subseteq$ **Morphx**.
   (c) If $V_1, V_2 \in$ **Morphx**, then $V_2 \circ V_1 \in$ **Morphx**.
   (d) Given $V_1, V_2 \in$ **Morphx**, $X \in \mathit{Var}$ and $E \in$ **Modx**, if $\mathit{Var}(E) = \emptyset$, then $\langle V_1, V_2[\,X::E\,]\rangle \in$ **Morphx**.
   (e) If $V_1, V_2 \in$ **Morphx**, then $V_1 \bullet V_2 \in$ **Morphx**.
   (f) If V_1 and $V_2 \in$ **Morphx**, then $V_1 \diamond V_2 \in$ **Morphx**.
   (g) If $E \in$ **Modx**, then $I(E) \in$ **Morphx**.
   (h) If $V_i \in$ **Morphx** for $i = 1, ..., 4$, then $\diamond(V_1, V_2, V_3, V_4) \in$ **Morphx**.
   (i) Given $V \in$ **Morphx** and $E_i \in$ **Modx** for $i = 1, 2$, if $\mathit{Var}(E_1) = \emptyset$, then $V\colon E_1 \Rightarrow E_2 \in$ **Exp**.
3. **Free Variables**: For $E, E_1, ... \in$ **Modx** and $V, V_1, ... \in$ **Morphx**, the free variable sets $\mathit{Var}_E(E)$ and $\mathit{Var}_V(V)$, may be defined in a familiar way.

□

We now explain the constituents of the above definition:

1. $E[X::E']$ indicates that a module expression E with interface E' is a function of morphism expressions X having source E', where E' must have no free variables; a more familiar notation might be $(\lambda X : E').E$.
2. $E \bullet V$ denotes the instantiation of a parameterized module E by a morphism V, which should have the interface of E as its source; this may also be written $E[V]$, especially when the notation $E[X::E']$ has been previously used.
3. $I(E)$ denotes the identity morphism with source E.
4. $V : (E_1 \Rightarrow E_2)$ indicates that V is a morphism from E_1 to E_2.
5. $V_2 \circ V_1$ denotes the composition of the morphism denoted by V_1 with that of V_2, where the target of V_1 should agree with the source of V_2.
6. $V_1 \diamond V_2$ denotes the morphism in the pushout of V_1 and V_2 opposite V_1 (see below), where V_1, V_2 have the same ground module source.
7. $\langle V_1, V_2[\,X::E\,]\rangle$ denotes a morphism from a (higher order) module M to another N, where V_1 is a morphism from the interface of N to E, the interface of M, and where if f is a morphism from E, then $V_2[\,X::E\,] \bullet f$ should be a morphism from $M \bullet f$ to $N \bullet (f \circ V_1)$. We may write $V_2[f]$ instead of $V_2[\,X::E\,] \bullet f$.
8. $\diamond(V_1, V_2, V_3, V_4)$ is as indicated in the right diagram below,

$$\begin{tikzcd}
\bullet \arrow[r, "V_1"] \arrow[d, "V_2"'] & \bullet \arrow[d, "V_2 \diamond V_1"] \\
\bullet \arrow[r, "V_1 \diamond V_2"'] & \bullet
\end{tikzcd}
\qquad
\begin{tikzcd}
\bullet \arrow[d, dashed] & \bullet \arrow[l, "V_1"'] \arrow[r, "V_2"] \arrow[d, "V_4"'] & \bullet \arrow[d, dashed] \arrow[ll, bend right, "V_3"'] \\
\bullet & \bullet \arrow[l, dashed, "V_1 \diamond V_4"] \arrow[r, dashed, "V_2 \diamond V_4"'] & \bullet \arrow[ll, bend left, "\diamond(V_1,V_2,V_3,V_4)"]
\end{tikzcd}$$

where the two squares are pushouts and where $V_1 = V_3 \circ V_2$. By the universal property of pushouts, there is a unique morphism as indicated.

These semantic expressions apply to many different frameworks, not just algebraic specification. Although different frameworks may require different scaffolding, we included pushouts because of BOBJ.

Example 11 Examples 3 and 4 defined modules SETW and LIST, respectively; here we give semantic expressions for them. Let SETW-BODY be the union of SETW with WEIGHT, with $i :$ WEIGHT $\to$ SETW-BODY the inclusion in **Th**. Then the semantic expression for SETW is $(i\Diamond \mathtt{X})[\mathtt{X::WEIGHT}]$. Similarly, the expression for LIST is $(\mathtt{j}\Diamond\mathtt{X})[\mathtt{X::TRIV}]$, where j is the inclusion of TRIV into LIST-BODY, which is the union of TRIV with LIST.

Example 4 defined SETW-TO-LIST. Let $d\colon$ TRIV $\to$ WEIGHT send Elt to Thing, and define $u :$ SETW $-$ BODY $\to j\Diamond d$ as follows, where j is as above: Nat and Thing map to themselves; Set maps to List; empty maps to new; insert maps to cons; and has maps to contains. Then the semantic expression for SETW-TO-LIST is $(d, (\Diamond(i, j \diamond d, u, \mathtt{X}))[\mathtt{X::WEIGHT}] \colon$ SETW $\Rightarrow$ LIST. $\Box$

Example 12 The semantic expression for second order telescope $J \to I \to M$, with $F_0\colon\ J \to I$, $F_1 :\ I \to M$, is $\mathtt{F} = ((\mathtt{Y} \bullet \mathtt{I(J)})\Diamond\mathtt{F}_1)[\mathtt{Y::}((\mathtt{X}\Diamond\mathtt{F}_0)[\mathtt{X::J}])]$. $\Box$

The full version of this paper [19] has semantic details showing how to compute abstract modules from semantic expressions, and semantic expressions from algebraic specifications.

References

1. Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.
2. María Victoria Cengarle and Martin Wirsing. A calculus of higher order parameterization for algebraic specification. *Bulletin of the Interest Group in Pure and Applied Logics*, 3(4):615–641, 1995.
3. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.
4. CoFI Task Group on Semantics. CASL – the common algebraic specification language, 1999. http://www.brics.dk/Projects/CoFI/Documents/CASL.
5. Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification.* World Scientific, 1998. AMAST Series in Computing, Volume 6.
6. Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules, 2002. Submitted for publication.
7. Steven Garland. LP – the Larch prover: version 3.1, 1995. MIT, Laboratory for Computer Science, http://larch.lcs.mit.edu:8001/larch/LP.
8. Joseph Goguen. Mathematical representation of hierarchically organized systems. In E. Attinger, editor, *Global Systems Dynamics*, pages 112–128. S. Karger, 1971.
9. Joseph Goguen. Abstract errors for abstract data types. In Eric Neuhold, editor, *Proceedings, First IFIP Working Conference on Formal Description of Programming Concepts*, pages 21.1–21.32. MIT, 1977. Also in *Formal Description of Programming Concepts*, Peter Neuhold, Ed., North-Holland, pages 491–522, 1979.
10. Joseph Goguen. Parameterized programming. *Transactions on Software Engineering*, SE–10(5):528–543, September 1984.

11. Joseph Goguen. Suggestions for using and organizing libraries in software development. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, First International Conference on Supercomputing Systems*, pages 349–360. IEEE Computer Society, 1985. Also in *Supercomputing Systems*, Steven and Svetlana Kartashev, Eds., Elsevier, 1986.
12. Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991. Proceedings of a Conference held at Oxford, June 1989.
13. Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992.
14. Joseph Goguen, Kai Lin, and Grigore Roşu. Circular coinductive rewriting. In *Automated Software Engineering '00*, pages 123–131. IEEE, 2000. Proceedings of a workshop held in Grenoble, France.
15. Joseph Goguen and William Tracz. An implementation-oriented semantics for module composition. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component-based Systems*, pages 231–263. Cambridge, 2000.
16. Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer, 2000.
17. Robert Goldblatt. *Topoi, the Categorial Analysis of Logic*. North-Holland, 1979.
18. Rosa Jiminénez and Fernando Orejas. An algebraic framework for higher-order modules. In Jeannette Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings, FM'99*, volume 1709 of *Lecture Notes in Computer Science*, pages 1778–1797. Springer, 1999.
19. Kai Lin and Joseph Goguen. Semantics and implementation of higher order parameterized programming. Technical Report CSE2002–0743, University of California at San Diego, July 2002.
20. David MacQueen and Mads Tofte. A semantics for higher-order functors. In *European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1994.
21. José Meseguer and Joseph Goguen. Initiality, induction and computability. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge, 1985.
22. Donald Sannella and Martin Wirsing. A kernel language for algebraic specification and implementation. In *Proceedings, Eleventh Colloquium on Foundations of Computation Theory*, pages 307–328. Springer, 1983. Lecture Notes in Computer Science, Volume 158.
23. William Tracz. LILEANNA: a parameterized programming language. In *Proceedings, Second International Workshop on Software Reuse*, pages 66–78, March 1993. Lucca, Italy.