Polytypic Programming in Maude

M. Clavel\textsuperscript{a,1}, F. Durán\textsuperscript{b,2}, and N. Martí-Oliet\textsuperscript{c,3}

\textsuperscript{a} Departamento de Filosofía, Universidad de Navarra, Spain
\textsuperscript{b} ETSII, Universidad de Málaga, Spain
\textsuperscript{c} Facultad de Matemáticas, Universidad Complutense, Madrid, Spain

Abstract
The idea of polytypic programming is to write programs that are defined by induction on the structure of user-defined datatypes. In this way, many functions with similar functionalities do not have to be written over and over again for different datatypes. So far, this programming style has been developed in functional languages like Haskell, extended with new syntactic constructs for defining polytypic programs. In this paper we show that polytypic programming can be reduced to metaprogramming, and that can be developed in a reflective first-order language like Maude, without having to extend the language. This has the additional advantage of allowing us to use standard formal tools to prove properties about polytypic programs. We illustrate our methodology via examples. In particular, we explain how to define in Maude two non-trivial generic functions, namely, the polytypic versions of the functions \textit{map} and \textit{cata}, and how to prove properties about them using an inductive theorem prover for Maude modules.

1 Introduction
Polytypic programming. The idea of generic programming \cite{1} is to write programs that solve a class of problems once and for all, instead of writing new code over and over again for each different instance. Advantages of generic programming include the greater potential for reuse and the increased reliability, since generic programs in which irrelevant details are abstracted away are often easier to construct and reason about. Two well-known techniques for generic programming are parameterized programming \cite{13} and polymorphic programming \cite{20}. A more recent development of generic programming concerns functions that are defined by induction on the structure of user-defined datatypes.

\textsuperscript{1} Partially supported by DARPA and NASA through contract NAS2-98073.
\textsuperscript{2} Partially supported by CICYT, TIC98-0445-C03-03.
\textsuperscript{3} Partially supported by CICYT, TIC98-0445-C03-02.

©2000 Published by Elsevier Science B. V. Open access under CC BY-NC-ND license.
Consider for example the function \( length : \text{List}[X] \rightarrow \text{Int} \), which counts the number of elements of type \( X \) in a list. There exists a similar function \( length : \text{BinTree}[X] \rightarrow \text{Int} \), which counts the number of occurrences of elements of type \( X \) in a binary tree. In fact, there exists a generic function \( length : D[X] \rightarrow \text{Int} \), that can be applied to any (free) datatype constructor \( D[\_] \), that is, the generic version of the function \( length \) is not only parametric in \( X \), but also in the datatype constructor \( D[\_] \). In \([17,1]\) functions like \( length \) that are parametric at this more abstract level are called polytypic functions.

**PolyP.** There exist various ways to implement polytypic programs in functional programming: using a universal datatype, using higher-order polymorphism and constructor classes, and using special syntactic constructs. Jeuring and Jansson argue in \([17]\) that the first two solutions are unsatisfactory, and propose to extend the Haskell programming language so that polytypic functions can be implemented. The extended language is called PolyP \([17,16]\). The extension introduces a new kind of definition, the polytypic construct, that is used to define functions by induction over pattern functors, which describe the structure of (a subset of) regular datatypes. The idea is that PolyP extracts the pattern functor from a datatype definition in Haskell and uses this structure information to instantiate generic programs on particular datatypes.

**Reflection and polytypic programming.** In a reflective programming language \([3]\) its modules can be represented as data at the metalevel, and can be passed as arguments to metalevel functions that can transform them in many different ways. These functions are called metaprograms. In this paper, we argue that polytypic programming can be reduced to metaprogramming in a simple and principled way. The idea is that a metalevel function that takes the representation of a module as one of its arguments has direct access to the structure of the datatypes defined in this module. This is all is needed in order to define polytypic functions. We propose then to define polytypic functions as metalevel functions that extend modules with the functions that correspond to their instantiations for particular datatypes.

By reducing polytypic programming to metaprogramming, we avoid having to extend the original language with new syntactic constructs, with the consequent implementation cost. Moreover, we are entitled to apply standard theorem proving tools, that may be available for the original language, when proving properties of generic programs, since they are after all standard programs.

**Maude.** Maude \([5,6]\) is a logical language based on rewriting logic. Its implementation has been designed with the explicit goals of supporting algebraic executable specifications and formal methods applications, of being easily extensible, and of supporting reflective computations. All these features make Maude an ideal candidate for testing the viability of our proposal. In this paper, we use the reflective capabilities of Maude to define in Maude generic functions as metalevel functions. Note that, unlike PolyP with respect to Haskell, we do not extend Maude with new constructors in order to write
polytypic programs; using reflection, we write polytypic programs directly in Maude.

**Proving properties about polytypic programs.** One additional advantage of our reflective declarative methodology is the possibility of using available standard theorem proving tools when proving properties about polytypic programs. Inductive theorem provers are particularly suited for reasoning over polytypic programs, since these are functions defined by induction on the structure of user-defined datatypes. We report in this paper on our experience using an inductive theorem prover for Maude—called the ITP tool [7]—to prove properties about polytypic programs written in Maude as metalevel functions.

2 Metaprogramming in Maude

In this section we review background material about the Maude language and its reflective capabilities for metaprogramming.

2.1 *The Maude Language*

Maude [5] is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Exploiting the fact that rewriting logic and equational logic are reflective [3,9], a key distinguishing feature of Maude is its systematic and efficient use of reflection, a feature that makes Maude remarkably extensible and powerful, and that allows many advanced metaprogramming applications [4,8]. In fact, we will use these reflective capabilities in order to write generic programs in Maude.

The most general Maude modules are called *system modules*. They have the syntax \texttt{mod T endm} with \( T \) a rewrite theory. Since equational logic is a sublogic of rewriting logic, Maude contains also a sublanguage of *functional modules* of the form \texttt{fmod E endfm} with \( E \) a Church-Rosser and terminating equational theory. Maude’s functional sublanguage is based on membership equational logic [19], an extension of order-sorted equational logic, that supports sorts, subsort relations, subsort polymorphic overloading of operators, and definition of partial functions with equationally defined domains. A system module \texttt{mod T endm} specifies the initial model of the rewrite theory \( T \). Similarly, a functional module \texttt{fmod E endfm} specifies the initial algebra of the equational theory \( E \) [5].

2.2 *Reflection in Maude*

Maude’s language design and implementation make systematic use of the fact that rewriting logic and equational logic are reflective [3,9], in the specific sense that its metalevel can be represented at the object level in a theory that is called *universal*; two metalevel notions that can be so represented are theories
and the entailment relation. Since the universal theory is representable in itself, a “reflective tower” can be achieved with an arbitrary number of levels of reflection. Maude’s reflective capabilities are directly based on these results. In particular, key functionalities of the universal theory for rewriting logic have been efficiently implemented in a functional module \texttt{META-LEVEL} where, in particular

(i) Maude terms are reified as elements of a sort \texttt{Term} of terms;
(ii) Maude modules are reified as terms in a sort \texttt{Module} of modules;
(iii) There are a number of built-in operations for metaevaluation that provide useful and efficient ways of reducing metalevel computations to object-level ones.

The module \texttt{META-LEVEL} provides sorts and constructors for metarepresenting each of the declarations that can appear in modules, and for terms appearing in equations, membership axioms, and rules. In particular, terms are metarepresented in a prefix form, all identifiers are quoted, constants are annotated with the corresponding sort, and parentheses become square brackets.

Functional and system modules are then metarepresented in a syntax very similar to their original user syntax. The main differences are that: (1) terms in equations, membership axioms, and rules are metarepresented as explained before; (2) in the metarepresentation of modules a fixed order is followed in introducing the different kinds of the declarations for sorts, subsorts, variables, equations, etc., whereas in the user syntax there is considerable flexibility for introducing such different declarations in an interleaved and piecemeal way; and (3) sets of identifiers—used in declarations of sorts—are represented as sets of quoted identifiers built with an associative and commutative operation \_\_\_. We refer the reader to \cite{4,5} for more details.

\subsection{2.3 Metaprogramming in Maude}

Metaprogramming is programming at the metalevel, or in other words, writing programs that manipulate other programs as data. In the case of Maude, metalevel functions operate on metarepresentations of lower-level rewrite theories. The idea is not new, but its realization in Maude has a clear logical semantics by defining such functions by means of rewrite theories that extend the universal theory of rewriting logic with new datatypes and new functions. In this way, one can reason formally about the correctness of metaprogramming constructions by analyzing their associated rewrite theories.

The largest metaprogramming application developed in Maude so far is Full Maude \cite{10}. Full Maude extends the \texttt{META-LEVEL} with new datatypes—extending its \texttt{Module} sort to richer sorts for structured and parameterized modules—and with new module operations—such as instantiations of parameterized modules by views, flattening of module hierarchies into single modules,
and so on. In particular, Full Maude supports an OBJ style of parameterized programming [15,13], with highly generic and reusable modules. Other interesting non-trivial metaprogramming applications developed in Maude include user-definable strategy languages, development of theorem proving tools [7], and representation of other languages and logics within rewriting logic; see [8] for more information about these applications.

2.4 The ITP Tool: An Inductive Theorem Prover

The ITP [7] is a formal tool—entirely written in Maude—to prove inductive properties about Maude modules. In fact, we have used this tool in the experimental work described in Section 4.

The ITP tool has a simple, reflective design that takes full advantage of both the good properties of rewriting logic as a logical framework [18], and the reflective capabilities of Maude. Figure 1 illustrates this novel design. The module T, about which we want to prove inductive theorems, is at the object level. An inference system I for inductive proofs uses T as data and therefore is specified as a module ITP at the metalevel. In particular, ITP encodes syntax and proof rules for first-order logic as well as induction principles for reasoning about Maude modules. Finally, different proof tactics to guide the application of the rewrite rules specifying the inference rules in I are strategies, which are defined at the meta-metalevel in a module S-ITP. See [5,3] for details on defining strategy languages in Maude.

Operationally, to use the ITP tool, the user submits as an initial goal the pair formed by (the metarepresentation of) a functional module and (the representation of) the first-order sentence over its signature that is to be proved, and then this goal will be successively transformed by rewriting—using the inference rules as rewrite rules—into different sets of subgoals, until (in the case of a successful proof) no subgoals are left. The application of the inference rules as rewrite rules is controlled by the user with strategies.

---

4 http://sophia.unav.es/~clavel provides the most recent version of this tool.
3 Polytypic Programming in Maude

We propose to use the reflective capabilities of Maude to write polytypic programs in Maude. The idea is to reduce polytypic programming to metaprogramming, so that a polytypic program is a metalevel function in Maude that transforms a Maude module into the Maude module that corresponds to the instantiation of the generic program under consideration. To illustrate this general methodology, we show below how to write polytypic functions such as map or cata in Maude.

Polytypic functions have in common that they are defined by induction on the structure of user-defined datatypes. Since, at the metalevel, the structure of user-defined datatypes can be accessed by looking at the term metarepresenting the module in which they are specified, polytypic functions can be easily defined as metalevel functions. They will take as one of its arguments the metarepresentation of a module, and will return the metarepresentation of the module that corresponds to the instantiation of the polytypic function under consideration, that is, the metarepresentation of the original module extended with the equations that define the polytypic function for particular datatypes.

In this presentation, we use the following convention for naming sorts: a sort can be either an identifier or an expression of the form $G_1[...G_k[I]...]$, where $G_i, I$ are identifiers, for $i = 1, ... , k$. We call $G_1[...G_k[I]...]$ a nested sort. The idea is that nested sorts represent parameterized sorts.

For example, a functional module defining lists of quoted identifiers is introduced as follows:

```
fmod LIST is
  protecting QID .
  sort List[Qid] .
  op [] : -> List[Qid] [ctor] .
  op _:_ : Qid List[Qid] -> List[Qid] [ctor] .
endfm
```

Recall that the attribute `ctor` is used in Maude to mark the constructors of a sort. Similarly, a functional module defining binary trees of lists of quoted identifiers and binary trees of lists of integers is introduced as follows:

```
fmod BINTREE-LIST is
  protecting QID .
  protecting MACHINE-INT .
  sorts List[Qid] List[MACHINE-INT]
      BinTree[List[Qid]] BinTree[List[MACHINE-INT]] .
  *** binary trees of lists of quoted identifiers
  op [] : -> List[Qid] [ctor] .
  op _:_ : Qid List[Qid] -> List[Qid] [ctor] .
endfm
```

5 The complete specification, a number of running examples, and some documentation can be found in [http://sophia.unav.es/~clavel/generic_programming.html](http://sophia.unav.es/~clavel/generic_programming.html)
op empty : BinTree[List[Qid]] [ctor] .

op _[_]_ : BinTree[List[Qid]] List[Qid] BinTree[List[Qid]]

-> BinTree[List[Qid]] [ctor] .

*** similarly for binary trees of lists of integers

... endfm

We define below the polytypic functions map and cata as two metalevel functions, map and cata, extending the module META-LEVEL. These definitions illustrate how polytypic programming can be reduced to metaprogramming in Maude, without having to extend the language. We use in Section 4 these definitions to show another advantage of our reflective methodology, namely, the possibility of using standard theorem proving tools when proving properties about polytypic programs. In Section 5 we describe an extension of Full Maude with the map polytypic function, so that this function appears to the user as part of the parameterized programming facilities available in Full Maude; internally, however, map is implemented as a first-order metalevel function analogous to the function map introduced below.

For the sake of simplicity, we define the polytypic versions of map and cata for modules whose sorts are parameterized at most by one parameter, but this restriction is lifted in the Full Maude version of map. In what follows, we use the overline symbol, $\overline{M}$, to denote the metarepresentation of modules, sorts, terms, operations, and so on.

3.1 A Generic Function map

The function map is one of the most well-known higher-order functions in functional programming. Usually its arguments are a function $f$ and a list $[e_1, e_2, \ldots, e_n]$, and its result is the list $[f(e_1), f(e_2), \ldots, f(e_n)]$ obtained by applying the given function to each of the elements in the original list. From a categorical point of view, the list construction defines a functor on sets, and the function map corresponds to the functor application on morphisms. The same idea can be applied to other datatypes; for example, in the case of binary trees with information of type $A$ in all the nodes, the corresponding map applies a given function of type $A \to B$ to the data in each node, obtaining in this way a binary tree over $B$. Therefore, in general, the function map is a generic function that can be instantiated in completely similar ways for different datatype constructors.

Using reflection, we can specify the generic function map as a metalevel function $\overline{\text{map}}(\overline{M}, f, \overline{S}, \overline{S}', \overline{P}, \overline{P}')$ that adds to $M$ the equations defining the function $\text{map}.f : S \to S'$ by structural induction over constructors (specified by means of the attribute ctor), where $S = G_1[\ldots G_k[P] \ldots]$ and $S' = G_1[\ldots G_k[P'] \ldots]$ are two parameterized sorts declared in $M$, and $f : P \to P'$ is a function also declared in $M$.

Specifically, the function map first adds to $M$ the equation
eq map.f(x) = f(x),

with x a variable of (parameter) sort P. Then, for each constant of sort S in M, op c : -> S [ctor], map adds the equation

eq map.f(c) = c,

and, for each n-ary constructor d of sort S in M, op d: S_1...S_n-> S [ctor], with n ≥ 1, map adds the equation

eq map.f(d(x_1,...,x_n)) = d(map.f(x_1),...,map.f(x_n)),

with x_i a variable of sort S_i, for 1 ≤ i ≤ n.

Finally, notice that, for each n-ary constructor d of sort S in M, op d: S_1...S_n-> S [ctor], with n ≥ 1, S_i can be either (i) a sort that is not parameterized by P and is different from P, or (ii) a sort that is parameterized by P. For each of such sorts S_i, if (i) is the case, then map adds the equation

eq map.f(x_i) = x_i,

with x_i a variable of sort S_i; and, if (ii) is the case, say S_i = H_1[...H_j[P]...], then map is recursively called in order to add to M the equations that specify the function map.f: H_1[...H_j[P]...] -> H_1[...H_j[P]...].

As an example, let us assume that foo: Qid-> MachineInt is a function defined in BINTREE-LIST. Then,

Maude> red map(BINTREE-LIST, foo, BinTree[List[Qid]], BinTree[List[MachineInt]], Qid, MachineInt)

returns the metarepresentation of the module that results from adding to BINTREE-LIST the following equations:

var E : Qid . var L : List[Qid] .
vars BT1 BT2 : BinTree[List[Qid]] .
eq map.foo(E) = foo(E) .
eq map.foo(empty) = empty .
eq map.foo(BT1 [L] BT2) = map.foo(BT1) [map.foo(L)] map.foo(BT2) .
eq map.foo([]) = [] .
eq map.foo(E : L) = map.foo(E) : map.foo(L) .

3.2 A Generic Function cata

The basic example of catamorphism is another well-known higher-order function over lists: foldr, also known as reduce. Basically, given a constant c, a binary operation ⊗, and a list [e_1, e_2, ..., e_n], the result of foldr over these arguments is e_1 ⊗ (e_2 ⊗ (... ⊗ (e_n ⊗ c)...)). For example, foldr + 0 [m_1, ..., m_n] returns Σ_i=1^n m_i, while foldr * 1 [m_1, ..., m_n] returns Π_i=1^n m_i for lists over some number domain. The foldr function can be used to define typical functions over lists like append, concatenation of a list of lists, and also map. Basically, as the definition shows, foldr for lists substitutes the two typical constructors for lists with two functions of appropriate ranks. This can be generalized by
applying a given function to the elements of the lists, as we will do below. From a categorical point of view, \texttt{foldr} is the construction over morphisms that guarantees that the list construction is free. Therefore, the same idea can be applied to other datatype constructors; for example, for binary trees over lists we have the notion of flattening, which is completely similar to the concatenation of a list of lists.

In general, a catamorphism is a function that replaces constructors by functions of appropriate ranks. As we have done with \texttt{map}, using reflection we can specify a generic function \texttt{cata} for defining certain classes of catamorphisms. This metalevel function \texttt{cata(M, f, S, S', P, t, g, h, t')} adds to \texttt{M} the equations that define the catamorphism \texttt{cata.f : S \rightarrow S'} by structural induction over constructors, where \(S = G_1[...G_k[P]...]\) and \(S'\) are two (parameterized) sorts declared in \(M\), \(t\) and \(t'\) are two terms of sort \(S'\) in \(M\), and \(g : S' \times S' \rightarrow S'\) and \(h : P \rightarrow S'\) are two functions declared in \(M\). Notice that the second argument of the metalevel function \texttt{cata} is used to give name to the resulting object function defining the catamorphism over the given datatypes.

Specifically, the function \texttt{cata} first adds to \(M\) the equation
\[
\text{eq cata.f(x) = h(x)},
\]
with \(x\) a variable of (parameter) sort \(P\). Then, for each constant of sort \(S\) in \(M\), \(\text{op } c : \rightarrow S \ [\text{ctor}]\), \texttt{cata} adds the equation
\[
\text{eq cata.f(c) = t},
\]
and, for each \(n\)-ary constructor \(d\) of sort \(S\) in \(M\), \(\text{op } d : S_1 S_2 \ldots S_{n-1} S_n \rightarrow S \ [\text{ctor}]\), with \(n \geq 1\), \texttt{cata} adds the equation
\[
\text{eq cata.f( d(x_1, x_2, \ldots, x_{n-1}, x_n) ) = g(cata.f(x_1), g(cata.f(x_2), \ldots, g(cata.f(x_{n-1}), cata.f(x_n)) \ldots))},
\]
with \(x_i\) a variable of sort \(S_i\), for \(1 \leq i \leq n\).

Finally, for each sort \(S_i\) in a constructor declaration \(\text{op } d : S_1 \ldots S_n \rightarrow S \ [\text{ctor}]\), if \(S_i\) is a sort that is not parameterized by \(P\) and is different from \(P\), then the function \texttt{cata} adds the equation
\[
\text{eq cata.f(x_i) = t'},
\]
with \(x_i\) a variable of sort \(S_i\); and if \(S_i\) is a sort that is parameterized by \(P\), then \texttt{cata} is recursively called to add to \(M\) the equations that specify the function \texttt{cata.f : S_i \rightarrow S'}.

The polytypic function \texttt{size} is an example of catamorphism that can be defined with our function \texttt{cata}. The function \texttt{size} takes a term \(x\) of a parameterized sort whose parameter is \(a\) and counts the number of occurrences of terms of sort \(a\) in the term \(x\). Let us assume that \texttt{one : Qid -> MachineInt} is a constant function defined in \texttt{BINTREE-LIST} that returns always 1. Then,

\texttt{Maude} \texttt{> red cata(BINTREE-LIST, size, BinTree[List[Qid]], MachineInt, Qid, 0, +, one, 0).}
returns the metarepresentation of the module that results from adding to BINTREE-LIST the equations below, specifying the function \textit{size} over binary trees of lists of identifiers:

\begin{verbatim}
var E : Qid . var L : List[Qid] .
vars BT1 BT2 : BinTree[List[Qid]] .
eq cata.size(E) = one(E) .
eq cata.size(empty) = 0 .
eq cata.size(BT1 [L] BT2)
  = cata.size(BT1) + cata.size(L) + cata.size(BT2) .
eq cata.size([]) = 0 .
eq cata.size(E : L) = cata.size(E) + cata.size(L) .
\end{verbatim}

The polytypic function \textit{flatten} is another example of catamorphism that can be defined with the function \textit{cata}. The function \textit{flatten} takes a term \(x\) of a parameterized sort whose parameter is \(a\), and flattens it into a list of terms of sort \(a\). Let us assume that \textit{list}:Qid→List[Qid] is a function defined in BINTREE-LIST that takes a quoted identifier and returns the list whose only element is this quoted identifier, and that \textit{append}:List[Qid] List[Qid]→List[Qid] is also defined in BINTREE-LIST. Then,

\begin{verbatim}
Maude> red cata(BINTREE-LIST, flatten,
  BinTree[List[Qid]], List[Qid],
  Qid, [], append, List, [])
\end{verbatim}

returns the metarepresentation of the module that results from adding to BINTREE-LIST the equations below, specifying the function \textit{flatten} over binary trees of lists of identifiers:

\begin{verbatim}
eq cata.flatten(E) = list(E) .
eq cata.flatten(empty) = [] .
eq cata.flatten(BT1 [L] BT2)
  = append(cata.flatten(BT1),
          append(cata.flatten(L), cata.flatten(BT2))) .
eq cata.flatten([]) = [] .
eq cata.flatten(E : L) = append(cata.flatten(E), cata.flatten(L)) .
\end{verbatim}

4 Proving Properties of Polytypic Programs

In Section 3 we have shown that, by reducing polytypic programming to metaprogramming, we avoid having to extend a declarative first-order language like Maude in order to write polytypic programs. In this section we illustrate another advantage of our reflective declarative methodology, namely, the possibility of using available standard theorem proving tools when proving properties about polytypic programs. Inductive theorem provers are ideally suited for reasoning over polytypic programs, since these are functions defined by induction on the structure of user-defined datatypes. We report here on our experience using the ITP tool, an inductive theorem prover for Maude func-

10
tional modules [7], to prove properties of the polytypic functions map and cata.

4.1 Functoriality of map

Our first example is the formalization and proof of the functoriality of map over lists. Let \( g: \text{Elem}_1 \to \text{Elem}_2 \) and \( f: \text{Elem}_2 \to \text{Elem}_3 \) be arbitrary functions. Then,

\[
(1) \quad \text{map} \ f \ (\text{map} \ g \ (x)) = \text{map} \ (f \cdot g) \ (x),
\]

and

\[
(2) \quad \text{map} \ \text{id} \ (x) = x,
\]

where \( x \) is a variable over lists of elements of type \( \text{Elem}_1 \), and \( \text{id} \) is the identity function over those elements. The functoriality of map is typically proved by structural induction on the range of \( x \), that is, on lists, and our proof will mirror this standard inductive proof. We focus here on the proof of (1). The proof of (2) is analogous.

4.1.1 Formalization

We first introduce a parameterized module LIST for specifying lists, with parameters displayed in italics. The module LIST declares also two functions \( f \) and \( g \) over elements of lists, and a function \( f \cdot g \) that is defined as the composition of \( f \) and \( g \).

\[
\text{fmod LIST is}
\]

\begin{verbatim}
  op nil : -> List[Elem_1] [ctor] .
  op cons : Elem_1 List[Elem_1] -> List[Elem_1] [ctor] .
  op nil : -> List[Elem_2] [ctor] .
  op nil : -> List[Elem_3] [ctor] .
  op g : Elem_1 -> Elem_2 .
  op f : Elem_2 -> Elem_3 .
  op f \cdot g : Elem_1 -> Elem_3 .
  var E : Elem_1 .
  eq f \cdot g \ (E) = f \ (g \ (E)) .
\end{verbatim}

Notice that the notion of parameterized modules that we use here is different from the one implemented in Full Maude [5,10]. The module LIST above is parameterized not only by the sorts \( \text{Elem}_1, \text{Elem}_2, \) and \( \text{Elem}_3 \), but also by (the names of) the constructors for lists, and (the names of) the functions \( f, g \), and \( f \cdot g \). This extra parameterization is very convenient for our purposes here, since (1) and (2) hold, of course, for any adequate specification of lists and any pair of functions over elements of lists of the appropriate ranks, independently of the names of the constructors for lists, or the names of the
functions. A formal definition of parameterized modules in the sense used here, and of its metarepresentation in the module META-LEVEL, is given in [2]. For the purpose of understanding this example, it is sufficient to know that the metarepresentation of parameterized modules is defined in such a way that their parametric nature is preserved at the metalevel. This is accomplished by replacing—in the term metarepresenting the parameterized module—any parameter by a variable, with the same name, of the appropriate sort. In our case, all the parameters will be replaced by variables of sort Qid.

We can now rephrase theorem (1) as follows. Let MAP-LIST be the module that extends LIST with the definitions of the functions map.g, map.f, and map.f · g, that is, MAP-LIST is the module that results from applying at the metalevel the function map over the module LIST, with the appropriate arguments. Then, the equality

\[ \text{map.f (map.g (l)) = map.(f \cdot g)(l)} \]

holds in MAP-LIST, for any term \( l \) of sort List[Elem] in LIST. Thus, (1) can not be a theorem about LIST—since (3) must be proved for functions defined in MAP-LIST—, neither can be a theorem about MAP-LIST—since (3) must be proved for terms defined in LIST. Notice that (1) is rather a metatheorem that relates the modules LIST and MAP-LIST. This is not surprising, since (1) states a property of the polytypic function map and, in our reflective setting, map is defined as a metalevel function map that relates two different modules in a precise way. Therefore, we formalize (1) as a theorem about the extension of META-LEVEL in which map is defined; our theorem formalizes then the relation stated by (1) between the modules LIST and MAP-LIST, namely, that (3) holds in MAP-LIST for any term of sort List[Elem] in LIST.

Let POLYTYPIC be an extension of META-LEVEL that includes:

- two Boolean functions, (\(_\cdot \_\text{in}\_\) and (\(_= \_\text{in}\_\), that reflect at the metalevel, respectively, the membership and the equality relations in membership equational logic\(^6\).
- the polytypic functions map and cata.

We claim that the following formula formalizes (1) as a theorem about the initial model of POLYTYPIC:

\[ \forall(Elem_1, Elem_2, Elem_3, nil, cons, g, f, f \cdot g):\text{Qid.}\forall(x):\text{Term.} \]

\[ (x: \text{List}[Elem_1] \text{ in LIST = true} \]

\[ \Rightarrow \text{map.f ['map.g [x]] = 'map.f \cdot g [x] in MAP-LIST = true)}, \]

where MAP-LIST abbreviates the term

\[ \text{map(map(map(LIST, g, 'List[Elem_1], 'List[Elem_2], Elem_1, Elem_2), f, 'List[Elem_2], 'List[Elem_3], Elem_2, Elem_3), f \cdot g, 'List[Elem_1], 'List[Elem_3], Elem_1, Elem_3)}, \]

\(^6\) In this experimental phase of our work, we have defined (\(_\cdot \_\text{in}\_\) and (\(_= \_\text{in}\_\) only for a restricted subclass of functional modules.
that is, \( \text{MAP-LIST} \) denotes the module that extends \( \text{LIST} \) with the definitions of the functions \( \text{map} \cdot g \), \( \text{map} \cdot f \), and \( \text{map} \cdot f \cdot g \).

### 4.1.2 Proof

We sketch here how we prove (4). Note that our proof mirrors the standard proof of this property by structural induction.

In general, when proving properties of object functions, the ITP tool uses an inductive inference rule \((\text{ind})\) that formalizes induction over the stages of the inductive definitions of sorts; see [2] for a formal presentation of this rule. However, this simple induction principle is not enough when proving properties about metalevel functions extending the module \( \text{META-LEVEL} \) such as theorem (1). This theorem is typically proved by structural induction on the range of \( x \), that is, on the sort of lists. In our reflective setting, however, this theorem is formalized as a theorem about the module \( \text{POLYTYPIC} \), in particular about the metalevel function \( \text{map} \) when it is applied to a module specifying lists. Notice that in our formalization—theorem (4)—the variable \( x \) ranges over terms of sort \( \text{Term} \). Thus, if we apply the inductive inference rule \((\text{ind})\) on \( x \), the cases that will be generated are those corresponding to the constructors of the sort \( \text{Term} \), but the only terms of sort \( \text{Term} \) that are interesting in this case are those that metarepresent terms representing lists in any instance of the module \( \text{LIST} \).

To prove this and similar inductive theorems about extensions of the module \( \text{META-LEVEL} \), the ITP tool uses an inductive inference rule \((\text{ind})\) that reflects up the induction principle \((\text{ind})\); see again [2] for the formal presentation. The crucial observation that allows to mirror inductive reasoning over a sort in a module by inductive reasoning over \( \text{META-LEVEL} \) is the following: the set of terms metarepresenting terms of a sort \( s \) in a module \( M \) is inductively defined in a way that reflects at the metalevel the inductive definition of \( s \) in \( M \).

Thus, to prove (4) we apply the reflected version of the induction principle for the parameterized sort \( \text{List}[\text{Elem}] \) in the module \( \text{LIST} \), that is, the corresponding instance of the inference rule \((\text{ind})\). This reduces proving (4) to proving the formula given in Figure 2. Notice that the two conjuncts correspond to the cases involved in proving (1) by structural induction. The first conjunct formalizes the case when \( x \) is the empty list, and the second conjunct formalizes the case when \( x \) is a list formed by an element followed by a list.

After applying the constants lemma for membership equational logic [19], both conjuncts are easily proved using the fact that the Boolean function \( \_\_\text{in}_\_ \) reflects at the metalevel the equality relation in any module—in our case, in the corresponding instance of \( \text{MAP-LIST} \). Of course, to prove the second conjunct we need to use, in addition, the induction hypothesis.
∀(Elem₁, Elem₂, Elem₃, nil, cons, g, f, f · g):Qid.
('map.f ['map.g [nil]] = 'map.f · g [nil] in MAP-LIST = true) ∧
∀(E, L):Term.
∀(Elem₁, Elem₂, Elem₃, nil, cons, g, f, f · g):Qid.
(E: Elem₁ in LIST = true ∧
L: 'List[Elem₁] in LIST = true ∧
'map.f ['map.g [L]] = 'map.f · g [L] in MAP-LIST = true)
⇒
∀(Elem₁, Elem₂, Elem₃, nil, cons, g, f, f · g):Qid.
('map.f ['map.g [cons [E, L]]] = 'map.f · g [cons [E, L]]
in MAP-LIST = true))

Fig. 2. Goal resulting after induction.

4.2 A General Property of map and cata

Our second example is the formalization and proof of a general property of map and cata over lists. As before, let g : Elem₁ → Elem₂ and f : Elem₂ → Elem₃ be arbitrary functions. Let us now assume that 0 is a constant in Elem₃, and that + is a binary operation on Elem₃. Then,

(5) cata f (map g (x)) = cata (f · g) (x),

where x is a variable for lists of elements of type Elem₁, and cata f denotes the catamorphism on lists obtained in the obvious way from +, 0 and f (similarly for cata (f · g)). This theorem is typically proved by structural induction on the range of x, and our proof will mirror this standard inductive proof.

4.2.1 Formalization

We assume that the parameterized module LIST has been extended with the following declarations:

\[
\begin{align*}
\text{op } 0 & : \to \text{Elem}_3. \\
\text{op } + & : \text{Elem}_3 \ \text{Elem}_3 \to \text{Elem}_3.
\end{align*}
\]

We claim that the following formula formalizes (5) as a theorem about the initial model of POLYTYPIC:

∀(Elem₁, Elem₂, Elem₃, nil, cons, 0, +, g, f, f · g):Qid.(x: Term.

(x: 'List[Elem₁] in LIST = true 
⇒ 'cata.f ['map.g [x]] = 'cata.f · g [x] in CATA-LIST = true),

where CATA-LIST abbreviates the term

cata(cata(map(LIST, g, 'List[Elem₁], 'List[Elem₂], Elem₁, Elem₂), f, 'List[Elem₂], Elem₃, Elem₂, 0, +, f, 0), f · g, 'List[Elem₂], Elem₃, Elem₁, 0, +, f · g, 0),

that is, CATA-LIST denotes the module that extends LIST with the definitions of the functions map.g, cata.f, and cata.f · g.
∀(Elem₁, Elem₂, Elem₃, nil, cons, 0, +, g, f, f · g):Qid.
(⌜'cata.f [⌜'map.g [nil]⌟] = 'cata.f · g [nil] in CATA-LIST = true⌟)
∧
∀(E, L):Term.
[⌜∀(Elem₁, Elem₂, Elem₃, nil, cons, 0, +, g, f, f · g):Qid.
(E:Elem₁ in LIST = true ∧
L:'List[Elem₁] in LIST = true ∧
⌜'cata.f [⌜'map.g [L]⌟] = 'cata.f · g [L] in CATA-LIST = true⌟)
⇒
∀(Elem₁, Elem₂, Elem₃, nil, cons, 0, +, g, f, f · g):Qid.
(⌜'cata.f [⌜'map.g [cons [E,L]]⌟] = 'cata.f · g [cons [E,L]]
in CATA-LIST = true⌟)
]

Fig. 3. Goal resulting after induction.

4.2.2 Proof
We sketch here how we prove (6), by following the corresponding standard
proof by structural induction. As before, we apply the reflected version of the
induction principle for the parameterized sort List[Elem₁] in the module
LIST. This reduces proving (6) to proving the formula given in Figure 3. As
expected, the two conjuncts correspond to the cases involved in proving (5)
by structural induction.

To complete the proof we use the constants lemma for membership equa-
tional logic [19] and the fact that the Boolean function _=_ in reflects at the
metalevel the equality relation in any module—in our case, in the correspond-
ing instance of CATA-LIST. In addition, to prove the second conjunct we need
to use the induction hypothesis.

5 An Interface in Full Maude for Polytypic Programs
We describe a simple extension of Full Maude implementing the polytypic
function map. First, we recall that parameterized modules in Full Maude use
theories to specify the requirements that the parameter must satisfy, so that
the corresponding instantiations make sense; see [10,5,6] for more details. The
simplest theory is the one simply requiring the existence of a sort:

fth TRIV is
    sort Elt .
endfth

The theory TRIV can then be used in a parameterized module as the one
below specifying lists over an arbitrary set.

fmod LIST[X :: TRIV] is
    protecting NAT .
    sorts NeList[X] List[X] .
    subsort NeList[X] < List[X] .
    op [] : -> List[X] [ctor] .
op length : List[X] -> Nat .
var E : Elt.X . var L : List[X] .
eq length([]) = 0 .
eq length(E : L) = succ(0) + length(L) .
endfm

To instantiate a parameterized module, we use another module that satisfies the requirement theory via a view. For example, by means of the view

view Nat from TRIV to NAT is
  sort Elt to Nat .
endv

from the theory TRIV to the module NAT we can obtain the instantiated module LIST[Nat], specifying lists over natural numbers.

In order to define a polytypic function map in Full Maude, we have used the methodology described in [10,11]. In particular, we have added to Full Maude a new module operation MAP(\_), which takes a module name as argument and returns a module with the appropriate map function defined in it. Currently, we assume that the module given as argument either is not parameterized (in which case the map functions are identities), or has only one or more instances of the theory TRIV as parameters. The module that results from solving a module expression built with MAP(\_) remains parameterized with respect to the following MAP theory, with as many MAP parameters as TRIV parameters in the argument module. Notice that the theory MAP makes the corresponding map function parametric with respect to the function to be mapped.

fth MAP is
  sorts A B .
  op f : A -> B .
endfth

The result of applying the module operation MAP(\_) to the module LIST above is the following parameterized module.

fmod MAP(LIST)[F :: MAP] is
  protecting LIST[A][F] .
  protecting LIST[B][F] .
  protecting MAP(NAT) .
  protecting MAP(BOOL) .
  var X : A.F . var L : List[A][F] .
  var N : NeList[A][F] .
  eq map.List(N) = map.NeList(N) .
  eq map.List([]) = [] .
  eq map.NeList(X : L) = f(X) : map.List(L) .
endfm
Notice that the module operation \( \text{MAP}(_) \) is recursively applied to all the modules imported by \( \text{LIST} \). In this case, those modules are \( \text{NAT} \) and \( \text{BOOL} \) (implicitly imported by default), and the resulting \( \text{map} \) functions for the sorts \( \text{Nat} \) and \( \text{Bool} \) are identities. Then, the module contains a copy of \( \text{LIST}[A] \), that is, lists over the source of the function \( f \) given in the theory \( \text{MAP} \), and another copy of \( \text{LIST}[B] \), that is, lists over the target of \( f \). Finally, there is a \( \text{map} \) function for each sort in the module, and the corresponding equations follow exactly the structure of the constructors defining the sorts, including the case for the subsort declaration. This module is automatically generated and saved in the Full Maude database of modules, so that the user does not even need to see its contents. Of course, since the module is parameterized, it is used by means of instantiations. For example, after introducing in Full Maude the following view mapping \( f \) to the successor function \( s \) on natural numbers,

\[
\text{view succ from MAP to NAT is}
\begin{align*}
\text{sort A to Nat.} \\
\text{sort B to NzNat.} \\
\text{op f to s.}
\end{align*}
\]

we can have the following reduction:

\[
\text{Maude}\geq \text{(red-in MAP(LIST)[succ] : map.List(0 : s(0) : s(s(0)) : []).)}
\]
\[
\text{result NeList[B][succ] : s(0) : s(s(0)) : s(s(s(0))) : []}
\]

showing that the list \([0, 1, 2]\) is mapped to the list \([1, 2, 3]\).

As another example, if \( \text{BIN-TREE}[X :: \text{TRIV}] \) is a parameterized module specifying binary trees over an arbitrary set, with constructors \( \text{empty} \) and \( \_\_\_ \), we can reuse the same view to execute the following reduction:

\[
\text{Maude}\geq \text{(red-in MAP(BIN-TREE)[succ] : map.BinTree( (empty [0] (empty [s(0)] empty)) [s(s(0))]) (empty [s(0)] empty).)}
\]
\[
\text{result NeBinTree[B][succ] : (empty [s(0)] (empty [s(s(0))] empty)) [s(s(s(0)))][s(s(s(0)))][s(s(s(s(0))))] empty)}
\]

We can also generate appropriate \( \text{map} \) functions for nested parameterized datatypes. A \( \text{map} \) function for lists of binary trees is obtained by first defining a view from the theory \( \text{MAP} \) to the module \( \text{MAP(BIN-TREE)[succ]} \) obtained before.

\[
\text{view bin-tree-map-succ from MAP to MAP(BIN-TREE)[succ] is}
\begin{align*}
\text{sort A to BinTree[A][succ].} \\
\text{sort B to BinTree[B][succ].} \\
\text{op f to map.BinTree.}
\end{align*}
\]
Then, we can instantiate the parameterized module $\text{MAP}(\text{LIST})[F :: \text{MAP}]$ and have the following reduction:

Maude> (red-in MAP(LIST)[bin-tree-map-succ] :
   map.List( empty [s(0)] empty
       : (empty [0] (empty [s(0)] empty))
           [s(s(0))]
       (empty [s(0)] empty)
       : empty [0] (empty [s(0)] empty)
       : [] ).
result NeList[B][bin-tree-map-succ] :
   empty [s(s(0))] empty
   : (empty [s(0)] (empty [s(s(0))]) empty))
       [s(s(s(0)))]
   (empty [s(s(0))] empty)
   : empty [s(0)] (empty [s(s(0))]) empty
   : []

Analogously, with the view

view list-map-succ from MAP to MAP(LIST)[succ] is
   sort A to List[A][succ] .
   sort B to List[B][succ] .
   op f to map.List .
endv

we can define a module $\text{MAP}(\text{BIN-TREE})[\text{list-map-succ}]$ where we apply the successor function to each element of a binary tree of lists of natural numbers.

The nesting of datatypes over themselves, like lists of lists or trees of trees, overloads the constructors, but this is unproblematic. Consider as an example the case of a function over lists of lists of natural numbers that applies the length function to each list.

view length from MAP to LIST[Nat] is
   sort A to List[Nat] .
   sort B to Nat .
   op f to length .
endv

Maude> (red-in MAP(LIST)[length] :
   map.List( ( 0 : 0 : 0 : [] )
       : ( 0 : s(0) : [] )
       : ( 0 : s(0) : 0 : [] )
       : ( [] )
       : ( 0 : s(0) : s(0) : [] )
       : [] ).
result NeList[B][length] :
   s(s(s(0))) : s(s(0)) : s(s(s(s(0)))) : 0 : s(s(s(0))) : []

In all the previous examples, all the constructors are free, but this is not
always the case in Maude modules. For example, lists can also be defined by means of the associative concatenation operation as constructor. Notice as well the identity attribute declaring nil as neutral element for concatenation.

\[
\text{fmod LIST}[X :: \text{TRIV}] \text{ is }
\begin{align*}
\text{sorts } & \text{NeList}[X] \ \text{List}[X] . \\
\text{subsort } & \text{Elt}[X] < \text{NeList}[X] < \text{List}[X] . \\
\text{op } & \text{nil : } -> \text{List}[X] [\text{ctor}] . \\
\text{op } & \_ \_ : \text{List}[X] \ \text{List}[X] -> \text{List}[X] [\text{ctor assoc id: nil}] . \\
\text{op } & \_ \_ : \text{NeList}[X] \ \text{NeList}[X] -> \text{NeList}[X] [\text{ctor assoc id: nil}] . 
\end{align*}
\]

Maude> (red-in MAP(LIST)[succ] : map.List(0 s(0) s(0) 0 0) .)
result NeList[B][succ] : s(0) s(s(0)) s(s(0)) s(0) s(0)

The following example illustrates the case of several parameters as well as the importation of other parameterized modules. The datatype consists of a kind of table where each entry associates a list over the second parameter to an element of the first parameter.

\[
\text{fmod TABLE}[X :: \text{TRIV}, Y :: \text{TRIV}] \text{ is }
\begin{align*}
\text{protecting LIST}[Y] . \\
\text{sort } & \text{Table}[X,Y] . \\
\text{op empty-table : } -> \text{Table}[X,Y] [\text{ctor}] . \\
\text{op add : Elt}[X] \ \text{List}[Y] \ \text{Table}[X,Y] -> \text{Table}[X,Y] [\text{ctor}] . 
\end{align*}
\]

Then, we can, for example, instantiate the first parameter of the parameterized module MAP(TABLE)[F1 :: MAP, F2 :: MAP] with the identity function over natural numbers, and the second one with the successor function as before, generating in this way the following module:

\[
\text{fmod MAP(TABLE)[id, succ] is }
\begin{align*}
\text{protecting TABLE}[A,A][id, succ] . \\
\text{protecting TABLE}[B,B][id, succ] . \\
\text{protecting MAP(LIST)][succ] . \\
\text{protecting MAP(BOOL)] . \\
\text{protecting NAT} . \\
\text{op map.Table : Table}[A,A][id, succ] -> Table[B,B][id, succ] . \\
\text{var } N : \text{Nat} . \\
\text{var } L : \text{List}[A][succ] . \\
\text{var } T : \text{Table}[A,A][id, succ] . \\
\text{eq map.Table(empty-table) = empty-table .} \\
\text{eq map.Table(add(N, L, T)) = add(N, map.List(L), map.Table(T)) .}
\end{align*}
\]

An example of reduction giving the expected result is the following:

Maude> (red map.Table(add(s(0), s(0) : []),
                          add(s(s(0)), s(s(0)) : s(0) : []),
                          empty-table))) .)
result Table[B, B][id, succ] :
    add(s(0), s(s(0)) : [],
    add(s(s(0)), s(s(s(0))) : s(s(0)) : [],
    empty-table))

We have also been working on implementing a polytypic function \textit{cata} at the Full Maude level. As explained in Section 3.2, a catamorphism is a function that replaces constructors with functions of appropriate ranks. In Full Maude, this correspondence can be directly expressed by means of a view over a theory that can be automatically generated from the corresponding module. For example, a theory for lists could have the following form:

\[\text{fth TH-LIST}[X :: \text{TRIV}]\text{ is}
\begin{align*}
\text{sort B}[X] .
\text{op c : } & \rightarrow \text{B}[X] .
\text{op f : Elt.X B}[X] & \rightarrow \text{B}[X] .
\end{align*}
\text{endth}\]

Note that it is essential in this context that the theory is parameterized in the same way as the given datatype. Otherwise, the \textit{cata} function would be defined only for a concrete instance of a datatype, and not for the generic case. However, parameterized theories and views are not yet available in Full Maude; indeed, they are currently under development, as described in some detail in [12]. Therefore, we have left this as part of our future work on the subject.

6 Conclusion and Future Work

The work presented in this paper shows that polytypic programming is possible in a reflective first-order language like Maude, without extending in any way the implementation of the language. In addition, the same framework allows proving non-trivial properties of polytypic functions. These results provide another example of the many advantages of reflection in programming languages in general, and in the Maude language in particular.

In [14] Goguen shows that typical higher-order programming examples can be captured with just first-order functions, by the systematic use of parameterized modules in OBJ. However, with the parameterized programming facilities implemented in OBJ, the class of higher-order functions that can be reduced to first-order functions are standard polymorphic functions over particular datatypes. Our work shares with Goguen’s work the intuition that higher-order programming can be captured with first-order functions. The novelty of our proposal consists in reducing higher-order functions to first-order \textit{metalevel} functions. This reflective approach allows us to specify as first-order functions a larger class of higher-order functions, including polytypic functions that are polymorphic also in the datatype constructor.

In addition to complete the implementation of the \textit{cata} function on Full
Maude, as mentioned at the end of the previous section, our future work will also consider the extension of map and cata to more general parameterized datatypes, like ordered lists or search trees over a poset, for example, where the parameter theory is no longer TRIV. In general, more examples are needed in order to be able to assess the advantages as well as possible disadvantages of polytypic programming in the context of Maude programming.

Experimentation on using the ITP tool to prove properties of polytypic functions written in Maude has just begun. We plan to use in the near future the methodology described here to formalize and prove other properties of polytypic programs, including generic properties, that is, properties that are not stated and proved relative to particular datatypes, like the ones we have dealt with in this paper, but relative to arbitrary datatypes.

References


