The Minimal Essence of Higher-Order Functions in Maude

Federico Bruzzone
Università degli Studi di Milano
Computer Science Department, Milan, IT
federico.bruzzone@unimi.it

Lorenzo Capra
Università degli Studi di Milano
Computer Science Department, Milan, IT
capra@di.unimi.it

Walter Cazzola
Università degli Studi di Milano
Computer Science Department, Milan, IT
cazzola@di.unimi.it

Carlos Olarte
Université Sorbonne Paris Nord,
LIPN, CNRS, UMR 7030, Villetaneuse, FR
olarte@lipn.univ-paris13.fr

***Abstract*—Rewriting logic, and its implementation in the Maude rewriting engine, provides a flexible semantic and logical framework that has been extensively used for the analysis of a wide range of systems, from programming languages to cyber-physical systems. Although pure type systems have been specified in rewriting logic as object logics, and state monads have been shown to be definable in Maude, there is still no full implementation in Maude of higher-order functions that is sufficiently flexible for general users. In this paper, we show how higher-order programming can be made available in Maude through a natural and declarative encoding of higher-order iterators and lambda functions. We present two alternative approaches to achieve this goal and compare them in terms of expressiveness and efficiency. Our ultimate objective is to give Maude specifiers not only the possibility of writing higher-order functions directly in Maude, but also the infrastructure needed to simplify the specification of rewrite-theory transformations. Such transformations are a common task in verification tasks using Maude, and they could substantially benefit from the modules proposed in this paper.**

***Index Terms*—Rewriting logic, Maude, higher-order programming.**

I. Introduction

Rewriting logic [1] (RL) is a powerful and expressive semantic and logical framework whose specification unit is a *rewrite theory*, combining an equational theory and rewriting rules. Its implementation in the high-performance rewriting engine Maude [2] has been extensively used for the specification of a wide variety of systems, ranging from programming languages and models of computation to protocols and distributed systems, among many others.

Despite its expressiveness, Maude is essentially a *first-order* declarative language. This fact has historically limited the ability to specify rewrite theories featuring higher-order constructs, such as lambda abstractions and functional combinators like `map` and `fold`, which are fully integrated into modern specification and programming languages. Hence, typical tasks such as mapping functions over data structures, performing systematic computations, or implementing theory transformations have usually been carried out manually, often by repeating similar pieces of code several times.

To bridge this gap, substantial research has focused on embedding higher-order capabilities within Maude. A landmark contribution by Rusu and Arusoaie [3] successfully incorporated higher-order functions (HOF) and state monads into Full Maude[1]. Their framework enables the concrete and symbolic execution of functional-imperative programs and supports sophisticated program-equivalence verification via Reachability Logic. However, this approach comes with intrinsic operational costs: it relies heavily on Full Maude's reflective features, external meta-level module manipulation, and a structured state-monad architecture to orchestrate execution. While highly effective for complex functional-imperative behaviors, it leaves open a fundamental question: *What is the "simpler" and purest mechanism required to natively support higher-order programming directly within Maude's standard equational logic?*

We address this question by presenting a lightweight infrastructure for higher-order programming in Maude. Moving away from state monads and complex meta-programming layers, our work identifies the basic primitives needed to express lambda functions and higher-order iterations. We explore two possible alternatives. The first relies on Maude's robust module system and defines a hierarchy of theories and views that provide the fundamental components of parameterized programming in Maude. The second alternative uses meta-programming in Maude *only* to define substitutions and then β-reductions in lambdas. We show that the first approach exhibits better efficiency, whereas the second offers greater expressiveness in the way users can write code. Both approaches are implemented in Maude itself, and the full specifications, together with the examples presented in this paper, are publicly available in a companion GitHub repository.[2]

Our approach is based on the following design principles and methods:

[1]Full Maude is an extension of Maude written in Maude itself using its meta-programming capabilities, and has been used as a laboratory to test ideas before being implemented at the core C++ level in Maude.

[2]https://github.com/lgcapra/MAUDE-LAMBDA

1) **Theories and Parameterized Views:** We leverage Maude's advanced theory and view system to define module interfaces (`COLLECTION`, `AS-COLLECTION`) that cleanly capture the mathematical essence of "iterable" data structures and allow for a flexible instantiation mechanism for higher-order functions. Although Maude parameterized module operations require expertise, we present uniform patterns that make these operations easier to apply.
2) **First-Class Lambda Abstractions:** Since substitutions cannot be defined at the object-level for arbitrary data-types, we implement a module `SUBST` that incorporates the machinery to perform substitutions at the meta-level for arbitrary data-types. This allows us to implement true anonymous lambda abstractions (`lambda_:_`) that evaluate and move the final result back to the object level. Our definition of lambda supports currying and partial application out-of-the-box.
3) **Decoupled Higher-Order Combinators:** We implement tail-recursive `map` and `fold` operations (`MAP-ITE`, `FOLD-ITE`) that operate over abstract iterable structures. This allows heterogeneous collections (e.g., standard lists, sets, etc.) to act interchangeably as reduction sources or targets without requiring ad-hoc or manual manipulations.

By avoiding the overhead of the monadic machinery and Full Maude extensions required in [3], our approach isolates the essential ingredients of an elegant and purely declarative form of higher-order programming in Maude. The equational theories introduced in this work may also help simplify recurring tasks in Maude specifications. Moreover, the resulting framework provides further evidence (see also Section VII) that Maude's native first-order engine already offers the mechanisms needed to support higher-order constructs in a natural and elegant way.

The remainder of this paper is organized as follows. Section II provides an overview of Maude and its semantics. Section III presents a core-language approach to introducing higher-order functions in Maude (in particular, the map operation) without using Maude's meta-programming facilities. Section IV describes a simple encoding of lambda functions in Maude, and Section V implements higher-order functions on "iterable" data-structures using this encoding. Section VI compares the two approaches discussed. Related work is discussed in Section VII and we conclude in Section VIII.

II. Maude and Rewriting Logic

Maude [2] is an expressive, high-performance, purely declarative language that implements the rewriting logic framework [1], [4]. The Maude runtime provides various facilities for model checking, verification of LTL formulae, and symbolic analyses (see a survey in [5], [6]). Maude has served as a semantic and logical framework for various other formalisms, including Petri nets, Automata, and Process Algebras, and as a platform for the verification of different systems, including biological systems, distributed systems, real-time systems, and deductive systems, among several others [5].

In rewriting logic, a specification consists of an equational theory, called a *functional* module in Maude, plus a set of rewrite rules, given by Maude *system* modules. The equational theory defines algebraic data types and deterministic computations, while the rewrite rules define transformations on terms representing system states.

Each side of a rule or equation is a *term* of a certain *kind*, possibly containing variables that are implicitly universally quantified. Rules and equations work by rewriting: Instances on the left-hand side are replaced by corresponding instances on the right-hand side. For executability and to preserve the mathematical meaning of Maude specifications, the equational theory must be confluent and terminating, ensuring the existence of canonical forms. These restrictions do not apply to the rewrite rules, which may describe nondeterministic and nonterminating system behaviors.

A *functional* module defines sorts and operations by means of equations, which are used as "simplifications". More precisely, an *equational theory* $(\Sigma, E \cup A)$ in membership equational logic [7] consists of the following components. The signature Σ includes the declaration of *sorts*, *subsorts*, and *kinds*. Kinds, denoted by $[s]$ for a given sort s, represent the equivalence classes that group connected components of sorts in the subsort partial order. Intuitively, a kind is used as the type of undefined or error terms (those in a kind without a specific sort) and to define the range of partial functions. The set E contains conditional equations of the form $t = t'$ `if` ϕ, where t and t' are terms of the same kind and ϕ is a conjunction of equations. It also contains conditional membership axioms of the form t `:` s `if` ϕ, stating that t belongs to the sort s. Finally, A is a set of operator attributes, such as `assoc`, `comm`, and `id:` ι, stating that the operator is associative, commutative, and/or has an identity element ι.

The mathematical model of $(\Sigma, E \cup A)$ is the *initial algebra* $T_{\Sigma/E \cup A}$, formed by the equivalence classes of the relation induced by $E \cup A$ in the ground-term algebra T_Σ. Under the conditions (modulo-A) of confluence, subsort decreasing, and termination, any ground term in the theory is rewritten in a unique canonical form. The canonical term algebra is isomorphic to the initial algebra, which makes the denotational and operational semantics consistent.

A *system* module includes *rewrite rules* that represent local transitions in a concurrent system. It defines a *rewrite theory* [4] $\mathcal{R} = (\Sigma, E \cup A, R)$. Here, $(\Sigma, E \cup A)$ is an equational theory, and R is a set of rewriting rules. A rewrite theory captures the behavior of a concurrent system, with $(\Sigma, E \cup A)$ defining the state of the systems as terms, and R describing the concurrent transitions. The initial model of $\mathcal{R}$ provides each kind k with a labeled transition system (TS) where states are elements of $T_{\Sigma/E \cup A, k}$ and state transitions occur as $[t] \xrightarrow{[\alpha]} [t']$, with $[\alpha]$ denoting an equivalence class of rewrites.

The main rewriting mechanism of Maude uses a matching modulo-A procedure to determine all admissible instances (that is, substitutions of variables by ground terms) for a given equation or rule. In this setting, the *coherence* property [2]

```
fmod M is ... endfm        --- Functional module
fth  T is ... endfth       --- Functional theory
view V from S              --- View
  to T is ... endv
pr / ex / gb / inc R .     --- Importing a theory R
sorts S ... Sk .           --- Declaration of sorts
subsort S1 < S2 .          --- Subsort relation
vars X1 ... Xm : S .       --- Logical vars of sort S
op f : S1 ... Sn -> S .    --- Operator
op c : -> S .              --- Constant of sort S
eq t = t' .                --- Equation
ceq t = t' if c .          --- Conditional equation
```

Listing 1: Syntax of Maude.

guarantees that the strategy used by the Maude rewrite engine—first reducing terms to their canonical forms and then applying rules—is both sound and complete.

Throughout this paper, we use Maude syntax to specify rewrite theories. In fact, we consider only equational theories since the use of rewrite rules is orthogonal to the developments presented here. The main Maude constructs used in the paper are summarized in Listing 1. Additional syntactic attributes will be introduced later when needed.

An equational theory corresponds in Maude to a functional module **fmod** ... **endfm**. In a *parameterized* functional module $M\{X_1 :: T_1, \cdots, X_n :: T_n\}$, each formal parameter X_i is declared with a theory T_i (**fth** T **is** ... **endfth**), with the same structure as a functional module, which acts as its "type".[3] A *view* **view** V_i **from** T_i **to** M_i **is** ... **endv** specifies the mapping of the sorts and operations of the theory T_i into those of the *actual parameter* M_i, where M_i may be either a module or a theory (in the latter case, one speaks of theory views); Any identity mapping whose sort or operator has the same name and/or the same arity and coarity as in the target can be left out.

For example, Maude's functional theory `TRIV` declares only the sort `Elt`. Importing **pr** `LIST{Int}` into another module, where the view `Int` maps `Elt` to the sort `Int`, specializes the parameterized module `LIST{X :: TRIV}` to lists of integers.

There are four different modes for importing modules. A module M **pr**otects an imported module M' if the import introduces neither "junk" (no new canonical ground terms of that sort are introduced) nor "confusion" (no new equations identify terms that were distinct in M). A module M **ex**tends M' if the import satisfies only the no-confusion condition, while the generated-by mode (**gb**) requires only the no-junk condition. The **inc**luding mode imposes neither condition.

Maude supports *meta-programming*, where a module M (resp., a term t) can be (meta)represented as a Maude *term* $\overline{M}$ of sort `Module` (resp. as a Maude term $\overline{t}$ of sort `Term`). These sorts are defined in Maude's `META-LEVEL` module, which also includes, among the others, the meta representation of sorts, equations, rewrite rules, views, etc. Moreover, analysis commands and model/module transformations can then be easily defined as ordinary Maude functions on such meta-terms. For this purpose, Maude provides built-in functions such as `metaReduce`, which is the meta-level version of the equational reduction command `reduce` (abbreviated as `red`) that computes the canonical form of a term. Moreover, the `upTerm` and `downTerm` operations allow movement between reflection levels.

[3] A theory can contain non-executable equations, interpreted as logical constraints

```
1  fth COLLECTION is
2    sorts Elt Coll .
3    op nil : -> Coll [pconst] .
4    op __ : Coll Elt -> Coll .
5    op __ : Coll Coll -> Coll [id: nil] .
6  endfth
8  fth AS-COLLECTION is
9    inc COLLECTION .
10   sort Iterable .
11   op iterator : Iterable -> Coll .
12 endfth
14 view ListColl{X :: TRIV} from COLLECTION
15     to LIST{X} is
16   sort Coll to List{X} .
17   sort Elt to X$Elt .
18 endv
20 view ElistAsColl{X :: TRIV}
21  from AS-COLLECTION to ELIST{X} is
22   sort Coll to List{X} .
23   sort Elt to X$Elt .
24   sort Iterable to Elist{X} .
25   op iterator to list .
26 endv
28 view COLLECTION from AS-COLLECTION
29     to COLLECTION is
30   sort Iterable to Coll .
31   op iterator(C:Iterable) to term C:Coll .
32 endv
34 view IdeColl from COLLECTION to COLLECTION is
35 endv
```

Listing 2: Theories specifying Iterables and Collections, views towards collection families, and theory-views

```
1  fmod MAP-ASCOLL{X :: AS-COLLECTION,
2      Y :: COLLECTION} is
3   op f : [X$Elt] -> [Y$Elt] .
4   op map : X$Iterable -> [Y$Coll] .
5   var l   : X$Iterable . var l'  : Y$Coll .
6   var l'' : X$Coll . var e : X$Elt .
7
8   eq map(l) = $map(iterator(l), Y$nil) .
9   op $map : X$Coll Y$Coll -> [Y$Coll] .
10  eq $map(X$nil, l') = l' .
11  eq $map ((l'' e),l') = $map(l'',(Y$nil f(e)) l') .
12 endfm
14 fmod MAP-COLL{X :: COLLECTION,Y :: COLLECTION} is
15   pr MAP-ASCOLL{COLLECTION, IdeColl}{X, Y}  .
16 endfm
18 fmod MAP-SAME-COLL{X :: COLLECTION} is
19   pr MAP-COLL{X, X}  .
20 endfm
```

Listing 3: Map operation implemented in a conventional way

III. HOF VIA PARAMETRIZED MODULES

This section introduces the rewrite theories needed to represent iterables and collections in Maude and describes a methodology for computing with higher-order functions in core Maude. For the sake of presentation we omit some details and the complete specification is available in the companion repository of this paper (see Footnote 2). To ease the presentation, we focus on linear structures, and the repository contains the general case of arbitrary data structures, including, for instance, examples involving trees. Moreover, we describe in detail the case of the `map` function, which applies a function $f : A \to B$ to an iterable structure that contains elements of type A and produces a collection of elements of type B. Other higher-order functions can be obtained in a similar way, and some of them are described later.

A. Collections and Iterables

Any algebraic data type can be specified in Maude. In fact, typical data structures such as lists, sets, and maps are already defined in the Maude prelude. We begin by showing how to specify linear data structures that can be regarded as "iterable" objects, a notion that is not explicitly provided by Maude's standard data structure specifications. This will allow us to define higher-order functions on such structures more easily.

The excerpts in Listing 2 are from our repository's file `COLLECTIONS.maude` (This file, along with those referenced in Section III-B, can be found in the `base` subdirectory of the online repository.). As already mentioned, theories define module interfaces—the syntactic and semantic conditions that parameter modules must satisfy—and have loose semantics: any algebra satisfying the theory is a valid model. Views describe how a target module or theory meets the requirements of a source theory.

The `COLLECTION` theory declares two sorts: `Elt` (elements) and `Coll` (collections). It defines `nil` as the empty collection and two juxtaposition operations: a cons-like constructor (adds one element) and an append operator for two collections, with `nil` as the identity. The `pconst` attribute makes `nil` a constant parameter, allowing the notation `X$nil` in a module parameterized by `X :: COLLECTION`.

The Maude's built-in modules `LIST`, `SET`, and `MAP`—each parameterized by `TRIV`, which declares only a sort `Elt`—conform to this specification. Observe that `COLLECTION` is strictly more general than `MONOID` since it does not impose associativity as a requirement (even though in Maude collections are typically implemented using an associative constructor). The companion theory `AS-COLLECTION` imports `COLLECTION` and characterizes iterable data structures via a sort `Iterable` and an `iterator` operation that projects any iterable into a collection.

A standard list in Maude, as defined by the `LIST` module, does not require explicit delimiters. For example, the term `-1 3 5` is a valid term of type `List{Int}`. In some of our examples, we use the module `ELIST{X :: TRIV}`, not shown here, which imports `LIST{X}` and declares an additional constructor to build lists enclosed in brackets. Thus, the term `[-1 3 5]` is a valid term of sort `EList{Int}`.

`LIST` and `ELIST` can be shown to be instances of both `COLLECTION` and `AS-COLLECTION` theory. The parameterized views `ListColl` and `ElistAsColl` in Listing 2 provide two of such instances.

Finally, the last two theory-views in Listing 2 serve to flexibly adapt the interface of an imported parameterized module to that of the surrounding module: the view `COLLECTION` maps the theory `AS-COLLECTION` to the theory of the same name, `COLLECTION` (Maude maintains distinct namespaces for views and theories), using an on-the-fly mapping for the `iterator`; `IdeColl` is simply an identity view for the theory `COLLECTION`.

B. Higher-order functions

The higher-order function `map`, that transforms any `AS-COLLECTION` structure to an arbitrary `COLLECTION`[4], is formalized by the functional module `MAP-ASCOLL` (repository's file `BASE-MAP.maude`) in Listing 3. The implementation of `map` is tail-recursive (via the auxiliary operator `$map`). It relies on the `iterator` to traverse the data structure and apply the function `f` to each element. Thus, we map the source elements to the target elements. To preserve full generality, the function `f` is left as an abstract symbol that is declared, but it needs to be defined equationally by the user later. This function is typed using kinds rather than sorts, allowing `f` to be instantiated with any operation, including partial ones such as those in the predefined `CONVERSION` module used in the examples below.

The next example shows how to obtain a concrete mapping by instantiating `MAP-ASCOLL`.

Example 1 (Using `map`)**.** *The module `EX1` below instantiates `MAP-ASCOLL` to obtain a concrete mapping:*

```
fmod EX1 is
 gb MAP-ASCOLL{ElistAsColl{Float},ListColl{String}} .
 pr CONVERSION .
 var x : Float .
 eq f(x) = string(ceiling(x)) .
endfm
```

This module allows for converting an encapsulated list of floats into a list of strings. For that, the function `f` returns the string representation of the ceiling of the input value. The two functions appearing in the definition of `f` are specified in the Maude predefined module `CONVERSION`.

`MAP-ASCOLL` is imported in `generated-by` mode, so some values from the imported module may satisfy additional equations. This formal setup matches defining the abstract operation symbol `f` by an equational specification. As previously noted, kind-based typing of `f` enables instantiation of the abstract operation with the function `string`, which is defined as a partial function in `CONVERSION`.

The following output of the execution of the `reduce` command in Maude, to find the normal form of a term, shows the expected result with the least sort of the returned term.

[4]In functional programming languages, `map` typically preserves the structure of its input. For generality, our version of `map` may produce a collection whose type differs from that of the input structure.

```
Maude> reduce in EX1 : map([1.0 2.5 3.2]) .
result NeList{String}: "1.0" "3.0" "4.0"
```

For convenience, the module MAP-COLL specializes MAP-ASCOLL in mapping COLLECTIONs, using the theory-views COLLECTION (from MAP-ASCOLL to COLLECTION) and the identity IdeColl (Listing 2) to instantiate the latter. In these cases, the module's parameters X and Y are now uninstantiated and must later be bound. The module MAP-SAME-COLL intuitively specializes MAP-COLL for endomorphic mappings.

Because Maude's rewriting engine is based on matching, MAP-COLL can be used when the addition operator of the source collection is a constructor. For example, if we instantiate ELIST via a view that maps its addition operator to the corresponding LIST operator, the result is always undefined terms of a specific *kind*. In this case, we must use MAP-ASCOLL instead.

The same methodology extends straightforwardly to other common higher-order operations such as *filter* and *fold*. The technique is sound, elegant, and efficient (see Section VI). Its main limitation, however, is that applying map with a different transformation function requires defining an entirely new module. This boilerplate cannot be eliminated simply by parameterizing MAP-ASCOLL over a theory grouping the source iterable, the target collection, and the transformation. We explored this alternative in the ITE-MAP module provided in the repository file ITERABLE-MAP.maude. However, putting this approach into practice turned out to be more cumbersome than the method described here.

IV. Modeling Lambda Functions

In this section and the subsequent one, we introduce an alternative method for representing higher-order functions using the Maude meta-level (The related lightweight module hierarchy can be found in the lambda subdirectory of the online repository). This approach is based on a compact encoding of lambda expressions provided by the ARROW module (Listing 4). Moreover, the module SUBST, omitted here, uses meta-operations to define substitutions for any term (These base modules are enclosed in the LAMBDA-DEF.maude file). The encoding presented here substantially extends and enhances the one proposed in [3].

A. Arrow types

The signature of a Maude operator, as discussed above, has the form $g : s_1 \cdots s_n \to s$, where s_i are the sorts of arguments of g, and s is the sort of values returned by g. Therefore, it is not possible to *directly* define a higher-order operator whose arguments include a function, such as $g' : (s_1 \to s_2)\ s_3 \to s$.

The ARROW module in Listing 4 specifies lambda abstractions whose domain and codomain are determined by the module parameters. More precisely, the sort Elt of the parameter X, denoted in Maude as X$Elt, is used as the domain of the function, whereas the corresponding sort of parameter Y is used as its codomain. For instance, the module instantiation ARROW{String,Nat}, where String and Nat are predefined Maude views for strings and natural numbers, respectively, defines the sort Arrow{String,Nat}. This sort represents anonymous functions of type String → Nat.

Using the sort Arrow{X, Y}, we can define lambdas using the operator **op** lambda_:_. For instance, the term lambda x:String : length(x:String), of sort Arrow{String,Nat}, specifies an anonymous function that maps a given string to its length.

Note that the sort of the first argument of lambda is X$Elt. In the running example, this means that *any* expression of sort String could, in principle, be used in this argument position, potentially producing ill-formed lambda expressions. To prevent this, the membership axiom in the module ARROW specifies that only variables, such as x:String, can occur in this position. We thus ensure that lambda abstractions can bind only variables of the appropriate sort.

A lambda term contains variables, which is allowed because the Maude rewriting engine treats them as ordinary ground terms. In this setting, these variables bypass standard matching and are handled at the meta-level.

```
1  fmod ARROW{X :: TRIV, Y :: TRIV} is
2    ex SUBST .
3    sort Arrow{X,Y} .
4    op lambda_:_ : X$Elt [Y$Elt] -> [Arrow{X,Y}] .
5    op __ : Arrow{X,Y} X$Elt -> [Y$Elt] .
6    op ARR-ERR{X,Y} : -> [Y$Elt] [ pconst ].
7    vars x acc : X$Elt . var y    : [Y$Elt] .
8    cmb lambda x : y : Arrow{X,Y}
9        if upTerm(x) :: Variable .
10   eq lambda x : y acc =
11      downTerm(subst(upTerm(x), upTerm(acc),
12         upTerm(y)), ARR-ERR{X,Y}) .
13 endfm
15 view Arrow{X :: TRIV, Y :: TRIV}
16     from  TRIV to ARROW{X,Y} is
17   sort Elt to Arrow{X,Y} .
18 endv
```

Listing 4: The ARROW module

B. β-reduction

As expected, evaluating lambda abstractions applied to terms requires the definition of substitutions. However, this operation cannot be defined at the object level, since we have no information about the constructors of the sort used in the domain of the function. Hence, we cannot define a substitution by case analysis on the input term.

We thus define substitutions at Maude's meta-level, where terms of all sorts share a common representation as terms of sort Term. For example, a variable declaration V:S, when lifted to the meta-level using upTerm, is represented as 'V:S; the constant c of sort S is represented as 'c.S; and an application of any operator f to its arguments $a_1, \cdots, a_n$ is represented as the term '$f[\overline{a_1}, \cdots, \overline{a_n}]$, where $\overline{a_i}$ denotes the meta-level representations of a_i. This allows us to define the function subst in the module SUBST, not shown here, simply by considering the possible constructors for (meta-)terms of sort Term as those illustrated above.

The operator **op** __ defines the function application by juxtaposition, as in the expression below:

```
(lambda x:String : length(x:String)) "abba".
```

The equational definition of this operator performs variable substitution at the meta-level on the meta-representation of the body of the lambda abstraction, obtained using `upTerm` in Listing 4, and then converts the resulting term back to the object level using the descent function `downTerm`. The second argument of `downTerm` provides a "default" value to be returned when the meta-level term cannot be converted into a valid object-level term of the expected sort. This is the role of the constant `ARR-ERR{X,Y}` in the Listing 4.[5]

Finally, the view declaration in Listing 4 will allow us later to define arrow types whose domain or codomain may themselves be function types.

Example 2 (β-reduction)**.** *Consider the following module:*

```
fmod EX2 is
  pr ARROW{Float, String} . pr CONVERSION .
endfm
```

that defines the sort for lambdas of type `Float` $\rightarrow$ `String`*. The following command illustrates the result of a β-reduction.*

```
Maude> red (lambda x:Float : string(x:Float)) 3.5 .
result String: "3.5"
```

V. Base Operations: Map, Filter and Fold

The formalization of lambda expressions presented in the previous section provides a flexible way to define higher-order operations without significant notational overhead. Building on this specification, we show in this section how to define the typical HO functions, notably the *fold* combinator, which requires the definition of binary functions.

A. Defining maps

The module `MAP-ASCOLL'` in Listing 5 defines the `map` operation using lambda abstractions, and it has essentially the same interface and structure as the corresponding module in Listing 3. The main difference is that it is built on top of `ARROW`, imported through the theory views `AsCollection` and `Collection`. These views align the interfaces of the surrounding theories, `AS-COLLECTION` and `COLLECTION`, with those expected by the imported module. For convenience, here we use the sort renaming when importing modules (see the renaming after the symbol "$*$").

The `map` operator now takes an additional argument `f` of sort `Arrow{X,Y}`, which specifies the transformation to be applied during the mapping operation `$map`. Function application is handled via β-reduction. Finally, by using theory views once again, we can directly construct the module `MAP-COLL'`, which implements mapping between arbitrary `COLLECTION`s.

[5]The inclusion of this data value explains why the `ARROW` module imports `SUBST` in **ex** mode.

Example 3. *Consider the following module (defining maps from list of strings to sets of natural numbers) and executions of the* `reduce` *command:*

```
fmod EX3 is
  pr MAP-COLL'{ListColl{String}, SetColl{Nat}} *
  (sort Arrow{ListColl{String}, SetColl{Nat}}
    to Arrow{String, Nat}) .
  op toLen  : String -> Arrow{String, Nat} .
  var x : String .
  eq toLen(x) = lambda x : length(x) .
endfm
```

```
Maude> red  map(toLen(x), "cat" "dog" "Tiger"
"Shark" "Snake" "leopard" "lion") .
result NeSet{Nat}: 3, 4, 5, 7
Maude> red map(lambda x:Char : ascii(x:Char),
"a" "d" "T" "S") .
result NeSet{Nat}: 83, 84, 97, 100
```

It should be noted that one may either define named aliases for lambda expressions (e.g., `toLen(x)`*) and subsequently reuse them within* `map`*, or instead employ anonymous lambda abstractions inline, as demonstrated in the second reduction.*

```
1  fmod MAP-ASCOLL'{X :: AS-COLLECTION,
2                   Y :: COLLECTION} is
3   pr ARROW{AsCollection,Collection}{X,Y} *
4   (sort Arrow{AsCollection,Collection}{X,Y}
5    to Arrow{X,Y}).
6   op map : Arrow{X,Y} X$Iterable -> [Y$Coll] .
7   var f   : Arrow{X,Y} . var l   : X$Iterable  .
8   var l'  : Y$Coll . var l'' : X$Coll .
9   var e : X$Elt .
10  eq map(f, l) = $map(f, iterator(l), Y$nil) .
11  op $map : Arrow{X,Y} X$Coll Y$Coll -> [Y$Coll] .
12  eq $map(f, X$nil, l') = l' .
13  eq $map (f,(l'' e),l') =
14     $map(f,l'',(Y$nil f e) l') .
15 endfm
17 fmod MAP-COLL'{X :: COLLECTION, Y :: COLLECTION} is
18   pr MAP-ASCOLL'{COLLECTION, IdeColl}{X, Y} *
19    (sort Arrow{COLLECTION, IdeColl}{X,Y}
20     to Arrow{X,Y}) .
21 endfm
```

Listing 5: HO map based on lambda

B. Fold: encoding binary functions

Within functional programming, the `fold` operator (also called a reduce) constitutes one of the most general higher-order combinators. In fact, both map and filter can be systematically expressed as special cases of `fold`.

A `fold` recursively replaces the constructors of a recursive (iterable) data structure with a combining function f and initial value acc. For instance, in a list, $fold\ f\ acc\ [x_1, x_2, \ldots, x_n] = f(x_1, f(x_2, \ldots f(x_n, acc)\ldots))$. The fold of the empty list is acc and f is a *binary* function of the appropriate type.

It is straightforward to define an n-ary function by using the curried form of lambdas. For instance, given the `ARROW` instantiation: `ARROW{String, Arrow{String,Bool}}`, the term:

```
lambda n : lambda m : length(n) < length(m)
```

where n and m are String variables, represents a binary function that is evaluated, (using the alias less(n, m)), by reducing, e.g., (less(n, m) "ab") "acc".

Unfortunately, the current mechanism for instantiating module parameters in Maude has certain shortcomings when theory-views are used to (temporarily) instantiate parameters in multi-level bindings. As a result, a generic instantiation of ARROW{X :: TRIV,Y :: TRIV} of the form:

```
ARROW{AsCollection, Arrow{TRIV,TRIV}}{X, Y}
```

fails at runtime, even though it is logically permissible.

To solve this "limitation" in Maude's module system, we use a simple pattern as a workaround, as illustrated in Listing 6. Rather than using a theory-view from TRIV to C, we instead declare an empty module DUMMY{X :: C} together with a parameterized view V{X :: C} from TRIV to DUMMY{X}. In this way, binary (and, more generally, n-ary) functions can be specified as particular instances of ARROW, thereby allowing us to reuse the corresponding β-reduction rules on these functions in a transparent way.

This approach is applied to the parameterized module BINFUN, which is derived from ARROW and designed so that its types match those of the binary combinator f used in the fold operation. The generic fold for iterables is then defined in the module FOLD-ASCOLL, which imports BINFUN via parameterized views on dummy modules that serve as substitutes for the theory-views. The fold operator is implemented in a tail-recursive way and is illustrated in the next example.

Example 4. *Consider the following module definition and execution of the* reduce *command:*

```
fmod FOLD-EXE is
 pr FOLD-ASCOLL{ElistAsColl{Float}, Float}
 * (sort BinFun{ElistAsColl{Float},Float}
       to BinFun{Float,Float}) .
 vars n acc : Float .
 op max : Float Float -> BinFun{Float,Float} .
 eq max(n, acc) = lambda n : lambda acc :
   if n > acc then n else acc fi .
endfm
```

```
Maude> red fold(max(n, acc), 0.0,
[3.0 1.0 4.0 1.0 5.0 9.0 2.0 6.0]) .
result FiniteFloat: 9.0
```

Here, fold *is used to determine the maximum value of an encapsulated list of* Float *elements. Analogously to map, we can define a fold on collections as a direct derivation of the one on iterables.*

To conclude, we illustrate how operations map and filter can be easily expressed as special cases of fold. The module FOLD-AS-MAP (Listing 7) is actually obtained directly from FOLD-ASCOLL (Listing 6) by instantiating the parameter Y of the module with the Coll sort of the theory COLLECTION via the parameterized view Collection' (from TRIV to DUMMY-COLL). Consequently, the second argument of the binary combinator f used in the fold is itself a collection.

```
1  fmod BINFUN{X :: TRIV,Y :: TRIV} is
2    pr ARROW{X,Arrow{Y,Y}} *
3    (sort Arrow{X,Arrow{Y,Y}} to BinFun{X,Y}) .
4  endfm
6  fmod DUMMY-ASCOLL{X :: AS-COLLECTION} is endfm
8  view AsColl{X :: AS-COLLECTION}
9   from  TRIV to DUMMY-ASCOLL{X} is
10  sort Elt to X$Elt .
11 endv
13 fmod FOLD-ASCOLL{X :: AS-COLLECTION, Y :: TRIV} is
14     pr BINFUN{AsColl{X}, Y} * (sort
15      BinFun{AsColl{X}, Y} to BinFun{X,Y}) .
16  op fold : BinFun{X,Y} Y$Elt X$Iterable -> [Y$Elt] .
17
18  var acc : Y$Elt . var l : X$Iterable .
19  var e : X$Elt . var l' : X$Coll .
20  var f : BinFun{X,Y} .
21  eq fold(f, acc, l) = $fold(f, acc, iterator(l)) .
22  op $fold : BinFun{X,Y} Y$Elt X$Coll -> [Y$Elt] .
23  eq $fold(f, acc, X$nil) = acc .
24  eq $fold(f, acc, l' e)  = $fold(f, f e acc, l') .
25 endfm
```

Listing 6: The HO fold operation

Example 5 (map as fold)**.** *In the following module, FOLD-AS-MAP is used to convert a list of Float s into the corresponding String s, while discarding all values that fall below a specified threshold.*

```
fmod FOLD-MAP-EXE is
  pr FOLD-AS-MAP{ElistAsColl{Float},
                 ListColl{String}} *
  (sort Arrow{Collection{ListColl{String}}}
   to  Arrow{List{String}},
   sort  BinFun{ElistAsColl{Float},
         Collection{ListColl{String}}}
   to BinFun{Float,List{String}}) .
  pr CONVERSION .
  var x : Float . var acc : List{String} .
  op bf : Float List{String} ->
        BinFun{Float, List{String}} .
  eq bf(x, acc) = lambda x : lambda acc :
    if abs(x) > 1e-6 then (string(x) acc)
    else nil acc fi .
endfm
```

```
Maude> red  fold(bf(x, acc), nil,
  [ 1.0 -2.3e-8 3.0 1.0 2.0 3.0e-10 ]) .
result NeList{String}: "1.0" "3.0" "1.0" "2.0"
```

VI. Comparison of the Approaches

The approach proposed in Section IV, based on lambda abstractions, is clearly closer to common functional-programming practice than the approach presented in Section III. However, this additional flexibility comes with a cost in terms of rewriting, as meta-level operations are required to implement β-reduction.

We compared three implementations of a simple map operation that collects, in a set, the distinct lengths of randomly generated strings that occur in a list. The first implementation, imp1, uses standard Maude data types and does not rely on

```
fmod DUMMY-COLL{X :: COLLECTION} is endfm

view Collection'{X :: COLLECTION} from TRIV
    to DUMMY-COLL{X} is
 sort Elt to X$Coll .
endv

fmod FOLD-AS-MAP{X :: AS-COLLECTION,
                 Y :: COLLECTION}
 is pr FOLD-ASCOLL{X, Collection'{Y}} *
 (sort Arrow{Collection'{Y},Collection'{Y}}
  to Arrow{Collection{Y}}, sort
  BinFun{X,Collection'{Y}}
  to BinFun{X,Collection{Y}}).
endfm
```

Listing 7: map as a particular fold

parameterized modules or lambda abstractions. Consequently, it is specialized for this particular task. The second implementation, `imp2`, follows the parameterized-module approach: different input and output types can be instantiated, but the function to be mapped must be defined in a separate module. Finally, `imp3` uses lambda abstractions, where the mapped function can be changed directly in the input term without defining additional modules.

We evaluated the three implementations on a mid-range laptop using `hyperfine`. Each benchmark measures the execution time required to process a deterministically generated list of random strings. Relative execution times have been observed to remain substantially stable, regardless of the length of the lists. The results for a list containing 50.000 strings are the following:

Implementation	Approach	Relative execution time
`imp1`	Specialized implementation	(1.00)
`imp2`	Parameterized modules	(1.02 $\pm$ 0.03)
`imp3`	Lambda abstractions	(1.68 $\pm$ 0.06)

These results indicate that the implementation based on parameterized modules achieves a performance that is very close to the "native" (albeit unflexible) Maude implementation. In contrast, the implementation relying on lambda abstractions is approximately 68% slower. This performance overhead can be regarded as acceptable in light of the increased expressiveness and flexibility afforded by the use of lambda abstractions.

The online repository contains the instructions for reproducing these experiments.

VII. Related Work

Due to the flexibility of rewriting logic as a logical framework, different approaches for representing binding operators and higher-order features have been proposed. For instance, Stehr and Meseguer [8] demonstrated that typed higher-order languages, in the form of pure type systems, can be naturally represented as object logics within rewriting logic. Similarly, Fernández and Meseguer proposed a general methodology in [9] for specifying languages with binding operators using a nominal approach.

Rusu and Arusoaie [3] showed that higher-order functions can be executed in Maude. However, their execution model relies on state monads and complex meta-programming transformations to control the execution state and variable binding. Unlike these previous works, which either treat higher-order features as object logics or introduce "complex" operational layers, our proposal is specifically aimed at providing a direct, simple, and declarative implementation of HOF that users can leverage natively while programming in core Maude.

VIII. Conclusions and Future Work

We have shown how to provide a native implementation of higher-order functions in core Maude. Our approach heavily exploits parameterized modules, while limiting the use of Maude's meta-level strictly to the definition of variable substitutions required by β-reduction.

One of the main modeling advantages of rewriting logic is that the system states can be represented as arbitrary data types, enabling the specification and verification of sophisticated distributed systems. Our contributions facilitate such specifications by simplifying the manipulation of complex data structures through declarative HOFs.

Our developments open up promising applications in existing Maude-based verification tools. For instance, in Real-Time Maude [10] (for real-time systems) and the L-framework [11] (for deductive systems), a recurring task is the definition of theory transformations. Since lambdas can operate directly on the meta-representation of rewrite rules (terms of sort `Rule` and `RuleSet`), our framework can be used to map, filter, and transform the components of modules more conveniently.

We plan to extend and optimize this framework as follows:

1) Generalized Abstractions; While the current evaluation focuses on linear data structures, we are developing algebraic interfaces inspired by standard Haskell typeclasses to define *functors* and *foldables*. This will bring general mapping and folding operations to a broader range of non-linear data structures such as trees.

2) Core Optimization; As shown in Section VI, the lambda-based approach is slower than traditional parameterized modules. To reduce this gap, we plan to integrate the substitution and evaluation techniques from Section IV directly into Maude's C++ core engine.

3) Syntactic Support; It would be interesting to introduce a user-friendly notation and dedicated syntactic sugar to make lambda abstractions more idiomatic and convenient for everyday Maude specifications. From a programmer's perspective, it would also be advantageous to have a concise way to specify an operation that is based on a higher-order function such as map, via a labeling mechanism analogous to the one currently employed to declare equational attributes of operators. For example, to transform a list of `Nat`s into the corresponding list of `String`s, it would be ideal to write a definition such as

```
op NatsToStrings : List{Nat} -> List{String}
    [ite : COLLECTION, map : toString].
```

In this case, the user explicitly and in a very clear way declares the kind of higher-order transformation that the operator `NatsToStrings` is performing.

References

[1] J. Meseguer, "Conditioned rewriting logic as a united model of concurrency," *Theor. Comput. Sci.*, vol. 96, no. 1, pp. 73–155, 1992.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*, ser. LNCS. Springer, 2007, vol. 4350.

[3] V. Rusu and A. Arusoaie, "Executing and verifying higher-order functional-imperative programs in maude," *Journal of Logical and Algebraic Methods in Programming*, vol. 93, pp. 68–91, 2017.

[4] R. Bruni and J. Meseguer, "Generalized rewrite theories," in *Automata, Languages and Programming*, J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, Eds. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 252–266.

[5] J. Meseguer, "Twenty years of rewriting logic," *J. Log. Algebraic Methods Program.*, vol. 81, no. 7-8, pp. 721–781, 2012.

[6] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. L. Talcott, "Programming and symbolic computation in maude," *J. Log. Algebraic Methods Program.*, vol. 110, 2020.

[7] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer, "Specification and proof in membership equational logic," *Theoretical Computer Science*, vol. 236, no. 1, pp. 35–132, 2000.

[8] M. Stehr and J. Meseguer, "Pure type systems in rewriting logic: Specifying typed higher-order languages in a first-order logical framework," in *From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl*, ser. LNCS, O. Owe, S. Krogdahl, and T. Lyche, Eds., vol. 2635. Springer, 2004, pp. 334–375.

[9] M. Fernández and J. Meseguer, "Formalizing languages with binding operators in rewriting logic," in *Proc. of PPDP'25*, M. Biernacka, C. Olarte, F. Ricca, and J. Cheney, Eds. ACM, 2025, pp. 14:1–14:15.

[10] K. Bae, C. Olarte, and P. C. Ölveczky, "Modeling and analyzing real-time systems in rewriting logic," in *Concurrent Programming, Open Systems and Formal Methods - Essays Dedicated to Gul Agha to Celebrate His Scientific Career*, ser. LNCS, J. Meseguer, C. A. Varela, and N. Venkatasubramanian, Eds., vol. 16120. Springer, 2026, pp. 494–535.

[11] C. Olarte, E. Pimentel, and C. Rocha, "A rewriting logic approach to specification, proof-search, and meta-proofs in sequent systems," *J. Log. Algebraic Methods Program.*, vol. 130, p. 100827, 2023.