On-demand evaluation for Maude [1]

Francisco Durán [a]   Santiago Escobar [b]   Salvador Lucas [b]

[a] *LCC, Universidad de Málaga, Campus de Teatinos, Málaga, Spain.*
`duran@lcc.uma.es`

[b] *DSIC, UPV, Camino de Vera s/n, 46022 Valencia, Spain.*
{`sescobar,slucas`}`@dsic.upv.es`

Abstract

Strategy annotations provide a simple mechanism for introducing some laziness in the evaluation of expressions. As an eager programming language, Maude can take advantage of them and, in fact, they are part of the language. Maude strategy annotations are lists of non-negative integers associated to function symbols which specify the ordering in which the arguments are (eventually) evaluated in function calls. A positive index enables the evaluation of an argument whereas 'zero' means that the function call has to be attempted. The use of negative indices has been proposed to express *evaluation on-demand*, where the *demand* is an attempt to match an argument term with the left-hand side of a rewrite rule. In this paper we show how to furnish Maude with the ability of dealing with on-demand strategy annotations.

> *Key words:* Declarative programming, Maude, reflection, demandedness, on-demand strategy annotations.

1   Introduction

Handling infinite objects is a typical feature of lazy (functional) languages. Although reductions in Maude [5,6] are basically *innermost* (or eager), Maude is able to exhibit a similar behavior by using *strategy annotations* [18]. Maude strategy annotations are lists of non-negative integers associated to function symbols which specify the ordering in which the arguments are (eventually) evaluated in function calls: when considering a function call $f(t_1, \ldots, t_k)$, only the arguments whose indices are present as *positive* integers in the local strategy $(i_1 \cdots i_n)$ for f are evaluated (following the specified ordering). If 0 is

found, a reduction step on the whole term $f(t_1, \ldots, t_k)$ is attempted. In fact, Maude gives a strategy annotation $(1 \ 2 \ \cdots \ k \ 0)$ to each symbol f without an explicit strategy annotation.

Example 1.1 Consider the following modules `LAZY-NAT` and `LIST-NAT` defining sorts `Nat` and `LNat`, and symbols `0` and `s` for defining natural numbers, and symbols `nil` (the empty list) and `_._` for the construction of lists.

```
fmod LAZY-NAT is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat [strat (0)] .
  op _+_ : Nat Nat -> Nat .
  vars M N : Nat .
  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
endfm

fmod LIST-NAT is
  pr LAZY-NAT .
  sorts LNat .
  subsort Nat < LNat .
  op _._ : Nat LNat -> LNat [strat (1 0)] .
  op nil : -> LNat .
  op nats : -> LNat .
  op incr : LNat -> LNat .
  op length : LNat -> Nat .
  vars X Y : Nat .    vars XS YS : LNat .
  eq incr(X . XS) = s(X) . incr(XS) .
  eq nats = 0 . incr(nats) .
  eq length(nil) = 0 .
  eq length(X . XS) = s(length(XS)) .
endfm
```

Strategy annotations can often improve the termination behavior of programs (by pruning all infinite rewrite sequences starting from any expression). In the example above, the strategies `(0)` and `(1 0)` for symbols `s` and `_._`, respectively, guarantee that the resulting program is terminating [2] (note that both strategies are necessary for such a proof of termination). Strategy annotations can also improve the efficiency of computations (e.g., by reducing the number of attempted matchings or avoiding useless or duplicated reductions) [11].

Nevertheless, the absence of some indices in the local strategies can also jeopardize the ability of such strategies to compute normal forms. For in-

---

[2] The termination of the specification can be formally proved by using the tool MU-TERM, see http://www.dsic.upv.es/~slucas/csr/termination/muterm.

stance, the evaluation of the expression `s(0) + s(0)` w.r.t. Example 1.1 using Maude [3] yields the following:

```
Maude> (red s(0) + s(0) .)
result Nat: s(0 + s(0))
```

Due to the annotation (0) for the symbol `s`, the contraction of the redex `0 + s(0)` is not possible and the evaluation stops here.

The handicaps, regarding correctness and completeness of computations, of using (only) positive annotations are discussed in, e.g., [1,2,16,20,21], and a number of solutions have been proposed:

 (i) Performing a *layered normalization*: when the evaluation stops due to the replacement restrictions introduced by the strategy annotations, it is resumed over concrete inner parts of the resulting expression until the normal form is reached (if any) [17];

 (ii) transform the program to obtain a different one which is able to obtain sufficiently interesting outputs (e.g., constructor terms) [2]; and

(iii) use strategy annotations with *negative* indices which allows for some extra evaluation *on-demand*, where the *demand* is an attempt to match an argument term with the left-hand side of a rewrite rule [20,21,1].

In [8], we have introduced two new commands (`norm` and `eval`) to make techniques i and ii available for the execution of Maude programs. In this paper we show how we have brought on-demand strategies into Maude. Before entering into details, we show how negative indices can improve Maude strategy annotations.

Example 1.2 (Continuing Example 1.1) The following `NATS-TO-BIN` module implements the binary encoding of natural numbers as lists of 0 and 1 (starting from the least significant bit).

```
fmod NATS-TO-BIN is
  ex LAZY-NAT .
  pr LIST-NAT .
  op 1 : -> Nat .
  op natToBin : Nat -> LNat .
  op natToBin2 : Nat Nat -> LNat .
  vars M N X : Nat .   vars XS YS : LNat .
  eq natToBin2(0, 0) = 0 .
  eq natToBin2(0, M) = 0 . natToBin(M) .
  eq natToBin2(s(0), 0) = 1 .
  eq natToBin2(s(0), M) = 1 . natToBin(M) .
  eq natToBin2(s(s(N)), M) = natToBin2(N, s(M)) .
  eq natToBin(N) = natToBin2(N, 0) .
endfm
```

---

[3] The Maude 2.1 interpreter [6] is available at http://maude.cs.uiuc.edu.

The evaluation of the expression `natToBin(s(0) + s(0))` should yield the binary representation of 2. However, we get:

```
Maude> (red natToBin(s(0) + s(0)) .)
result LNat: natToBin2(s(0 + s(0)), 0)
```

The problem is that the current strategy annotations disallow the evaluation of subexpression `0 + s(0)` in `natToBin2(s(0 + s(0)), 0)`, thus disabling the application of the last equation for `natToBin2`. The use of the command `norm` introduced in [8] does not solve this problem, since it just normalizes non-reduced subexpressions:

```
Maude> (norm natToBin(s(0) + s(0)) .)
result LNat: natToBin2(s(s(0)), 0)
```

As we show below, on-demand strategy annotations can solve this problem. In fact, the use of the strategy `(-1 0)` for symbol `s`, declaring its first argument as evaluable only on-demand, permits to recover the desired behavior while keeping termination of the program (see Examples 4.1 and 4.2 below).

In this paper, we furnish Maude with the ability of dealing with on-demand strategy annotations. The reflective capabilities of Maude are the key for building such language extensions, which turn out to be very simple to use thanks to the infrastructure provided by Full Maude. Full Maude is an extension of Maude written in Maude itself, that endows Maude with notation for object-oriented modules and with a powerful and extensible module algebra [5]. Its design, and the level of abstraction at which it is given, make of it an excellent metalevel tool to test and experiment with features and capabilities not present in (Core) Maude [9,10,5]. We make use of the extensibility and flexibility of Full Maude to permit the use of both `red` (the usual evaluation command of Maude) and `norm` (introduced in [8]) with Maude programs using on-demand strategy annotations.

2 On-demand evaluation strategy

As explained in the introduction, the absence of some indices in the local strategies of Maude programs can jeopardize the ability of such strategies to compute normal forms. In [20,21,1], *negative* indices are proposed to indicate those arguments that should be evaluated only 'on-demand', where the 'demand' is an attempt to match an argument term with the left-hand side of a rewrite rule [21]. For instance, the evaluation of the subterm `0 + s(0)` of the term `natToBin2(s(0 + s(0)), 0)` in Example 1.2 is *demanded* by the last equation for symbol `natToBin2`, i.e., by its left-hand side `natToBin2(s(s(N)), M)`: the argument of the outermost occurrence of the symbol `s` in `natToBin2(s(0 + s(0)), 0)` is rooted by a defined function symbol, `_+_`, whereas the corresponding operator in the left-hand side is `s`. Thus, before being able to apply the rule, we have to further evaluate `0 + s(0)`.

4

As for our running example, we may conclude that the evaluation with (only) positive annotations either enters in an infinite derivation —e.g., for the term `length(nats)`, with the strategy `(1 0)` for symbol `s`— or does not provide the intended normal form —e.g., with the strategy `(0)` for symbol `s`, see Example 1.2—. The strategy `(-1 0)`, however, gives an appropriate local strategy for symbol `s`, since it makes its argument to be evaluated only "on demand". Then, the evaluation of the expression `natToBin(s(0) + s(0))` under the strategy `(-1 0)` for `s` is able to reduce the symbol `natToBin2`, and to remove it from the top position, thus obtaining a head-normal form (see Example 4.1 below). This also permits to use the resulting expression as the starting point of a layered evaluation (with `norm`) leading to the normal form (see Example 4.2 below). Note that this is achieved without entering in a non-terminating evaluation. We refer the reader to [12] for a recent and detailed discussion about the use of on-demand strategy annotations in programming.

In this paper, we follow the computational model defined in [1] for dealing with negative annotations. A local strategy for a k-ary symbol $f \in \mathcal{F}$ is a sequence $\varphi(f)$ of integers in $\{-k, \ldots, -1, 0, 1, \ldots, k\}$, which are given inside parentheses. A mapping φ that associates a local strategy $\varphi(f)$ to every $f \in \mathcal{F}$ is called an E-strategy map [20]. In order to evaluate an expression e, each symbol in e is conveniently annotated according to the E-strategy map. The evaluation of the annotated expression takes a term and the strategy associated to its top symbol, and then proceeds by considering the annotations of such a strategy sequentially [1]: if a positive argument index is provided, then the evaluation jumps to the subterm at such argument position; if a negative argument index is provided, then the index is consumed but nothing is done; if a zero is found, then we try to find a rule to be applied on such a term. If no rule can be applied, then we proceed to perform their (demanded) evaluation, that is, we try to reduce one of the subterms in positions with (consumed or present) negative indices. All consumed indices (positive and negative) are kept associated to each symbol in the term using an extra strategy list, so that demanded positions can be searched. See [1] for a formal description of the procedure and for details about why the memory list is necessary compared to other frameworks for negative annotations as OBJ3 [15] and CafeOBJ [20,21].

In this paper, we do not consider AC symbols or rules with non-linear left-hand side. Furthermore, strategy annotations are explicitly prohibited for AC symbols (see [13,14]) and the completeness of evaluation with strategy annotations is only guaranteed for linear left-hand sides and constructor-based programs (see [17,1]).

3 Reflection and the META-LEVEL module

Maude's design and implementation systematically exploits the reflective capabilities of rewriting logic [5], providing key features of the universal theory in its built-in META-LEVEL module. In particular, META-LEVEL has sorts Term

and `Module`, so that the representations of a term t and of a module $\mathcal{R}$ are, respectively, a term $\overline{t}$ of sort `Term` and a term $\overline{\mathcal{R}}$ of sort `Module`.

The basic cases in the representation of terms are obtained by subsorts `Constant` and `Variable` of the sort `Qid` of quoted identifiers. Constants are quoted identifiers that contain the name of the constant and its type separated by a dot, e.g., `'0.Nat`. Similarly, variables contain their name and type separated by a colon, e.g., `'N:Nat`. Then, a term is constructed in the usual way, by applying an operator symbol to a list of terms.

```
subsorts Constant Variable < Qid Term .
subsort Term < TermList .
op _,_ : TermList TermList -> TermList [ctor assoc] .
op _[_] : Qid TermList -> Term [ctor] .
```

For example, the term `natToBin2(s(s(0)), 0)` of sort `LNat` in the module `NATS-TO-BIN` is metarepresented as `'natToBin2['s['s['0.Nat]], '0.Nat]`.

The `META-LEVEL` module also includes declarations for metarepresenting modules. For example, a functional module can be represented as a term of sort `Module` using the following operator.

```
op fmod_is_sorts_.____endfm : Qid ImportList SortSet
  SubsortDeclSet OpDeclSet MembAxSet EquationSet
  -> FModule [ctor] .
```

Similar declarations allow us to represent the different types of declarations we can find in a module.

The module `META-LEVEL` also provides key metalevel functions for rewriting and evaluating terms at the metalevel, namely, `metaApply`, `metaRewrite`, `metaReduce`, etc., and also generic parsing and pretty printing functions `metaParse` and `metaPrettyPrint` [7,5]. For example, the function `metaReduce` takes as arguments the representation of a module $\mathcal{R}$ and the representation of a term t in that module:

```
op metaReduce : Module Term -> [ResultPair] .
op {_,_} : Term Type -> ResultPair [ctor] .
```

`metaReduce` returns the representation of the fully reduced form of the term t using the equations in $\mathcal{R}$, together with its corresponding sort or kind.

All these functionalities are very useful for metaprogramming, and in particular when building formal tools. Moreover, Full Maude provides a powerful setting in which additional facilities are available, making the addition of new commands or the redefinition of previous ones, as in this paper, simpler. The specification of Full Maude and its execution environment can then be used as the infrastructure on which building new features.

4 Extending Full Maude to handle on-demand strategy annotations

We provide the reduction of terms taking into account on-demand annotations as a redefinition of the usual evaluation command `red` of Maude (which considers only positive annotations).

Example 4.1 Consider the specification resulting from replacing in Example 1.2 the declaration of the operator `s` by this other one:

```
op s : Nat -> Nat [strat (-1 0)] .
```

The on-demand evaluation of `natToBin(s(0) + s(0))` obtains a head-normal form:

```
Maude> (red natToBin(s(0) + s(0)) .)
result LNat : 0 . natToBin(s(0))
```

As for other commands in Full Maude, we may define the actions to take when the new commands are used by defining its corresponding meta-function. For instance, a `red` command is executed by appropriately calling the metalevel `metaReduce` function. In order to furnish Maude with on-demand evaluation we provide a new metalevel operation `metaReduceOnDemand` which extends the reflective and metalevel capabilities of Maude, as explained in Section 3. The operation `metaReduceOnDemand` takes arguments of sorts `Module`, `OpDeclSet` and `Term`, and returns a term of sort `ResultPair`. Its arguments represent, respectively, the module on which the reduction takes place, the operation declarations in such a module, and the term to be reduced. The result returned is as the one given by `metaReduce` (see Section 3). Note that (Core) Maude cannot handle negative annotations, and therefore, the function takes a valid module, i.e. a module without negative annotations, and the set of operation declarations with any kind of annotation. The redefined command `red` must then select between `metaReduce` and `metaReduceOnDemand` depending on whether negative annotations are present or not.

Basically, `metaReduceOnDemand` calls the auxiliary function `procStrat` which is the function that really processes the strategy list associated to the top symbol of the term.

```
var M : Module .   var OPDS : OpDeclSet .  var T T' : Term .

op metaReduceOnDemand : Module OpDeclSet Term
   -> [ResultPair] .
op procStrat : Module OpDeclSet AnnTerm -> AnnTerm .

ceq metaReduceOnDemand(M, OPDS, T)
  = {T', leastSort(M, T')}
  if T' := erase(procStrat(M, OPDS, annotate(M, OPDS, T))) .
```

7

In order to include annotations into Maude's representation of terms, we transform the Maude's metalevel sort `Term` into a sort `AnnTerm` (of annotated terms), where symbols are equipped with a memory list and a strategy list (as explained in Section 2). Furthermore, we provide two functions: `annotate` and `erase` to move between the sorts `Term` and `AnnTerm`.

```
sorts AnnVariable AnnTerm AnnTermList .
subsorts AnnVariable < AnnTerm < AnnTermList .
op _{} : Variable -> AnnVariable .
op _{_} : Constant IntListNil -> AnnTerm .
op _{_|_}[_] : Qid IntListNil IntListNil AnnTermList
               -> AnnTerm .
op _,_ : AnnTermList AnnTermList -> AnnTermList [assoc] .

op annotate : Module OpDeclSet TermList -> AnnTermList .
op erase : AnnTermList -> TermList .
```

The function `procStrat` follows the description given in Section 2 when processing the strategy list associated to the top symbol of the term to evaluate. When a positive index is found, the evaluation of such argument is forced, and the positive index is moved from the strategy list (right component) to the memory list (left component) of the top symbol. For example, the equation for an annotated term rooted by a symbol with arity greater than 0 is as follows.

```
var N N' : Int .    var NL NL' : IntListNil .
var F : Qid .       var ATL : AnnTermList .

ceq procStrat(M, OPDS, F{NL | N NL'}[ATL])
  = procStrat(M, OPDS,
      F{NL @@ N | NL'}[procStratSel(M, OPDS, ATL, 1, N)])
  if N > 0 .
```

When a negative index is found, no evaluation in that argument is started, and the negative index is moved from the strategy list (right component) to the memory list (left component).

```
ceq procStrat(M, OPDS, F{NL | N NL'}[ATL])
  = procStrat(M, OPDS, F{NL @@ N | NL'}[ATL] )
  if N < 0 .
```

When an index 0 is found, the function `procStrat` attempts to match the term against the left-hand sides of the rules using the metalevel function `metaApply`. [4] If there is a match, then the rule is applied. If no match is obtained, then we determine if any demanded position exists using the function `procStratOD`, which performs a matching algorithm to detect which positions

---

[4] The function `metaApply` applies only rules, and therefore equations must be turned into rules before `metaApply` is applied.

under negative annotations are actually demanded by some rule (see [1] for details). If a demanded position exists, then the evaluation of such a position is started, and then we will retry the matching against the left-hand sides of the rules after the evaluation is completed. If no demanded position exists, the current index 0 is removed from the strategy list and the rest of the strategy list is considered.

```
var MA : ResultTriple? .

ceq procStrat(M, OPDS, F{NL | 0 NL'}[ATL])
  = if MA == failure
    then procStratOD(M, OPDS, F{NL | 0 NL'}[ATL])
    else procStrat(M, OPDS, annotate(M, OPDS, getTerm(MA)))
    fi
  if MA := metaApply(moveEqsToRls(M), F[erase(ATL)],
                     'on-demand, none, 0).
```

When the function procStratOD is executed, i.e. when a demanded position is being searched, the computational model of [1] specifies that the search order defined by the position order in the strategy must be followed, i.e. if (-1 -2 0) is the strategy for symbol _._, then any demanded subterm under the first argument would be selected first, despite any demanded subterm under the second argument (see [1] for details).

Once implemented the function metaReduceOnDemand, we need to redefine parts of Full Maude so that the command red can be able to execute metaReduce or metaReduceOnDemand. There is no need to define a new command and extend Full Maude to accept that command, as it was done for norm and eval commands in [8]. We just need to modify the way the red command is processed.

In the current version of Maude, input/output is accomplished by the predefined LOOP-MODE module, which provides a generic read-eval-print loop. In the case of Full Maude, the persistent state of the loop is given by a single object of class Database which maintains the database of the system. This object has an attribute db, to keep the actual database in which all the modules being entered are stored (a set of records), an attribute default, to keep the identifier of the current module by default, and attributes input and output to simplify the communication of the read-eval-print loop given by the LOOP-MODE module with the database. Using the notation for classes in object-oriented modules we can declare such a class as follows:

```
class DatabaseClass | db : Database,    default : ModName,
                      input : TermList, output : QidList .
```

The state of the read-eval-print loop is then given by an object of class DatabaseClass. In the case of Full Maude, the handling of the read-eval-print loop is defined in the modules DATABASE-HANDLING and FULL-MAUDE.

The module FULL-MAUDE includes the rules to initialize the loop (rule

init), and to specify the communication between the loop—the input/output of the system—and the database (rules in and out). Depending on the kind of input that the database receives, its state will be changed, or some output will be generated. To parse some input using the built-in function metaParse, Full Maude needs the metarepresentation of the signature in which the input is going to be parsed. In Full Maude, such a grammar is provided by the FULL-MAUDE-SIGN module, in which we can find the appropriate declarations so that any valid input, namely modules, theories, views, and commands, can be parsed. Since we do not want to change the grammar FULL-MAUDE-SIGN, which is used for parsing the inputs, we do not need to change the FULL-MAUDE module.

The module DATABASE-HANDLING defines the behavior of the database upon new entries. The behavior associated to commands is managed by rules describing transitions which call the function procCommand. For example, the rule defining what to do when the red command is received is as follows.

```
rl [red] :
   < O : X@Database | db : DB, input : ('red_.[T]),
      output : nil, default : MN, Atts >
   => < O : X@Database | db : DB, input : nilTermList,
         output : procCommand('red_.[T], MN, DB),
         default : MN, Atts > .
```

When a red command is entered, the parsing of the input returns a term of the form red_.[T], where T is a variable of sort T representing a bubble. The result of the parsing is placed in the input attribute of the database object. The function procCommand specifies what to do when the term red_.[T] is received, with MN and DB variables with values the name of the current default module and the state of the database, respectively. In the original case of the command red, procCommand calls the function procRed with the appropriate arguments, namely the name of the default module, the flatten module itself, the bubble representing the argument of the command, the variables in the default module, and the database. Note that depending on whether the default module is a built-in or not, and whether it is compiled or not, procCommand will do different things, so that the arguments for procRed are obtained. In the redefinition for command red, procCommand calls a new function procReduceOnDemand which redefines procRed.

```
eq procCommand('red_.['bubble[T]], MN, DB)
  = if MN inModNameSet builtIns
     then procReduceOnDemand(MN, DUMMY(MN), 'bubble[T],
            none, DB)
     else if compiledUnit(MN, DB)
        then procReduceOnDemand(MN, getFlatUnit(MN, DB),
              'bubble[T], getVbles(MN, DB), DB)
        else procReduceOnDemand(MN,
```

```
                getFlatUnit(MN, evalModExp(MN, DB)),
                'bubble[T], getVbles(MN, evalModExp(MN, DB)),
                evalModExp(MN, DB))
        fi
    fi .
```

The function procReduceOnDemand is in charge of evaluating the bubble given as argument of the red command, calling the function metaReduce or metaReduceOnDemand, and then preparing the results (a list of quoted identifiers that will be passed to the output channel of the read-eval-print loop to be shown to the user). The function procReduceOnDemand detects whether negative annotations are present in the module or not [5] (using the function noNegAnns), then calling metaReduceOnDemand or metaReduce. As said above, since Core Maude does not accept strategies with negative annotations, the function procReduceOnDemand must call the function metaReduceOnDemand with the module without such negative annotations (remNegAnns is in charge of removing them) and the operator declarations with them. Finally, the equations defining procReduceOnDemand are as follows.

```
op procReduceOnDemand : ModExp Module Term OpDeclSet Database
    -> QidList .
ceq procReduceOnDemand(MN, M, T, VDS, DB)
*** No negative annotation -> Use metalevel metaReduce
  = if RP? :: ResultPair
    then ('\b 'reduce 'in
          ...
    else ('\r 'Error: '\o 'Incorrect 'command. '\n)
    fi
  if noNegAnns(getOps(M))
        ...
    /\ TM := solveBubblesRed(T, remNegAnns(M), B, VDS, DB)
    /\ RP? := metaReduce(getModule(TM), getTerm(TM)) .

ceq procReduceOnDemand(MN, M, T, VDS, DB)
*** Negative annotations -> Use metalevel metaReduceOnDemand
  = if RP? :: ResultPair
    then ('\b 'reduce 'on-demand 'in
          ...
    else ('\r 'Error: '\o 'Incorrect 'command. '\n)
    fi
  if not noNegAnns(getOps(M))
        ...
    /\ TM := solveBubblesRed(T, remNegAnns(M), B, VDS, DB)
```

---

[5] The "classical" Maude evaluation is not affected when only positive annotations are provided.

```
/\ RP? := metaReduceOnDemand(getModule(TM), getOps(M),
            getTerm(TM)) .
```

4.1 Extending Full Maude with on-demand strategy annotations to layered normalization

As explained along the paper, our goal is to provide appropriate normal forms to programs with strategy annotations. However, the redefinition of command red is not able to provide the normal form 0 . 1 for the program in Example 1.2, since the annotation 2 is missing in the strategy list for symbol _._ (see the output of the red command in Example 4.1). However, as it was explained in Section 1, this concrete problem is solved using either a layered normalization, or a transformation. In this section, we redefine the command norm of [8] to perform a layered normalization of the output given by the on-demand evaluation previously presented.

Example 4.2 Consider the modules of Example 4.1. The redefinition of command norm now is able to provide the intended value associated to the expression natToBin(s(0) + s(0)).

```
Maude> (norm natToBin(s(0) + s(0)) .)
result LNat : 0 . 1
```

The redefinition of command norm is almost identical to the implementation of the command norm given in [8]. We do not give the details here, but basically, the idea is that we keep the metalevel function metaNorm and define a new metalevel function metaNormOnDemand which calls metaReduceOnDemand instead of metaReduce to reduce the initial term.

```
eq metaNormODRed(M, OPDS, T)
  = procStratOD(M, getTerm(metaReduceOnDemand(M, OPDS, T)),
      OPDS) .
```

We refer the reader to [8] for details about the implementation of the norm command. Note that it is also necessary to perform similar changes to those explained in Section 4:

- we redefine procCommand to call a new function procNormOnDemand, which redefines procNorm, when the term norm_.[T] is received;

- the function procNormOnDemand calls metaNorm or metaNormOnDemand depending on whether negative annotations are present or not (using again the function noNegAnns).

5 Conclusions

We have used Full Maude to furnish Maude with the ability to perform on-demand evaluations, a more sophisticated form of lazy behavior for languages such as Maude. We make use of the extensibility and flexibility of Full Maude

to permit the use of both `red` (the usual evaluation command of Maude) and `norm` (introduced in [8]) with Maude programs using on-demand strategy annotations. The full specification is available at

<div align="center">

http://www.dsic.upv.es/users/elp/toolsMaude

</div>

These features have been integrated into Full Maude, making them available inside its programming environment. The high level at which the specification (implementation) of Full Maude is given makes this approach particularly attractive when compared to conventional implementations (see e.g. [3]). The flexibility and extensibility that Full Maude affords has made the extension quite simple and in a very short time.

It is worth noting however that our prototype of on-demand evaluation is not comparable in efficiency to other implementations of evaluation with negative annotations such as in CafeOBJ [6] or OnDemandOBJ [7] . The goal of this piece of work is not to provide a competitive implementation, but to provide on-demand evaluation for a language such as Maude. Note that OnDemandOBJ does not include all the capabilities of Maude, and that the computational model of CafeOBJ for dealing with negative annotations has some drawbacks (see [1]). In fact, it is not fair looking at it as an implementation of the on-demand strategies, not even as a prototype. It should be seen as an executable specification of it, closer to its mathematical definition (given in [1]) than to its implementation. Although a more efficient executable specification/implementation of the on-demand evaluation following a similar approach could be given, we are convinced that a direct implementation of the on-demand evaluation into Maude is desirable.

Finally, as future work we plan to consider whether it is possible to express the on-demand evaluation strategy in terms of strategy languages such as [19,4].

References

[1] M. Alpuente, S. Escobar, B. Gramlich, and S. Lucas. Improving On-Demand Strategy Annotations. In M. Baaz and A. Voronkov, editors, *Proc. 9th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'02*, LNAI 2514:1-18, Springer, 2002.

[2] M. Alpuente, S. Escobar, and S. Lucas. Correct and complete (positive) strategy annotations for OBJ. In F. Gadducci and U. Montanari, editors, *Proc. of the 4th International Workshop on Rewriting Logic and its Applications, WRLA'02*, ENTCS 71. Elsevier, 2004.

[3] M. Alpuente, S. Escobar, and S. Lucas. OnDemandOBJ: A Laboratory for Strategy Annotations. In J.-L. Giavitto and P.-E. Moreau, editors, *Proc. of*

---

the 4th International Workshop on Rule-Based Programming, RULE'03, ENTCS 86(2). Elsevier, 2003.

[4] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: A functional semantics. International Journal of Foundations of Computer Science, 12:69 95, 2001.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science* 285(2):187-243, 2002.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In R. Nieuwenhuis, editor, Proc. of *14th International Conference on Rewriting Techniques and Applications, RTA'03*, LNCS 2706:76-87, Springer, 2003.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 manual. Available in http://maude.cs.uiuc.edu, 2003.

[8] F. Durán, S. Escobar, and S. Lucas. New evaluation commands for Maude within Full Maude. In N. Martí-Oliet, editor, *Proc. of the 5th International Workshop on Rewriting Logic and its Applications, WRLA'04*, ENTCS to appear 2004.

[9] F. Durán and J. Meseguer. An extensible module algebra for Maude. In C. Kirchner and H. Kirchner, editors, *Proceedings of 2nd International Workshop on Rewriting Logic and its Applications, WRLA'98*, ENTCS 15. Elsevier, 1998.

[10] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language.* PhD thesis, Universidad de Málaga, June 1999.

[11] S. Eker. Term Rewriting with Operator Evaluation Strategies. *Electronic Notes in Theoretical Computer Science*, volume 15, 20 pages, 1998. In C. Kirchner and H. Kirchner, editors, *Proceedings of 2nd International Workshop on Rewriting Logic and its Applications, WRLA'98*, ENTCS 15. Elsevier, 1998.

[12] S. Escobar. *Strategies and Analysis Techniques for Functional Program Optimization.* PhD Thesis, Universidad Politécnica de Valencia, October 2003.

[13] M.C.F. Ferreira and A.L. Ribeiro. Context-Sensitive AC-Rewriting. In *Proc. of the 10th International Conference on Rewriting Techniques and Applications, RTA'99*, LNCS 1631:173-181. Springer, 1999.

[14] J. Giesl and A. Middeldorp. Transformation Techniques for Context-Sensitive Rewrite Systems. Journal of Functional Programming, to appear, 2004.

[15] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action.* Kluwer, 2000.

[16] S. Lucas. Termination of on-demand rewriting and termination of OBJ programs. In *Proc. of 3rd International Conference on Principles and Practice of Declarative Programming, PPDP'01*, pages 82-93, ACM Press, 2001.

[17] S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):293-343, 2002.

[18] S. Lucas. Semantics of programs with strategy annotations. Technical Report DSIC-II/08/03, DSIC, Universidad Politécnica de Valencia, 2003.

[19] N. Martí-Oliet, J. Meseguer, A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, editor, *Proc. of the 5th International Workshop on Rewriting Logic and its Applications, WRLA'04*, ENTCS to appear 2004.

[20] M. Nakamura and K. Ogata. The evaluation strategy for head normal form with and without on-demand flags. In K. Futatsugi, editor, *Proc. of 3rd International Workshop on Rewriting Logic and its Applications, WRLA'00*, ENTCS 36. Elsevier, 2001.

[21] K. Ogata and K. Futatsugi. Operational semantics of rewriting with the on-demand evaluation strategy. In *Proc. of 2000 International Symposium on Applied Computing, SAC'00*, pages 756–763. ACM Press, 2000.