Towards (constructor) normal forms for Maude within Full Maude[⋆]

Francisco Durán[1], Santiago Escobar[2], and Salvador Lucas[2]

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

[2] Universidad Politécnica de Valencia, Valencia, Spain
{`sescobar`,`slucas`}`@dsic.upv.es`

Abstract. Maude is able to deal with infinite data structures and avoid infinite computations by using *strategy annotations*, that is, positive indices that indicate which positions can be evaluated. However, they can eventually make the computation of the normal form(s) of some input expressions impossible. In [6,7], we used Full Maude to implement two new commands `norm` and `eval` which furnish Maude with the ability to compute (constructor) normal forms of initial expressions even when the use of strategy annotations together with the built-in computation strategy of Maude is not able to obtain them. These commands were integrated into Full Maude, making them available inside the programming environment like any other of its commands. Moreover, the type of annotations allowed was extended, giving to Maude the ability of dealing with on-demand strategy annotations, that is, negative indices that express *evaluation on-demand*, where the *demand* is a failed attempt to match an argument term with the left-hand side of a rewrite rule. In this paper, we recall these new commands and extensions.

1 Introduction

Handling infinite objects is a typical feature of lazy (functional) languages. Although reductions in Maude [4,5] are basically *innermost* (or eager), Maude is able to exhibit a similar behavior by using *strategy annotations* [17]. Maude strategy annotations are lists of non-negative integers associated to function symbols which specify the order in which the arguments are (eventually) evaluated in function calls. Specifically, 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 order). Besides, if 0 is found, a reduction step on the whole term $f(t_1, \ldots, t_k)$ is attempted. Every symbol without an explicit strategy annotation receives a standard local strategy $(1\ 2\ \cdots\ k\ 0)$ by Maude, where k is the number of its arguments. Let us illustrate by an example how local strategies are used in Maude.

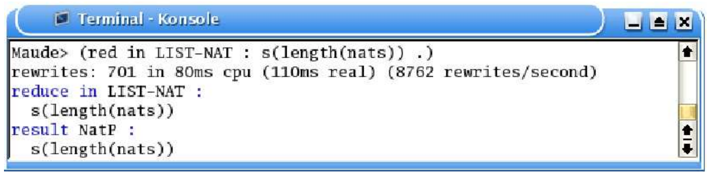
⋆ Work partially supported by CICYT under grants TIC2001-2705-C03-01 and TIC2001-2705-C03-02, by MCyT under grant HU 2003-0003, by Generalitat Valenciana under grants GRUPOS03/025 and GV04B-477, and by the EU-India Cross-Cultural Dissemination project ALA/95/23/2003/077-054.

Example 1. Consider the following Maude modules[3] `LAZY-NAT`, which provides symbols `0` and `s` for specifying natural numbers in Peano syntax (sort `NatP`), `LAZY-LIST`, with a 'polymorphic' sort `List(X)` and symbols `nil` (the empty list) and `_._` for the construction of polymorphic lists, and an instance of module `LAZY-LIST` for natural numbers, called `LIST-NAT`:

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

(fth TRIV is
   sort Elt .
 endfth)

(fmod LAZY-LIST(X :: TRIV) is
   protecting LAZY-NAT .
   sort List(X) .
   subsort X@Elt < List(X) .
   op nil : -> List(X) .
   op _._ : X@Elt List(X) -> List(X) [strat (1 0)] .
   op length : List(X) -> NatP .
   var X : X@Elt .   var XS : List(X) .
   eq length(nil) = 0 .
   eq length(X . XS) = s(length(XS)) .
 endfm)

(view NatP from TRIV to NAT is
   sort Elt to NatP .
 endv)

(fmod LIST-NAT is
   protecting LAZY-NAT .
   protecting LAZY-LIST(NatP) .
   op incr : List(NatP) -> List(NatP) .
   op nats : -> List(NatP) .
   var X : NatP .   var XS : List(NatP) .
   eq incr(X . XS) = s(X) . incr(XS) .
   eq nats = 0 . incr(nats) .
 endfm)
```
■

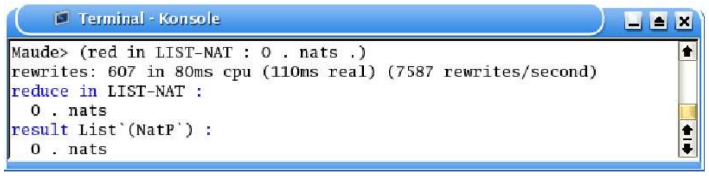
[3] Actually, we will use the Full Maude extension instead of Maude in the examples. This is denoted by enclosing each module or command with parenthesis.

Strategy annotations can often improve the termination behavior of programs, i.e., they prune all infinite rewrite sequences starting from an expression at an argument position that is not present in the local strategy of the corresponding symbol. In the example above, the strategies `(0)` and `(1 0)` for symbols `s` and `_._`, respectively, guarantee that the resulting program is terminating[4]. For instance, the absence of `1` in the local strategy of `s` stops any infinite reduction in the term `s(length(nats))`, as shown by the standard reduction command `red` in Maude[5]



```
Maude> (red in LIST-NAT : s(length(nats)) .)
rewrites: 701 in 80ms cpu (110ms real) (8762 rewrites/second)
reduce in LIST-NAT :
  s(length(nats))
result NatP :
  s(length(nats))
```

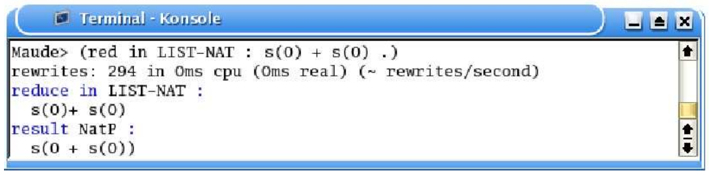
and the absence of `2` in the local strategy of `_._` stops any infinite reduction in the term `0 . nats`



```
Maude> (red in LIST-NAT : 0 . nats .)
rewrites: 607 in 80ms cpu (110ms real) (7587 rewrites/second)
reduce in LIST-NAT :
  0 . nats
result List`(NatP`) :
  0 . nats
```

Moreover, apart from improving termination, strategy annotations can also improve the efficiency of computations [10], e.g., by reducing the number of attempted matchings or avoiding useless or duplicated reductions.

Nevertheless, the absence of some indices in the local strategies can also jeopardize the ability of such strategies to compute normal forms. For instance, the evaluation of the expression `s(0) + s(0)` w.r.t. Example 1 yields the following:



```
Maude> (red in LIST-NAT : s(0) + s(0) .)
rewrites: 294 in 0ms cpu (0ms real) (~ rewrites/second)
reduce in LIST-NAT :
  s(0)+ s(0)
result NatP :
  s(0 + s(0))
```

Due to the local strategy `(0)` for the symbol `s`, the contraction of the redex `0 + s(0)` is not possible and the evaluation stops here with an expression that is not the intended value expected by the user.

The handicaps, regarding correctness and completeness of computations, of using (only) positive annotations have been pointed out in the literature, e.g. in [1,2,14,21,22], and a number of solutions have been proposed:

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

[5] The Maude 2.1 interpreter [5] is available at `http://maude.cs.uiuc.edu`.

1. 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) [15];
2. transforming the program to obtain a different one which is able to obtain sufficiently interesting outputs (e.g., constructor terms) [2]; and
3. using strategy annotations with *negative* indices which allows for some extra evaluation *on-demand*, where the *demand* is a failed attempt to match an argument term with the left-hand side of a rewrite rule [21,22,1], like the (functional) lazy evaluation strategy.

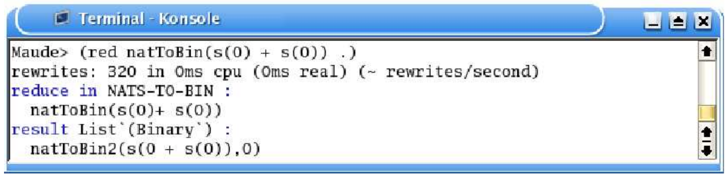
In [6], we introduced two new Maude commands (`norm` and `eval`) to make techniques 1 and 2 available for the execution of Maude programs, and in [7], we extended the original Maude command `red` and the new command `norm` to make technique 3 also available into Maude. In this paper, we recall these new commands and extensions and refer the reader to [6,7] for details.

First, let us illustrate with the following example how negative indices can also improve the behavior of Maude programs.

Example 2. Continuing Example 1. The following `NATS-TO-BIN` module implements the binary encoding of natural numbers as lists of symbols `0` and `1` of sort `Binary` (starting from the least significative bit).

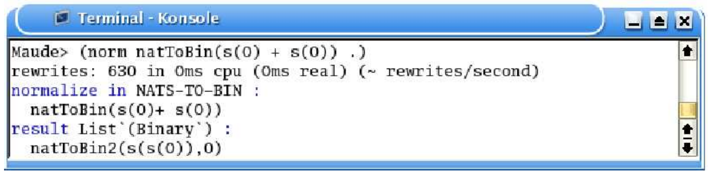
```
(fmod BINARY is
   sort Binary .
   ops 0 1 : -> Binary .
 endfm)

(view Binary from TRIV to BINARY is
   sort Elt to Binary .
 endv)

(fmod NATS-TO-BIN is
   protecting LAZY-NAT .
   protecting LAZY-LIST(Binary) .
   op natToBin : NatP -> List(Binary) .
   op natToBin2 : NatP NatP -> List(Binary) .
   vars M N : NatP .
   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)
```

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)) .)
rewrites: 320 in 0ms cpu (0ms real) (~ rewrites/second)
reduce in NATS-TO-BIN :
  natToBin(s(0)+ s(0))
result List`(Binary`) :
  natToBin2(s(0 + s(0)),0)
```

The problem is that the current local strategy `(0)` for symbol `s` disallows the evaluation of subexpression `0 + s(0)` in `natToBin2(s(0 + s(0)), 0)`, and thus disables the application of the last equation for `natToBin2`. The use of the techniques 1 or 2 (through commands `norm` and `eval` introduced in [6]) do not solve this problem, since they just normalize non-reduced subexpressions but never replace the symbol at the root position:



```
Maude> (norm natToBin(s(0) + s(0)) .)
rewrites: 630 in 0ms cpu (0ms real) (~ rewrites/second)
normalize in NATS-TO-BIN :
  natToBin(s(0)+ s(0))
result List`(Binary`) :
  natToBin2(s(s(0)),0)
```

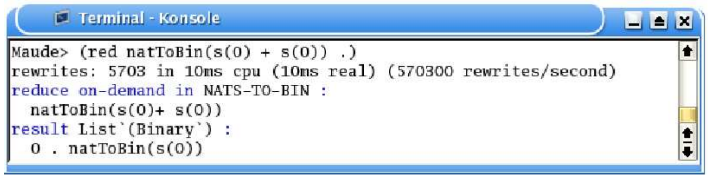
■

In [21,22,1], *negative* indices are proposed to indicate those arguments that should be evaluated only "on-demand", where the "demand" is a failed attempt to match an argument term with the left-hand side of a rewrite rule [22]. For instance, the evaluation of the subterm `0 + s(0)` of the term `natToBin2(s(0 + s(0)), 0)` in the previous example is *demanded* by the last equation for symbol `natToBin2`, i.e., by its left-hand side `natToBin2(s(s(N)), M)`. Specifically, 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)` in order to eventually remove `_+_` by `s` at the root position.

Example 3. Continuing Example 2. Consider the specification resulting from replacing the declaration of the operator `s` by the following one with a new local strategy including a negative annotation that declares the first argument of `s` as evaluable only on-demand:

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

The on-demand evaluation of `natToBin(s(0) + s(0))` (through the extended command `red` introduced in [7]) obtains the appropriate head-normal form:



```
Maude> (red natToBin(s(0) + s(0)) .)
rewrites: 5703 in 10ms cpu (10ms real) (570300 rewrites/second)
reduce on-demand in NATS-TO-BIN :
  natToBin(s(0)+ s(0))
result List`(Binary`) :
  0 . natToBin(s(0))
```

Specifically, from the expression `natToBin2(s(0 + s(0)), 0)` considered above, we have that the following evaluation step is demanded by the left-hand side `natToBin2(s(s(N)), M)`:

$$\texttt{natToBin2(s(}\underline{\texttt{0 + s(0)}}\texttt{), 0)} \to \texttt{natToBin2(s(s(0 + 0)), 0)}$$

Then, the rule with left-hand side `natToBin2(s(s(N)), M)` can be applied, which implies that no evaluation is demanded:

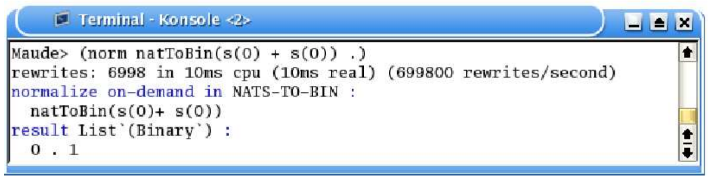
$$\underline{\texttt{natToBin2(s(s(0 + 0)), 0)}} \to \texttt{natToBin2(0 + 0, s(0))}$$

Finally, the remaining reductions leading to the expression `0 . natToBin(s(0))` are not demanded and follow from the positive local indices given above. ■

¿From the previous examples, we can infer that, in some situations, 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`—, whereas the strategy `(-1 0)` gives an appropriate local strategy for symbol `s`, since it makes its argument to be evaluated only "on demand", and thus obtains the appropriate head-normal forms while still avoiding infinite derivations.

Nevertheless, our main motivation is to provide appropriate normal forms to programs with strategy annotations, and thus the redefinition of command `red` for on-demand evaluation is not enough, since it is not able to provide the normal form `0 . 1` for the program in Example 2 because the annotation `2` is missing in the strategy list for symbol `_._` (see the output of the `red` command in Example 3). As it was explained above, this concrete problem is solved using either the layered normalization —technique 1— or a program transformation —technique 2—. And therefore, in [7], the output given by an on-demand evaluation with command `red` is used as the starting point of a layered evaluation coded into an appropriate redefinition of the command `norm` of [6] for dealing with negative indices.

Example 4. Consider the modules of Example 3. The layered normalization technique extended for on-demand annotations (through the extended command `norm` introduced in [7]) is able to provide the intended normal form associated to the expression `natToBin(s(0) + s(0))`.



```
Maude> (norm natToBin(s(0) + s(0)) .)
rewrites: 6998 in 10ms cpu (10ms real) (699800 rewrites/second)
normalize on-demand in NATS-TO-BIN :
  natToBin(s(0)+ s(0))
result List`(Binary`) :
  0 . 1
```

■

It is worth to note that all these features and appropriate evaluations are achieved without entering in non-terminating evaluations. We refer the reader to [11] for a recent and detailed discussion about the use of on-demand strategy annotations in programming.

2 New commands for Maude within Full Maude

In the following, we provide some useful insights about how these features are implemented in Full Maude. The complete specifications can be found in

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

2.1 Full Maude, reflection, and the META-LEVEL module

The reflective capabilities of Maude are the key for building these 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 [4]. 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 [8,9,4]. In fact, reflection, together with the good properties of the underlying *rewriting logic* [18] as a logical framework [20,19], makes quite easy the development of formal tools and execution environments in Maude for any logic $\mathcal{L}$ of interest, including rewriting logic itself (see e.g. [9,3]).

Maude's design and implementation systematically exploit the reflective capabilities of rewriting logic, 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.NatP`. Similarly, variables contain their name and type separated by a colon, e.g., `'N:NatP`. 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 [assoc] .
op _[_] : Qid TermList -> Term .
```

For example, the term `natToBin2(s(s(0)), 0)` of sort `List(Binary)` in the module `NATS-TO-BIN` is metarepresented as

```
'natToBin2['s['s['0.NatP]], '0.NatP]
```

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

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

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` [5,4]. 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 .
```

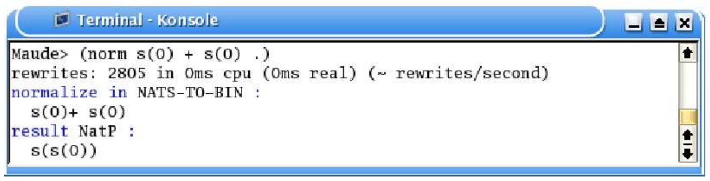
`metaReduce` returns the representation of the fully reduced form of the term t using the equations in $\mathcal{R}$, together with its corresponding sort or kind.

All these functionalities are very useful for metaprogramming. In particular, we have used the extensibility and flexibility of Full Maude to permit the definition of new commands such as `norm` and `eval` but also to adapt the usual Maude evaluation command `red` and the new command `norm` to the on-demand evaluation.

2.2 Extending Full Maude to handle the command `norm`

The *normalization via μ-normalization* procedure was introduced in [15,16] as a simple mechanism to furnish the *context-sensitive rewriting* (*CSR*) [13] with the ability to compute normal forms of initial expressions even though *CSR* itself is not able to compute them. The idea is very simple: if we are able to ensure that the normal forms computed by *CSR* are always head-normal forms, then it is safe to get into the maximal "*non-replacing*" subterms of the normal form s of a term t computed by *CSR* to (recursively) continue the computation. The technique works for left-linear, confluent TRSs $\mathcal{R}$ that use a *canonical* replacement map; see [15]. The procedure is sound in the context of Maude due to an appropriate formal connection between *CSR* and the evaluation mechanism of Maude [6].

This normalization via μ-normalization procedure has been implemented in [6] as a new command `norm` that uses the outcome of `red` to perform this layered evaluation of the initial expressions. The new command `norm` permits to obtain the intended value of many expressions, e.g. for the term `s(0) + s(0)` given above:



```
Maude> (norm s(0) + s(0) .)
rewrites: 2805 in 0ms cpu (0ms real) (~ rewrites/second)
normalize in NATS-TO-BIN :
  s(0)+ s(0)
result NatP :
  s(s(0))
```

As for other commands in Maude, we may define the actions to take when the command is used by defining its corresponding meta-function. For instance, a `red` command is executed by appropriately calling the `metaReduce` function. In the case of `norm`, we define an operation `metaNorm`. Basically, `metaNorm` calls the auxiliary function `metaNormRed`, which first reduces the term using `metaReduce`, and then proceeds recursively on each of the arguments of the resulting term. That is, for each argument of the symbol at the root of the resulting term, if the argument has evaluation restrictions, then it calls `metaNormRed`, otherwise it calls `metaNormNoRed`. `metaNormNoRed` proceeds as `metaNormRed`, but without reducing the term before going on the arguments, since it is not necessary.

2.3 Extending Full Maude to handle the command `eval`

When considering how the evaluation command `norm` works for a concrete input term t, we understand that it is interesting to isolate the replacement restrictions needed to achieve the head-normal form of t (produced by the command `red`) from the restrictions needed to get its normal form (produced by the command `norm`). In the case of constructor normal forms, we can rather consider a constructor (prefix) context $C[\,]$ of the normal form of t such that $C[\,] \in \mathcal{T}(\mathcal{B} \cup \{\Box\}, \mathcal{X})$ for some $\mathcal{B} \subseteq \mathcal{C}$. The intuitive idea is that the set of constructor symbols $\mathcal{B}$ characterizes those constructor symbols which are (or could eventually be) present in the normal form of t. Then, reductions *below* the outermost defined symbols should be performed only up to (constructor) head-evaluation (i.e., a term rooted by a symbol in $\mathcal{B}$ computed by command `red`), thus providing an incremental computation of the normal form of t. In [2], a program transformation aimed at achieving this goal was given. The key idea for the transformation is to duplicate every symbol in $\mathcal{B}_\tau$, for each sort τ in the specification that is involved in $\mathcal{B}$, to a fresh constructor symbol in the new set $\mathcal{B}'_\tau$, which does not have any restriction on its arguments. More specifically, given a sort τ, the set $\mathcal{C}^*_\tau \subseteq \mathcal{C}$ is the set of constructor symbols that can be found in constructor terms of sort τ. For instance, $\mathcal{C}^*_{\texttt{NatP}} = \{\texttt{0}, \texttt{s\_}\}$ and $\mathcal{C}^*_{\texttt{List(NatP)}} = \{\texttt{nil}, \texttt{\_.\_}, \texttt{0}, \texttt{s\_}\}$. Hence, the set $\mathcal{C}^*_\tau$ will tell us which constructor symbols must be renamed.

The renaming of the constructor symbols $c \in \mathcal{C}^*_\tau$ into new constructor symbols c' is performed by the following new rules, which are added to the original program during the transformation:

$$\texttt{quote}_{sort(c)}(c(x_1, \ldots, x_k)) \to c'(\texttt{quote}_{sort(x_1)}(x_1), \ldots, \texttt{quote}_{sort(x_k)}(x_k))$$

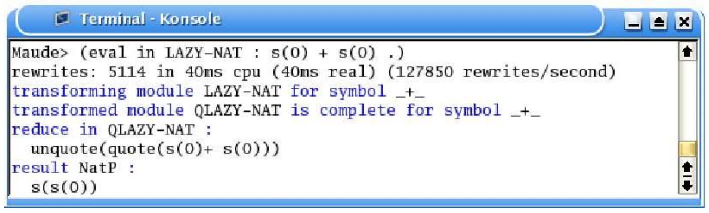
In practice, we use the overloading facilities of Maude and introduce a single (overloaded) symbol `quote`.

The evaluation of a term t would proceed by reducing $\texttt{quote}(t)$ into a term with a constructor prefix context in $\mathcal{C}'$, which do not have any evaluation restriction. Then, we perform a postprocessing that recovers the original names of the constructor symbols after each argument of a symbol $c' \in \mathcal{C}'$ has been evaluated:

$$\texttt{unquote}(c'(x_1, \ldots, x_k)) \to c(\texttt{unquote}(x_1), \ldots, \texttt{unquote}(x_k))$$

Again, the symbol `unquote` is conveniently overloaded.

In [6], we implement a new command `eval` that uses this transformation to obtain the constructor normal form (if any) associated to a given input expression t. In contrast to the command `norm`, we first transform the module and, then, we simply reduce the expression `unquote`(`quote`(t)) within the new module (using the `metaReduce` function). The new command `eval` permits to obtain the intended value of the term `s(0) + s(0)` given above:



```
Maude> (eval in LAZY-NAT : s(0) + s(0) .)
rewrites: 5114 in 40ms cpu (40ms real) (127850 rewrites/second)
transforming module LAZY-NAT for symbol _+_
transformed module QLAZY-NAT is complete for symbol _+_
reduce in QLAZY-NAT :
  unquote(quote(s(0)+ s(0)))
result NatP :
  s(s(0))
```

2.4 Extending Full Maude to handle on-demand strategy annotations

We have followed the computational model defined in [1] for dealing with negative annotations, where a local strategy for a k-ary symbol $f \in \mathcal{F}$ is a sequence of integers in $\{-k, \ldots, -1, 0, 1, \ldots, k\}$. In this computational model, each symbol in the term t to be evaluated is conveniently annotated with its local strategy, yielding a new class of annotated terms. The evaluation takes the annotated term and the strategy glued to its top symbol, and then proceeds as follows [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 at the root position when a zero is found, then we proceed to look for its (demanded) evaluations, that is, we try to reduce one of the subterms at positions under a (consumed or present) negative index. All consumed indices (positive and negative) are glued also to each symbol in the term using an extra strategy list, so that demanded positions can appropriately be searched. See [1] for a formal description.

In [7], 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). As for other commands in Full Maude, we provide a new metalevel operation `metaReduceOnDemand` which extends the reflective and metalevel capabilities of `Maude`. The redefined command `red` must then select between `metaReduce` and `metaReduceOnDemand` depending on whether negative annotations are present or not in the module associated to the expression to execute. This on-demand extension of the usual command `red` permits to produce the appropriate head-normal form of the term `natToBin(s(0) + s(0))`, as it was shown in Example 3.

It is worthy to note that although eager (or innermost) evaluation strategies correspond to local strategies of the shape (1 2 $\cdots$ k 0) (see [17] for the exact relationship) lazy (or outermost) evaluation strategies do not directly correspond to local strategies of the shape $(-1\ -2\ \cdots\ -k\ 0)$ [12]. Indeed, we plan to investigate this as future work.

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

The redefinition of command `norm` to deal with on-demand strategy annotations is almost identical to the implementation of the command `norm` given in [6] and reviewed in Section 2.2. The idea is that we keep the metalevel function `metaNorm` and define a new metalevel function `metaNormOnDemand` which calls `metaReduceOnDemand` instead of `metaReduce` when negative annotations are present in the module associated to the expression to execute. The redefinition of the command `norm` permits to obtain the intended value of the term `natToBin(s(0) + s(0))`, as it was shown in Example 4.

3 Conclusions

In this paper, we have presented two new commands `norm` and `eval` which furnish Maude with the ability to compute (constructor) normal forms of expressions even when the built-in computation strategy is not able to obtain them. We have also presented the extension of the type of annotations allowed in Maude, that gives to Maude the ability of dealing with on-demand strategy annotations, that is, negative indices that express evaluation on-demand like functional lazy evaluation.

Furthermore, the reader may note that strategy annotations have been used for many years in programming languages such as Maude but the formal analysis of computations under such annotated strategies has been recently addressed.

Acknowledgements. Thanks to the anonymous referees for their helpful remarks.

References

1. M. Alpuente, S. Escobar, B. Gramlich, and S. Lucas. Improving on-demand strategy annotations. In *Proc. of the 9th Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'02)*, LNCS 2514:1–18. Springer-Verlag, 2002.
2. M. Alpuente, S. Escobar, and S. Lucas. Correct and complete (positive) strategy annotations for OBJ. In *Proc. of the 4th Int'l Workshop on Rewriting Logic and its Applications, WRLA 2002*, ENTCS 71. Elsevier, 2003.
3. M. Clavel, F. Durán, S. Eker, J. Meseguer, and M. Stehr. Maude as a formal meta-tool. In *Proc. of the FM'99 — Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Volume II*, LNCS 1709:1684–1703. Springer-Verlag, 1999.
4. 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*, 285:187–243, 2002.

5. M. Clavel, F. Durán, S. Eker, P. Lincoln, J. Meseguer N. Martí-Oliet, , and C. Talcott. The Maude 2.0 System. In *Rewriting Techniques and Applications, 12th Int'l Conf., RTA 2003*, LNCS 2706:76–87. Springer-Verlag, 2003.
6. F. Durán, S. Escobar, and S. Lucas. New evaluation commands for Maude within Full Maude. In *5th Int'l Workshop on Rewriting Logic and its Applications, WRLA'04*, ENTCS to appear. Elsevier, 2004.
7. F. Durán, S. Escobar, and S. Lucas. On-demand evaluation for Maude. In *Proc. of the 5th Int'l Workshop on Rule-Based Programming, RULE'04*, ENTCS to appear. Elsevier, 2004.
8. F. Durán and J. Meseguer. An extensible module algebra for Maude. In *Proc. of the 2nd Int'l Workshop on Rewriting Logic and its Applications, WRLA'98*, ENTCS 15:185–206. Elsevier, 1998.
9. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language.* PhD thesis, Universidad de Málaga, Spain, June 1999.
10. S. Eker. Term rewriting with operator evaluation strategies. In *Proc. of the 2nd Int'l Workshop on Rewriting Logic and its Applications, WRLA 98*, ENTCS 15. Elsevier, 2000.
11. S. Escobar. *Strategies and Analysis Techniques for Functional Program Optimization.* PhD thesis, Universidad Politécnica de Valencia, Spain, October 2003.
12. O. Fissore, I. Gnaedig, and H. Kirchner. *Outermost ground termination.* In *Proc. of the 4th Int'l Workshop on Rewriting Logic and its Applications, WRLA 2002*, ENTCS 71. Elsevier, 2003.
13. S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1):1–61, 1998.
14. S. Lucas. Termination of on-demand rewriting and termination of obj programs. In *Proc. of the 3rd Int'l ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming, PPDP'01*, pages 82–93. ACM Press, 2001.
15. S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):294–343, 2002.
16. S. Lucas. Termination of (Canonical) Context-Sensitive Rewriting. In *Proc. of the 13th Int'l Conf. on Rewriting Techniques and Applications, RTA'02*, LNCS 2378:296–310. Springer-Verlag, 2002.
17. S. Lucas. Semantics of programs with strategy annotations. Technical Report DSIC-II/08/03, DSIC, Universidad Politécnica de Valencia, 2003.
18. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
19. J. Meseguer. Research directions in rewriting logic. In *Computational Logic, Proceedings of the NATO Advanced Study Institute on Computational Logic held in Marktoberdorf, Germany, July 29 – August 6, 1997*, volume 165 of *NATO ASI Series F: Computer and Systems Sciences*, pages 347–398. Springer-Verlag, 1998.
20. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In *Handbook of Philosophical Logic*, volume 9, pages 1–88. Kluwer Academic Publishers, 2002.
21. M. Nakamura and K. Ogata. The evaluation strategy for head normal form with and without on-demand flags. In *Proc. of the 3rd Int'l Workshop on Rewriting Logic and its Applications, WRLA 2000*, ENTCS 36. Elsevier, 2001.
22. K. Ogata and K. Futatsugi. Operational semantics of rewriting with the on-demand evaluation strategy. In *Proc. of 2000 Int'l Symposium on Applied Computing, SAC'00*, pages 756–763. ACM Press, 2000.