Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 124 (2005) 25–39
www.elsevier.com/locate/entcs

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.

Keywords: 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

[1] Work partially supported by CICYT TIC2001-2705-C03-01 and TIC2001-2705-C03-02, MCyT Acción Integrada HU 2003-0003, Agencia Valenciana de Ciencia y Tecnología GR03/025, and EU-India Cross-Cultural Dissemination project ALA/95/23/2003/077-054.

doi:10.1016/j.entcs.2004.07.013

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 anno-

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