Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2010) 165–175
www.elsevier.com/locate/entcs

A Natural Implementation of Plural Semantics in Maude[1]

Adrián Riesco and Juan Rodríguez-Hortalá

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
{ariesco,juanrh}@fdi.ucm.es

Abstract

Recently, a new semantics for non-deterministic lazy functional(-logic) programming has been presented, in which the treatment of parameter passing was different to previous proposals like call-time choice (CRWL) and run-time choice (term rewriting). There, the semantics was formalized through the πCRWL calculus, and a program transformation to simulate πCRWL with term rewriting was proposed. In the present work we use the Maude system to implement that transformation and to get an interpreter for πCRWL, thus providing a first implementation of this new semantics. Besides, in order to improve the performance of the prototype, an implementation of the natural rewriting on-demand strategy has been developed, therefore taking the first steps towards obtaining a framework for on-demand evaluation of Maude system modules.

Keywords: Language prototyping, Plural semantics, Maude, Natural rewriting, Rewriting logic.

1 Introduction

State-of-the-art implementations of functional-logic programming (*FLP*) languages (see [12] for a recent survey) use possibly non-terminating and non-confluent constructor-based term rewrite systems (*CS's*) as programs, thus defining possibly non-strict non-deterministic functions, which are one of the most distinctive features of the paradigm [11,2]. Nevertheless, although CS's can be used as a common syntactic framework for FLP and term rewriting, the behavior of current implementations of these formalisms differ fundamentally, because different semantics can be assigned to a lazy functional language after introducing non-determinism. Considering the program $\mathcal{P} = \{f(c(X)) \to d(X,X), X \ ? \ Y \to X, X \ ? \ Y \to Y\}$ and the expression $f(c(0) \ ? \ c(1))$, let us see what are the values for that expression under the traditional semantics for non-deterministic functions [19,13]:

[1] This work has been partially supported by the Spanish projects *MERIT-FORMS*-UCM (TIN2005-09207-C03-03), *DESAFIOS* (TIN2006-15660-C02-01), *PROMESAS-CAM* (S-0505/TIC/0407), and *FAST-STAMP* (TIN2008-06622-C03-01/TIN).

doi:10.1016/j.entcs.2010.08.039

- Under *call-time choice* parameter passing to compute a value for the term $f(c(0)\ ?\ c(1))$ we must first compute a (partial) value for $c(0)\ ?\ c(1)$, and then we may continue the computation with $f(c(0))$ or $f(c(1))$ which yield $d(0,0)$ or $d(1,1)$. Note that $d(0,1)$ and $d(1,0)$ are not correct values for $f(c(0)\ ?\ c(1))$ in that setting. Modern functional-logic languages like Toy [15] or Curry [12] adopt call-time choice.
  From the point of view of a denotational semantics, call-time choice parameter passing is equivalent to having a *singular semantics*, in which the substitutions used to instantiate the program rules for function application are such that the variables of the program rules range over single objects of the set of considered values.
- On the other hand, under *run-time choice* parameter passing, which corresponds to call-by-name, each argument is copied without any evaluation and so the different copies of any argument may evolve in different ways afterwards. However in $f(c(0)\ ?\ c(1))$ the evaluation of the subexpression $c(0)\ ?\ c(1)$ is needed in order to get an expression that matches the left hand side $f(c(X))$. Hence the derivations $f(c(0)\ ?\ c(1)) \to f(c(0)) \to d(0,0)$ and $f(c(0)\ ?\ c(1)) \to f(c(1)) \to d(1,1)$ are sound and compute the values $d(0,0)$ and $d(1,1)$, but neither $d(0,1)$ nor $d(1,0)$ are correct values for $f(c(0)\ ?\ c(1))$. Term rewriting is considered the standard semantics for run-time choice, and is the basis for the semantics of languages like Maude [5]. Traditionally it has been considered that run-time choice has its denotational counterpart on a *plural semantics*, in which the variables of the programs rules take their values over sets of objects of the set of considered values. But in this example we may consider the set $\{c(0), c(1)\}$ which is a subset of the set of values for $c(0)\ ?\ c(1)$ in which every element matches the argument pattern $c(X)$. Therefore, the set $\{0,1\}$ can be used for parameter passing obtaining a kind of "set expression" $d(\{0,1\},\{0,1\})$ that yields the values $d(0,0)$, $d(1,1)$, $d(0,1)$, and $d(1,0)$.

The conclusion is clear: *the traditional identification of run-time choice with a plural semantics is wrong when pattern matching is involved.* This fact was pointed out in [18] for the first time, where the $\pi CRWL$ logic was proposed as a novel formulation of a plural semantics with pattern matching. This logic shares with *CRWL* (the standard logic for call-time choice) some compositionality properties that make it more suitable than term rewriting (the standard formulation for run-time choice) for a value-based language like current implementations of FLP. For example, it is easy to see that for the previous program the expression $f(c(0\ ?\ 1))$ has more values than the expression $f(c(0)\ ?\ c(1))$ under run-time choice, even when the only difference between them is the subexpressions $c(0\ ?\ 1)$ and $c(0)\ ?\ c(1)$, which have the same values both in call-time choice, run-time choice, and plural semantics. This is pretty incompatible with a value-based semantic view, although it can be the right choice for other kind of rewriting based languages like Maude, not limited to CS's but able to handle general term rewrite systems (*TRS's*), and in which the goal is describing the evolution of a system instead of computing values.

Maude [5] is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Maude modules correspond to specifications in *rewriting logic* [16], a simple and expressive logic which allows the representation of many models of concurrent and distributed systems. This logic is an extension of equational logic; in particular, Maude *functional modules* correspond to specifications in *membership equational logic* [3], which, in addition to equations, allows the statement of *membership axioms* characterizing the elements of a sort. In this way, Maude makes possible the faithful specification of data types (like sorted lists or search trees) whose data are defined not only by means of constructors, but also by the satisfaction of additional properties. Rewriting logic extends membership equational logic by adding rewrite rules, that represent transitions in a concurrent system. Maude *system modules* are used to define specifications in this logic. Rewriting logic is also a good semantic framework for formally specifying programming languages as rewrite theories [5, Chap. 20][17]. Moreover, since those specifications usually can be executed in Maude, they in fact become interpreters for these languages.

Exploiting the fact that rewriting logic is reflective [6], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [5, Chap. 14], a feature that makes Maude remarkably extensible and that allows many advanced metaprogramming and metalanguage applications. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. In addition, the Maude system provides another module, `LOOP-MODE` [5, Chap. 17], which can be used to specify input/output interactions with the user. Thus, our program transformation, its execution, and its user interactions are implemented in Maude itself.

Although Maude provides commands to execute expressions in (metarepresented) modules, including a `metaSearch` function that performs a breadth-first search of the state space,[2] the highly non-deterministic nature of the programs obtained with the transformation avoids its use in practice. To solve this problem we have implemented the natural rewriting strategy [9], that evolves only the terms needed in the execution of an expression, avoiding to rewrite unnecessary terms. This is the first implementation of an on-demand strategy for Maude system modules,[3] and it can be considered a first stage towards on-demand execution of general rewrite theories.

This on-demand strategy has been combined with depth-first and breadth-first search, which allows to traverse the search tree in a flexible way, allowing to evaluate programs with potentially infinite branches. Furthermore, the tool also provides the option of searching with a bound in the number of rewrites, thus enhancing the performance of programs with large (possibly infinite) error branches.

The rest of the article is organized as follows: A session describing how to use the tool through examples is given in Section 2. Section 3 describes the main features

[2] The usual Maude strategy, consisting in rewrite terms with the first possible rule is not available here because it leads to results that are not necessarily cterms, i.e., terms made only of data constructors.

[3] On-demand strategies for Maude functional modules are described in [7].

Prog	::=	plural Name is [*Rl*] endp		Program
Rl	::=	$Lh \to Exp$.		Rule
Lh	::=	$f(p_1, \ldots, p_n)$	$f \in FS^n, p_i \in CTerm$	Lefthand side
Exp	::=	X	$X \in Var$	Expression
	|	$h(e_1, \ldots, e_n)$	$h \in FS^n \cup CS^n, e_i \in Exp$	

Fig. 1. Syntax

of the implementation. Finally, Section 4 outlines the main characteristics of the tool and gives some future work. We refer the reader to https://gpd.sip.ucm.es/trac/gpd/wiki/PluralSemantics/Maude for the source code, examples, and further information.

2 Examples

We illustrate in this section how to use the tool by means of examples. The session is started by executing the file `plural.bin` available in the web page above. Once the tool is running we can introduce modules, that must follow the syntax shown in Figure 1 and fulfill that the rules are left-linear and their righthand sides do not use variables not present in the corresponding lefthand sides.

First, we specify the clerks example shown in [18], where we have shops with some employees, and we want to find a pair of clerks:

```
Maude> (plural CLERKS is
  branches -> madrid .
  branches -> sevilla .
  employees(madrid) -> e(john, men, clerk) .
  employees(madrid) -> e(larry, men, boss) .
  employees(vigo) -> e(mary, women, clerk) .
  employees(vigo) -> e(james, men, boss) .
  twoclerks -> find(employees(branches)) .
  find(e(N,S,clerk)) -> p(N,N) .
endp)

Module introduced.
```

Under plural semantics, the expression `twoclerks` leads to any combination $\texttt{p(name}_1\texttt{, name}_2\texttt{)}$, where $\texttt{name}_i$ can be any clerk name (`john` and `mary` in the example), while run-time choice and call-time choice only lead to pairs where $\texttt{name}_1$ and $\texttt{name}_2$ coincide.

The tool reads the module and applies it the `pST` transformation, that simulates plural semantics with ordinary rewriting (see Section 3.1 for details). This transformed module can be seen with the command `(showTr .)`. We can now change the default depth-first strategy to breadth-first by using the command

```
Maude> (breadth-first .)

Breadth-first strategy selected.
```

We try now to compute the result of evaluating `twoclerks` by typing

```
Maude> (eval twoclerks .)

Result: p(john,john)
```

Since we try to find results with different names in the pair, we can ask for more answers with

```
Maude> (more .)

Result: p(john,mary)
```

Using the `more` command repeatedly we obtain all the different pairs reachable by the program until the following answer is prompted:

```
Maude> (more .)

No more results.
```

Now that we are familiar with the tool we show how to execute a more complex problem. The fearless Ulysses has been captured in his travel from Troy to Ithaca, but he knows he can persuade one of his four guardians to interchange the key for some items, that Ulysses has to obtain from the other guardians with his initial possessions:

```
(plural LAIR is
  guardians -> circe ? calypso ? aeolus ? polyphemus .
  ask(circe, trojan-gold) -> item(treasure-map) ? sirens-secret .
  ask(calypso, sirens-secret) -> item(chest-code) .
  ask(aeolus, item(M)) -> combine(M,M) .
  ask(polyphemus, combine(treasure-map, chest-code)) -> key .
```

Notice that the information given to the fourth guardian can be only obtained with our semantics, because a pair of the same variable becomes a pair of different constants. To acquire these items he uses the function `discover`, that uses the current information or tries to ask the guardians for more.

```
  discover(M) -> M ? discover(discStep(M) ? M) .
  discStep(M) -> ask(guardians, M) .
```

Finally, Ulysses escapes if he obtains the key from his initial belongings: an immeasurable amount of `trojan-gold`.

```
  escape -> open(discover(trojan-gold)) .
  open(key) -> true .
endp)
```

We use the depth-first strategy to check if the evasion is possible:

```
Maude> (depth-first .)

Depth-first strategy selected.
```

We evaluate now the term `escape` with `80` as upper bound in the number of rewrites with the command:

```
Maude> (eval [depth= 80] escape .)

Result: true
```

That is, there is a way to interchange the information in order to escape.

3 Implementation

We describe in this section the main topics of the implementation. First, we describe how the program transformation described in [18] has been accomplished. Then, we describe how to improve its execution by using the natural rewriting strategy [9].

3.1 Program transformation

The first component of our implementation is a source-to-source transformation of CS, whose adecuacy has been proved in [18]. The main idea in this transformation is postponing the pattern matching to avoid it to force an early resolution of non-determinism. To illustrate this concept, let us see the result of applying the transformation over the program $\mathcal{P}$ of Section 1:

$$\begin{aligned}\hat{\mathcal{P}} = \{\ & f(Y) \to \mathit{if}\ \mathit{match}(Y)\ \mathit{then}\ d(\mathit{project}(Y), \mathit{project}(Y)),\\ & \mathit{match}(c(X)) \to \mathit{true}, \mathit{project}(c(X)) \to X,\\ & \mathit{if}\ \mathit{true}\ \mathit{then}\ X \to X, X\ ?\ Y \to X, X\ ?\ Y \to Y\}\end{aligned}$$

Now, to evaluate the expression $f(c(0)\ ?\ c(1))$, we are not forced anymore to solve the non-deterministic choice between $c(0)$ and $c(1)$, because any expression matches the variable pattern Y, therefore the step

$$f(c(0)\ ?\ c(1)) \to \mathit{if}\ \mathit{match}(c(0)\ ?\ c(1))\ \mathit{then}\ d(\mathit{project}(c(0)\ ?\ c(1)), \mathit{project}(c(0)\ ?\ c(1)))$$

is sound. Note that the guard $\mathit{if}\ \mathit{match}(c(0)\ ?\ c(1))$ is needed to ensure that at least one of the values of the argument matches the original pattern, otherwise the soundness of the step could not be granted. Later on, after resolving the guard, different evaluation of the occurrences of $\mathit{project}(c(0)\ ?\ c(1))$ will lead us to the final values $d(0,0)$, $d(1,1)$, $d(0,1)$, and $d(1,0)$, which are the expected values for the expression in the original program under the plural semantics.

Program transformations like this can be easily handled in Maude thanks to its efficient use of reflection [5], that allows to manipulate (metarepresented) Maude modules (and more concretely Maude rules) as data. In [18] the transformation above is defined through the function pST shown in Figure 2, which for any program rule returns a rule to replace it, and a small set of auxiliary *match* and *project* rules

Given a *CRWL*-program P, for every rule $(f(p_1,\ldots,p_n) \to r) \in \mathcal{P}$ its transformation is defined as:

$$pST(f(p_1,\ldots,p_n) \to r)$$

$$= \begin{cases} f(p_1,\ldots,p_n) \to r & \text{if } \rho_1 \ldots \rho_m \text{ is empty} \\ f(\tau(p_1),\ldots,\tau(p_n)) \to \textit{ if } match(Y_1,\ldots,Y_m) & \text{otherwise} \\ \qquad\qquad\qquad\qquad \textit{then } r[\overline{X_{ij}/project_{ij}(Y_i)}] & \end{cases}$$

where $\rho_1 \ldots \rho_m = p_1 \ldots p_n \mid \lambda p.(p \notin \mathcal{V} \wedge var(p) \neq \emptyset)$.
- $\forall \rho_i$, $\{X_{i1},\ldots,X_{ik_i}\} = var(\rho_i) \cap var(r)$ and $Y_i \in \mathcal{V}$ is fresh.
- $\tau : CTerm \to CTerm$ is defined by $\tau(p) = p$ if $p \in \mathcal{V} \vee var(p) = \emptyset$ and $\tau(p) = Y_i$ otherwise, for $p \equiv \rho_i$.
- $match \in FS^m$ fresh is defined by the rule $match(\rho_1,\ldots,\rho_m) \to true$.
- Each $project_{ij} \in FS^1$ is a fresh symbol defined by the single rule $projectij(\rho_i) \to X_{ij}$.

Fig. 2. pST transformation

for the replacement. Using the reflection features of Maude, we can implement it with an operator `pST`, that receives the rule that must be transformed and an index to create fresh function names related to this rule and returns a set of rules composed by the new rule and the associated *match* and *project* rules. If the list of ρ_i is empty, the rule is not transformated.

```
op pST : Rule Nat -> RuleSet .
ceq pST(rl T => T' [AtS] ., N) = rl T => T' [AtS] .
 if computeRhos(T) == empty .
```

If the list of ρ_i is not empty, then we must transform the rule. The `match` expression used in the *if* condition is computed with the function `createMatchExp`, that receives as argument the length of the list of ρ_i to create the same number of fresh variables. The rule that will be applied when this condition holds is defined with the operator `createMatchRule`. The operator `computeSubstitutions` calculates the project function that substitutes each ρ_i on the righthand side, and keeps the result in the table `ProjectTable`. This table is then used to make the substitution and obtain the result of the if statement. The rules associated with each projection are obtained by means of `createProjectRules`. Finally, the application of the τ function to the arguments on the lefthand side of the rule is made with `applyTau`.

```
ceq pST(rl T => T' [AtS] ., N) =
             rl applyTau(T) => 'if_then_[MatchExp, NewRHS] [AtS] .
             MatchRule ProjectRules
 if Rhos := computeRhos(T) /\ Rhos =/= empty /\
```

```
VarsRHS := getVars(T') /\
MatchExp := createMatchExp(size(Rhos), N) /\
MatchRule := createMatchRule(Rhos, N) /\
ProjectTable := computeSubstitutions(Rhos, VarsRHS, N) /\
NewRHS := substitute(T', ProjectTable) /\
ProjectRules := createProjectRules(Rhos, VarsRHS, N) .
```

3.2 Natural rewriting

The second component of our system is an implementation of the natural rewriting on-demand strategy [10], which became necessary to deal with the highly non-deterministic programs obtained after the transformation. This transformation is implemented by the operator `natNext` that computes the set of reachable terms by evolving the needed positions with all the possible rules.

As it is usual in other on-demand strategies, a data structure called *definitional tree* is used to encode the demand information associated to the program rules. What makes natural rewriting different and more appropriate than other on-demand strategies is that it uses a special kind of definitional tree called *matching definitional tree* [1], that allows us to keep the pattern matching process separated from the evaluation through demanded positions. In previous strategies the encoding of these two processes were interleaved in the definitional trees, and as a consequence they lost opportunities to prune the search space. A matching definitional tree is built for each function present in the program, after a static analysis performed during the compilation. We implement this by the operator `MDTMap`, which takes the Maude representation of the transformed program and returns a map from function symbols to its corresponding definitional trees.

Once the matching definitional trees have been computed, we can use the function `mt` [9] to compute the needed positions and the rules that must be applied. Moreover, we combine this evaluation strategy with (bounded) depth-first and breadth-first search, keeping *all* the possible terms obtained from `mt` and its depth in a list that works as a stack for the depth-first strategy and as a queue for the breadth-first strategy.

Our implementation of natural rewriting is a contribution by itself, because it can be used to perform on-demand evaluation of any CS specified in Maude, not only of those used to simulate $\pi CRWL$. Nevertheless it has several limitations. First of all, although there are extensions of this strategy to deal with general TRS's [10], we have only implemented the version presented in [9], which is only able to deal with left-linear CS's. Besides, the strategy is formulated for pure CS's, and does not consider the possibility of combining the rewriting rules with eager and confluent equations, which is one of the most distinctive and useful features of Maude. Anyway the current implementation is powerful enough to help us to improve the efficiency of our simulation of $\pi CRWL$, as well as dealing with "pure" CS's specified in Maude, and we contemplate the aforementioned extensions as interesting subjects of future work.

4 Conclusions

In this work we have described a prototype implementation of the $\pi CRWL$ logic in Maude, based on a source-to-source transformation of CS. This transformation, as well as the execution of the resulting program with the natural rewriting on-demand strategy, is implemented in Maude, taking advantage of its efficient implementation of reflection, that allows to use specifications as data.

Our implementation is natural in two different ways. First of all, the natural rewriting strategy has been used to deal with the wild non-determinism explosion arising in Maude system modules. While this non-determinism is necessary to completely model check a system, because then we want to test if a given property holds in every state reachable during the evolution of the system, it is not the case when we are just computing the set of values for a given expression. In $\pi CRWL$ we only care about the final reachable values, hence we may use an on-demand strategy like natural rewriting to prune the search space without losing interesting values.

On the other hand, both the transformation and the strategy upon which our implementation is based have been formulated at the abstraction level of source programs, that is, using the syntax and theory of TRS's, which are the foundations of Maude too. Therefore Maude was the natural choice for an object language, and as a consequence there is a small distance between the specification of the transformation and the strategy, and the code which implements them. Moreover, since the adecuacy of the transformation and the completeness and optimality of the strategy have been proved, we can affirm that the resulting implementation is correct by construction. We think that this is one of the strengths of our prototype.

Another important contribution is our implementation of the natural rewriting strategy, which has been implemented in Maude for the first time. The corresponding `natNext` operator can be used for performing on-demand evaluation of any CS specified in a Maude system module, and it is especially relevant because is the first on-demand strategy for this kind of modules, complementing the default rewrite and breadth-first search Maude commands.

As a subject of future work we contemplate the extension of natural rewriting to TRS's which are not necessarily CS's, following the theory developed in [10]. Another orthogonal extension of the strategy could go in the direction of combining non-deterministic rewriting rules evaluated on demand (from Maude system modules) with deterministic and terminating equations evaluated eagerly (from Maude functional modules). This is particularly interesting when there are fragments of a TRS which constitute a confluent and terminating TRS (like the `match` and `project` functions introduced by our transformation), that could be executed dramatically more efficiently by treating them as Maude equations.

Other extensions go in the line of adding features to the $\pi CRWL$ interpreter. Adding higher order capabilities by means of the classic transformation of [20] would be interesting and it is standard in the field of FLP. More novel would be using the matching-modulo capacities of Maude to enhance the expressivity of plural semantics, after a corresponding revision of the theory of $\pi CRWL$. Another interesting

extension could come from the combination of the plural and singular semantics, using the framework for combining run-time choice with call-time choice developed in [14]. Besides, some aditional research must be done to improve the performance of the interpreter by means of some kind of sharing in the line of [4].

Last but not least, some additional effort must be invested in producing a larger collection of program examples and programming patterns that exploit the capabilities of this new semantics. In fact the development of this prototype is a step in that direction, as it allows us to do empirical experimentation with $\pi CRWL$.

Acknowledgement

The authors would like to thank Santiago Escobar for his very valuable comments and suggestions and Francisco Javier López-Fraguas and Alberto Verdejo for their useful observations of earlier versions of this paper.

References

[1] S. Antoy. Definitional trees. In H. Kirchner, G. Levi, editors, *Proceedings of the 3rd International Conference on Algebraic and Logic Programming (ALP'92)*, *LNCS* 632, pages 143–157. Springer, 1992.

[2] S. Antoy and M. Hanus. Functional logic design patterns. In Z. Hu and M. Rodríguez-Artalejo, editors, *Proceedings of the 5th International Symposium on Functional and Logic Programming (FLOPS'2002)*, *LNCS* 2441, pages 67–87. Springer, 2002.

[3] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[4] B. Braßel and F. Huch. On a tighter integration of functional and logic programming. In Z. Shao, editor, *Proceedings of the 5th ASIAN Symposium on Programming Languages and Systems (APLAS 2007)*, *LNCS* 4807, pages 122–138. Springer, 2007.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, *LNCS* 4350. Springer, 2007.

[6] M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2):70–91, 2007.

[7] F. Durán, S. Escobar, and S. Lucas. On-demand evaluation for Maude. In *Proc. of 5th International Workshop on Rule-Based Programming, RULE'04*, ENTCS 124(1), pages 263–284. Elsevier, 2005.

[8] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. In M. Archer, T. B. de la Tour, and C. A. Muñoz, editors, *Proceedings of the 6th International Workshop on Strategies in Automated Deduction (STRATEGIES 2006)*, *ENTCS* 174(11), pages 3–25. Elsevier, 2007.

[9] S. Escobar. Implementing natural rewriting and narrowing efficiently. In Y. Kameyama and P. Stuckey, editors, *Proceedings of the 7th International Symposium on Functional and Logic Programming (FLOPS'2004)*, *LNCS* 2998, pages 147–162. Springer, 2004.

[10] S. Escobar, J. Meseguer, and P. Thati. Natural rewriting for general term rewriting systems. In S. Etalle, editor, *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, *LNCS* 3573, pages 101–116. Springer, 2004.

[11] J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.

[12] M. Hanus. Functional logic programming: From theory to Curry. Technical report, Christian-Albrechts-Universität Kiel, 2005.

[13] H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.

[14] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A flexible framework for programming with non-deterministic functions. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, 2009. To appear.

[15] F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, *LNCS* 1631, pages 244–247. Springer, 1999.

[16] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[17] J. Meseguer and G. Roşu. The rewriting logic semantics project. In P. Mosses and I. Ulidowski, editors, *Proceedings of the Second Workshop on Structural Operational Semantics (SOS 2005), Lisbon, Portugal, 10 July 2005*, *ENTCS* 156(1), pages 27–56. Elsevier, 2006.

[18] J. Rodríguez-Hortalá. A hierarchy of semantics for non-deterministic term rewriting systems. In *Proc. Foundations of Software Technology and Theoretical Computer Science*, 2008.

[19] H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.

[20] D. H. Warren. Higher-order extensions to prolog: are they needed? In J. Hayes, D. Michie, and Y.-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd., 1982.