A Lightweight Combination of Semantics for Non-deterministic Functions ⋆

F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, jrodrigu@fdi.ucm.es, jaime@sip.ucm.es

Abstract. The use of non-deterministic functions is a distinctive feature of modern functional logic languages. The semantics commonly adopted is *call-time choice*, a notion that at the operational level is related to the *sharing* mechanism of lazy evaluation in functional languages. However, there are situations where *run-time choice*, closer to ordinary rewriting, is more appropriate. In this paper we propose an extension of existing call-time choice based languages, to provide support for run-time choice in localized parts of a program. The extension is remarkably simple at three relevant levels: syntax, formal operational calculi and implementation, which is based on the system *Toy*.

1 Introduction

Non-strict non-deterministic functions are a distinctive feature of modern functional logic languages (see [5] for a recent survey). It is known that the introduction of non-determinism in a functional setting gives rise to a variety of semantic decisions (see e.g. [12]). For term-rewriting based specifications, Hussmann [7] established a major distinction between *call-time choice* and *run-time choice*. Call-time choice is closely related to call-by-value and, in the case of strict semantics, it is easily implemented by innermost rewriting. In the case of non-strict semantics, things are more complicated, since the call-by-value view of call-time choice must include partial values. Operationally, this needs something similar to the sharing mechanism followed, for efficiency reasons, in (deterministic) functional languages under lazy evaluation. In contrast, run-time choice does not share, corresponds rather to call-by-name, and is realized by ordinary rewriting. For deterministic programs, run-time and call-time are able to produce the same set of values, but in general the set of values reachable by run-time choice is larger than that of call-time choice.

Non-deterministic functions with non-strict and call-time choice semantics were introduced in the functional logic setting with the *CRWL* framework [4],

⋆ This work has been partially supported by the Spanish projects MERIT-FORMS-UCM (TIN2005-09207-C03-03), FAST-STAMP (TIN2008-06622-C03-01/TIN) and Promesas-CAM (S-0505/TIC/0407) .

in which programs are possibly non-confluent and non-terminating constructor-based term rewriting systems (*CTRS*). Since then, they are common part of daily programming in systems like *Curry* [6] or *Toy* [11]. Run-time choice has been rarely [1] considered as a valuable global alternative to call-time choice.

However, there might be parts in a program or individual functions for which run-time choice could be a better option, and therefore it would be convenient to have both possibilities (run-time/call-time) at programmer's disposal. The following example illustrates the interest of combining both semantics.

Example 1. Modeling grammar rules for string generation can be directly done by CTRS like the following (non-confluent and non-terminating) one, in which we assume that texts (terminals) are represented as strings (lists of characters), that can be concatenated with $++$ (defined in a standard way):

$$letter \to \text{"}a\text{"} \;\; .... \;\; letter \to \text{"}z\text{"} \qquad word \to \text{" "} \qquad word \to letter{+}{+}word$$

Disregarding syntax, this CTRS is a valid program in functional logic systems like *Curry* or *Toy*. The program acts as a non-deterministic generator of the texts in the language defined by the grammar. Each individual reduction leads to a string in the language.

The generation of palindromes (of even length, for simplicity) could be done by the rewrite rules:

$$palindrome \to palAux(word) \qquad palAux(X) \to X \;{+}{+}\; reverse(X)$$

where *reverse* is defined in any standard way. It is important to remark that the definition of *palindrome/palAux* works fine only if call-time choice is adopted for non-determinism, meaning operationally that in the (partial) reduction

$$palindrome \to palAux(word) \to word \;{+}{+}\; reverse(word)$$

the two occurrences of *word* created by the rule of *palAux* must be shared. If run-time choice (i.e., ordinary rewriting) were used, the two occurrences of *word* could follow independent ways, and therefore *palindrome* could be reduced, for instance, to *"oops"*, which is not a palindrome. Two useful operators to structure grammar specifications are the alternative '|' and Kleene's '$*$' for repetitions:

$$X \mid Y \to X \qquad X \mid Y \to Y \qquad star\ X \to \text{" "} \qquad star\ X \to X{+}{+}star(X)$$

With them *letter* and *word* could be redefined as follows:

$$letter \to \text{"}a\text{"} \mid \text{"}b\text{"} \mid ... \mid \text{"}z\text{"} \qquad word \to star(letter)$$

The annoying fact is that this does not work! At least not under call-time choice, which implies that this is an uncorrect definition of *star* in systems like *Curry* or *Toy*. The problem with call-time choice here is that all the occurrences of *letter* created by *star* will be shared and therefore *word* will only generate words like *aaa*, *nnnn*, ..., made with repetitions of the same letter. To overcome this problem, we would like that in the definition of *word*, the application of the *star* operation to the string generator *letter* could follow a run-time choice regime, so that each of the two occurrences of *letter* created in the rewriting steps

$$\mathit{word} \to \mathit{star(letter)} \to \mathit{letter} \mathbin{++} \mathit{star(letter)}$$

could evolve independently. In our proposed extension this would be expressed by writing the definition of *word* as follows:

$$\mathit{word} \to \mathit{star(rt(letter))}$$

where *rt* is a special unary function symbol indicating that its argument (*letter* in this case) is not going to be shared in the evaluation of the surrounding application (*star(rt(letter))* in this case).

We remark that in this example neither call-time nor run-time choice are a good single option as semantics for the whole program. The definition of *palindrome* requires call-time choice, while the use of *star* in *word* requires run-time choice. To the best of our knowledge, no existing implementation for functional logic programming offers the possibility of combining in the same program both kind of semantics. This paper addresses that problem at a practical level, aiming at a solution that can be easily realized by modifying existing Prolog-based functional logic systems. Although our main interest is easiness of implementation, we provide also formal calculi attempting to reflect at an abstract level the operational behavior of the extended language. These calculi could be the technical basis for a thorough investigation of the formal properties of our proposal, a matter that is out of the scope of this paper.

2 A tiny functional logic language with run-time choice annotations

We shortly present here a functional logic language with run-time choice annotations. To keep the presentation simple, we consider only a first order untyped core with the usual first order syntax of term rewriting systems. However, the implementation described in Sect. 5 extends the existing system *Toy*, which is a HO typed system using curried notation.

We consider a signature Σ made of constructor symbols $c, d, \ldots \in CS$, function symbols $f, g, \ldots \in FS$, the special unary symbol rt, and a set of variables $X, Y, \ldots \in \mathcal{V}$. We sometimes write $c \in CS^n$ ($f \in FS^n$) to denote a constructor (function) symbol of arity n. Constructor terms (or c-terms) $t, s, \ldots \in CTerm$ follow the syntax: $t ::= X \mid c(t_1, \ldots, t_n)$, and expressions (with run-time choice annotations) $e, \ldots \in \mathit{RtExpr}$ follow the syntax: $e ::= X \mid c(e_1, \ldots, e_n) \mid f(e_1, \ldots, e_n) \mid rt(e)$. An intermediate set between $CTerm$ and RtExpr is the set $\mathit{RtCTerm}$ of annotated c-terms $\mathit{RtCTerm} \ni t ::= X \mid c(t_1, \ldots, t_n) \mid rt(e)$, where $t_1, \ldots, t_n$ are also from $\mathit{RtCTerm}$ and e is any expression.

A *program* is a set of function defining rules, each of the form

$$f(t_1, \ldots, t_n) \to e$$

where $(t_1, \dots, t_n)$ is a linear tuple of c-terms from $CTerm$, and e is any expression from $RtExpr$. We remark that annotated c-terms play no special role in the syntax of programs, but play an important role in the parameter passing mechanism, which informally can be explained as follows: to apply a program rule $f(t_1, \dots, t_n) \to e$ to a function application $f(e_1, \dots, e_n)$, a matching substitution θ such that $f(t_1, \dots, t_n)\theta \equiv f(e_1, \dots, e_n)$ must exist, and then $f(e_1, \dots, e_n)$ reduces to $r\theta$, but following the informal criterion about *sharing*: the copies of subexpressions e of $f(e_1, \dots, e_n)$ created in $r\theta$ are not shared –i.e. follow run-time choice– if e is under a rt annotation, and shared –i.e. follow call-time choice– otherwise. These ideas are formalized in the next section in the form of two alternative operational calculi.

3 Formal operational calculi

In this section we will try to design some calculi able to express an extension of the standard call-time choice semantics for FLP [4], to support the primitive rt for run-time choice evaluation. Our approach to formalize this extension is based in two main ideas:

- The new calculus will be a modification of the simple rewrite calculus presented in [9]. As we will have to express run-time evaluation for parts of the computation, we will need to have partially evaluated expressions at our disposal. A calculus in the line of those used in [4] would not be a suitable tool, as it returns only partial values for the expressions, but no intermediate states of the computation.
- Instead of giving a semantics for annotations $rt(e)$ directly, we will think about it as a syntactic sugar for the annotation of the function symbols that appear in e with a rt superscript, indicating that those function symbols will be treated as a constructor symbol as far sharing and parameter passing is concerned. Therefore, an expression containing only variables, constructor symbols and function symbols annotated with rt could be copied freely, thus getting a run-time behaviour for it, as a function argument. We write FS^{rt} for the set of function symbols with superscript rt, $FS^?$ for $FS \cup FS^{rt}$ and $f^?$ for function symbols in $FS^?$, i.e., for possibly superscripted function symbols.

The desugaring of expressions to eliminate the rt primitive transforming it into rt annotations is performed according to the following definition:

Definition 1 (Desugaring of the rt primitive).

$$
\begin{array}{lll}
desugar(rt(X)) & = X & \textit{if } X \in \mathcal{V} \\
desugar(rt(c(e_1, \dots, e_n))) & = c(desugar(rt(e_1)), \dots, desugar(rt(e_n))) & \textit{if } c \in CS \\
desugar(rt(f(e_1, \dots, e_n))) & = f^{rt}(desugar(rt(e_1)), \dots, desugar(rt(e_n))) & \textit{if } f \in FS \\
desugar(rt(rt(e))) & = desugar(rt(e)) &
\end{array}
$$

According to this syntactic desugaring for $rt(e)$, the syntax of annotated c-terms and expressions can be reformulated as follows:

- $RtCTerm \ni t ::= X \mid c(t_1, \ldots, t_n) \mid f^{rt}(t_1, \ldots, t_n)$, if $X \in \mathcal{V}$, $c \in CS^n$, $f \in FS^n$, $t_1, \ldots, t_n \in RtCTerm$
- $RtExpr \ni e ::= X \mid c(e_1, \ldots, e_n) \mid f^?(e_1, \ldots, e_n)$, if $X \in \mathcal{V}$, $c \in CS^n$, $f^? \in FS^?$, $e_1, \ldots, e_n \in RtExpr$

To express parameter passing in function applications with $rt-$annotated arguments we will need to consider rt-c-substitutions, defined by: $\theta \in RtCSubst$ iff $X\theta \in RtCTerm, \forall X \in \mathcal{V}$.

Now we will define calculi to work with annotated expressions. In [9] two rewrite notions for call-time choice were defined, each of them being interesting for different applications. Here we will modify both of them to get two (hopefully) equivalent characterizations of a semantics for annotated run-time choice under a call-time choice environment.

(B)	$\mathcal{C}[e] \rightarrowtail \mathcal{C}[\perp]$	for any context $\mathcal{C}$ and expression $e \in RtExpr_\perp$
(OR)	$\mathcal{C}[f^?(\overline{p})\theta] \rightarrowtail \mathcal{C}[r\theta]$	for any context $\mathcal{C}$, $(f(\overline{p}) \to r \in)\mathcal{P}$, and $\theta \in RtCSubst_\perp$

Fig. 1. A one-step reduction relation for non-strict call-time choice with rt annotations

The first characterization is shown in Fig. 1. Its drawback is that the rule **(B)** involves a 'magical' guessing in advance of the fact that the reduction of a (sub)-expression is not going to be needed, and replaces this 'no need of reduction in the future' by an artificial anticipated reduction to the undefined value $\perp$. However, because of its simplicity, the relation is helpful to understand what are the possible results of a reduction.

The second characterization is the rewrite relation of Fig. 2. It expresses in a more realistic manner (specially, if a reduction strategy would be added, which is not our focus here) the way in which computations are to be performed. To express sharing (when needed), local bindings are created via a *let* construct.

(Fapp) $f^?(\overline{p})\theta \;\to_l\; r\theta$, if $(f(\overline{p}) \to r) \in \mathcal{P}$, $\theta \in RtCSubst$

(LetIn) $h(\ldots, e, \ldots) \to_l let\ X = e\ in\ h(\ldots, X, \ldots)$, if $h \in \Sigma$, $e \equiv f(\overline{e'})$ with $f \in FS$ or $e \equiv let\ Y = e'\ in\ e''$, and X is a fresh variable

(Bind) $let\ X = t\ in\ e \;\to_l\; e[X/t]$, if $t \in RtCTerm$

(Elim) $let\ X = e_1\ in\ e_2 \to_l e_2$, if $X \notin FV(e_2)$

(Flat) $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \;\to_l\; let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$ if $Y \notin FV(e_3)$

(Contx) $\mathcal{C}[e] \to_l \mathcal{C}[e']$, if $\mathcal{C} \neq [\,]$, $e \to_l e'$ using any of the previous rules, and in case $e \to_l e'$ is a (Fapp) step using $(f(\overline{p}) \to r)\theta \in [\mathcal{P}]$ then $vRan(\theta|_{\setminus var(\overline{p})}) \cap BV(\mathcal{C}) = \emptyset$.

Fig. 2. Rules of *let*-rewriting extended with rt annotations

Note how in the rule (LetIn), in the case a function application is extracted to a *let*, it is needed that f is not marked with rt, which tell us that it is not allowed to duplicate it, and therefore it may be needed to put it in a *let* in order to progress with the evaluation (for example if it appears in an argument of another function application whose reduction is needed).

Example 1. Given the program

$$\begin{array}{ll} coin \to 0 & f(X) \to g(X, coin) \\ coin \to 1 & g(X,Y) \to (X,X,Y,Y) \end{array}$$

we want to evaluate the expression $rt(f(coin))$, which is desugared as $f^{rt}(coin^{rt})$. With the calculus of Fig. 1 we can do:

$$\begin{array}{l} f^{rt}(coin^{rt}) \rightarrowtail g(coin^{rt}, coin) \rightarrowtail g(coin^{rt}, 0) \rightarrowtail (coin^{rt}, coin^{rt}, 0, 0) \\ \rightarrowtail (0, coin^{rt}, 0, 0) \rightarrowtail (0,1,0,0) \end{array}$$

Note how in the first step the expression $f^{rt}(coin^{rt})$ can be evaluated as every function symbol present in $coin^{rt}$ is annotated with rt. On the other hand we cannot apply (OR) to $g(coin^{rt}, coin)$, as one of its arguments contains a function symbol that it is not annoted for run-time, and thus the value $(0,1,0,1)$ is not reachable from $f^{rt}(coin^{rt})$. This is even more evident in the version of this evaluation got with the calculus of Fig. 2:

$$\begin{array}{l} f^{rt}(coin^{rt}) \to_l g(coin^{rt}, coin) \to_l let\ X = coin\ in\ g(coin^{rt}, X) \\ \to_l let\ X = coin\ in\ (coin^{rt}, coin^{rt}, X, X) \to_l let\ X = coin\ in\ (0, coin^{rt}, X, X) \\ \to_l let\ X = coin\ in\ (0,1,X,X) \to_l let\ X = 0\ in\ (0,1,X,X) \\ \to_l (0,1,0,0) \end{array}$$

When we reach the expression $let\ X = coin\ in\ (coin^{rt}, coin^{rt}, X, X)$ it is clear that the first two components of the tuple may evolve in different ways while the values of the last two components will be shared.

4 A variant of run-time annotations

In the present section we will show another primitive to express run-time choice that we will build on top of the previous primitive rt, through a simple program transformation. We will call that primitive rRt, and define its behaviour by the following inference rule that should be added to the *CRWL* logic [4]:

$$\frac{e \to^*_{\mathcal{P}'} e' \quad t \sqsubseteq |e'|}{\mathcal{P} \vdash_{CRWL} rRt(e) \twoheadrightarrow t} \quad \textbf{(rRt)}$$

where $\mathcal{P}'$ is the program resulting of adding to $\mathcal{P}$ the new rule $rRt(e) \to e$, and $e \to^*_{\mathcal{P}'} e'$ indicates that e' can be reached from e by zero or more steps of ordinary rewriting [2] using the program $\mathcal{P}'$. The approximation ordering $t \sqsubseteq t'$ between partial values expresses that t is less defined than t' (see [4] for details).

The rule (**rRt**) itself is already suggesting a possible implementation for rRt. This implementation will be based on the fact that, for any program in which every function symbol that appears in a right hand side of a program rule is *rt*-annotated, the evaluation of an expression that has each of its function symbols *rt*-annotated too returns the same results as it was evaluated under run-time choice but discarding the annotations. This ideas are formalized in the following definition:

Definition 2. *Given a CRWL-program $\mathcal{P}$:*

- *We build the signature of a new program $\_\mathcal{P}$ adding to it any constructor symbol in the signature of $\mathcal{P}$, and for any function symbol f in the signature of $\mathcal{P}$ considering a fresh function symbol $\_f$ which we add to the signature of $\_\mathcal{P}$.*
- *We define the transformation of expressions rRt as:*

$$\begin{array}{lll} rRtT(X) & = X & \text{if } X \in \mathcal{V} \\ rRtT(c(e_1, \ldots, e_n)) & = c(rRtT(e_1), \ldots, rRtT(e_n)) & \text{if } c \in CS \\ rRtT(f(e_1, \ldots, e_n)) & = \_f^{rt}(rRtT(e_1), \ldots, rRtT(e_n)) & \text{if } f \in FS \\ rRtT(rRtT(e)) & = rRtT(e) & \end{array}$$

- *For any $(f(p_1, \ldots, p_n) \rightarrow r) \in \mathcal{P}$ we add the rule $\_f(p_1, \ldots, p_n) \rightarrow rRtT(r)$ to $\_\mathcal{P}$.*

Finally, any expression $rRt(e)$ to be evaluated under $\mathcal{P}$ is desugared into $rRtT(e)$ and evaluated under $\mathcal{P} \uplus \_\mathcal{P}$

Example 2. Starting with the program of Example 1 we get the program

$$\{coin \rightarrow 0, coin \rightarrow 1, f(X) \rightarrow g(X, coin), g(X, Y) \rightarrow (X, X, Y, Y)\}$$
$$\uplus$$
$$\{\_coin \rightarrow 0, \_coin \rightarrow 1, \_f(X) \rightarrow \_g^{rt}(X, coin), \_g(X, Y) \rightarrow (X, X, Y, Y)\}$$

under which we can do:

$$\begin{array}{l} rRt(f(coin)) \equiv \_f^{rt}(\_coin^{rt}) \rightarrowtail \_g^{rt}(\_coin^{rt}, \_coin^{rt}) \\ \rightarrowtail (\_coin^{rt}, \_coin^{rt}, \_coin^{rt}, \_coin^{rt}) \rightarrowtail^* (0, 1, 0, 1) \end{array}$$

5 Implementation issues

In order to study the practicability of the proposal we have implemented it as an extension of the functional logic system *Toy* ([3]). This system, as well as other modern systems like *Curry* ([6]), operates under call-time choice. We introduce the new syntactic construct *rt e* into the syntax of *Toy* to instruct the system to evaluate the expression e under a run-time choice regime. The system will use run-time choice for evaluating the expressions annotated with *rt*, and call-time choice as usual for the rest of computations, i.e., we have within the same language both regimes of evaluation.

The extension is well supported by the system and requires only some lightweight modifications. In fact, the traditional problem is how to achieve sharing in a non-deterministic language like this, and our goal now is to inhibit this sharing mechanism at the points required by the programmer with *rt*.

Toy is implemented in Prolog and uses Prolog as target code (see [8, 3] for details). Sharing is implemented by means of *suspensions*, that are Prolog terms of the form:

susp(FunctionName,Arguments,Result,Evaluated)

The *FunctionName* and its *Arguments* represent the expression e to be evaluated, while *Result* is the resulting value (if evaluated, variable in other case) and *Evaluated* is a flag that indicates if the expression has been evaluated (flag *on*) or not (flag variable). Every function call is translated into a suspension in order to share its value when the expression is passed as argument and copied. As an example of the use of this representation consider the following program:

```
coin = 0
coin = 1

double X = X + X

test1 = double coin
test2 = rt (double coin)
```

Consider the evaluation of *test1*. As all the function calls are translated into suspended forms, in particular *coin* will have the form *susp(coin,[],R,E)*. The evaluation of *double* does not demand the evaluation of its argument *coin*, so it will produce

susp(coin,[],R,E) + susp(coin,[],R,E)

Later, when one of the calls to *coin* is evaluated, for example to 0, the other one automatically gets the same value:

susp(coin,[],0,on) + susp(coin,[],0,on)

The result of the addition is 0, that is a value obtained for *test1*. If we evaluate *coin* to 1 we have

susp(coin,[],1,on) + susp(coin,[],1,on)

and then result 2, that is the other value obtained for *test1*. With this sharing mechanism we can not obtain the value 1 for *double coin* as it would require to evaluate both calls to *coin* to two different values.

For the function function *test2* we would want to obtain the values 0 and 2 as before, but also the value 1 (evaluating separately both calls to *coin*). In this case *rt* will deactivate the sharing mechanism. This can be easily achieved by translating the call *coin* into the suspended form *susp(coin,[],R,rt)*. The flag *rt* will indicate to the system that the value of this expression must not be shared (and neither kept in the variable *R*). For *test2* we evaluate *susp(coin,[],R,rt) + susp(coin,[],R,rt)*

The first suspension can be reduced to 0 (without annotating the result in R), and the second one to 1, obtaining 1 for *test2* as expected.

The extension implemented in *Toy* provides this behaviour with *test1* and *test2*. In fact, for *test2* it obtains 0, 2 and 1 twice (evaluating the first *coin* to 0 and the second to 1 and viceversa). As another example, consider the problem of generating numbers as combinations of the digits 0, 1 and 2. Using *take*, *repeat* and the alternative operator '|' (introduced in Sec. 1) we could define:

```
number N = take N (repeat (0 | 1 | 2))
```

but then the expression *number 3* will produce only the answers [0,0,0], [1,1,1] and [2,2,2], because the expression *0 | 1 | 2* is evaluated only once and then its value is shared when evaluating *repeat*. For achieving the expected behaviour we have to instruct the system for choosing the digits under run-time choice (to avoid sharing):

```
number N = take N (repeat (rt (0 | 1 | 2)))
```

Now we obtain the 27 possible combinations that include $[1, 1, 2]$ or $[3, 1, 2]$ as instance. The example of palindromes of Sect. 1 also works as expected.

The prototype and some examples can be found at
`https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems`.

6 Conclusions

We have proposed a simple way of combining in the same program run-time choice and call-time choice, two semantics commonly adopted for non-determinism in rewriting-based declarative languages, but that cannot coexist within the same program in current systems.

The approach presented here starts from a call-time choice ambient (as given by most popular functional logic systems like *Curry* [6] or *Toy* [11]) and adds to it the possibility of annotating the evaluation of (sub)-expressions as following a run-time choice regime. We have proposed two variants of this idea, the first being more 'local' in the effect of an annotation $rt(e)$, while the second is more global. In both cases we have proposed a formal definition of the intended semantics.

For the first variant we have given formal operational descriptions, by adapting to the new setting two one-step reduction relations proposed in [9] as a simple notion of rewriting adequate for call-time choice. As for implementation, this variant has been achieved by modifying of the system *Toy*. Essentially, we have needed to change the management of *suspensions*, that are the technical key to implement sharing for call-time choice. The resulting prototype can be found at `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems`.

For the second variant we give a logical semantics that extends, to cope with *rt* annotations, the proof calculus of the *CRWL* framework [4]. We have seen how to transform annotations of this variant into the first one. This mapping can be used to implement the second variant.

Recently, we have tried a different alternative to the combination of call-time and run-time choice [10], following a way complementary to the one in this paper: there we start from ordinary rewriting and enhance it with local bindings through a *let* construct to express sharing and call-time choice. The resulting framework seems to be more amenable to formal treatments, as shown by the good number of technical results obtained in [10]. On the other hand, the approach here seems to be more easily implementable, at least if one wants to reuse existing call-time-choice based implementations.

References

1. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.
2. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, United Kingdom, 1998.
3. R. Caballero and J. Sánchez (eds.). TOY: A multiparadigm declarative language, version 2.2.3. Technical report, UCM, Madrid, July 2006.
4. 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.
5. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
6. M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.
7. H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.
8. R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. International Symposium on Programming Language Implementation and Logic Programming (PLILP'93)*, pages 184–200. Springer LNCS 714, 1993.
9. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proc. Principles and Practice of Declarative Programming*, pages 197–208. ACM Press, 2007.
10. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A flexible framework for programming with non-deterministic functions (extended version). Technical report, 2008. http://gpd.sip.ucm.es/juanrh/pubs/tchrRTCT08.pdf.
11. F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
12. H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.