Lazy Context Cloning for Non-Deterministic Graph Rewriting ⋆

Sergio Antoy Daniel W. Brown Su-Hui Chiang

Department of Computer Science
Portland State University
P.O. Box 751
Portland, OR 97207
USA

Abstract

We define a rewrite strategy for a class of non-confluent constructor-based term graph rewriting systems and discuss its correctness. Our strategy and its extension to narrowing are intended for the implementation of modern functional logic programming languages. Our strategy avoids the construction of large contexts of redexes with distinct replacements, an expensive and frequently wasteful operation executed by competitive complete techniques.

Key words: non-determinism, functional logic programming

1 Introduction

Non-determinism is one of the most appealing features of functional logic programing. A program is *non-deterministic* when its execution may evaluate some expression that has multiple results. To better understand this concept, consider a program to find a donor for a blood transfusion to a patient. The following declarations, in Curry [10], define the blood types and which type can be given to which other type:

```
data BloodTypes = Ap | An | ABp | ABn | Op | On | Bp | Bn

giveTo Ap = Ap ? ABp
giveTo Op = Op ? Ap ? Bp ? ABp                    (1)
giveTo Bp = Bp ? ABp
...
```

⋆ Partially supported by the NSF grant CCR-0218224.

[1] Emails: antoy@cs.pdx.edu, brownda@cs.pdx.edu, suhui@cs.pdx.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

For example, the first rule of `giveTo` states that the blood type $A+$, encoded as `Ap`, can be given to patients with blood types $A+$ and $AB+$. The evaluation of `giveTo Ap` non-deterministically returns `Ap` or `ABp`. The infix operator "`?`", called *choice operation*, selects either of its arguments. There are 5 other `giveTo` rules that are not shown.

A small database of people, patients and/or donors, and their blood types follow:

```
btype "John" = ABp
btype "Doug" = ABn                                    (2)
btype "Lisa" = An
```

The goal, given a patient, is to find a suitable donor for a transfusion. A non-deterministic program to solve this problem is natural, terse and elegant.

```
donorFor x
    | giveTo (btype y) =:= btype x & x =/= y          (3)
    = y where y free
```

The condition of operation `donorFor` holds when the blood of some donor `y` can be given to patient `x` and ensures that `y` is not `x`, since self donation is not intended. For example, the execution of `donorFor "John"` yields `"Doug"` or `"Lisa"` non-deterministically, whereas no donor is found for `"Lisa"` in our very small database of people (2). The evaluation of the program is by narrowing. In particular, when the condition of `donorFor` is evaluated, `y` is initially unknown and becomes instantiated to a suitable value, if one exists.

Non-determinism reduces the effort of designing and implementing data structures and algorithms to encode this problem into a program. The simplicity of the program inspires confidence in its correctness.

This paper addresses both theoretical and practical aspects of the implementation of non-determinism. Section 2 highlights some deficiencies of typical implementations of non-determinism and sketches our proposed solution. Section 3 discusses the background of our work. Section 4 defines our strategy and related concepts. Section 5 briefly addresses related work.

2 Motivation

Functional logic programs are traditionally seen as term rewriting systems (TRSs) [5] with the constructor discipline [13]. The execution of a program is the repeated application of narrowing steps to a term until either a constructor term is reached, in which case the computation *succeeds*, or an unnarrowable term with some occurrence of a defined operation is reached, in which case the computation *fails*. Examples of the latter are an attempt to divide by zero or to return the first element of an empty list.

A TRS with non-deterministic operations is typically non-confluent. Operationally, there are two main approaches to computations in a non-confluent

TRS: *backtracking* and *copying*. While the former is standard terminology, we do not know any commonly accepted name for the latter. Copying is more powerful since steps originating from distinct non-deterministic choices can be interleaved, which is essential to ensure the completeness of the results. We informally describe a computation of a term with each approach. Let $t[u]$ be a term in which $t[\,]$ is a context and u is a subterm that non-deterministically evaluates to x or y.

With *backtracking*, the computation of $t[u]$ first requires the evaluation of $t[x]$. If this evaluation fails to produce a constructor term, the computation continues with the evaluation of $t[y]$. Otherwise, if and when the evaluation of $t[x]$ completes, the interpreter may give the user the option of evaluating $t[y]$.

With *copying*, the computation of $t[u]$ consists in the simultaneous, e.g., by interleaving steps, independent evaluations of $t[x]$ and $t[y]$. If either evaluation produces a constructor term, this term is a result of the computation, and the interpreter may give the user the option of continuing the evaluation of the other term. If the evaluation of one term fails to produce a constructor term, the evaluation of the other term continues unaffected.

Both backtracking and copying have been used in the implementation of FL languages. For example, Pakcs [9] and TOY [12] are based on backtracking, whereas the FLVM [4] and the interpreter of Tolmach et al. [15] are based on copying. Unfortunately, both backtracking and copying as described above have non-negligible drawbacks. Consider the following program, where `div` denotes the usual integer division operator and n is some positive integer.

```
loop = loop
f x = 1+(2+(...+(n `div` x)...))
```
(4)

We describe the evaluation of $t =$ `f (loop ? 1)` with backtracking. If the first choice for the non-deterministic expression is `loop`, no value of t is ever computed although t has a value, since the evaluation of `f loop` does not terminate. This is a well-known problem of backtracking referred to as loss of *completeness*. Since narrowing computations are complete with an appropriate strategy [3], in this example the culprit is backtracking.

We describe the evaluation of $t =$ `f (0 ? 1)` with copying. Both `f 0` and `f 1` are evaluated. Of course, the evaluation of the first one fails. The problem in this case is the construction of the term `1+(2+(...+(`n `` `div` `` `0)...))`. The effort to construct this term, which becomes arbitrarily large as n grows, is wasted, since the first step of the computation, which is needed [3], is a division by zero and consequently the computation fails.

Thus, copying may needlessly construct terms, and backtracking may fail to produce results. To avoid these drawbacks, we propose a new approach to non-deterministic computations. Instead of evaluating only one non-deterministic choice or copying the entire context for each non-deterministic choice, we slowly "bubble" the non-deterministic choices up their contexts. Informally, the evaluation of `f (0 ? 1)` goes through the following intermediate

terms, where `fail` is a distinguished symbol denoting any expression that cannot be evaluated to a constructor term:

```
f (0 ? 1)
    → 1+(2+(...+(n `div` (0 ? 1))...))
    → 1+(2+(...+((n `div` 0) ? (n `div` 1))...))
    → 1+(2+(...+(fail ? (n `div` 1))...))
    → 1+(2+(...+(n `div` 1)...))
```
(5)

Because `fail` occurs at a position where a constructor-rooted term is needed for the execution of a needed step, the `fail` choice is eliminated. Since no rewrite rule matches `fail` in any position, no constructor term can be derived from that choice.

In this example, the obvious advantages of our approach are that no choice is left behind and no unnecessarily large context is copied. However, determining when and how to bubble a choice is less trivial than it appears in this example. Consider the following operation:

```
f x = (not x, not x)
```
(6)

and the term t = `f (True ? False)`. The evaluation semantics of non-right linear rewrite rules, such as (6), is called *call-time* choice [11]. Informally, the non-deterministic choice for the argument of `f` is made at the time of `f`'s invocation. Therefore, the instances of `x` in the right-hand side of (6) should all evaluate to `True` or all to `False`. The term being evaluated is graphically depicted in the left-hand side of the following figure:

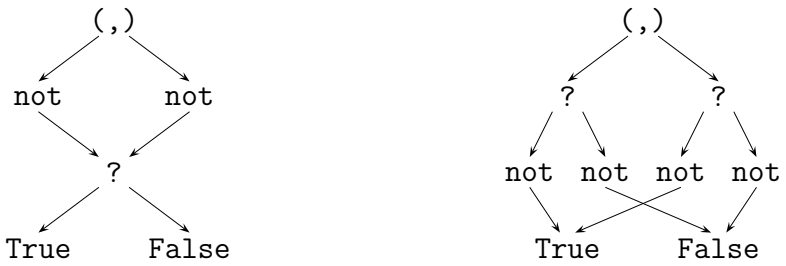

Fig. 1. The left-hand side depicts a term graph. The right-hand side is obtained from the left-hand side by bubbling up to the parents the non-deterministic choice. The two term graphs have a different set of constructor normal forms.

The right-hand side of the above figure shows the result of bubbling up the non-deterministic choice in a way similar to (5). This term has 4 normal forms. One is `(True,False)` which is not obtainable with either backtracking or copying, and it is not intended by the call-time choice semantics. Therefore, although advantageous in some situations, unrestricted bubbling can be unsound.

In the following sections we formalize a sound approach to non-deterministic computations with shared terms based on the idea of bubbling introduced in this section.

3 Background

Modern FL languages use narrowing for computing. Echahed and Janodet [6] define a theoretically efficient narrowing strategy for the inductively sequential *graph* rewriting systems. This strategy adequately models sharing with graphs but does not support the non-deterministic programs of this paper. Antoy [2] defines a theoretically efficient strategy for the *overlapping* inductively sequential term rewriting systems This class adequately models non-determinism—as in the programs of this paper—but it does not consider sharing.

An adequate background theory for our work would be the combination of the above extensions. Unfortunately, this combination has not yet been formalized. We do not foresee any substantial problem in combining [6] and [2]. The formalization of term graphs does not depend on inductive sequentiality, and the strategy of [6] depends on the rule's left-hand sides. Extending it from the inductively sequential TRSs to the overlapping inductively sequential TRSs poses no problem, since the rule's left-hand sides are the same for terms and term graphs. Likewise, the notion of overlapping inductive sequentiality does not depend on differences between terms and graphs, and the strategy of [2] depends on the rule's left-hand sides. Extending this strategy from terms to term graphs poses no problem as well, since the rule's left-hand sides are the same for overlapping and non-overlapping inductively sequential TRSs.

In the rest of this paper, we assume that programs are possibly overlapping inductively sequential *admissible* term graph rewriting systems, abbreviated GRSs. We adopt the notation and definitions of [6]. In particular, we recall that a graph is admissible if none of its defined operations belongs to a cycle. We need an additional definition:

Definition 3.1 *A node d* dominates *a node n in a rooted graph g if every path from the root of g to n contains d. If d and n are distinct, then d* properly dominates *n in g.*

4 Formalization

We consider an overlapping inductively sequential GRS S. The GRS S includes the *choice operation*, shown in the introduction, denoted by the infix operator "?" and defined by the following rewrite rules:

```
x ? y = x
x ? y = y
```
(7)

We assume that these are the only overlapping rules of S. Any other overlap can be eliminated, without altering the computations, using the choice operation [2]. The evaluation of an admissible term graph g_0 in S is a sequence of graphs $g_0 \overset{\sim}{\rightarrow} g_1 \overset{\sim}{\rightarrow} g_2 \cdots$ where for every natural number i, g_{i+1} is obtained from g_i either with a rewrite step of S or with a *bubbling* step, which will be formalized shortly.

Definition 4.1 (Partial renaming) *Let $g = \langle \mathcal{N}_g, \mathcal{L}_g, \mathcal{S}_g, \mathcal{R}\text{oots}_g \rangle$ be a term graph over $\langle \Sigma, \mathcal{N}, \mathcal{X} \rangle$, $\mathcal{N}_p$ a subset of $\mathcal{N}_g$ and $\mathcal{N}_q$ a set of nodes disjoint from $\mathcal{N}_g$. A* partial renaming *of g with respect to $\mathcal{N}_p$ and $\mathcal{N}_q$ is a bijection $ren_{pq} : \mathcal{N} \to \mathcal{N}$ such that:*

$$ren_{pq}(n) = \begin{cases} n' & \text{where } n' \in \mathcal{N}_q, \text{ if } n \in \mathcal{N}_p; \\ n & \text{otherwise.} \end{cases}$$

We overload ren_{pq} to graphs as follows: $ren_{pq}(g) = g'$ is a graph over $\langle \Sigma, \mathcal{N}, \mathcal{X} \rangle$ such that:

- $\mathcal{N}_{g'} = ren_{pq}(\mathcal{N}_g)$,
- $\mathcal{L}_{g'}(m) = \mathcal{L}_g(n)$, *iff* $m = ren_{pq}(n)$,
- $m_1 m_2 \dots m_k = \mathcal{S}_{g'}(m_0)$ *iff* $n_1 n_2 \dots n_k = \mathcal{S}_g(n_0)$, *where for* $i = 0, 1, \dots k$, $k \geqslant 0$, $m_i = ren_{pq}(n_i)$,
- $\mathcal{R}\text{oots}_{g'} = \mathcal{R}\text{oots}_g$.

In simpler words, g' is equal to g in all aspects except that some nodes in $\mathcal{N}_g$, more precisely all and only those in $\mathcal{N}_p$, are consistently renamed, with a "fresh" name, in g'. Obviously, in any partial renaming ren_{pq}, the cardinalities of $\mathcal{N}_p$ and $\mathcal{N}_q$ are the same.

Definition 4.2 (Bubbling) *Let g be a graph and c a node of g such that the subgraph of g at c is of the form $x \,?\, y$, i.e., $g|_c = x \,?\, y$. Let d be a proper dominator of c in g and $\mathcal{N}_p$ the set of nodes that are on some path from d to c in g, including d and c, i.e., $\mathcal{N}_p = \{n \mid n_1 n_2 \dots n_k \in \mathcal{P}(d, c)$ and $n = n_i$ for some $i\}$. Let $\mathcal{N}_x$ and $\mathcal{N}_y$ be set of nodes disjoint from $\mathcal{N}_g$ and from each other, such that ren_{px} and ren_{py} are partial renamings of g. Let $g_q = ren_{pq}(g|_d[c \leftarrow q])$, for $q \in \{x, y\}$. The* bubbling *relation on graphs is denoted by "$\simeq$" and defined by $g \simeq g[d \leftarrow g_x \,?\, g_y]$.*

In simpler words, bubbling moves a choice in a graph up to a dominator node. To execute this move some portions of the graph, more precisely those between the end points of the move, must be cloned. An example of bubbling is shown in Figure 2.

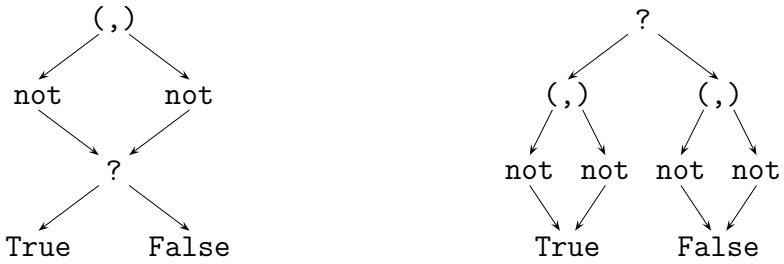

Fig. 2. The left-hand side depicts a term graph. The right-hand side is obtained from the left-hand side by bubbling up to a proper dominator the non-deterministic choice. The two term graphs have the same set of constructor normal forms.

The bubbling relation entails 3 graph replacements. The graphs involved in these replacements are all compatible [6, Def. 6] with each other. Therefore,

the bubbling relation is well defined according to [6, Def. 9].

Our approach never applies a rule of the choice operation. In a constructor-based GRSs, this is equivalent to considering the choice symbol a constructor rather than an operation. This has far reaching consequences.

One consequence is that every operation of the GRS becomes incompletely defined, e.g., `not` $(x$ `?` $y)$ cannot be reduced even if x and y are Boolean values. Therefore, we handle reductions involving the choice symbol in a needed position using the strategy that we define below.

A second consequence is that the results of computations change, but this change is more apparent than substantial. For example, the standard evaluation of $t =$ `True ? False` has two results, `True` and `False`. With our approach, t is a normal form. To a large extent, the difference is only in the *representations* of the results. Simple transformations allow us to manipulate non-standard representations as the standard ones.

A third consequence is a significant change in the characteristics of both the program and its computation space. If overlapping rules are eliminated, and only admissible graphs are considered, the program becomes confluent. The computation space of a graph in a non-deterministic program is a *tree* of graphs where a child is obtained from its parent with a reduction step. A branch occurs when a redex admits two or more non-deterministic replacements. In our framework, non-deterministic replacements are eliminated, and consequently the computation space of a graph is a *sequence* of graphs. The graph at the position $i + 1$ in the sequence is obtained from the graph at the position i with either a reduction step or a bubbling step.

Since there are no non-deterministic steps, a redex has only one replacement. In particular, at the machine or implementation level, a redex can always be replaced *in place*, i.e., in the execution of a step, the context of the redex becomes the context of the redex's replacement. Although in this paper we consider only rewriting, we believe that these appealing characteristics of the search space could be extended to narrowing steps as well. We leave this extension to future work.

We are now ready to define the strategy. In constructor-based TRSs and GRSs the strategy [3,6] takes an operation rooted term or term graph and uses a definitional tree of the root symbol to compute either a step or a set of steps depending on the class of programs. The following definition is adapted from [6, Def. 29] and uses the same notation and terminology.

Definition 4.3 (Strategy) *The function φ takes two arguments, an admissible operation-rooted term graph t and a partial definitional tree $\mathcal{T}$ such that $pattern(\mathcal{T}) \leqslant t$. The function φ yields a set of pairs (p, R), where p is a node*

of t and R is a rewrite rule or the distinguished symbol "$\simeq$".

$$\varphi(t, \mathcal{T}) \ni \begin{cases} (\mathcal{R}oot_t, R) & \text{if } \mathcal{T} = rule(\pi, R); \\ (p, R) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\ & \quad pattern(\mathcal{T}_i) \leqslant t, \text{ for some } i, \text{ and} \\ & \quad \varphi(t, \mathcal{T}_i) \ni (p, R); \\ (p, R) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\ & \quad \pi \text{ matches } t \text{ at the root by homom. } h : \pi \to t, \\ & \quad h(o) \text{ is labeled with "?" in } t, \\ & \quad q \text{ is a successor of } h(o) \text{ in } t, \text{ and} \\ & \quad \psi(h(o), t|_q, \mathcal{T}) \ni (p, R); \\ (p, R) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\ & \quad \pi \text{ matches } t \text{ at the root by homom. } h : \pi \to t, \\ & \quad h(o) \text{ is labeled with an operation } f \text{ in } t, \\ & \quad \mathcal{T}' \text{ is a definitional tree of } f, \text{ and} \\ & \quad \varphi(t|_{h(o)}, \mathcal{T}') \ni (p, R). \end{cases}$$

where

$$\psi(c, t, \mathcal{T}) \ni \begin{cases} (c, \simeq) & \text{if } \mathcal{T} = rule(\pi, R); \\ (p, R) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\ & \quad pattern(\mathcal{T}_i) \leqslant t, \text{ for some } i, \text{ and} \\ & \quad \psi(c, t, \mathcal{T}_i) \ni (p, R); \\ (p, R) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\ & \quad \pi \text{ matches } t \text{ at the root by homom. } h : \pi \to t, \\ & \quad h(o) \text{ is labeled with "?" in } t, \\ & \quad q \text{ is a successor of } h(o) \text{ in } t, \text{ and} \\ & \quad \psi(c, t|_q, \mathcal{T}) \ni (p, R); \\ (p, R) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\ & \quad \pi \text{ matches } t \text{ at the root by homom. } h : \pi \to t, \\ & \quad h(o) \text{ is labeled with an operation } f \text{ in } t, \\ & \quad \mathcal{T}' \text{ is a definitional tree of } f, \text{ and} \\ & \quad \varphi(t|_{h(o)}, \mathcal{T}') \ni (p, R). \end{cases}$$

A pair (p, R) in the set computed by φ on a graph t is interpreted as a step of the computation of t as follows. If R is a rule, then the rule is applied at the node p of t. If R is the symbol "$\simeq$", then the choice at p is bubbled in t according to Def. 4.2.

Our strategy is structurally similar to previously proposed strategies [3] except for the third case. Intuitively, when a choice is encountered in an inductive position of a definitional tree, the strategy "glides" over the choice and continues with the choice's arguments, but its behavior changes. This is why the

function ψ is introduced and carries an extra argument. The function ψ is very similar to φ, but it returns a bubbling step instead of a reduction step if it finds a *rule* node in the definitional tree. This means that a reduction would be possible if the choice were not in the way. Therefore, the strategy clones some portion of the context of a non-deterministic choice if and only if bubbling enables a reduction step.

We are implementing our strategy in the FLVM to assess its properties. We expect that its efficiency will surpass that of other strategies for computations that make frequent non-deterministic choices that quickly fail. A modest overhead may occur in deterministic-only computations.

To prove the correctness of our strategy we are working on two aspects: computations and the representation of the results. Computations combine both bubbling and rewriting steps. We must prove that the results obtainable by a computation with combined steps are all and only those obtainable by pure rewriting.

Standard notions of soundness and completeness must be proved for our strategy as well. Since in our approach the representation of the result of a computation is non-standard, some extra work is involved. However, extracting results in standard representation from our representation of results is straightforward.

5 Related work

Although strategies for functional logic computations [3] and term graph rewriting [14] have been intensely investigated, the work on strategies for term graph rewriting systems as models of functional logic programs has been relatively scarce. The line of work closest to ours is [6,7]. A substantial difference of our work with this line is the class of programs we consider, namely non-deterministic ones. The attempt to minimize the cost of non-deterministic steps by limiting the cloning of the context of a redex is original.

Other efforts on handling non-determinism in functional and functional logic computations with shared subexpressions include [11], which introduces the *call-time choice semantics* to ensure that shared terms are evaluated to the same result; [8], which defines a rewriting logic that among other properties provides the call-time choice; and [1] and [15], which define operational semantics based on *heaps* and *stores* specifically for the problem we are discussing. Our work is in line with these efforts, but it is explicitly based on term graph rewriting rather than computational data structures.

References

[1] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of functional logic programs based on needed narrowing. *Theory and Practice of Logic Programming*, 5(3):273–303, 2005.

[2] S. Antoy. Optimal non-deterministic functional logic computations. In *Proceedings of the Sixth International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30, Southampton, UK, September 1997. Springer LNCS 1298.

[3] S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.

[4] S. Antoy, M. Hanus, J. Liu, and A. Tolmach. A virtual machine for functional logic computations. In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pages 108–125, Lubeck, Germany, Sept. 2005. Springer LNCS 3474.

[5] M. Bezem, J. W. Klop, and R. de Vrijer (eds.). *Term Rewriting Systems*. Cambridge University Press, 2003.

[6] R. Echahed and J.-C. Janodet. On constructor-based graph rewriting systems. Research Report 985-I, IMAG, 1997. Available at ftp://ftp.imag.fr/pub/LEIBNIZ/ATINF/c-graph-rewriting.ps.gz.

[7] R. Echahed and J.-C. Janodet. Admissible graph rewriting and narrowing. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 325 – 340, Manchester, June 1998. MIT Press.

[8] J. C. González Moreno, F. J. L. Fraguas, M. T. H. González, and M. R. Artalejo. An approach to declarative programming based on a rewriting logic. *The Journal of Logic Programming*, 40:47–87, 1999.

[9] M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at http://www.informatik.uni-kiel.de/~pakcs, 2003.

[10] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.7). Available at http://www.informatik.uni-kiel.de/~curry, 2000.

[11] H. Hussmann. Nondeterministic algebraic specifications and nonconfluent rewriting. *Journal of Logic Programming*, 12:237–255, 1992.

[12] F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proceedings of RTA '99*, pages 244–247. Springer LNCS 1631, 1999.

[13] M. J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.

[14] D. Plump. Term graph rewriting. In H.-J. K. H. Ehrig, G. Engels and G. Rozenberg, editors, *Handbook of Graph Grammars*, volume 2, pages 3–61. World Scientific, 1999.

[15] A. Tolmach, S. Antoy, and M. Nita. Implementing functional logic languages using multiple threads and stores. In *Proc. of the Ninth International Conference on Functional Programming (ICFP 2004)*, pages 90–102, Snowbird, Utah, USA, Sept. 2004. ACM Press.