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 prove its correctness. Our strategy and its extension to narrowing are intended for the implementation of non-strict non-deterministic functional logic programming languages. Our strategy is based on a graph transformation, called bubbling, that 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-deterministic computations, functional logic programming, bubbling, graph transformations

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 [18], 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 follows:

```
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 proves the soundness and completeness of our strategy. Section 6 very briefly discusses problems and some solutions of the extension of the strategy to narrowing. Section 7 briefly addresses related work. Section 8 offers our conclusion.

2 Motivation

Functional logic programs are traditionally seen as term rewriting systems (TRSs) [9,11,12,21] with the constructor discipline [23]. 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 strategy computes steps on a term. The set of all the terms obtained from a term t with repeated applications of the strategy S is the *computation space* of t (according to S). For the TRSs that we consider, the computation space of a term is a tree-like structure. A child is obtained through the application of a step to its parent. A tree branch occurs when the strategy computes two or more distinct steps on the same term. When a parent has several children, the order in which the strategy is applied to these children and their descendants is important, although most strategies for functional logic languages [4] are unconcerned with this order. The order of application affects only how the computation space of a term is traversed or explored, not the content of the space itself.

In practice, there are two main approaches: *depth-first* and *breadth-first*. Operationally, these approaches are implemented by *backtracking* and *copying*, respectively. While the former is standard terminology, we do not know any commonly accepted name for the latter. 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 [17] and $\mathcal{TOY}$ [22] are based on backtracking, whereas the FLVM [8] and the interpreter of Tolmach et al. [25] 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 [4], 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 [4], 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. In the second step, we have distributed the parent of an occurrence of the choice operation over its arguments. Unfortunately, a "distributive property" of the kind $f(x \,?\, y) = f(x) \,?\, f(y)$ is unsound in the presence of sharing. 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 [20]. 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:

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 is unsound.

Section 4 introduces a definition of bubbling that is sound and central to

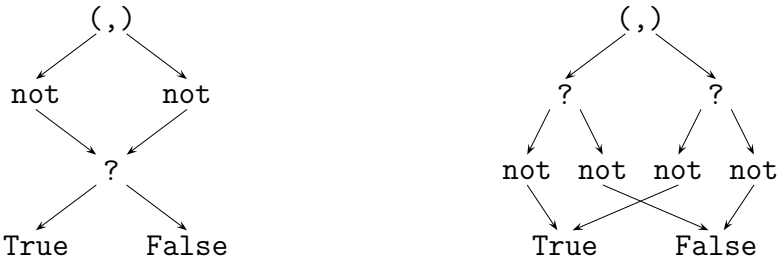

Fig. 1. The left-hand side depicts a term graph. The right-hand side is obtained from the left-hand side by bubbling up to the parents the non-deterministic choice. The two term graphs have a different set of constructor normal forms.

the strategy we present. Some properties of bubbling are discussed in [5].

3 Background

Modern FL languages use narrowing for computing [16]. Echahed and Janodet [13] 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 [3] 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 [13] and [3]. The formalization of term graphs does not depend on inductive sequentiality, and the strategy of [13] 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 [3] 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.

The theory of graph rewriting is significantly more complicated than that of term rewriting. Furthermore, there are multiple presentations with non-trivial variations in the literature. In this paper, we follow the systemization of Echahed and Janodet [13] because the class that they consider is a good fit for our programs, as we will discuss later. The space allotted to this paper allows us only to recall the key concepts. The complete details can be found in [13].

Definition 3.1 Let Σ be a *signature*, $\mathcal{X}$ a countable set of *variables*, and $\mathcal{N}$ a countable set of *nodes*. A (*rooted*) *graph* over $\langle \Sigma, \mathcal{N}, \mathcal{X} \rangle$ is a 4-tuple

$g = \langle \mathcal{N}_g, \mathcal{L}_g, \mathcal{S}_g, \mathcal{R}\text{oots}_g \rangle$ such that:

1. $\mathcal{N}_g \subset \mathcal{N}$ is the set of nodes of g;
2. $\mathcal{L}_g : \mathcal{N}_g \to \Sigma \cup \mathcal{X}$ is the *labeling* function mapping each node of g to a signature symbol or a variable;
3. $\mathcal{S}_g : \mathcal{N}_g \to \mathcal{N}_g^*$ is the *successor* function mapping each node of g to a possibly empty string of nodes of g, such that if $\mathcal{L}_g(n) = s$, where $s \in \Sigma \cup \mathcal{X}$, and (for the following condition, we assume that a variable has arity zero) $\mathit{arity}(s) = k$, then there exist $n_1, \ldots, n_k$ in $\mathcal{N}_g$ such that $\mathcal{S}_g(n) = n_1 \ldots n_k$;
4. $\mathcal{R}\text{oots}_g \subseteq \mathcal{N}_g$ is a subset of nodes of g called the *roots* of g;
5. if $\mathcal{L}_g(n_1) \in \mathcal{X}$ and $\mathcal{L}_g(n_2) \in \mathcal{X}$ and $\mathcal{L}_g(n_1) = \mathcal{L}_g(n_2)$, then $n_1 = n_2$, i.e., every variable of g labels one and only one node of g; and
6. for each $n \in \mathcal{N}_g$, either $n \in \mathcal{R}\text{oots}_g$ or there is a path from r to n where $r \in \mathcal{R}\text{oots}_g$, i.e., every node of g is reachable from some root of g.

A graph is called a *term* (*graph*) if $\mathcal{R}\text{oots}_g$ is a singleton.

In the pictorial representation of graphs, e.g., as in Fig. 1, we do not show the nodes' names, but only their labels. The nodes' names are arbitrary and irrelevant to most purposes.

The following definition is essential to our approach.

Definition 3.2 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.

For example, in the left-hand side graph of Fig. 1, the occurrence of "?" is properly dominated by the root only. Every other occurrence, except the root, is properly dominated by its predecessor.

4 Formalization

The formulation of the strategy comprises various pieces. In *constructor-based* TRSs and GRSs the core of a strategy [4,13] is a function that 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. Definitional trees were introduced in [2]. We recall this notion below. Examples are found in [4]. A *pattern* is a term graph of the form $f(t_1, \ldots, t_n)$ where f is an operation symbol and $t_1, \ldots, t_n$ are constructor terms.

Definition 4.1 [Definitional tree] A *definitional tree* of an operation f is a finite non-empty set $\mathcal{T}$ of linear patterns partially ordered by subsumption and having the following properties up to a renaming of variables:

- [leaves property] The maximal elements, referred to as the *leaves*, of $\mathcal{T}$ are all and only variants of the left hand sides of the rules defining f. Non-maximal

elements are referred to as *branches*.

- [root property] The minimum element, referred to as the *root*, of $\mathcal{T}$ is $f(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are fresh, distinct variables.
- [parent property] If π is a pattern of $\mathcal{T}$ different from the root, there exists in $\mathcal{T}$ a unique pattern π' strictly preceding π such that there exists no other pattern strictly between π and π'. π' is referred to as the *parent* of π and π as a *child* of π'.
- [induction property] All the children of a pattern π differ from each other only at common position which is referred to as *inductive*. The inductive position is the position of a variable in π.

In a constructor-based GRS, a rewrite rule is a pair $l \to r$ of term graphs where l is a pattern. A rewrite rule $l \to r$ *defines* an operation f iff the root node of l is labeled by f. An operation f is *inductively sequential* if the set of the left-hand sides of the rules defining f has a definitional tree. A GRS is *inductively sequential* iff all its operations are inductively sequential.

We consider an *overlapping inductively sequential* [3] GRS S. In this class, the left-hand sides of two rules can overlap, but only if they are variants of each other, i.e., they differ at most by a renaming of their variables. 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. Overlapping originating from other rules can be eliminated, without altering the computations, using the choice operation [3].

Definition 4.2 [Limited overlapping] A *limited overlapping* inductively sequential GRS, abbreviated *LOIS*, is a constructor based GRS, S, such that S contains the choice operation "`?`" defined in (7) and every other defined operation of S is inductively sequential.

In the rest of this paper, we assume that programs are possibly overlapping inductively sequential *admissible* term graph rewriting systems. These programs will be abbreviated as GRSs. We recall that a graph is *admissible* [13] if none of its defined operations belongs to a cycle.

A computation in our approach consists of two kinds of steps: an ordinary rewrite step or multistep and the new kind of step that we called bubbling earlier. The following definitions formalize bubbling steps.

Definition 4.3 [Partial renaming] Let $g = \langle \mathcal{N}_g, \mathcal{L}_g, \mathcal{S}_g, \mathcal{R}\mathrm{oots}_g \rangle$ 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 $\Theta : \mathcal{N} \to \mathcal{N}$

such that:

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

By analogy with the terminology for substitutions, we call $\mathcal{N}_p$ and $\mathcal{N}_q$ the *domain* and *image* of Θ, respectively. We overload Θ to graphs as follows: $\Theta(g) = g'$ is a graph over $\langle \Sigma, \mathcal{N}, \mathcal{X} \rangle$ such that:

- $\mathcal{N}_{g'} = \Theta(\mathcal{N}_g)$,
- $\mathcal{L}_{g'}(\Theta(n)) = \mathcal{L}_g(n)$,
- $\mathcal{S}_{g'}(\Theta(n)) = \Theta(n_1)\Theta(n_2)\ldots\Theta(n_k)$ iff $\mathcal{S}_g(n) = n_1 n_2 \ldots n_k$, for $k \geqslant 0$,
- $\mathcal{R}\text{oots}_{g'} = \Theta(\mathcal{R}\text{oots}_g)$.

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, the cardinalities of the domain and the range are the same.

Definition 4.4 [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., $\mathcal{N}_p = \{n \mid n_1 n_2 \ldots n_k \in \mathcal{P}_g(d, c) \text{ and } n = n_i \text{ for some } i\}$, where $\mathcal{P}_g(d, c)$ is the set of all paths from d to c in g. Let Θ_x and Θ_y be partial renamings of g with domain $\mathcal{N}_p$ and disjoint images. Let $g_q = \Theta_q(g|_d[c \leftarrow q])$, for $q \in \{x, y\}$. The *bubbling* relation on graphs is denoted by "$\simeq$" and defined by $g \simeq g[d \leftarrow g_x ?\, g_y]$, where the root node of the replacement of g at d is fresh. We call c and d the *origin* and *destination*, respectively, of the bubbling step, and we denote the step with "$\simeq_{cd}$" when this information is relevant.

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.

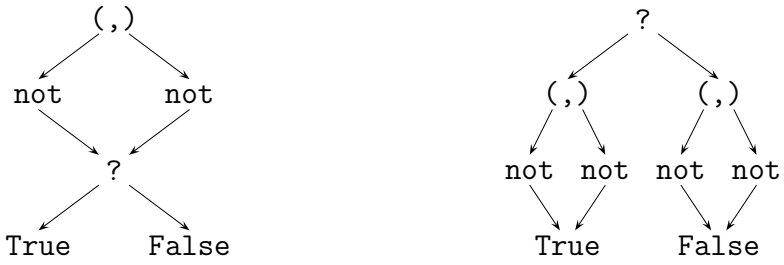

Fig. 2. The left-hand side depicts a term graph. The right-hand side is obtained from the left-hand side by bubbling up to a proper dominator the non-deterministic choice. The two term graphs have the same set of constructor normal forms.

The bubbling relation entails 3 graph replacements. The graphs involved in these replacements are all compatible [13, Def. 6] with each other. Therefore, the bubbling relation is well defined according to [13, 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 never applied, and only admissible graphs are considered, the program becomes confluent. Non-deterministic replacements are eliminated, and consequently the computation space of a graph is a *sequence*, rather than a tree, 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.

For defining our strategy, which extends [13, Def. 29], we need a representation of definitional trees. Since "`?`" is the only overlapping operation, and we never apply its rules, we only need to represent trees of non-overlapping operations. Our representation not only stores the patterns but also makes explicit the inductive positions and the parent-child relationship. In the representation, the symbols *rule* and *branch* are uninterpreted functions used to classify the elements of a tree. The representation of a leaf with pattern π is $rule(\pi \to r)$, where $\pi \to r$ is a variant of a rewrite rule. We represent the entire rule, rather than its left-hand side only, because this eases formulating the strategy. The representation of a branch with pattern π is $branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$, where o is the inductive position of π, and $\mathcal{T}_1, \ldots, \mathcal{T}_n$ are the representations of all the children of π.

Definition 4.5 [HNF Strategy] The function φ takes two arguments, an admissible operation-rooted term graph t and a partial definitional tree $\mathcal{T}$ such that $pattern(\mathcal{T}) \leqslant t$, and yields a set of pairs (p, R), where p is a node of t and R is either a rewrite rule or the distinguished symbol "$\simeq$", according to the definition in Figure 3.

A pair (p, R) in the set computed by φ on a graph t is interpreted as follows. If R is a rule, then a rewrite step with this rule is applicable at the

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

where

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

Fig. 3. The function φ defines the strategy subject of this paper on the operation-rooted admissible term graphs of a limited overlapping inductively sequential graph rewriting system. The conditions for the application of φ are described in Definition 4.5.

node p of t. If R is the symbol "$\simeq$", then a bubbling step with origin p is applicable to t according to Def. 4.4.

Our strategy, defined by cases, is structurally similar to previously proposed strategies [4] 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.

When more than one choice is in the way of a redex, only the highest is selected as the origin of a bubbling step. This can be inferred by the third case of the definition of ψ. The choice passed down to the recursive invocation is the same as the original invocation.

We extend Definition 4.5 from *operation-rooted* terms to term graphs rooted by constructors and choices. We overload the symbol φ, since the intent is the same.

Definition 4.6 [Strategy] Let $\mathcal{R}$ be a LOIS and t an admissible term graph over the signature of $\mathcal{R}$. We define:

$$\varphi(t) = \begin{cases} \cup_{i=1}^{n} \varphi(t_i) & \text{if } t = c(t_1, \ldots, t_n) \text{ and either} \\ & \quad c \text{ is a constructor or } c = \texttt{?}; \\ \varphi(t, \mathcal{T}) & \text{if } t = f(t_1, \ldots, t_n),\ f \text{ is an operation and} \\ & \quad \mathcal{T} \text{ is a definitional tree of } f. \end{cases}$$

Since the strategy applied to an admissible term graph t computes a set containing several rewriting and/or bubbling steps, in the following definition we specify how these steps are to be applied to t. Observe that if the strategy computes a bubbling step $(p, \simeq)$, then p has an operation-rooted ancestor, which we denote with $o(p)$, such that every node in a path from $o(p)$ to p is labeled by a constructor symbol. If the destination of the bubbling step $(p, \simeq)$ is $o(p)$ or an ancestor of $o(p)$, a redex is created at $o(p)$. If the destination of the bubbling step $(p, \simeq)$ is a node labeled by a constructor symbol in the path from $o(p)$ to p, no redex is created. Instead, a further application of φ would compute another bubbling step of the choice just bubbled and so on until the choice is eventually bubbled at or above $o(p)$.

Definition 4.7 [Computation] Let $\mathcal{R}$ be a LOIS and t an admissible term graph over the signature of $\mathcal{R}$. A *computation* of t according to φ is a sequence $t = t_0 \rightarrow_\varphi t_1 \rightarrow_\varphi \cdots$ such that, for all $i > 0$, t_i is obtained from t_{i-1} as follows. Let $S = \varphi(t_{i-1})$. If S contains some bubbling step $(p, \simeq)$, then $t_{i-1} \simeq_{pq} t_i$, where q is $o(p)$ or some ancestor of $o(p)$. Otherwise $S = \{(p_k, R_k)\}_{k=1,n}$, $n > 0$, and $t_{i-1} = u_0 \rightarrow_{(p_1,R_1)} u_1 \rightarrow_{(p_2,R_2)} \cdots \rightarrow_{(p_n,R_n)} u_n = t_i$.

The above definition is non-deterministic both in the choice of a bubbling step, when more than one is computed by φ, and in the order in which rewrite steps are applied, when more than one rewrite step is computed. The following claims show that this non-determinism does not affect the results of a computation.

Theorem 4.8 *Let $\mathcal{R}$ be a LOIS and t an admissible term graph over the signature of $\mathcal{R}$. If $t \simeq u$, for some term u, then the constructor normal forms*

of u are all and only those of t.

Proof. A constructor normal form of u is a normal form of t [5, Lemma 5]. A constructor normal form of t is a normal form of u [5, Theorem 2]. □

The bubbling relation is not terminating. However, our strategy never computes infinite sequences consisting exclusively of bubbling steps.

Lemma 4.9 *Let $\mathcal{R}$ be a LOIS and t_0 an admissible term graph over the signature of $\mathcal{R}$. If $t_0 \to_\varphi t_1 \to_\varphi \cdots$ is an infinite sequence of steps computed by φ, then for every $p \geqslant 0$ there exists a $q \geqslant p$ such that the step $t_q \to_\varphi t_{q+1}$ is not a bubbling step.*

Proof. We define a well-founded ordering on terms and prove that bubbling sequences are decreasing with respect to this ordering. If t is a term graph and s a node of t, let $\delta(s)$ be the minimal number of nodes labeled by an operation symbols in any path from the root of t to s, and let $\xi(s)$ be either the sequence $1\,0\,0\ldots0$, where there are $\delta(s)$ zeros if the label of s is "`?`", or the empty sequence if the label of s is not "`?`". Intuitively, δ is the *depth* and ξ is the *momentum* of a node labeled by "`?`". We overload ξ on terms: for any term graph t, $\xi(t) = \Sigma_{s\in\mathcal{N}(t)}\xi(s)$, i.e., $\xi(t)$ is the component-wise sum of $\xi(s)$ for every node s of t, i.e., the momentum of a term is total of the momentums of all its nodes. Let t and u be term graphs with $\xi(t) = a_n\, a_{n-1} \ldots a_0$, where a_i, $n \geqslant i \geqslant 0$, is a natural and $a_n > 0$, and likewise $\xi(u) = b_m\, b_{m-1} \ldots b_0$. We define $\xi(t) \succ \xi(u)$ iff either $n > m$ or $n = m$ and there exists some k, $0 \leqslant k \leqslant n$, such that $a_k > b_k$ and for all i, $k < i \leqslant n$, $a_i = b_i$. Finally, we extend $\succ$ on terms: $t \succ u$ iff $\xi(t) \succ \xi(u)$.
We now show that if $t \simeq_{cd} u$ is computed by φ, then $t \succ u$. By Definition 4.7, d is at or above $o(c)$ and $o(c)$ is operation-rooted. Let k be the depth of c. The choice at c is "moved" to d, hence above $o(c)$. In the bubbling step, the nodes between c and d are cloned. The depth of these nodes is strictly smaller than k because they are above c. The depth of any other node labeled by "`?`" is unchanged. Thus, either $\delta(u) < k$ or $a_k > b_k$. This entails that $\xi(t) \succ \xi(u)$ and hence $t \succ u$. By Noetherian induction on $\succ$, it follows that there is no infinite sequence of bubbling steps computed by φ. □

Finally, we show that the order in which the rewrite steps computed by φ are applied is irrelevant.

Theorem 4.10 *Let $\mathcal{R}$ be a LOIS, t_0 an admissible term graph over the signature of $\mathcal{R}$ and $S = \varphi(t)$. For all distinct rewrite steps (p_1, R_1) and (p_2, R_2) in S, the redex patterns of R_1 at p_1 and R_2 at p_2 are non-overlapping.*

Proof. The labels of p_1 and p_2 in t are not "`?`" since φ does not computes steps of "`?`". If $p_1 = p_2$, since $\mathcal{R}$ is a LOIS, it follows that $R_1 = R_2$ and contrary to the hypothesis the steps are not distinct. Thus, $p_1 \neq p_2$ and the limited inductive sequentiality of $\mathcal{R}$ ensures that distinct redexes are non-overlapping. □

Thus, informally, a computation of a term t according to φ executes a finite number of bubbling steps on t, that produce a new term that has exactly the same values of t, and/or a set of rewrite steps whose order of application is irrelevant. These conditions lead to the correctness of the strategy. Its soundness is a consequence of the soundness of bubbling [5]. Its completeness is a consequence of the completeness of *INS* [3]. These claims are proved in the next section.

5 Correctness

In this section we prove the correctness of our strategy. The main purpose of a strategy is to compute a subset of the steps that could be executed on a term so that all and only the values of the term are reached. A good strategy does not compute steps that do not help to reach any value of a term, although this should be achieved without look-ahead.

Computing only the values of a term is referred to as *soundness*. Strategies that compute a subset of all the steps of a term are obviously sound. Since we allow bubbling steps, the soundness of the strategy is not immediate. Computing all the values of a term is referred to as *completeness*. A good strategy should attempt to eliminate as many steps as possible. Proving that a strategy is complete is generally difficult, since if some steps are eliminated, some values might be lost.

In our context, namely constructor TRSs, a *value* of a term t is a constructor normal form derived from t. Since our strategy does not employ the rules of "`?`", formulating the statements of soundness and completeness require a certain amount of work. The following examples make this point.

Example 5.1 Using the standard functional logic notation for lists [18], consider the term $t =$ `[(0 ? 1) + 0]`. Our strategy reduces this term to $u =$ `[0 ? 1]`. It is easy to verify that there exist no derivation of t into u. However, both terms rewrite to either `[0]` or `[1]`, which are the values of t. The term u rewrites to these values using only the rules of "`?`".

Example 5.2 Consider the operation `loop` defined in (4) and the term $t =$ `loop ? 0`. It is immediate to verify that $t \to_\varphi t \to_\varphi t \to_\varphi \cdots$ and that `0` is the only value of t.

To deal with the problems highlighted by the previous examples, we formulate the correctness of the strategy as follows. Given a term t, a computation of t according to φ produces a term that does not prevent us from reaching any value of t (completeness) and, likewise, does not enables us to reach a term that would not be reachable from t (soundness). Since this is true for some "uninteresting strategies", e.g., the strategy that computes no steps, or the strategy that only reduces a term to itself, we also show that given "enough steps", the strategy computes a term u from which any value of t can be "extracted" simply by picking either alternative of every occurrence of a choice

in u.

Our strategy does not reduce terms rooted by "?". A term with nodes labeled by "?" represents a set of terms. Hence, the strategy computes a set of values so represented. Subsets of this set, including singletons, can be obtained using the following formalization.

Definition 5.3 [Extraction] Let t be an admissible term graph. The graph u is *extracted* from t, denoted by $u \in t$, if either of the following conditions holds:

- $u = t$;
- If c is a node of t labeled by "?" and x and y are the successors of c in t, then u is extracted from either $t[c \leftarrow t|_x]$ or $t[c \leftarrow t|_y]$.

Below, we present some simple properties of extraction.

Lemma 5.4 *Let $\mathcal{R}$ be a LOIS and t an admissible term graph over the signature of $\mathcal{R}$. For all terms u, $u \in t$ if and only there exists a derivation $t \xrightarrow{*} u$ that applies rules of "?" only.*

Proof. The "if" is by induction on the number of steps of $t \xrightarrow{*} u$. The "only if" is by induction on the number of node of t labeled by "?". □

Lemma 5.5 *The order in which nodes labeled by "?" are selected in the second case of Definition 5.3 does not affect the result.*

Proof. Using the equivalence between extraction and reduction of ?-rooted redexes, we show that the order in which ?-rooted redexes are replaced does not affect the result. Let t be a term graph, p and q distinct nodes of t labeled by "?", and R and R' rules of "?". If $t \to_{p,R} u$ and $t \to_{q,R'} v$, for some terms u and v, by case analysis of the relative positions of p and q, there exists a term w such that $u \to^{=}_{q,R'} w$ and $v \to^{=}_{p,R} w$. □

Lemma 5.6 *Let $\mathcal{R}$ be a LOIS and t an admissible term graph over the signature of $\mathcal{R}$. If, for some term u and nodes c and d, $t \simeq_{cd} u$, then, for all terms v, $v \in t$ if and only if $v \in u$, modulo a renaming of nodes.*

Proof. Let t' be the term obtained from t by replacing the subgraph at c with its left (resp. right) successor, and let u' be term obtained from u by replacing the subgraph at d with its left (resp. right) successor. By the definition of bubbling, $t' = u'$. Thus, if the same side is chosen at both c in t and at d in u, by Lemma 5.5 the claim follows. □

We are now ready to state and prove the soundness of our strategy. Informally, given a term t, any value that can be extracted from a term reached from t by our strategy can be reached from t by pure rewriting.

Theorem 5.7 (Soundness) *Let $\mathcal{R}$ be a LOIS and t an admissible term graph over the signature of $\mathcal{R}$. If $t \xrightarrow{*}_{\varphi} v$ and $u \in v$, where u is a constructor normal form, then $t \xrightarrow{*} u$, modulo a renaming of nodes.*

Proof. By induction on the length of the computation. Base case: $t = v$. By Lemma 5.4, $u \in v$ implies $v \xrightarrow{*} u$ and the claim immediately follows. Ind. case: $t \to_\varphi t' \xrightarrow{*}_\varphi v$. If $t \to_\varphi t'$ is not a bubbling step, then, by Definition 4.7, $t \xrightarrow{*} t'$. By the induction hypothesis, $t' \xrightarrow{*} u$, and, by the transitivity of $\xrightarrow{*}$, $t \xrightarrow{*} u$. If the step $t \to_\varphi t'$ is a bubbling step, by Theorem 4.8, every constructor normal form of t' is a constructor normal form of t. By the induction hypothesis, $t' \xrightarrow{*} u$, and consequently $t \xrightarrow{*} u$. □

We now turn to the completeness of our strategy. We need some preliminary results. Our first claim is similar to the *Parallel Moves Lemma*. The notion of *residual* of a rewrite step by another rewrite step is defined [19, Sect. 2] for orthogonal TRSs. We consider *LOIS* GRSs, which are not orthogonal, but have a very disciplined form of overlapping. Unless the two rules of "`?`" are applied at the same node, the usual definitions of residual of a step by a step, of a step by a derivation, and of a derivation by a derivation can be formulated without significant changes to *LOIS*. In particular, we make use of the following lemmas.

Lemma 5.8 (Parallel Rewriting Moves) *Let $\mathcal{R}$ be a LOIS and t an admissible term graph over the signature of $\mathcal{R}$. If $t \to_{p,R} u$ is a rewrite step and $t \to_{q,R'} v$ is a rewrite step where R' is not a rule of "`?`", then there exists a term w such that $u \xrightarrow{*} w$ is the residual of (q, R') by (p, R) and $v \xrightarrow{*} w$ is the residual of (p, R) by (q, R') modulo a renaming of nodes.*

Proof. Similar to [19, Lemma 2.2], the proof is by case analysis of the respective positions of p and q. □

Since computations by φ execute bubbling steps, to discuss commutative diagrams we need to consider the residual of a rewrite step by a bubbling step and vice versa. The conditions under which executing residual steps ensure the commutativity of a diagram are investigated in Lemma 5.10.

Definition 5.9 [Set of Residuals] Let S be a *LOIS* and t an admissible term graph over the signature of S. Let $t \simeq_{cd} t'$, for some graph t' and nodes c and d of t, and $t \to_{p,R} u$, for some node p of t and rule R of S.

- We define the set of *residuals* A of $t \to_{p,R} u$ by $t \simeq_{cd} t'$.

  Let l and r be the successors of c in t. By the definition of bubbling, $t' = t[d \leftarrow \Theta_l(t|_d[c \leftarrow l])\,?\,\Theta_r(t|_d[c \leftarrow r])]$, where Θ_l and Θ_r are partial renamings with the same domain and disjoint images.

$$A = \begin{cases} \{(d, R)\} & \text{if } p = c; \\ \{(p, R)\} & \text{if } p \text{ is not in the domain of } \Theta_l; \\ \{(\Theta_l(p), R), (\Theta_r(p), R)\} & \text{otherwise} \end{cases}$$

- We define the set of *residuals* B of $t \simeq_{cd} t'$ by $t \to_{p,R} u$ as follows.

$$B = \begin{cases} \varnothing & \text{if } p = c; \\ \{(c, \simeq_{cd})\} & \text{otherwise.} \end{cases}$$

Lemma 5.10 (Parallel Bubbling Moves) *Let S be a* LOIS *and t an admissible term graph over the signature of S. If $t \simeq_{cd} t'$, for some graph t' and nodes c and d of t, and $t \to_{p,R} u$, for some node p of t and rule R of S, and d is not in the redex pattern of R at p in t, then there exists u' such that $t' \xrightarrow{+} u'$ is the application of the residual of $t \to_{p,R} u$ by $t \simeq_{cd} t'$ and $u \simeq^{=}_{cd} u'$ is the application of the residual of $t \simeq_{cd} t'$ by $t \to_{p,R} u$ modulo a renaming of nodes.*

Proof. [5, Lemma 3]. □

We now discuss a relatively simple, but essential claim for the completeness of our strategy. The claim is about the difference between φ and *INS* [3]. Our proof is less rigorous than the other proofs of this paper, because comparing φ and *INS* is not straightforward. Although φ and *INS* are very similar—in fact, we regard φ as an evolution of *INS*—they are formulated for different frameworks. *INS* is a *narrowing* strategy for *term* rewriting systems, whereas φ is a *rewriting* strategy for *graph* rewriting systems. In order to compare the two strategies, we would need to restrict *INS* to rewriting, which is trivial, and re-phrase it for graph rewriting, which is much more laborious. A rigorous treatment of this situation would take us well beyond the scope of this paper. Fortunately, significant portions of the required theory are already available. We will only informally fill the gaps.

Echahed and Janodet [13] extend many results of *needed narrowing* [6] to term *graph* rewriting. *INS* extends needed narrowing by adding one dimension of non-determinism—the choice of one of the possibly many rules with the same left-hand side. This aspect of *INS* is independent of the differences between terms and graphs. Thus, the treatment in [13] provides the core of the missing theory.

The description and formalization of computations in the framework of term rewriting differ from that of graph rewriting. The former uses positions and substitutions, whereas the latter uses nodes and homomorphisms, respectively. Converting from one framework to the other is mostly a syntactic undertaking. For example, sets of positions are used in *standard term graphs*[24, Def. 3.3] to uniquely identify nodes without isomorphisms. Apart from these syntactic variations, there are only two significant differences between φ and *INS*. (1) *INS* does not compute bubbling steps whereas φ does. If a node c labeled by "`?`" matches an inductive position of a definitional tree used in the computation of a term t, then *INS* reduces the redex at c. By contrast, φ either bubbles the node or recursively attempts to independently reduce *both* successors of c. (2) *INS* is inherently a non-deterministic strategy, because some redexes have distinct replacements. *INS* computes a set of steps on a

term t, but only *one* step in this set is non-deterministically selected and applied to t. By contrast, φ rewrites deterministically. (Although the choice of a bubbling step is non-deterministic, by [5, Lemma 3] the non-determinism is *don't care*.) Similar to *INS*, φ computes a set of steps on a term t. If this set contains only rewrite steps, then *all* the steps in the set are simultaneously applied to t.

Both *INS* and φ use definitional trees to compute steps. Some operations have definitional trees that are not isomorphic, e.g., see [2, Sect.7]. For these strategies, the choice of a particular tree does not affect the computation of constructor normal forms. However, it may affect the order of some steps of a computation. Therefore, in comparing the behavior of the strategies, we will fix one tree of any operation, the same for both strategies, and use that tree in our reasoning.

Lemma 5.11 (Inclusion) *Let $\mathcal{R}$ be a LOIS and t an admissible term graph over the signature of $\mathcal{R}$. If* INS *computes a step $t \rightarrow_{p,R} u$ and R is not a rule of "?", then $(p, R) \in \varphi(t)$ when a uniquely chosen definitional tree is used by both strategies for each operation symbol.*

Proof. With the same choice of definitional trees, *INS* and φ go through the same cases except for the rules of "?". If *INS* computes a step that does not apply a rule of "?", then the same step is computed by φ. □

Lemma 5.12 (Persistence by extraction) *Let $\mathcal{R}$ be a LOIS and t an admissible term graph over the signature of $\mathcal{R}$. If $u \in t$ and $p \in \mathcal{N}_u$, then $(p, R) \in \varphi(t)$ if and only if $(p, R) \in \varphi(u)$.*

Proof. By Definition 4.6, if $t = v_l \,?\, v_r$, then $\varphi(t) = \varphi(v_l) \cup \varphi(v_r)$. The claim follows by induction on the number of nodes labeled by "?" in t. □

Lemma 5.13 (Persistence by bubbling) *Let $\mathcal{R}$ be a LOIS and t an admissible term graph over the signature of $\mathcal{R}$. If $(p, R) \in \varphi(t)$, where R is not a rule of "?", and $t \simeq_{cd} u$, then there exists some (q, R) in the set of residuals of (p, R) by $t \simeq_{cd} u$ such that $(q, R) \in \varphi(u)$.*

Proof. Let $P_t = n_0 n_1 \ldots n_k$ be a path from the root of t to p that in the computation of $\varphi(t)$ produced (p, R). By the definition of bubbling, there exist a path $P_u = m_0 m_1 \ldots m_l$ in u such that the sequence of labels in P_t and P_u is the same except for the possible insertion of a single "?" and the possible removal of a single "?". Every node in P_u except for those corresponding to the possible insertion and removal of "?", is the renaming of a corresponding node of P_t. In particular, m_0 is a renaming of n_0, m_l is a renaming of n_k, a node labeled by "?" is inserted in P_u w.r.t. P_t if and only if P_t contains d, and a node labeled by "?" is removed from P_u w.r.t. P_t if and only if P_t contains c. By the definition of φ, the computation of a step at node p depends only on the labels of a path ending at p. Moreover, if the step is a rewriting step, inserting into or removing from the path nodes labeled by "?" is irrelevant by the third

case of the definition of φ. Thus, if $(p, R) \in \varphi(t)$, then $(q, R) \in \varphi(u)$. □

Theorem 5.14 (Completeness) *Let $\mathcal{R}$ be a LOIS and t an admissible term graph over the signature of $\mathcal{R}$. If $t \xrightarrow{*} u$, where u is constructor normal form, and $t = t_0 \rightarrow_\varphi t_1 \rightarrow_\varphi \cdots$ is computed by φ, then for some n, $u \in t_n$ modulo a renaming of nodes.*

Proof. The proof is in two parts. First we prove the existence of a certain diagram, then we use the diagram to prove the theorem's claim. The definition of the diagram and the proof of its commutativity are by induction on the structure of the diagram. In the diagram, a vertical arrow represents either a step calculated by φ or the residual of one such step by horizontal arrows. Likewise, a horizontal arrow represents either a step calculated by *INS* or the residual of one such step by vertical arrows.

(8)
$$\begin{array}{ccccccc}
t_{00} & \longrightarrow & t_{01} & \longrightarrow & \cdots & \longrightarrow & t_{0k} \\
\downarrow & & \downarrow & & & & \downarrow \\
t_{10} & \longrightarrow & t_{11} & \longrightarrow & \cdots & \longrightarrow & t_{1k} \\
\downarrow & & \downarrow & & & & \downarrow \\
\vdots & & \vdots & & \ddots & & \vdots \\
\downarrow & & \downarrow & & & & \downarrow \\
t_{n0} & \longrightarrow & t_{n1} & \longrightarrow & \cdots & \longrightarrow & t_{nk}
\end{array}$$

The base case consists of several items that define the top row and left column of the diagram. We let $t_{00} = t$. By hypothesis $t \xrightarrow{*} u$. Since *INS* is complete [3, Corollary 31], there exists an *INS* derivation of t into u. Thus, we let $t_{0j} \rightarrow t_{0(j+1)}$ be a step computed by *INS* such that $t_{0k} = u$. Finally, we let $t_{i0} \rightarrow t_{(i+1)0}$ be a step, either bubbling or rewriting, computed by φ. The inductive case defines $t_{(i+1)(j+1)}$ and the steps into it from the neighboring terms in the diagram. The induction hypothesis ensures that these terms and the steps into them, if any, are defined. The situation that we consider is represented by the next diagram:

(9)
$$\begin{array}{ccc}
t_{ij} & \xrightarrow{A_i} & t_{i(j+1)} \\
{\scriptstyle B_j}\downarrow & & \downarrow{\scriptstyle B_{j+1}} \\
t_{(i+1)j} & \xrightarrow[A_{i+1}]{} & t_{(i+1)(j+1)}
\end{array}$$

We consider two cases. (1) If B_j is not a bubbling step, then it satisfies the condition of Lemma 5.8 which ensures the existence of $t_{(i+1)(j+1)}$ and the commutativity of the diagram. (2) If B_j is a bubbling step, then the destination of this step cannot be inside a redex pattern [5, Def. 7], since φ is outermost. Thus, [5, Lemma 3] ensures the existence of $t_{(i+1)(j+1)}$ and the commutativity of the diagram. Since in both cases Diagram (9) commutes, by induction the whole Diagram (8) commutes.

Now, we define an integer function χ on the horizontal arrows of Diagram (8) and then we overload it for entire rows. A horizontal arrow A is either an *INS* step or the residual of the step above it in the diagram. Obviously, a residual can be an equality (the null derivation). We define $\chi(A) = 0$ if A applies rules of "`?`" or if it is an equality. We define $\chi(A) = 1$ in all other cases. Intuitively, χ is the *cost* of moving in a given row from one column to the next in the diagram. We place no cost on rewrites that apply rules of "`?`" because this fits our purposes. The rows of the diagram are identified by a natural number. We define $\chi(i)$, the cost of the i-th row, as the sum of the costs of all the row's arrows, i.e., $\Sigma_{j=0}^{k-1}\chi(t_{ij} \to t_{i(j+1)})$. It follows from the definitions of χ and of residual that the cost of a step cannot be greater than the cost of the step directly above it, and consequently that the cost of a row cannot be greater than the cost of any preceding row.

Finally, we prove the theorem's claim by Noetherian induction on the cost of the rows. Preliminarily, observe that by assumption $t_{0k} = u$ and u is a constructor normal form. Since the diagram commutes, for all i, $t_{ik} = u$. Base case: $\chi(0) = 0$. Only rules of "`?`" are applied to t to derive u, hence $u \in t$ by Lemma 5.4 and the claim trivially holds. Ind. case: $\chi(0) = m$ for some $m > 0$. Since $\chi(0) > 0$, there exists a smallest index j such that $\chi(t_{0j} \to_{p,R} t_{0(j+1)}) = 1$, where $(p, R) \in \mathit{INS}(t_{0j})$. By Lemma 4.9, there exists a smallest index i such that $t_{i0} \to t_{(i+1)0}$ is not a bubbling step. We show that $t_{(i+1)j} = t_{(i+1)(j+1)}$, and so $\chi(t_{(i+1)j} \to t_{(i+1)(j+1)}) = 0$ and consequently $\chi(i + 1) < \chi(0)$.

Let S be the set of residuals of (p, R) by $t_{0j} \xrightarrow{*} t_{ij}$. If S is empty, then the claim is proved. Otherwise, there exists a node p' of t_{ij} such that (p', R) is in the set of residuals of (p, R) in S. By Lemma 5.11, $(p, R) \in \varphi(t_{0j})$. By Lemma 5.12, $(p, R) \in \varphi(t_{00})$. By Lemma 5.13, the set of residuals of (p, R) in t_{i0} by $t_{00} \xrightarrow{*} t_{i0}$ includes some (q, R) such that, by the commutativity of the diagram, the set of residuals of (q, R) in t_{ij} by $t_{i0} \xrightarrow{*} t_{ij}$ includes (p', R). Thus, the step (p', R) is executed by both the horizontal and the vertical arrows originating from t_{ij}. Consequently, the set of residuals of (p', R) in $t_{(i+1)j}$ is empty, the cost of the horizontal arrow from $t_{(i+1)j}$ is zero, and $\chi(i + 1) < \chi(0)$. Since the cost of the rows cannot decrease forever, there exists an n such that $\chi(n) = 0$. By the definition of χ, this implies that $t_{n0} \xrightarrow{*} u$ where any replacement applies a rule of "`?`". By Lemma 5.4, $u \in t_{n0}$. □

6 Narrowing

The strategy of Section 4 computes rewriting steps. Many of the results and ideas that we have presented can be used for extending the strategy to narrowing. The correctness of bubbling is independent of whether a bubbling step is performed in a rewriting or narrowing computation.

Narrowing is inherently non-deterministic and therefore naturally expressed using the choice operation [7]. Many data types have several data constructors, hence there are many possible instantiations of an unbound variable in

a narrowing step. For example, consider the definition of the operation `not` discussed earlier:

```
data Boolean = True | False
not True = False                                    (10)
not False = True
```

To narrow `not x`, where `x` is a free variable, we bind `x` to `True ? False`, since `True` and `False` are the patterns in the definition of `not`, and we rewrite `not (True ? False)` using our strategy. The binding of a variable is obtained, as for many other narrowing strategies [4], using definitional trees.

Terms with nodes labeled by "`?`" abstract *sets* of terms that in an intuitive sense are more deterministic. For example, the term `True ? False` abstracts the set {`True`, `False`}. Definition 5.3 formalizes this intuitive notion. A variable x within a term t with a node labeled by "`?`" may belong to two distinct terms, say u and v, in the set abstracted by t. Before instantiating x in a narrowing step of t, x must be "standardized apart" in u and v. The transformation that standardizes apart a variable in a graph is very similar to a bubbling step.

7 Related work

Although strategies for functional logic computations [4] and term graph rewriting [24] have been intensely investigated, the work on strategies for term graph rewriting systems as models of functional logic programs has been relatively scarce; for instance, the results of [10] pertain to a different class of GRSs. The line of work closest to ours is [13,14]. 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 some steps by limiting the cloning of the context of a redex with a non-deterministic replacement is original. The results of [5] complement those of this paper by describing how bubbling and rewriting steps interact with each others.

Other efforts on handling non-determinism in functional and functional logic computations with shared subexpressions include [20], which introduces the *call-time choice semantics* to ensure that shared terms are evaluated to the same result; [15], which defines a rewriting logic that among other properties provides the call-time choice; and [1] and [25], 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.

8 Conclusion

This paper defines a strategy well suited for the execution of functional logic programming languages. Programs in these languages execute non-deterministic steps on shared terms. Our strategy has several distinctive and highly

desirable features. Of course, it is sound and complete. Although it is intended for non-deterministic computations, its steps are deterministic (the non-determinism in the choice of a bubbling step is "*don't care*"). Since the steps of our strategy are deterministic, its implementation is not required to copy the context of a non-deterministic redex. Since a non-deterministic computation generally leads to some failures, our strategy has the potential of improving the performance of functional logic programs by avoiding cloning the entire contexts of some redexes. Future work will implement the strategy and measure its performance for realistic programs.

Acknowledgement

Sunita Marathe offered valuable corrections to a draft of this paper.

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. Definitional trees. In H. Kirchner and G. Levi, editors, *Proceedings of the Third International Conference on Algebraic and Logic Programming*, pages 143–157, Volterra, Italy, September 1992. Springer LNCS 632.

[3] 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.

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

[5] S. Antoy, D. Brown, and S. Chiang. On the correctness of bubbling. In F. Pfenning, editor, *17th International Conference on Rewriting Techniques and Applications (RTA'06)*. Springer, 2006.

[6] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, July 2000.

[7] S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Twenty Second International Conference on Logic Programming*, pages 87–101, Seattle, WA, Aug. 2006. Springer LNCS 4079.

[8] 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.

[9] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[10] H. P. Barendregt, M. van Eekelen, J. Glauert, J. Kennaway, M. Plasmeijer, and M. Sleep. Term graph rewriting. In *PARLE Parallel Architectures and Languages Europe*, volume LNCS 259, pages 141–158, Eindhoven, 1987. Springer-Verlag. also technical report SYS-C87-01.

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

[12] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, 1990.

[13] R. Echahed and J.-C. Janodet. On constructor-based graph rewriting systems. Research Report 985-I, IMAG, 1997. Available at http://citeseer.ist.psu.edu/echahed97constructorbased.html.

[14] 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.

[15] 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.

[16] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[17] M. Hanus (ed.). PAKCS 1.7.1: The Portland Aachen Kiel Curry System. Available at http://www.informatik.uni-kiel.de/~pakcs, March 27, 2006.

[18] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at http://www.informatik.uni-kiel.de/~curry, March 28, 2006.

[19] G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*. MIT Press, Cambridge, MA, 1991.

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

[21] J. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

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

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

[24] 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.

[25] 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.