17th International Conference on Rewriting Techniques and Applications (RTA'06)
Seattle, WA, USA, August 12–14, 2006

On the Correctness of Bubbling*

Sergio Antoy Daniel W. Brown Su-Hui Chiang

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

Abstract. Bubbling, a recently introduced graph transformation for functional logic computations, is well-suited for the reduction of redexes with distinct replacements. Unlike backtracking, bubbling preserves operational completeness; unlike copying, it avoids the up-front construction of large contexts of redexes, an expensive and frequently wasteful operation. We recall the notion of bubbling and offer the first proof of its completeness and soundness with respect to rewriting.

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 color a map of the Pacific Northwest so that no pair of adjacent states shares a color. The following declarations, in Curry [15], define the well-known topology of the problem:

```
data State = WA | OR | ID | BC
states = [WA,OR,ID,BC]                                              (1)
adjacent = [(WA,OR),(WA,ID),(WA,BC),(OR,ID),(ID,BC)]
```

The colors to be used for coloring the states and a non-deterministic operation, `paint`, to pair its argument to a color are defined below. The library operation "`?`" non-deterministically selects either of its arguments.

```
data Color = Red | Green | Blue
paint x = (x, Red ? Green ? Blue)                                   (2)
```

The rest of the program follows:

```
solve | all diffColor adjacent = theMap
      where theMap = map paint states
            diffColor (x,y) = colorOf x /= colorOf y
            lookup ((s,c):t) x = if s==x then c                     (3)
                                 else lookup t x
            colorOf = lookup theMap
```

* Partially supported by the NSF grant CCR-0218224.

The evaluation of `solve` solves the problem. In particular, `theMap` associates a color to each state and so represents the map, `diffColor` tells whether the colors associated to two states are different, `lookup` looks up the color associated to a state in the map, `all` and `map` are well-known library functions for list manipulation, and the condition of `solve` ensures that no adjacent states have been assigned the same color.

Non-determinism reduces the effort of designing and implementing data structures and algorithms to encode this problem into a program. The simplicity of the non-deterministic solution inspires confidence in the program's correctness. The implementation of non-deterministic functional logic programs has not been studied as extensively as that of deterministic programs.

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 a relation on graphs that is at the core of our approach. Section 5 proves the correctness of our approach. Section 6 briefly addresses related work. Section 7 offers our conclusion.

2 Motivation

We regard a functional logic program as a term rewriting system (TRS) [8–10, 18] or a graph rewriting system (GRS) [11, 21] with the constructor discipline [20]. Source-level constructs such as data declarations, currying, higher-order and anonymous functions, partial application, nested scopes, etc. can be transformed by a compilation process into ordinary rewrite rules [15]. 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.

The instantiation of free variables in narrowing steps does not play any specific role in our discussion as well as in the program we presented in the introduction. In this paper, we are mostly concerned with rewriting. For many problems in this area, extending results from rewriting to narrowing requires only a moderate effort. We will sketch the extension of our work to narrowing in the final section.

Our focus is on the interaction of non-determinism and sharing. In a deterministic system, evaluating a shared subexpression twice is merely inefficient; in a non-deterministic system, it can lead to unsoundness. For instance, in the map coloring example, the value of `theMap` is any possible association of a color to a state. In the program, there are two occurrences of `theMap`, besides its definition. One occurrence is returned as the output of the program; the other is constrained to be a correct solution of the problem. Obviously, if the values of these occurrences were not the same, the output of the program would likely be wrong.

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 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 $t[x]$ succeeds, the interpreter may give the user the option of evaluating $t[y]$.

With *copying*, the computation of $t[u]$ consists of 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 [14] and TOY [19] are based on backtracking, whereas the FLVM [7] and the interpreter of Tolmach et al. [22] 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 even though t has a value, since the evaluation of `f loop` does not terminate. This is a well-known problem of backtracking referred to as the loss of *completeness*.

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, is a division by zero, and consequently the computation fails.

Thus, copying may needlessly construct terms and backtracking may fail to produce results. A recently proposed approach [5], called *bubbling*, avoids these drawbacks. The idea is to slowly "move" a choice up its context and evaluate both its arguments. Bubbling is a compromise between evaluating only one non-deterministic choice, as in backtracking, and duplicating the entire context of each non-deterministic choice, as in copying. Bubbling is free from the drawbacks of backtracking and copying discussed earlier.

The evaluation of `f (loop ? 1)` by bubbling produces `(f loop) ? (f 1)`. Contrary to backtracking, no unrecoverable choice is made in this step. Both argu-

ments of "?" can be evaluated concurrently, e.g., as in [5]. The evaluation of the first argument does not terminate; however, this does not prevent obtaining the value of the second argument.

Likewise, the evaluation of `f (0 ? 1)` goes (roughly) through the following intermediate terms, where `fail` is a distinguished symbol denoting any expression that cannot be evaluated to a constructor term:

$$\begin{aligned} &\texttt{f (0 ? 1)} \\ &\quad \to \texttt{1+(2+(...+(}n\ \texttt{`div`\ (0 ? 1))...))} \\ &\quad \to \texttt{1+(2+(...+((}n\ \texttt{`div`\ 0) ? (}n\ \texttt{`div`\ 1))...))} \\ &\quad \to \texttt{1+(2+(...+(fail ? (}n\ \texttt{`div`\ 1))...))} \\ &\quad \to \texttt{1+(2+(...+(}n\ \texttt{`div`\ 1)...))} \end{aligned} \tag{5}$$

The `fail` alternative is dropped. Since `fail` occurs at a position where a constructor-rooted term is needed, it cannot lead to a successful computation.

In (5), the obvious advantages of bubbling 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 $f(x\,\texttt{?}\,y) = f(x)\,\texttt{?}\,f(y)$ 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 [17]. Informally, the non-deterministic choice for the argument of `f` is made at the time of `f`'s application. Therefore, the instances of `x` in the right-hand side of (6) should all evaluate to `True` or all to `False`. With an eager strategy, the call-time choice is automatic, and the only available option. With a lazy strategy, the call-time choice is relatively easy to implement by "sharing" the occurrences of `x`. That is, there is only one occurrence of the term bound to `x`. All the occurrences of `x` refer to this term. The term being evaluated is the *graph* depicted in the left-hand side of the following figure:

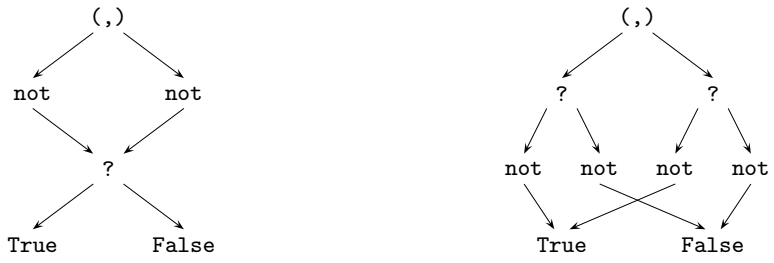

Fig. 1. The left-hand side depicts a term graph. The right-hand side is obtained from the left-hand side by bubbling up to the parents the non-deterministic choice. The two term graphs have a different set of constructor normal forms.

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 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. This formalization is the foundation of a recently discovered strategy [5] that computes both rewriting and bubbling steps.

3 Background

TRSs have been used extensively to model FL programs. This modeling has been very successful for some problems, e.g., the discovery of efficient narrowing strategies and the study of their properties; see [4] for a survey. However, a TRS only approximates a FL program, because it does not adequately capture the sharing of subexpressions in an expression. As we discussed in the previous section, and our introductory example shows, sharing is an essential semantic component of the execution of a non-deterministic program.

GRSs [11, 21] model FL programs more accurately than do TRSs. Unfortunately, they are also more complex than TRSs, and non-trivial variations exist in their formalization. In this paper, we follow the systemization of Echahed and Janodet [11] because the class of GRSs that they consider is a good fit for our programs. The space alloted to this paper prevents us from recalling relevant definitions and results of [11]. Luckily, this paper is easily accessible on-line at `http://citeseer.ist.psu.edu/echahed97constructorbased.html`.

In this paper, we assume that programs are *overlapping inductively sequential* [4, 2] term graph rewriting systems, abbreviated GRSs, and computations are rewriting sequences of admissible term graphs. We recall that a graph is *admissible* [11, Def. 18] if none of its defined operations belongs to a cycle.

Our choice of programs is motivated by the expressiveness of this class (e.g., as shown by the introductory example), by the existence of a strategy that performs only steps that are needed modulo a non-deterministic choice [2], and by the fact that computations for the entire constructor based programs can be implemented by this class via a transformation [3]. Non-deterministic computations in this class are supported by the single operation defined below.

Definition 1 [Choice operation] The *choice operation*, denoted by the infix operator "`?`", is defined by the following rules:

```
x ? y = x
x ? y = y
```
□

We assume that this is the only overlapping operation of a GRS. Any other overlapping can be eliminated, without changing the meaning of a program, using the choice operation, as discussed in [2] and shown in our introductory example.

Definition 2 [Limited overlapping] A *limited overlapping* inductively sequential GRS, abbreviated *LOIS*, is a constructor based GRS, S, such that the signature of S contains the choice operation "`?`" presented in Def. 1 and every other defined operation of S is inductively sequential. □

We need an additional definition, which is crucial to our approach.

Definition 3 [Dominance] A node d *dominates* a node n in a rooted graph t if every path from the root of t to n contains d. If d and n are distinct, then d *properly dominates* n in t. □

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.

Echahed and Janodet [11] formalize rewriting, including an efficient strategy, for the inductively sequential term graph rewriting systems. This class is similar to ours, except for the presence of the choice operation. Following their lead, we always use "fresh" rules in rewrite steps. This is justified by the following example:

```
t = (ind, ind)
ind = coin                                    (7)
coin = 0 ? 1
```

The intended semantics is that each occurrence of `ind` in `t` is evaluated independently of the other (`ind` is not a variable) and therefore `t` has four values, every pair in which each component is either `0` or `1`. To compute all the intended values of `t`, it is imperative that a rewrite step uses a *variant* [11, Def. 19] of a rewrite rule, namely a clone of the rule with fresh nodes (and variables). A consequence of using variants of rules is that the equality of graphs resulting from rewrite steps can be assessed only modulo a renaming of their nodes [11, Def. 15].

4 Bubbling

Computations that perform non-deterministic steps must preserve in some form the context of a redex when the redex has distinct replacements. Typically, some portions of the context are reconstructed, as in backtracking, or are duplicated, as in copying. An overall goal of bubbling is to limit these activities. In the following, we precisely define which portions of a context of a redex are cloned in our approach.

Definition 4 [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:

$$\Theta(n) = \begin{cases} n' & \text{where } n' \in \mathcal{N}_q, \text{ if } n \in \mathcal{N}_p; \\ n & \text{otherwise.} \end{cases} \tag{8}$$

Similar to 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:

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

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, the cardinalities of the domain and image of a partial renaming are the same.

Lemma 1. *If g is a graph and g' is a partial renaming of g with respect to some $\mathcal{N}_p$ and $\mathcal{N}_q$, then g and g' are compatible.*

Proof. Immediate from the notion of compatibility [11, Def. 6] and the construction of g' in Definition 4. □

The evaluation of an admissible term graph t_0 in a GRS S is a sequence of graphs $t_0 \rightsquigarrow t_1 \rightsquigarrow t_2 \cdots$ where for every natural number i, t_{i+1} is obtained from t_i either with a rewrite step of S or with a *bubbling* step, which is defined below.

Definition 5 [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., $\mathcal{N}_p = \{n \mid n_1 n_2 \dots n_k \in \mathcal{P}_g(d, c) \text{ and } n = n_i \text{ for some } i\}$, 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 obviously 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. In executing 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. In this example, the dominator is the root of the graph, but in general the destination node can be any proper dominator of the origin. In practice, it is convenient to bubble a choice only to produce a redex. The strategy introduced in [5] ensures this desirable property.

The bubbling relation entails 3 graph replacements. By Lemma 1, the graphs involved in these replacements are all compatible with each other. Therefore, the bubbling relation is well defined according to [11, Def. 9]. In particular, except for the nodes being renamed, g_x and g_y can share nodes between themselves and/or with g. Any sharing among these (sub)graphs is preserved by bubbling.

Two adjacent bubbling steps can be composed into a "bigger" step.

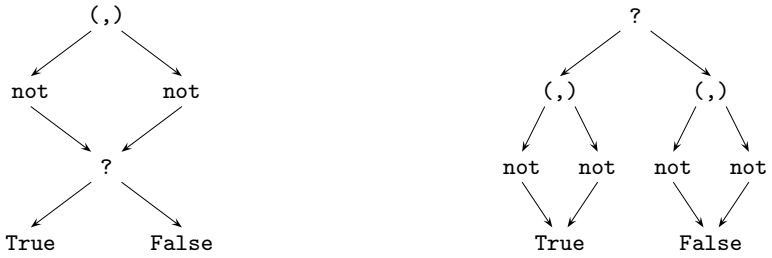

Fig. 2. The left-hand side depicts a term graph. The right-hand side is obtained from the left-hand side by bubbling the non-deterministic choice up to a proper dominator. The two term graphs have the same set of constructor normal forms.

Theorem 1 (Transitivity of bubbling). *Let S be a GRS. For all term graphs t, u and v over the signature of S and for all c and d nodes of t and d and e nodes of u, modulo a renaming of nodes, if $t \simeq_{cd} u$ and $u \simeq_{de} v$ then e is a node of t and $t \simeq_{ce} v$.*

Proof. If c is a node labeled by a choice operation, c_l and c_r denotes the left and right successors of c. Let w be defined by $t \simeq_{ce} w$ and consider the expressions defining u, v and w:

$$\begin{array}{l} u = t[d \leftarrow (\Theta_{dc_l}(t|_d[c \leftarrow t|_{c_l}]) \,?\, \Theta_{dc_r}(t|_d[c \leftarrow t|_{c_r}]))] \\ v = u[e \leftarrow (\Theta_{ed_l}(u|_e[d \leftarrow u|_{d_l}]) \,?\, \Theta_{ed_r}(u|_e[d \leftarrow u|_{d_r}]))] \\ w = t[e \leftarrow (\Theta_{ec_l}(t|_e[c \leftarrow t|_{c_l}]) \,?\, \Theta_{ec_r}(t|_e[c \leftarrow t|_{c_r}]))] \end{array} \tag{9}$$

where Θ_{xy} is a renaming whose domain is the set of the nodes in any path between x and y. Also, we assume that the images of all renamings are disjoint.

We prove that $v = w$ modulo a renaming of nodes. The portion of u at and above e is the same as in t. Using this condition *twice*, we only have to prove $\Theta_{ed_l}(t|_e[d \leftarrow u|_{d_l}]) = \Theta_{ec_l}(t|_e[c \leftarrow t|_{c_l}])$ and the analogous equation for the right-hand side argument. By construction, $u|_{d_l} = \Theta_{dc_l}(t|_d[c \leftarrow t|_{c_l}])$. Thus, $\Theta_{ed_l}(t|_e[d \leftarrow u|_{d_l}]) = \Theta_{ed_l}(t|_e[d \leftarrow \Theta_{dc_l}(t|_d[c \leftarrow t|_{c_l}])])$. Since no node is duplicated by renamings, we have that $\Theta_{ed_l}(t|_e[d \leftarrow \Theta_{dc_l}(t|_d[c \leftarrow t|_{c_l}])]) = \Theta_{ed_l} \circ \Theta_{dc_l}(t|_e[d \leftarrow t|_d[c \leftarrow t|_{c_l}]])$. Since $t|_d$ is modified only at c and c is below d, $t|_e[d \leftarrow t|_d[c \leftarrow t|_{c_l}]] = t|_e[c \leftarrow t|_{c_l}]$. Thus, by equational reasoning, $v = w$ except for the renamings of nodes, and the claim holds. □

Bubbling creates a natural mapping between two graphs. If $t \simeq u$, then every node of u "comes" from a node of t. This mapping, which is instrumental in proving some of our claim, is formalized below.

Definition 6 [Natural mapping] Let S be a *GRS*, t a graph over the signature of S and $t \simeq_{cd} u$, for some graph u and nodes c and d of t. We call *natural* the mapping $\mu : \mathcal{N}_u \to \mathcal{N}_t$ defined as follows. By construction, $u = t[d \leftarrow t']$, for some term graph t'. Let d' be the root node of t'. The construction of u involves

two renamings in the sense of Def. 4; let us call them Θ_x and Θ_y. We define μ on n, a node of u, as follows:

$$\mu(n) = \begin{cases} c & \text{if } n = d'; \\ \Theta_x^{-1}(n) & \text{if } n \text{ is in the image of } \Theta_x; \\ \Theta_y^{-1}(n) & \text{if } n \text{ is in the image of } \Theta_y; \\ n & \text{otherwise.} \end{cases} \tag{10}$$

Observe that the images of Θ_x and Θ_y are disjoint, hence the second and third cases of (10) are mutually exclusive. □

The next lemma shows that a rule of "?" applied before a bubbling step at the origin or after a bubbling step at the destination produces the same outcome.

Lemma 2 (Same rule). *Let S be a* LOIS *and t an admissible term graph over the signature of S. If $t \simeq_{cd} u$, $t \to_{c,R} v$ and $u \to_{d,R} w$, then $v = w$ modulo a renaming of nodes.*

Proof. R is a rule of "?". Without loss of generality, we assume that it is the rule that selects the left argument. By assumption the subgraph of t at c is of the form $x \,?\, y$. Hence $t = t[c \leftarrow x \,?\, y]$ and $t \to_{c,R} v = t[c \leftarrow x]$. By definition of bubbling, $u = t[d \leftarrow (\Theta_x(t|_d[c \leftarrow x]) \,?\, \Theta_y(t|_d[c \leftarrow y]))]$, for some renamings Θ_x and Θ_y. Therefore $u \to_{d,R} w = t[d \leftarrow \Theta_x(t|_d[c \leftarrow x])] = \Theta_x(t[d \leftarrow t|_d[c \leftarrow x]]) = \Theta_x(t[c \leftarrow x])$. □

5 Correctness

In this section we state and prove the correctness of our approach. The notion of a *redex pattern* defines the set of nodes below a node n labeled by an operation f that determines that a rule of f can be applied at n. Recall that a matcher is a function that maps the nodes of one graph to those of another, preserving the labeling and successor functions.

Definition 7 [Redex pattern] Let t be a graph, $l \to r$ a rewrite rule, and n a node of t such that l matches t at n with matcher h. We call *redex pattern* of $l \to r$ in t at n the set of nodes of t that are images according to h of a node of l with a constructor label. □

We are convening that a node n is not in any redex pattern at n. This is just a convenient convention.

The following example shows that some pairs of bubbling and rewriting steps do not commute. This is a significant condition that prevents proof techniques based on *parallel moves* [16]. Although our GRSs are not orthogonal, some form of parallel moves is available for *LOIS* [2]. Consider the term $t =$ `snd(1,2?3)`, where `snd` is the function that returns the second component of a pair and, obviously, there is no sharing. Bubbling the choice to its parent (see Figure 3) produces $u =$ `snd((1,2)?(1,3))`. The term u cannot be obtained from t by

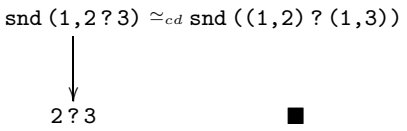

Fig. 3. Bubbling and rewriting do not always commute. No *parallel moves* are available for this diagram. Note that the term on the right cannot be reached from the original term by rewriting.

rewriting. Furthermore, the redex at the root of t has been destroyed by the bubbling step. The following result offers a sufficient condition, namely an appropriate choice of the destination of a bubbling step, for recovering the commutativity of bubbling and rewriting. As customary, for any relation R, $R^=$ denotes the reflexive closure of R.

Lemma 3 (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 $t' \xrightarrow{+} u'$ and $u \simeq^{=}_{cd} u'$ modulo a renaming of nodes.*

Proof. Let $P = \mu^{-1}(p)$ be the set of nodes of t' that map to p in t. Observe that P contains either 1 or 2 nodes. We show that R can be applied to any node in P and we define u' as the result of applying R to all the nodes of P. If $p = c$, then by definition $P = \{d\}$. In this case, R is a rule of "?" and consequently $u = u'$ (modulo a renaming of nodes) and the claim immediately holds. If $p \neq c$ and $P = \{p\}$, then the reduction in t is independent of the bubbling step. The redex pattern of R at p in t is either entirely below c, since the label of c is an operation, or entirely above d, since by hypothesis d is not in the redex pattern of R at p in t. The redex pattern of R at p is the same in t and t' and the redex is equally replaced in t and t'. Hence $u \simeq_{cd} u'$. If $p \neq c$ and $P = \{p_1, p_2\}$, with $p_1 \neq p_2$, then the reduction in t is in the portion of t cloned by the bubbling step. The redex pattern of R at p in t is entirely contained in this portion. The redex pattern is entirely below d by hypothesis, and cannot include c since the the label of c is an operation. Thus the redex pattern is entirely cloned in two disjoint occurrences in t'. By reducing both occurrences, in whatever order, t' reduces to u' in two steps and $u \simeq_{cd} u'$. □

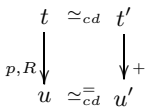

Fig. 4. Graphical representation of Lemma 3. If the destination of the bubbling step of t is not in the redex pattern of the rewrite step of t, then, for a suitable graph u', the diagram commutes.

Definition 8 [Combined step] We denote with "$\rightsquigarrow$", called a *combined step*, the union of the bubbling and rewriting relations in a *LOIS*, i.e., $\rightsquigarrow \; = \simeq \cup \to$. □

We now address the completeness of the combined step relation. Since this relation is an extension of the rewrite relation, a traditional proof of completeness would be trivial. Instead, we prove a more interesting claim, namely, no result of a computation is lost by the execution of bubbling steps. Therefore, an implementation of rewriting is allowed to execute bubbling steps, if it is convenient. The completeness of bubbling is not in conflict with the example of Figure 3. Although a bubbling step may destroy a redex, the redex is not irrevocably lost—there always exists a second bubbling step to recover the redex lost by the first step. In the case of Figure 3, a second bubbling step results in `snd(1,2)?snd(1,3)`.

Theorem 2 (Completeness of bubbling). *Let S be a* LOIS*, t an admissible term graph and u a constructor graph such that $t \xrightarrow{*} u$. If $t \simeq_{cd} v$ for some graph v and nodes c and d of t, then $v \overset{*}{\rightsquigarrow} u$ modulo a renaming of nodes.*

Proof. The proof is by induction on the length of $t \xrightarrow{*} u$. Base case: If $t = u$, then v does not exist, and the claim vacuously holds. Ind. case: There exist some node p, rule R and graph t_1 such that $t \to_{p,R} t_1 \xrightarrow{*} u$. We consider two exhaustive cases on d. If d is not in the redex pattern of R at p in t, then, by Lemma 3, there exists a graph v_1 such that $v \xrightarrow{+} v_1$ and $t_1 \simeq^{=} v_1$ modulo a renaming of nodes. By the induction hypothesis, $v_1 \overset{*}{\rightsquigarrow} u$ modulo a renaming of nodes. If d is in the redex pattern of R at p in t, then d is neither the root of t nor the root of v. There exists a dominator e of d in v, witness the root of v, such that $v \simeq_{de} w$; by Theorem 1 $t \simeq_{ce} w$, and e is not in the redex pattern of R at p in t. By Lemma 3, there exists a graph w_1 such that $w \xrightarrow{+} w_1$ and $t_1 \simeq^{=} w_1$ modulo a renaming of nodes. As in the previous case $w \overset{*}{\rightsquigarrow} u$ modulo a renaming of nodes. Since $v \simeq w$ implies $v \rightsquigarrow w$, $v \overset{*}{\rightsquigarrow} u$ modulo a renaming of nodes. □

We now turn our attention to the soundness of combined steps. This is somewhat the complement of the completeness. We prove that bubbling non-deterministic choices does not produce results that would not be obtainable without bubbling. Of course, a bubbling step of a term graph t creates a term u that is not reachable from t by rewriting, but any result (constructor normal form) obtainable from u via combined steps can be reached from t via pure rewriting. We begin by proving that a single bubbling step with destination the root node is sound.

Lemma 4 (Single copying soundness). *Let S be a* LOIS *and t_0 an admissible term graph over the signature of S. If $t_0 \simeq_{cd} t_1 \xrightarrow{*} t_n$, where c is a node of t_0, d is the root of t_0 and t_n is a constructor graph, then $t_0 \xrightarrow{+} t_n$ modulo a renaming of nodes.*

Proof. A diagram of the graphs and steps in the following proof are shown in Fig 5. Since in t_1 the label of the root node d is "`?`" and t_n is a constructor normal form, there must be an index j such that in the step $t_j \to t_{j+1}$ a rule R_j of "`?`" is applied at d. Without loss of generality, we assume that R_j is the rule of "`?`" that selects the left argument and we denote with d_l the left successor of d in t_i for $i = 1, 2, \ldots, j$. In different graphs, d_l may denote different nodes. We

prove the existence of a sequence $t_0 \to u_1 \xrightarrow{*} u_n$, such that for all $i = 1, 2, \ldots, n$, $t_i \to^{=} u_i$ modulo a renaming of nodes. By induction on i, for all $i = 1, 2, \ldots, j$, we define u_i and we prove that $t_i|_{d_l} = u_i$ modulo a renaming of nodes. The latter implies $t_i \to_{d,R_j} u_i$. Base case: $i = 1$. The rule R_j can be applied to t_0 at c and we define u_1 as the result, i.e., $t_0 \to_{c,R_j} u_1$. By Lemma 2, $t_1|_{d_l} = u_1$ modulo a renaming of nodes. Ind. case: We assume the claim for i, where $0 < i < j$, and prove it for $i + 1$. Let $t_i \to_{p,R_i} t_{i+1}$. If p is a node of $t_i|_{d_l}$, then since $t_i|_{d_l} = u_i$ modulo a renaming of nodes, there exists a node q in u_i that renames p. We define $u_i \to_{q,R_i} u_{i+1}$ and the claim holds for $i+1$. If p is not a node of $t_i|_{d_l}$, then $t_{i+1}|_{d_l} = t_i|_{d_l}$. We define $u_{i+1} = u_i$ and the claim holds for $i + 1$ in this case too. Now, since $t_j \to_{d,R_j} t_{j+1}$, we have $t_{j+1} = t_j|d_l$ and therefore $u_{i+1} = t_{j+1}$ modulo a renaming of nodes. For every i such that $j < i < n$, if $t_i \to_{p,R} t_{i+1}$, we define $u_i \to_{q,R} u_{i+1}$, where as before q renames p. Clearly, for every i, $j < i \leqslant n$, $t_i = u_i$ modulo a renaming of nodes. Thus, $t_0 \xrightarrow{+} t_n$ exists as claimed. □

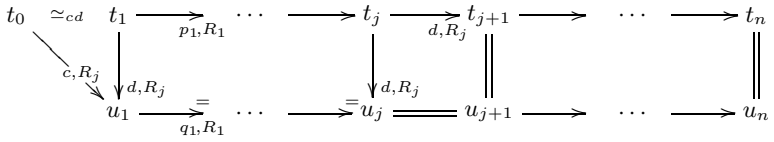

Fig. 5. Diagram of the main graphs and steps involved in the proof of Lemma 4. d is the root node of t_0. R_j is the rule of "?" that selects the left argument. q_i renames p_i.

We believe that the previous proof could be generalized to any bubbling step. However, a simpler and more elegant proof is available by taking advantage of the transitivity and the completeness of bubbling. We show this proof below.

Lemma 5 (Single bubbling soundness). *Let S be a* LOIS *and t_0 an admissible term graph over the signature of S. If $t_0 \simeq_{cd} t_1 \xrightarrow{*} t_n$, where c and d are nodes of t_0 and t_n is a constructor graph, then $t_0 \xrightarrow{+} t_n$ modulo a renaming of nodes.*

Proof. Suppose that d is not the root of t_0; otherwise the claim is already proved by Lemma 4. Let e be the root node of t_0 and also of t_1. Let u_1 be defined by $t_1 \simeq_{de} u_1$. By the transitivity of bubbling, Th. 1, $t_0 \simeq_{ce} u_1$. By the completeness of bubbling, Th. 2, there exists a sequence $u_1 \xrightarrow{+} t_n$ modulo a renaming of nodes. Therefore, $t_0 \simeq_{ce} u_1 \xrightarrow{*} t_n$. Since e is the root node of t_0, by Lemma 4, $t_0 \xrightarrow{+} t_n$ modulo a renaming of nodes. □

Theorem 3 (Soundness of bubbling). *Let S be a* LOIS *and t an admissible term graph over the signature of S. If $t \overset{*}{\rightsquigarrow} u$, for some constructor graph u, then $t \xrightarrow{*} u$ modulo a renaming of nodes.*

Proof. By induction on the number of bubbling steps in $t \overset{*}{\rightsquigarrow} u$. □

6 Related work

Bubbling is introduced in [5] with a rewriting strategy for the overlapping inductively sequential GRSs. This strategy determines, in theory very efficiently, when to execute ordinary rewrite steps and/or bubbling steps. A bubbling step is computed only if it promotes a needed (modulo a non-deterministic choice) rewrite step. Our work proves that the execution of the bubbling steps computed by this strategy preserves all and only the constructor normal forms reachable from a term by pure rewriting. The use of bubbling in the strategy eliminates the incompleteness of backtracking and the inefficiency of copying.

Although strategies for functional logic computations [4] and term graph rewriting [21] have been extensively 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 [11, 12]. A substantial difference of our work with this line is the class of programs we consider, namely non-deterministic ones. Non-determinism is a major element of functional logic programming. Hence, our work fills a major conceptual and practical gap in this area. The attempt to minimize the cost of non-deterministic steps by limiting the copying of the context of a redex by bubbling is original.

Other efforts on handling non-determinism in functional and functional logic computations with shared subexpressions include [17], which introduces the *call-time choice semantics* to ensure that shared terms are evaluated to the same result; [13], which defines a rewriting logic that among other properties provides the call-time choice; and [1] and [22], which define operational semantics based on *heaps* and *stores* specifically for the interaction of non-determinism and sharing.

These efforts, prompted by implementations, abstract the interactions between non-determinism and sharing. In practice, all these implementations adopt strategies, summarized in [4], that have been designed and proved correct for *term* rather than *graph* rewriting or narrowing. Although for a strategy this difference is small, addressing sharing indirectly through computational data structures such as heaps and stores rather than directly prevents graph operations, such as bubbling, which are potentially beneficial.

7 Conclusion and Future Work

Bubbling, with interleaving steps on the arguments of an occurrence of the choice operation, ensures the soundness and completeness of computations without incurring the cost of copying the contexts of redexes with distinct replacements. Programs in which *don't know* non-determinism is appropriately used are likely to produce some terms that fail to evaluate to constructor normal forms. Hence, avoiding the construction of the contexts of these terms can improve the efficiency of these programs.

For example, this situation can be seen in our program for coloring a map. In finding the first solution of the problem, the operation `paint` is called 10 times. Since only four calls are needed, six choices of some color for some state eventually fail. Saving the partial construction of six contexts of `paint` can potentially

improve the efficiency of execution. We are working on an implementation, within the FLVM [7], to quantify the expected improvements. The results of this paper ensure the theoretical correctness of a component of our implementation.

Bubbling steps can be executed any time a choice operation occurs at a non-root node. The problem of determining when it is appropriate to execute a bubbling step and the destination of this step is elegantly solved in [5]. A strategy similar in intent to [11] and [2] determines when a bubbling step promotes a needed (modulo a non-deterministic choice) rewrite step. Thus, bubbling steps are executed only when they are necessary to keep a computation going. This result complements quite nicely several optimality properties known for strategies for functional logic computations [4]

The focus of continued work on this topic is to extend the theory and the implementation to cover narrowing. Narrowing steps are inherently non-deterministic and therefore naturally expressed using the choice operation [6]. For example, to narrow `not x`, where `x` is a free variable, we bind `x` to `True ? False`—the patterns in the definition of `not`—and continue the evaluation of the instantiated term. In our framework, this would require a bubbling step.

Variables are singletons in their contexts. This is a key reason to represent expressions with graphs. However, in our framework, expressions with choice operations represent *sets* of ordinary expressions. Therefore, a variable that has an ancestor node labeled by a choice operation must be handled with care. For example, consider the following contrived program:

```
f x = g x ? h x
g 0 = 0                                                    (11)
h _ = 1
```

The expression `f x`, where `x` is free, evaluates to two different terms with two different bindings. In evaluating the right-hand side of `f`, before instantiating `x` in a narrowing step, `x` must be "standardized apart" as if evaluating `(g u ? h v)` where `u` and `v` are distinct and free. The situation exemplified in (11) is characterized by a variable x that belongs to two terms encoded within a single expression of our framework. The standardization apart of a variable is accomplished by a graph transformation similar to a bubbling step.

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. Constructor-based conditional narrowing. In *Proceedings of the Third ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 199–206. ACM Press, 2001.

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. Lazy context cloning for non-deterministic graph rewriting. In *Proc. of the 3rd International Workshop on Term Graph Rewriting, Termgraph'06*, pages 61–70, Vienna, Austria, April 2006.
6. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP'06)*, Seattle, WA, August 2006. To appear.
7. 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.
8. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
9. M. Bezem, J. W. Klop, and R. de Vrijer (eds.). *Term Rewriting Systems*. Cambridge University Press, 2003.
10. 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.
11. R. Echahed and J.-C. Janodet. On constructor-based graph rewriting systems. Research Report 985-I, IMAG, 1997.
12. 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.
13. J. C. González Moreno, F. J. López Fraguas, M. T. Hortalá González, and M. Rodríguez Artalejo. An approach to declarative programming based on a rewriting logic. *The Journal of Logic Programming*, 40:47–87, 1999.
14. 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.
15. 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.
16. 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.
17. H. Hussmann. Nondeterministic algebraic specifications and nonconfluent rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
18. J.W. 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.
19. F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proceedings of RTA '99*, pages 244–247. Springer LNCS 1631, 1999.
20. M. J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.
21. D. Plump. Term graph rewriting. In H.-J. Kreowski H. Ehrig, G. Engels and G. Rozenberg, editors, *Handbook of Graph Grammars*, volume 2, pages 3–61. World Scientific, 1999.
22. 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.