

Computing with Subspaces^{*}

Sergio Antoy

Computer Science Department
Portland State University
P.O. Box 751
Portland, OR 97207, USA
antoy@cs.pdx.edu

Bernd Braßel

Institute of Computer Science
Christian-Albrechts-University of Kiel
Olshausenstr. 40
D-24098 Kiel, Germany
bbr@informatik.uni-kiel.de

Abstract

We propose a new definition and use of a primitive `getAllValues`, for computing all the values of a non-deterministic expression in a functional logic program. Our proposal restricts the validity of the argument of `getAllValues`. This restriction ensures that essential language features like the call-time choice semantics, the independence of the order of evaluation, and the referential transparency of the language are preserved when `getAllValues` is executed. Up to now, conflicts between these language features and primitives like `getAllValues` have been seen as one of the main problems for employing such primitives in functional logic languages.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Semantics—Subspaces; D.3.3 [Programming Languages]: Language Constructs and Features—Non-determinism; F.4.2 [Mathematical Logic and Formal Languages]: Grammars and Other Rewriting Systems—Term Graph Rewriting Systems

General Terms Algorithms, Design, Languages

Keywords Functional Logic Programming Languages, Non-Determinism, Subspaces, Rewrite Systems

1. Introduction

This paper addresses theoretical and practical aspects of non-determinism in functional logic languages modeled by graph rewriting. One of these aspects is a facility for manipulating within a computation all the values of another computation. We refer to the values produced by the nested computation as a *subspace*. Computing with subspaces is a necessary feature of functional logic languages that support non-determinism. The semantics and the implementation of computing with subspaces have been elusive so far. Especially, practical applications of such computations have shown considerable problems concerning the integration with other features of functional logic languages, cf. [12]. This paper attempts to better understand this feature and to discipline its use such that

^{*} Partially supported by the NSF grant CCR-0218224, the DFG grant Ha 2457/5-1 and the DAAD grant D/06/29439.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP'07 July 14–16, 2007, Wrocław, Poland.
Copyright © 2007 ACM 978-1-59593-769-8/07/0007...\$5.00

a seamless integration with other features of functional logic languages, in particular the independence of the order of evaluation and the sharing of subexpressions, is no longer problematic. We believe that this is the first published effort of this kind.

Section 2 motivates the need of computing with subspaces. Section 3 formally defines the key concepts of our discussion, in particular, a primitive operation, `getAllValues`, for computing a subspace. Section 4 recalls background and notations essential to our discussion. Section 5 specifies an abstract implementation of `getAllValues`. Sections 6 and 7 respectively show that the order of evaluation and the sharing of expressions affect the value of a subspace. Section 8 identifies run-time conditions sufficient for ensuring that computations with subspaces produce only intended results. Section 9 discusses how to code programs whose executions lead only to valid computations. Section 10 briefly addresses related work. Section 11 offers our conclusion.

2. Motivation

Non-determinism is the most appealing feature of functional logic programming. A program, which is modeled by a graph rewriting system, is *non-deterministic* when its execution may reduce a redex with multiple replacements. For example, consider a program to find a donor for a blood transfusion to a patient. The following declaration, in Curry [17], defines the blood types:

```
data BloodTypes=Ap | An | ABp | ABn | Op | On | Bp | Bn (1)
```

The identifiers `Ap`, `An`, `ABp`, etc. encode the well-known blood types $A+$, $A-$, $AB+$, etc., respectively.

Blood of a type can be given only to people with blood of some, but not all, other types. For example, $A+$ can be given only to $A+$ and $AB+$, and $O+$ can be given only to $O+$, $A+$, $B+$, and $AB+$. To encode this information in a deterministic language, most programmers would associate with each blood type t a set of all the blood types to which t can be given, or from which t can be received, or both.

In a non-deterministic language, instead, the programmer encodes this information in a simple relational or tabular form:

```
give Ap = Ap
give Ap = ABp
give Op = Op
give Op = Ap
give Op = Bp
give Op = ABp
...
```

The definition of this relation looks like a function definition in the sense that `give Ap` evaluates to both `Ap` and `ABp`, non-deterministically. During a program execution, either value may be

selected. The programmer has no control over this choice but can constrain it if appropriate. Non-determinism supports an expressive programming style because often specifications of problems are non-deterministic as well. For example, suppose that the problem is to find a donor for a blood transfusion and assume the existence of a database of people, donor and/or patients, and their blood types.

```
has "John" = ABp
has "Doug" = ABn
has "Lisa" = An
```

(3)

The specification of this problem is: *a donor for x is a person y that has a blood type that can be given to the blood type of x*. A non-deterministic program to solve this problem consists of a single conditional rewrite rule:

```
donorFor x
| give (has y) := has x & x /= y
= y
where y free
```

(4)

(The condition ensures that y is not x, since self donation is not intended as implied by the specification.) 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 (3). The evaluation of the program is by narrowing [25, 14]. In particular, when the condition of `donorFor` is evaluated, y is initially unknown but becomes instantiated to a suitable value, if one exists.

Non-determinism substantially reduces the effort of designing and implementing data structures and algorithms to encode this problem into a program. In particular, there are neither explicit lists or sets of blood types or people nor operations to process them. The absence of these constructs is quite desirable, but it makes some computations more difficult or impossible.

For example, it is impossible to compute the set of all the donors for a patient. This information could be useful for further processing, e.g., to find the donor closest to a certain donation center or to find the donor with the longest elapsed time since the previous donation. For this reason, the language should provide a primitive function, that we will denote with `getAllValues`, to compute all the values of a non-deterministic expression. Informally, `getAllValues` applied to an argument *e* returns a list containing all the values of *e*. For example, `getAllValues (give Ap)` should return the list [Ap, ABp].

Using all the values of an expression to compute one value of another expression is referred to as *computing with subspaces*. Computing with subspaces is relevant whenever different values of an expression have to be compared in some way to find the intended solution which is the gist of all optimization problems. Without primitives like `getAllValues` the programmer would have to eliminate all uses of non-determinism as soon as an optimization problem arises. We fear that this leads in many cases to generally avoid using non-determinism and we therefore consider programming with subspaces as a crucial feature of functional logic languages.

This paper addresses the problem of computing with subspaces. It formally defines some concepts, proves some of their properties, and discusses the implementation and use of `getAllValues`.

3. Definitions

In this section, we formalize the meaning of “computing all the values of an expression” in the sense discussed earlier. We believe that this is the first published effort of this kind. We present our defini-

tions within the framework of abstract reduction systems [10, 11] since they do not depend on specific properties of the objects being rewritten and of the reduction relation. The concepts introduced in this section will be applied to programs that can be regarded as specializations of abstract reduction systems. An *abstract reduction system*, abbreviated with ARS, is a pair $\mathcal{R} = (T, \rightarrow)$, where T is a set of *objects* and \rightarrow is binary relation on T called *reduction*. If \mathcal{R} is an abstract reduction system, an element of the set of objects of \mathcal{R} is called an *expression of \mathcal{R}* , and a normal form with respect to the reduction relation is called a *value*.

DEFINITION 1 (Set of values). *Let \mathcal{R} be an ARS and e an expression of \mathcal{R} . We define the set of values of e as $\mathcal{A}_e = \{v \mid e \xrightarrow{*} v, v \text{ is a value}\}$, i.e., \mathcal{A}_e is the set of all the values reachable from e .*

Referring to the code fragment in (2), if $e = \text{give Ap}$, then $\mathcal{A}_e = \{\text{Ap}, \text{ABp}\}$.

DEFINITION 2 (Fair enumeration). *Let \mathcal{A} be a set. A fair enumeration of \mathcal{A} is a denumerable sequence $E_{\mathcal{A}} = a_0, a_1, \dots$ such that:*

1. *for every a_i in $E_{\mathcal{A}}$, $a_i \in \mathcal{A}$, and*
2. *for every $a \in \mathcal{A}$, there exists a natural i such that $a_i = a$.*

Continuing with the previous example, both list [Ap, ABp] and [ABp, Ap] are fair enumerations of {Ap, ABp}. Obviously, a non-empty set has many fair enumerations. For example, any finite or infinite sequence a, a, \dots is a fair enumeration of the singleton {*a*}. This indetermination may not appear ideal; nevertheless, we believe that our definition is sensible. We will shortly explain why.

DEFINITION 3 (getAllValues). *Let \mathcal{R} be an ARS and e an expression of \mathcal{R} . We denote with `getAllValues` any function that applied to e returns a fair enumeration of \mathcal{A}_e , the set of values of e .*

The above definition characterizes a class of functions rather than a single function. For our purposes, identifying a single function in this class is impractical. To clarify this point, we define the computation space of an expression.

We begin with the notion of *evaluation strategy*. A strategy determines which step(s) to perform during a computation. A strategy is an essential theoretical and practical component of a programming language. For example, notions like laziness and strictness are direct consequences of the strategy. A strategy should be efficient and it should guarantee that all and only the values of an expression are computed, properties referred to completeness and soundness. [5] surveys several concrete strategies and related concepts within the domain of our discussion.

DEFINITION 4 (Evaluation strategy).

Let $\mathcal{R} = (T, \rightarrow)$ be an ARS. An evaluation strategy of \mathcal{R} is a function $s : T \rightarrow 2^T$ such that, for all expressions e and e' of \mathcal{R} , $s(e)$ is finite, and if $e' \in s(e)$, then $e \xrightarrow{} e'$.*

Many strategies for programming languages are concerned with a single rewriting or narrowing step. For these strategies, that we call *single-step*, referring to Def. 4, if $e' \in s(e)$, then it would suffice to say $e \rightarrow e'$. However, many interesting strategies, e.g., [6, 8, 22, 27], are not single-step.

DEFINITION 5 (Computation space).

Let \mathcal{R} be an ARS and e an expression of \mathcal{R} . The computation space of e for a strategy s , denoted $\mathcal{S}_s(e)$ or simply $\mathcal{S}(e)$ when s is understood, is a possibly infinite, finitely branching tree defined as:

1. *e is the root of $\mathcal{S}(e)$,*

2. for all e' in $\mathcal{S}(e)$, if $s(e') = \{e_1, \dots, e_n\}$, then the children of e' are all and only $\mathcal{S}(e_1), \dots, \mathcal{S}(e_n)$, i.e., the computation spaces of e_1, \dots, e_n .

In most current language implementations, the result of evaluating `getAllValues` e , for some expression e , is obtained by constructing the computation space of e and traversing it to collect the values. When $s(e)$ has more than one element, the order in which these elements are installed into the space and/or collected is not important and may be difficult to control. Therefore, it seems unwise to be too specific about the order in which values are produced. In fact, iterators of collections even in imperative languages, e.g., in the `java.util` package, leave undetermined the order in which the elements of a collection are returned.

Furthermore, the computation space of a given expression e might be infinite even if the set of values of e is finite, and a value may be computed infinitely often [12, Ex. 1.12]. Thus, it seems unwise to be too specific about duplicates, too.

A trivial example is the space of the expression `zeros`, where the operation `zeros` is defined [12] by:

$$\begin{aligned} \text{zeros} &= 0 \\ \text{zeros} &= \text{zeros} \end{aligned} \quad (5)$$

Removing duplicates from lists computed by `getAllValues`, is straightforward.

Finally, we have not specified the conditions under which `getAllValues` should return the empty list `[]`. The undecidability of termination implies that this result can only be produced under special circumstances ensuring that the computation space of e is *finite* and contains no values, cf. Section 5.

To conclude, we prefer the definition of `getAllValues` independent of the order in which values are produced and the presence of duplicates in its result.

4. Background and Notation

In this section we recall some key notions and notations used in this paper. Various formulations of graph rewriting have been proposed, among others [15, 23, 24]. In this paper, we follow the systemization of Echahed and Janodet [15] because it best fits our programs. We cannot recall the definitions in the space allotted to this paper, but we faithfully use the same terminology and notation adopted by [15], which is available on-line.

The space allotted to this paper allows us only to recall the key concepts. The complete details can be found in [15].

DEFINITION 6 (Graph).

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

1. $\mathcal{N}_g \subseteq \mathcal{N}$ is the set of nodes of g ;
2. $\mathcal{L}_g : \mathcal{N}_g \rightarrow \Sigma \cup \mathcal{X}$ is the labeling function mapping each node of g to a signature symbol or a variable;
3. $\mathcal{S}_g : \mathcal{N}_g \rightarrow \mathcal{N}_g^*$ 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) $\text{arity}(s) = k$, then there exist n_1, \dots, n_k in \mathcal{N}_g such that $\mathcal{S}_g(n) = n_1 \dots n_k$;
4. $\text{Roots}_g \subseteq \mathcal{N}_g$ 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 $n \in \text{Roots}_g$ or there is a path from r to n where $r \in \text{Roots}_g$, i.e., every node of g is reachable from some root of g .

A graph g is called a term (graph) if Roots_g is a singleton.

For a given graph g and a node n in g we denote by $g|_n$ the *sub-graph* of g rooted by n , cf. [15, Definition 5]. Moreover, we make explicit use of the notion of *pointer redirection* [15, Definition 8] which intuitively moves all the edges of the graph going into a node (typically the root of a redex) to another node (typically the root of the redex's replacement) while leaving everything else unchanged.

A program is a constructor based, limited overlapping, inductively sequential graph rewriting system, abbreviated GRS. In short, *constructor based* [22] means that the signature is partitioned into (*data*) *constructor* and (*defined*) *operation* symbols. In a *rewrite rule*, which is a pair of terms denoted $l \rightarrow r$, the root of the left-hand side l is labeled by an operation whereas every other node is labeled by a constructor or a variable. *Inductively sequential* [2] means that the set of left-hand sides of all the rules rooted by the same operation, modulo a renaming of nodes and/or variables, can be organized in a hierarchical structure called a *definitional tree* [2]. The existence of this structure is instrumental for the definition of an efficient evaluation strategy and for ensuring other interesting properties of computations. *Overlapping inductively sequential* [3] means that overlapping of (the left-hand sides of) rewrite rules may only be trivial, i.e., two overlapping left-hand sides may differ only in the names of nodes and variables. *Limited overlapping inductively sequential* [7] means that the only overlapping rules are those of an operation, called *choice* and denoted by the infix operator “?”, defined by:

$$\begin{aligned} x \text{ ? } y &= x \\ x \text{ ? } y &= y \end{aligned} \quad (6)$$

Overlapping originating from other rules can be eliminated, without altering the computations, using the choice operation [3]. For example, the operation `give` defined in (2) can be re-coded without overlapping as:

$$\begin{aligned} \text{give } \text{Ap} &= \text{Ap} \text{ ? } \text{ABp} \\ \text{give } \text{Op} &= \text{Op} \text{ ? } \text{Ap} \text{ ? } \text{Bp} \text{ ? } \text{ABp} \\ &\dots \end{aligned} \quad (7)$$

An *expression* is a term graph labeled by the symbols of the signature or by variables in which no cycle contains an operation. A *value* is a term graph labeled by variables and constructor symbols only. With some liberty, we say that a symbol, a rewrite rule, etc., belong to a GRS, if they belong to the signature of the GRS or to the set of rewrite rules of the GRS, etc.

A *computation* is a sequence of *rewrite steps*, where a rewrite step *replaces* in an expression an instance of the left-hand side of a rewrite rule with the corresponding instance of the right-hand side. The reflexive and the transitive reflexive closure of \rightarrow are denoted \rightarrow^* and \rightarrow^+ , respectively. When useful, a step will be denoted with attributes such as the node of the replaced subgraph and/or the applied rule as in $\rightarrow_{n,R}$.

Functional logic computations are sequences of rewriting and/or narrowing steps. A *narrowing step* generalizes a rewrite step in that it is applied to expressions that may contain logic (unbound) variables. These variables may be instantiated by the narrowing step. For example, when `has y` is evaluated in an execution of (4), `y` is unbound, and a narrowing step would instantiate it to “John” or “Doug”, etc., non-deterministically. Loosely speaking, the following program is equivalent to (4) but executes rewriting steps only.

$$\begin{aligned} \text{donorFor } x \\ | \text{ give } (\text{has } y) &:= \text{has } x \ \& \ x \neq y \\ &= y \\ \text{where } y &= \text{"John"} \text{ ? } \text{"Doug"} \text{ ? } \text{"Lisa"} \end{aligned} \quad (8)$$

Narrowing is a very convenient programming feature, but at the computation level it can be replaced by rewriting. [9] formalizes the equivalence of narrowing and rewriting when logic variables are replaced by non-deterministic expressions and vice versa. Thus, in this paper, we regard computations as sequences of rewrite steps without too much loss of generality.

Finally, we recall that a node d *dominates* a node n in a rooted graph g if every path from the root of g to n contains d .

5. Implementation

We do not know how to define `getAllValues` with ordinary rewrite rules. We think that this is impossible because the semantics of non-determinism should ensure that the programmer has no control on which non-deterministic alternative of choice is selected. The lack of an ordinary definition of `getAllValues` forces us to rely on some *built-in* mechanism of an interpreter or virtual machine. Built-ins are unavoidable in practical programming languages; e.g., arithmetic operators, I/O functions and many system activities, such as getting the current time, are built-in. The rewriting model accommodates built-ins relatively easily.

Below, we describe an abstract implementation of the primitive `getAllValues`. Our treatment is independent of the details of a concrete implementation. We assume the existence of a rewrite rule:

$$\text{getAllValues } x \rightarrow L \quad (9)$$

where x is a variable and L is an expression inaccessible to the programmer (i.e., built-in), which represents “the yet uncomputed portion of the computation space of x .”

The evaluation of L , which depends on x , is conceptually quite similar to the top-level computation of the program, but nested inside it. For every x , the computation space of x is constructed or traversed—on-demand or as-needed, since it may be infinite. The value of L is obtained as follows:

1. Initially, L is a representation of $S(x)$.
2. If L contains some values, then L evaluates to $v:L'$, where v is a value of x and L' is a new built-in expression, similar to L , which represents the same space as L , but with the portion that produced v removed. The evaluation of L' may produce another value v' of x and a new built-in tail L'' , and so on.
3. If L is finite and contains no values, then L evaluates to \square .
4. Otherwise, L is undefined and evaluating `getAllValues` x does not terminate, i.e., the execution of `getAllValues` x takes forever.

The implementation sketched above is compatible with a rewriting model with a lazy evaluation strategy such as the one we are considering. However, some problems arise from the interaction of `getAllValues` with other features of the language, in particular, sharing and the order of evaluation. These interactions are the subjects of the next two sections.

6. Order of Evaluation

A key semantic property of declarative languages is the “independence of the order of evaluation.” This property intends to free the programmer from some difficult details of a computation and to make reasoning about programs simpler. If an expression e has a value, that value should be produced in a finite amount of time using only a finite amount of memory. It is up to an implementation’s strategy to ensure this property by choosing which steps of e to execute and in which order. Generally, the programmer has no detailed knowledge of these choices and limited control over them.

When a program is modeled by a rewrite system \mathcal{R} , the independence of the order of evaluation is often formulated as the confluence of \mathcal{R} . However, the non-deterministic GRSs that model functional logic languages are not confluent. Thus, some attention is required to formalize the concept that we want to capture. Informally, if an expression allows two steps at *different* locations, the results of a computation are not affected by which step is executed first.

DEFINITION 7 (Deterministic confluence).

Let \mathcal{R} be a GRS. We say that \mathcal{R} is deterministically confluent if and only if for every expression t and steps $t \rightarrow_{n_1} t_1$ and $t \rightarrow_{n_2} t_2$ with $n_1 \neq n_2$ there exist expressions u_1 and u_2 such that $t_1 \rightarrow u_1$ and $t_2 \rightarrow u_2$ and $u_1 = u_2$ modulo a renaming of nodes.

Our definition differs from the standard notion of confluence [10, 11] in that we explicitly exclude different steps at the *same* location. Obviously, these steps break the confluence. Referring to the initial example, `give Ap` allows two different steps at the root. These steps yield `Ap` and `ABp` which are normal forms and therefore cannot be joined into a common expression.

THEOREM 1.

A constructor based GRS is deterministically confluent.

PROOF. In a constructor based GRSs two redex patterns overlap if and only if they have the same root. Condition 5 of [15, Def. 2], i.e., that all the occurrences of a variable are shared, is essential to the claim. \square

The operation `getAllValues` invalidates the deterministic confluence of GRSs.

EXAMPLE 1.

Consider the expression $t = \text{getAllValues}(\text{give Ap})$. If the evaluation starts at the root of t , the value of t is $[\text{Ap}, \text{ABp}]$. However, if `give Ap` is evaluated first, t has two values, $[\text{Ap}]$ and $[\text{ABp}]$. The intended value of t is produced by the first computation. In the second computation, the argument of `getAllValues` is “prematurely evaluated.”

Example 1 provides an intuitive notion of “premature evaluation”. Formalising this notion requires some care. In the computation of `getAllValues` $(1+1)$, the evaluation of $1+1$ should not be considered as premature no matter where or when it takes place.

DEFINITION 8 (Deterministic step).

Let \mathcal{R} be a GRS, t an expression of \mathcal{R} , and $B = t \rightarrow_n u$ a step of t at some node n . We say that B is a deterministic step iff there exists no other step $B' = t \rightarrow_n u'$ of t distinct from B .

THEOREM 2. *Let \mathcal{R} be a limited overlapping inductively sequential GRS, and t a term graph of \mathcal{R} . A step $t \rightarrow_R u$ is deterministic iff R is not a rule of “?”.*

PROOF. In a limited overlapping inductively sequential GRS, the only rules with overlapping left-hand sides are the rules defining “?”. \square

DEFINITION 9 (Premature evaluation). *Let \mathcal{R} be a GRS, t an expression of \mathcal{R} , and `getAllValues` e a subexpression of t rooted by a node n . We say that a step $t \rightarrow_m t'$ prematurely evaluates e iff the step is non-deterministic and there exists a path in t from n to m .*

The dependence on the order of evaluation introduced by the primitive `getAllValues` is undesirable. The premature evaluation of the argument of `getAllValues` is a problem only in particular situations that we will discuss shortly. We begin with simple conditions sufficient to ensure that an expression is not prematurely evaluated.

THEOREM 3. *Let \mathcal{R} be a TRS, f an operation of \mathcal{R} defined by a single rule of the form $f(x) \rightarrow r$, where x is a variable and r is any term, t any term of \mathcal{R} of the form $C[f(e)]$, where $C[\]$ is any context and e is any term. In any outermost computation of t , $f(e)$ is reduced before any reduction inside e .*

PROOF. If, for any context $C[\]$ and term e , a step of the computation of t were of the form $C[f(e)] \rightarrow_n C[f(e')]$, with n a node of e , the computation of t would not be outermost, since $f(e)$ is a redex. \square

Both conditions of Th. 3, i.e., that \mathcal{R} is a *term* rewriting system (as opposed to a *graph*) and that the strategy is *outermost*, are necessary and are violated when it comes to functional logic languages.

An *outermost* strategy for a TRS never reduces an expression if there exists another reducible expression above it. In a GRS, the notion of being “above” becomes ambiguous. A subexpression e of an expression t could be reachable from the root of t through two different paths, a situation informally referred to as sharing and addressed in the next section. Thus, e could be outermost along one path, but not along another one. Therefore, we must either develop new strategies or reconsider the notion of outermostness [15, Def. 27].

DEFINITION 10 (Outermost strategy for GRSs).

We say that a strategy s for a GRS is outermost iff for every step $t \rightarrow_n t'$ computed by s , there exists a path from the root of t to n such that the only redex on this path is $t|_n$.

Issues concerning the order of evaluation are even more complicated in practice because current functional logic languages, such as Curry [17] and \mathcal{TOY} [20], are not executed with an outermost strategy. The reason is that outermost strategies are incomplete for these languages, see [4] for a discussion and an example. The incompleteness of outermost strategies stems from unrestricted overlapping of the left-hand sides of rules, which is allowed in these languages. We argue [4] that unrestricted overlapping is unnecessary, if not harmful, and that the class of the overlapping inductively sequential programs [3] is a better model for functional logic languages. Any constructor based conditional rewrite system can be mapped to an overlapping inductively sequential rewrite system that executes the same computations [4]. A sound, complete and optimal *outermost* strategy is available for the overlapping inductively sequential rewrite systems [3, 5]. Optimality in the presence of non-deterministic steps is a subtle concept. A non-deterministic choice is appropriately coded when the programmer does not know which alternative of the choice is best. Obviously, the strategy cannot be expected to choose the best alternative. In this context, a strategy is optimal only if it chooses alternative of choices that must be made to obtain the result(s) of a computation.

7. Sharing

The semantics of non right-linear rules is particularly relevant for non-deterministic expressions. For example, consider the program:

$$f\ x = x + x \quad (10)$$

When the expression $f\ (0\ ?\ 1)$ is evaluated, the two occurrences of x in any instance of the right-hand side of (10) share the same binding at all times. This semantics is called *call-time choice* in [19]. Under the call-time choice semantics, multiple occurrences of a variable are referred to as *shared*, since they share the same value. By contrast, ordinary term rewriting ignores sharing originating from non right-linear rewrite rules. Without sharing, the expression $f\ (0\ ?\ 1)$ may evaluate to 1, which is not obtainable using the

call-time choice semantics. In this context, the behavior of ordinary term rewriting is referred to as the *need-time choice* semantics.

Graph rewriting elegantly captures the call-time choice semantics in condition 5 of [15, Def. 2]. Various other frameworks have been proposed to formalize the behavior of the call-time choice semantics (e.g., [1, 16]). We find term graph rewriting [15] appealing because it is very close to implementation models in imperative languages and promises evaluation strategies [6] inherently more efficient than those developed for term rewriting.

Sharing substantially complicates computing with subspaces. For example, suppose that we group the donors of Example (3) into families:

$$\begin{aligned} \text{family} = & (\text{Family } ("John" ? "Lisa")) \\ & ? (\text{Family } "Doug") \dots \end{aligned} \quad (11)$$

Now, we could write a function like this:

$$\begin{aligned} f\ (\text{Family } x) \mid x == "John" \\ = & \text{getAllValues } (\text{has } x) \end{aligned} \quad (12)$$

The condition is syntactic sugar. From a rewriting viewpoint, the program is:

$$\begin{aligned} f\ (\text{Family } x) = & \text{if } x == "John" \\ & \text{then } \text{getAllValues } (\text{has } x) \\ & \text{else } \text{failed} \end{aligned} \quad (13)$$

where the *if-then-else* function is defined as usual, the variable x is shared in the right-hand side of the rule of f and the symbol *failed* is a non-reducible function [18, Sect. 2.3.2] (i.e., the expression *failed* is not a value).

Suppose that we evaluate $t = f\ \text{family}$. What should the value of t be?

If the programmer had the call-time choice semantics in mind, both occurrences of x in the rule of f have the same binding. Hence, t would evaluate to $[\text{ABp}]$. The call-time choice makes *getAllValues* in the rule of f totally ineffective, which probably is not what the programmer intended.

If the programmer had the need-time choice semantics in mind, each occurrence of x in the rule of f is independent of the other. Hence, t would evaluate to all the blood types of John’s family, i.e. $[\text{ABp}, \text{An}]$. It seems obvious that this is a more sensible semantics for this expression. However, just a slight change of the example strongly favors the opposite view:

$$\begin{aligned} f\ (\text{Family } x) \mid x == "John" \\ = & \text{getAllValues } (\text{give } x) \end{aligned} \quad (14)$$

Now the *getAllValues* in the rule of f is effective, since *give* is a non-deterministic function. The call-time choice semantics now yields all the blood types that can be given to John whereas the need-time choice semantics yields all those that can be given to one of the members of John’s family. We have the impression that the choice of the right semantics is not simple and that function definitions like the ones in Examples (12) and (14) are dangerously ambiguous.

In both cases, the problem originates from the fact that x in the right-hand side of the rules is shared between a subspace and the subspace’s context. In this situation both the call-time and the need-time choice semantics are problematic.

[12] contains a similar discussion of the problems of sharing between an “encapsulation” and its “outside”. Encapsulation and outside more or less correspond to our notions of a subspace and its context, although these concepts are not formally defined in [12]. The focus of the discussion in [12] is whether the evaluation of

`getAllValues` is ensured to be deterministic or not. “Strong encapsulation” ensures a deterministic evaluation whereas “weak encapsulation” does not in all cases. [12] points out that strong encapsulation is desirable for instance in the context of I/O operations.

Our approach is to avoid the ambiguities of interpretation shown by the above examples by a discipline of sharing described in the next section. We restrict the derivations in such a way that obtaining a strong encapsulation is straightforward. In particular, for derivations which are *valid* in the sense to be defined below the differences between the approaches discussed in [12] disappear.

8. Validity

The previous sections have shown that, when computing with subspaces, sharing may lead to premature evaluation of the argument of `getAllValues` that in turn might lead to unintended results. Furthermore, some *a priori* fixed choices to handle sharing, such as weak and strong encapsulation, permit coding and executing programs, such as (12), that appear questionable. To correct the situation, some programs and/or computations must be prohibited. In this section, we define computations that we accept as valid. In the next section, we will show that it is easy to code programs leading only to valid computations and with minimal loss of expressive power.

Characterizing undesirable sharing in a subspace is not a trivial task. [12] contains a wish list for future implementations of functional logic languages, which says that “Some work has to be invested to clearly define “the outside” [of an encapsulated search]”.

We would like to characterize non-deterministic expressions, i.e., expressions whose evaluation executes a non-deterministic step. Unfortunately, this problem is undecidable. But there exists a sufficient condition to ensure that the evaluation of an expression will not execute non-deterministic steps. In other words, ensure that an expression is deterministic.

DEFINITION 11 (Depend on). *Let \mathcal{R} be a GRS and f and g operations of \mathcal{R} . We say that f depends on g if there exists a rule of f in whose right-hand side there occurs an operation h such that either $g = h$ or h depends on g .*

DEFINITION 12 (Deterministic expression).

Let \mathcal{R} be a limited overlapping inductively sequential GRS and t an expression of \mathcal{R} . We say that t is deterministic if and only if no operation occurring in t depends on “?”.

DEFINITION 13 (Unshared expression). *Let \mathcal{R} be a GRS, t an expression of \mathcal{R} of the form $C[f(e)]$, where f is an operation, e an expression and $C[\]$ is a context. Let n and m be the roots of $f(e)$ and e , respectively. We say m is unshared for n in t , or more simply that e is unshared for f when nodes and context are understood from the discussion, if and only if either $t|_m$ is a deterministic expression or n is a dominator of m and of all the descendants of m in t .*

The above definitions intend to capture the situation in which a non-deterministic expression e is shared by a subspace and its context. In many cases, the intent is that the evaluation of e should contribute all the values of e to the subspace, but only one value to the context. Since e is shared, the premature evaluation of e and the call-time choice semantics become problematic. We will soon show that it is not difficult to avoid undesired sharing in most practical programs.

THEOREM 4. *Let \mathcal{R} be a limited overlapping inductively sequential GRS and t an expression of \mathcal{R} . If e is an unshared expression for `getAllValues` in t , then e will not be prematurely evaluated in any outermost computation of t .*

PROOF. If e is deterministic, the claim stems directly from Def. 9. Otherwise, by (9), `getAllValues` e is a redex. Since e is unshared, by Def. 10 no redex at or below e is outermost. So e will not be evaluated before `getAllValues` e is reduced. \square

The previous result provides a sufficient condition to ensure that e is not prematurely evaluated in $t = C[\text{getAllValues } e]$. Unfortunately, when the evaluation of t starts, it is possible that the computation space of e has already been pruned. The following program shows the point.

$$\begin{aligned} f \ x &= g \ (x, \ x) \\ g \ (0, \ y) &= \text{getAllValues } y \end{aligned} \quad (15)$$

The evaluation of $f(0?1)$ leads to $t' = \text{getAllValues } 0$. When t' is evaluated, 0 is unshared for `getAllValues` in t' , but of course it is the result of pruning the space of $0?1$.

We would like to be able to tell when an argument e' of an application of `getAllValues` originates from some expression e , i.e., $e \xrightarrow{+} e'$, such that $\mathcal{S}(e') \neq \mathcal{S}(e)$. We do not know how to determine that, but we have a practical sufficient condition to ensure that $\mathcal{S}(e') = \mathcal{S}(e)$.

DEFINITION 14 (Tagging). *Let \mathcal{R} be a limited overlapping inductively sequential GRS, t_0 an expression of \mathcal{R} , and $\mathcal{B} = t_0 \rightarrow t_1 \rightarrow \dots$ a computation of \mathcal{R} . For every t_i in \mathcal{B} , tag_i is a mapping from the nodes of t_i to $\{\circ, \bullet\}$ defined by induction on the index i as follows:*

1. *Base case: the index is 0. We define $\text{tag}_0(n) = \circ$, for every $n \in \mathcal{N}_{t_0}$.*
2. *Ind. case: the index is $i + 1$, and by the induction hypothesis, tag_i is defined. Let $t_i \xrightarrow{p,R,h} t_{i+1}$ be the i -th step of \mathcal{B} , where p is a node of t_i , $R = l \rightarrow r$ is a rule of \mathcal{R} , and h is the homomorphism matching l to $t_i|_p$. We recall that t_{i+1} is defined [15, Def. 23] as $t_i[p \leftarrow h(r)]$, which is defined [15, Def. 9] as $\rho(t_i + h(r))|_{\rho(\mathcal{R}_{\text{oots}_{t_i}})}$, where ρ is the pointer redirection [15, Def. 8] function.*

We define $\text{tag}_{i+1}(n)$ by cases as follows:

$$\text{tag}_{i+1}(n) = \begin{cases} \bullet & \text{if } R \text{ is a rule of “?”}, \\ & \text{or, for some non-variable} \\ & \text{node } m \in \mathcal{N}_l, \text{tag}_i(h(m)) = \bullet; \\ \text{tag}_i(p) & \text{otherwise.} \end{cases}$$

$$\text{tag}_{i+1}(n) = \begin{cases} \circ & \text{if } n = \rho(h(n')), n' \in \mathcal{N}_r, \\ & \text{and } \mathcal{L}_r(n) \text{ is not a variable;} \\ \text{tag}_i(n) & \text{otherwise.} \end{cases}$$

Intuitively, the root of a replacement resulting from a non-deterministic step or depending on a non-deterministic step (1st case) is tagged with \bullet , otherwise it assumes the tag of the root of the redex (2nd case). A node created by a step is tagged with \circ (3rd case). Finally, a node passed along by a step maintains its tag (4th case).

For example, consider again program (15). To ease the reading, we display the tag of a node as a superscript and we omit the \circ tag.

$$\begin{aligned} f \ (0?1) &\rightarrow g \ (0?1, 0?1) \\ &\rightarrow g \ (0^\bullet, 0^\bullet) \\ &\rightarrow \text{getAllValues } 0^\bullet \end{aligned} \quad (16)$$

Def. 14 involves some subtle points that we try to clarify with the following example. Consider the function that computes the length of a list:

$$\begin{aligned} \text{len } [] &= 0 \\ \text{len } (.:xs) &= 1 + \text{len } xs \end{aligned} \quad (17)$$

In the following computations tagging is displayed as in the previ-

ous example:

$$\begin{aligned}
\text{len}[(0?1)] &\rightarrow \text{len}[0^\bullet] \rightarrow 1+\text{len}[] \rightarrow 1+0 \rightarrow 1 \\
\text{len}([0]?[]) &\rightarrow \text{len}[0]^\bullet \rightarrow 1+^\bullet\text{len}[] \rightarrow 1+^\bullet 0 \rightarrow 1^\bullet \\
\text{len}([0]?[1]) &\rightarrow \text{len}[0]^\bullet \rightarrow 1+^\bullet\text{len}[] \rightarrow 1+^\bullet 0 \rightarrow 1^\bullet
\end{aligned}
\tag{18}$$

The first computation executes a non-deterministic step which is “forgotten” since the non-determinism of the step does not affect following steps. The second computation correctly recognizes that the result depends on a non-deterministic step. The third computation is a “false positive.” The result depends on a non-deterministic step, but any other choice would have produced the same result. We do not think that there is a practical way of avoiding false positives.

Tagging allows us to detect when a step of `getAllValues` might produce unintended results because previous steps of the computation might have pruned the computation space of the argument.

THEOREM 5. *Let \mathcal{R} be a limited overlapping inductively sequential GRS and $\mathcal{B} = t_0 \rightarrow t_1 \rightarrow \dots \rightarrow t_i$ a computation of \mathcal{R} . If, for every node n of t_i , $\text{tag}_i(n) = \circ$, then $\mathcal{S}(t_0) = \mathcal{S}(t_i)$ modulo a renaming of nodes.*

PROOF. Obviously, $\mathcal{S}(t_i) \subseteq \mathcal{S}(t_0)$, thus we only need to prove the opposite containment. Let u be a value of \mathcal{R} such that $t_0 \xrightarrow{*} u$. We prove that $t_i \xrightarrow{*} v$, where u and v differ only for a renaming of nodes. If every node of every expression of \mathcal{B} is \circ -tagged, then the claim stems from the deterministic confluence of \mathcal{R} ensured by Th. 1. Otherwise, we construct a new computation of \mathcal{R} , $\mathcal{B}' = t'_0 \xrightarrow{*} t'_1 \xrightarrow{*} \dots \xrightarrow{*} t'_i$ and we show that $t'_0 = t_0$ and $t'_i = t_i$ and every node of \mathcal{B}' is \circ -tagged. Intuitively, to construct \mathcal{B}' , first, we “mark” in every expression of \mathcal{B} every \bullet -tagged node and every node dominated by it. The marking defines a “waterline” that partitions each expression. The portion above the waterline is unaffected by the portion below the waterline and non-deterministic steps affect only the portion below the waterline. Then, we obtain \mathcal{B}' from \mathcal{B} by “skipping” every step whose replacement has some marked node. This defines a sequence of expressions t'_0, t'_1, \dots, t'_i . Formally, by induction on k , we define the term t'_k , and we claim that $t'_k \xrightarrow{*} t'_{k+1}$, the non-marked portions of t_k and t'_k are isomorphic (equal modulo a renaming of nodes [15, Def. 10]), and every node of t'_k is \circ -tagged. The base case, is immediate. For the inductive case, assume the claim for all the expression of \mathcal{B}' up to k . If the replacement of the step $t_k \rightarrow t_{k+1}$ has some marked node, then $t'_k = t'_{k+1}$ and the claim is a direct consequence of the induction hypothesis. If the replacement of the step $t_k \rightarrow t_{k+1}$ has no marked node, then $t'_k \rightarrow t'_{k+1}$ uses the same rule on an isomorphic redex and, again, the claim is a direct consequence of the induction hypothesis. Since by hypothesis t_i has no marked nodes, $t'_i = t_i$ modulo a renaming of nodes and this proves the claim. \square

DEFINITION 15 (Validity). *Let \mathcal{R} be a limited overlapping inductively sequential GRS, t_0 an expression of \mathcal{R} , and $\mathcal{B} = t_0 \rightarrow t_1 \rightarrow \dots$ a computation of \mathcal{R} . We say that \mathcal{B} is valid iff, for every step $t_i \rightarrow t_{i+1}$ that reduces `getAllValues` e , e is unshared for `getAllValues` in t_i and every node of e is \circ -tagged.*

It is not difficult to check the validity of a computation at run-time. Tagging is a very fast operation. It consumes a single bit of information for every node of an expression, and its time-complexity is linear in the size of a redex replacement. Checking whether the argument of an application of `getAllValues` is unshared is an efficient computation, too. It must be performed only once when `getAllValues` is applied, and its time-complexity is linear, in the worst case, in the size of the expression being evaluated.

With an outermost strategy, valid computations in limited overlapping inductively sequential GRSs do not suffer from the problems discussed in Sections 6 and 7. In particular, all the approaches discussed in [12] behave identically for valid computations, since there is no sharing of non-deterministic expressions of a subspace with the “outside.” The overall cost of ensuring the validity of a computation seems modest and acceptable. The only remaining problem is to assess how difficult and/or inconvenient it is for a programmer to code programs that prevent invalid computations. This is addressed in the next section.

9. Programming

In this section, we show how to resolve the ambiguities, discussed in Sections 6 and 7, arising when a non-deterministic expression is shared between a subspace and its context. There is no single solution to resolve all the situations. Rather, we present two approaches.

One approach eliminates undesirable sharing between a subspace and its context using the results of Section 8. Roughly speaking, we lift the non-deterministic shared expressions above the subspace and share only deterministic expressions. This is achieved by simulating the need-time choice semantics, which is not directly available in the language. For the simulation, we use only standard features—interestingly enough, we rely on subspaces. The other approach preserves non-deterministic expressions shared between a space and its context for problems where this sharing is intended. These expressions must be evaluated according to the standard call-time choice semantics. Our definition of `getAllValues` does not handle this situation since we stipulated that `getAllValues` should return a fair enumeration of *all* the values of its argument. Also Curry does not handle this situation since there are no syntactic or semantic constructs in the language to distinguish the two approaches.

We begin with the approach that eliminates non-deterministic sharing. To simulate the need-time choice semantics, we define the following non-deterministic operation:

$$\text{chooseValue } (u:v) = u ? \text{chooseValue } v \tag{19}$$

The operation `chooseValue` is a left inverse of `getAllValues`. The following result formalizes this relationship.

THEOREM 6 (Left Inverse).

Let \mathcal{R} be an ARS and t and v an expression and a value of \mathcal{R} , respectively. Let \mathcal{R}' extend \mathcal{R} with the data type list and the operations `getAllValues`, `chooseValue`, “?”. If $t \xrightarrow{} v$ in \mathcal{R} , then $\text{chooseValue}(\text{getAllValues}(t)) \rightarrow v$ in \mathcal{R}' , and vice versa.*

PROOF. If v is a value of t and $L = \text{getAllValues}(t)$, then, for some $i \geq 0$, the i -th element of L is v . By induction on i , v is a value of `chooseValue`(L). The vice versa is analogous. \square

An example shows how to simulate the need-time choice semantics using the operation `chooseValue`. The following program is very similar to (10), which was used to explain the call-time choice semantics.

$$\begin{aligned}
\text{f } x &= \text{chooseValue } z + \text{chooseValue } z \\
&\text{where } z = \text{getAllValues } x
\end{aligned}
\tag{20}$$

The `where` clause is syntactic sugar. The meaning of the program is [17, p. 80]:

$$\begin{aligned}
\text{f } x &= g (\text{getAllValues } x) \\
\text{g } z &= \text{chooseValue } z + \text{chooseValue } z
\end{aligned}
\tag{21}$$

Observe that x is unshared for `getAllValues` in the above fragment. Thus, any outermost computation of $t = \text{f } (0 ? 1)$, when f

is defined by (20), is valid. Th. 6 ensures that if u and v are any two values of x , then $u + v$ is a value of t . By contrast, program (10) only adds to itself any value of x .

In general, the sharing of an expression between a subspace and its context originates from a non-right linear rule of the form:

$$f(\dots x \dots) \rightarrow g(\dots x \dots x \dots) \quad (22)$$

For simplicity, we assume that this is the only rule creating multiple occurrences of x . During a computation, an instance of the right-hand side evaluates to $t = C_2[x][\text{getAllValues}(C_1[x])]$, where $C_2[[]]$ and $C_1[[]]$ are contexts. If the intended semantics of the evaluation of x is the need-time choice, we replace (22) with:

$$\begin{aligned} f(\dots x \dots) &\rightarrow h(\dots \text{getAllValues}(x) \dots) \\ h(\dots y \dots) &\rightarrow g(\dots \text{chooseValue}(y) \dots \text{chooseValue}(y) \dots) \end{aligned} \quad (23)$$

With the above transformation, $\text{chooseValue}(y)$ eventually replaces x in t . In the transformed program, any expression bound to y is shared between a subspace and its context, but this expression is deterministic. Furthermore, any expression bound to x , which might be non-deterministic, is no longer shared between a subspace and its context. The correctness of the above transformation is formalized below.

THEOREM 7 (Transformation correctness).

Let \mathcal{R} be a limited overlapping inductively sequential GRS, t an expression of \mathcal{R} of the form $C_2[x][\text{getAllValues}(C_1[x])]$, where $C_2[[]]$ and $C_1[[]]$ are contexts and the intended semantics of the evaluation of x is the need-time choice. Let

$$t' = C_2[\text{chooseValue}(y)] [\text{getAllValues}(C_1[\text{chooseValue}(y)])]$$

$$\text{where } y = \text{getAllValues}(x).$$

1. (Soundness) If $t' \xrightarrow{*} v$, then $t \xrightarrow{*} v$;
2. (Completeness) If v_a and v_b are values of x such that

$$C_2[v_a][\text{getAllValues}(C_1[v_b])] \rightarrow v,$$

$$\text{then } t' \xrightarrow{*} v.$$

PROOF. Soundness: let $[\dots v_a \dots v_b \dots]$ be a fair enumeration of $\text{getAllValues}(x)$, where v_a and v_b are the choices made by chooseValue for the two occurrences of x in t' , i.e.,

$$t' \xrightarrow{*} C_2[v_a][\text{getAllValues}(C_1[v_b])] \xrightarrow{*} v.$$

By Th. 6, $t \xrightarrow{*} C_2[v_a][\text{getAllValues}(C_1[v_b])]$ as well, and the claim follows. Completeness: let v_a and v_b values of x such that $t \xrightarrow{*} C_2[v_a][\text{getAllValues}(C_1[v_b])] \rightarrow v$.

Then, $[\dots v_a \dots v_b \dots]$ is fair enumeration of $\text{getAllValues}(x)$. By Th. 6, for some choices of chooseValue ,

$$t' \xrightarrow{*} C_2[v_a][\text{getAllValues}(C_1[v_b])]$$

as well, and the claim follows. \square

The existence of values of x in the statement of the completeness of the above theorem is a necessary condition. Without this condition the transformation does not preserve the semantics of some programs. For example, consider:

$$\begin{aligned} \text{f } x \mid \text{head } (\text{getAllValues } (x \neq [])) \\ = \text{head } x \end{aligned} \quad (24)$$

and the expression $t = \text{f } [1..]$. There is a rewrite derivation of t to 1. Recall that $[1..]$ evaluates to the infinite list $[1, 2, 3, \dots]$, hence it has no values. The computation space of $[1..]$ is infinite and has no leaves.

Our transformation applied to (24) yields:

$$\begin{aligned} \text{f } x \mid \text{head } (\text{getAllValues } (\text{chooseValue } y \neq [])) \\ = \text{head } (\text{chooseValue } y) \\ \text{where } y = \text{getAllValues } x \end{aligned} \quad (25)$$

Since $\text{getAllValues } [1..]$ does not terminate, according to (25) $\text{f } [1..]$ does not terminate as well.

Our transformation works in programs where the result of a computation depends only on the *values* of some expression e shared between a subspace and its context, where by *value* we mean a constructor normal form of e , as opposed to some term or head normal form derived from e . We have not found practical programs that violate the above condition. Sharing an expression e between a subspace and its context should be an unfrequent circumstance in a meaningful program, since in the subspace the program depends on all the values of e , whereas in the context the program depends on one specific value e only. In fact, when this kind of sharing occurs, see the n -queens program below, the non-determinism of e is better left out of the subspaces, i.e., the program depends on only one value of e . Furthermore, the situation is compounded by the fact that e does *not* have a value, but it still determines the result of the computation, another unfrequent circumstance.

We now turn our attention to programs where sharing between a subspace and its context is intended. Shared subexpressions are evaluated according to the regular semantics, i.e., the call-time choice. The problem is the non-determinism of a shared subexpression. In this case, loosely speaking, an occurrence under getAllValues should not contribute more than one value to the computation space. We clarify this subtle point with an example.

The problem is the well-known n -queens puzzle: place n queens on a $n \times n$ board so that no queen captures any other queen. A typical solution represents a placement of the queens on the board as a permutation p of the integers $1, 2, \dots, n$. If the i -th element of p is j , a queen occupies the board square at coordinates (i, j) .

A program designed around the generate and test pattern non-deterministically generates a permutation p and tests whether p is *safe* according to the rules of the puzzle. A conceptually simple approach to test the safety of a permutation is by means of a constraint, `unsafe`, that takes a placement of the queens on the board and succeeds if two non-deterministically chosen queens of the placement capture each other. If, for a permutation p , the computation space of `unsafe p` has no values — i.e., there are no pairs of queens capturing each other — then p is a solution of the puzzle.

In Curry we code the above program as follows, where `length`, `abs` and `permute` are library operations with the obvious meanings.

$$\begin{aligned} \text{queens } n = \text{safe } (\text{permute } [1..n]) \\ \text{safe } p \mid \text{getAllValues } (\text{unsafe } p) == [] = p \\ \text{unsafe } (._+i:z++j:_) = \text{abs } (i-j)-1 := \text{length } z \end{aligned} \quad (26)$$

The definition of `unsafe` relies on *function patterns*, a feature recently added to the PAKCS implementation of Curry. A slightly more verbose and less elegant formulation that relies only on rewriting is easy to code.

There are two noteworthy aspects of the above program. The variable p in the rule of `safe` is shared between a subspace and its context. Although p is bound to a non-deterministic expression, i.e., `permute [1..n]`, the intent is that p should contribute only a single value to the subspace. This is the value being returned by the program. Informally, the specification of the program is “*If there exists a placement p of queens on the board such that, for all pairs (i, j) of queens of the placement, i does not capture j , then p is a solution.*” The code is a direct translation of this specification, and it relies on a subspace for ensuring that all the pairs of queens on the board are checked for safety.

The execution of this program is particularly challenging because it is required to evaluate the entire subspace in which p occurs before the evaluation of p in the context of that subspace. The natural order of evaluation would use all the values of p to compute the subspace before returning one value of p , when the condition of `safe` is satisfied. In Curry, there is no explicit mechanism to encode in a program that the non-determinism of an expression should be excluded from a subspace.

In the above example, an unsatisfactory solution to exclude the non-determinism of p from the computation space of `unsafe p` is to evaluate p to a normal form *before* `unsafe p` is evaluated. This is relatively simple, since in a conditional rule, the condition is evaluated before the right-hand side. However, in general, this approach has substantial drawbacks. The evaluation of some expression e to a value to exclude the non-determinism of e from a subspace potentially changes both the semantics and the efficiency of a program.

10. Related Work

Primitives for computing with subspaces are present in many implementations of Curry. Differences among implementations concern sharing between a subspace and its context. The MCC compiler [21] adopts a particular form of call-time choice for shared encapsulation. By contrast, the KICS compiler [13] adopts the need-time choice. In addition, both compilers feature the encapsulation proposed in [12] (see below). The PAKCS [18] interpreter adopts a particular mix of both semantics that depends on the order of evaluation and is strict.

Work on formalizing computations with subspaces is scarce. In [12] the space of a computation is explicitly represented as a tree-like data structure, functions are encoded to traverse this structure according to a depth-first and a breadth-first strategy, and an operational semantics based on [1] is defined for computing the computation space. The intent is to compute with subspaces only within the IO monad to ensure that top-level computations are deterministic.

Encapsulated search is also an important topic in the closely related field of *constraint programming*. A comprehensive description of different operational aspects of encapsulated search in the context of the programming constraint services and constraint combinators is [26]. Search spaces are organized in a so called *space tree*. [26, Section 10.3.2] also constitutes a validity condition called *admissibility* restricting certain manipulations of the space tree, namely *merging* and *injection*. The key idea of admissibility is to keep the space tree free of cycles. In contrast, the validity condition discussed in Section 8 is concerned – in terms of [26] – with the relation of a *superordinated* space to its *subordinated* spaces. The problems discussed here do not transfer to constraint programming; the operational behavior of *tell* [26, Section 10.2] forwards a non-deterministic choice to all *subordinated* spaces. This has great resemblance to *weak encapsulation* [13] and would suffer from the problems discussed in Section 7 when transferred from constraint programming to our setting.

With respect to [12], we offer more formal definitions and prove non-trivial properties of our concepts. In several areas, our work and [12] complement each other. E.g., our definition of `getAllValues` is abstract. An implementation of this abstraction is likely to rely on a representation of a subspace similar to that proposed in [12, Sect. 2.1]. We are not explicitly concerned with ensuring that top-level computations are deterministic, but our approach can be used to the same aim. By contrast to [12], we adopt a declarative semantics, we argue that the order of evaluation is a crucial element of computing with subspaces, and we define a concept of validity that makes the difference between call-time and need-time choice vacuous for encapsulation. We also show that some programs intend to share a non-deterministic expression e between

a subspace and its context by excluding the non-determinism of e from the computation of the subspace. A construct for this situation is missing from current functional logic languages.

11. Conclusion

This paper explores problems and potential solutions of an essential feature of functional logic programming languages modeled by graph rewriting, the accessibility of all the values of a non-deterministic expression within a program. We motivate this feature with some examples, formally define this feature, and show that without some restrictions this feature is incompatible with other features of the language, in particular the independence of the order of evaluation and the call-time choice semantics.

To resolve these incompatibilities, we define several concepts, such as deterministic confluence, premature evaluation, tagging, and eventually the validity of computations with subspaces. We propose a simple transformation that replaces non-deterministic expressions shared between a subspace and its context with deterministic expressions. The transformation preserves the semantics of programs that depend only on the values of shared expressions, and the transformed programs execute only valid computations.

We also show that for some programs this transformation does not capture the intended semantics. A functional logic language should enable some construct for excluding from a subspace all the values of some non-deterministic expression.

Acknowledgments

We are grateful to Michael Hanus, Wolfgang Lux and Khai Pham for comments and discussions on the subject of this paper.

References

- [1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 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. 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.
- [5] S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.
- [6] 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. Extended version to appear in ENTCS.
- [7] 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.
- [8] S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proceedings of the 14th International Conference on Logic Programming (ICLP'97)*, pages 138–152, Leuven, Belgium, July 1997.
- [9] 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.

- [10] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- [11] M. Bezem, J. W. Klop, and R. de Vrijer (eds.). *Term Rewriting Systems*. Cambridge University Press, 2003.
- [12] B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.
- [13] B. Braßel and F. Huch. Translating curry to haskell. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 60–65. ACM Press, 2005.
- [14] N. Dershowitz and D. A. Plaisted. Equational programming. In J. E. Hayes, D. Mitchie, and J. Richards, editors, *Machine Intelligence 11*, chapter 2, pages 21–56. Clarendon Press, Oxford, 1988.
- [15] 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>.
- [16] 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.
- [17] 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.
- [18] M. Hanus (ed.). PAKCS 1.7.3: The Portland Aachen Kiel Curry System. Available at <http://www.informatik.uni-kiel.de/~pakcs>, Sept. 4, 2006.
- [19] H. Hussmann. Nondeterministic algebraic specifications and nonconfluent rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
- [20] F. J. López-Fraguas and J. Sánchez-Hernández. TOY: A multi-paradigm declarative system. In *Proceedings of the Tenth International Conference on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
- [21] W. Lux. An abstract machine for the efficient implementation of Curry. In H. Kuchen, editor, *Workshop on Functional and Logic Programming*, Arbeitsbericht No. 63. Institut für Wirtschaftsinformatik, Universität Münster, 1998.
- [22] M. J. O'Donnell. *Computing in Systems Described by Equations*. Springer LNCS 58, 1977.
- [23] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer-Verlag, 2002.
- [24] 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.
- [25] Uday S. Reddy. Narrowing as the operational semantics of functional languages. In *Proceedings of the IEEE International Symposium on Logic in Computer Science*, pages 138–151, Boston, 1985.
- [26] Christian Schulte. *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*, volume 2302 of *Lecture Notes in Computer Science*. Springer, 2002.
- [27] R. C. Sekar and I. V. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. *Information and Computation*, 104(1):78–109, 1993.