Set Functions for Functional Logic Programming *

Sergio Antoy
Computer Science Department
Portland State University
Portland, OR 97207, USA
antoy@cs.pdx.edu

Michael Hanus
Institute of Computer Science
Christian-Albrechts-University of Kiel
D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract

We propose a novel approach to encapsulate non-deterministic computations in functional logic programs. Our approach is based on set functions that return the set of all the results of a corresponding ordinary operation. A characteristic feature of our approach is the complete separation between a usually-non-deterministic operation and its possibly-non-deterministic arguments. This separation leads to the first provably order-independent approach to computing the set of values of non-deterministic expressions. The proof is provided within the framework of graph rewriting in constructor-based systems. We propose an abstract implementation of our approach and prove its independence of the order of evaluation. Our approach solves easily and naturally problems mishandled by current implementations of 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; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures; 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

Non-determinism is one of the most characterizing features of functional logic languages. A consequence of non-determinism is that some expression may have multiple values, hence some computation may produce multiple results. The programmer cannot select a specific value of a non-deterministic expression, but can constrain a value for a specific purpose. For example, consider the problem of computing a flight itinerary between two cities. Flights are represented by the non-deterministic operation `flight` that stores flight information in a simple tabular form (we use the syntax of the functional logic language Curry [20] for the examples in this paper):

```
flight = (LH469, Portland, Frankfurt,10:.15)
flight = (NWA92, Portland, Amsterdam,10:.00)
flight = (LH10,  Frankfurt,Hamburg,   1:.00)
flight = (KL1783,Amsterdam,Hamburg,   1:.52)
...
```

For example, flight Lufthansa 469 connects Portland, Oregon, to Frankfurt, Germany, in 10 hours and 15 minutes.

An operation that computes an itinerary constrained by at most one intermediate stop is defined as:

```
itinerary orig dest
   | flight =:= (num,orig,dest,len)
   = [num]
   where num, len free
itinerary orig dest
   | flight =:= (num1,orig,stop,len1)
   & flight =:= (num2,stop,dest,len2)
   = [num1,num2]
   where num1, len1, num2, len2, stop free
```

In an interactive environment, the programmer can obtain all the itineraries one after another in some non-deterministic order. Coding the operation `itinerary` using the operation `flight` is much simpler than if the flights were represented using an adjacency matrix or adjacency lists. Using `flight` avoids defining these structures and coding and invoking operations to extract information from them.

However, a drawback of this simplicity is that, without additional machinery, it is impossible to compute, e.g., an itinerary of shortest duration. The reason is that the proposed representation has no notion of *all* the flights and any non-deterministic result of operation `itinerary` is independent of any other result. What is needed is a way to access *all* the values of a non-deterministic expression. In the above example, we would like a function that returns all the itineraries from Portland to Hamburg in a set so that some element with minimal duration can be computed. This function would also be useful in other situations which arise frequently in practice, e.g., detecting if a constraint has solutions. We will discuss an example of this kind in some detail in Section 5.

Primitive operations for the purpose discussed above have been available in functional logic languages since the early days. Unfortunately, every proposal so far suffers from one or more major flaws, in particular the behavior of the primitive operation is unnatural in some cases and/or its result depends on the evaluation order. [5, 12] discuss at length these problems. In this paper, we propose a novel approach characterized by the separation of the non-determinism of an operation from that of its argument(s). We

* This work was partially supported by the German Research Council (DFG) under grant Ha 2457/5-2 and the DAAD grants D/06/29439 and D/08/11852.

PPDP'09, September 7–9, 2009, Coimbra, Portugal.

prove that our approach is independent of the order of evaluation and that it exhibits a natural behavior in situations in which previous approaches misbehave.

We assume some familiarity with functional logic programming and term graph rewriting. Section 2 tersely recalls key concepts and notations. Section 3 motivates our definitions, formalizes our approach, and proves the key result of this paper—the independence of the order of evaluation. Section 4 describes an abstract implementation of our approach and proves its correctness. Section 5 shows that our approach easily and naturally solves a problem outside the domain of previous approaches. Section 6 compares our proposal with previous proposals. Section 7 offers our conclusion.

2. Background

Functional logic programs define data types and operations through a rich set of syntactic and semantics features that include modules with visibility control, nested declarations with local scope, partially applied and higher-order functions, and monadic I/O. The compilation of a *source* program produces a *target* program, semantically equivalent to the source one, in which these features have been removed. The target program belongs to a *core* language that is simpler to both reason about and implement. We define the core language below and describe our results and prove them for this language.

The core language is modeled by a class of graph rewriting systems called *limited overlapping inductively sequential*, abbreviated *LOIS* [4]. In the rest of this paper, a *program* will be a system in this class. In the *LOIS* systems, each rule defining an operation f has the form[1] $f(\bar{t}) = r$ and is required to be left-linear (i.e., there are no multiple occurrences of variables in $\bar{t}$) and constructor-based (i.e., $\bar{t}$ does not contain any operation symbol). Moreover, the left-hand sides of all the rules defining each operation are organized in a hierarchical structure called a definitional tree [1] that guides the evaluation strategy [4]. This class of programs is the intersection [18] of the strongly sequential systems [22] and the constructor-based systems [27]. In order to support non-deterministic operations, as desired for functional logic programming, a *LOIS* system additionally contains a single operation whose rules' left-hand sides are overlapping [2]. This operation is called *choice*. It is denoted by the infix symbol "?" and defined by the following rules:

$$\begin{array}{ll} \texttt{x ? \_ = x} & R_1 \\ \texttt{\_ ? y = y} & R_2 \end{array} \qquad (1)$$

LOIS systems are an adequate core language for functional logic programming since any functional logic program (i.e., constructor-based conditional rewrite system) can be translated into a *LOIS* system [3].

The treatment of graph rewriting is non-trivial. For lack of space, we are unable to recall graph rewriting in this paper. Instead, for the convenience of the reader, we strictly adhere to the definitions and notations of [15] (see also [16]). Below, we recall a couple of essential concepts. Graph rewriting stipulates that a variable labels at most one node of a graph [15, Def. 2, cond. 5]. This is relevant when a variable occurs repeatedly in the right-hand side of a rule. For example, consider the rule (for consistency with the programs presented later, functional application is curried):

```
double x = x + x
```

where '`x`' is a variable and '`+`' is the integer addition. By design, functional logic programs are not confluent. Thus, some expression t matching '`x`' may have two values (normal forms), say 0 and 1. Since there is only one node labeled by '`x`' the values of '`double` t' are only 0 and 2. In other formal models of functional logic programs, e.g., [17], this stipulation is referred to as the *call-time choice* semantics [23].

A component of a graph is a set of nodes labeled by variables and the symbols of (the signature of) the program. Every time a non-collapsing rule, i.e., a rule whose right-hand side is not a variable, is applied, some nodes are introduced by the replacement. These nodes are *fresh* or *new*, i.e., are nodes that do not appear anywhere else in the expression being replaced and in other expressions discussed in the same context. In some situations, e.g., when we reason about commutative diagrams, this difference matters. Sometime, we will apply twice the same step to the same term. The two terms resulting from the two equal steps are not equal. They are "structurally" equal, but the identities of some of their nodes differ because the nodes introduced by one step are not the same as the nodes introduced by the other step. Graphs that differ only for the identities of some nodes are said to be *equal up to a renaming of nodes* [15, Def. 15].

A redex is *deterministic* iff it is an instance of the left-hand side of only one rule. Hence, in a *LOIS* system, a redex is deterministic iff it is not rooted by the choice operation. A step is *deterministic* iff it replaces a deterministic redex, otherwise it is *non-deterministic*. We use "*expression*" as a generic name for a term graph over the signature of a program. An irreducible expression is called a *failure* if it has occurrences of defined operations, otherwise it is called a *value*.

Without loss of generality, we discuss the computation of all the values of some expressions only for rewriting, as opposed to narrowing. [7, Th. 2] shows that narrowing computations in the inductively sequential programs with extra variables are equivalent to rewriting computations in the overlapping inductively sequential programs without extra variables [7, Th. 2]. *LOIS* systems, our programs, are a subset of the latter.

In the following, we write $e_1 \rightarrow_n e_2$ for a rewrite step that performs a replacement at node n in e_1, where the index n is omitted if it is irrelevant. In reasoning about a computation space, we will need to determine if two steps are "substantially" equal. This concept is similar to the equality we consider for graphs. Let e_1 and e_2 be graphs equal up to a renaming of nodes and let $\phi : e_1 \rightarrow e_2$ be a graph isomorphism witnessing this equality. Let $e_1 \rightarrow_{n_1} e_1'$ and $e_2 \rightarrow_{n_2} e_2'$ be steps at nodes n_1 and n_2, respectively, that apply the same rule. If $\phi(n_1) = n_2$, we say that these steps are *equal up to a renaming of nodes*, or more simply that they are equal.

We call v a *value of an expression* e if v is a value with $e \stackrel{*}{\rightarrow} v$. An operation f is *non-deterministic* iff, for some list of values $\bar{v}$, $f(\bar{v})$ has two or more values (up to renaming of node), otherwise it is *deterministic*. For instance, the operation '`?`' is non-deterministic whereas '`double`' is deterministic.

A strategy defines the steps to be executed on an expression. A *strategy*, denoted by φ, maps each expression into a *finite* set of expressions. The meaning is that the steps (multisteps, if the strategy is parallel) that can be executed on e are all and only $e \rightarrow e'$, for any $e' \in \varphi(e)$. A strategy φ is *complete* (*normalizing*) iff, for any expression e and any value v of e, there exists a derivation of e into v consisting only of steps computed by φ. A strategy φ is *sensible* iff, for all expressions e, distinct elements of $\varphi(e)$ originate from distinct *non-deterministic* steps of e. If a

[1] Throughout this paper, we write $\bar{t}$ for a list of (argument) terms $t_1, \ldots, t_n$ and use notions defined for terms also for list of terms. This is justified by the equivalence of n-ary functions and unary functions with tuple arguments.

strategy is not *sensible*, a value of an expression could be repeatedly computed. For example, a strategy that maps $e =$ `1*2+3*4` into $S = \{$`2+3*4`, `1*2+12`$\}$ would not be sensible. A sensible strategy would map e into *either* element of S, or into $\{$`2+12`$\}$, if it is parallel.

A strategy controls the computations. To keep the notation lighter, we assume a fixed strategy and refrain from parameterizing with the strategy concepts defined in this paper, in particular, the computation space. The computation space of an expression e abstracts all the computations of e with a given strategy, and consequently all the expressions derived from e. The *computation space* of e, denoted $C(e)$, according to a (fixed) strategy φ is a possibly-infinite, finitely-branching tree inductively defined by:

$$C(e) = \langle e, \{C(e_1), \ldots, C(e_n)\}\rangle, \quad \text{when } \varphi(e) = \{e_1, \ldots, e_n\}$$

For a *sensible* strategy, a branch has a single child if the step is deterministic and has multiple children if the step is non-deterministic. A leaf of $C(e)$ is an expression that cannot be further evaluated (according to the given strategy). For a *complete* strategy, a leaf is either a value of e or is an expression that cannot be derived to value, i.e., a failure. A *computation* of e is a path in $C(e)$ starting at the root.

A goal of our work is the formulation of an *order-independent* approach to computing all the values of some expression. Informally, this means that the order in which the steps of a computation are executed does not affect the computed result. This property is desirable in programming languages: it frees the programmer from considering a particular order of evaluation for reasoning about a program and makes it easier to perform computations in parallel to exploit the opportunities provided by current and future hardware.

Formalizing the concept of "order independence" for a functional logic language takes a bit of work. In a rewriting-based framework, order independence can be formalized by confluence. Typically, many steps can be applied to an expression. The confluence of a program ensures that the choice of the applied step does not affect the value of the expression. Unfortunately, this is not true for some functional logic programs. Consider the program:

```
coin = 0 ? 1
```

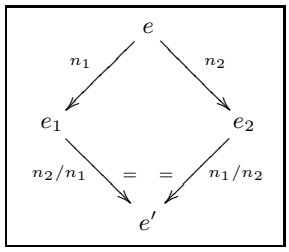
Any complete strategy must map '`coin`' into $\{0, 1\}$ which shows that the choice of the step does affect the result. Therefore, a more specialized concept of confluence is needed for a functional logic language. We recall the following concept from [5].

DEFINITION 1. *A program is* deterministically confluent *iff for every expression t and steps $t \to_{n_1} t_1$ and $t \to_{n_2} t_2$ with $n_1 \neq n_2$ there exist expressions u_1 and u_2 such that $t_1 \xrightarrow{=} u_1$ and $t_2 \xrightarrow{=} u_2$ and $u_1 = u_2$ up to a renaming of nodes.*

Our definition differs from the standard notion of confluence [10, 11] in two aspects: (1) we explicitly exclude *different* steps at the *same* node, and (2) to complete the diagram we use the reflexive closure ($\xrightarrow{=}$) instead of the reflexive transitive closure ($\xrightarrow{*}$) of the rewrite relation. This second difference is only apparent. Condition 5 of [15, Def. 2], i.e., any variable labels at most one node of an expression, ensures that both a node and a redex have at most one descendent (residual) by a step.

Deterministic confluence is an appropriate formalization of the notion of independence of the order of evaluation. Steps at different nodes can be executed in any order without affecting the result of a computation. We have shown that different steps at the same node may break the (ordinary) confluence of a program, but this is the essence of non-determinism, a feature that we highly value in the language. The following result is from [5].

LEMMA 1. *Any program is deterministically confluent.*

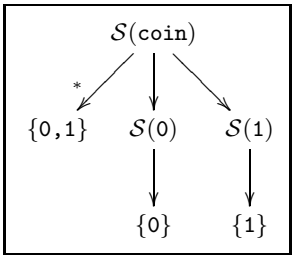


Proof. If e is an expression and n_1 and n_2 are the roots of distinct redexes of e, then, since all rules are constructor-based, node n_2 is not in the redex pattern[2] of the redex at n_1 and vice versa. Thus, if the redex at n_1 is replaced, and node n_2 is not erased [11, pag. 391] by the reduction, the redex pattern at n_2 is unchanged. Thus, the redex at n_2 is replaced exactly in the same way whether or not the redex at n_1 has been replaced. As for the Parallel Moves Lemma [22] in orthogonal systems, the diagram of the two steps commutes, with the simplification that in graph rewriting a redex has at most one descendant, whereas in orthogonal systems it can have multiple descendants. □

Since every program is deterministically confluent, we say that the language of these programs is *order independent*. To date, all the approaches to compute all the values of some expression are based on a primitive operation [5]. Thus, we begin with formalizing this concept and showing its potential pitfall.

DEFINITION 2. *Let $\mathcal{R}$ be a program and e an expression of $\mathcal{R}$. We define the* set of values of e *as $\mathcal{S}(e) = \{v \mid e \xrightarrow{*} v, v \text{ is a value}\}$, i.e., $\mathcal{S}(e)$ is the set of all the values that can be derived from e.*

The steps executed to compute $\mathcal{S}(e)$ depend on the strategy. However, for every expression e, every complete strategy computes the same $\mathcal{S}(e)$, since it computes the same values of e.



Unfortunately, a language augmented with an unrestricted "set of values" operation is not order independent. To see this, assume that such an operation takes an argument and returns the corresponding value. Thus, e.g., if we replace $\mathcal{S}($`coin`$)$ at the root we obtain $\{0, 1\}$, whereas if we replace the argument '`coin`' with one of its values we end up with either $\{0\}$ or $\{1\}$. The popularity of computing the sets of values of an expression through a primitive operation, despite the loss of order independence, is justified by the following paragraphs.

A compiler or interpreter of a functional logic language, which we will refer to as *the device*, is intended to compute all the values of an expression regardless of our treatment. In an interactive environment, *the device* loads a program and executes a "read-eval-print" loop that reads an expression input by the user, evaluates it to a value, and prints this value. Similar to a logic language device, and in contrast to a functional language device, a computation can produce more than one value. For this reason, after printing a value of an expression, *the device* gives the user the option to either compute and print another value of the same expression, if it exists, or to read and process a new expression.

To accomplish this behavior, *the device*, on input an expression e, traverses $C(e)$, the computation space of e. When it visits a

[2] These are [11, Def 2.7.3] the nodes of the redex isomorphically mapped [15, Defs. 10-12] to the non-variable nodes of the left-hand side of the applied rule.

leaf containing a value v, it presents v to the user and waits for instructions. Thus, *the device* must be capable of enumerating, in some order implicitly defined by the strategy and/or the traversal, the set of values on an input expression e and to present these elements incrementally and interactively to the user. Therefore, the same device can be used to compute the set of values of e with one small change: every value found during the traversal of $C(e)$ is placed in a structure suitable for representing a set rather than being presented to the user. This behavior only approximates the computation of a set of values. For example, consider the definition:

```
zs = 0 ? zs
```

It is easy to see that $\mathcal{S}(\texttt{zs}) = \{0\}$. However, the computation space of '`zs`' is infinite. Hence, the traversal and accumulation of the values of this space does not terminate. This is an asset rather than a liability. In a programming environment in which computations are non-strict, infinite structures are common. Thus, the behavior of *the device* is not only acceptable, but even desirable. A program computes a set of values lazily as demanded by its context like any other expression evaluated during an execution.

Within a program, a set of values is an instance of an algebraically defined data type suitable for abstracting a set and it is defined entirely within the language. The program manipulates instances of this type through operations that, e.g., check whether a set is empty, or some element belongs to a set, or some set is contained in some other set, etc. Instances of this type are evaluated lazily as any other instance of any other type. The rewriting framework that we use to formulate and prove properties of our approach effortlessly handles potentially infinite expressions, if the strategy is complete, and it is therefore adequate to model the behavior of a lazy programming language.

3. Formalization

Since an unrestricted operation to compute sets of values is order dependent, as the example of $\mathcal{S}(\texttt{coin})$ shows, we are interested in more advanced concepts. For instance, one can only check for emptiness of sets of values [24, 28] or try to define a "computation barrier" to strongly separate non-deterministic and deterministic computations [12, 13] (see also the discussion in Section 6). We are interested in a more flexible approach that supports the separation of different regions of non-determinism without being dependent on an evaluation order. For this purpose, we propose to associate to any operation f of a program a new operation, denoted by $f_{\mathcal{S}}$ and called set function of f, intended to compute the set of values computed by f. This concept is delicate because we want to capture the non-determinism of f but exclude any non-determinism originating from the argument(s) to which f is applied. Obviously, the operation $f_{\mathcal{S}}$, which is deterministic, is interesting only when f is non-deterministic. We begin with a contrived example whose merit is only its simplicity. Consider the following definitions, where '`coin`' was defined earlier:

```
bigCoin = 2 ? 4
f x = coin + x
```
(2)

The set of values of '`f bigCoin`' is $\{2, 3, 4, 5\}$. By contrast, '$\texttt{f}_{\mathcal{S}}$ `bigCoin`' is a non-deterministic expression with two values, $\{2, 3\}$ and $\{4, 5\}$. These values are obtained when '`bigCoin`' evaluates to 2 and 4, respectively. The non-determinism of '$\texttt{f}_{\mathcal{S}}$ `bigCoin`' originates entirely from '`bigCoin`'. In this case, it turns out that:

$\texttt{f}_{\mathcal{S}}$ `x` $= \mathcal{S}(\texttt{coin} + c)$, where c is a value of `x`

This example suggests a simple approach for computing the values of $f_{\mathcal{S}}(\bar{t})$: for each *value* $\bar{c}$ of $\bar{t}$, $\mathcal{S}(f(\bar{c}))$ is a value of $f_{\mathcal{S}}(\bar{t})$. This approach works well for program (2), but it is inadequate in other situations. For some operation f and argument $\bar{t}$, $f_{\mathcal{S}}(\bar{t})$ may have values even if $\bar{t}$ has no values. For example, consider:

```
head (x:xs) = x
coinList = coin : coinList
```
(3)

The expression '`coinList`' has no values, but '$\texttt{head}_{\mathcal{S}}$ `coinList`' has two values, $\{0\}$ and $\{1\}$. This example suggests an approach for computing the values of $f_{\mathcal{S}}(\bar{t})$ more appropriate to a lazy language: evaluate $\bar{t}$ to some expression $\bar{c}$ such that in $\mathcal{S}(f(\bar{c}))$ there are no steps of $\bar{c}$. This approach is consistent with a non-strict order of evaluation, and it works well for program (3), but it is still inadequate in some situations. For some operation f and argument $\bar{t}$, there is no expression $\bar{c}$ derived from $\bar{t}$, such that $\mathcal{S}(f(\bar{c}))$ executes no steps of $\bar{c}$. For example, consider the program:

```
g x = x : g (x+1)
f (x:xs) = x ? f xs
```
(4)

It is easy to verify that $\texttt{f}_{\mathcal{S}}\texttt{(g 0)} = \{0, 1, 2, \ldots\}$. The computation of '$\texttt{f}_{\mathcal{S}}$`(g 0)`' executes infinitely many steps inside '`g 0`'. The notion that a step is "inside" some expression is intuitive and it will be formalized in Definition 3. Therefore, to compute $f_{\mathcal{S}}(\bar{t})$, for some operation f and argument $\bar{t}$, we cannot hope to evaluate $\bar{t}$ once and for all to some expression $\bar{c}$ and then to compute $\mathcal{S}(f(\bar{c}))$ without touching $\bar{c}$ again. This fact shapes how we define and implement set functions. We begin by formalizing the notion that a step is inside or "originates from" some expression.

DEFINITION 3. *Let $\mathcal{R}$ be a program, e an expression of $\mathcal{R}$, and s a subexpression of e rooted at some node n. Let $A : e = e_0 \to e_1 \to \cdots$ be a derivation of e. We* flag *the nodes of e_i in derivation A with either of two flags,* in *or* out*, by induction on i as follows. Base case: $i = 0$. By assumption, $e_0 = e$. Node n and every other node of e reachable from n is flagged* in*. Every other node of e is flagged* out*. Inductive case: $i > 0$. By assumption, $e_{i-1} \to_{n_{i-1}} e_i$, for some node n_{i-1}. By the induction hypothesis, every node of e_{i-1} is flagged. If a node m of e_i is also a node of e_{i-1}, then the flag of m in e_i is the same as the flag of m in e_{i-1}, i.e., the flag of every node is preserved by a step. Any node created*[3] *by the step $e_{i-1} \to_{n_{i-1}} e_i$ is flagged with the flag of n_{i-1}, the root of the redex replaced in the step. Finally, we say that a node m of an expression of A is* inside *s iff m is flagged* in*. We extend this notion of* inside *to a redex, if the redex is rooted by an inside node, and to a step, if the step replaces an inside redex. We say* outside *as a synonym of not inside.*

The notions of the above definition provided for an expression can be applied to a tuple of expressions, such as the arguments of an operation, because the tuple itself is an expression.

We are now ready for the key definition of our approach. This definition references a set which could be infinite. This set is an abstraction used in the context of rewriting, but it is not produced by a rewriting derivation. We will not implement or define a set function by rewrite rules. Programs in lazy functional and functional logic languages ordinarily compute with infinite structures, such as list, trees, and sets, as long as only a finite number of elements of these structure become explicit. The following definition only discusses whether a few elements belong to this potentially infinite set.

[3] These are the nodes introduced by the replacement [15, Def. 23]. Each such node is isomorphically mapped [15, Defs. 10-12] to a non-variable node of the right-hand side of the applied rule. See also [11, pag. 391].

DEFINITION 4. *Let $\mathcal{R}$ be a program and f an operation of $\mathcal{R}$. We call a new operation $f_{\mathcal{S}}$* set function of f *if it satisfies the following properties:*

1. *If $f \subseteq A_1 \times \cdots \times A_n \times B$, then $f_{\mathcal{S}} \subseteq A_1 \times \cdots \times A_n \times 2^B$, i.e., the operation $f_{\mathcal{S}}$ takes the same arguments as f and returns a set whose elements are values returned by f.*
2. *For any arguments $\bar{t}$ of f and value v, $f(\bar{t}) \xrightarrow{*} v$ iff there exists a set of values V such that V is a value of $f_{\mathcal{S}}(\bar{t})$ and $v \in V$.*
3. *For any arguments $\bar{t}$ of f and values v and w of $f(\bar{t})$, v and w belong to the same value of $f_{\mathcal{S}}(\bar{t})$ iff there exist an expression $\bar{c}$ derived from $\bar{t}$ such that any step of both $f(\bar{c}) \xrightarrow{+} v$ and $f(\bar{c}) \xrightarrow{+} w$ inside $\bar{c}$ replaces a deterministic redex.*

Display (2) is a very simple example showing all the components of the above definition. An implementation, which can be regarded as a constructive definition, of set functions is provided in Section 4. It is immediate to see that set functions are deterministic.

COROLLARY 1. *Let $\mathcal{R}$ be a program, f an operation of $\mathcal{R}$, and $f_{\mathcal{S}}$ a set function of f. Then $f_{\mathcal{S}}$ is a deterministic operation.*

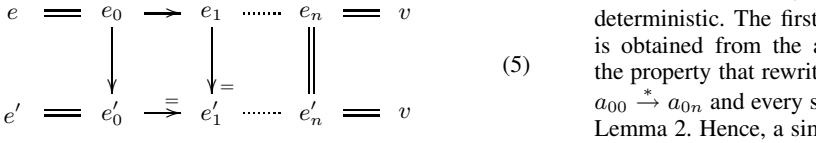
Proof: For any list of values $\bar{v}$, there is only one set of values of $f(\bar{v})$, hence only one value of $f_{\mathcal{S}}(\bar{v})$. □

To support infinite computations, Definition 4 is less direct than we would like. For a list of arguments $\bar{t}$, we are interested in $\mathcal{S}(f_{\mathcal{S}}(\bar{t}))$ which is an element of $2^{(2^B)}$. Definition 4 would be simpler and more direct if we could impose the condition that $\mathcal{S}(f_{\mathcal{S}}(\bar{t}))$ is a *finite* set of *finite* sets. Although the execution of a program evaluates only a finite portion of $\mathcal{S}(f_{\mathcal{S}}(\bar{t}))$, $C(f(\bar{t}))$ could be infinite and its traversal could generate an infinite set of infinite sets. Every value of $C(f(\bar{t}))$ is an element of some element of $\mathcal{S}(f_{\mathcal{S}}(\bar{t}))$. When a value v of $C(f(\bar{t}))$ is visited during a traversal *the device* must determine the element of $\mathcal{S}(f_{\mathcal{S}}(\bar{t}))$ to which v belongs. This element is determined only in relation to other values of $C(f(\bar{t}))$. Definition 4 gives us a criterion to tell whether two values of $C(f(\bar{t}))$ belong to the same element of $\mathcal{S}(f_{\mathcal{S}}(\bar{t}))$. This criterion may not appear operationally adequate, but with further work presented in Section 4 it will lead us to a provably correct implementation.

The above definition suggests to compute $f_{\mathcal{S}}(\bar{t})$ in two phases: first evaluate $\bar{t}$ to $\bar{c}$, then compute the set of values of $f(\bar{c})$. The second phase is relatively easy and, as we discussed, already available in most implementations. The first phase, however, is problematic. We do not know how far to evaluate $\bar{t}$ to find a suitable (in the sense of Definition 4) $\bar{c}$. Before resolving this problem, in Section 4, we present the crucial result of this paper: computing a set of values through a set function is order independent. We begin with a key lemma.

LEMMA 2. *Let $\mathcal{R}$ be a program, e an expression of $\mathcal{R}$, s a subexpression of e, v a value of e and $A : e \xrightarrow{*} v$ a derivation such that every redex replaced in A is both inside s and deterministic. If $e \to e'$ is a step inside s, then there exists a derivation $e \to e' \xrightarrow{*} v$ up to a renaming of nodes.*

Proof: Consider the following reduction diagram, where the top row and leftmost column of the diagram below represent the assumptions of the claim:



(5)

To complete the diagram, by induction on i, for $i = 1, \ldots n$, we:

1. define the horizontal step of e'_{i-1} and vertical step of e_i;
2. prove that either $e_i = e'_i$ up to a renaming of nodes or $e_i \to e'_i$ replaces a redex inside s;
3. prove that the diagram commutes at e_{i-1}.

Point (1): let i be an index in $\{1, \ldots n\}$. The horizontal step from e_{i-1} is defined by the assumption and the vertical step from e_{i-1} is defined either by the assumption (for $i = 1$) or by the induction hypothesis (for $i > 1$). We define the (horizontal) step from e'_{i-1} as the residual of the horizontal step from e_{i-1} by the vertical step from e_{i-1}. Likewise, we define the vertical step from e_i as the residual of the vertical step from e_{i-1} by the horizontal step from e_{i-1}.

Point (2): in graph rewriting, a node has exactly one residual (itself) by a step, unless it is erased. Let n_0 be the root node of the redex replaced in the step $e_0 \to e'_0$. If the step $e_{i-1} \to e_i$ erases n_0, then $e_i = e'_i$ up to a renaming of nodes. Otherwise, the step $e_i \to e'_i$ will replace the redex at n_0, which is inside s by assumption.

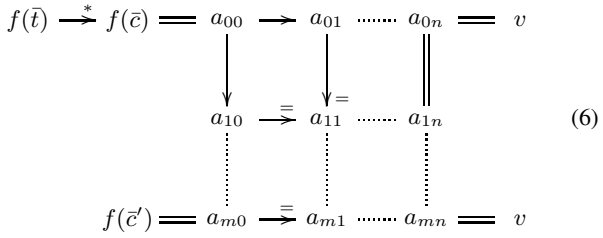
Point (3): All the graph equalities in this paragraph are intended up to a renaming of nodes. By point (2), if $e_{i-1} = e'_{i-1}$ then the steps at e_{i-1} and e'_{i-1} are the same, consequently $e_i = e'_i$ and the diagram commutes; otherwise let n_h and n_v be the roots of the redexes of the horizontal and vertical steps at e_{i-1}, where n_v is inside s. If $n_h \neq n_v$, then, by Lemma 1, the diagram commutes at e_{i-1}; otherwise $n_h = n_v$. By assumption every step of $e = e_0 \xrightarrow{*} e_n = v$ is deterministic. Hence, the two steps are applied to the same term and they replace the same redex with the same replacement. Thus, $e_i = e'_{i-1} = e'_i$ up to a renaming of nodes and the diagram trivially commutes in this case too.

Since the diagram commutes and e_n is a value, e'_n is the same value up to a renaming of nodes. Thus, the diagram constructs a derivation $e \to e' \xrightarrow{*} v$. □

The following result shows that "over-evaluating" the argument of a set function during an evaluation of some expression does not affect the result. Hence computing with set functions is order independent.

THEOREM 1. *Let $\mathcal{R}$ be a program, f an operation of $\mathcal{R}$, $\bar{t}$ a list of arguments of f, and v a value of $f(\bar{t})$. Let $\bar{c}$ be an expression derived from $\bar{t}$ such that the computation of $f(\bar{c})$ to v replaces only redexes that are inside $\bar{c}$ and deterministic. If $\bar{c}'$ is derived from $\bar{c}$, then there exists a derivation $f(\bar{c}') \xrightarrow{+} v$ such that each replaced redex is inside $\bar{c}'$ and deterministic.*

Proof: Consider the reduction diagram shown below.



(6)

The first row represents two assumptions of the claim: $\bar{t} \xrightarrow{*} \bar{c}$, hence $f(\bar{t}) \xrightarrow{*} f(\bar{c})$ since rewriting is closed under context, and $f(\bar{c}) \xrightarrow{+} v$. Furthermore, every step inside $\bar{c}$ of $f(\bar{c}) = a_{00} \xrightarrow{+} a_{0n} = v$ is deterministic. The first column, $f(\bar{c}) = a_{00} \xrightarrow{*} a_{m0} = f(\bar{c}')$, is obtained from the assumption that $\bar{c}'$ is derived from $\bar{c}$ and the property that rewriting is closed under context. The derivation $a_{00} \xrightarrow{*} a_{0n}$ and every step of $a_{00} \xrightarrow{*} a_{m0}$ satisfy the conditions of Lemma 2. Hence, a simple induction over the indexes of the rows proves the commutativity of the diagram. Since $a_{0n} = v$ is a value

and $a_{0n} \xrightarrow{*} a_{mn}$, $a_{mn} = v$ up to a renaming of nodes. Thus, the last row of the diagram proves that $f(\bar{c}') \xrightarrow{+} v$. Every redex inside $\bar{c}$ of this derivation is a descendant of a deterministic redex, hence it is deterministic as well. □

Our work discourages computing the set of values of an arbitrary expression because in some cases the result is affected by the order of evaluation. Therefore, it is interesting to ask whether it is still possible to cast the computation of the set of values of *any* expression in terms of some set function. The following simple result offers a positive answer.

LEMMA 3. *If $\mathcal{R}$ is a program and '$\mathtt{t}$' is a nullary operation (a possibly non-deterministic constant) of $\mathcal{R}$, then $\mathtt{t}_{\mathcal{S}} = \mathcal{S}(\mathtt{t})$.*

Proof: Vacuously, from Definition 4, since '$\mathtt{t}$' takes no argument. □

For example, to compute the set of values of '`f bigCoin`' it suffices to define:

```
t = f bigCoin
```

and to evaluate $\mathtt{t}_{\mathcal{S}}$. We remark that there is a substantial difference between evaluating $\mathtt{t}_{\mathcal{S}}$, as we are proposing in this paper, and evaluating $\mathcal{S}(\mathtt{f\ bigCoin})$. The result of the latter depends on the order of evaluation. If '`bigCoin`' is evaluated before $\mathcal{S}$, i.e., the traversal of the computation space of '`f bigCoin`' and the collection of the values in the leaves, the result is not intended. Obviously, this unintended evaluation cannot occur with $\mathtt{t}_{\mathcal{S}}$ since there is no argument to evaluate.

4. Implementation

Given a function f, it might be possible to define $f_{\mathcal{S}}$ within the language, i.e., with rewrite rules. However, we do not pursue this approach because it is not appealing for a programmer. It would be burdensome to code $f_{\mathcal{S}}$ and it could lead to errors because the mutual dependence between an operation and its set function is implicit. If f is modified during program maintenance, $f_{\mathcal{S}}$ must be modified accordingly. Therefore, it is preferable to let *the device* compile or interpret $f_{\mathcal{S}}$ from f automatically since all the information necessary to compute $f_{\mathcal{S}}$ is already in f.

Given a function f and an argument $\bar{t}$, we want to compute the values of $f_{\mathcal{S}}(\bar{t})$. Any such value is a set of values of $f(\bar{t})$. For a correct placement of the values of $f(\bar{t})$ into the values of $f_{\mathcal{S}}(\bar{t})$, we should evaluate $\bar{t}$ to some $\bar{c}$ according to Definition 4. Unfortunately, we do not know how much to evaluate $\bar{t}$. Thus, we take a different route. We evaluate $\bar{t}$ lazily, like any other expression of the program, and we keep track of the non-deterministic steps inside $\bar{t}$. We will use this information to ensure the intended meaning.

Let us assume that $f(\bar{t})$ has some values, otherwise the only value of $f_{\mathcal{S}}(\bar{t})$ is $\varnothing$. We start with a few preliminary concepts and an informal explanation. Given a set X and p and q subsets of X, we say that p and q are *comparable* iff one is contained in the other. We say that p and q are *incomparable* if they are not comparable. The comparability relation is an equivalence. In the computation space of $f(\bar{t})$ we associate to every leaf L_v containing a value v the set N_{L_v} of non-deterministic steps inside $\bar{t}$ executed to derive v. Let $N = \{N_{L_v} \mid L_v \text{ is a leaf of } C(f(\bar{t})) \text{ containing a value } v\}$, i.e., N contains the sets of non-deterministic steps inside $\bar{t}$ executed to compute all the values of $f(\bar{t})$. Let $[N]$ be the set of equivalence classes of N modulo the comparability relation. For each $[n] \in [N]$, we define $V_{[n]} = \{v \mid v \text{ is a value of } f(\bar{t}) \text{ and } N_{L_v} \in [n]\}$. V is a value of $f_{\mathcal{S}}(\bar{t})$ iff $V = V_{[n]}$, for some $[n] \in [N]$.

A program, during its execution, will not compute $[N]$ or the other concepts discussed in the previous paragraph. Typically, it will compute some portion of a set of values of some expression as demanded by a context. Suppose that some context applies some operation to $f_{\mathcal{S}}(\bar{t})$, e.g., to check whether the resulting set is empty, or some element belongs to it, or to compute its size. We have discussed that *the device* traverses the computation space of $f(\bar{t})$ as far as demanded by the context. When a value is visited, the only potentially difficult task is to decide whether to place it in a new value of $f_{\mathcal{S}}(\bar{t})$ or in some value of $f_{\mathcal{S}}(\bar{t})$ already partially computed.

EXAMPLE 1. *Consider the following program, where the symbols* `f` *and* `bigCoin` *are defined in (2):*

```
main = f_S bigCoin
```

Fig. 1 shows the computations space of '`main`*'. Symbols in the expressions of the computation space are decorated with two types of information. The subscript of a symbol s identifies the node labeled by s, e.g., the expression '*$\mathtt{main}_0$*' tells us that* 0 *is a node of this expression (the only one) and the label of node 0 is the symbol* `main`*. Nodes are identified by* $0, 1, 2, \ldots$ *Operations symbols flagged by $\bullet$ label nodes inside '*`bigCoin`*', the argument of the set function that we are considering. We flag these symbols to detect non-deterministic steps. To improve readability, we do not flag numbers even when they are inside '*`bigCoin`*', since numbers are never redexes and so they do not undergo any step whether or not non-deterministic. The leaves of the computation space of '*`main`*' contain elements of the values '*$\mathtt{f}_{\mathcal{S}}$ `bigCoin`*'. There are four such elements. The table to the right associates these elements to the non-deterministic steps inside '*`bigCoin`*' that were executed to produce them. It is simple to verify that '*$\mathtt{f}_{\mathcal{S}}$ `bigCoin`*', hence '*`main`*', has two values:* $\{2, 3\}$ *and* $\{4, 5\}$ *because* $N_{L_2} = N_{L_3} \neq N_{L_4} = N_{L_5}$.*

2_8	$\{ R_1$ *at* $?_9^\bullet \}$
4_{10}	$\{ R_2$ *at* $?_9^\bullet \}$
3_{11}	$\{ R_1$ *at* $?_9^\bullet \}$
5_{12}	$\{ R_2$ *at* $?_9^\bullet \}$

The computation space depicted in Fig. 1 shows the execution of two *steps at node $?_9^\bullet$. In our depiction of the computation space, the redexes of the two steps have the same root, but in practice a more sophisticated mechanism to identify the roots would be needed. Whether two steps are actually executed in the space of Fig. 1 depends on the implementation. Implementations that traverse the computation space depth-first with backtracking, such as [21, 25], indeed executes this step twice. These implementations are incomplete. A complete implementation may or may not execute two steps. The conceptually simplest approach, which is shown in Fig. 1, clones any expression that undergoes a non-deterministic step. In this case, two steps would be executed at distinct nodes that are clones of the same node. For the purpose of our discussion, these steps should be considered the same step. More sophisticated implementations, such as [8, 14], do not execute the same step twice, since no information would be gained by repeating a step. These implementations, for the example we are discussing, execute the step at node $?_9^\bullet$ only once and, therefore, they are not only more efficient, but also conceptually simpler and in line with our discussion. The details of concrete implementations cannot be discussed in this paper.*

The following simple lemma is instrumental for the proof of Theorem 2.

LEMMA 4. *Let $\mathcal{R}$ be a program, e an expression of $\mathcal{R}$, s a subexpression of e, and $e \to_o e_o \to_i e_i$ a 2-step derivation of e where the root o of the first step is outside s and the root i of the second step is inside s. There exists a 2-step derivation $e \to_i e'_i \to_o e'_o$ such that $e_i = e'_o$ up to a renaming of nodes.*

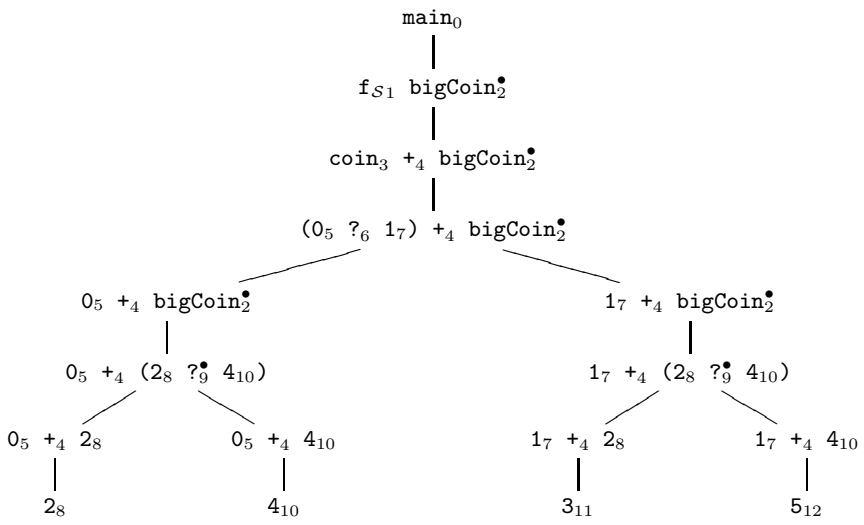


Figure 1. Computation space of '`main`'. The notation identifies the node of each symbol and tells whether the node is inside '`bigCoin`', the argument of '$\mathtt{f}_{\mathcal{S}}$'.

Proof: The nodes o and i are distinct, since one is outside s whereas the other is inside s. Furthermore, the step at i cannot erase the redex at o, since, by Definition 3, o is not reachable from i and, hence, it cannot be matched by variables that may be erased by the step at i. By Lemma 1, $\mathcal{R}$ is deterministically confluent. Hence, by Definition 1, the two steps commute. □

DEFINITION 5. *Let $\mathcal{R}$ be a program, f an operation of $\mathcal{R}$, $\bar{t}$ an argument of f, and L_v a leaf of $C(f(\bar{t}))$ containing a value v. We call* fingerprint *of L_v the set of non-deterministic steps inside $\bar{t}$ in the path of $C(f(\bar{t}))$ from the root to L_v.*

The next theorem states the correctness of an implementation that uses fingerprints to compute set functions. Informally, let f be an operation, $\bar{t}$ an argument (tuple) of f. Each value of $f(\bar{t})$ must be an element of some set computed by $f_{\mathcal{S}}(\bar{t})$. When are two distinct values of $f(\bar{t})$ elements of a same set computed by $f_{\mathcal{S}}(\bar{t})$? When they have comparable fingerprints. The converse does not hold because a value of $f(\bar{t})$ may occur in several leaves of the computation space of $f(\bar{t})$ and two different leaves containing the same value may have either comparable or incomparable fingerprints.

THEOREM 2. *Let $\mathcal{R}$ be a program, f an operation of $\mathcal{R}$, $f_{\mathcal{S}}$ a set function of f, $\bar{t}$ an argument of f, and L_v and L_w leaves of $C(f(\bar{t}))$ containing values v and w, respectively. If the fingerprints of L_v and L_w are comparable, then there exists a value V of $f_{\mathcal{S}}(\bar{t})$ such that $v, w \in V$.*

Proof: Define a function $W : S \to \mathbb{N}$, where S is the set of the finite derivations of $f(\bar{t})$, as follows. If D is a derivation in S executing exactly n steps (numbered $1, 2, \ldots n$), then $W(D) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} w_{ij}$, where $w_{ij} = 1$ if the i-th step of D is outside $\bar{t}$ and the j-th step of D is inside $\bar{t}$; otherwise $w_{ij} = 0$. Informally, $W(D)$ counts the pairs of steps (i, j) of D, with $i < j$, such that step i is outside $\bar{t}$ and step j is inside $\bar{t}$. Let N_{L_v} and N_{L_w} be the fingerprints of L_v and L_w, respectively. Since the fingerprints are comparable, without loss of generality we assume that $N_{L_w} \subseteq N_{L_v}$. Since L_v is a leaf of $C(f(\bar{t}))$, there exists a derivation $A : f(\bar{t}) \xrightarrow{+} v$. We prove by induction on $W(A)$ that there exists an expression $\bar{c}$ derived from $\bar{t}$ such that there exists a derivation $B : f(\bar{t}) \xrightarrow{*} f(\bar{c}) \xrightarrow{+} v$ where no step of the portion $f(\bar{c}) \xrightarrow{+} v$ of B is inside $\bar{c}$. Informally, B brings to the front all the steps of A inside $\bar{t}$. Base case: $W(A) = 0$. By the definition of W, $W(A) = 0$ implies that every step of A inside $\bar{t}$ precedes every step of A outside $\bar{t}$. Since $f(\bar{t})$ is operation rooted, A executes at least one step at the root, and this step is outside $\bar{t}$. Let $k \geqslant 1$ be the index of the first step of A outside $\bar{t}$. The first $k-1$ steps of A are inside $\bar{t}$. Hence A has the form $f(\bar{t}) \xrightarrow{k-1} f(\bar{c}) \xrightarrow{+} v$, for some $\bar{c}$ derived from $\bar{t}$, and $B = A$ witnesses the claim. Inductive case: $W(A) > 0$. By the definition of W, $W(A) > 0$ implies that some step of A inside $\bar{t}$ follows some step of A outside $\bar{t}$. Hence, there exists an index $k \geqslant 1$ such that the k-th step of A is outside $\bar{t}$ and the $k+1$-th step of A is inside $\bar{t}$. By Lemma 4, swapping these steps produces a new derivation $A' : f(\bar{t}) \xrightarrow{+} v$ such that $W(A') = W(A) - 1$. By the induction hypothesis, the claim holds for A', hence it holds for A as well. In particular, starting from A, by swapping $W(A)$ adjacent steps such that the first is inside $\bar{t}$ and the second is outside $\bar{t}$ we obtain the desired derivation B. To complete the proof, we show the existence of a derivation $C : f(\bar{t}) \xrightarrow{*} f(\bar{c}) \xrightarrow{+} w$ such that every step of C in the portion $f(\bar{c}) \xrightarrow{+} w$ is deterministic. Consider the following diagram:

$$\begin{array}{ccccccc} f(\bar{t}) & = & a_{00} & \to & a_{01} & \cdots & a_{0n} & = & w \\ & & \downarrow & & \downarrow{=} & & \| & & \\ & & a_{10} & \overset{=}{\to} & a_{11} & \cdots & a_{1n} & & \\ & & \vdots & & \vdots & & \vdots & & \\ f(\bar{c}) & = & a_{m0} & \overset{=}{\to} & a_{m1} & \cdots & a_{mn} & & \end{array} \tag{7}$$

The top row is the derivation in $C(f(\bar{t}))$ from the root to leaf L_w which is assumed by the claim. The leftmost column is the portion of a derivation B, whose existence has just been proved, which derives $f(\bar{t})$ into $f(\bar{c})$. The remaining entries of the diagram are defined by induction exactly as in diagram (5) point 1. To prove that the diagram commutes at a_{ij}, for $0 \leqslant i < m$ and $0 \leqslant j < n$, let n_h and n_v be the roots of the redexes in the horizontal and vertical, respectively, steps at a_{ij}. If n_h and n_v are distinct, then the commutativity of these steps is a consequence of the deterministic confluence of the program. If $n_h = n_v$, then if the redex is deterministic, the steps are the same and obviously they

commute; otherwise the redex is non-deterministic. Remember that by construction n_v is inside $\bar{t}$, hence in the fingerprint of N_{L_v} and that $N_{L_w} \subseteq N_{L_v}$. Hence, the steps are the same in this case too, and they commute in all cases. Now we prove that every step inside $\bar{c}$ of the bottom row is deterministic. For $i = 0, 1, \ldots n-1$, the step $a_{mi} \to a_{m(i+1)}$ is the descendant by $a_{00} \xrightarrow{*} a_{m0}$ of $a_{0i} \to a_{0(i+1)}$. If the latter step is non-deterministic, it is in the fingerprint of N_{L_w} which is contained in N_{L_v}, hence it is executed in $a_{00} \xrightarrow{*} a_{m0}$ because by construction this derivation executed all the steps in the fingerprint of N_{L_v}. This implies that $a_{mi} = a_{m(i+1)}$ up to a renaming of nodes. Thus, there are no non-deterministic steps inside $\bar{c}$ in the bottom row of the diagram. Finally, by construction $a_{0n} = w$. Since w is a value, for $i = 0, 1, \ldots m$, $a_{in} = w$ up to a renaming of nodes. Thus, by concatenating the leftmost column and the bottom row, we have a derivation $f(\bar{t}) \xrightarrow{*} f(\bar{c}) \xrightarrow{+} w$ such that every redex inside $\bar{c}$ replaced in the portion $f(\bar{c}) \xrightarrow{+} w$ is deterministic. By Definition 4, this proves that v and w belong to some $V = f_{\mathcal{S}}(\bar{t})$. □

A reader concerned with an actual implementation may wonder how a fingerprint could be represented. Conceptually, a fingerprint is a set of non-deterministic steps. Because our programs are *LOIS* systems, a step is non-deterministic iff it is applied to a node labeled by the *choice* operation. Thus, a fingerprint is a set of a pairs $\langle n, r \rangle$, where n is a node identifier and r is either one of the two rules of '?' displayed in (1). In some implementations, a node is conveniently identified by the memory location of its representation.

5. Programming

We apply the approach presented in the previous sections to a problem. The problem is solving the n-queens puzzle: place n queens on an $n \times n$ board so that no queen captures any other queen according to the rules of chess. This is a classic example of problem solving by *backtracking* in an imperative language [30]. Of course, no backtracking is coded in a functional logic language.

A typical implementation of this problem represents a placement of queens on the board as a permutation of $1, 2, \ldots n$, where the i-th element of the permutation is the row of the queen placed in column i. Representing a placement as a permutation pays off to determine whether a placement is safe since it makes it impossible to place two queens in the same row and/or the same column. Thus, to determine whether two queens capture each other it suffices to determine whether they are on a same diagonal. This condition occurs only if the distance between two queens' rows is the same as the distance between the two queens' columns. This condition must be tested for any set of two queens on the board.

The program adopts a generate-and-test algorithm. The program generates (lazily) every permutation of $1, 2, \ldots n$ and tests whether it represents a safe placement of the queens. Coding the generation of every set of two queens and the control to test whether they capture each other is laborious. Since functional logic languages have built-in searching capability, we code an operation that takes a placement as its argument and searches two queens in the placement that captures each other. If two such queens are found, the placement is discarded since it is *not* a solution of the problem.

In the following code, `abs` returns the absolute value of an integer, `length` the length of a list, '`[1..n]`' the list $1, 2, \ldots n$, and `permute` a permutation of its argument. The program is executed by narrowing, which was not considered in our treatment, but conceptually narrowing can be replaced by rewriting if generator functions [7] are used instead of variables. Some compiler/interpreters, e.g., KICS [14], indeed translate source code with variables into executable code without variables so that variables are considered as syntactic sugar for generator functions.

```
unsafe (_++[x]++y++[z]++_)
   = abs (x-z) =:= length y + 1
```

The argument is matched against the concatenation of five lists (here we use the convenient notation of functional pattern [6]), two of which contain a single element. Each such element stands for the row of a queen. The list between these elements represents the portion of the placement between the two queens. Hence the right-hand side succeeds iff the number of columns between the two queens is the same as the number of rows between the two queens

Now, a placement p is a solution of the puzzle iff '`unsafe` p' fails, i.e., there is no pair of queens of p capturing each other. Using the concepts of the previous sections, this is equivalent to '$\texttt{unsafe}_{\mathcal{S}}\ p$' $= \varnothing$. Thus, our program is:

```
queens n | isEmpty (unsafeS p) = p
         where p = permute [1..n]                    (8)
```

All current implementations of the functional logic language Curry [20], e.g., [14, 21, 25], cannot execute the above code because they lack set functions. A published implementation of this problem with the same algorithm [9] replaces $\texttt{unsafe}_{\mathcal{S}}$ with `unsafe`. This could (likely would) produce an unintended result depending on the order of evaluation. The reason is that the execution of the program invokes `unsafe` with a non-deterministic argument, but the programmer intention is to exclude the non-determinism of this argument from the evaluation of `unsafe`. The program in [9] produces the correct result because it forces the complete evaluation of `p` before computing $\mathcal{S}$(`unsafe p`) to separate the non-determinism of `p` from that of `unsafe`. This programming technique is quite undesirable in a declarative language since it requires the programmer to think about the order of evaluation.

As a final example, consider the definition of `itinerary` given in Section 1. Using set functions, we can provide the following executable definition of an itinerary with shortest air time: an itinerary, '`it`', is the shortest one if there is no itinerary shorter than '`it`':

```
shortestItin s e
  | isEmpty (shorterItinThanS (duration it))
  = it
  where it = itinerary s e
        shorterItinThan itduration
           | duration its < itduration
           = its
           where its = itinerary s e
```

where '`duration`' is a trivial function that computes the total time in the air of an itinerary.

Note that the distinction between the different sets of values (e.g., to compute all values of `it` and all values of `its`) is essential to compute the intended results. The above code directly reflects the formal definition of a shortest itinerary, but it is highly inefficient from an operational point of view since for each itinerary we have to compute other itineraries to check the "shortest" property. Another alternative is to compute all the itineraries and select the shortest by comparing pairwise the durations of itineraries. In the following code, '`minSet`' computes the minimal element of a set w.r.t. a total ordering provided as the first argument:

```
shortestItin s e
   = minSet shorter (itineraryS s e)
   where
```

```
shorter it1 it2
   = duration it1 <= duration it2
```

6. Related work

A satisfactory approach to computing with sets of values in a demand-driven non-deterministic functional logic language such as Curry [20] has been elusive. This situation prompted, over a decade, several proposals that we briefly review in this section. Let $\mathcal{S}(e)$ be the set of values of some expression e. A difficulty of computations involving $\mathcal{S}(e)$ originates from subexpressions of e that are needed both by $\mathcal{S}(e)$ and independently of $\mathcal{S}(e)$. An expression of this kind is said to be shared inside and outside the set. In (8), the expression '`p`' is shared in this sense because it is returned by the operation '`queens`' (outside $\mathcal{S}(\mathtt{p})$) and it is needed (inside $\mathcal{S}(\mathtt{p})$) as the argument of '$\mathtt{unsafe}_{\mathcal{S}}$'. Every approach proposed to date—except ours—computes all the values of an expression inside the set, but only one value outside the set.

One of the first approaches [19] was intended to control the traversal of the computation space of an expression. This approach is based on a primitive operator, called '`try`', that evaluates an expression until a non-deterministic step is executed. The '`try`' allows to easily program different search strategies [29]. This approach is based on term rewriting and, thus, does not easily accommodate sharing in general and sharing inside and outside a set of values in particular. Lux [26] presents a sophisticated implementation of the '`try`' operator and discusses the problem of evaluating subexpressions shared inside and outside the set of values. He proposes a "weak encapsulation" view where sharing inside and outside of sets of values is preserved. As a consequence, non-deterministic steps on subexpressions shared inside and outside a set are never performed inside the set. Similarly to our approach, this leads to the intended behavior in the queens example (8). In contrast to our approach, the set of subexpressions covered by the '`try`' operator depends on the concrete syntactic structure of expressions so that apparently equal expressions yield different results (see [12] for details). No formal properties are stated for his approach.

[12] presents a detailed discussion of various options and pitfalls to computing set of values in the presence of sharing. The authors propose a "strong encapsulation" view where sharing inside and outside sets is severed and the search primitive is put into the I/O monad. The latter restriction is relaxed in [13] where a demand-driven implementation of the search primitive is presented. Since the implementation is based on the strong encapsulation view, it does not support a distinction between different levels of non-determinism, i.e., non-determinism caused by arguments vs. non-determinism caused by operations. A consequence of this design decision is that in various situations, e.g., that shown in Section 5, unintended results are produced.

[5] presents a formal abstract treatment of computations with sets of values and proposes another primitive, '`getAllValues`', to compute all the values of an expression. Although some interesting properties are stated for this primitive, the authors acknowledge a weakness of this approach due to the fact that some applications demand the sharing of non-deterministic expressions inside and outside a set of values. Our new approach solves this problem by introducing set functions that allow a clear distinction between different sources of non-determinism in a computation.

The computation of sets of values has been also considered in works related to computing with failures in functional logic programming [24, 28]. In these works, an operation *fails* is proposed that computes the set of all values of its argument expression and returns true, if this set is empty, or false, otherwise. The semantics is defined by an extension of the rewriting calculus CRWL [17] and implementations are given by computing the set of all values of an expression. Similarly, our approach enables such an implementation of constructive negation, as shown by various examples above. In contrast to the works on constructive failure, we also support the further processing of the collected sets (as shown in the final example of Section 5) in order to extract information from them.

7. Conclusion

We presented an approach to computing the set of values of an expression. The key contribution of our approach is the form of this expression, a set function—a novel concept introduced by this work—applied to some argument(s). The form of this expression allows the separation of the non-determinism of a function from that of its argument(s). This separation ensures the order independence of our approach which we proved within the framework of graph rewriting.

We defined an abstract implementation of our approach based on fingerprints—a novel concept introduced by this work—and we proved the correctness of this implementation. We showed, through a textbook example, that our approach easily and naturally solves problems that previous approaches could not solve or could solve only with some tricks.

The independence of the order of evaluation is an essential property of declarative languages. Consequently, our approach positively solves a long-standing problem. For this reason, we believe that our approach will eventually replace all the previously proposed approaches found in current implementations of functional logic languages.

References

[1] 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.

[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, Florence, Italy, 2001. ACM Press.

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

[5] S. Antoy and B. Braßel. Computing with subspaces. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN international conference on principles and practice of declarative programming*, pages 121–130, New York, NY, USA, 2007. ACM.

[6] S. Antoy and M. Hanus. Declarative programming with function patterns. In *15th Int'nl Symp. on Logic-based Program Synthesis and Transformation (LOPSTR 2005)*, pages 6–22, London, UK, September 2005. Springer LNCS 3901.

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

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

[9] Sergio Antoy. Evaluation strategies for functional logic programming. In Bernhard Gramlich and Salvador Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 57. Elsevier Science Publishers, 2001.

[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):1–28, 2004.

[13] B. Braßel and F. Huch. On a tighter integration of functional and logic programming. In *Proc. APLAS 2007*, pages 122–138. Springer LNCS 4807, 2007.

[14] B. Brassel and F. Huch. The Kiel Curry System KiCS. In Dietmar Seipel, Michael Hanus, and Armin Wolf, editors, *Applications of Declarative Programming and Knowledge Management*, pages 195–205, Springer LNAI 5437, 2009.

[15] R. Echahed and J.-C. Janodet. On constructor-based graph rewriting systems. Technical Report 985-I, IMAG, 1997. Available at `http://www-lsr.imag.fr/Les.Publications/c-graph-rewriting.ps`.

[16] Rachid Echahed. Inductively sequential term-graph rewrite systems. In *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom*, volume 5214 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2008.

[17] 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.

[18] M. Hanus, S. Lucas, and A. Middeldorp. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters*, 67(1):1–8, 1998.

[19] M. Hanus and F. Steiner. Controlling search in declarative programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pages 374–390. Springer LNCS 1490, 1998.

[20] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.

[21] M. Hanus (ed.). PAKCS 1.9.1: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs`, 2008.

[22] 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.

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

[24] F.J. López-Fraguas and J. Sánchez-Hernández. A proof theoretic approach to failure in functional logic programming. *Theory and Practice of Logic Programming*, 4(1):41–74, 2004.

[25] 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.

[26] W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji Intl. Symposium on Functional and Logic Programming (FLOPS '99)*, pages 100–113. Springer LNCS 1722, 1999.

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

[28] J. Sánchez-Hernández. Constructive failure in functional-logic programming: From theory to implementation. *Journal of Universal Computer Science*, 12(11):1574–1593, 2006.

[29] C. Schulte and G. Smolka. Encapsulated search for higher-order concurrent constraint programming. In *Proc. of the 1994 International Logic Programming Symposium*, pages 505–520. MIT Press, 1994.

[30] N. Wirth. *Algorithms and Data Structures*. Prentice Hall, 1976.