Representing Functional Logic Computations*

Sergio Antoy[1] Michael Hanus[2] Sunita Marathe[1]

[1] Computer Science Department, Portland State University,
P.O. Box 751, Portland, OR 97207, U.S.A.
{antoy,marathes}@cs.pdx.edu

[2] Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany.
mh@informatik.uni-kiel.de

Abstract. The execution of declarative programs is largely independent of concrete execution strategies. For instance, functional logic languages support various strategies to execute programs, e.g., sequential, parallel, concurrent, fair, depth-first etc. This freedom causes difficulties for tools intended to visualize, trace, or debug functional logic computations. To improve this situation, we describe structures for representing a functional logic computation independently of its concrete execution strategy. Our representation serves as an interface to connect different implementations of functional logic languages, such as PAKCS, KiCS, MCC, FLVM, or TOY, to current and future tools, such as tracers or debuggers, for visualizing and understanding computations.

1 Motivation

One of the main ideas of declarative programming is to separate the logic description of a problem from the control necessary to solve it ("Algorithm = Logic + Control"). Purely logic languages do not fully support this idea and often provide non-declarative features for application programming (e.g., I/O operations with side effects). By contrast, functional logic languages (see [16] for a recent survey) like Mercury [25] or Curry [15, 18] exploit functional programming techniques (e.g., monadic I/O [26]) to better support independence of the evaluation strategy. Thus, functional logic languages support various controls to execute programs, e.g., sequential [4], parallel [3], concurrent [15], depth-first search [5], fair search [7], etc. Actually, there are implementations that allow the execution of the same functional logic programs with different strategies (e.g., KiCS [11]).

In general, an *evaluation strategy* is a crucial component of every implementation of a functional logic programming language. The strategy determines the steps of a computation. Typically, executing all the possible steps of an expression is an unnecessary and inefficient operation. In most cases, only a small number of steps must be executed to obtain every result of a given expression, an essential property of computations and implementations referred to as *completeness*. An ideal situation, known as *sequentiality*, is when for any expression there exists a single step whose execution ensures the

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

completeness of computations. Even when an expression has several necessary steps, it may suffice to execute only one of these steps and to ignore the remaining ones, perhaps without even computing them, because the ignored steps will be eventually computed later. Some programs require a non-sequential control to ensure the completeness of some computations, but sequentiality contributes to the simplicity and efficiency of an implementation to such a degree that sometimes it is adopted even if the completeness is lost.

There are very few evaluation strategies for functional logic computations, and they are precisely defined. However, the needs of and/or the opportunities presented by an implementation, as discussed in the previous paragraph, disconnect the practice from the theory. When the strategy computes several steps on an expression, different implementations of functional logic languages may vary in the order in which these steps are applied to the expression. This variation is often referred to as *control*. For instance, PAKCS [17], TOY [20], and MCC [21] always select a single needed step in the expression being evaluated and use backtracking to accommodate non-deterministic computations.[1] KiCS [11] and FLVM [7] use more sophisticated strategies to deal with non-determinism. In particular, they support sharing over non-determinism. Subexpressions common to different non-deterministic branches may be evaluated only once.

Example 1. Consider the following rewrite rules (using the notation of Curry) defining an operation `head` that returns the first element of a list, a non-terminating operation `self`, and an operation `coin` that non-deterministically returns `0` or `1`:

```
head (x:xs) = x          coin = 0
self = self              coin = 1
```

Now, consider the expression `coin+head(2*3:self)` ("+" and "$\star$" are primitive operations defined elsewhere). An eager or innermost evaluation is incomplete since it evaluates the subexpression `self` forever. A lazy evaluation that evaluates the arguments of "+" from left to right reduces the whole expression non-deterministically to `0+head(2*3:self)` or `1+head(2*3:self)`. Thus, the subexpression `head(2*3:self)` will be evaluated twice during the rest of the computation of the non-deterministic branches of the search space. However, more sophisticated strategies, as implemented in KiCS or FLVM, evaluate this subexpression only once and share the result with the other non-deterministic branch, since the value of this subexpression does not depend on the chosen alternative.

Although the differences in the evaluation strategies support efficient and/or complete implementations of functional logic languages, the details of the evaluation are difficult to comprehend for the programmer. In particular, if a programmer wishes to browse through a computation for the purpose of debugging, the steps of the computation should be presented in an order which is more closely related to the program text than to the idiosyncrasies of a particular implementation of the evaluation strategy. This fact, well-known for lazy functional languages, becomes even more relevant for modern functional logic languages which extend the functional computation model.

[1] In addition, MCC also implements encapsulated search to provide more sophisticated search strategies.

The difficulty of understanding a computation through the sequence of its steps prompted the development of various debugging tools for specific implementations of declarative languages. To ease the development of such tools, and to make them portable across various language implementations, we propose both an abstract representation of functional logic computations and a protocol to support the incremental construction and transmission of this representation. Our work enables different language implementations to generate a protocol that can be used by any protocol-aware tool to visualize a functional logic computation in a form which is more natural and easier to understand than the sequence of steps in the order in which they are executed.

2 Computation Spaces

This section introduces the notion of computation space, an abstraction of the steps and non-deterministic choices performed during a functional logic computation.

We formalize a functional logic program with a constructor-based graph rewriting system [14, 22]. An "expression" to be evaluated according to a program is therefore a graph. A *graph* is defined by a set of *nodes*, a distinguished node called the *root* and two functions on nodes, (1) a *labeling* function that associates a symbol of the signature or a variable to a node, and (2) a *successor* function that associates a sequence of nodes to a node. For any variable v, there is at most one node labeled by v. To use a familiar terminology, a graph will be also called a *term* or *expression* and a node a *position*. The notions of *redex*, *replacement* and *rewrite* are more laborious to present for graphs than for terms, but are substantially similar. [14] contains an in-depth treatment of graph rewriting which is well suited to this paper. A *functional logic computation* consists of a sequence of rewrites of one expression e into another by applying a rewrite rule of the program at some subexpression of e.

We allow non-deterministic rewrites, or in other words, redexes that have distinct replacements, but we ignore logic variables and the bindings that would arise from narrowing. This is a simplification, not a restriction, since logic variables can be replaced by non-deterministic generator functions [6]. We allow parallel rewrites to keep our discussion independent of any particular functional logic machine and to accommodate future implementations. A *step* is a pair of terms denoted $t \to u$ such that t rewrites to u. When appropriate, we will decorate a step with the position of the redex and/or the rule applied to the redex. Consider Example 1 and the following step that replaces (with itself) the first component of a pair:

```
(self,self) →1 (self,self)
```

In this particular example, without decoration it would be impossible to tell what is replaced in the step. A step is *non-deterministic* when the step's redex admits two or more replacements, i.e., when more than one rule can be applied to this redex. A *parallel step* is a pair of terms denoted $t \to u$ such that $t = t_0 \to_{p_1} t_1 \to_{p_2} \cdots \to_{p_n} t_n = u$ in which $P = \{p_1, \ldots p_n\}$ is a set of distinct positions. Since programs are constructor-based, the redex patterns at distinct positions of a term do not overlap. This condition ensures that the order of the elements in P does not affect u. Therefore, a parallel step is well-defined without specifying the order of the replacements, a fact that can be proved as in the *Parallel Moves Lemma* [19, Sect. 2]. Obviously, a parallel step generalizes a

step and a step is a special case of a parallel step. In the rest of this paper, the word "step" will refer to a step that may be parallel and/or non-deterministic.

A *strategy* is a mapping $\mathcal{S}$ from terms into sets of terms such that if t is a term and u is in $\mathcal{S}(t)$, then $t \to u$. Simply put, a strategy tells which steps to execute on a term. Good strategies are minimalistic in the sense that they compute more than one step only for non-deterministic redexes.

Definition 1 (Minimalistic strategy). *We say that a strategy $\mathcal{S}$ is* minimalistic *if and only if the following two conditions hold, where t is any term. (1) If u_1 and u_2 are distinct terms in $\mathcal{S}(t)$, where $t \to_{P_1} u_1$ and $t \to_{P_2} u_2$, then there exists a position p such that $p \in P_1 \cap P_2$ and the subterm of t at p is a non-deterministic redex. (2) Conversely, if $\mathcal{S}(t) = \{u\}$, where $t \to_P u$, then for all p in P, the subterm of t at p is deterministic.*

A *computation* of a term t w.r.t. a strategy $\mathcal{S}$ is a finite or infinite sequence of steps $t_i \to u_i$, $i \geqslant 0$, such that $u_i \in \mathcal{S}(t_i)$ for all i, and $t_i = u_{i-1}$ for $i > 0$. The *computation space* of a term t w.r.t. a strategy $\mathcal{S}$, denoted $\textsc{Space}(t)$, is a (possibly infinite) tree-like structure representing every computation (according to $\mathcal{S}$) originating from t. More precisely, for any t, if $\mathcal{S}(t) = \{u_1, \ldots u_n\}$, then $\textsc{Space}(t) = (t, \{\textsc{Space}(u_1), \ldots, \textsc{Space}(u_n)\})$.

For a minimalistic strategy, a branch in a computation space abstracts a term that undergoes a non-deterministic step. The computation space is an unordered tree, i.e., the order of the children of a node is not specified, since there is no natural order among the various replacements of a non-deterministic redex. A computation space has similarities to proof trees w.r.t. operational semantics of functional logic programs. A computation space is an abstract, declarative concept. The execution of a program generates or traverses, partially or entirely, the computation space of a term in an attempt to produce the term's values. The order in which the program generates or visits the nodes of a space is not perceptible in our model. This is intended, since we want to abstract from an evaluation order that might be difficult to understand for a programmer.

3 Communication Protocol

A computation space is an abstraction for reasoning about computations. It is not intended to be explicitly stored or generated during the execution of a program. To transmit and manipulate a computation space, we introduce a communication protocol for the space itself. Loosely speaking, a protocol contains the same information as a computation space. The protocol is meant to be generated on-the-fly during the execution of program and serialized over a stream that, e.g., can be stored on some medium or consumed by a protocol-aware tool. By contrast to a computation space that consists of a monolithic, typically large and possibly infinite piece of information, the protocol consists of elements that refer to more localized information and are easy to serialize. This eases producing and exchanging the protocol.

Basically, a machine for functional logic computations (subsequently abbreviated "FLM") has to report the following information via the protocol:

- construction of terms (e.g., instances of right-hand sides of applied rules)

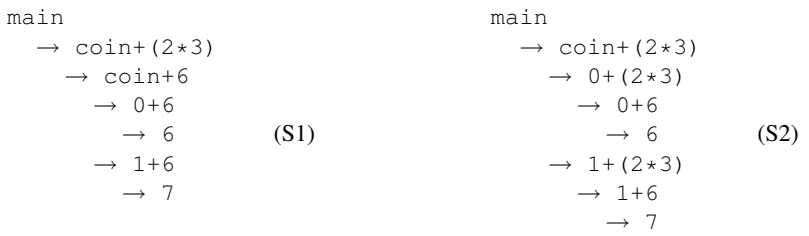

Fig. 1. Example computation spaces

- application of rewrite rules
- creation of computation branches

An important issue about the protocol is the fact that it makes no assumption about the order in which the alternatives of a non-deterministic step are executed. This condition, e.g., makes backtracking indistinguishable from breadth-first search. Thus, it must be clear to which branch of a computation a step belongs. For instance, implementations supporting sharing over non-determinism, like KiCS or FLVM, might perform a rewrite step *after* a branch of a computation space has been created. If a redex of this step is shared by another branch, the same redex and step can be perceived as duplicated in the other branch or as performed *before* the branch was created. For instance, consider the non-deterministic function `coin` defined in Example 1 and the initial expression defined by the function

```
main = coin+(2*3)
```

Depending on the order in which the arguments of "+" are evaluated, we have structurally different computation spaces. For instance, if the right argument is evaluated first, we obtain computation space (S1) of Fig. 1 (where we align all children of a node vertically and omit the labeling of the steps). If the left argument is evaluated first, we obtain computation space (S2) of Fig. 1. Although both computation spaces lead to the same results, their structures are different. Note that simple implementations, e.g., those based on backtracking, perform all the individual steps shown in (S2) if the arguments of "+" are evaluated from left to right. More advanced implementations, like KiCS or FLVM, detect that the evaluation of the subterm `(2*3)` is independent of the branch chosen by `coin`. Thus, the evaluation of `(2*3)` is shared over the non-deterministic branches caused by `coin`, i.e., the two rewrite steps of `(2*3)` to `6` occurring in (S2) correspond to the execution of a single step in the implementation. Although such sharing over non-determinism can have a dramatic impact on the efficiency of functional logic computations, the details are difficult to comprehend so that a representation of the overall computation as a tree structure as in (S2) is quite adequate, in particular, for abstract views supported by declarative debuggers.

To reconstruct a computation space from the steps of a computation, we attach to each step the information of the branch it belongs to. We call *segment* of a computation space a sequence of steps of maximal length which ends in either a leaf or a branch node of the computation space, and in which any other node is not a branch. Each segment has a unique identifier, e.g., a natural number. The first segment of a computation

space originates at the root of the computation space, which abstracts the initial term of a computation. Any other segment originates at a child u of some branch node abstracting a term t. When one such segment is introduced, the protocol must provide the information about the segment of t. In this situation, we say that the segment of u *evolves* from the segment of t.

Another subtle point in the reconstruction of computation spaces is the distinction between sequential and parallel rewrite steps. For example, consider the term $t =$ `(2*3,4+5)`. A sequential strategy might reduce `2*3` in a first step followed by a rewrite step on `4+5`. A parallel strategy could reduce both expressions in one parallel step. Although this difference is not relevant for the computed result, our goal is to reconstruct the correct computation space according to the strategy performed by an FLM so that one can visualize the underlying computation in a high-level manner. Therefore, the protocol groups together reductions that conceptually belong to a same step according to a strategy (an alternative would be to add a unique identifier to all those steps).

Altogether, the protocol has the following basic structure (in practice, one could refine it to distinguish different kinds of terms, e.g., operations, constructors, numbers):

$$
\begin{array}{rll}
b & \in & Int \quad (\textit{segment identifier}) \\
n & \in & Int \quad (\textit{node identifier}) \\
p & ::= & pe^* \quad (\textit{protocol}) \\
tc & ::= & T(n, f, n^*) \quad (\textit{term construction}) \\
rp & ::= & R(b, n_1, n_2) \quad (\textit{term replacement}) \\
pe & ::= & tc \quad (\textit{protocol elements}) \\
 & | & I(b, n) \quad (\textit{initialization}) \\
 & | & B(b_1, b_2) \quad (\textit{segment creation}) \\
 & | & S \cdot (tc \mid rp)^* \cdot E \quad (\textit{replacements})
\end{array}
$$

According to the definition of graph presented earlier, a graph (term) is represented by $T(n, f, n_1 \ldots n_k)$ where n is the root node, f is the label of n (which can be an operation symbol, a constructor symbol, or a literal), and $n_1 \ldots n_k$ are the successors of n. Nodes have identifiers, e.g., natural numbers. For instance, Fig. 2(a) shows a protocol defining the term `coin+(2*3)` rooted at node 14.[2]

A protocol is initialized by the element $I(b, n)$, where b is the identifier of the first segment and n is the root node of the initial term. $B(b_1, b_2)$ introduces a new segment identifier b_2 that evolves from the segment identified by b_1. An element $R(b, n_1, n_2)$ denotes the step of the term rooted by n which replaces the redex at node n_1 by the replacement at node n_2 in segment b. Note that the same redex can be replaced by a different term in a different segment. Finally, S and E enclose parallel computation steps. Figure 2(b) shows the protocol of the first 3 steps of the computation space (S1).

The protocol leaves some freedom in the order of reduction steps and segments which is important for sophisticated implementations. For instance, the first 3 steps of the computation space (S2) could be represented by the protocol shown in Fig. 2(c).

[2] When we use sequences in this paper, Λ denotes the empty sequence and $s_1 \cdot s_2$ denotes the concatenation of the sequences s_1 and s_2. Moreover, sequences containing only one element are identified with this element.

	$I(0, 9)$	$I(0, 9)$
	$T(9, \texttt{main}, \Lambda)$	$T(9, \texttt{main}, \Lambda)$
	S	S
	$R(0, 9, 14)$	$R(0, 9, 14)$
	$<$ insert here Fig. 2(a) $>$	$<$ insert here Fig. 2(a) $>$
$T(10, \texttt{coin}, \Lambda)$	E	E
$T(11, \texttt{2}, \Lambda)$	S	$B(0, 1)$
$T(12, \texttt{3}, \Lambda)$	$R(0, 13, 15)$	S
$T(13, \star, 11 \cdot 12)$	$T(15, \texttt{6}, \Lambda)$	$R(1, 10, 16)$
$T(14, +, 10 \cdot 13)$	E	$T(16, \texttt{0}, \Lambda)$
	$B(0, 1)$	E
	S	S
	$R(1, 10, 16)$	$R(0, 13, 15)$
	$T(16, \texttt{0}, \Lambda)$	$T(15, \texttt{6}, \Lambda)$
	E	E
(a)	(b)	(c)

Fig. 2. Example protocols

Note that the reduction step $R(0, 13, 15)$ in the root segment `0` is performed *after* a reduction step in segment `1` (which evolved from segment `0`). This makes sense for an FLM that supports sharing over non-determinism as discussed earlier: since the value of `(2*3)` is independent on the branch chosen by `coin`, the rewrite step performed for this redex is associated to the root segment.

4 Protocol Correctness

In this section we discuss the correctness of our approach. Intuitively, we want to ensure that a computation space can be encoded in a protocol and a protocol encodes a computation space. Clearly, this informal equivalence must be refined. A computation space can be infinite, while a protocol is always finite. Furthermore, while it is true that any finite computation space can be encoded in a protocol, the converse is false. We defined only the syntax of a protocol. Given a protocol that encodes a computation space, it is possible to add, remove and/or modify elements so that the result is a syntactically compliant, but meaningless protocol. One could define a class of meaningful protocols by stating a number of conditions on such protocols, e.g., for each element $R(b, n_1, n_2)$, there are elements $T(n_1, \ldots)$ and $T(n_2, \ldots)$, etc. Instead of enumerating all these conditions, we implicitly define them by a protocol decoding that fails on protocols that are not meaningful.

The terms of a computation are trivially encoded by the protocol's T elements via the enc mapping defined below. The enc mapping is overloaded since we will use the same name and notation to encode computation spaces as well.

Definition 2 (Term encoding). *The function* enc *maps a term into a sequence of protocol elements. Let* t *be a term where* $\{n_1, \ldots, n_k\}$ *is the set of the nodes of* t *and* label *and* succ *are the labeling and successor functions of* t*. Then we define* $enc(t) = T(n_1, label(n_1), succ(n_1)) \cdots T(n_k, label(n_k), succ(n_k))$.

The T elements produced by $enc(t)$, for some term t, do not encode the root of t. The root of a term is captured by the I and R elements, as shown below.

The protocol is computed and transmitted during the evaluation of a term. As reductions are executed on the term, a topmost, finite, typically partial, portion of the computation space is constructed or discovered. This portion is formalized by the following definition.

Definition 3 (State of traversal). *Let t be a term, $\mathcal{S}$ a strategy and $\textsc{Space}(t)$ the computation space of t w.r.t. $\mathcal{S}$. A* state of traversal *of (the computation space of) t is a finite tree defined by $\overline{\textsc{Space}}(t) = (t, \{\overline{\textsc{Space}}(u_1), \ldots, \overline{\textsc{Space}}(u_k)\})$, where $\{u_1, \ldots, u_k\} \subseteq \mathcal{S}(t)$.*

We are now ready to define the protocol of a computation. The definition of the space encoding is laborious, but conceptually simple. The initialization element is produced only for the top-level term. Each term of the computation space is encoded using the term encoding function. A possibly parallel step is abstracted by an $S \cdots E$ group containing a sequence of R elements defining the root of each redex and corresponding replacement, and T elements defining the nodes of the replacements. For non-deterministic steps, also segment information is encoded using B elements.

Definition 4 (Space encoding). *The function enc maps a state of traversal into a sequence of protocol elements. Let t be a term, $\overline{\textsc{Space}}(t) = (t, \{\overline{\textsc{Space}}(u_1), \ldots, \overline{\textsc{Space}}(u_k)\})$ a state of traversal of t, and s the segment identifier of t in $\textsc{Space}(t)$, where $t \to_{P_i} u_i$, $1 \leqslant i \leqslant k$, is a step according to $\mathcal{S}$. We define enc by structural induction on $\overline{\textsc{Space}}(t)$ as follows. If t is the root of the computation space, let $j = I(s, n_t)$, where n_t is the root of t, otherwise let $j = \Lambda$. If $k = 1$, then $b = \Lambda$, otherwise, let $b_i = B(b_t, b_{u_i})$ and $b = b_1 \cdots b_k$, where b_t and b_{u_i} are the segment identifiers of t and u_i. For $1 \leqslant i \leqslant k$, P_i is the set of positions of a parallel step. For each position p_{ih} in P_i, let $r_{ih} = R(b_i, n_{ih}, n_{u_{ih}})$, where n_{ih} and $n_{u_{ih}}$ are the roots of the redex and the replacement. Let r_i be the concatenation of all the r_{ih} with $p_{ih} \in P_i$, and $r = S \cdot r_1 \cdot enc(u_1) \cdot E \cdots S \cdot r_k \cdot enc(u_k) \cdot E$. By the induction assumption, for $1 \leqslant i \leqslant k$, $enc(\overline{\textsc{Space}}(u_i))$ is defined and we denote it with e_i. Finally, we define $enc(\overline{\textsc{Space}}(t)) = j \cdot enc(t) \cdot b \cdot r \cdot e_1 \cdots e_k$.*

In the above definition, the order of the elements inside an $S \cdots E$ group and between different segments is conceptually irrelevant. This is intended to allow the protocol generation by FLMs that use different search strategies (e.g., depth-first vs. breadth-first) or evaluation orders (e.g., sharing over non-determinism).

In practice, each reduction step $t \to u$ provides a set of T elements for the replacements, an obvious set of R elements, and an optional set of B elements, if the redex is non-deterministic. The term encoding of a replacement very frequently contains T elements that are repeated in the protocol. These are the terms matched to the rule's variables occurring in the right-hand side of rules. Filtering out repeated elements is an obvious optimization that reduces the size of the protocol.

Now, we have the foundations to discuss the correctness of the protocol. Informally, we consider a protocol correct when it encodes the same information as a computation space. More formally, given a state of traversal $\mathcal{T}$, the encoding of $\mathcal{T}$ can be decoded

to obtain $\mathcal{T}$ back. This is a notion of partial correctness because it disregards information about segments which is not present in the computation space. A treatment of the decoding function with the same degree of formality that we used for defining the encoding function would take us well beyond the limits of this paper. Thus, we only sketch the qualifying issues.

First, we focus on terms only. Let e be a protocol. Each T element of e is associated to a node of a term of a state of traversal. An element $T(n, l, s)$ is associated with a node n with label l and successors s. Some conditions of "meaningfulness" must be met. E.g., each node in s must be defined by some other T element of e. Furthermore, if two T elements in e have the same first argument, then they have the same second argument and the same third argument. Thus, the decoding function either produces a set of terms from a protocol or it fails, if the protocol is not meaningful.

Claim 1 *If $\mathcal{T}$ is a state of traversal, decoding $enc(\mathcal{T})$ produces all and only the set of terms of $\mathcal{T}$.*

Proof (Sketch). By definition of the function enc, each node of each term of $\mathcal{T}$ produces a T element in $enc(\mathcal{T})$ and, conversely, a T element in $enc(\mathcal{T})$ is produced only by a node of a term of $\mathcal{T}$. Thus, the proof reduces to verify that the T elements are meaningful.

Second, we focus on the other elements of the protocol. We have argued that all the terms of the state of traversal are decoded from the protocol, thus, we only have to decode the structure of a state of traversal. Again, some conditions of "meaningfulness" must be met. E.g., the redex and replacement of an R element must be defined by T elements. The structure of a computation space, hence a state of traversal $\mathcal{T}$, is a binary relation on the terms of $\mathcal{T}$ that for obvious reasons we call "is a reduct of." For each term t in $\mathcal{T}$, a term u *is a reduct* of t if and only if $t \to u$ according to the strategy. Any reduction of t is defined by an R element in which the second argument is the root of t.

The *decoding* function, dec maps a protocol into a state of traversal. This function is partial, i.e., it is only defined on meaningful protocols. First, we consider the unique initialization element by defining

$$dec(p_1 \cdot I(b, n) \cdot p_2) = dec(b, n, p_1 \cdot p_2)$$

where the ternary function $dec(b, n, p)$ is defined as follows. Let t be the term with root n according to the protocol decoding as above. We distinguish the following cases.

- If there is an R element with b as the first argument in protocol p, take the first $S \cdots E$ group of elements containing this R element. All the R elements in this group define a possibly parallel step which is defined by examining their second and third argument. Let $n' = n$ if there is no replacement for n in this group, otherwise let n' be the root of the new replacement. If $U = dec(b, n', p')$ where p' is the remaining protocol without this $S \cdots E$ group, we define $dec(b, n, p) = (t, \{U\})$.
- Otherwise, there is no R element with b as the first argument. Consider the set $\{B(b, b_1), \ldots, B(b, b_n)\}$ of all B elements in protocol p with b as the first argument, where $n = 0$ is possible if there is no such element. Let p' be the remaining protocol without these elements and $U_i = dec(b_i, n, p')$, $i = 1, \ldots, n$. Then we define $dec(b, n, p) = (t, \{U_1, \ldots, U_n\})$.

Claim 2 *Let $\mathcal{T}$ be a state of traversal. If $\mathcal{T}'$ is obtained by decoding $enc(\mathcal{T})$, then $\mathcal{T}' = \mathcal{T}$.*

Proof (Sketch). The terms of $\mathcal{T}'$ are all and only those of $\mathcal{T}$ by claim 1. Hence, we only have to show that the roots of $\mathcal{T}'$ and $\mathcal{T}$ are the same and the reducts of every term t of $\mathcal{T}'$ and $\mathcal{T}$ are the same. The proof is by induction on the structure of $\mathcal{T}$. Base case: $\mathcal{T}$ is of the form $(t, \{\})$, for some term t, i.e., there are no steps. The protocol contains only an I element beside the T elements defining t. Decoding the T elements of the protocol produces t except for the definition of its root. Decoding the I element establishes the root of t. Since t is not produced by any reduction, t is the root of $\mathcal{T}'$. Ind. case: $\mathcal{T}$ is of the form $(t, \{\overline{\text{SPACE}}(u_1), \ldots, \overline{\text{SPACE}}(u_n)\})$, for some terms $t, u_1, \ldots, u_n$, where $t \to u_i$, for i in $\{1, \ldots, n\}$. According to Definition 4, using the same notation, the protocol is of the form $j \cdot enc(t) \cdot b \cdot r \cdot e_1 \cdots e_k$. By the induction hypothesis, decoding e_i produces $\overline{\text{SPACE}}(u_i)$ except for the definition of u_i. r stands for n groups of the form $S \cdot r_i \cdot enc(u_i) \cdot E$. Each group defines u_i and its root node. Finally, $enc(t)$ defines the root of the state of traversal and j its root node.

The procedure for decoding a protocol and re-constructing a state of traversal is not intended to be ever executed. It is a definition provided only to address the correctness of the protocol in the sense discussed at the beginning of this section.

5 Additional Information

It is expected that the compiler, virtual machine and/or interpreter of an implementation upon request will produce the protocol of an execution. Since these components of an implementation are complicated and strive to execute efficiently, the protocol is designed to contain only essential information. Some information used by advanced tools is intentionally left out of our protocol. However, we expect that useful information can be constructed from the protocol generally with a modest effort, since we have shown, in the previous section, that the protocol is equivalent to the computation space.

For example, declarative debuggers [12, 24] are more effective when only the steps that contributed to the result of a computation are presented to the user. For the time being, we are intentionally vague on the meaning of "contributed," but will become more precise shortly. Consider the following contrived program:

```
f 2 0 = 0
f _ 1 = 1
```

and the evaluation of $t =$ `f (1+1) (0?1)`, where "`?`" is a standard operation that non-deterministically returns one of its arguments [16].

The computation space of t contains two computations ending in a normal form due to the non-determinism of `0?1`. If an implementation shares the evaluation of the subterm `1+1` between the two computations, which in principle is very desirable, both computations will contain the step `1+1` $\to$ `2` similar to what is shown in space (S2). The replacement of `1+1` contributes to one computation, but not to the other. High-performance interpreters, such as KiCS and the FLVM, will execute this step only once and regard it as belonging to both computations. It would be unnecessarily burdensome

and inefficient for them to determine which computations need this step. Instead, an off-line analysis of the computation may determine this fact more easily.

The notion that a step "contributes" to the result of a computation, is quite technical. It is formalized in [19] for the *sequential* TRSs through the notion of *needed redex*. The situation for functional logic programming is way more complicated. First, functional logic programs are not necessarily sequential. Second, when logic variables are present, a redex may or may not be needed depending on the instantiation of some variables [2, Example 6]. These issues, together with various degrees of need for various classes of functional logic programs, are discussed in [2]. Programs for functional logic languages such as Curry and Toy are modeled by the whole class of the constructor based, conditional graph rewriting systems. For these programs, there are terms that have no needed redex [2, Example 10]. However, *source* programs in this large class are often compiled into *target* programs belonging to more restrictive classes, often either the inductively sequential or the overlapping inductively sequential programs, for which a notion of need is meaningful. Therefore, it may be possible to determine whether a step is needed for a target program into which a source program has been compiled. This ultimately explains why a step is executed. Below, we sketch how to accomplish this.

We discuss the problem for the overlapping inductively sequential programs. This class is large enough that any constructor based, conditional program can be compiled into a program belonging to this class. The source program and the target program produce the same results in the sense of [1, Theorem 2] and there exists a strategy for the target program that performs only steps needed modulo a non-deterministic choice. We regard these steps as those that *contribute* to a computation.

The following definition formalizes the intuition that a term t might be reduced by some rules, but not others. This definition is based on an approximation, referred to as "arbitrary reductions" [23], that assumes that a term t *might* be reducible to a term u if t is a redex, regardless of u and the rewrite rules applicable to t. This assumptions is commonly made to prove interesting properties of most practical strategies for functional logic computations. Another way of putting it, is that properties so proved hold for strategies that "do not look at the right-hand sides of the rules" the reason being that reasoning about the right-hand sides is very difficult and often leads to undecidable conditions.

Definition 5 (Pre-redex w.r.t. to a rule). *Let P be a (possibly overlapping) inductively sequential program, t a term, n the root of t, f the label of n, where f is a defined operation, and $l \to r$ a rewrite rule of f. We call* constructor pattern *of t the term t' obtained from t by replacing every operation-rooted subterm of t, except t itself, with a fresh variable. We say that t is a* pre-redex w.r.t. to $l \to r$ *iff l and t' unify, i.e., there exists a substitution σ such that $\sigma(t')$ and $\sigma(l)$ are isomorphic (equal modulo a renaming of nodes).*

Definition 6 (Directly needed). *Let P be a (possibly overlapping) inductively sequential program, t a term, n the root of t, f the label of n, where f is a defined operation, and S the set of rules of P such that t is a pre-redex w.r.t. every rule in S. Let t' be a subterm of t, n' the root of t', f' the label of n', where f' is a defined operation, and $n = n_0 n_1 \dots n_k = n'$ a path from n to n' such that the label of n_i, $0 < i < k$, is a*

constructor symbol. Observe that in the constructor pattern of t, t' has been replaced by a fresh variable. We say that t' is directly needed *by t iff for every rule $l \to r$ in S, the isomorphism unifying l to the constructor pattern of t maps a node m_l of l labeled by a constructor symbol to n'.*

We justify very informally the previous definition. A term t can be reduced at the root only by the rules of P w.r.t. which t is a pre-redex. Every left-hand side l of one such rule has a constructor symbol at a position that corresponds to the root of t', whereas t' is operation rooted. Hence, unless t' is reduced (to a constructor-rooted term), t cannot be reduced (to a value). Obviously, reducing t' is a necessary condition to compute the value of t, but it is not sufficient (it becomes sufficient under the assumption of arbitrary reductions).

Definition 7 (Contributing step). *Let P be a (possibly overlapping) inductively sequential program, t a term and t' a subterm of t. We say that t' is* needed *by t iff either of the following conditions hold:*

1. *t' is a maximal operation-rooted subterm of t, i.e., there exists a path from the root of t to the root of t' in which the only node labeled by an operation symbol is the root of t', or*
2. *there exists a needed subterm t'' of t such that t' is directly needed by t''.*

Let $A : t = t_0 \to t_1 \to \cdots$ be a computation of t. We say that a step $t_i \to t_{i+1}$ at some position p contributes to A iff the subterm of t_i at p is needed by t_i.

Example 2. Referring to the code of the operation `f` presented earlier in the section, two computations of $t =$ `f (1+1) (0?1)` are:

$t \to$ `f (1+1) 0` $\to$ `f 2 0` $\to 0$
$t \to$ `f (1+1) 1` $\to$ `f 2 1` $\to 1$

It is simple to verify that, according to the previous definitions, the step $1 + 1 \to 2$ contributes to the first computation, but not to the second.

6 Reducing Protocol Information

We have shown that a meaningful protocol contains enough information to reconstruct the corresponding computation space. If a user wants to browse through a computation, e.g., in order to find some bugs using a tracer or declarative debugger, the steps of a computation space should be shown in an order more closely related to the program text. In particular, the lazy evaluation order is not adequate and should be replaced by a view corresponding to an eager evaluation. Due to the fact that eager evaluation is incomplete in general (see Example 1), an eager evaluation view is often implemented by executing the program with the standard lazy strategy and storing information about the performed reduction steps during this execution. Then, separate visualization or debugging tools can use this information to reconstruct an appropriate eager view of the program execution [10, 13].

Although a similar approach could be based on our protocol, too, it is known that the size of the stored information can be huge for real-life programs. Therefore, Brassel

et al. [9] proposed to reduce the size of the information to be stored by allowing the visualization or debugging tool to recompute some of this information. This framework, called "lazy call-by-value evaluation," is based on two phases. First, the program is executed with the standard (lazy) evaluation strategy. During the execution, the number of reduction steps *in eager evaluation order* is computed and stored. Since a lazy strategy may not perform all reduction steps compared to an eager strategy, the number of reduction steps is represented as a *step list* of the form `[`s_1`,`s_2`,...,`s_k`]` which has the following interpretation: perform (in leftmost innermost order!) s_1 reduction steps, then skip the next function call (i.e., replace its result by the "undefined" element _), then perform s_2 steps, skip the next function call, etc. For instance, consider the rewrite rules

```
length [] = 0
length (x:xs) = length xs

main = length [fib 0]
```

(where the definition of the function `fib` is omitted). The step list to execute `main` is `[1,2]`. This specifies an evaluation in leftmost innermost order where one step is performed (to reduce `main`), then the next function call `fib 0` is skipped (i.e., replaced by _ since its value is not needed), and, finally, two steps are performed to evaluate the result `0`.

The computed step list can be used by tracers or debuggers that execute the program with an eager strategy respecting the step list. For instance, the declarative debugger described in [9] first asks whether

```
main --> 0
```

is intended (where the result `0` is computed by executing `main` with the lazy call-by-value strategy described above). If we answer "no", it asks a question about the innermost call of the right-hand side of the `main` rule, i.e., whether

```
fib 0 --> _
```

is intended (where the undefined result indicates that the value of this redex is not needed). If we stay undecided, the debugger proceeds with the next call in innermost order and ask whether

```
length [_] --> 0
```

is correct. If we answer "no", it asks the final question

```
length [] --> 0
```

After answering with "yes", the debugger indicates that the second rule of `length` must contain an error.

Hence, these two phases with the intermediate construction of the step list ensures a comprehensible execution order during debugging time. The compactness of this representation is indicated by the fact that a computation that demands the evaluation of all its redexes is represented by a step list containing a single number (the total number of executed steps). Thus, this approach trades time (to recompute results during the interactive debugging phase) for memory space.

In [9] it has been shown how to construct step lists by transforming a functional program into an instrumented program that behaves as the original program, but stores

the step list as a side effect during its execution. In order to demonstrate the generality of our protocol, we show how to reduce a meaningful protocol to a step list. In practice, this reduction is done on the fly, i.e., during the computation of the protocol by the FLM.

The following information is relevant to generate a step list:

1. The application of a rewrite rule to some node of a term. For instance, if no rewrite rule has been applied to some node, the evaluation of this node was not needed, i.e., its evaluation can be skipped in an eager order.
2. If a rewrite rule has been applied, the operation-rooted nodes of the replacement in innermost order are relevant. The constructor nodes are irrelevant since they are not counted in the step list.
3. The segment identifier of each term replacement is relevant in order to associate reductions to the corresponding results.

Note that the possible parallelism in computation steps is not relevant here so that we can ignore this information. Altogether, the information about replacements contained in the protocol can be reduced to *events* of the form $n \xrightarrow{b} n_1 \dots n_k$ specifying that node n is replaced in segment b by a replacement with $n_1 \dots n_k$ as operations in leftmost innermost order. For instance, the three replacements shown in Fig. 2(b) are reduced to the events

$$9 \xrightarrow{0} 10 \cdot 13 \cdot 14$$
$$13 \xrightarrow{0} \Lambda$$
$$10 \xrightarrow{1} \Lambda$$

These events can be incrementally transformed into a step list by associating step counters to nodes and incrementing them when such a node is replaced, i.e., when an event with this node as the left-hand side occurs. The details of this transformation for purely deterministic computations (i.e., where the segment identifier is identical in all events) are described in [9].

The extension to the non-deterministic case is not straightforward (and, actually, not discussed in [9]). For this purpose, the information about segments contained in our protocol structure is quite useful since we can compute a step list for each individual answer. For instance, the element $B(0,1)$ in the protocol of Fig. 2(b) indicates that the events $9 \xrightarrow{0} 10 \cdot 13 \cdot 14$ and $13 \xrightarrow{0} \Lambda$ do also count for the segment identified by 1 (with the result 6). Thus, by considering the tree structure given by all B elements of the protocol, we can construct for each final result a separate step list. In the case of the protocol corresponding to computation space (S1), this would be the list of step lists

```
[[4],[4]]
```

(since in both cases all redexes are evaluated). Obviously, these step lists do not contain enough information, since both step lists are identical, although one should specify the evaluation of `main` to `6` and the other the evaluation to `7`. The problem stems from the definition of `coin` by two overlapping rules (overlapping rules are not considered in [9]). The lazy call-by-value interpreter cannot deduce from this step list which of the two alternative rules should be applied. This problem can be fixed by extending step

lists with *choice steps*: a choice step of the form "`?`n" indicates that the n-th rewrite rule[3] should be applied. In this case, the protocol of computation space (S1) would be reduced to the list of step lists

```
[[1,?1,2],[1,?2,2]]
```

For instance, the step list `[1,?1,2]` specifies the eager evaluation of `main` by one reduction step (resulting in the term `coin+(2*3)`), applying the first `coin` rule to the leftmost innermost subterm `coin` (giving `0+(2*3)`), followed by two innermost steps yielding `6`. Note that, in contrast to [9], skip steps are not performed before and after the specific choice elements in the step list.

A debugging tool could use the list of step lists `[[1,?1,2],[1,?2,2]]` to compute the different results (`6` and `7`) corresponding to each of these lists and asks the user which of these answers is wrong. Then, it would proceed with the selected step list as described above.

It is interesting to note that the protocol of computation space (S2) would be reduced to the identical lists of step lists, although the underlying computation space is different. This is due to the fact that the leftmost innermost evaluation order is identical in both cases.

We have prototypically implemented this reduction schema and a declarative debugger for the specific case that all non-determinism is expressed by the binary choice operation "`?`", e.g., `coin` could be defined by "`coin = 0 ? 1`". In this case, the choice steps can be easily detected in a protocol by checking whether the name of the replaced operation is "`?`".

Of course, there are also other applications where the protocol information can be reduced in other ways. For instance, one could try to visualize the structure of the non-determinism of a computation (e.g., the search tree explored during a computation), or to observe the evaluation of distinct expressions (as in observation debuggers [8]). The development of such reductions and corresponding tools is left for future work.

7 Conclusions

We proposed a protocol for the transmission of a functional logic computation from an interpreter or virtual machine to tools intended to visualize and/or analyze the computation. The protocol serializes the computation space in a text stream that can be stored or processed for specific purposes. The proposed protocol is fairly general in that it supports non-deterministic and parallel computations in a form independent of any concrete strategy and control. We offered a proof sketch of the protocol correctness. The protocol encodes the same information of a finite portion of the computation space of a term.

The protocol is versatile and we intend to use it for a variety of purposes. To this aim we showed that, e.g., information about needed steps can be extracted from the protocol. This information is used, among others, by declarative debuggers. We also showed

[3] In practice, arbitrary choices are often compiled into intermediate languages with binary choices only. In this case, two elements like "`?l`" (left choice) and "`?r`" (right choice) are sufficient.

how to process the protocol information. E.g., the steps executed during a computation with the usual lazy evaluation strategy can be re-ordered for a presentation known as lazy call-by-value view. We believe that the protocol is adequate for the purposes of common debuggers and tracers, and we are open to suggestions from the functional logic research community about extensions for increasing the generality and usefulness of our proposal.

We have started to extend some implementations of functional logic languages with capabilities to produce the protocol to support the development of protocol-aware tools.

References

1. S. Antoy. Constructor-based conditional narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pages 199–206, 2001.
2. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.
3. S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 138–152. MIT Press, 1997.
4. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
5. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.
6. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.
7. S. Antoy, M. Hanus, J. Liu, and A. Tolmach. A virtual machine for functional logic computations. In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pages 108–125. Springer LNCS 3474, 2005.
8. B. Braßel, O. Chitil, M. Hanus, and F. Huch. Observing functional logic computations. In *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pages 193–208. Springer LNCS 3057, 2004.
9. B. Brassel, S. Fischer, M. Hanus, F. Huch, and G. Vidal. Lazy call-by-value evaluation. In *Proc. of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 265–276, 2007.
10. B. Braßel, M. Hanus, F. Huch, and G. Vidal. A semantics for tracing declarative multi-paradigm programs. In *Proc. of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'04)*, pages 179–190, 2004.
11. B. Braßel and F. Huch. The Kiel Curry system KiCS. In *Proc. 17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007) and 21st Workshop on (Constraint) Logic Programming (WLP 2007)*, pages 215–223. Technical Report 434, University of Würzburg, 2007. To appear in Springer LNAI.
12. R. Caballero and M. Rodríguez-Artalejo. A declarative debugging system for lazy functional logic programs. *Electronic Notes in Theoretical Computer Science*, 64, 2002.
13. O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood – a comparative evaluation of three systems for tracing and debugging lazy functional programs. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, pages 176–193. Springer LNCS 2011, 2001.

14. R. Echahed and J.-C. Janodet. Admissible graph rewriting and narrowing. In *Proc. Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 325–340, 1998.
15. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.
16. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
17. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2007.
18. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, 2006.
19. G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 395–443. MIT Press, 1991.
20. F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
21. W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 100–113. Springer LNCS 1722, 1999.
22. D. Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*, pages 3–61. World Scientific, 1999.
23. R.C. Sekar and I.V. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. *Information and Computation*, 104(1):78–109, 1993.
24. E. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts, 1983.
25. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.
26. P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.