Exploiting Non-Determinism through Laziness in Guarded Functional Languages

Roland Dietrich, Hendrik C.R. Lock*
GMD Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1, D 7500 Karlsruhe, FRG
e-mail:{*dietrich,lock*}*@karlsruhe.gmd.dbp.de*

Abstract

Guarded Functional Programming is an approach to integrate functional programming, represented by equations and rewriting, and logic programming, represented by Horn clauses and SLD-resolution: the selection of a guarded equation for rewriting an expression is determined by (1) pattern matching with the left hand side and (2) solving the guard which is a Horn logic goal. Besides an introduction to this style of programming, we present an extension of guarded functional programming by the concept of stream comprehensions. The main principle is that, instead of committed choice of one solution, a stream of multiple solutions of a guard is produced lazily. These streams can be consumed in functional expressions. This approach makes it possible to exploit in the functional programming world the non-determinism which is inherent in the logic programming world. Thereby, the deterministic nature of functional programming is maintained, and an efficient implementation is possible through the interleaving of functional and logic computations.

1 Introduction

Guards have been introduced in logic programming languages as a means for expressing non-determinism and parallelism. In Parlog [CG86] and Guarded Horn Clauses (GHC) [Ued85], the clause to be selected for resolving a goal is determined not only

*This work has been funded by the CEC under ESPRIT Basic Research Action No. 3147 (PHOENIX).

by unification with clause heads but also by parallel processing of guards of clauses. The guards themselves are goals, too. The first clause whose guard succeeds is (non-deterministically) selected; all other clauses are discarded *(committed choice)*. The general idea of guards as a means to express non-deterministic control was first shown through Dijkstras *guarded commands* [Dij75].

If we introduce this concept to equation-based functional programming languages, we obtain the notion of *guarded equations* and *guarded term rewriting*. If guards still remain Horn logic goals which can be solved w.r.t. a Horn logic program, the guards constitute a call interface from the functional programming world to the logic programming world, and one achieves an integration of functional and logic programming (*guarded functional programming, GFP*). A guarded equation can be applied to a ground expression if (1) pattern matching with the left hand side succeeds and (2) the guard is satisfiable. Communication between the functional and logical world is established in a natural manner by means of the term universe which is common to both paradigms, and sharing of logical variables. An example of such a guarded functional language is *Guarded Term ML* (GTML) [Loc88] which, as well as guarded term equations, incorporates λ-expressions and higher order functions.

The most striking advantage of GFP is that the functional and logic worlds are kept strongly separated. This enables the direct use of efficient implementation techniques available for both paradigms (e.g. [War83, Pey87]). One needs not cope with the enormous efficiency problems of more unified approaches based on equational logic and narrowing (see [Höl89, Boc90] for an overview).

On the other hand, this conceptual separation of functional and logic prgramming within one framework gives rise to some compatibility problems concerning data and control flow between the worlds: the evaluation of guards may involve logical variables, backtracking, and multiple solutions (or non-determinism), which are incompatible with functional programming. Furthermore, higher order functional programming is incompatible with the logic world.

In GTML, a "brute force" solution to these problems has been taken: the left hand sides of guarded equations and the solution of guards are tried in textual order, and whenever both succeed, the remaining equations as well as additional solutions of the guard are discarded. Thus, the committed choice principle of Parlog and GHC has been adopted replacing parallelism and non-determinism by textual order. Dataflow of non-ground terms to variables of functional expressions and instantiation of logical variables occuring in guards with higher order expressions are simply "forbidden" (i.e. the resulting expressions are undefined). GTML is therefore purely deterministic: every expression evaluates to a unique value (or it is undefined).

In this paper, we propose a solution to one part of the problems outlined above: for guarded functional programming, we show how non-determinism caused by multiple solutions of guards can be made available in the functional world by means of streams of solutions which are evaluated lazily. We characterize the resulting language as *guarded functional programming with lazy streams* (GFP*).

Guarded functional programming with lazy streams is non-deterministic, not in the sense that an expression can be evaluated to more than one value, but in the sense that it can be evaluated to a stream of values. The non-determinism lies in the unspecified order in which the values occur in this stream. Lazy evaluation ensures that if only one value is needed only one is produced, and that one is able to cope with expressions yielding infinitely many solutions. Moreover, the functional and logic worlds are still strongly separated which is one of the most important benefits of this approach to functional logic programming.

In the next section we introduce the concept of guarded term rewriting as a model for guarded functional programming (GFP). In section 3 we show how GFP can be extended to exploit non-determinism of guards by lazy streams in GFP*. For both languages, GFP and GFP*, we define a reduction semantics and an evaluation function. Finally, references to related work, conclusions, and future work are given in section 4 and section 5, respectively.

2 Guarded Functional Programming (GFP)

Notations. Throughout this paper, Σ denotes a signature of *function symbols* with arity ($f/n \in \Sigma$, $n \geq 0$), Π a signature of *predicate symbols* with arity ($p/n \in \Pi$, $n \geq 0$), and V denotes an (infinite) set of *variables* ($x, z, y \in V$). The set of *ground terms* over Σ and the set of *(free) terms* over Σ and V are denoted by T_Σ and $T_\Sigma(V)$, respectively. For $t \in T_\Sigma(V)$, $var(t)$ denotes the set of variables occuring in t.

The set of *occurences* of a term t is denoted by $Occ(t)$. Occurences of terms are usually denoted by sequences of natural numbers specifying access paths to the respective subterms. For $u \in Occ(t)$, t/u denotes the subterm of t at occurence u and $t[u \leftarrow s]$ denotes t where the subterm of t at occurence u is replaced by the term s. The relation $<$ on occurences denotes the usual prefix order on sequences of natural numbers. ($u < v$ iff u is a prefix of v).

The set of *substitutions* on $T_\Sigma(V)$, i.e. mappings which define the replacement of variables by terms, is denoted by $S_\Sigma(V)$. The set of *ground substitutions* which always replace variables by ground terms is denoted by S_Σ.

An *atom* is of the form $p(t_1, \ldots, t_n)$ where $p/n \in \Pi$ and $t_i \in T_\Sigma(V)$ ($t_i \in T_\Sigma$ for *ground atoms*). We denote the set of all atoms or ground atoms over Π and Σ by $A_{\Pi,\Sigma}(V)$ and $A_{\Pi,\Sigma}$, respectively. A *goal* is a conjunction of atoms. The set of all goals is denoted by $G_{\Pi,\Sigma}(V)$ ($G_{\Pi,\Sigma}$ for *ground goals*). We denote goals by $\Leftarrow A_1, \ldots, A_n$ ($A_i \in A_{\Pi,\Sigma}$, $n \geq 0$).

Definition 1 (Horn logic programs.) A *(Horn) logic program* $\mathcal{L}$ is a set of *definite clauses*

$$A_0 \Leftarrow A_1, \ldots, A_n$$

where $A_i \in A_{\Pi,\Sigma}(V)$, $n \geq 0$.

We say that a goal $G \in G_{\Pi,\Sigma}(V)$ *succeeds with* $\tau \in S_\Sigma(V)$ ($G \vdash_{\mathcal{L}} \tau$), iff G can be derived to the empty goal by SLD-resolution with the answer substitution τ.
$\square$

Throughout the paper, we abstract from the execution of Horn logic programs by SLD-resolution, and we assume the relation $\vdash_{\mathcal{L}}$ to be given. See [Llo84] for details.

Definition 2 (Guarded Functional Programs) A *guarded functional program* (GFP-program for short) is a pair $\mathcal{P} = \langle \mathcal{L}, \mathcal{E} \rangle$ such that

- $\mathcal{L}$ is a Horn logic program and
- $\mathcal{E}$ is a set of *guarded equations*

  $$l \,[]\, G \;=\; r$$

  where $l, r \in T_\Sigma(V)$, $G \in G_{\Pi,\Sigma}(V)$, and $var(r) \subseteq var(l) \cup var(G)$. G is called the *guard* of the equation, l the left hand side, and r the right hand side.

$\square$

Informally, the semantics of a guarded equation $l \,[]\, G = r$ is as follows: l and r are equal if the guard G succeeds w.r.t. the related Horn logic program.

Formally, a ground term or expression can be rewritten to another expression w.r.t. a GFP-program, according to the rewrite relation described below. The normalforms w.r.t. this relation can be considered as the value or the *meaning* of an expression w.r.t. a GFP-program. That is, the *guarded term rewriting relation* constitutes a reduction semantics for GFP-programs.

Definition 3 (Guarded Term Rewriting) A GFP-program $\mathcal{P} = \langle \mathcal{L}, \mathcal{E} \rangle$ defines a rewrite relation $\longrightarrow_{\mathcal{P}}$ on T_Σ as follows:

$s \longrightarrow_{\mathcal{P}} t$ iff there are $u \in Occ(s)$, $l \; [] \; G = r \in \mathcal{E}$, and $\sigma \in S_\Sigma$, $\tau \in S_\Sigma(V)$ such that

(1)	$s/u = \sigma l$,	*(pattern match)*
(2)	$\sigma G \vdash_{\mathcal{L}} \tau$ such that $\tau\sigma r \in T_\Sigma$,	*(success of guard)*
(3)	$t = s[u \leftarrow \tau\sigma r]$	*(replacement)*

A term $t \in T_\Sigma$ is called a *$\mathcal{P}$-normalform* of $s \in T_\Sigma$ iff $s \xrightarrow{*}_{\mathcal{P}} t$ and there is no $t' \in T_\Sigma$ such that $t \longrightarrow_{\mathcal{P}} t'$, where $\xrightarrow{*}_{\mathcal{P}}$ is the reflexive and transitive closure of $\longrightarrow_{\mathcal{P}}$.
□

Example 1. The following GFP-program $\mathcal{P}_1 = \langle \mathcal{L}_1, \mathcal{E}_1 \rangle$ consists of a logic program which defines an *append*-predicate on lists and a guarded equation defining a function *split* which splits a list into two parts. *split* is defined using the *append* predicate as a guard.

$$\begin{aligned} \mathcal{L}_1: \; & append([], L, L). \\ & append([X|L1], L2, [X|L3]) \Leftarrow append(L1, L2, L3). \end{aligned}$$

$$\mathcal{E}_1: \; split(L) \; [] \; append(L1, L2, L) \; = \; (L1, L2)$$

□

When applying a guarded equation, data flow can take place from the left hand side to the guard (caused by the pattern match substitution), from the left hand side to the right hand side (also caused by the pattern match substitution), and from the guard to the right hand side (caused by the answer substitution computed for the success of the guard).

Note that we restrict the answer substitutions for the success of guards to those which are ground on the variables of the right hand sides of a guarded equation because we do not want to deal with logical variables in functional expressions. Pattern match substitutions are ground anyway, because the expressions to be reduced are ground.

In general, an expression has more than one normalform w.r.t. a GFP-program, since guards can have more than one answer substitution. For example, the expression $split([1,2])$ has three $\mathcal{P}_1$-normalforms: $([], [1,2])$, $([1], [2])$, and $([1,2], [])$. That is, the function *split* is non-deterministic, in other words, the relation $\longrightarrow_{\mathcal{P}}$ is not confluent.

The non-determinism of expressions causes some problems for the functional part of GFP. Consider the following example:

Example 2. Let $\mathcal{P}_2$ be the GFP-program $\langle \mathcal{L}_1, \mathcal{E}_1 \cup \mathcal{E}_2 \rangle$ where

$$\begin{aligned} \mathcal{E}_2: \quad eq\_split(L) = \; & if\ length(fst(split(L))) = length(snd(split(L))) \\ & then\ split(L) \\ & else\ ([],[]). \end{aligned}$$

$$\begin{aligned} & length([]) = 0. \\ & length([X|L]) = 1 + length(L). \end{aligned}$$

$$\begin{aligned} & fst((X,Y)) = X. \\ & snd((X,Y)) = Y. \end{aligned}$$

The function *eq_split* is intended to split a list into two parts with equal length if possible, otherwise it should return a pair of empty lists.[1]

□

With respect to the reduction semantics, the meaning of the split function covers its intended meaning. That is, whenever a list can be split into two lists of equal length, the corresponding pair of lists is a $\mathcal{P}_2$-normalform of the original list. For example,

$$eq\_split([1,2]) \xrightarrow{*}_{\mathcal{P}_2} ([1],[2])$$

But the formal meaning specifies also the result $([],[])$ and even $([],[1,2])$ or $([1,2],[])$ as a correct output. This is because the three occurences of $split(L)$ in the right hand side of the *eq_split*-equation can be rewritten independently at all occurences (see above). If the language is extended to allow sharing of expressions by means of local definitions, the situation would be slightly improved, but still unsatisfying. For example, replacing the *eq_split*-equation in Example 2 by the following equation

$$\begin{aligned} eq\_split(L) \;=\; & let\ (L1,L2) = split(L)\ in \\ & if\ length(L1) = length(L2) \\ & then\ (L1,L2) \\ & else\ ([],[]). \end{aligned}$$

would still imply two possible solutions for the expression $eq\_split([1,2])$, namely $([1],[2])$ and $([],[])$.

Thus, *computing* the values of an expression needs backtracking even in functional expressions in order to be complete w.r.t. the reduction semantics. A backtracking semantics in the sense of Prolog would provide one solution of a guard after another. In

[1]We consider elementary functions like arithmetic, list constructors and if-then-else to be "built-in" and do not define them equationally here.

order to consider all possible solutions of a functional expression, backtracking must even affect the functional computations.

In summary, the functional nature of the functional sublanguage of GFP is destroyed when we allow unrestricted use of the non-determinism and backtracking provided by the logic sublanguage. The advantages of the approach as stated in the introduction, namely a separation of the functional and logic worlds which enables efficient implementation, is lost.

In the language Guarded Term ML (GTML) [Loc88, Loc89], this problem has been solved pragmatically by imposing a fixed evaluation strategy: equations are tried for pattern matching in textual order. If this is successful, the guard is evaluated w.r.t. the logic part using the usual Prolog derivation strategy, i.e. depth first search resolving leftmost atoms of goals first. Whenever a guard succeeds, all other choices are committed (other solutions of guards as well as other applicable equations). This evaluation strategy is expressed by the evaluation function presentend in Definition 4 below. In this definition, we abstract from the search strategy for the Horn logic part of GFP, we only assume the existence of a function $solve_{\mathcal{L}}$ which outputs for any goal an (arbitrary but deterministically computed) correct answer substitution for the goal w.r.t. $\mathcal{P}$. In GTML, the Horn logic part is identical to Prolog.

Definition 4 (Evaluation Function for GFP) Let $\mathcal{P} = \langle \mathcal{L}, \mathcal{E} \rangle$ be a GFP program such that $\mathcal{E} = \{l^i \; [] \; G^i = r^i \mid 1 \leq i \leq n\}$ is an ordered set of guarded equations. The function $eval_{\mathcal{P}} : T_\Sigma \longrightarrow T_\Sigma$ is defined by

$$\begin{aligned}
eval_{\mathcal{P}}(s) \;&=\; eval_{\mathcal{P}}(t) \text{ iff } \exists l^i \; [] \; G^i = r^i \in \mathcal{E} \, \exists u \in Occ(s) \, \exists \phi \in S_\Sigma(V) : \\
&\quad (1) \;\; apply_{\mathcal{P}}(s, u, l^i \; [] \; G^i = r^i, \phi) \;\wedge\; t = s[u \leftarrow \phi r^i] \;\wedge \\
&\quad (2) \;\; \forall v \in Occ(t) : [v < u \Rightarrow \forall j \in \{1, ..., n\} \forall \rho \in S_\Sigma(V) : \\
&\qquad\qquad \neg apply(s, v, l^j \; [] \; G^j = r^j, \rho)] \;\wedge \\
&\quad (3) \;\; \forall j \in \{1, \ldots, k-1\} \forall \rho \in S_\Sigma(V) : \neg apply_{\mathcal{P}}(s, u, l^j \; [] \; G^j = r^j, \rho) \\
eval_{\mathcal{P}}(s) \;&=\; s \text{ otherwise.}
\end{aligned}$$

where for $s \in T_\Sigma$, $u \in Occ(t)$, $k \in \{1, \ldots, n\}$, and $\tau \in S_\Sigma(V)$

$$\begin{aligned}
&apply_{\mathcal{P}}(s, u, l^k \; [] \; G^k = r^k, \phi) \text{ iff } \exists \tau \in S_\Sigma(V), \sigma \in S_\Sigma : \\
&\qquad s/u = \sigma l^k \;\wedge\; solve_{\mathcal{L}}(\sigma G) = \tau \;\wedge\; \phi = \tau\sigma \;\wedge\; \phi r^k \in T_\Sigma
\end{aligned}$$

and the function $solve_{\mathcal{L}} : G_{\Pi,\Sigma}(V) \longrightarrow S_\Sigma(V)$ satisfies for all $G \in G_{\Pi,\Sigma}(V)$:

$$solve_{\mathcal{L}}(G) = \tau \quad \Rightarrow \quad G \vdash_{\mathcal{L}} \tau \; .$$

□

The conditions (1) - (3) occuring in the first $eval_{\mathcal{P}}$ equation mean that (1) there is an guarded equation which is applicable, (1) there is no larger occurence (i.e. no subexpression) where an equation is applicable *(outermost rewriting)*, and no equation with a smaller index is applicable at the same occurence *(minimality)*. The conditions on the function $solve_{\mathcal{C}}$ applied to a goal G ensure that every substitution computed is a correct answer substitution for G (correctness). It is not required that whenever there is a correct answer substitution for G, one of these answer substitutions is computed. In GTML a standard Prolog interpreter is used to implement such a function: the first solution for G delivered by the interpreter is taken as the result of $solve_{\mathcal{C}}(G)$. In particular, this means that an evaluation is undefined if the first solution of a guard instantiates variables of the right hand side with non-ground terms.

From the definition of the $apply_{\mathcal{P}}$-predicate and the conditions for the $solve_{\mathcal{C}}$-function it is obvious that the $eval_{\mathcal{P}}$-function is correct w.r.t. the reduction semantics:

Theorem 5 *Let $\mathcal{P}$ be a GFP program and $s, t \in T_{\Sigma}$. If $eval_{\mathcal{P}}(s) = t$ then t is a $\mathcal{P}$-normalform of s.*
□

Because of the commited choice principle realized in the $apply_{\mathcal{P}}$ predicate, $eval_{\mathcal{P}}$ is not complete. For example, with the Prolog derivation strategy for Horn logic programs we have

$$eval_{\mathcal{P}_2}(eq\_split([1,2])) = ([],[]).$$

For the reasons discussed above, we do *not* even want to be complete, that is we do not want to introduce backtracking when evaluating functional expressions. On the other hand, it would be an advantage, if we could exploit the power of non-determinism in the functional world instead of just ignoring it. For example, we would like to be able to select the "right" normalforms of a *split*-expression when evaluating an *eq_split*-expression (namely those which split into two lists of equal length).

3 GFP and Stream Producers (GFP*)

In the previous section, we have discussed the problem of how to exploit non-determinism, i.e. multiple solutions of guards, in GFP-programs without introducing non-determinism into functional expressions. This problem can be solved by means of *streams* of values which collect the (multiple) solutions of guards. These streams can be consumed deterministically in the functional world. A lazy evaluation strategy enables dealing with infinite streams, i.e. infinitely many solutions of a goal. Furthermore, lazy evaluation of streams ensures that no unnecessary or unwanted solutions of guards are computed.

Throughout this section, the signature Σ is defined to contain a nullary and a binary function symbol for constructing streams. We streams as:

$<>$	(the empty stream) and
$< a :: s >$	(a stream with first element a followed by a stream s).

In principle, there is no difference between lists and streams. We distinguish them only for pragmatic reasons.

Definition 6 (GFP with Streams) A *GFP-program with streams* (GFP* - program for short) is a pair $\mathcal{P} = \langle \mathcal{L}, \mathcal{E} \rangle$ such that

- $\mathcal{L}$ is a Horn logic program, and
- $\mathcal{E}$ is a set of equations which are either guarded equations (cf. Definition 2) or *stream producers* of the form

  $$l = < r \,\|\, G >$$

  where $l, r \in T_\Sigma(V)$, $G \in G_{\Pi,\Sigma}(V)$ and $var(r) \subseteq var(l) \cup var(G)$. The right hand side of a stream producer is called a *stream comprehension.* We denote the set of all stream comprehensions w.r.t. Π and Σ by $SC_{\Pi,\Sigma}$ (which is isomorphic to the cartesian product $T_\Sigma(V) \times G_{\Pi,\Sigma}(V)$).

□

To define a reduction semantics for GFP*-programs, we must explain how to apply stream producers and how to rewrite stream comprehensions. Informally, a stream producer $l = < r \,\|\, G >$ can be applied to a ground term t if l matches a subterm of t and the subterm is replaced by the stream comprehension, where the matching substitution has been applied to r and G. A stream comprehension $< r \,\|\, G >$ rewrites to a stream of all terms τt such that τ is a correct answer substitution for G which makes t ground. This cannot be expressed by pure term rewriting. Therefore, we introduce a rewriting operator $\Longrightarrow_{\mathcal{L}}$ which reduces stream comprehensions to streams w.r.t. a Horn logic program $\mathcal{L}$.

Definition 7 (Reduction Semantics for GFP*) Let $\mathcal{P} = \langle \mathcal{L}, \mathcal{E} \rangle$ be a GFP*-program such that $\mathcal{E} = \mathcal{E}_g \uplus \mathcal{E}_s$ where $\mathcal{E}_g$ contains only guarded equations and $\mathcal{E}_s$ contains only stream producers. The relation $\Longrightarrow_{\mathcal{P}} \subseteq T_\Sigma \cup SC_{\Pi,\Sigma} \times T_\Sigma$ is defined by

$$\Longrightarrow_{\mathcal{P}} = \longrightarrow_{\mathcal{P}_0} \cup \Longrightarrow_{\mathcal{E}_s} \cup \Longrightarrow_{\mathcal{L}}$$

where for $t, t_i \in T_\Sigma$, $r, l \in T_\Sigma(V)$, and $G \in G_{\Pi,\Sigma}(V)$

- (*guarded term rewriting*)
  $\mathcal{P}_0 = \langle \mathcal{L}, \mathcal{E}_g \rangle$ is a GFP-program, i.e. $\longrightarrow_{\mathcal{P}_0}$ is the guarded term rewriting relation (cf. Definition 3)
- (*application of stream producers*)
  $t \Longrightarrow_{\mathcal{E}_s} t[u \leftarrow < \sigma r \,\|\, \sigma G >]$ iff $\exists u \in Occ(t), \exists l = < r \,\|\, G > \in \mathcal{E}_s$: $t/u = \sigma l$
- (*rewriting of stream comprehensions*)
  $< r \,\|\, G > \Longrightarrow_{\mathcal{L}} < t_1, t_2, \ldots >$ iff

$$\begin{array}{lll} (1) & \text{If } G \vdash_{\mathcal{L}} \tau \text{ then } \exists i : \sigma r = t_i & (\textit{completeness}\,) \\ (2) & \forall i \, \exists \tau : t_i = \sigma r \wedge G \vdash_{\mathcal{L}} \tau & (\textit{soundness}\,). \end{array}$$

A term $t \in T_\Sigma$ is called a *$\mathcal{P}$-normalform* of $s \in T_\Sigma$ iff $s \overset{*}{\Longrightarrow}_{\mathcal{P}} t$ and there is no $t' \in T_\Sigma$ such that $t \Longrightarrow_{\mathcal{P}} t'$ where $\overset{*}{\Longrightarrow}_{\mathcal{P}}$ is the reflexive and transitive closure of $\Longrightarrow_{\mathcal{P}}$.
□

Note that the relation $\Longrightarrow_{\mathcal{L}}$ still models non-deterministic rewriting: the order in which values occur in streams is not specified. The completeness and soundness conditions ensure that every solution of the goal of a stream comprehension is represented in the stream and that every value occuring in the stream corresponds to a solution of the goal. Because the guarded term rewriting relation is only defined on ground terms, rewriting of non-ground stream elements is undefined.

Example 3. Let $\mathcal{P}_3$ be the GFP*-program $\langle \mathcal{L}_1, \mathcal{E}_3 \rangle$ where $\mathcal{L}_1$ is defined as in Example 1 and

$$\mathcal{E}_3 : \; split(L) = < (L1, L2) \,\|\, append(L1, L2, L) > .$$

$$eq\_split(L) = eqs(split(L)).$$

$$\begin{array}{rl} eqs(<>) = & ([], []). \\ eqs(< (L1, L2) :: S >) = & if\ length(L1) = length(L2) \\ & then\ (L1, L2) \\ & else\ eqs(S). \end{array}$$

$$length([]) = 0.$$
$$length([X|L]) = 1 + length(L).$$

In contrast to Example 1, the function *split* as defined here no longer produces just *one* possible splitting of a list (cf. Example 1), but generates all possible splittings as a stream of pairs of lists. The function *eq_split* initiates the production of this stream and directs it to the function *eqs*, which selects the first pair of equal length if there is one. If there is none, it returns a pair of empty lists.

□

The following definition extends the evaluation function for GFP-programs ($eval_{\mathcal{P}}$) presented in Definition 4 to a correct evaluation function $eval^*_{\mathcal{P}}$ for GFP-programs in a straightforward manner. It still realizes outermost rewriting and minimality, but it no longer discards multiple solutions of goals if they occur in stream comprehensions.

Definition 8 (Evaluation Function for GFP*) Let $\mathcal{P} = \langle \mathcal{L}, \mathcal{E} \rangle$ be a GFP*-program such that $\mathcal{E} = \{e^i \mid 1 \leq i \leq n\}$ is an ordered set of equations where e^i is either a guarded equation or a stream producer. The function $eval^*_{\mathcal{P}} : T_\Sigma \cup SC_{\Pi,\Sigma} \longrightarrow T_\Sigma$ is defined as follows:

$eval^*_{\mathcal{P}}(s) = eval^*_{\mathcal{P}}(t)$ iff $\exists k \in \{0, \dots n\}\, \exists u \in Occ(s)\, \exists \phi \in S_\Sigma(V)$:

(1) $apply^*_{\mathcal{P}}(s, u, e^k, \phi) \wedge t = replace^*_{\mathcal{P}}(s, u, e^k, \phi) \wedge$

(2) $\forall v \in Occ(t) : [v < u \Rightarrow \forall j \in \{1, ..., n\}\, \forall \tau \in S_\Sigma(V) : \neg apply^*_{\mathcal{P}}(s, v, e^j, \tau)] \wedge$

(3) $\forall j \in \{1, \dots, k-1\} \forall \tau \in S_\Sigma(V) : \neg apply^*_{\mathcal{P}}(s, u, e^j, \tau)$

$eval^*_{\mathcal{P}}(s) = s$ otherwise,

where

$apply^*_{\mathcal{P}}(s, u, l \,[]\, G = r, \phi)$ iff

$\exists \sigma \in S_\Sigma\, \exists \tau \in S_\Sigma(V) : s/u = \sigma l \wedge solve_{\mathcal{L}}(\sigma G) = \tau \wedge \phi = \tau\sigma \wedge \phi r \in T_\Sigma$

$apply^*_{\mathcal{P}}(s, u, l =< r \,\|\, G >, \sigma)$ iff

$\sigma \in S_\Sigma \wedge s/u = \sigma l$

$apply^*_{\mathcal{P}}(s, u, e^0, id_{S_\Sigma(V)})$ iff

$\exists l \in T_\Sigma(V) \exists G \in G_{\Pi,\Sigma}(V) : s/u =< l \,\|\, G >$

and

$replace^*_{\mathcal{P}}(s, u, l \,[]\, G = r, \phi) = t$ *iff*

$s/u \in T_\Sigma \wedge t = [u \leftarrow \phi r]$

$replace^*_{\mathcal{P}}(s, u, l =< r \,\|\, G >, \sigma) = t$ *iff*

$s/u \in T_\Sigma \wedge t = s[u \leftarrow< \sigma r \,\|\, \sigma G >]$

$replace^*_{\mathcal{P}}(s, u, e^0, id_{S_\Sigma(V)}) = t$ *iff*

$\exists l \in T_\Sigma(V) \exists G \in G_{\Pi,\Sigma}(V) : s/u =< l \,\|\, G > \wedge t = s[u \leftarrow solve^*_{\mathcal{L}}(< r, G >)]$.

and the functions $solve_{\mathcal{L}} : G_{\Pi,\Sigma} \longrightarrow S_{\Sigma}(V)$ and $solve^*_{\mathcal{L}} : T_{\Sigma}(V) \times G_{\Pi,\Sigma}(V) \longrightarrow T_{\Sigma}$ must satisfy the following conditions:

(1) If $solve_{\mathcal{L}}(G) = \tau$ then $G \vdash_{\mathcal{L}} \tau$, and

(2) $solve^*_{\mathcal{L}}(r, G) = s$ iff $< r \,\|\, G > \Longrightarrow_{\mathcal{L}} s$

□

Theorem 9 *Let $\mathcal{P}$ be a GFP*-program and $s, t \in T_{\Sigma}$. If $eval^*_{\mathcal{P}}(s) = t$ then t is a $\mathcal{P}$-normalform of s.*
□

The evaluation function defined above treats streams (generated by a realization of the function $solve^*_{\mathcal{L}}$) as a whole. This has two drawbacks: when a goal of a stream comprehension has infinite solutions, evaluation will not terminate and eventually not all values represented by a stream comprehension will be needed. For example, the computation

$$\begin{aligned} eq\_split([1,2]) &= eqs(split([1,2])) \\ &= eqs(< (L1, L2) \,\|\, append(L1, L2, [1,2]) >) \\ &= eqs(< ([], [1,2]), ([1],[2]), ([1,2],[]) >) \\ &= \ldots \\ &= ([1],[2]) \end{aligned}$$

w.r.t. $\mathcal{P}_3$ (Example 3), only needs the first two elements of the stream produced by $split([1,2])$.

Therefore, a lazy evaluation strategy for stream comprehensions is required. A corresponding evaluation function can be defined by means of continuations which involves also a continuation semantics for Prolog ([NF89]). For the implementation of this strategy, a coroutine mechanism based on two continuations [Sch88] which represent functional and logic computations can be adopted as follows:

(1) A functional expression is evaluated until it suspends. This evaluation can be eager or lazy.

(2) The evaluation suspends when it demands an element from a stream which is not yet provided. By means of the stream description a logical computation is started.

(3) A failure returns the empty stream.

(4) A success of the logical goal causes the suspension of the logical evaluation. A stream constructed of the solution element and a tail element is returned. The tail element describes the computation of the rest stream in terms of the suspended goal.

(5) The functional computation which consumes the stream is resumed.

Stream producers and guarded equations can co-exist in GFP*-programs and can be used in different ways. If one does not care about non-determinism of goals and if one solution is sufficient, guarded equations are the appropriate way to program. If one needs more than one solution, or if one cares *which* solution a goal can have, stream producers should be used.

In particular, there is the following difference between stream comprehensions and guards: if a guard does not succeed, the corresponding equation is not applicable. If the goal of a stream comprehension does not succeed, the result is the empty stream.

4 Related Work

In the recent years, a great number of proposals for the integration of the logic and the functional programming paradigms has been developed. It is beyond the scope of this paper to summarize them all. For a systematic approach to classify them, see [GL90].

Guarded Functional Programming has essential analogies with conditional term rewriting systems ([Kap84, Boc86]), where the conditions correspond to guards. But in the latter approach, the conditions themselves are equational goals, too, and they have to be solved w.r.t. the whole equational program, and not w.r.t. a separated logic program as in GFP. As a consequence, approaches based on conditional term rewriting cannot be implemented by the coroutining mechanism outlined in the last section. They have to deal with general equation solving which is usually realized by narrowing [Höl89, Boc90] and causes serious computational and efficiency problems [Boc87]. Less general, that is less powerful but more efficient approaches restrict the use of eqational theories within logic programs to reduction of ground terms [SY86, Gol88].

Stream comprehensions in GFP* are comparable to list comprehensions in Miranda [Tur87]. The "qualifiers" of a list comprehension in Miranda are list generators, i.e. functional expressions evaluating to a list. In GFP* the qualifiers are Horn logic goals which evaluate to a list of substitutions. Another approach is followed in Hope with absolute set abstraction [DFP86]: equational goals are used to specify set comprehensions, The goals refer to an (unconditional) equational program and they are solved by means of narrowing.

A extension to functional programming which is not based on set or list comprehensions and compares to GFP is *Constraint Functional Programming* (CFP, [DG89]): general constraints over a constraint system correspond to guards in GFP, and constraint solving in the constraint system corresponds to SLD-resolution. If one considers Horn logic with SLD-resolution as a special constraint system, GFP is a special case of constraint functional programming. On the other hand, whereas constraint systems are defined over special domains, the domain of computation used in GFP is a most general one, namely the initial term algebra. The approach to exploit non-determinism presented here can be adapted to CFP, too.

A more powerful extension of CFP is *Constraint Equational Deduction* (CED, [DG90]), where functional term rewriting is replaced by equational deduction. As for all other approaches based on equational logic, efficient implementation of CED is a crucial and as yet unsolved problem, because one has no conceptual separation of functional and logic computations.

5 Conclusions

Guarded functional programming (GFP) is an approach to the integration of functional and logic programming where both paradigms are still conceptually separated and separately implementable. Its drawback is that the non-determinism is always treated by committed choice of solutions of Horn logic goals and multiple solutions cannot be dealt with.

In this paper, we have introduced the concept of stream comprehensions to guarded functional programming (GFP*): the evaluation of a Horn logic goal results in a stream of solutions (substitutions) which generates a stream of functional expressions (the substitutions are successively applied to the same term). The resulting stream of expressions can be consumed in the expression where the stream comprehension occurs. Thus we are able to exploit the power of non-determinism which is inherent in logic programming also in functional expressions. A lazy evaluation strategy for stream comprehensions enables to cope with infinitely many solutions of goals, and to avoid unnecessary computations. We have presented a reduction semantics and a correct evaluation function for both, guarded functional programming and guarded functional programming with lazy streams. We have outlined an implementation of the lazy evaluation strategy which is based on interleaving functional and logic computations.

A precise formal specification of this operational model can be achieved by defining a continuation semantics which will be subject of future work. Furthermore, we are currently extending the guarded functional language *Guarded Term ML* [Loc88] and its implementation based on an abstract machine which integrates functional and logic components [Loc90] to support stream comprehensions and their lazy evaluation.

Further extensions to GFP* are intended to include facilities to deal with functional expressions also in the logic world. This can be achieved by extending the unification algorithm. In order to keep the functional nature, we do not envisage full equational unification but *functional unification* in the style of [SY86] or [Gol88]: expressions are reduced to their normalform before unification. An implementation of this concept can still be achieved with the coroutine mechanism described in section 3.

Acknowledgements

We are grateful to our colleagues of the ESPRIT Basic Research Action PHOENIX at GMD Karlsruhe, Imperial College and University of Nijmegen. They provided an excellent research environment, and this work has been influenced by endless discussions with them. Special thanks to Nick Graham for carfully reading earlier drafts of this paper.

References

[Boc86] Alexander Bockmayr. Conditional rewriting and narrowing as a theoretical framework for logic-functional programming -- a survey. SEKI MEMO, Univ. Kaiserslautern and Univ. Karlsruhe, 1986.

[Boc87] A. Bockmayr. A Note on a Canonical Theory with Undecidable Unification and Matching Problem. *J. Autmated Reasoning 3*, pages 379–381, 1987.

[Boc90] A. Bockmayr. *Beiträge zur Theorie des logisch-funktionalen Programmierens.* PhD thesis, Universität Karlsruhe, Fakultät für Informatik, July 1990.

[CG86] Keith Clark and Steve Gregory. PARLOG: Parallel Programming in Logic . *ACM TOPLAS*, 8(1):1–49, Jan. 1986.

[DFP86] J. Darlington, A.J. Field, and H. Pull. The unification of functional and logic languages. In D. DeGroot and G. Lindstrom, editors, *Logic Programming*, pages 37–70. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[DG89] John Darlington and Yike Guo. Constraint functional programming. Technical report, Imperial College, November 1989.

[DG90] J. Darlington and Yi-Ke Guo. Constraint Equational Deduction . In *Proc. CTRS'90, Montreal, Canada*, June 1990. (to appear in LNCS).

[Dij75] E.W. Dijkstra. Guarded Commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8), Aug. 1975.

[GL90] Yi-Ke Guo and H. C. R. Lock. A Classification Scheme for Declarative Programming Languages. - Syntax, Semantics, and Operational Models. GMD-Studien Nr. 182, August 1990.

[Gol88] H.-J. Goltz. Functional Data Term Models and Semantic Unification. In J. Grabowski, P. Lescanne, and W. Wechler, editors, *Proc. Int. Workshop on Algebraic and Logic Programming, Gaussig, GDR*, pages 158–167. LNCS 343, November 1988.

[Höl89] Steffen Hölldobler. *Foundation of Equational Logic Programming.* LNAI 353. Springer Verlag, 1989.

[Kap84] S. Kaplan. Conditional rewrite rules. *Journal of Theoretical Computer Science*, 33:175–193, 1984.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, Heidelberg, 1984.

[Loc88] Hendrik C.R. Lock. Guarded Term ML. In *Workshop on Implementations of Lazy Functional Languages*, Aspenas, Sept. 1988. Report 53, PMG, Univ. of Goteborg, Sweden.

[Loc89] Hendrik C.R. Lock. An amalgamation of functional and logic programming languages. GMD Forschungsstelle an der Universität Karlsruhe, GMD-report 408, Sept 1989.

[Loc90] H. C. R. Lock. The Implementation of Functional Logic Programming Languages. PHOENIX report GMD/Phoenix/13/1, GMD Research Laboratory Karlsruhe, 1990.

[NF89] T. Nicholson and N. Foo. A Denotational Semantics for Prolog . *ACM Transactions on Programming Languages and Systems*, 11(3):650–665, September 1989.

[Pey87] Simon L. Peyton-Jones. *The Implementation of Functional Programming Languages.* Series in Computer Science. Prentice-Hall, 1987.

[Sch88] D.A. Schmidt. *Denotational Semantics.* Wm.C. Brown Publishers, Iowa, 1988.

[SY86] P.A. Subrahmanyam and Jia-Huai You. FUNLOG: a Computational Model Intergrating Logic Programming and Functional Programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming*, pages 157–198. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Tur87] D. Turner. An Introduction to Miranda. In *S. L. Peyton Jones: The Implementation of Functional Programming Languages.* Prentice-Hall International, Series in Computer Science, 1987.

[Ued85] K. Ueda. Guarded Horn Clauses. Technical Report 103, ICOT, Tokyo, 1985.

[War83] D.H.D Warren. An abstract Prolog instruction set. Techn. Note 309, SRI International,Menlo Park,Calif., October 1983.