Extending Constructive Negation for Partial Functions in Lazy Functional-logic Languages

Juan José Moreno-Navarro

U. Politécnica, Facultad de Informática, Campus de Montegancedo,
Boadilla del Monte, 28660 Madrid, Spain,
e-mail: jjmoreno@fi.upm.es, URL: http://gedeon.ls.fi.upm.es/~jjmoreno

Abstract. In this paper the mechanism of Default Rules for narrowing-based languages proposed in [24] is adapted to lazy narrowing. Every partial definition of a function can be completed with a default rule. In a concrete function call, the default rule is applicable when the normal ones determine that they cannot compute the value of the call. Furthermore, when the goal has variables the evaluation mechanism provides constraints to the variables to make the default rule applicable. Lazy narrowing semantics are extended with the technique of constructive negation [4, 5, 27]. The main advantage is that the coroutining implementation technique described in [5], which is the basis for an efficient implementation, can be fully formalized in our framework.

1 Introduction

The integration of different declarative concepts is an active area of research. The expressive power of new declarative languages can be improved by accounting for a mature and up-to-date understanding of previously studied features. In particular, we are interested in the addition of new declarative features to *functional logic languages*, proposed for the integration of functional and logic programming (see [12]). These languages use functional syntax (conditional term rewriting systems) and narrowing as operational mechanism. *Lazy narrowing* is a strategy which only evaluates the arguments of a function application, if their evaluation is really demanded. The use of lazy narrowing enhances the expressivity of the language because it allows to write highly modular programs, to define partial and non strict functions and to use the technique of infinite objects.

The use of partial functions, even in a lazy context, has some problems because the computation of such these functions can fail. In pure functional programming the situation can be dynamically detected and solved giving, for instance, a default value. The solution is no longer directly applicable in a functional logic languages because values can be searched for applying the function. The default value can be returned only if the other rules cannot apply what imposes some constraints on the calling parameters. As a predicate p can be seen as a partially defined boolean function, Prolog negation as failure is a particular case of our problem.

In a recent paper [24] we have generalized this notion of completion into an eager functional logic language. Every (partial) definition of a function with

a number of rules can be completed by an extra default rule. In a concrete function call, the default rule is applicable when the normal ones determine that they cannot compute the value of the call. In order to manage default rules, symbolic constraints are needed to express that the arguments have not the shape required by the normal rules. The default rules also impose universal quantifications over the free variables of the normal rules. In this paper we develop the same feature in the framework of lazy narrowing. The operational mechanism is a lazy extension of *constructive negation* proposed for Prolog [4, 5, 27] which incorporates constraints as the answers into negative subgoals. For the declarative semantics we use an infinite domain that distinguishes between finite failure and the $\perp$ value for divergence, in the style of Kunen's 3-valued semantics [18].

The main idea of constructive negation is the following: in order to compute when a function call $f\ (e_1, \ldots, e_n)$ is not defined, we start a narrowing computation with $f\ (e_1, \ldots, e_n)$ as goal. The computation could be infinite but any finite part of the evaluation tree (frontier) defines where the function call can be made, hence the complement of this frontier specifies where it is undefined. An essential point for an efficient implementation is the choice of the evaluation tree that defines the frontier: If it is very small, further evaluation will be needed later; if it very large we have some overhead. We investigate here the adequate way to fix the frontier by using lazy evaluation.

In fact, the problem by itself is an example of lazy evaluation: the computation of $f(e_1, \ldots, e_n)$ is a potentially infinite process but we only use the part of it needed. This solution is sketched in Chan's papers [4, 5] by providing a coroutining implementation technique. It is clear that it is a good basis for the efficient implementation of a very heavy process. However, the presentation of the coroutining technique is not related as all with the operational semantics and it appears as an implementation trick. The combination of lazy narrowing and constructive negation allows to formalize this implementation technique. The operational understanding of the technique can be the basis for an efficient implementation of constructive negation. The formalized semantics allows us to express soundness and completeness results, which prove the adequacy of the coroutining implementation technique.

2 A Lazy Functional Logic Language with Default Rules

First of all, let us start with an example. Even if we postpone the detailed description of the language, the reader can understand the following program to motivate the new construction.

```
type nat = 0 | s(nat).
fun  member: A × list A → bool.
     member (X, [ ]) = false.
     member (X, [Y | L]) =
     X == Y → true
            □ member (X, L).

type list A = [ ]|[A|list A].
fun  first: nat × list A → list A.
     first (s (N), [X | L]) =
       member (X, first (N, L)) →
         first (N, L) □ [X | first (N, L)].
     default first (N, L) = [ ].

fun nats: nat → list nat.
    nats (N) = [N | nats (s (N))].
```

The language has a Hindley-Milner like polymorphic type system and relies on a constructor discipline. Capital letters are used for variables and small letters for constructors and user-defined functions. Functions are defined by equations (or rules). The function member is a predicate, implemented as a boolean function. The expression X == Y → true □ member (X, L) must be interpreted as an **if_then_else** construction, where the condition checks for the equality between the values stored in X and Y.

The function first shows the use of default rules. It computes the first n elements of a list and then deletes the repetitions. The function is designed to do both things at the same time. If there are not n elements in the list, less elements are considered. The partially defined version is used to compute the list using the "positive" information about the behaviour of the function. The equation describes the result when it is applied to a number greater than 0 and a nonempty list. It is completed with a default rule in order to return the empty list, what covers the "negative" information, namely two cases: the call with 0 and the call with the empty list. We can query this program as follows:

eval first (s (s(0)), [X | L]).
> *result* [X, Y] *answer* X ≠ Y, L = [Y | L']
> *result* [X] *answer* X = Y, L = [Y | L']
> *result* [X] *answer* L ≠ [Y | L'] (≡ L = [ ])
> no (more) solutions.

In the first result, the constraint $X \neq Y$ cannot be replaced by any equivalent finite set of equalities.

The function nats defines the infinite list of natural numbers greater than N. It can only be managed with lazy evaluation. Another query should be:

eval first (s (s (s(0))), nats (X)).
> *result* [X, s(X), s(s(X))]
> no (more) solutions.

when the reader can see the interaction of lazy evaluation, logical variables and default rules.

The concrete syntax is a simplification of the functional-logic language BABEL [23, 16]. Terms t and expressions e are defined as follows:

$t ::= X$	% X variable	$e ::= t$	% term
$\mid (t_1, \ldots, t_n)$	% tuples	$\mid (e_1, \ldots, e_n)$	% tuples
$\mid d\ (t)$	% d constructor	$\mid d\ (e)$	% d constructor
		$\mid f\ (e)$	% f user-defined function

and all of them must be well typed.

The syntax allows to build expressions involving some *primitive function symbols*: $\neg b$ (negation – moves *true* to *false* and vice versa), $(b_1\ ,\ b_2)$ (conjunction), $(b_1\ ;\ b_2)$ (disjunction), $(b \to e)$ (guarded expression, meaning: **if** b **then** e **else** undefined), $(b \to e_1 \Box e_2)$ (conditional, meaning: **if** b **then** e_1 **else** e_2), and $(e_1 == e_2)$ (weak equality, both expressions denote the same object), where b is a boolean expression.

Programs consist of declarations of types and *defining* or *default rules* for every function symbol f, with the following shape, where guards (that may contain free variables – i.e. not appearing in the left hand side) are optional:

Defining rules	Default rules
$\underbrace{f\ (t_1,\ldots,t_n)}_{\text{left hand side}} = \underbrace{\underbrace{\{b \rightarrow\}}_{\text{guard}}\ \underbrace{e}_{\text{body}}}_{\text{right hand side}}$	**default** $f\ (X_1,\ldots,X_n) = b \rightarrow e$

These functions are functions in the mathematical sense. In order to ensure confluence, some restrictions must be imposed imposed (see [23]).

Now, we can intuitively define the meaning of a function definition. The left column shows a set of function rules and the right column their meaning:

$$\begin{array}{ll} f(t_1(\overline{X}^1)) & = b_1(\overline{X}^1,\overline{Y}^1) \rightarrow e_1 \\ & \ldots \\ f(t_n(\overline{X}^n)) & = b_n(\overline{X}^n,\overline{Y}^n) \rightarrow e_n \\ \textbf{default}\ f(\overline{X}) & = b(\overline{X},\overline{Y}) \rightarrow e \end{array} \qquad f(\overline{Z}) = \begin{cases} e_1 & \text{if } \exists \overline{X}^1(\overline{Z} = t_1 \wedge \exists\ \overline{Y}^1 b_1) \\ \ldots & \\ e_n & \text{if } \exists \overline{X}^n(\overline{Z} = t_n \wedge \exists\ \overline{Y}^n b_n) \\ e & \text{if } \exists \overline{Y} b \wedge \delta\ (f(\overline{Z})) \end{cases}$$

where δ is the definitionless operator that means that the defining rules for f do not define $f\ (\overline{Z})$.

Let us conclude the section with another example that involves infinite objects and variable quantification. We want to calculate the integer square root of a natural number. We compute it as the nth element of the infinite list of integer square roots of the natural numbers. We accumulate the previous square root and it is increased when a perfect square is found. A natural number is a perfect square if it is the product of another natural number by itself. Otherwise the number is not perfect. This last statement corresponds to a default rule which involves an implicit universal quantification. Arabic numbers (1, 2, 46, ...) are sometimes used instead of their successor representation.

```
type square_att = perfect | no_perfect.
     % Some rules for arithmetic operations plus, times, <, ....
fun  sq_att : nat → square_att.
     sq_att (N) = (Y < X, times (Y, Y) == X) → perfect.
     default sq_att (X) = no_perfect.
fun  int_roots: nat × nat → list nat.
     int_roots (X, Y) = sq_att (X) = perfect → [s(Y) | int_roots (s(X), s(Y))]
                                             □ [Y | int_roots (s(X), Y)]
fun nth: list A → nat → A.          fun int_root: nat → nat.
    nth ([X | L], 0) = X.               int_root (N) =
    nth (([X | L], s (N)) = nth (L, N).             nth (int_roots (2, 1), N).
                        eval int_root (2197).
                              > result 46
```

3 Declarative Semantics of Default Rules

This section sketches the declarative semantics of our language by extending the semantics of [23]. It is worth to mention that the semantics have some "external" similarities with the strict case (reported in [24]). However, the similarities are mainly apparent because both semantics are essentially different. The use of an infinitary domain complicates the construction and the results. The presentation is intentionally similar to allow a deep comparison of both semantics.

3.1 The Domain

The *infinitary Herbrand Universe* $\mathcal{H}$ is the set of the (finite or infinite) correctly typed terms built up with the constructors of the program and two failure values: $fail$ (for finite failure) and $\bot$ (for divergence). We distinguish two kinds of elements in $\mathcal{H}$: *finite elements* and *total* elements (those elements without any occurrence of $fail$, $\bot$). The ordering of $\mathcal{H}$ is defined as the transitive closure of:

- $\bot \sqsubseteq s$ for every s
- $(t_1, \ldots, t_n) \sqsubseteq (s_1, \ldots, s_n)$ if $t_1 \sqsubseteq s_1, \ldots, t_n \sqsubseteq s_n$
- $fail \sqsubseteq s$ for every concrete element s
- $d\ (t) \sqsubseteq d\ (s)$ if d is a constructor and $t \sqsubseteq s$

We denote $t \sqcup s$ the least upper bound of two consistent elements t, s. $\bigsqcup S$ denotes the l.u.b. of a consistent set S.
A function $f : \mathcal{H} \to \mathcal{H}$ is continuous if it monotonous and for every consistent (infinite) set S $f(\bigsqcup S) = \bigsqcup f(S)$.

Our model combines infinite domains and Kunen's 3-valued semantics [18]. Two different values for failure are needed because $false$ is a correct result for functions and predicates.

3.2 Interpretations, Environments and Valuations

A Herbrand interpretation I is a collection of well typed continuous functions $f_I : \mathcal{H}^n \to \mathcal{H}$, one for every function f such that $f_I(\bot) = \bot$ and $f_I(fail) = fail$. Interpretations can be equipped with the partial ordering:

$$I \sqsubseteq J \text{ iff for all } f/n \in FS\ f_I(\overline{s}) \sqsubseteq f_J(\overline{s}) \text{ for all } \overline{s} \in \mathcal{H}^n$$

The domain of Herbrand interpretations is noted $\mathcal{H}\bot\mathcal{INT}$.
An environment is any mapping $\rho : VS \to \mathcal{H}$ such that $\rho(X)$ has the same type of the variable X. We say that ρ is total if no $\rho(X)$ contains $\bot$ or $fail$

Now, we proceed with the definition of the valuation $[\![e]\!]_I(\rho)$ for a given well typed expression e into a concrete interpretation I under an environment ρ by induction on the syntactical structure of the expression.

$$\begin{aligned}
[\![X]\!]_I(\rho) &= \rho(X) \quad \text{for } X \in VS \\
[\![(e_1, .., e_n)]\!]_I(\rho) &= ([\![e_1]\!]_I(\rho), .., [\![e_n]\!]_I(\rho)) \\
[\![d\ (e)]\!]_I(\rho) &= d\ ([\![e]\!]_I(\rho)) \quad \text{for } d/n \in CS, n \geq 0 \\
[\![f\ (e)]\!]_I(\rho) &= f_I\ ([\![e]\!]_I(\rho)) \quad \text{for } f/n \in FS, n \geq 0 \\
[\![e_1 = e_2]\!]_I(\rho) &= eq([\![e_1]\!]_I(\rho), [\![e_2]\!]_I(\rho)) \\
[\![\neg b]\!]_I(\rho) &= not([\![b]\!]_I(\rho)) \\
[\![b_1, b_2]\!]_I(\rho) &= and([\![b_1]\!]_I(\rho), [\![b_2]\!]_I(\rho)) \\
[\![b_1; b_2]\!]_I(\rho) &= or([\![b_1]\!]_I(\rho), [\![b_2]\!]_I(\rho)) \\
[\![b \to e]\!]_I(\rho) &= if([\![b]\!]_I(\rho), [\![e]\!]_I(\rho)) \\
[\![b \to e_1 \Box e_2]\!]_I(\rho) &= if_else([\![b]\!]_I(\rho), [\![e_1]\!]_I(\rho), [\![e_2]\!]_I(\rho))
\end{aligned}$$

where *and*, *not*, and *or* are the 3-valued logical connectives, and:

$$eq(s_1, s_2) = \begin{cases} true & \text{if } s_1 = s_2 \text{ is finite and total} \\ false & \text{if } s_1, s_2 \text{ are inconsistent} \\ fail & \text{if } s_1 = s_2 = fail \\ \bot & \text{otherwise} \end{cases}$$

$$if(b, s) = \begin{cases} s & \text{if } b = true \\ fail & \text{if } b = false \\ & \text{or } b = fail \\ \bot & \text{otherwise} \end{cases} \qquad if\_else(b, s_1, s_2) = \begin{cases} s_1 & \text{if } b = true \\ s_2 & \text{if } b = false \\ fail & \text{if } b = fail \\ \bot & \text{otherwise} \end{cases}$$

3.3 Models

We start with some auxiliary definitions: Given a function f, a program Π, and a Herbrand interpretation I we define Ythe functions $T_I(f)$, $\widehat{T}_I(f) : \mathcal{H} \to \mathcal{P}(\mathcal{H})$ as follows:

$$T_I(f)(s) = \{[\![R]\!]_I(\rho) / f\ (t) = R \in \Pi, \rho(t) = s\}$$

$$\widehat{T}_I(f)(s) = \{[\![R]\!]_I(\rho) / \mathbf{default} f\ (X_1, \ldots, X_n) = R \in \Pi, \rho(X_1, \ldots, X_n) = s\}$$

where ρ is total environment.

We also define the definitionless function $\delta_I(f) : \mathcal{H} \to bool$ such that

$$\delta_I(f)(s) = \begin{cases} true & \text{if } T_I(f)(s) = \emptyset \text{ or } T_I(f)(s) = \{fail\} \\ \bot & \text{if } \bot \in T_I(f)(s) \\ false & \text{otherwise.} \end{cases}$$

The δ function establishes where the function f has not a definition using the defining rules. Notice that the definitions of T_I and δ involve some implicit universal quantification if a defining rule contains free variables in the guard.

An interpretation I is a model of a program Π (in symbols $I \models \Pi$) if

a).- I is a model of every defining rule in Π, and

b).- for every default rule $f\ (X_1, \ldots, X_n) = e$ in Π $\quad f_I(\rho(\overline{X})) = [\![e]\!]_I(\rho)$ if $\delta_I(f)(\rho(\overline{X})) = true$ for any environment ρ.

I is a Herbrand model of a defining rule $L = R$ iff $[\![L]\!]_I(\rho) \sqsupseteq [\![R]\!]_I(\rho)$ for all environments ρ.

3.4 Interpretation transformer and the semantics of a program

The interpretation transformer associated to a program Π is the mapping $\mathcal{T}_\Pi : \mathcal{H}\bot\mathcal{I}\mathcal{N}\mathcal{T} \to \mathcal{H}\bot\mathcal{I}\mathcal{N}\mathcal{T}$ defined as follows: for any interpretation I, $\mathcal{T}_\Pi(I)$ is the Herbrand interpretation J such that for any $f \in FS$ and a well typed element $s \in \mathcal{H}$

$$f_J(s) = \begin{cases} m \text{ if } m \sqsupset fail & \text{where} \\ n \text{ if } \delta_I(f)(s) = true & m = max\ T_I(f)(s) \\ \bot\ otherwise & n = max\ \widehat{T}_I(f)(s) \end{cases}$$

Proposition 1. *$\mathcal{T}_\Pi$ is well defined and monotonous.*

Proof. $\mathcal{T}_\Pi$ is well defined due to the non-ambiguity property. The set $T_I(f)(s)$ can only have a single concrete value, then the maximum exists. The definition of the transformer allows us to prove monotonicity. Notice that the distinction between $fail$ and $\bot$ means that the unknown $\bot$-values cannot be used by a default rule. $fail$ is only used when a finite failure is calculated and then the default rules can be applied.

Unfortunately, $\mathcal{T}_\Pi$ is not always continuous. Monotonicity ensures the existence of a least fixpoint, but it may occur at any recursive ordinal. Following Fitting and Ben-Jacobs [9], the natural cut off point for computability is after ω steps. Fortunately, this is what we can compute with our operational semantics.

Therefore, we adopt

$$I_\Pi = T_\Pi \uparrow \omega$$

as the semantics of the program Π.

I_Π is an interpretation and the definitions for interpretations apply to it. We call δ the function δ_{I_Π}. The complement of this function, called Δ is also interesting: $\Delta(f\ (s)) = not\ \delta(f)(s)$. Intuitively $\Delta(f\ (e))$ means that the expression $f\ (e)$ is positively defined, i.e. can be calculated without the default rule.

4 Operational Semantics

In order to provide a stepwise definition for all the concepts we will first present how lazy narrowing works with constraints. In the next subsection the definition of lazy narrowing will be extended in order to cope with default rules. This section describe how to compute with default rules. The key idea is to manage the δ function in a syntactic way.

4.1 Lazy Narrowing with Constraints over the Herbrand Universe

A computation involves a constraint c and an expression e. The constraint c contains equalities and disequalities between terms. The simplest *normal form* of a constraint (see [17]) is the conjunction of (possibly universally quantified) equalities and disequalities between a variable and an expression. In symbols, a constraint in *normal form* is:

$$\underbrace{\bigwedge (X_i = e_i)}_{\text{positive information}} \quad \wedge \quad \underbrace{\bigwedge \forall\, \overline{Z}_j\ (Y_j \neq e'_j)}_{\text{negative information}}$$

where each X_i appears only in $X_i = e_i$, none e'_j is equal to Y_j and the universal quantification could be empty (leaving a simple disequality).

A more complex notion of constraint normal form is used in [24] for innermost narrowing, however it is not longer valid because the possibility of constraints between infinite terms. Disjunctions of disequalities are included in the constraint. Disjunctions appear because of disequalities between tuples, e.g. $(t_1 \ldots t_n) \neq (s_1 \ldots s_n)$ is equivalent to $t_1 \neq s_1 \vee \ldots \vee t_n \neq s_n$. Laziness implies that a disequality may hold even between infinite objects. This should imply an infinitary disjunction. Similarly, we cannot force the evaluation of expressions e_j's to a (finite) term without loosing completeness. For this reason we use our simple notion of normal form, and backtracking is used for managing the situations where disjunctions are introduced, namely disequations between constructor applications. The interested reader can consult [6, 20] for papers devoted to this and related subjects.

The equalities collected into the constraint can be seen as the substitutions in usual narrowing, similarly as it is done within the CLP scheme [15]. A more abstract and general description of the operational semantics of the constraint functional logic programming paradigm can be found in [19].

The notation $c \vdash c'$ indicates that the constraint c' is the lazy simplification of c. $c \vdash false$ means that the constraint c is unsatisfiable. In order to maintain the normal form representation of constraints a strong interaction between $\vdash$ and the lazy narrowing relation is needed. As disjunctions are not introduced in the constraint representation, $\vdash$ is a non deterministic relation. In general, we consider two alternatives for $Y \neq d\ (e)$: either $Y = d'\ (X)$ for some $d' \neq d$ and a new variable X, or $Y = d\ (X)$, but $X \neq e$. Notice that the first alternative is really *lazy*, that is, it does not require further evaluation of e. Furthermore, the second alternative only requires the evaluation of the argument to HNF.

Due to the lack of space, we are not able to describe the complete rules for $\vdash$. The rules described in [17] and formalized in [1] can be adapted to include

universally quantified disequalities in the vein of [24]. $\vdash$ is also responsible of the decision about if a constraint c is satisfiable or not. In general, it is enough to have each e_i, e'_j in *head normal form* (HNF)., i.e. a variable or an expression with a constructor at the top.

The *one step lazy narrowing relation* is denoted $c \parallel e \implies c' \parallel e'$, where c' is not $false$. It is specified by the following rules[1]:

1. *Narrowing the argument in a construction*

$$\frac{c \parallel e \stackrel{*}{\implies} c' \parallel e'}{c \parallel d\ (e) \implies c'' \parallel d\ (e')} \quad \text{if } d \in CS \text{ and } c \wedge c' \vdash c''$$

2. *Narrowing the arguments in a tuple*

$$\frac{c \parallel e_i \stackrel{*}{\implies} c' \parallel e'_i}{c \parallel (e_1, \ldots, e_i, \ldots, e_n) \implies c'' \parallel (e_1, \ldots, e'_i, \ldots, e_n)} \quad 1 \leq i \leq n \text{ and } c \wedge c' \vdash c''$$

3. *Outermost rule application* $\quad c \parallel f\ (e) \implies c' \parallel e''$

   if
   - e is evaluated as demanded by the rules for f and there is a variant $f\ (t) = \{b \rightarrow\} e'$ of one rule in the program (sharing no variables with the goal) such that $f\ (t)$ and $f\ (e)$ are unifiable with m.g.u. $\sigma = \sigma_{in} \dot{\cup} \sigma_{out}$ (where σ_{in} collects the bindings for variables in the t_i and σ_{out} records the bindings for variables in e), and e'' is $(\{b \rightarrow\} e')_{\sigma_{in}}$
   - $c \wedge \bigwedge (X_i = e'_i) \vdash c'$, where $\sigma_{out} = \{\ldots, X_i / e'_i, \ldots\}$

4. *Inner narrowing step*

$$\frac{c \parallel e \stackrel{*}{\implies} c' \parallel e'}{c \parallel f(e) \implies c'' \parallel f(e')}$$

   if e is not evaluated yet as demanded for the rules for f and $c \wedge c' \vdash c''$.

5. *Rules for the equality*

$$c \parallel (e_1 = e_2) \implies c' \parallel true \quad \text{if } c \wedge (e_1 = e_2) \vdash c'$$
$$c \parallel (e_1 = e_2) \implies c' \parallel false \quad \text{if } c \wedge (e_1 \neq e_2) \vdash c'$$

6. *Interaction with constraints*

$$\frac{c \parallel e_1 \stackrel{*}{\implies} c' \parallel e'_1}{c \wedge ce \parallel e \implies c'' \parallel e}$$

   where ce is $t = e_1$ $(e_1 = t, t \neq e_1, e_1 \neq t)$
   e_1 is not in HNF and $c \wedge c' \wedge ce' \vdash c''$,
   ce' is $t = e'_1$ $(e'_1 = t, t \neq e'_1, e'_1 \neq t$ respectively)

The rest of the primitive functions can be computed by some predefined rules (see [23]), which must be added to every program.

The notation $c \parallel e \equiv c_0 \parallel e_0 \implies \ldots c_i \parallel e_i \ldots \implies c_n \parallel e_n \equiv c' \parallel e'$, or simply $c \parallel e \stackrel{*}{\implies} c' \parallel e'$ denotes the *lazy narrowing relation* (composition of several narrowing steps).

There is some degree of freedom in the previous scheme. On one hand, there could be several points, mainly the selected expression to be reduced into a

[1] The rules assume that the condition holds, i.e. the resulting constraint is not $false$. Furthermore, the rules are not applicable if the input constraint c is not in normal form, except rule 5

tuple (rule 2), where one of the previous rules are applicable. We call this point a *lazy redex* and the order of reduction should be fixed by a *computation rule* $\mathcal{R}$. The usual implemented rule is the selection of the left-most redex that it is not yet in the demanded form. As we have mentioned, the $\vdash$ simplification is also nondeterministic. For simplicity, we assume that $\mathcal{R}$ covers also this case. In particular $\mathcal{R}$ decides when the satisfiability check is performed. As the check usually imposes several evaluations it is desirable to delay it at most as possible and only the evaluations demanded by other rules can be performed.

On the other hand, the notion of *as demanded by the rules for* f is not clearly defined. In general, not all the rules demand the same evaluation for all the arguments. Several papers have discussed the importance of fixing statically the degree of evaluation (avoiding the nontermination and the reevaluation problem see [21, 22]). One proposal [21] uses a program transformation in order to get a program with the following property: All the rules demand HNF or nothing. However, the more the rules demand the more efficient is the process. The alternative solution is the use of abstract interpretation to fix the degree of evaluation demanded by a rule [22]. *Demandedness analysis* is used to infer the so called *dependent demand patterns*, that specify the finite or infinite (regular) degree of evaluation for every function, without penalization of the lazy behaviour. Consider, for instance, the function nth of the second example. The lhs's of the program demand HNF for both arguments. However, in order to compute a result, we can infer that the second argument needs normal form instead. The computation of an expression $nth(a\_list, f(e))$ can force $f(e)$ to return a term.

For the purpose of this paper, we assume that some degree of evaluation p_f for every program function f is detected, enough to apply the rules. By abuse of notation, we will still call each p_f the *demand pattern* for f. We also assume a method (usually ensured by the implementation) to detect if an expression e has the shape of a demand pattern p_f, in symbols $e \preceq p_f$.

In summary, lazy narrowing is guided by a computation rule $\mathcal{R}$ and a set of demand patterns $\mathcal{P} = \{p_f / f \in FS\}$.

4.2 Children, Narrowing Tree and Frontier

Given a constraint expression $c \parallel e$ and a program Π we call any possible application of the one step narrowing relation guided by $\mathcal{R}, \mathcal{P}$ a *child* of $c \parallel e$. All the possible narrowing reductions of a constraint expression $c \parallel e$ given a program Π form the *narrowing tree* for $c \parallel e$. Every node is labelled with a constraint expression[2].

The possible paths into the narrowing tree of a goal expression $c \parallel e$ can be classified as:

- *success* when $c \parallel e \stackrel{*}{\Longrightarrow} c' \parallel t$ with t a term and c' satisfiable;
- *failure* when $c \parallel e \stackrel{*}{\Longrightarrow} c' \parallel e'$, e' is not a term and e' is not further narrowable or c' unsatisfiable; and
- *nontermination:* when we may still have $c \parallel e \stackrel{*}{\Longrightarrow} c' \parallel e'$ where e' is not a term and e''s constructors give a partial result.

[2] The references to Π, $\mathcal{R}$, and $\mathcal{P}$ will be omited when no ambiguity is possible.

When a successful narrowing relation reaches a term $c' \parallel t$ we speak of a *computation* for the *goal* $c \parallel e$ with *result* t and *answer* c'.

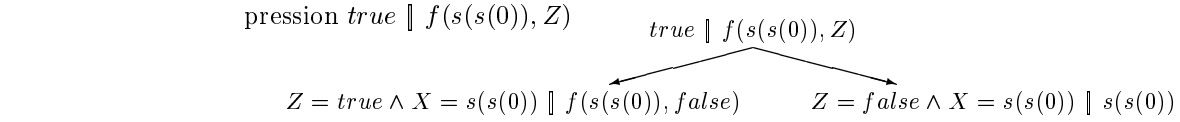
A very important notion for our purpose is the concept of the *frontier* of a narrowing tree. We differ from the classical notion. Our definition depends on the demand patterns selected. Once one has chosen them, the frontier is unique. We want to stress the fact that, from the operational point of view, this is one of the main contributions of the paper.
Let p_f be a demand pattern in $\mathcal{P}$. The *p_f-frontier of an expression* $c \parallel e'$, where $e' = f(e)$, is the set of nodes $\{c_1 \parallel e_1, \ldots, c_m \parallel e_m\}$ of the narrowing tree of $c \parallel e$ such that every narrowing reduction of $c \parallel e$ is either a failure or passes through exactly one node in the set, and each e_i is the first expression in the branch such that $e_i \preceq p_f$.

If the p_f-frontier is well defined then it is finite and unique, what is not true in the classical notion of frontier. The argument is evaluated as least as possible to preserve the lazy behaviour. If the narrowing tree has an infinite branch without getting an expression $e_i \preceq p_f$ the frontier does not exist.

Suppose we use the program of our first example adding the function f, defined by the rules:

$$\mathsf{f\ (X,\ true)} = \mathsf{f\ (X,\ false)}. \qquad \mathsf{f\ (X,\ false)} = \mathsf{X}.$$

and p_{first} requires HNF for both arguments. Now, if we consider the expression $true \parallel first(f(s(s(0)), Z), L)$ and the following narrowing tree for the subexpression $true \parallel f(s(s(0)), Z)$



we get the frontier $\{Z = true \wedge X = s(s(0)) \parallel f(s(s(0)), false), Z = false \wedge X = s(s(0)) \parallel s(s(0))\}$.

The rest of the section adapts the ideas of [24] and constructive negation to this refined concept of frontier.

4.3 Narrowing for default rules

In order to compute with default rules, the δ function is managed syntactically. The semantical definition of the δ function includes an implicit universal quantification, that is moved into a explicit $\forall$-quantification. δ-expressions take the form $\forall\ \overline{X}\ \delta(c \parallel f(e))$ that must be read "under the constraint c, $f(e)$ is undefined using the defining rules". Therefore, the question to answer is "when is the expression $f(e)$ is defined ?". The expression is defined if e is as defined as demanded by the rules for f and then at least one rule is applicable giving a defined result. From the operational semantics point of view, this means two situations:

1. The argument expression e is as defined as p_f.
2. One rule is applicable, i.e. one unification is possible and the corresponding *rhs* is defined.

To manage the first point we can adapt the result of [24]: if the p_f-frontier of $c \parallel f(e)$ is $F = \{c_1 \parallel e_1, \ldots, c_n \parallel e_n\}$ then

$$\Delta(c \parallel e) \leftrightarrow \exists\ \overline{X}_1\ (c_1 \wedge \Delta(e_1)) \vee \ldots \vee \exists\ \overline{X}_n\ (c_n \wedge \Delta(e_n))$$

where $\overline{X}_i$ are the free variables in $c_i \parallel e_i$ which are not in $c \parallel e$. As the δ function is the negation of the Δ function, this result can be used to calculate a δ expression by the negation of the right hand side formula. The notion of *normal complement* of a frontier is used for this purpose.

The negation of the second point is straightforward. Let $c \parallel e_i$ an element of the p_f-frontier. Either e_i is not unifiable with any *lhs* or e_i is unifiable with the *lhs* of a rule but the *rhs* is undefined. We formalize these situations by the notion of *rule complement* of a frontier. Let us describe in details these two concepts:

Let $F = \{c_1 \parallel e_1, \ldots, c_n \parallel e_n\}$ be the p_f-frontier of $c \parallel f(e)$ and V a set of variables (denoting the free variables in the goal expression). The *normal complements* of F under V are all the possible conjunctions of complements of each $c_i \parallel e_i$ under V: $c'_1 \wedge \ldots \wedge c'_m \parallel b_1, \ldots, b_m$ is a complement of F iff each $c'_i \parallel b_i$ is a complement of $c_i \parallel e_i$ under V. The *rule complements* of F under V are the rule complements of every constrained expression $c_i \parallel e_i$ of F under V.

In order to define the complements of a constraint expression $c \parallel e$, let us briefly discuss the quantification of variables. First, we only need to focus on those useful variables, i.e. those that are free in the goal expression. We collect them in the set V. A variable X not in V appearing in the positive part of the constraint as $X = t$ can be eliminated. If t is a variable $X' \in V$ we substitute X for X' in $c \parallel e$. Otherwise, the equation is irrelevant.

Next, we identify the variables that will have a universal quantification when the complement is calculated. They are the free variables in e that are neither in V nor in the positive part (that have an implicit quantification). We collect them in the set U. The inequalities with variables in U cannot be separated from e. The rest of the disequalities can be negated separately from e. When a disequality is negated it is moved into a equality. This equality have not quantification even if the original disequation is not quantified because the free variables are (implicitly) quantified outside. In summary, we can organize the simplification of c as:

$$c = c' \wedge c'' \qquad \text{where } c' = \bigwedge_{i=1}^{m} (X_i = e_i) \wedge \bigwedge_{j=1}^{n} \forall\ \overline{Z}_j\ (Y_j \neq e's_j)$$

and c'' collects all the inequalities with a variable in U.
The *complements* of $c \parallel e$ under V are:

- $\forall\ \overline{Z}^i\ (X_i \neq e_i) \parallel true$, for all $i \leq m$, where $\overline{Z}^i$ are the variables of e_i that are not quantified outside, i.e. $var(e_i) \cap V$.
- $\bigwedge_{i=1}^{m} (X_i = e_i) \wedge (Y_k = e'_k) \quad \wedge \quad \bigwedge_{j=1}^{k-1} \forall\ \overline{Z}_j\ (Y_j \neq e'_j) \parallel true$, for all $k < n$.
- $X = \forall\ \overline{Z}\ \delta(c'' \parallel e) \wedge c' \parallel true$, if e is not a term[3], where X is a new variable and $\overline{Z}$ are the variables in U.

Let $\{f(t_1) = e'_1, \ldots f(t_l) = e'_l\}$ be the set of rules for f in Π. The *rule complements* of $c \parallel e$ under V are:

[3] $\delta(t) = true$ if t is a term

- $c \wedge \bigwedge_{k=1}^{l} \forall\, \overline{Z}^k\, e \neq t_k \parallel true$, where $\overline{Z}^k = var(t_k) \cup (var(e) \cap V)$.
- $c' \wedge e = t_k \wedge X = \forall\, \overline{Z}\ \delta(c'' \parallel e'_k) \parallel true$ (where X is a new variable and $\overline{Z}$ are the variables in U) for each $1 \leq k \leq l$ such that e'_k is not a term.

The δ-expression can affect a predefined function. In this case, the predefined rules can be applied. In practice, predefined functions are implemented directly and it is convenient to deduce some simplification rules for δ-expressions.

Let us come back to our example. The computed frontier has a single goal variable Z. Equalities for X are eliminated.

The complements of the frontier are: $Z \neq true \parallel true$ and $Z = true \wedge Z' = \delta(true \parallel f(s(s(0)), false)) \parallel true$, and $Z \neq false \parallel true$, respectively

Finally, the normal complements are obtained by combining the previous complements: $Z \neq true \wedge Z \neq false \parallel true$ (unsatisfiable in the boolean domain), $Z = true \wedge Z = false \wedge Z' = \delta(true \parallel f(s(s(0)), false)) \wedge true$ (also unsatisfiable), and $Z = true \wedge Z' = \delta(true \parallel f(s(s(0)), false)) \parallel true$.

The rule complements are:

1. $Z = true \wedge \forall\, N\ s(N) \neq s(s(0)) \parallel true$ (unsatisfiable).
2. $Z = false \wedge \forall\, N\ s(N) \neq s(s(0)) \parallel true$ (unsatisfiable),
3. $Z = true \wedge \forall\, X', L'\ L \neq [X'|L'] \parallel true$,
4. $Z = false \wedge \forall\, X', L'\ L \neq [X'|L'] \parallel true$,
5. $Z = true \wedge N = s(0) \wedge L = [X'|L'] \wedge Z' = \delta(true \parallel member(X', first(L', s(0)) \rightarrow first(L', s(0)) \square [X'|first(L', s(0))]) \parallel true$, and
6. $Z = false \wedge N = s(0) \wedge L = [X'|L'] \wedge Z' = \delta(true \parallel member(X', \ldots) \parallel true$.

The narrowing rules are completed to use the default rule (if present) and to compute a $\delta(c \parallel e)$ expression:

6. *Default rule*
$$\frac{\delta(c \parallel f\ (e_1, \ldots, e_n)) \stackrel{*}{\Longrightarrow} c' \parallel true}{c \parallel f\ (e_1, \ldots, e_n) \Longrightarrow c'' \parallel e\sigma}$$
where **default** $f\ (X_1, \ldots, X_n) = e \in \Pi$, $\sigma = \{.., X_i/e_i, ..\}$ and $c \wedge c' \vdash c''$.
7. *Definitionless expressions rule*
Let us suppose that there exists a p_f-frontier F of the expression $c \parallel e'$, where $e' = f(e)$, and let V be the set of free variables of $c \parallel e'$ that are not in $\overline{X}$.
$$\forall\, \overline{X}\ \delta(c \parallel e') \Longrightarrow c \parallel b$$
if $(c \parallel b)$ is either a normal complement or a rule complement of F under V

For each complement of F we have a different child in the narrowing tree. If F does not exist the rule is not applicable and the δ expression cannot be reduced (remains undefined as indicated by the declarative semantics).

4.4 Soundness and Completeness

Finally, we can establish the soundness and completeness of our narrowing semantics. Due to the lack of space we will omit the proofs.

Theorem 1. Soundness
Let Π be a program. Any narrowing sequence $c \parallel e \stackrel{*}{\Longrightarrow} c' \parallel e'$ computes a sound outcome in the sense that

$$[\![e]\!]_{I_\Pi}(\rho) \sqsupseteq [\![e']\!]_{I_\Pi}(\rho) \qquad \textit{for all environments } \rho \textit{ satisfying } c \wedge c'.$$

The expected soundness result for successful computations comes when e' is a term (and its valuation corresponds to itself).

Furthermore we can claim the following completeness theorem.

Theorem 2. Completeness
Let Π be a program, e an expression, c a constraint, s an element in $\mathcal{H}$ and θ a substitution that binds any variable of c, e into a ground term (ground substitution). If $[\![(c \rightarrow e)\theta]\!] \sqsupseteq s$ then there exists a narrowing sequence $c \parallel e \stackrel{*}{\Longrightarrow} c' \parallel t$, (with t a term), and a ground substitution σ such that $t\sigma = s$, and $c_\sigma \wedge c_\theta \wedge c'$ is satisfiable (where c_σ, c_θ denote the constraints with only positive part σ, θ).

Proof. (Idea): The proof proceeds by using the $\mathcal{T}_\Pi$ operator, combining the completeness of lazy narrowing and the completeness proof of [27].

5 Related work

The work uses some of the techniques developed for constructive negation [4, 5, 27, 7]. However, they are adapted to a more general framework. The reader can find a discussion about the differences in [24]. In [4, 5] an implementation technique based on coroutining is proposed: every negated predicate is annotated with some *delay* clauses to suspend the execution until the variables are enough instantiated. Then the negation with respect to the clauses is computed (similarly to our rule complements) which could introduce new negated predicates. This technique is generalized here and it is described in the more abstract context of lazy evaluation. The key point is the inclusion of new δ-expressions into the constraint, where they are handled lazily.

We compare our work with some related papers in to areas. First, we deal with works about the combination of laziness and constraints. As far as we know, only [19, 17, 1] attack the problem. The first one introduces a general framework to integrate functional programming, logic programming and constraints. [17] presents a method to compute equality and disequality constraints over the (infinitary) Herbrand universe in a lazy way. The description is very related to an abstract machine implementation. The work reported in [1] formalizes this approach in the framework of [19]. Our work needs to use their way of combining laziness and constraints but goes beyond this problem.

On the other hand, there are very few papers devoted to the integration of narrowing and constructive negation. Maybe [26] is the closest approach but they treat a different problem: they compute the solutions to a disequation $f(t) \neq s$ in a Conditional Term Rewriting system without extra variables by computing the answers for $f(t) = s$ and then negating the obtained formula. The undefined values for $f(t)$ are not taken into account (or f must be total). The evaluation mode is eager and no strategy is discussed about the depth of the frontier. The work in [8] describes an innermost narrowing mechanism to calculate disunification (i.e. disequations over the Herbrand universe modulo an equational theory).

A special use of innermost narrowing and negative information appears in the languages SLOG [10] and ALF [11]. Narrowing interleaves with *simplification*

using some rules for pure rewriting. Under the CWA, sometimes it is possible to define a rewriting rule indicating when a function fails. Besides the fact that then computation can be optimized, an infinite computation ($\perp$) can be moved into a finite failure ($fail$). Recently, the technique has been extended to lazy narrowing [13]. Although the technique is independent of our work and can be used to optimize it also, sometimes the effects are similar.

6 Conclusion

In this paper we have studied the completion of partial functions with a default rule in lazy functional logic languages. The lazy narrowing rule has been modified to cope with these new rules. The basis of the technique of constructive negation ("subderivations are used to detect when a function call will finitely fail by using the defining rules") is kept but it is complemented with lazy narrowing to compute the exact part of the subderivation needed. The coroutining implementation technique is fully formalized as the natural lazy operational semantics of the language in such a way that lazy evaluation is an effective solution to a heavy process. We want to further explore the use of this technique in a concrete implementation.

PROLOG programs with negation can be expressed in our framework. The programmer can write a default rule with body *false* to express the negated part of a predicate. But he/she can also write a defining rule with body *false*, emulating negation as failure. This fact carries good and bad news. The good news is that the programmer can freely mix explicit negation and negation as failure. Kowalski pointed out the advantages of this distinction for knowledge representation.

The bad news is that default rules are, at least, as difficult to implement as negation in logic programming. We can adapt the transformational approach [2, 3] to treat negation in Prolog. The negative information is expressed by means of new predicates, that are added to the program. In our case, this approach can be easily adapted when the program has no guards or guards without free variables. The default rule can be expressed as several normal rules. For instance, in the *first* example, the new rules are:

$$\mathsf{first}\ (0,\ [X \mid L]) = [\,]. \qquad \mathsf{first}\ (s\ (N),\ [\,]) = [\,].$$

Nevertheless, the program could produce infinitely many solutions when a constraint of the $X \neq Y$ appears.

In the presence of logical variables in the guards, the transformation is still possible but the guard(s) of the new rule(s) contain arbitrary formulas and, in particular, universal quantifications, hard to be efficiently implemented. Prolog versions with these facilities, like Gödel [14] or NU-Prolog [25], usually perform residuation which is incomplete in general.

Although we restrict ourselves to the Herbrand Universe, following [27] it is more general (and natural) to study the problem in a CLP framework. However, to our knowledge only [19] addresses seriously the definition of the Constraint Functional Logic Programming paradigm.

As a future work we plan to implement a prototype by modifying the current lazy BABEL implementations [21, 22].

Acknowledgements

This research was supported in part by the spanish project TIC/93-0737-C02-02.

References

1. P. Arenas, A. Gil, F. López-Fraguas. Combining Lazy Narrowing with Disequality Constraints. *PLILP'94, Springer LNCS 844*, 385-399, 1994.
2. R. Barbuti, D. Mancarella, D. Pedreschi, F. Turini. Intensional Negation of Logic Programs. *Proc. TAPSOFT'87, Springer LNCS* 250, 96-110, 1987.
3. R. Barbuti, D. Mancarella, D. Pedreschi, F. Turini. A Transformational Approach to Negation in Logic Programming. *J. of Logic Programming*, 8(3):201-228, 1990.
4. D. Chan. Constructive Negation Based on the Complete Database *Proc. Int. Conference on Logic Programming'89, The MIT Press*, 111-125, 1988.
5. D. Chan. An Extension of Constructive Negation and its Application in Coroutining. *Proc. NACLP'89, The MIT Press*, 477-493. 1989.
6. H. Comon, P. Lescanne. Equational Problems and Disunification. *J. of Symbolic Computation*, 7:371-425, 1989.
7. W. Drabent. What is Failure? An Approach to Constructive Negation. *Acta Informatica 32, 27-59, Springer Verlag*, 1995.
8. M. Fernández. Narrowing Based Procedures for Equational Disunification. *Applicable Algebras in Eng. Communications and Computing*, 3:1-26, 1992.
9. M. Fitting, M. Ben-Jacob. Stratified and Three-valued Logic Programming Semantics *Proc. Int. Conf. and Symp. on Logic Programming*, 1988, pp. 1054-1069.
10. L. Fribourg. SLOG: A Logic Programming Language Interpreter based on Clausal Superposition and Rewriting. *Proc. Symp. on Logic Programming, IEEE Comp. Soc. Press*, 1985.
11. M. Hanus. Compiling Logic Programs with Equality. *Proc. PLILP'90, Springer LNCS*, 1990.
12. M. Hanus. The Integration of Functions into Logic Programs: A Survey. *J. of Logic Programming*, 1994.
13. M. Hanus. Combining Lazy Narrowing and Simplification. *PLILP'94, Springer LNCS 844*, 370-384, 1994.
14. P. Hill, J. lloyd. The Gödel Programming Language. *The MIT Press*, 1994.
15. J. Jaffar, J.L. Lassez. Constraint Logic Programming. *Proc. 14th ACM Symp. on Princ. of Prog. Lang.*, 114-119, 1987.
16. H. Kuchen, R. Loogen, J. J. Moreno-Navarro, M. Rodríguez-Artalejo. Graph-based Implementation of a Functional Logic Language *Proc. ESOP, Springer LNCS* 432, 271-290, 1990.
17. H. Kuchen, F. López-Fraguas, J.J. Moreno-Navarro, M. Rodríguez-Artalejo. Implementing a Lazy Functional Logic Language with Disequality Constraints. *Joint International Conference and Symposium on Logic Programming, The MIT Press*, 189-223, 1992.
18. K. Kunen. Negation in Logic Programming. *J. of Logic Programming* 4:289-308, 1987.
19. F.J. López-Fraguas. A General Scheme for Constraint Functional Logic Programming. *Proc. ALP'92, Springer LNCS*, 1992.
20. M. Maher. Complete Axiomatization of the Algebras of Finite, Rational and Infinite Trees. *Proc. 3rd Symp. on Logic in Computer Science*, 348-357, 1988.
21. J.J. Moreno-Navarro, H.Kuchen, R.Loogen, M. Rodríguez-Artalejo. Lazy Narrowing in a Graph Machine. *Proc. ALP'90, Springer LNCS* 463, 298-317, 1990.
22. J.J.Moreno-Navarro, H. Kuchen, J. Mariño-Carballo, S. Winkler, W. Hans. Efficient Lazy Narrowing using Demandedness Analysis. *Proc. PLILP'93, Springer LNCS* 714, 167-183, 1993.

23. J.J. Moreno Navarro, M. Rodríguez Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *J. of Logic Programming* 12:189-223, 1992.
24. J.J. Moreno Navarro. Default Rules: An Extension of Constructive Negation for Narrowing-based Languages. *Proc. ICLP'94, The MIT Press*, 1994.
25. L. Naish. Negation and Quantifiers in NU-Prolog. *ICLP'86, Springer LNCS* 225, 624-634, 1986.
26. M.J. Ramírez, M. Falaschi. Conditional Narrowing with Constructive Negation. *Proc. ELP'92, Springer LNCS* 660, 59-79, 1993.
27. P. Stuckey. Constructive Negation for Constraint Logic Programming *Proc. IEEE Symp. on Logic in Computer Science, IEEE Comp. Soc. Press*, 1991.