Implementing Constructive Failure in Functional-Logic Programming *

Jaime Sánchez-Hernández

Dept.de Sistemas Informáticos y Programación

Univ. Complutense de Madrid. Fac. Informática, 28040 Madrid, Spain

email: jaime@sip.ucm.es   Phone: +34 91 3947637

Abstract

Functional-logic programming amalgamates some of the main features of both functional and logic styles into a single paradigm. Nevertheless, negation is a widely investigated feature in logic programming that has not received much attention in such programming style. It is not difficult to incorporate some kind of *negation as finite failure* for ground goals, but we are interested in a *constructive* version able to deal with non-ground goals. With this aim we have built a formal framework for checking *(finite) failure of reduction* in previous works. In this paper we adapt it for implementing a prototype for a functional-logic language with *constructive failure* as the natural counterpart to negation in logic programming.

1 Introduction

Functional-logic programming (*FLP*, for short) tries to incorporate the most relevant features of both functional and logic programming styles (*FP* and *LP*, for short) such as lazy evaluation and non-deterministic computations (see [9] for a survey). Negation is an important feature that has been widely studied from the beginning of *LP* [4], but it has not received much attention in *FLP* with the exception of [19], that does not consider non-deterministic functions, an essential feature of modern *FLP* languages. Curry [3, 10] and the latest version of $\mathcal{TOY}$ [12] incorporate *finite failure* as a direct counterpart of *negation as failure* used in Prolog. Such kind of failure is easy to implement but, as in the case of Prolog, it is not suitable for goals with free variables, i.e., it is not *constructive*, using a standard terminology of *LP*. Constructive negation has also been studied in *LP* [6, 7] and implemented [20, 1]. In the *FLP* setting, constructive failure is a difficult issue due to the combination of laziness, sharing and non-determinism. The problem of binding variables while evaluating functions which collect values of a search space (an issue somehow related to our constructive failure) has been addressed in [5]. But this work focuses only on operational questions (sometimes system dependant), while our approach also provides a logical semantics that allows to prove correctness and completeness results.

We address the problem from a general point of view: the natural counterpart of logic negation, in *FLP* is *failure in reduction.* Such a notion can be expressed by means of a function *fails* with a clear meaning:

$$fails(e) = \begin{cases} true & \text{if } e \text{ can not be reduced to a head normal form} \\ false & \text{otherwise} \end{cases}$$

FLP follows a constructor discipline, and therefore by head normal form we mean a variable or a constructor-rooted term. The important fact is that *fails* can not be defined within the language, but it requires a theoretical framework in order to provide a formal semantics for it. We have investigated

*Work partially supported by the Spanish CICYT (project TIC 2002-01167 'MELODIAS')

this construction in previos papers, taking the well established framework *CRWL* [8, 21] as starting point. Following it, this approach, we have developed an specific framework for dealing with failure: the rewriting logic is introduced in [13, 18] and transformed into a set oriented logic in [14]; the narrowing relation is presented in [15] and extended with built-in equality in [16, 17]. In this paper, we incorporate a redex-selection mechanism to the narrowing relation and implement the prototype OOPS based on such a relation.

OOPS is not intended to replace $\mathcal{TOY}$ or Curry, but to motivate the introduction of *constructive failure* in such systems. It is designed as a research tool for studying failure in *FLP* and incorporates an interesting tracing tool that generates a LaTeX file with the detailed computation steps (formally justified by the narrowing relation *DDSNarr* that we will show). This prototype has been very useful for developing the theoretical narrowing mechanism and for exploring the potential that failure can offer to *FLP* by means of simple but not trivial running examples. Moreover, apart from failure, the set-oriented view used in the prototype is itself an interesting and clean way for a better understanding of relevant features of *FLP* such as non-determinism and sharing.

The paper is organized as follows: in Section 2 we present an example motivating the use of failure in *FLP*, Section 3 introduces the set-oriented view. Section 4 briefly presents rewriting logic *SRLF* and the narrowing relation *SNarr*. The main contributions of this work are Sections 5 and 6. In the first we modify the relation *SNarr* to get a demand driven mechanism for selecting redexes and show the correctness and completeness results for it. In the next Section we point out some implementation details of the system OOPS (available at `http://babel.dacya.ucm.es/jaime/systems.html`) based on the new relation. Finally, Section 7 contains some conclusions.

2 Using Failure in *FLP*

As a counterpart of negation in *LP*, failure in *FLP* allows to use negative information within programs in many situations. Moreover, there are problems for which to use this kind of information is the natural way for solving them.

As an example, consider the problem of searching paths in a graph. We assume the nodes a, b, c and d, the non-deterministic function *next* to define arcs and the function *path* for deciding if there is a path between two given nodes:

$$\begin{array}{ll} next(a) \to b & next(b) \to c \\ next(a) \to c & next(b) \to d \\ \multicolumn{2}{l}{path(X,Y) \to if\ (X == Y)\ then\ true} \\ \multicolumn{2}{l}{\qquad\qquad\qquad\ \ else\ path(next(X),Y)} \end{array}$$

Now we use failure to define the function *safe*, understanding that a node is safe if there is not a path from it to the node d:

$$safe(X) = fails(path(X,d))$$

This simple function can not be programmed in *FLP* without failure (of course it would be possible by changing the full program, in particular the definition of *next*). We can evaluate $safe(a)$ to *false* and $safe(c)$ to *true*.

This example could be programmed in Curry or $\mathcal{TOY}$, but they can not reduce a non-ground expression like $safe(X)$ for which OOPS gets *true* with the substitution $[X/c]$, and *false* with $[X/a]$, $[X/b]$ and $[X/d]$.

3 A Set-Oriented View of *FLP*

In the following we will write *CS* (*FS*) for the set of constructor (function) symbols of the program. We assume a countable set of variables $\mathcal{V} = \{X, Y, Z, \ldots\}$. *Exp* is the set of expressions built over $CS \cup FS \cup \mathcal{V}$ and *Term* is the set of terms built over $CS \cup \mathcal{V}$. Any object of the form $\overline{o}$ denotes a sequence $o_1, \ldots, o_n$.

In order to illustrate the set-oriented syntax, in this section we assume a program with $CS = \{z, s\}$ (for natural numbers), and $FS = \{add, double, two, coin\}$ defined as:

$$\begin{array}{ll} \multicolumn{2}{l}{add(z,X) \to X} \\ \multicolumn{2}{l}{add(s(X),Y) \to s(add(X,Y))} \\ double(X) \to add(X,X) & coin \to z \\ two(s(s(z))) \to s(s(z)) & coin \to s(z) \end{array}$$

This program could be a functional one except for the non-deterministic function *coin*. Such kind of functions are one of the nicest features in *FLP* that allow to express search problems in a direct way. But they also make more difficult to deal with failure. The problem arises from the fact that proving failure in the reduction (to head normal form) of an expression in a non-deterministic context means to prove that *any* possible reduction to head normal form fails. For example, the expression $two(coin)$ fails: *coin* can be reduced to z or $s(z)$, and *two* is not defined for any of these values. But none of the possible values for *coin* in isolation produces the failure; we must consider both reductions of *coin* simultaneously, i.e., to check that *two* fails for every value of the set $\{z, s(z)\}$. This idea suggests to use a set oriented semantics for collecting reductions of expressions. In fact, taking failure apart, non-deterministic functions themselves induce this kind of semantics.

In [18, 13] we formalize this idea with the introduction of statements of the form $coin \lhd \{z, s(z)\}$, where $\{z, s(z)\}$ is what we call a *S*ufficient *A*pproximation *S*et (*SAS*) for *coin* (Section 4 formalizes the notion of *SAS*). In [15] the set flavor was extended to expressions and programs. For instance, the expression $double(add(s(z), coin))$ is transformed into the *set-expression*:

$$\bigcup_{\alpha \in \bigcup_{\beta \in coin} add(s(z),\beta)} double(\alpha)$$

Formally, a set-expression $\mathcal{S} \in SetExp$ is defined as:

$$\mathcal{S} ::= \{t\} \mid f(\overline{t}) \mid t == t' \mid \bigcup_{\alpha \in \mathcal{S}_1} \mathcal{S}_2 \mid \mathcal{S}_1 \cup \mathcal{S}_2$$

where $t, t' \in Term$, $\overline{t} \in Term \times \overset{n}{\ldots} \times Term$, $f \in FS^n$ and $\mathcal{S}_1, \mathcal{S}_2 \in SetExp$. Indexed variables like α are taken from a distinguished set Γ and are called *produced variables* of the set-expression. The set of produced variables is notated as $PV(\mathcal{S})$ and the rest are *free variables*, notated as $FV(\mathcal{S})$. We consider the set *Subst* of substitutions for the free variables.

The notation used for set-expressions is clearly inspired in the standard mathematical one, and the formal semantics matches the intuitive meaning. But the relevant aspect is that it makes explicit *sharing* (by means of indexed variables like α) and non-deterministic computations (by means of unions), and it facilitates to build the operational mechanism.

It is easy to transform any usual expression e into its corresponding set-expression $\widehat{e}$, using (fresh) produced variables $\alpha_1, \ldots, \alpha_n$:

- $\widehat{c(\overline{e})} = \bigcup_{\alpha_1 \in \widehat{e_1}} \cdots \bigcup_{\alpha_n \in \widehat{e_n}} \{c(\overline{\alpha})\}$, $\forall c \in CS^n$
- $\widehat{f(\overline{e})} = \bigcup_{\alpha_1 \in \widehat{e_1}} \cdots \bigcup_{\alpha_n \in \widehat{e_n}} f(\overline{\alpha})$, $\forall f \in FS^n$
- $\widehat{X} = \{X\}$, $\forall X \in \mathcal{V}$

The transformation also introduces a new constant symbol F to explicitly denote failure of reduction. Programs are transformed into *set-programs* in such a way that function rules have a set-expression in the body. But the transformation is deeper, obtaining the following **unicity property**: for any function call $f(\overline{t})$ where $\overline{t}$ are ground terms (without variables) with no occurrence of the constant F there exists exactly one applicable rule in the program; moreover, all the heads of the rules of a function demand exactly the same positions (this is useful for the narrowing relation).

Here we use the standard notions of *positions* and *demanded positions* in the heads. For example, the head $f(s(s(z)), X, s(Y))$ has s at positions 1, 1.1 and 3, z at position 1.1.1, X at position 2, and Y at position 3.1; and this head demands the positions 1, 1.1 and 3, i.e., those that contain constructor symbols.

To achieve this kind of (inductively sequential) rules, our algorithm performs a demand analysis on the head of the rules (following the ideas of definitional trees of [2, 11]) and also completes the program introducing specific failure rules for those cases not defined in the original program. The concrete algorithm and the correctness results can be found in [22]. As an example of transformation, for the program of graphs, we obtain the following set-program:

$$\begin{array}{ll} next(a) \twoheadrightarrow \{b\} \cup \{c\} & next(c) \twoheadrightarrow \{\mathsf{F}\} \\ next(b) \twoheadrightarrow \{c\} \cup \{d\} & next(d) \twoheadrightarrow \{\mathsf{F}\} \\ safe(X) \twoheadrightarrow fails(path(X,d)) & \\ path(X,Y) \twoheadrightarrow & \\ \quad \bigcup_{\alpha \in X==Y} \bigcup_{\beta \in \bigcup_{\gamma \in next(X)} path(\gamma,Y)} iTe(\alpha, true, \beta) & \end{array}$$

(1) $\dfrac{}{\mathcal{S} \lhd \{\bot\}}$ **(2)** $\dfrac{}{\{X\} \lhd \{X\}}$ $X \in \mathcal{V}$

(3) $\dfrac{\{t_1\} \lhd \mathcal{C}_1 \ldots \{t_n\} \lhd \mathcal{C}_n}{\{c(t_1,\ldots,t_n)\} \lhd \{c(\overline{t}') \mid \overline{t}' \in \mathcal{C}_1 \times \ldots \times \mathcal{C}_n\}}$ $c \in CS \cup \{\mathsf{F}\}$

(4) $\dfrac{\mathcal{S}\theta \lhd \mathcal{C}}{f(\overline{t})\theta \lhd \mathcal{C}}$ if $\mathcal{C} \neq \{\bot\}$, $(f(\overline{t}) \twoheadrightarrow \mathcal{S}) \in \mathcal{P}$ and $\theta \in Subst_{\bot,\mathsf{F}}$

(5) $\dfrac{}{t == t' \lhd \{true\}}$ if $t \downarrow t'$ **(6)** $\dfrac{}{t == t' \lhd \{false\}}$ if $t \uparrow t'$

(7) $\dfrac{\mathcal{S}_1 \lhd \mathcal{C}_1 \quad \mathcal{S}_2[\alpha/\mathcal{C}_1] \lhd \mathcal{C}}{\bigcup_{\alpha \in \mathcal{S}_1} \mathcal{S}_2 \lhd \mathcal{C}}$ **(8)** $\dfrac{\mathcal{S}_1 \lhd \mathcal{C}_1 \quad \mathcal{S}_2 \lhd \mathcal{C}_2}{\mathcal{S}_1 \cup \mathcal{S}_2 \lhd \mathcal{C}_1 \cup \mathcal{C}_2}$

(9) $\dfrac{}{f(\overline{t}) \lhd \{\mathsf{F}\}}$ for all $(f(\overline{s}) \twoheadrightarrow \mathcal{S}') \in \mathcal{P}$, $\overline{t}$ and $\overline{s}$ have a $CS \cup \{\mathsf{F}\}$-conflict

(10) $\dfrac{}{t == t' \lhd \{\mathsf{F}\}}$ if $t \not\downarrow t'$ y $t \not\uparrow t'$

(11) $\dfrac{\mathcal{S} \lhd \{\mathsf{F}\}}{fails(\mathcal{S}) \lhd \{true\}}$

(12) $\dfrac{\mathcal{S} \lhd \mathcal{C}}{fails(\mathcal{S}) \lhd \{false\}}$ if $\exists t \in \mathcal{C} - \{\bot, \mathsf{F}\}$

Table 1: Rewriting logic *SRLF*

Cntx $C\ [\mathcal{S}]\Box\delta \rightsquigarrow_{\theta} C\theta\ [\mathcal{S}']\Box\delta'$ if $\mathcal{S}\Box\delta \rightsquigarrow_{\theta} \mathcal{S}'\Box\delta'$

Nw$_1$ $f(\overline{t})\Box\delta \ {}_{\theta}|_{var(\overline{t})}\!\!\rightsquigarrow \mathcal{S}\theta\Box\delta'$
if $(f(\overline{s}) \twoheadrightarrow \mathcal{S}) \in \mathcal{P}$, $\theta \in Sust_{\mathsf{F}}$ is a m.g.u. for $\overline{s}, \overline{t}$ with $Dom(\theta) \cap \Gamma = \emptyset$ and $\delta' \in solve(\delta\theta)$

Nw$_2$ $f(\overline{t})\Box\delta \rightsquigarrow_{\epsilon} \{\mathsf{F}\}\Box\delta$
if for every rule $(f(\overline{s}) \twoheadrightarrow \mathcal{S}) \in \mathcal{P}$ $\overline{s}$ and $\overline{t}$ have a $CS \cup \{\mathsf{F}\}$-conflict

Eq $t == s\Box\delta \ {}_{\theta}|_{var(t) \cup var(s)}\!\!\rightsquigarrow \{\omega\}\Box\delta'$
if $t == s \rightarrowtail_{\theta} \omega|_{\delta''}$ and $\delta' \in solve(\delta\theta \cup \delta'')$

Fail$_1$ $fails(\mathcal{S})\Box\delta \rightsquigarrow_{\epsilon} \{true\}\Box\delta$ if $\mathcal{S}^* = \{\mathsf{F}\}$

Fail$_2$ $fails(\mathcal{S})\Box\delta \rightsquigarrow_{\epsilon} \{false\}\Box\delta$ if $\exists t \in \mathcal{S}^* - \{\bot, \mathsf{F}\}$

Flat $\bigcup_{\alpha \in \bigcup_{\beta \in \mathcal{S}_1} \mathcal{S}_2} \mathcal{S}_3\Box\delta \rightsquigarrow_{\epsilon} \bigcup_{\beta \in \mathcal{S}_1} \bigcup_{\alpha \in \mathcal{S}_2} \mathcal{S}_3\Box\delta$

Dist $\bigcup_{\alpha \in \mathcal{S}_1 \cup \mathcal{S}_2} \mathcal{S}_3\Box\delta \rightsquigarrow_{\epsilon} \bigcup_{\alpha \in \mathcal{S}_1} \mathcal{S}_3 \cup \bigcup_{\alpha \in \mathcal{S}_2} \mathcal{S}_3\Box\delta$

Bind $\bigcup_{\alpha \in \{t\}} \mathcal{S}\Box\delta \rightsquigarrow_{\epsilon} \mathcal{S}[\alpha/t]\Box\delta$

Elim $\bigcup_{\alpha \in \mathcal{S}'} \mathcal{S}\Box\delta \rightsquigarrow_{\epsilon} \mathcal{S}\Box\delta$ if $\alpha \notin FV(\mathcal{S})$

Table 2: Rules for *SNarr*

Now, *next* collects all the possible reductions for each argument in a single rule and it is completed with failures for the cases c and d. The equality function $==$ (used in *path*) is a three-valued function that can produce *true*, *false* or F (*iTe* stands for *if _ then _ else*).

4 Rewriting Logic and Narrowing Calculus

The rewriting logic *SRLF* of Table 1 provides the semantics for any set-expression $\mathcal{S}$ with respect to a set-program $\mathcal{P}$, i.e., it proves statements of the form $\mathcal{S} \lhd \mathcal{C}$, where $\mathcal{C}$ is a *SAS* for $\mathcal{S}$. Here we only show a brief explanation of the rules (for a detailed discussion see [14]). Rule (1) provides the trivial (totally undefined) *SAS* for any set-expression allowing lazy derivations. Rule (3) stands for term decomposition. Rule (4) uses an instance of a rule of the program for evaluating a function call; such instance is obtained by means of $\theta \in Subst_{\bot,\mathsf{F}}$, a substitution that includes the undefined value $\bot$ and F in its range. Rule (9) detects a failure in parameter passing. Rules (5), (6) and (10) defines the function $==$ be means of the syntactic relations $\downarrow$, $\uparrow$, $\not\downarrow$ and $\not\uparrow$ that operate on constructed terms (without function symbols). Rules (7) and (8) are inspired in usual set manipulations and rules (11) and (12) define the function *fails* (in rule (12) $t \in \mathcal{C} - \{\bot, \mathsf{F}\}$ stands for a head normal form).

With the program of Section 2, this logic can prove $safe(a) \lhd \{false\}$ or $safe(c) \lhd \{true\}$. But it has nothing to do with $safe(X)$, as it is designed as a *rewriting* logic and not as a narrowing relation able to bind variables of expressions.

Then, as operational mechanism we define the narrowing relation *SNarr*. An step for this relation has the form:

$$\mathcal{S}\Box\delta \rightsquigarrow_{\theta} \mathcal{S}'\Box\delta'$$

where $\mathcal{S}, \mathcal{S}'$ are set-expressions, θ is the *answer substitution*, and δ, δ' are sets of disequalities in solved form, i.e., disequalities for variables (that may be introduced by reducing the function $==$). Table 2 shows the rules for *SNarr*. There are quite a lot technical subtlety in these rules, but here we only point out the most relevant aspects (see [15, 16, 17] for details):

- in the rule **Cntx**, C denotes a context as usual in *FP* and it allows to select any

sub-set-expression as redex in order to apply some other rule;

- **Nw**$_1$ performs narrowing in a proper sense, unifying the arguments of the call with those of a rule of the program. The condition $Dom(\theta) \cap \Gamma = \emptyset$ ensures that the produced variables are not affected by the substitution and the function *solve* makes the propagation of bindings to the disequality store;

- **Nw**$_2$ checks a failure in reduction. Although the set-program satisfies the unicity property, if a call contains F at some demanded position there is not any applicable rule of the program;

- **Eq** evaluates a call to the function $==$ using the relation $\rightarrowtail$, that requires a non trivial narrowing mechanism [16, 17]. It calculates the appropriate substitution θ and produces *true* if the terms unify, *false* if there is a conflict of constructors (or a disequality is introduced in δ), and F otherwise (this case holds for example in $\text{F} == X$);

- **Fail**$_1$ and **Fail**$_2$ evaluate $fails(\mathcal{S})$ by analyzing the *information set* $\mathcal{S}^*$ of $\mathcal{S}$, that reflects its constructed part. Formally $\mathcal{S}^*$ is defined as: $(\{t\})^* = \{t\}$; $(\mathcal{S}_1 \cup \mathcal{S}_2)^* = \mathcal{S}_1^* \cup \mathcal{S}_2^*$; $(f(\overline{t}))^* = (fails(\mathcal{S}))^* = (t == s) = \{\perp\}$; $(\bigcup_{\alpha \in \mathcal{S}'} \mathcal{S})^* = (\mathcal{S}[\alpha/\perp])^*$

- finally, rules **Flat**, **Dist**, **Bind** and **Elim** have a clear mathematical sense.

The aim of this relation is to narrow any set-expression to a *normal form*, i.e., a set-expression of the form $\{t_1\} \cup \ldots \cup \{t_n\}$ (that represents the collection of terms $\{t_1, \ldots, t_n\}$). As usual we consider the transitive closure $\mathcal{S}\Box\delta \overset{*}{\underset{\theta}{\leadsto}} \mathcal{S}'\Box\delta'$: $\mathcal{S}\Box\delta \underset{\theta_1}{\leadsto} \mathcal{S}_1\Box\delta_1 \underset{\theta_2}{\leadsto} \ldots \underset{\theta_n}{\leadsto} \mathcal{S}'\Box\delta'$ where $\theta = \theta_1\theta_2\ldots\theta_n$ is the answer substitution of the derivation.

Correctness and completeness results of the relation *SNarr* with respect to *SRLF* have been established in [15, 16, 17].

5 A Demand Driven Narrowing Relation: *DDSNarr*

The rule **Cntx** of *SNarr* allows to select any possible sub-set-expression as redex. In this Section we will show a modified version of this rule that works only on the demanded sub-set-expressions following the philosophy of lazy functional languages. First we introduce the concept of demand in set-expressions.

Definition 1 (Demanded Variables) *Given a set-expression $\mathcal{S}$, the set $DV(\mathcal{S}) \subseteq \Gamma$ of its demanded variables is defined as:*

- $DV(\{t\}) = var(t) \cap \Gamma$
- $DV(f(\overline{t})) = DVFun(f(\overline{t})) \cap \Gamma$, *where* $DVFun(f(\overline{t})) = \{X \in var(\overline{t}) \mid X$ *appears in a demanded position by* $f\}$
- $DV(t == s) = var(\overline{t}) \cup var(\overline{s})$
- $DV(fails(\mathcal{S})) = \begin{cases} \emptyset & \text{if } \mathcal{S}^* = \{\text{F}\} \text{ or } \exists t \in (\mathcal{S}^* - \{\perp, \text{F}\}) \\ DV(\mathcal{S}) & \text{otherwise} \end{cases}$
- $DV(\bigcup_{\alpha \in \mathcal{S}_1} \mathcal{S}_2) = \begin{cases} DV(\mathcal{S}_1) \cup DV(\mathcal{S}_2) & \text{if } \alpha \in DV(\mathcal{S}_2) \\ DV(\mathcal{S}_2) & \text{otherwise} \end{cases}$
- $DV(\mathcal{S}_1 \cup \mathcal{S}_2) = DV(\mathcal{S}_1) \cup DV(\mathcal{S}_2)$

Using this notion we can now define an special kind of contexts in such a way that the set-expression of the argument is needed for reduction, i.e., it is demanded. A **context of demand** is defined as:

$$C ::= [\,] \mid fails(C') \mid C' \cup \mathcal{S} \mid \mathcal{S} \cup C' \mid \bigcup_{\alpha \in \mathcal{S}} C' \mid \bigcup_{\alpha \in C'} \mathcal{S}$$

where C' is a context of demand, $\mathcal{S}$ is a set-expression and $\alpha \in DV(\mathcal{S})$ in the last case. We can now restrict the application of rule **Cntx** in the following way:

DDCntx	$C\ [\mathcal{S}]\Box\delta \underset{\theta}{\leadsto} C\theta\ [\mathcal{S}']\Box\delta'$ if C is a context of demand and $\mathcal{S}\Box\delta \underset{\theta}{\leadsto} \mathcal{S}'\Box\delta'$

For example, if we consider the expression $add(add(coin, X), add(Y, Z))$ and its corresponding set-expression:

$$\underline{\bigcup_{\alpha \in \underline{\bigcup_{\beta \in \underline{coin}} \underline{add(\beta, X)}}} \underline{\bigcup_{\gamma \in add(Y,Z)} \underline{add(\alpha, \gamma)}}}$$

the demanded variables are α and β, and the contexts of demand are those that have as arguments the underlined set-expressions like $add(\beta, X)$ or *coin* (notice that $add(Y, Z)$ does not correspond to a context of demand).

The relation *DDSNarr* is the result of replacing the rule **Cntx** by the new rule **DDCntx** in *SNarr*. The correctness of *DDSNarr* arises from the correctness of *SNarr*:

Theorem 1 (Correctness of *DDSNarr*) *Let $\mathcal{S}, \mathcal{S}'$ be set-expressions, θ a substitution and δ and δ' sets of disequalities in solved form. If $\mathcal{S}\Box\delta \overset{*}{\underset{\theta}{\leadsto}} \mathcal{S}\Box\delta'$ is a DDSNarr-derivation then for any $\sigma \in Sol(\delta)$ and $\sigma' \in Sol(\delta')$ we have: $\mathcal{S}\sigma \lhd \mathcal{C} \Leftrightarrow \mathcal{S}'\sigma' \lhd \mathcal{C}$.*

Proof: trivial: any *DDSNarr*-derivation is a *SNarr*-derivation, for which the result holds.□

The completeness result requires rather more elaboration. The proof is an extension of the proof for the completeness result for *SNarr* [15, 16, 17], but needs to show that each derivation step with *DDSNarr* makes the set-expression to evolve to a normal form (if such form exists for the set-expression). We define the *complexity* of a *SRLF*-proof $\mathcal{S} \lhd \mathcal{C}$ as the number of *SRLF*-steps of such a proof. Given a set-expression $\mathcal{S}$ with $\mathcal{S} \lhd \mathcal{C}$, we will show that we can derive $\mathcal{S}'$ by means of *DDSNarr* with $\mathcal{S}' \lhd \mathcal{C}$ (this is only correctness), but in such a way that the complexity of the proof $\mathcal{S}' \lhd \mathcal{C}$ decreases (progress of *DDSNarr*). The formal results that follow must also consider the possible disequalities δ associated to the set-expressions.

The first result shows that for a set-expression (not in normal form) *DDSNarr* is able to apply some rule that reduces complexity, except for the rules **Flat** and **Dist**, that preserve this complexity.

Proposition 1 (Partial Progress) *Let $\mathcal{S}$ be a set-expression not in normal form such that $\mathcal{S} \lhd \mathcal{C}$ with $\mathcal{C} \neq \{\perp\}$. Then it is possible to perform a derivation step $\mathcal{S}\Box\emptyset \underset{\epsilon}{\leadsto} \mathcal{S}'\Box\emptyset$ with DDSNarr such that $\mathcal{S}' \lhd \mathcal{C}$ and:*

- *if the step is performed by* **Flat** *or* **Dist** *applied to $\mathcal{S}$ or to some sub-set-expression of $\mathcal{S}$ by means of* **DDCntx***, then the complexity of the proofs $\mathcal{S} \lhd \mathcal{C}$ and $\mathcal{S}' \lhd \mathcal{C}$ are the same;*
- *otherwise the complexity of the proof for $\mathcal{S}' \lhd \mathcal{C}$ is less than the one for $\mathcal{S} \lhd \mathcal{C}$.*

Proof sketch: we proceed by induction on the structure of $\mathcal{S}$ (not in normal form) and taking into account that $\mathcal{C} \neq \{\perp\}$, pointing out the way in which complexity of the *SRLF*-proofs is affected:

- if $\mathcal{S} = f(\bar{t})$ and there is a rule of the program for reducing it, then we can apply $\mathbf{Nw}_1$ in such a way that $\mathcal{S}'$ is the body of such rule and the complexity of the *SFRL*-proof decreases (*SRLF* contains a rule that replaces $f(\bar{t})$ by the body of the corresponding program rule). Otherwise, if no rule of the program matches $f(\bar{t})$ then we can apply $\mathbf{Nw}_2$ to obtain $\{\mathsf{F}\}$;
- if $\mathcal{S} = t == s$ it is possible to use **Eq** and finish the derivation. Notice that it does not need to introduce any disequality because we have $\mathcal{S} \lhd \mathcal{C}$ (without any restriction to the variables of $\mathcal{S}$);
- if $\mathcal{S} = fails(\mathcal{S}_1)$ then if it is possible to apply $\mathbf{Fail}_1$ or $\mathbf{Fail}_2$ the derivation finish. Otherwise, by induction hypothesis some rule will be applicable to $\mathcal{S}_1$ by means of **DDCntx** reducing the complexity of the *SRLF*-proof;
- if $\mathcal{S} = \bigcup_{\alpha \in \mathcal{S}_1} \mathcal{S}_2$ there are three possibilities:
  - if $\alpha \notin FV(\mathcal{S}_2)$ then **Elim** applies;
  - if $\alpha \in FV(\mathcal{S}_2) - DV(\mathcal{S}_2)$, as α is not demanded in $\mathcal{S}_2$ it does not block any reduction in $\mathcal{S}_2$ and i.h. applies to $\mathcal{S}_2$ by means of **DDCntx**;
  - otherwise $\alpha \in FV(\mathcal{S}_2) \cap DV(\mathcal{S}_2)$ and the rule depends on the form of $\mathcal{S}_1$: if $\mathcal{S}_1 = \{t\}$ then **Bind** applies; if $\mathcal{S}_1 = \bigcup_{\beta \in \mathcal{S}_3} \mathcal{S}_4$ or $\mathcal{S}_1 = \mathcal{S}_3 \cup \mathcal{S}_4$ then **Flat** or **Dist** respectively can

be applied and the complexity of the *SRLF*-proofs do not change; otherwise i.h. applies to $\mathcal{S}_1$ by means of **DDCntx**;

- if $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ we can apply i.h. to both set-expression by means of **DDCntx**.$\Box$

Lemma 1 (Progress of *DDSNarr*) *Let $\mathcal{S} \in SetExp$ not in normal form such that $\mathcal{S} \triangleleft \mathcal{C}$ with $\mathcal{C} \neq \{\perp\}$. Then it is possible to perform a finite number of derivation steps $\mathcal{S}\Box\emptyset \overset{*}{\underset{\epsilon}{\leadsto}} \mathcal{S}'\Box\emptyset$ with DDSNarr such that there is a proof for $\mathcal{S}' \triangleleft \mathcal{C}$ with less complexity than the one for $\mathcal{S} \triangleleft \mathcal{C}$.*

Proof sketch: notice that rules **Flat** and **Dist** can only be applied a finite number of times over a set-expression; after that, the previous result ensures that some other rules will be applicable reducing the complexity of the proof.$\Box$

In order to obtain the result of completeness we need to prove that *DDSNarr* is able to find the appropriate values for variables. Moreover, the narrowing relation can provide a more general substitution that the one used for the *SRLF*-derivation: if $\mathcal{S}\theta \triangleleft \mathcal{C}$ then *DDSnarr* can obtain the same *information* with an answer substitution θ' more general than θ (i.e. $\theta = \theta'\mu$ for some μ, except for the new variables Π that can introduce the body of a rule program by means of $\mathbf{Narr}_1$). The actual relevance of this result is to obtain the Theorem 2 and we skip other technical subtles.

Lemma 2 (Answer Substitutions) *Given $\mathcal{S}\Box\delta$ and $\theta \in Sol(\delta)$, if DDSNarr allows to derive $\mathcal{S}\theta\Box\emptyset \overset{*}{\underset{\epsilon}{\leadsto}} \mathcal{S}'\Box\delta'$, then it also allows to derive $\mathcal{S}\Box\delta \overset{*}{\underset{\theta'}{\leadsto}} \mathcal{S}''\Box\delta''$ with new variables Π such that for some substitution μ we have:*

$$\theta = (\theta'\mu)|_{\mathcal{V}-\Pi};\ (\mathcal{S}')^* \subseteq (\mathcal{S}''\mu)^*;\ Sol(\delta') \subseteq Sol(\delta'\mu)$$

Proof sketch: this Lemma was proved for *SNarr* [16, 17] by imitating the steps of the derivation for $\mathcal{S}\theta\Box\emptyset \overset{*}{\underset{\epsilon}{\leadsto}} \mathcal{S}'\Box\delta'$ in the derivation for $\mathcal{S}\Box\delta \overset{*}{\underset{\theta'}{\leadsto}} \mathcal{S}''\Box\delta''$. This proof is also valid for the calculus *DDSNarr*.$\Box$

Now, the result of completeness is obtained by considering simultaneously the progress of *DDSNarr* and its ability for finding answer substitutions. Part *ii*) reflects the fact that *DDSNarr* is able to extract the values of the semantics of a set-expression. The final corollary has a more readable presentation, assuming termination and normal forms.

Theorem 2 (Completeness of *DDSNarr*) *Let $\mathcal{S}$ be a set-expression, δ a set of disequalities in solved form and $\theta \in Sol(\delta)$. If $\mathcal{S}\theta \triangleleft \mathcal{C}$ then there exists a DDSNarr-derivation $\mathcal{S}\Box\delta \overset{*}{\underset{\theta'}{\leadsto}} \mathcal{S}'\Box\delta'$, with new variables Π such that for some substitution μ we have: i) $\theta = (\theta'\mu)|_{\mathcal{V}-\Pi}$ ii) $\mathcal{S}'\mu \triangleleft \mathcal{C}$ iii) $\mu \in Sol(\delta')$*

Proof sketch: It follows from Lemmas 1,2.$\Box$

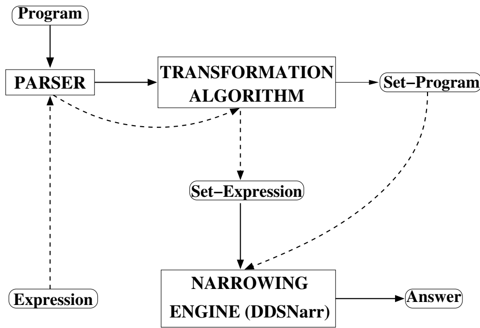
The next result clarifies the relevance of the completeness result: *DDSNarr* allows to obtain any *SAS* of the semantics of a set-expression as a normal form. Furthermore, this relation allows to generalize any substitution used for obtaining such *SAS*, possibly by introducing disequalities as a part of the answer (for example, $X == s(Y)$ will produce the answer $\{true\}$ with $[X/s(Y)]$, and $\{false\}$ with the disequality $X \neq s(Y)$ instead of infinite answers for *false* using only substitutions).

Corollary 1 (Completeness of *DDSNarr*) *Let $\mathcal{S} \in SetExp$ and $\theta \in Subst$ (with $Dom(\theta) \subseteq FV(\mathcal{S})$) be such that $\mathcal{S}\theta \triangleleft \{t_1, \ldots, t_n\}$. Then there exists a DDSNarr-derivation $\mathcal{S}\Box\emptyset \overset{*}{\underset{\theta'}{\leadsto}} \{t_1\} \cup \ldots \cup \{t_n\}\Box\delta$ such that for some substitution μ we have: $\theta = \theta'\mu,\ \mu \in Sol(\delta)$*

6 Implementation

OOPS, like Curry or $\mathcal{TOY}$ is implemented in Prolog (SWI-Prolog) and users of these systems must be comfortable using it (type `:h`

from the prompt of OOPS for a list of commands). Programs use the standard (currified functional) syntax and the interpreter is analogous to $\mathcal{TOY}$ or Curry. The architecture of the system is as follows:



The parser analyzes the source code of programs and produces a flat representation as Prolog facts, from which the transformation algorithm generates the corresponding set-program. This process is transparent for the user, that does not have to know about the set-view of the system. Expressions to be reduced are processed in a similar way (the transformation algorithm includes the conversion of expressions into set-expression) and they are throwed into the narrowing engine that implements the relation *DDSNarr*. Of course, this engine consults the set-program for evaluating functions (the predefined functions for equality and disequality are implemented as a subsidiary mechanism).

With respect to the Prolog representation, for set-expressions we use Prolog terms trying to facilitate the work of the reduction engine. For example, $\bigcup_{\alpha\in\mathcal{S}_1}\mathcal{S}_2$ is represented by a term of the form `in(pv(A),`$tp_{\mathcal{S}_1}$`,`$tp_{\mathcal{S}_2}$`)`, where $tp_{\mathcal{S}_1}$ and $tp_{\mathcal{S}_2}$ are the representations of $\mathcal{S}_1$ and $\mathcal{S}_2$ resp. Produced variables are clearly distinguished because they are represented as $pv(\ldots)$, while standard variables are self-represented. This is useful because of the different behavior of both types of variables in the narrowing relation. An equality $t == t'$ is represented as `eq([(t,t')])`. We use lists of pairs that will be eventually produced by decomposition of compound terms. The mechanism for solving equalities use this representation for improving efficiency avoiding multiple decomposition of the same terms.

Program rules of the transformed program are stored as Prolog facts of the form:

setRule(function_ name, list_ of_ arguments, result, demanded_ positions).

The Prolog term *result* represents a set-expression corresponding to the body of the rule, and *demanded_ positions* stores the positions demanded by the head of the rule (positions in the *list_ of_ arguments* that contain constructor symbols). This information about such positions is stored in order to improve the efficiency in the reduction engine.

6.1 The Reduction Engine of OOPS

The relation *DDSNarr* reduces the amount of non-determinism with respect to *SNarr* but is not completely deterministic because there may be more than one possible redex. For example, for the set-expression:

$$\underline{\underline{\bigcup_{\alpha\in\underline{\bigcup_{\beta\in\underline{coin}} \underline{add(\beta,X)}}} \bigcup_{\gamma\in add(Y,Z)} \underline{add(\alpha,\gamma)}}}$$

after some reduction steps (with $\mathbf{Nw}_1$, **Dist** and **Bind**) we obtain:

$$\bigcup_{\alpha\in\underline{add(z,X)}} \bigcup_{\gamma\in add(Y,Z)} add(\alpha,\gamma) \ \cup \ \bigcup_{\alpha\in\underline{add(s(z),X)}} \bigcup_{\gamma\in add(Y,Z)} add(\alpha,\gamma)$$

Now there are two possible redexes. OOPS performs a reduction over both of them. In general, if there are several redexes, the system will use all of them for reduction. This is a kind of width search that can improve the completeness properties of the system in some situations. Nevertheless OOPS allows to change this mode of operation to depth search (with the command `:nw`).

A reduction *step* in OOPS may be done by application of several rules (at least one) of the relation *DDSNarr*. Given a set-expression, OOPS analyzes recursively the syntactic structure trying to evaluate the innermost sub-set-expressions. For example, in the above set-expression it firstly analyzes the left sub-set-expression

Step	Expression	Rule
(1)	$\underline{safe(X)}\ \square\{\}$	(Nw_1)
(2)	$fails(\underline{path(X,d)})\ \square\{\}$	(Nw_1)
(3)	$fails(\bigcup_{\pi_1 \in \underline{==[(X,d)]}} \bigcup_{\pi_2 \in \ldots}$	(Eq)
(4)	$fails(\underline{\bigcup_{\pi_1 \in \{false\}} \bigcup_{\pi_2 \in \ldots}}$	(Bind)
(5)	$fails(\ldots \underline{iTe(false, true, \pi_2)})\ \square\{X \neq d\}$	(Nw_1)
	$\ldots$ $\boxed{X = c}$	
(15)	$fails(\underline{\bigcup_{\pi_2 \in \{\mathsf{F}\}} \{\pi_2\}})\ \square\{\}$	(Bind)
(16)	$\underline{fails(\{\mathsf{F}\})}\ \square\{\}$	(Fail_1)
(17)	$\{true\}\ \square\{\}$	
	Answer: $\{true\}\ \square\{\}$ (with $X = c$)	

Table 3: Derivation example for $safe(X)$

(*i*) $\bigcup_{\alpha \in add(z,X)} \bigcup_{\gamma \in add(Y,Z)} add(\alpha,\gamma)$. An step over it requires an step over the sub-set-expression (*ii*) $\bigcup_{\gamma \in add(Y,Z)} add(\alpha,\gamma)$, that requires an step over the innermost part (*iii*) $add(\alpha,\gamma)$. This call demands the produced variable α, so no rule can be applied, but α is stored as demanded variable. Then, at the back of recursion we explore the indexed sub-set-expressions. In (*ii*) the sub-set-expression is indexed by γ that is not demanded, so it remains identical. But in (*i*) we find α indexing the sub-set-expression $add(z,X)$ that can be reduced to $\{X\}$ by the first rule of *add*.

Analogously, for the right part the redex $add(s(z),X)$ is found and it is reduced to $\bigcup_{\pi \in add(z,X)} \{s(\pi)\}$, so the complete set-expression is reduced to:

$$\bigcup_{\alpha \in \{X\}} \bigcup_{\gamma \in add(Y,Z)} add(\alpha,\gamma)\ \cup$$
$$\bigcup_{\alpha \in \bigcup_{\pi \in add(z,X)} \{s(\pi)\}} \bigcup_{\gamma \in add(Y,Z)} add(\alpha,\gamma)$$

This way of proceeding corresponds to evaluate outermost sub-expressions in an standard expression, i.e., to lazy evaluation.

As an example of derivation Table 3 shows a part of one of the possible evaluations for the expression $safe(X)$ using the set-program of graphs (see Section 3). The (underlined) redexes are selected with the previous explained mechanism. Notice that in step (3) the equality $X == d$ is reduced to *false* by imposing the disequality $X \neq d$. In a further step the disequality disappears by binding $X = c$. This trace is automatically generated by OOPS in a LaTeX file and showed in postscript file by activating the debug mode (command `:d`).

7 Conclusions and Future Work

We have connected our previous theoretical research about constructive failure in *FLP* with an effective implementation of a functional-logic language providing such a resource. In order to understand the implementation we have sketched the set-oriented view from which we study the failure and the way for transforming the classic elements of *FLP* (expressions, programs) into this new formalism. The resulting framework allows to introduce failure and makes explicit some important aspects of *FLP* like non-determinism or sharing.

OOPS is closer enough to the formalism to allow to reason about the formalism itself, but also to explore the practical implications of integrating failure in *FLP*. We illustrate the last point with an example in which failure has a prominent role. As a tool for analyzing the narrowing mechanism, OOPS includes an interesting option for showing the transformed programs and for tracing computations, in such a way that every step of the trace is formally justified by the theoretical relation.

As a general aim, we have tried to motivate the use of constructive failure in *FLP* and its incorporation into Curry or $\mathcal{TOY}$. It is quite easy for non-constructive failure (for ground expressions), but requires additional effort for the constructive version presented here.

References

[1] J. Álvez, P. Lucio, F. Orejas, E. Pasarella, and E. Pino. Constructive negation by bottom-up computation of literal answers. In *Proc. SAC'04*, 2004, pages 1468–1475. ACM Press, 2004.

[2] S. Antoy. Definitional trees. In *Proc. ALP'92*, pages 143–157. Springer LNCS 632, 1992.

[3] S. Antoy and M. Hanus. Curry. a tutorial introduction. Available at `http://www.informatik.uni-kiel.de/~curry/tutorial/`, November 2002.

[4] K.R. Apt and R. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19&20:9–71, 1994.

[5] Braßel, B. and Hanus, M. and Huch, F. *Encapsulating Non-Determinism in Functional Logic Computations.* In Journal of Functional and Logic Programming, 2004(6), 2004.

[6] Chan, D. *Constructive negation based on the completed database.* In Proc. ICSLP'88, pages 111–125, 1988.

[7] W. Drabent. What is failure? An approach to constructive negation. *Acta Informatica, Springer*, 32(1):27–59, 1995.

[8] J.C. González-Moreno, T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *JLP*, 40(1):47–87, 1999.

[9] M. Hanus. The integration of functions into logic programming: A survey. *JLP*, 19-20:583–628, 1994. Special issue "Ten Years of Logic Programming".

[10] M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8). Available at `http://www.informatik.uni-kiel.de/curry/report.html`, 2003.

[11] R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. PLILP'93*, pages 184–200. Springer LNCS 714, 1993.

[12] F.J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. RTA'99*, pages 244–247. Springer LNCS 1631, 1999.

[13] F.J. López-Fraguas and J. Sánchez-Hernández. Proving failure in functional logic programs. In *Proc. CL'00*, pages 179–193. Springer LNAI 1861, 2000.

[14] F.J. López-Fraguas and J. Sánchez-Hernández. Functional logic programming with failure: A set-oriented view. In *Proc. LPAR'01*, pages 455–469. Springer LNAI 2250, 2001.

[15] F.J. López-Fraguas and J. Sánchez-Hernández. Narrowing failure in functional logic programming. In *Proc. FLOPS'02*, pages 212–227. Springer LNCS 2441, 2002.

[16] F.J. López-Fraguas and J. Sánchez-Hernández. Failure and equality in functional logic programming. *ENTCS*, 86(3), 2003.

[17] F.J. López-Fraguas and J. Sánchez-Hernández. Functional logic programming with failure and built-in equality. In *Proc. WFLP'03*, pages 61–74, 2003. Available at `http://www.dsic.upv.es/rdp03/wflp/proceedings.html`

[18] F.J. López-Fraguas and J. Sánchez-Hernández. A proof theoretic approach to failure in functional logic programming. *TPLP*, 4(1&2):41–74, 2004.

[19] J.J. Moreno-Navarro. Extending constructive negation for partial functions in lazy functional-logic languages. In *Proc. ELP'96*, pages 213–227. Springer LNAI 1050, 1996.

[20] J.J. Moreno-Navarro and S. Muñoz-Hernández. Implementation results in classical constructive negation. In *Proc. ICLP'94*, pages 284–298. Springer LNCS 3132, 2004.

[21] M. Rodríguez-Artalejo. Functional and constraint logic programming. In *Revised Lectures of CCL'99*, pages 202–270. Springer LNCS 2002, 2001.

[22] Sánchez-Hernández, J. Una aproximación al fallo en programación declarativa multiparadigma. PhD thesis, SIP-UCM, 2004.