Journal of Universal Computer Science, vol. 12, no. 11 (2006), 1574-1593
submitted: 1/5/06, accepted: 15/10/06, appeared: 28/11/06 © J.UCS

Constructive Failure in Functional-Logic Programming: From Theory to Implementation[1]

Jaime Sánchez-Hernández
(Universidad Complutense de Madrid, Spain
jaime@sip.ucm.es)

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, in previous works we have built a formal framework for checking *(finite) failure of reduction.* 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.

Key Words: negation, constructive failure, narrowing, functional-logic programming
Category: D.1.1 D.1.6 F.3.2

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 into a unified paradigm (see [Hanus 1994] for a survey). Negation is an important feature that has been widely studied from the beginning of *LP* [Apt et al. 1994], but it has not received much attention in *FLP* with the exception of [Moreno 1996], that does not consider non-deterministic functions, an essential feature of modern *FLP* languages. Curry [Antoy et al. 2002, Hanus 2003] and the latest version of $\mathcal{TOY}$ [López et al. 1999] 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* [Chan 1998, Drabent 1995] and implemented [Moreno et al. 2004, Álvez et al. 2004]. 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

[1] Work partially supported by the Spanish projects MERIT-FORMS (TIN2005-09027-C03-03) and PROMESAS-CAM (S-0505/TIC/0407)

addressed in [Braßel et al. 2004]. But this work focuses only on operational questions (sometimes system dependant), while our approach also provides a logical semantics that allows to prove soundness 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:

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

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 this construction in previous papers, taking the well established framework *CRWL* [González et al 1999, Rodríguez 2001] as starting point. Following this approach we have developed a specific framework for dealing with failure: the rewriting logic is introduced in [López et al. 2000, López et al. 2004] and transformed into a set oriented logic in [López et al. 2001]; the narrowing relation is presented in [López et al. 2002] and extended with built-in equality in [López et al. 2003a, López et al. 2003b]. 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 some examples 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 one we modify the relation *SNarr* to get a demand driven mechanism for selecting redexes and show the soundness 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*

Failure in *FLP* allows to use negative information within programs in many situations, as it happens with negacion in *LP*. Moreover, there are problems for which to use this kind of information is the natural way for solving them. In this section we show a couple of examples that use failure as a natural resource (the reader may try to solve these problems in *FLP* without using failure).

Example 1: 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:

$$\begin{array}{ll} \mathit{next}(a) \rightarrow b & \mathit{next}(b) \rightarrow c \\ \mathit{next}(a) \rightarrow c & \mathit{next}(b) \rightarrow d \end{array}$$

$$\begin{array}{l} \mathit{path}(X,Y) \rightarrow \mathit{if}\ (X == Y)\ \mathit{then}\ \mathit{true} \\ \qquad\qquad\qquad\quad \mathit{else}\ \mathit{path}(\mathit{next}(X),Y) \end{array}$$

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:

$$\mathit{safe}(X) \rightarrow \mathit{fails}(\mathit{path}(X,d))$$

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

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

Example 2: failure may be used for programming two-person finite games in an easy and elegant way [López et al. 2004] (following the idea of [Apt 2000] in *LP*). Assume such a two-person game in which players make a move in turn, until it is not possible to continue. At this stage, the player that cannot move loses, so the winner is the one that makes the last movement. We assume a function *move* such that *move(State)* produces (in a non-deterministic way) a new state from the given *State*, using a legal movement. A winning movement can be found using *fails* with a function like:

$$\begin{array}{l} \mathit{winMove}(S) \rightarrow \mathit{let}\ S' == \mathit{move}(S) \\ \qquad\qquad\qquad\quad \mathit{in}\ \mathit{if}\ \mathit{fails}(\mathit{winMove}(S'))\ \mathit{then}\ S' \end{array}$$

The idea is to find a legal movement, S', such that the other player fails to find a winning movement from it. This scheme can be used for the Nim's game for instance: we have a set of rows with sticks and a movement consists in dropping one or more sticks from a single row. We can represent a state by a list of natural numbers (each one representing the number of sticks of a row). A legal movement is defined in a non-deterministic way as (we assume natural numbers represented with z and $s(\ldots)$ as usual):

$$\begin{array}{ll} \mathit{pick}(s(N)) \rightarrow N & \mathit{move}([N|Ns]) \rightarrow [\mathit{pick}(N)|Ns] \\ \mathit{pick}(s(N)) \rightarrow \mathit{pick}(N) & \mathit{move}([N|Ns]) \rightarrow [N|\mathit{move}(Ns)] \end{array}$$

The function *pick* drops a positive number of sticks from a non-empty row and *move* chooses any row from the list for dropping sticks.

With this program we could evaluate $\mathit{winMove}([s(z), s(s(z)), s(s(z))])$ obtaining the answer: $[z, s(s(z)), s(s(z))]$. It is easy to check that for any possible movement of the other player we can win. For example, if the adversary gets the state $[z, s(z), s(s(z))]$, our next movement will be $[z, s(z), s(z)]$ and now, the adversary must take one of the sticks and we will take the remaining one, so we win the game.

As in the previous example, this game could be adapted to $\mathcal{TOY}$ or Curry with negation as finite failure and evaluate a winning movement for ground states as the previous one. But none of them will work with the state $[s(s(z)), X, s(z)]$, for which OOPS provides an infinite number of answers: $[s(z), z, s(z)]$ with the substitution $X = z$, $[z, s(z), s(z)]$ with $X = s(z)$,.... Notice that this reflects the fact that the function *fails* is an approach to *constructive failure* in *FLP*.

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 *total* expressions built over $CS \cup FS \cup \mathcal{V}$ and *Term* is the set of *total* terms built over $CS \cup \mathcal{V}$. The sets of *partial* terms and expressions are built in a similar way but they can include the special (constant) symbol $\perp$ that stands for the *undefined value.* Any object of the form $\overline{o}$ denotes a sequence $o_1, \ldots, o_n$.

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

$$\begin{array}{ll} \mathit{add}(z, X) \rightarrow X & \mathit{coin} \rightarrow z \\ \mathit{add}(s(X), Y) \rightarrow s(\mathit{add}(X, Y)) & \mathit{coin} \rightarrow s(z) \\ \mathit{double}(X) \rightarrow \mathit{add}(X, X) & \mathit{two}(s(s(z))) \rightarrow s(s(z)) \end{array}$$

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 [López et al. 2004, López et al. 2000] we formalize this idea with the introduction of statements of the form $coin \triangleleft \{z, s(z)\}$, 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 [López et al. 2002] 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:

$$\mathcal{S} ::= \{t\} \mid f(\overline{t}) \mid fails(S_1) \mid t == t' \mid \bigcup_{\alpha \in \mathcal{S}_1} \mathcal{S}_2 \mid \mathcal{S}_1 \cup \mathcal{S}_2$$

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 $\alpha_1, \ldots, \alpha_n$:

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

- $\widehat{fails(e)} = fails(\widehat{e})$

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)[2].

To achieve this kind of *inductively sequential* rules [Antoy 1992], our algorithm performs a demand analysis on the head of the rules (following the ideas of definitional trees of [Antoy 1992, Loogen et al. 1993]) 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 [Sánchez 2004]. As an example of transformation, for the program of graphs, we obtain the following set-program:

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

Now, *next* collects all the possible reductions for each argument in a single rule and it is completed with failure rules for the cases c and d. The equality function $==$ (used in *path*) is a three-valued function that can produce *true*, *false* or F. The function *iTe* is a prefix version of the standard *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 derives statements of the form $\mathcal{S} \triangleleft \mathcal{C}$, where $\mathcal{C}$ is a *SAS* for $\mathcal{S}$. Here we only show a brief explanation of the rules (for a detailed discussion see [López et al. 2001]). 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 $\theta \in Subst_{\perp,\mathsf{F}}$, a substitution that includes the undefined value $\perp$ and F in its

[2] 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.

$$\textbf{(1)}\ \frac{}{\mathcal{S} \lhd \{\bot\}} \qquad \textbf{(2)}\ \frac{}{\{X\} \lhd \{X\}} \quad X \in \mathcal{V}$$

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

$$\textbf{(4)}\ \frac{\mathcal{S}\theta \lhd \mathcal{C}}{f(\overline{t})\theta \lhd \mathcal{C}} \quad \begin{array}{l}\text{if } \mathcal{C} \neq \{\bot\},\ (f(\overline{t}) \twoheadrightarrow \mathcal{S}) \in \mathcal{P} \\ \text{and } \theta \in Subst_{\bot,\mathsf{F}}\end{array}$$

$$\textbf{(5)}\ \frac{}{t == t' \lhd \{true\}} \text{ if } t \downarrow t' \qquad \textbf{(6)}\ \frac{}{t == t' \lhd \{false\}} \text{ if } t \uparrow t'$$

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

$$\textbf{(9)}\ \frac{}{f(\overline{t}) \lhd \{\mathsf{F}\}} \quad \begin{array}{l}\text{for all } (f(\overline{s}) \twoheadrightarrow \mathcal{S}') \in \mathcal{P}, \\ \overline{t} \text{ and } \overline{s} \text{ have a } CS \cup \{\mathsf{F}\}\text{-conflict}\end{array}$$

$$\textbf{(10)}\ \frac{}{t == t' \lhd \{\mathsf{F}\}} \quad \text{if } t \not\downarrow t' \text{ y } t \not\uparrow t' \qquad \textbf{(11)}\ \frac{\mathcal{S} \lhd \{\mathsf{F}\}}{fails(\mathcal{S}) \lhd \{true\}}$$

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

Table 1: Rewriting logic *SRLF*

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) $\exists t \in \mathcal{C} - \{\bot, \mathsf{F}\}$ stands for a head normal form).

Using the first example of Section 2, this logic can derive $safe(a) \lhd \{false\}$ 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:

$$\mathcal{S} \Box \delta \underset{\theta}{\rightsquigarrow} \mathcal{S}' \Box \delta'$$

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 $==$). A substitution σ is a solution of δ,

Cntx	$C\ [\mathcal{S}]\Box\delta \underset{\theta}{\leadsto} C\theta\ [\mathcal{S}']\Box\delta'$ if $\mathcal{S}\Box\delta \underset{\theta}{\leadsto} \mathcal{S}'\Box\delta'$
Nrrw$_1$	$f(\overline{t})\Box\delta \underset{\theta\,\vert_{var(\overline{t})}}{\leadsto} \mathcal{S}\theta\Box\delta'$ 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 $\delta' \in solve(\delta\theta)$
Nrrw$_2$	$f(\overline{t})\Box\delta \underset{\epsilon}{\leadsto} \{\mathsf{F}\}\Box\delta$ 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	$t == s\Box\delta \underset{\theta\,\vert_{var(t)\cup var(s)}}{\leadsto} \{\omega\}\Box\delta'$ if $t == s \rightarrowtail_{\theta} \omega\vert_{\delta''}$ and $\delta' \in solve(\delta\theta \cup \delta'')$
Fail$_1$	$fails(\mathcal{S})\Box\delta \underset{\epsilon}{\leadsto} \{true\}\Box\delta$ if $\mathcal{S}^* = \{\mathsf{F}\}$
Fail$_2$	$fails(\mathcal{S})\Box\delta \underset{\epsilon}{\leadsto} \{false\}\Box\delta$ if $\exists t \in \mathcal{S}^* - \{\bot, \mathsf{F}\}$
Flat	$\bigcup_{\alpha\in\bigcup_{\beta\in\mathcal{S}_1}\mathcal{S}_2} \mathcal{S}_3\Box\delta \underset{\epsilon}{\leadsto} \bigcup_{\beta\in\mathcal{S}_1}\bigcup_{\alpha\in\mathcal{S}_2}\mathcal{S}_3\Box\delta$
Dist	$\bigcup_{\alpha\in\mathcal{S}_1\cup\mathcal{S}_2} \mathcal{S}_3\Box\delta \underset{\epsilon}{\leadsto} \bigcup_{\alpha\in\mathcal{S}_1}\mathcal{S}_3 \cup \bigcup_{\alpha\in\mathcal{S}_2}\mathcal{S}_3\Box\delta$
Bind	$\bigcup_{\alpha\in\{t\}} \mathcal{S}\Box\delta \underset{\epsilon}{\leadsto} \mathcal{S}[\alpha/t]\Box\delta$
Elim	$\bigcup_{\alpha\in\mathcal{S}'} \mathcal{S}\Box\delta \underset{\epsilon}{\leadsto} \mathcal{S}\Box\delta$ if $\alpha \notin FV(\mathcal{S})$

Table 2: Rules for *SNarr*

notated as $\sigma \in Sol(\delta)$, if σ transform δ in a set of trivial disequalities (pairs of terms with a conflict of constructor symbols at the same position). For example, $\sigma = [X/z, Y/s(z)]$ is a solution for $\{X \neq s(z), Y \neq X\}$.

Table 2 shows the rules for the narrowing relation *SNarr*. There are quite a lot technical subtlety in these rules, but here we only point out the most relevant aspects (see [López et al. 2002, López et al. 2003a, López et al. 2003b] 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;
- **Nrrw**$_1$ 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;

- $\mathbf{Nrrw}_2$ 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 [López et al. 2003a, López et al. 2003b]. 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 $\mathsf{F} == X$);

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

- 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 $\{t_1\} \cup \ldots \cup \{t_n\}$ (where $t_1, \ldots, t_n$ are total terms). As usual we consider the transitive closure $\mathcal{S}\Box\delta \overset{*}{\underset{\theta}{\rightsquigarrow}} \mathcal{S}'\Box\delta'$: $\mathcal{S}\Box\delta \underset{\theta_1}{\rightsquigarrow} \mathcal{S}_1\Box\delta_1 \underset{\theta_2}{\rightsquigarrow} \ldots \underset{\theta_n}{\rightsquigarrow} \mathcal{S}'\Box\delta'$ where $\theta = \theta_1\theta_2\ldots\theta_n$ is the answer substitution of the derivation.

Soundness and completeness results for the relation *SNarr* with respect to *SRLF* were established in [López et al. 2002] without considering the equality function $==$ and they were extended with equality in [López et al. 2003a, López et al. 2003b] (see [Sánchez 2004] for detailed proofs).

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 of a Set-Expression). Given a set-expression $\mathcal{S}$, the set $DV(\mathcal{S})$ of demanded variables of $\mathcal{S}$ is a subset of Γ defined as:

- $DV(\{t\}) = var(t) \cap \Gamma$

- $DV(f(\overline{t})) = DVFun(f(\overline{t})) \cap \Gamma$, where

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

Using this notion, now we can 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. 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 (*add* demands its first argument), 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).

Now we can restrict the application of rule **Cntx** in the following way:

$$\boxed{\begin{array}{l} \textbf{DDCntx}\ C\ [\mathcal{S}]\Box\delta \underset{\theta}{\leadsto} C\theta\ [\mathcal{S}']\Box\delta' \\ \\ \qquad\qquad \text{if } C \text{ is a context of demand and } \mathcal{S}\Box\delta \underset{\theta}{\leadsto} \mathcal{S}'\Box\delta' \end{array}}$$

The relation *DDSNarr* is the result of replacing the rule **Cntx** by the new rule **DDCntx** in *SNarr*. The new relation limits the application of rules of *SNarr* and we must prove that soundness and completeness are not missing with such a restriction. Soundness of *DDSNarr* arises from the soundness of *SNarr*:

Theorem 2 (Soundness 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.$\square$

The completeness result requires rather more elaboration. It is not true that any derivation with *SNarr* can be done with *DDSNarr*. As **DDCntx** is more restrictive than **Cntx** it could be possible that *DDSNarr* become blocked or loop in a derivation for which *SNarr* could progress. We must show that in fact this does not happen, i.e., for any set-expression not in normal form *DDSNarr* can perform a derivation step. Moreover, this step makes the set-expression to evolve to a normal form if such form exists for the set-expression, or at least, to decrease the *complexity* in some order of complexity. Such a notion of *complexity* is defined with respect to the corresponding derivations within *SRLF*: given a *SRLF*-derivation $\mathcal{S} \lhd \mathcal{C}$ we define its complexity as the number of *SRLF*-steps of it.

We will show in Lemma 4 that given a set-expression $\mathcal{S}$ with $\mathcal{S} \lhd \mathcal{C}$, we can derive $\mathcal{S}'$ by means of *DDSNarr* with $\mathcal{S}' \lhd \mathcal{C}$ (this is only soundness), but in such a way that the complexity of the derivation for $\mathcal{S}' \lhd \mathcal{C}$ decreases. In order to prove this result, the first proposition shows that for any 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 3 (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 $\mathcal{S}\square\emptyset \underset{\epsilon}{\leadsto} \mathcal{S}'\square\emptyset$ 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 derivations for $\mathcal{S} \lhd \mathcal{C}$ and $\mathcal{S}' \lhd \mathcal{C}$ is the same;*
- *otherwise the complexity of the derivation for $\mathcal{S}' \lhd \mathcal{C}$ is less than the one for $\mathcal{S} \lhd \mathcal{C}$.*

Proof: 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*-derivations is affected:

- if $\mathcal{S} = f(\overline{t})$ and there is a rule of the program for reducing it, then the derivation for $\mathcal{S} \lhd \mathcal{C}$ has the form $\dfrac{\mathcal{S}_1\theta \lhd \mathcal{C}}{f(\overline{t}) \lhd \mathcal{C}}$ by rule 4 of *SRLF* and using a rule of the program $f(\overline{s}) \twoheadrightarrow \mathcal{S}_1$ such that $t = s\theta$. The corresponding *DDSNarr*-derivation is $f(\overline{t})\square\emptyset \underset{\epsilon}{\leadsto} \mathcal{S}_1\theta\square\emptyset$ by rule **Nrrw**$_1$. Then we have $\mathcal{S}' = \mathcal{S}_1\theta$ and clearly the complexity of the *SRLF*-derivation for $\mathcal{S}' \lhd \mathcal{C}$ is less than the one for $\mathcal{S} \lhd \mathcal{C}$, because the first is a sub-derivation of the second.

On the other hand, if no rule of the program matches $f(\overline{t})$ then the *SRLF*-derivation is done by rule (9) as $\frac{}{f(\overline{t}) \lhd \{\mathsf{F}\}}$, and the *DDSNarr*-derivation is $f(\overline{t})\Box\emptyset \underset{\epsilon}{\leadsto} \{\mathsf{F}\}\Box\emptyset$ by rule $\mathbf{Nrrw}_2$;

- if $\mathcal{S} = t == s$ it is possible to use **Eq** and finish the derivation. Notice that it does not need to introduce any substitution or disequality because we have $\mathcal{S} \lhd \mathcal{C}$ (with $\mathcal{C} \neq \{\bot\}$ and without any restriction to the variables of $\mathcal{S}$). In this case one of the rules 5, 6 or 10 of *SRLF* is applicable;
- if $\mathcal{S} = \mathit{fails}(\mathcal{S}_1)$ then if it is possible to apply $\mathbf{Fail}_1$ or $\mathbf{Fail}_2$ the derivation finish. Otherwise **DDCntx** allows to apply some rule to $\mathcal{S}_1$ and induction hypothesis applies;
- 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 and the complexity of the *SRLF*-derivation decreases;
  - 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 $\mathcal{S}_2$ by means of **DDCntx**, and induction hypothesis applies;
  - 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*-derivations do not change; otherwise induction hypothesis applies to $\mathcal{S}_1$ by means of **DDCntx**;
- if $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ we can apply induction hypothesis to both set-expression by means of **DDCntx**.$\Box$

Lemma 4 (Progress of *DDSNarr*). *Let $\mathcal{S}$ be a set-expression not in normal form such that there exists a SRLF-derivation for $\mathcal{S} \lhd \mathcal{C}$ for it with $\mathcal{C} \neq \{\bot\}$. Then it is possible to perform a (finite) derivation of the form $\mathcal{S}\Box\emptyset \overset{*}{\underset{\epsilon}{\leadsto}} \mathcal{S}'\Box\emptyset$ with DDSNarr in such a way that there is a SRLF-derivation for $\mathcal{S}' \lhd \mathcal{C}$ with less complexity than the one for $\mathcal{S} \lhd \mathcal{C}$.*

Proof: notice that rules **Flat** and **Dist** can only be applied a finite number of times over a set-expression and they can not obtain a normal form; then, Proposition 3 ensures that some other rule will be applicable reducing the complexity of the *SRLF*-derivation.$\Box$

Lemma 4 uses *DDSNarr* as a rewriting relation instead of a narrowing one: no substitutions are involved. In order to obtain the result of completeness we need to prove that *DDSNarr* is able to find the appropriate values (and disequalities) for variables, i.e., the appropriate substitutions (and disequality constraints δ) for narrowing a set-expression. Moreover, in general the narrowing relation will provide a more general substitution than the one used for the *SRLF*-derivation: if $\mathcal{S}\theta \lhd \mathcal{C}$ 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 $\mathbf{Nrrw}_1$).

Theorem 6 formalizes this idea. The next Lemma is a rather technical result needed to prove such a theorem. It establishes the bridge between pure rewriting and narrowing within *DDSNarr*. Here the rewriting logic *SRLF* is only used to ensure that the progress of *DDSNarr* (in the sense of Lemma 4) is not affected.

Lemma 5 (Answer Substitutions). *Let $\mathcal{S}$ be a set-expression, δ a set of disequalities in solved form, $\theta \in Sol(\delta)$. 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:*

i) $\theta = (\theta'\mu)\mid_{\mathcal{V}-\Pi}$

ii) if SRLF can derive $\mathcal{S}' \lhd \mathcal{C}$ then it also can derive $\mathcal{S}''\mu \lhd \mathcal{C}$ with the same complexity

iii) $Sol(\delta') \subseteq Sol(\delta''\mu)$

Proof sketch: this Lemma (with minor changes) was introduced for *SNarr* without disequalities in [López et al. 2002] and it was extended for manipulating disequalities in [López et al. 2003a, López et al. 2003b]. The proof was done 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 $\mathcal{S}\Box\delta \overset{*}{\underset{\theta'}{\leadsto}} \mathcal{S}''\Box\delta''$ (see [Sánchez 2004] for a detailed proof). This idea is also applicable to the relation *DDSNarr* and the proof of the current result is essentially the same.$\Box$

Now, the completeness theorem is obtained by considering simultaneously the progress of *DDSNarr* (Lemma 4) and its ability for finding answer substitutions (Lemma 5). Part *ii*) of the next theorem reflects the fact that *DDSNarr* makes the set-expressions to evolve to less complex form, what means that it is able to obtain the values of the semantics of a set-expression.

Theorem 6 (Completeness of ***DDSNarr*****).** *Let $\mathcal{S}$ be a set-expression not in normal form, δ a set of disequalities in solved form, $\theta \in Sol(\delta)$ and a SRLF-derivation for $\mathcal{S}\theta \lhd \mathcal{C}$ with $\mathcal{C} \neq \{\perp\}$. 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) $\theta = (\theta'\mu)\mid_{\mathcal{V}-\Pi}$

ii) *there is a SRLF-derivation for $\mathcal{S}'\mu \lhd \mathcal{C}$ with less complexity than the one for $\mathcal{S}\theta \lhd \mathcal{C}$*

iii) $\mu \in Sol(\delta')$

Proof: if $\mathcal{S}\theta \lhd \mathcal{C}$, then by Lemma 4, *DDSnarr* can perform a finite derivation $\mathcal{S}\theta\Box\emptyset \overset{*}{\underset{\epsilon}{\leadsto}} \mathcal{S}_1\Box\emptyset$, such that there is a derivation for $\mathcal{S}_1 \lhd \mathcal{C}$ with less complexity than the one for $\mathcal{S}\theta \lhd \mathcal{C}$. By Lemma 5 there exist a derivation $\mathcal{S}\Box\delta \overset{*}{\underset{\theta'}{\leadsto}} \mathcal{S}'\Box\delta'$ with new variables Π such that

i) $\theta = (\theta'\mu)\mid_{\mathcal{V}-\Pi}$

ii) there is a derivation for $\mathcal{S}'\mu \lhd \mathcal{C}$ with the same complexity of the one for $\mathcal{S}_1 \lhd \mathcal{C}$. Then the complexity of $\mathcal{S}'\mu \lhd \mathcal{C}$ decreases with respect to $\mathcal{S}\theta \lhd \mathcal{C}$, as we expect;

iii) $Sol(\emptyset) \subseteq Sol(\delta'\mu)$, what means $\mu \in Sol(\delta')$. $\Box$

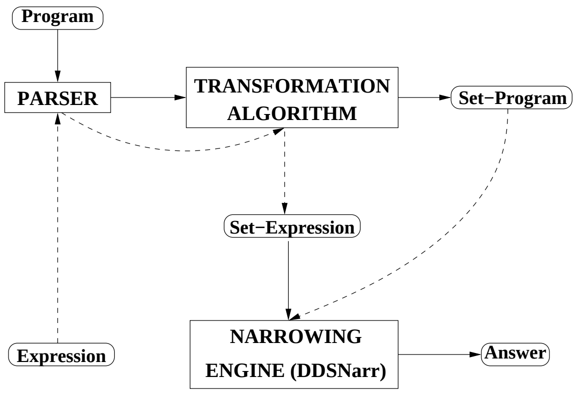
As a corollary, we assume the existence of a normal form for a set-expression and show that *DDSNarr* allows to obtain such a normal form. A normal form is any set-expression of the form $\{t_1\} \cup \ldots \{t_n\}$ with $t_1, \ldots, t_n$ total terms (without any $\perp$). This set-expression corresponds to the *SAS* $\{t_1, \ldots, t_n\}$, so the result we are looking for claims that any total *SAS* of the semantics of a set-expression can be obtained by a *DDSNarr*-derivation as a normal form. Furthermore, such a derivation generalizes any substitution used for obtaining such a *SAS*, possibly by introducing disequalities as a part of the answer.

Corollary 7 (Normalization). *Let $\mathcal{S}$ be a set-expression and $\theta \in Subst$ (with $Dom(\theta) \subseteq FV(\mathcal{S})$) such that $\mathcal{S}\theta \lhd \{t_1, \ldots, t_n\}$, with $t_1, \ldots, t_n$ total terms (without $\perp$). 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: $\theta = \theta'\mu$ and $\mu \in Sol(\delta)$.*

As an easy example of the use of disequalities and its benefits consider the set expression $X == s(Y)$. It can be reduced by *SRLF* to $\{true\}$ under the substitution $[X/s(Y)]$, and it reduces to $\{false\}$ under the substitutions $[X/z]$ or $[X/s(z), Y/s(z)]$ or $[X/s(z), Y/s(s(z))]\ldots$ (an infinite number of substitutions allows such a derivation). *DDSNarr* will instead produce the answer $\{false\}$ with a single disequality $X \neq s(Y)$ (of course, it also produces the answer $\{true\}$ with the substitution $[X/s(Y)]$).

6 Implementation

OOPS, like Curry or $\mathcal{TOY}$ is implemented in Prolog (SWI-Prolog) and users of these systems must be comfortable using it. Programs use the standard (currified functional) syntax and the interpreter is analogous to $\mathcal{TOY}$ or Curry (type `:h` from the prompt of OOPS for a list of commands). 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).

Set-expressions have an internal representation as Prolog terms as follows:

- $\{t\}$ is represented as the unitary list `[t]`.
- $f(\bar{t})$ is self-represented.
- $fails(\mathcal{S})$ is represented as `fails`$(tp_{\mathcal{S}})$, where $tp_{\mathcal{S}}$ is the Prolog representation for $\mathcal{S}$.

- $t == t'$ is represented as `eq([(t,t')])`. In general the argument of `eq` is a list of pairs of terms 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 terms. For example, assuming the constructors z and s for natural numbers and $c \in DC^2$, the equality $c(z, s(X)) == c(Y, Z)$ is represented initially as $eq([(c(z, s(X)),\ c(Y, Z))])$. After decomposition it is transformed into $eq([(z, Y),\ (s(X), Z))])$.

- $\bigcup_{\alpha\in\mathcal{S}_1} \mathcal{S}_2$ is represented as `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 due to the different behavior of both types of variables in the narrowing relation.

- $\mathcal{S}_1 \cup \mathcal{S}_2$ corresponds to the term $tp_{\mathcal{S}_1} + tp_{\mathcal{S}_2}$, where $tp_{\mathcal{S}_1}$ and $tp_{\mathcal{S}_2}$ are the representations of $\mathcal{S}_1$ and $\mathcal{S}_2$ resp.

- finally, the term `F` is represented by the Prolog constant `fail`.

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:

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

after some reduction steps (with $\mathbf{Nrrw}_1$, **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 as default[3]. 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. In particular, if a set-expression $\mathcal{S}$ has a SAS $\{t_1, \ldots, t_n\}$ where $t_1, \ldots, t_n$ are total terms, Corollary 7 claims that *there exists* a *DDSNarr* derivation that obtains this SAS as a normal form, but does not point out how to choose the redexes for perfoming such a derivation. In OOPS with policy of selecting all possible redexes ensure that we proceed with the appropriate one and we will reach the expected normal form. To illustrate the advantage of such a policy assume the following two functions $loop = loop$ and $f = true$ and the set-expression $fails(loop \cup f)$. There are two possible redexes: $loop$ and f. We can choose the first and go into an infinite computation or we can choose the second and derive $fails(loop \cup \{true\})$ and then $\{false\}$. If we choose both simultaneously we ensure that we get the desired behavior.

As we have seen, a reduction *step* in OOPS may correspond to several derivation steps (at least one) with the rules of *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

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

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 $\bigcup_{\pi \in add(z,X)} \{s(\pi)\}$ by $\mathbf{Nrrw}_1$ and the first rule for *add*, 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.

[3] It allows to change this mode of operation to depth search (with the command `:nw`) and force the system to choose the first possible redex.

(1)	$fails(\underline{path}(X,Y))\Box\emptyset \rightsquigarrow_{\epsilon}$	*(Nrrw$_1$)*
(2)	$fails(\bigcup_{\alpha\in\underline{X==Y}}\bigcup_{\beta\in\bigcup_{\gamma\in next(X)} path(\gamma,Y)} iTe(\alpha, true, \beta))\Box\emptyset \rightsquigarrow_{\epsilon}$	*(Nrrw$_3$)*
(3)	$fails(\underline{\bigcup_{\alpha\in\{false\}}\bigcup_{\beta\in\bigcup_{\gamma\in next(X)} path(\gamma,Y)} iTe(\alpha, true, \beta))}\Box\{X\neq Y\} \rightsquigarrow_{\epsilon}$	*(Bind)*
(4)	$fails(\bigcup_{\beta\in\bigcup_{\gamma\in next(X)} path(\gamma,Y)} \underline{iTe(false, true, \beta)})\Box\{X\neq Y\} \rightsquigarrow_{\epsilon}$	*(Nrrw$_1$)*
(5)	$fails(\underline{\bigcup_{\beta\in\bigcup_{\gamma\in next(X)} path(\gamma,Y)}\{\beta\})}\Box\{X\neq Y\}$	*(Flat)*
(6)	$fails(\bigcup_{\gamma\in next(X)}\bigcup_{\beta\in\underline{path(\gamma,Y)}}\{\beta\})\Box\{X\neq Y\} \rightsquigarrow_{\epsilon}$	*(Nrrw$_1$)*
(7)	$fails(\bigcup_{\gamma\in next(X)}$ $\underline{\bigcup_{\beta\in\bigcup_{\tau\in\gamma==Y}\bigcup_{\mu\in\bigcup_{\nu\in next(\gamma)} path(\nu,Y)} iTe(\tau,true,\mu)}\{\beta\})}\Box\{X\neq Y\} \rightsquigarrow_{\epsilon}$	*(Flat)*
(8)	$fails(\bigcup_{\gamma\in next(X)}\bigcup_{\tau\in\gamma==Y}$ $\underline{\bigcup_{\beta\in\bigcup_{\mu\in\bigcup_{\nu\in next(\gamma)} path(\nu,Y)} iTe(\tau,true,\mu)}\{\beta\})}\Box\{X\neq Y\} \rightsquigarrow_{\epsilon}$	*(Flat)*
(9)	$fails(\bigcup_{\gamma\in\underline{next(X)}}\bigcup_{\tau\in\gamma==Y}\bigcup_{\mu\in\bigcup_{\nu\in(next(\gamma)} path(\nu,Y)}$ $\bigcup_{\beta\in(iTe(\tau,true,(\mu)}\{\beta\})\Box\{X\neq Y\}\rightsquigarrow_{[X/d]}$	*(Nrrw$_1$)*
(10)	$fails(\underline{\bigcup_{\gamma\in\{\mathsf{F}\}}\bigcup_{\tau\in\gamma==Y}\bigcup_{\mu\in\bigcup_{\nu\in next(\gamma)} path(\nu,Y)}}$ $\underline{\bigcup_{\beta\in iTe(\tau,true,\mu)}\{\beta\})}\Box\{Y\neq d\} \rightsquigarrow_{\epsilon}$	*(Bind)*
(11)	$fails(\bigcup_{\tau\in\underline{Y==\mathsf{F}}}\bigcup_{\mu\in\bigcup_{\nu\in next(\mathsf{F})} path(\nu,Y)}\bigcup_{\beta\in iTe(\tau,true,\mu)}\{\beta\})\Box\{Y\neq d\} \rightsquigarrow_{\epsilon}$	*(Nrrw$_3$)*
(12)	$fails(\underline{\bigcup_{\tau\in\{\mathsf{F}\}}\bigcup_{\mu\in\bigcup_{\nu\in next(\mathsf{F})} path(\nu,Y)}\bigcup_{\beta\in iTe(\tau,true,\mu)}\{\beta\})}\Box\{Y\neq d\} \rightsquigarrow_{\epsilon}$	*(Bind)*
(13)	$fails(\underline{\bigcup_{\mu\in\bigcup_{\nu\in next(\mathsf{F})} path(\nu,Y)}\bigcup_{\beta\in iTe(\mathsf{F},true,\mu)}\{\beta\})}\Box\{Y\neq d\} \rightsquigarrow_{\epsilon}$	*(Elim)*
(14)	$fails(\bigcup_{\beta\in\underline{iTe(\mathsf{F},true,\mu)}}\{\beta\})\Box\{Y\neq d\} \rightsquigarrow_{\epsilon}$	*(Nrrw$_2$)*
(15)	$fails(\underline{\bigcup_{\beta\in\{\mathsf{F}\}}\{\beta\})}\Box\{Y\neq d\} \rightsquigarrow_{\epsilon}$	*(Bind)*
(16)	$\underline{fails(\{\mathsf{F}\})}\Box\{Y\neq d\} \rightsquigarrow_{\epsilon}$	*(Fail$_1$)*
(17)	$\{true\}\Box\{Y\neq d\}$	
	ANSWER: $\{true\}$ with $[X/d]$ and $Y\neq d$	

Table 3: A derivation for $fails(path(X,Y))$

As an example of derivation Table 3 shows a derivation for one of the possible evaluations for the expression $fails(path(X, Y))$ using the set-program of graphs (see Section 3). The (underlined) redexes are selected with the previous explained mechanism. Notice that in step (2) the equality $X == Y$ is reduced to *false* by imposing the disequality $X \neq Y$. In step (9) Y is binded to d and the disequality is transformed into $X \neq d$. As a relevant feature of OOPS, it is able to automatically generate these kind of traces by turning the system into debugging mode (command `:d`). Then the derivation steps are written in a LaTeX file, compiled into a postscript file and showed in a readable format. The traces also contain the set-program obtained by transformation in order to facilitate the analysis of derivations.

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 not only to reasoning about the formalism itself but also to explore the practical implications of the integration of failure in the *FLP* paradigm. 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

[Álvez et al. 2004] 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.

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

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

[Apt 2000] K.R. Apt. Logic programming and Prolog. Unpublished tutorial, 2000.

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

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

[Chan 1998] Chan, D. Constructive negation based on the completed database. In *Proc. ICSLP'88*, pages 111–125, 1988.

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

[González et al 1999] 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. Journal of Logic Programming, 40(1):47–87, 1999.

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

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

[Loogen et al. 1993] 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.

[López et al. 1999] 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.

[López et al. 2000] 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.

[López et al. 2001] 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.

[López et al. 2002] 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.

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

[López et al. 2003b] 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`

[López et al. 2004] F.J. López-Fraguas and J. Sánchez-Hernández. A proof theoretic approach to failure in functional logic programming. Theory and Practice of Logic Programming, 4(1&2):41–74, 2004.

[Moreno 1996] 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.

[Moreno et al. 2004] 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.

[Reynolds 1998] J.C. Reynolds. Theories of Programing Languages. Cambridge University Press, 1998.

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

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