Narrowing Failure in Functional Logic Programming

Francisco Javier López-Fraguas and Jaime Sánchez-Hernández

Dep. Sistemas Informáticos y Programación, Univ. Complutense de Madrid
{fraguas,jaime}@sip.ucm.es

Abstract. Negation as failure is an important language feature within the logic programming paradigm. The natural notion generalizing negation as failure in a functional logic setting is that of finite failure of reduction. In previous works we have shown the interest of using such programming construct when writing functional logic programs, and we have given a logical status to failure by means of proof calculi designed to deduce failures from programs. In this paper we address the problem of the operational mechanism for the execution of functional logic programs using failure. Our main contribution is the proposal of a narrowing relation able to deal with failures, which is constructive in the usual sense of the term in the context of negation, that is, narrowing is able to find substitutions for variables even in presence of failures. As main technical results, we prove correctness and completeness of the narrowing relation with respect to the proof-theoretic semantics.

1 Introduction

Motivation Functional logic programming (*FLP* for short) tries to combine the nicest properties of functional and logic programming (see [9] for a now 'classical' survey on *FLP*). Theoretical aspects of *FLP* are well established (e.g. [8]) and there are also practical implementations such as *Curry* [10] or $\mathcal{TOY}$ [1, 11]. Disregarding syntax, both pure Prolog and (a wide subset of) Haskell are subsumed by those systems. The usual claim is then that by the use of an *FLP* system one can choose the style of programming better suited to each occasion.

There is nevertheless a very important feature in the logic programming (*LP* for short) paradigm, namely *negation as failure* [6], yet not available in *FLP* systems. Negation is a major issue in the *LP* field, both at the theoretical and the practical levels. There are hundreds of papers about negation in *LP* (see e.g. [4] for a survey), and almost all real Prolog programs use negation at some point.

This situation poses some problems to *FLP*, since logic programs using negation cannot be directly seen as *FLP* programs. This would be a not very serious inconvenience if other features of *FLP* could easily replace the use of failure. This happens in some cases, when the possibility of defining two-valued boolean functions is a good alternative to the use of negation in a logic program.

Z. Hu and M. Rodríguez-Artalejo (Eds.): FLOPS 2002, LNCS 2441, pp. 212–227, 2002.

But there are problems where the use of failure as a programming construct is a real aid to the writing of concise declarative programs. We give an example – to be used several times throughout the paper – of that situation in the context of *FLP*. Some other examples can be found in [15].

Example Consider the problem of determining if there is a path connecting two nodes of an acyclic directed graph. A typical way of representing a graph in current *FLP* languages is by means of a non-deterministic function *next*, with rules of the form $next(N) \to N'$, indicating that there is an arc from N to N'. A concrete graph with nodes a, b, c and d could be given by the rules:

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

And to determine if there is a path from X to Y we can define:

$$path(X,Y) \to if\ \ eq(X,Y)\ then\ true\ else\ path(next(X),Y)$$

where *eq* stands for strict equality, which can be defined by the rules $eq(a,a) \to true$, $eq(a,b) \to false$ and so on.

Notice that *path* behaves as a semidecision procedure recognizing only the positive cases, and there is no clear way (in 'classical' *FLP*) of completing its definition with the negatives ones, unless we change from the scratch the representation of graphs. Therefore we cannot, for instance, program in a direct way a property like

$$safe(X) \Leftrightarrow X\ is\ not\ connected\ with\ d$$

To this purpose, something like negation as failure would be very useful. Since predicates can be programmed in *FLP* systems like *true*-valued functions, a natural *FLP* generalization of negation as failure is given by the notion of *failure of reduction to head normal form*. This could be expressed by means of a 'primitive' function *fails*, with the following intended behavior:

$$fails(e) ::= \begin{cases} true & \text{if } e \text{ fails to be reduced to hnf} \\ false & \text{otherwise} \end{cases}$$

Using this primitive it is now easy to define the property *safe*:

$$safe(X) \to fails(path(X,d))$$

With this definition and the previous graph, $safe(X)$ becomes $true$ for $X = c$ and $false$ for $X = a$, $X = b$ and $X = d$.

Previous and related work. Aim of the paper In some previous works [12, 15, 13] we addressed the problem of failure within *FLP*, from the point of view of its semantic foundations. Our starting point was *CRWL* [7, 8], a general framework for *FLP*, based on a *C*onstructor-based *Re*writing *L*ogic. From the point of view of programming, a fundamental notion of *CRWL* (as well as for existing *FLP* systems like *Curry* [10] or $\mathcal{TOY}$ [11, 1]) is that of non-strict

non-deterministic function, for which *call-time choice semantics* is considered. Programs in *CRWL* have a logical semantics given by the logical consequences of the program according to a proof calculus able to prove reduction statements of the form $e \to t$, meaning that one of the possible reductions of an expression e results in the (possibly partial) value t.

Our first step [12, 15] for dealing with failure in *FLP*, was to extend the rewriting logic *CRWL* to *CRWLF* (*CRWL* "with failure"). The main insight was to replace statements $e \to t$ by statements $e \lhd \mathcal{C}$, where $\mathcal{C}$ are sets of partial values (called *S*ufficient *A*pproximation *S*ets or *SAS*'s) corresponding to the different possibilities for reducing e. In [13] we realized the benefits of making more explicit the set nature of non deterministic functions, and therefore we reformulated *CRWLF* by giving a set-oriented view of programs. This was done even at the syntactic level, by introducing classical mathematical set notation, like union or indexed unions; union is useful to transform programs to an inductively sequential format [2], and indexed unions turn to be useful for expressing call-time choice semantics and sharing.

One of the motivations for the set-oriented view adopted in [13] was our thought that such approach, by reflecting better the meaning of programs, would be an appropriate semantic basis upon which develop a suitable operational semantics for functional logic programs using failure. This is exactly the purpose of this paper. We propose an operational mechanism given by a narrowing relation which can operate with failure, that is, which is able to narrow expressions of the form *fails(e)*. Since we start from a precise semantic interpretation of failure, we are able to prove correctness and completeness of narrowing with respect to the semantics.

A major feature of our proposal is that the narrowing relation is *constructive* in the usual sense given to the term in the context of negation [5, 19, 20]: if an expression *fails(e)* contains variables, it can still be narrowed to obtain appropriate substitutions for them. For instance, in the example of the graph, our narrowing relation is able to narrow the expression *safe(X)* to obtain the value *true* together with the substitution $X = c$ and the value $false$ together with the substitutions (corresponding to different computations) $X = a$, $X = b$ and $X = d$.

There are very few works about negation in *FLP*. In [16] the work of Stuckey about constructive negation [19, 20] is adapted to the case of *FLP* with strict functions and innermost narrowing as operational mechanism. In [17] a similar work is done for the case of non-strict functions and lazy narrowing. This approach, apart from being in the technical side very different from ours, does not take into account non-determinism of functions, an essential aspect in our work.

The organization of the paper is as follows: Section 2 contains some technical preliminaries. In Section 3 we present (a slight variant of) the set-oriented semantic framework for failure of [13], including the proof calculus, together with some new results which are needed for subsequent sections. Section 4 is the core of the paper, where we define the narrowing relation and prove the results of correctness and completeness with respect to the logical semantics. Finally, Sec-

tion 5 summarizes some conclusions and hints for future work. Due to lack of space, proofs are omitted, but can be found in [14].

2 Technical Preliminaries

We assume a signature $\Sigma = DC_\Sigma \cup FS_\Sigma \cup \{fails\}$ where $DC_\Sigma = \bigcup_{n\in\mathbb{N}} DC^n_\Sigma$ is a set of *constructor* symbols containing at least the usual boolean ones *true* and $false$, $FS_\Sigma = \bigcup_{n\in\mathbb{N}} FS^n_\Sigma$ is a set of *function* symbols, all of them with associated arity and such that $DC_\Sigma \cap FS_\Sigma = \emptyset$, and $fails \notin DC \cup FS$ (with arity 1). We also assume a countable set $\mathcal{V}$ of *variable* symbols. We write $Term_\Sigma$ for the set of (total) *terms* (we say also *expressions*) built over Σ and $\mathcal{V}$ in the usual way, and we distinguish the subset $CTerm_\Sigma$ of (total) constructor terms or (total) *cterms*, which only make use of DC_Σ and $\mathcal{V}$. The subindex Σ will usually be omitted. Terms intend to represent possibly reducible expressions, while cterms represent data values, not further reducible.

The constant (0-arity constructor) symbol F is explicitly used in our terms, so we consider the signature $\Sigma_\mathsf{F} = \Sigma \cup \{\mathsf{F}\}$. This symbol F will be used to express failure of reduction. The sets $Term_\mathsf{F}$ and $CTerm_\mathsf{F}$ are defined in the natural way. The denotational semantics also uses the constant symbol $\bot$, that plays the role of the undefined value. We define $\Sigma_{\bot,\mathsf{F}} = \Sigma \cup \{\bot, \mathsf{F}\}$; the sets $Term_{\bot,\mathsf{F}}$ and $CTerm_{\bot,\mathsf{F}}$ of (partial) terms and (partial) cterms respectively, are defined in a natural way. Partial cterms represent the result of partially evaluated expressions; thus, they can be seen as approximations to the value of expressions in the denotational semantics.

As usual notations we will write $X, Y, Z, ...$ for variables, c, d for constructor symbols, f, g for functions, e for terms and s, t for cterms. In all cases, primes (') and subindices can be used.

The sets of substitutions $CSubst, CSubst_\mathsf{F}$ and $CSubst_{\bot,\mathsf{F}}$ are defined as applications from $\mathcal{V}$ into $CTerm, CTerm_\mathsf{F}$ and $CTerm_{\bot,\mathsf{F}}$ respectively. We will write θ, σ, μ for substitutions and ϵ for the identity substitution. The notation $\theta\sigma$ stands for the usual composition of substitutions. All the considered substitutions are idempotent ($\theta\theta = \theta$). We write $[X_1/t_1, ..., X_n/t_n]$ for the substitution that maps X_1 into t_1, ..., X_n into t_n.

Given a set of constructor symbols D, we say that the terms t and t' have a *D-clash* if they have different constructor symbols of D at the same position. We say that two tuples of cterms $t_1, ..., t_n$ and $t'_1, ..., t'_n$ have a *D-clash* if for some $i \in \{1, ..., n\}$ the cterms t_i and t'_i have a *D-clash*.

A natural *approximation ordering* $\sqsubseteq$ over $Term_{\bot,\mathsf{F}}$ can be defined as the least partial ordering over $Term_{\bot,\mathsf{F}}$ satisfying the following properties:

- $\bot \sqsubseteq e$ for all $e \in Term_{\bot,\mathsf{F}}$,
- $h(e_1, ..., e_n) \sqsubseteq h(e'_1, ..., e'_n)$, if $e_i \sqsubseteq e'_i$ for all $i \in \{1, ..., n\}$, $h \in DC \cup FS \cup \{fails\} \cup \{\mathsf{F}\}$

The intended meaning of $e \sqsubseteq e'$ is that e is less defined or has less information than e'. Two expressions $e, e' \in Term_{\bot,\mathsf{F}}$ are *consistent* if they can be refined to

obtain the same information, i.e., if there exists $e'' \in Term_{\perp,\mathsf{F}}$ such that $e \sqsubseteq e''$ and $e' \sqsubseteq e''$. Notice that according to this F is maximal. This is reasonable from an intuitive point of view, since F represents 'failure of reduction' for an expression and this gives a no further refinable information about the result of the evaluation of such expression. This contrasts with the status given to failure in [17], where F is chosen to verify $\mathsf{F} \sqsubseteq t$ for any t different from $\perp$. We also use the relation $\sqsubseteq$ referred to substitutions: $\sigma \sqsubseteq \sigma'$ iff $X\sigma \sqsubseteq X\sigma'$ for all $X \in \mathcal{V}$. And for tuples of cterms: $\overline{t} \sqsubseteq \overline{t}'$ iff the ordering relation is pairwise satisfied.

Extending $\sqsubseteq$ to sets of terms results in the *Egli-Milner* preordering (see e.g. [18]): given $D, D' \subseteq CTerm_{\perp,\mathsf{F}}$ we say that D' is more refined than D and write $D \sqsubseteq D'$ iff for all $t \in D$ there exists $t' \in D'$ with $t \sqsubseteq t'$ and for all $t' \in D'$ there exists $t \in D$ with $t \sqsubseteq t'$. The sets D and D' are **consistent** iff there exists D'' such that $D \sqsubseteq D''$ and $D' \sqsubseteq D''$.

3 A Semantic Framework for *FLP* with Failure

3.1 Set Expressions

A set-expression is a syntactical construction designed for manipulating sets of values. A (total) set-expression $\mathcal{S}$ is defined as:

$$\mathcal{S} ::= \{t\} \mid f(\overline{t}) \mid fails(\mathcal{S}_1) \mid \bigcup_{X \in \mathcal{S}_1} \mathcal{S}_2 \mid \mathcal{S}_1 \cup \mathcal{S}_2$$

where $t \in CTerm_{\mathsf{F}}$, $\overline{t} \in CTerm_{\mathsf{F}} \times ... \times CTerm_{\mathsf{F}}$, $f \in FS^n$, and $\mathcal{S}_1, \mathcal{S}_2$ are set-expressions. We write $SetExp$ for the set of (total) set-expressions. The set $SetExp_\perp$ of *partial* set-expressions is analogous but $t, \overline{t}$ can contain $\perp$.

Indexed unions bind variables in a similar way to other more familiar constructs like first order quantifications or λ-abstraction. The set $PV(\mathcal{S})$ of bound or *produced variables* of a set-expression $\mathcal{S}$ can be formally defined as:

- $PV(\{t\}) = PV(f(\overline{t})) = \emptyset$
- $PV(fails(\mathcal{S})) = PV(\mathcal{S})$
- $PV(\bigcup_{X \in \mathcal{S}_1} \mathcal{S}_2) = \{X\} \cup PV(\mathcal{S}_1) \cup PV(\mathcal{S}_2)$
- $PV(\mathcal{S}_1 \cup \mathcal{S}_2) = PV(\mathcal{S}_1) \cup PV(\mathcal{S}_2)$

We can also define the set $FV(\mathcal{S})$ of *free variables* of a set-expression $\mathcal{S}$ as:

- $FV(\{t\}) = var(t)$
- $FV(f(\overline{t})) = var(\overline{t})$
- $FV(fails(\mathcal{S}_1)) = FV(\mathcal{S}_1)$
- $FV(\bigcup_{X \in \mathcal{S}_1} \mathcal{S}_2) = (FV(\mathcal{S}_2) - \{X\}) \cup FV(\mathcal{S}_1)$
- $FV(\mathcal{S}_1 \cup \mathcal{S}_2) = FV(\mathcal{S}_1) \cup FV(\mathcal{S}_2)$

In order to avoid variable renamings and simplify further definitions, we add an *admissibility condition* to set expressions: for $\bigcup_{X \in \mathcal{S}_1} \mathcal{S}_2$ we require $X \notin var(\mathcal{S}_1) \cup PV(\mathcal{S}_2)$ and $PV(\mathcal{S}_1) \cap PV(\mathcal{S}_2) = \emptyset$; and $\mathcal{S}_1 \cup \mathcal{S}_2$ must verify $PV(\mathcal{S}_1) \cap FV(\mathcal{S}_2) = \emptyset$ and $PV(\mathcal{S}_2) \cap FV(\mathcal{S}_1) = \emptyset$. Notice that with this conditions, the sets $PV(\mathcal{S})$ and $FV(\mathcal{S})$ define a partition over $var(\mathcal{S})$. In the following we always assume this admissibility condition over set-expressions.

As an example, if $f \in FS^2$ and $c \in DC^2$, then

$$\mathcal{S} = \bigcup_{A \in \bigcup_{B \in f(X,Y)} \{B\}} \{c(A, X)\} \cup \bigcup_{C \in \{X\}} f(C, Y)$$

is an admissible set-expression with $PV(\mathcal{S})=\{A,B,C\}$ and $FV(\mathcal{S})=\{X,Y\}$.

We assume also some admissibility conditions over substitutions: given a set-expression $\mathcal{S}$ we say that σ is an *admissible substitution for* $\mathcal{S}$, if $Dom(\sigma) \cap PV(\mathcal{S}) = \emptyset$ and $Ran(\sigma) \cap PV(\mathcal{S}) = \emptyset$. In such case, the set-expression $\mathcal{S}\sigma$ is naturally defined as:

$$\bullet\ \{t\}\sigma = \{t\sigma\} \qquad \bullet\ f(\overline{t})\sigma = f(\overline{t\sigma}) \qquad \bullet\ fails(\mathcal{S})\sigma = fails(\mathcal{S}\sigma)$$
$$\bullet\ (\bigcup_{X\in\mathcal{S}_1)} \mathcal{S}_2)\sigma = \bigcup_{X\in\mathcal{S}_1\sigma} \mathcal{S}_2\sigma \qquad \bullet\ (\mathcal{S}_1 \cup \mathcal{S}_2)\sigma = \mathcal{S}_1\sigma \cup \mathcal{S}_2\sigma$$

Notice that produced variables are not affected by the substitution due to the condition $Dom(\sigma)\cap PV(\mathcal{S}) = \emptyset$; and we avoid capture of produced variables with the condition $Ran(\sigma)\cap PV(\mathcal{S}) = \emptyset$. We will also use (admissible) *set-substitutions* for set-expressions: given a set $D = \{s_1, ..., s_n\} \subseteq CTerm_{\perp,\mathsf{F}}$ we write $\mathcal{S}[Y/D]$ as a shorthand for the distribution $\mathcal{S}[Y/s_1] \cup ... \cup \mathcal{S}[Y/s_n]$. Extending this notation, we also write $\mathcal{S}[X_1/D_1, ..., X_n/D_n]$ (where $D_1, ..., D_n \subseteq CTerm_{\perp,\mathsf{F}}$) as a shorthand for $(...(\mathcal{S}[X_1/D_1])...)[X_n/D_n]$. In the following all the substitutions we use are admissible.

The ordering $\sqsubseteq$ defined in Section 2 for terms and sets of terms can be extended also to set expressions: the relation $\sqsubseteq_{SetExp_\perp}$ is the least reflexive and transitive relation satisfying:

1. $\{\perp\} \sqsubseteq_{SetExp_\perp} \mathcal{S}$, for any $\mathcal{S} \in SetExp_\perp$
2. $\{t\} \sqsubseteq_{SetExp_\perp} \{t'\}$, if $t \sqsubseteq t'$
3. $f(\overline{t}) \sqsubseteq_{SetExp_\perp} f(\overline{t}')$, if $f(\overline{t}) \sqsubseteq f(\overline{t}')$
4. $fails(\mathcal{S}) \sqsubseteq_{SetExp_\perp} fails(\mathcal{S}')$, if $\mathcal{S} \sqsubseteq_{SetExp_\perp} \mathcal{S}'$
5. $\bigcup_{X\in\mathcal{S}_1} \mathcal{S}_2 \sqsubseteq_{SetExp_\perp} \bigcup_{X\in\mathcal{S}'_1} \mathcal{S}'_2$, if $\mathcal{S}_1 \sqsubseteq_{SetExp_\perp} \mathcal{S}'_1$ and $\mathcal{S}_2 \sqsubseteq_{SetExp_\perp} \mathcal{S}'_2$
6. $\mathcal{S}_1 \cup ... \cup \mathcal{S}_n \sqsubseteq_{SetExp_\perp} \mathcal{S}'_1 \cup ... \cup \mathcal{S}'_m$, if for all $i \in \{1, ..., n\}$ there exists some $j \in \{1, ..., m\}$ such that $\mathcal{S}_i \sqsubseteq \mathcal{S}'_j$ and conversely, for all $j \in \{1, ..., m\}$ there exists some $i \in \{1, ..., n\}$ such that $\mathcal{S}_i \sqsubseteq \mathcal{S}'_j$

In the following the subindex of $\sqsubseteq_{SetExp_\perp}$ will be omitted.

3.2 Programs

A program is a set of rules of the form: $f(t_1, \ldots, t_n) \twoheadrightarrow \mathcal{S}$, where $f \in FS^n$, $t_i \in CTerm$, the tuple $(t_1, \ldots, t_n)$ is linear (each variable occurs only once), $\mathcal{S} \in SetExp$ and $FV(\mathcal{S}) \subseteq var((t_1, \ldots, t_n))$.

Notice that we allow F to appear in $\mathcal{S}$, but not in $t_1, \ldots, t_n$. This is not essential, and is done only for technical convenience. Notice also that known definitions which refer only to heads of rules, like that of *definitional tree* or *inductively sequential program* [2], can be applied also to our programs. Concretely, we are interested in the class of *C*omplete *I*nductively *S*equential Programs (*CIS*-programs for short), introduced in [2]. 'Complete' means that at every *case* distinction in the definitional tree of a function there is a branch for every constructor symbol from DC. By considering *CIS*-programs we ensure that, for any ground $t_1, \ldots, t_n \in CTerm$, exactly *one* program rule can be used to reduce a call $f(t_1, \ldots, t_n)$. And there is no loss of generality: in [3, 13] it

is shown how to convert programs into *overlapping* inductively sequential programs, where several rules might have the same head; as mentioned in [13], by using $\cup$ it straightforward to achieve inductive sequentiality, just by merging with $\cup$ several body rules; in [13] it is also shown how to translate 'classical' syntax for expressions in bodies into set-oriented syntax; finally, to achieve completeness we only need to add, for every 'missing' constructor in a *case* distinction of a definitional tree, a rule with $\{\mathsf{F}\}$ as body.

As an example, the *CIS*-program corresponding to the program of Sect. 1 is:

$$\begin{array}{cc} next(a) \twoheadrightarrow \{b,c\} & next(c) \twoheadrightarrow \{\mathsf{F}\} \\ next(b) \twoheadrightarrow \{c,d\} & next(d) \twoheadrightarrow \{\mathsf{F}\} \end{array}$$

$$path(X,Y) \twoheadrightarrow \bigcup_{A \in eq(X,Y)} \bigcup_{B \in \bigcup_{C \in next(X)} path(C,Y)} ifThenElse(A, true, B)$$

$$safe(X) \twoheadrightarrow fails(\bigcup_{A \in eq(X,d)} \bigcup_{B \in \bigcup_{C \in next(X)} path(C,d)} ifThenElse(A, true, B))$$

In practice, this syntax can be obtained by automating the translation.

3.3 Proof Calculus for Programs and Set-Expressions

Table 1 shows the *S*et *R*eduction *L*ogic that determines the semantics of set-expressions with respect to *CIS*-programs, by defining the provability relation $\mathcal{S} \triangleleft \mathcal{C}$ between set-expressions $\mathcal{S}$ and sets $\mathcal{C}$ of partial cterms. When $\mathcal{S} \triangleleft \mathcal{C}$ is provable we say that $\mathcal{C}$ is a *sufficient approximation set* (*SAS*) for $\mathcal{S}$.

Rules 1 to 4 are "classical" in *CRWL(F)* [8, 13, 12]. Rule 4 uses a c-instance of a program rule; the set of such c-instances is defined as: $[\mathcal{P}]_{\bot,\mathsf{F}} = \{R\theta \mid R = (f(\overline{t}) \twoheadrightarrow \mathcal{S}) \in \mathcal{P}, \theta \in CSubst_{\bot,\mathsf{F}}\}$. Notice that this c-instance is unique if it exists (due to the non-overlapping condition imposed to programs). If such c-instance does not exist then, by rule 5, the corresponding set-expression reduces to $\{\mathsf{F}\}$.

Rules 6 and 7 establish the meaning of the function $fails(\mathcal{S})$: we must reduce $\mathcal{S}$; if we achieve $\{\mathsf{F}\}$ as a *SAS* for it, this means that any attempt to reduce $\mathcal{S}$ effectively fails. On the other hand, if we obtain a *SAS* with a cterm of the form $c(...)$ or X, then there is a possible reduction of $\mathcal{S}$ to a cterm. This is a "constructive" way of proving failure. Moreover, the only *SAS*'s for $\mathcal{S}$ that do not provide enough information for reducing $fails(\mathcal{S})$ are $\{\bot\}$ or $\{\bot, \mathsf{F}\}$. Finally, rules 8 and 9 have a natural set-theoretic reading.

Given a program $\mathcal{P}$ and $\mathcal{S} \in SetExp$ we write $\mathcal{P} \vdash_{SRL} \mathcal{S} \triangleleft \mathcal{C}$ if the relation $\mathcal{S} \triangleleft \mathcal{C}$ is provable with respect to *SRL* and the program $\mathcal{P}$. The *denotation* of $\mathcal{S}$ is defined as $[\![\mathcal{S}]\!] = \{\mathcal{C} \mid \mathcal{S} \triangleleft \mathcal{C}\}$. Then the denotation of a set-expression is a set of sets of (possible partial) cterms.

It is easy to check that the symbol $\cup$ is associative and commutative, i.e., $[\![(\mathcal{S}_1 \cup \mathcal{S}_2) \cup \mathcal{S}_3]\!] = [\![\mathcal{S}_1 \cup (\mathcal{S}_2 \cup \mathcal{S}_3)]\!]$ and $[\![\mathcal{S}_1 \cup \mathcal{S}_2]\!] = [\![\mathcal{S}_2 \cup \mathcal{S}_1]\!]$. So, in the following we avoid unnecessary parentheses and consider the expression $\mathcal{S}_1 \cup ... \cup \mathcal{S}_n$ as equivalent to $\mathcal{S}_{i_1} \cup ... \cup \mathcal{S}_{i_n}$, where $(i_1, ..., i_n)$ is any permutation of $(1, ..., n)$. Moreover, $\mathcal{S}_1 \cup ... \cup \mathcal{S}_n \triangleleft \mathcal{C}$ iff $\mathcal{S}_1 \triangleleft \mathcal{C}_1, ..., \mathcal{S}_n \triangleleft \mathcal{C}_n$, where $\mathcal{C} = \mathcal{C}_1 \cup ... \cup \mathcal{C}_n$.

In order to prove the results of Section 4 we have needed to generalize some results of [13] and to prove new ones. In particular, Theorem 1 below is a key technical result about the *SRL* calculus.

Lemma 1 (Lifting). *Let $\sigma, \sigma' \in CSubst_{\bot,\mathsf{F}}$ with $\sigma \sqsubseteq \sigma'$. If $\mathcal{S}\sigma \lhd \mathcal{C}$ then there exist a proof in SRL for $\mathcal{S}\sigma' \lhd \mathcal{C}$ with the same length and structure.*

Theorem 1 (Refinement). *Let $\mathcal{S}_1, ..., \mathcal{S}_n, \mathcal{S}$ be set-expressions such that $\mathcal{S}_1 \sqsubseteq \mathcal{S}$, ..., $\mathcal{S}_n \sqsubseteq \mathcal{S}$. Then*

$$\mathcal{S}_1 \lhd \mathcal{C}_1, ..., \mathcal{S}_n \lhd \mathcal{C}_n \Rightarrow \mathcal{S} \lhd \mathcal{C}$$

for some $\mathcal{C}$ such that $\mathcal{C}_1 \sqsubseteq \mathcal{C}, ..., \mathcal{C}_n \sqsubseteq \mathcal{C}$.

The next two properties can be easily proved from the previous theorem.

Proposition 1 (Consistency). *If $\mathcal{S}$ and $\mathcal{S}'$ are consistent and $\mathcal{S} \lhd \mathcal{C}$, $\mathcal{S}' \lhd \mathcal{C}'$, then $\mathcal{C}$ and $\mathcal{C}'$ are also consistent.*

As a consequence of this property we have consistency of failure: if $\mathcal{S} \lhd \{\mathsf{F}\}$, then $[\![\mathcal{S}]\!] = \{\{\bot\}, \{\mathsf{F}\}\}$.

Proposition 2 (Monotonicity). *If $\mathcal{S} \sqsubseteq \mathcal{S}'$ and $\mathcal{S} \lhd \mathcal{C}$, then $\mathcal{S}' \lhd \mathcal{C}'$ for some $\mathcal{C}'$ with $\mathcal{C} \sqsubseteq \mathcal{C}'$.*

4 Operational Semantics

The narrowing relation to be defined in 4.2 makes an extensive use of a particular notion of *context*. The next section is devoted to formalize contexts in our framework and some other related aspects.

Table 1. Rules for *SRL*-provability

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

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

(4) $\dfrac{\mathcal{S} \lhd \mathcal{C}}{f(\overline{t}) \lhd \mathcal{C}}$ $(f(\overline{t}) \twoheadrightarrow \mathcal{S}) \in [\mathcal{P}]_{\bot,\mathsf{F}}$

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

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

(7) $\dfrac{\mathcal{S} \lhd \mathcal{C}}{fails(\mathcal{S}) \lhd \{false\}}$ there is some $t \in \mathcal{C}$, $t \neq \bot$, $t \neq \mathsf{F}$

(8) $\dfrac{\mathcal{S}_1 \lhd \mathcal{C}_1 \quad \mathcal{S}_2[X/\mathcal{C}_1] \lhd \mathcal{C}}{\bigcup_{X \in \mathcal{S}_1} \mathcal{S}_2 \lhd \mathcal{C}}$ (9) $\dfrac{\mathcal{S}_1 \lhd \mathcal{C}_1 \quad \mathcal{S}_2 \lhd \mathcal{C}_2}{\mathcal{S}_1 \cup \mathcal{S}_2 \lhd \mathcal{C}_1 \cup \mathcal{C}_2}$

4.1 Contexts

A *context* is a set-expression with some holes in the places of subexpressions. Syntactically a context is:

$$C ::= \mathcal{S} \mid [\,] \mid \mathit{fails}(C_1) \mid \bigcup_{X \in C_1} C_2 \mid C_1 \cup C_2$$

where $\mathcal{S}$ is a set-expression, and C_1 and C_2 are contexts. A *principal context* is:

$$C ::= f(\overline{t}) \mid [\,] \mid \mathit{fails}(\mathcal{S}) \mid \bigcup_{X \in \mathcal{S}} C_1 \mid C_1 \cup C_2$$

where $\mathcal{S}$ is a set-expression, and C_1 and C_2 are principal contexts. Intuitively, a principal context is a context without indexed holes, i.e., it has all its holes at the highest level.

Analogous to the case of set-expressions, we consider both *total (principal) contexts* when $\mathcal{S} \in \mathit{SetExp}$ and *partial (principal) contexts* if $\mathcal{S} \in \mathit{SetExp}_\perp$. From the definitions of produced variables and free variables of a set-expression it is direct to define *produced variables* $PV(C)$ and *free variables* $FV(C)$ *of a context* C. As in the case of set-expressions, for contexts we also impose *admissibility conditions*: contexts of the form $\bigcup_{X \in C_1} C_2$ must satisfy $X \notin var(C_1) \cup PV(C_2)$ and $PV(C_1) \cap PV(C_2) = \emptyset$; and contexts of the form $C_1 \cup C_2$ must satisfy $PV(C_1) \cap FV(C_2) = \emptyset$ and $PV(C_2) \cap FV(C_1) = \emptyset$. How to apply substitutions to contexts is defined in a natural way.

The *arity* of a context C, written $|C|$, is the number of its holes. A context C of arity n can be understood as a function that takes n contexts $C_1, ..., C_n$ as arguments and returns another context resulting of fulfilling the holes with the contexts of the arguments. Formally, the *application of a context* C to the tuple of arguments $C_1, ..., C_n$, notated as $C\ [C_1, ..., C_n]$, is defined as:

- $[\,]\ [C] = C$
- $\mathit{fails}(C)\ [C_1, ..., C_n] = \mathit{fails}(C\ [C_1, ..., C_n])$
- $(\bigcup_{X \in C} C')\ [C_1, ..., C_n, C'_1, ..., C'_m] = \bigcup_{X \in C\ [C_1, ..., C_n]} C'\ [C'_1, ..., C'_n]$, where $|C| = n$ and $|C'| = m$
- $(C \cup C')\ [C_1, ..., C_n, C'_1, ..., C'_m] = (C\ [C_1, ..., C_n]) \cup (C'\ [C'_1, ..., C'_m])$, where $|C| = n$ and $|C'| = m$

In the rest of the paper all context applications are done in such a way that admissibility conditions are satisfied by the resulting context.

Notice that the application of a context C of arity n to a tuple of contexts $C_1, ..., C_n$ of arities $m_1, ..., m_n$ results in another context of arity $m_1 + ... + m_n$. And of course, if we apply this context C to a tuple of set-expressions (contexts with arity 0) the resulting context has arity 0, that is, a set-expression. With respect to substitutions its is easy to check that for any $\sigma \in \mathit{CSubst}_{\perp,\mathsf{F}}$ we have $(C\ [\mathcal{S}_1, ..., \mathcal{S}_n])\sigma = C\sigma\ [\mathcal{S}_1\sigma, ..., \mathcal{S}_n\sigma]$.

Given a set-expression $\mathcal{S}$, it is possible to determine a principal context $C_\mathcal{S}$ and a tuple of cterms $t_1, ..., t_n$ such that $\mathcal{S} = C_\mathcal{S}\ [\{t_1\}, ..., \{t_n\}]$. We can see it reasoning on the structure of $\mathcal{S}$:

- if $\mathcal{S} = \{t\}$, then clearly $\mathcal{S} = [\,]\ [\{t\}]$
- if $\mathcal{S} = f(\overline{t})$ or $\mathcal{S} = \mathit{fails}(\mathcal{S}')$, then $C_\mathcal{S} = \mathcal{S}$ (a 0-arity principal context) and the tuple of arguments is empty.

- if $\mathcal{S} = \bigcup_{X \in \mathcal{S}_1} \mathcal{S}_2$, we can assume, inductively, that $\mathcal{S}_2$ is of the form $C_{\mathcal{S}_2}\ [\{t_1\}, ..., \{t_n\}]$ (where $C_{\mathcal{S}_2}$ is a principal context). Then, clearly $\mathcal{S} = (\bigcup_{X \in \mathcal{S}_1} C_{\mathcal{S}_2})\ [\{t_1\}, ..., \{t_n\}]$, and $(\bigcup_{X \in \mathcal{S}_1} C_{\mathcal{S}_2})$ is a principal context.
- if $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$, then if $\mathcal{S}_1 = C_{\mathcal{S}_1}\ [\{t_1\}, ..., \{t_n\}]$ and $\mathcal{S}_2 = C_{\mathcal{S}_2}\ [\{s_1\}, ..., \{s_m\}]$, we have $\mathcal{S} = (C_{\mathcal{S}_1} \cup C_{\mathcal{S}_2})\ [\{t_1\}, ..., \{t_n\}, \{s_1\}, ..., \{s_m\}]$.

We say that $C_{\mathcal{S}}\ [\{t_1\}, ..., \{t_n\}]$ is the *principal contextual form* (in short p.c.f) of $\mathcal{S}$. In the following, we frequently refer to a set-expression by its p.c.f. Moreover, as an abuse of notation we will omit the braces '{', '}' and write $C_{\mathcal{S}}\ [t_1, ..., t_n]$ or simply $C_{\mathcal{S}}\ [\overline{t}]$.

Notice that the equality $\mathcal{S} = C_{\mathcal{S}}[t_1, ..., t_n]$, viewed as a context application, is a trivial identity. The important point of such format is that the cterms $t_1, ..., t_n$ reflect the skeleton or constructed part of the set-expression. With this idea, if $C_{\mathcal{S}}[t_1, ..., t_n]$ is the p.c.f. for $\mathcal{S}$, the *information set* $\mathcal{S}^*$ of $\mathcal{S}$ is defined as:

- $\{\bot\}$ if $n = 0$;
- $\{t_1\tau, ..., t_n\tau\}$ if $n > 0$, where $X\tau = \bot$ if $X \in PV(\mathcal{S})$ and $X\tau = X$ otherwise

For example, consider the set-expression introduced in Sect. 3

$$\mathcal{S} = \bigcup_{A \in \bigcup_{B \in f(X,Y)}\{B\}} \{c(A, X)\} \cup \bigcup_{C \in \{X\}} f(C, Y)$$

The context $C_{\mathcal{S}}$ of the p.c.f. has only a hole in the place of $\{c(A, X)\}$, which is the only argument of the p.c.f., that is:

$$(\bigcup_{A \in \bigcup_{B \in f(X,Y)}\{B\}}\ [\,] \cup \bigcup_{C \in \{X\}} f(C, Y))\ [c(A, X)]$$

In this case $\mathcal{S}^* = \{c(\bot, X)\}$ (obtained by replacing the variable A by $\bot$).

Proposition 3. *The information set of a set-expression is a SAS for $\mathcal{S}$, i.e., $\mathcal{S} \triangleleft \mathcal{S}^*$. Moreover, this SAS can be obtained by using rules 1, 2, 3, 8 and 9 of SRL (in other words, no function rule is needed to obtain $\mathcal{S}^*$).*

4.2 A Narrowing Relation for Set-Expressions

The rewriting logic *SRL* showed in Sect. 3 fixes the denotational semantics of set-expressions. Now we are interested in a operational mechanism for narrowing set-expressions into simplified or normal forms, while finding appropriate values for the variables of the expressions.

The syntactic structure of a *normal form* is: $\{t_1\} \cup ... \cup \{t_n\}$, with $t_1, ..., t_n \in CTerm_{\mathsf{F}}$. According to this, $\{\mathsf{F}\}$ is a normal form itself. Notice also that a normal form cannot contain any function symbol or indexed union. On the other hand, the undefined value $\bot$ has not any sense in the operational mechanism that deals only with total set-expressions.

One-Narrowing-Step Relation: Given a program $\mathcal{P}$, two set-expressions $\mathcal{S}, \mathcal{S}' \in SetExp$, $\theta \in CSubst_{\mathsf{F}}$ and $\Gamma \subseteq \mathcal{V}$, a narrowing step is expressed as: $\mathcal{S} \overset{\Gamma}{\underset{\theta}{\rightsquigarrow}} \mathcal{S}'$ where the relation $\mathcal{S} \overset{\Gamma}{\underset{\theta}{\rightsquigarrow}} \mathcal{S}'$ is defined in the Table 2. The set Γ is the *set of protected variables*: produced variables of the external scope are stored into Γ in order to avoid to apply substitutions for them. For narrowing

Table 2. *SNarr* relation

Cntx	$C\ [\mathcal{S}] \overset{\Gamma}{\underset{\theta}{\rightsquigarrow}} C\theta\ [\mathcal{S}']$	if $\mathcal{S} \overset{\Gamma \cup PV(C)}{\underset{\theta}{\rightsquigarrow}} \mathcal{S}'$
Nrrw$_1$	$f(\overline{t}) \overset{\Gamma}{\underset{\theta\vert_{var(\overline{t})}}{\rightsquigarrow}} \mathcal{S}\theta$	if $(f(\overline{s}) \twoheadrightarrow \mathcal{S}) \in \mathcal{P}$, $\theta \in CSubst_{\mathsf{F}}$ is a m.g.u. for $\overline{s}$ and $\overline{t}$ with $Dom(\theta) \cap \Gamma = \emptyset$
Nrrw$_2$	$f(\overline{t}) \overset{\Gamma}{\underset{\epsilon}{\rightsquigarrow}} \{\mathsf{F}\}$	if for every rule $(f(\overline{s}) \twoheadrightarrow \mathcal{S}_1) \in \mathcal{P}$, $\overline{s}$ and $\overline{t}$ have a $DC \cup \{\mathsf{F}\}$-clash
Fail$_1$	$fails(\mathcal{S}) \overset{\Gamma}{\underset{\epsilon}{\rightsquigarrow}} \{true\}$	if $\mathcal{S}^* = \{\mathsf{F}\}$
Fail$_2$	$fails(\mathcal{S}) \overset{\Gamma}{\underset{\epsilon}{\rightsquigarrow}} \{false\}$	if $\exists t \in \mathcal{S}^*\ t \neq \bot, t \neq \mathsf{F}$
Flat	$\bigcup_{X \in \bigcup_{Y \in \mathcal{S}_1} \mathcal{S}_2} \mathcal{S}_3 \overset{\Gamma}{\underset{\epsilon}{\rightsquigarrow}} \bigcup_{Y \in \mathcal{S}_1} \bigcup_{X \in \mathcal{S}_2} \mathcal{S}_3$	
Dist	$\bigcup_{X \in \mathcal{S}_1 \cup \mathcal{S}_2} \mathcal{S}_3 \overset{\Gamma}{\underset{\epsilon}{\rightsquigarrow}} \bigcup_{X \in \mathcal{S}_1} \mathcal{S}_3 \cup \bigcup_{X \in \mathcal{S}_2} \mathcal{S}_3$	
Bind	$\bigcup_{X \in \{t\}} \mathcal{S} \overset{\Gamma}{\underset{\epsilon}{\rightsquigarrow}} \mathcal{S}[X/t]$	
Elim	$\bigcup_{X \in \mathcal{S}'} \mathcal{S} \overset{\Gamma}{\underset{\epsilon}{\rightsquigarrow}} \mathcal{S}$	if $X \notin FV(\mathcal{S})$

a set-expression $\mathcal{S}$ we must take Γ such that $\Gamma \cap FV(\mathcal{S}) = \emptyset$; in fact, the simplest choice is $\Gamma = \emptyset$. The calculus itself will protect the appropriate variables in derivations. In the following we use *SNarr* as a name for this narrowing relation, and refer to it as *set-narrowing*.

Some comments about the rules in Table 2:

- **Cntx** performs a sub-derivation on a sub-set-expression and then replaces the original by the resulting one. Notice that it ensures the protection of produced variables of the context by adding them to Γ.
- The rule **Nrrw**$_1$ narrows a function call $f(\overline{t})$ using a rule of the program. The m.g.u. θ used for the parameter passing is not allowed to affect to protected variables due to the condition $Dom(\theta) \cap \Gamma = \emptyset$. It is also clear that $Ran(\theta) \cap \Gamma = \emptyset$ because the variables of the rule are fresh variables. Notice also that, for technical convenience, the substitution θ is projected over the variables of the narrowed expression. Notice finally that this is *the* rule which really binds variables, since **Cntx** is merely a contextual rule and the rest produce an empty substitution.
- **Nrrw**$_2$ operates only in the case that **Nrrw**$_1$ cannot find an applicable rule of the program, i.e., all of them have a clash. As our programs are ***DC*-complete** this situation is only possible if the call $f(\overline{t})$ has F at a position in which the heads of program rules have a constructor symbol $c \in DC$ (the transformation to *CIS-programs* does not introduce F in heads).
- Rules **Fail**$_1$ and **Fail**$_2$ are direct counterpart of rules 6 and 7 of *SRL*.

- Rules **Flat**, **Dist**, **Bind** and **Elim** directly reflect some properties of sets in the mathematical sense. From an intuitive point of view, they all reduce the structural complexity of the set-expression.

The closure $\mathcal{S} \overset{\Gamma}{\underset{\theta}{\rightsquigarrow}}{}^{*} \mathcal{S}'$ is defined in the usual way:

$$\mathcal{S} = \mathcal{S}_0 \overset{\Gamma}{\underset{\theta_1}{\rightsquigarrow}} \mathcal{S}_1 \overset{\Gamma}{\underset{\theta_2}{\rightsquigarrow}} \ldots \overset{\Gamma}{\underset{\theta_n}{\rightsquigarrow}} \mathcal{S}_n = \mathcal{S}', \quad \text{with } \theta = \theta_1\theta_2\ldots\theta_n$$

It is easy to see that at any step $\mathcal{S}_i\theta_i = \mathcal{S}_i$ and also $\mathcal{S}'\theta = \mathcal{S}'$.

Example Consider the *CIS*-program of graphs of Sect. 3. The node c is safe according to definitions, so *SNarr* must be able to narrow the expression $safe(X)$ to *true* with the substitution $X = c$. Table 3 shows the steps of such derivation; we underline the redex at each step and annotate the rule of *SNarr* applied. The applications of **Cntx** are omitted for simplicity. We assume the function *ifThenElse* defined by the natural rules, and we shorten *ifThenElse* by *iTe*. We recall that the function *eq* is defined by the rules $eq(a, a) \twoheadrightarrow \{true\}$, $eq(a, b) \twoheadrightarrow \{false\}$ and so on. The answer substitution $[X/c]$ is found in the third step by $\mathbf{Nrrw}_1$ applied to the expression $eq(X, d)$.

This derivation has been performed by a little prototype that implements the relation *SNarr* with an appropriate criterion for selecting the redex. In fact this prototype provides four possible reductions for the expression $safe(X)$: $false$ with $[X/a]$; $false$ with $[X/b]$; $true$ with $[X/c]$ and $false$ with $[X/d]$.

4.3 Correctness and Completeness of *Snarr*

A desirable property is the preservation of semantics under context application: consider two set-expressions $\mathcal{S}$ and $\mathcal{S}'$ with $[\![\mathcal{S}]\!] = [\![\mathcal{S}']\!]$ and a 1-arity context C; then, we would like $[\![C\ [\mathcal{S}]]\!] = [\![C\ [\mathcal{S}']]\!]$. But this is not true in general. For example, assume the rules $f(z) \twoheadrightarrow \{z\}$ and $g(z) \twoheadrightarrow \{s(z)\}$. Then, for the set-expressions $f(X)$ and $g(X)$, $[\![g(X)]\!] = [\![f(X)]\!] = \{\bot\}$. But if we consider $C = \bigcup_{X\in\{z\}}[\ ]$ then $C\ [f(X)] = \bigcup_{X\in\{z\}} f(X)$ and $C\ [g(X)] = \bigcup_{X\in\{z\}} g(X)$, whose denotations are $\{\{\bot\}, \{z\}\}$ and $\{\{\bot\}, \{s(z)\}\}$ resp. What is true is:

Lemma 2. *Let $\mathcal{S}$ and $\mathcal{S}'$ be set-expressions such that $[\![\mathcal{S}\sigma]\!] = [\![\mathcal{S}'\sigma]\!]$ for any $\sigma \in CSubst_{\bot,\mathcal{F}}$ admissible for $\mathcal{S}$ and $\mathcal{S}'$. Then $[\![(C\ [\mathcal{S}])\sigma']\!] = [\![(C\ [\mathcal{S}'])\sigma']\!]$, for any context C of arity 1 and any $\sigma' \in CSubst_{\bot,\mathcal{F}}$ admissible for $C\ [\mathcal{S}]$ and $C\ [\mathcal{S}']$.*

Notice that in the example above, the hypothesis $[\![g(X)\theta]\!] = [\![f(X)\theta]\!]$ does not hold, for instance taking $\theta = [X/z]$. This lemma is used for proving correctness:

Theorem 2 (Correctness of *SNarr*). *Let $\mathcal{S}$, $\mathcal{S}' \in SetExp$, $\theta \in CSubst_{\mathcal{F}}$, $\Gamma \subseteq \mathcal{V}$ any set of variables, and $\sigma \in CSubst_{\bot,\mathcal{F}}$ any admissible substitution for $\mathcal{S}\theta$ and $\mathcal{S}'$. Then: $\mathcal{S} \overset{\Gamma}{\underset{\theta}{\rightsquigarrow}}{}^{*} \mathcal{S}' \Rightarrow [\![\mathcal{S}\theta\sigma]\!] = [\![\mathcal{S}'\sigma]\!]$. In particular, taking $\sigma = \epsilon$ we have: $\mathcal{S} \overset{\Gamma}{\underset{\theta}{\rightsquigarrow}}{}^{*} \mathcal{S}' \Rightarrow [\![\mathcal{S}\theta]\!] = [\![\mathcal{S}']\!]$*

Table 3. A narrowing derivation for $safe(X)$

$\underline{safe(X)}$		$(Nrrw_1)$
$fails(\underline{path(X,d)})$		$(Nrrw_1)$
$fails(\bigcup_{A \in \underline{eq(X,d)}} \bigcup_{B \in \bigcup_{C \in next(X)} path(C,d)} iTe(A, true, B))$	X/c	$(Nrrw_1)$
$fails(\underline{\bigcup_{A \in \{false\}} \bigcup_{B \in \bigcup_{C \in next(c)} path(C,d)} iTe(A, true, B)})$		$(Bind)$
$fails(\bigcup_{B \in \bigcup_{C \in next(c)} path(C,d)} \underline{iTe(false, true, B)})$		$(Nrrw_1)$
$fails(\underline{\bigcup_{B \in \bigcup_{C \in next(c)} path(C,d)} \{B\}})$		$(Flat)$
$fails(\bigcup_{C \in next(c)} \bigcup_{B \in \underline{path(C,d)}} \{B\})$		$(Nrrw_1)$
$fails(\bigcup_{C \in next(c)} \underline{\bigcup_{B \in \bigcup_{D \in eq(C,d)} \bigcup_{E \in \bigcup_{F \in next(C)} path(F,d)} iTe(D,true,E)} \{B\}})$		$(Flat)$
$fails(\bigcup_{C \in next(c)} \bigcup_{D \in eq(C,d)} \underline{\bigcup_{B \in \bigcup_{E \in \bigcup_{F \in next(C)} path(F,d)} iTe(D,true,E)} \{B\}})$		$(Flat)$
$fails(\bigcup_{C \in \underline{next(c)}} \bigcup_{D \in eq(C,d)} \bigcup_{E \in \bigcup_{F \in next(C)} path(F,d)} \bigcup_{B \in iTe(D,true,E)} \{B\})$		$(Nrrw_2)$
$fails(\underline{\bigcup_{C \in \{\mathsf{F}\}} \bigcup_{D \in eq(C,d)} \bigcup_{E \in \bigcup_{F \in next(C)} path(F,d)} \bigcup_{B \in iTe(D,true,E)} \{B\}})$		$(Bind)$
$fails(\bigcup_{D \in \underline{eq(\mathsf{F},d)}} \bigcup_{E \in \bigcup_{F \in next(\mathsf{F})} path(F,d)} \bigcup_{B \in iTe(D,true,E)} \{B\})$		$(Nrrw_2)$
$fails(\underline{\bigcup_{D \in \{\mathsf{F}\}} \bigcup_{E \in \bigcup_{F \in next(\mathsf{F})} path(F,d)} \bigcup_{B \in iTe(D,true,E)} \{B\}})$		$(Bind)$
$fails(\bigcup_{E \in \bigcup_{F \in next(\mathsf{F})} path(F,d)} \bigcup_{B \in \underline{iTe(\mathsf{F},true,E)}} \{B\})$		$(Nrrw_2)$
$fails(\underline{\bigcup_{E \in \bigcup_{F \in next(\mathsf{F})} path(F,d)} \bigcup_{B \in \{\mathsf{F}\}} \{B\}})$		$(Elim)$
$fails(\underline{\bigcup_{B \in \{\mathsf{F}\}} \{B\}})$		$(Bind)$
$\underline{fails(\{\mathsf{F}\})}$		$(Fail_1)$
$\{true\}$		

We split completeness in two simpler lemmas. First, the relation *SNarr* can refine the information of any *SAS* obtained by *SRL*.

Lemma 3 (Completeness I). *If* $\mathcal{P} \vdash_{SRL} \mathcal{S} \lhd \mathcal{C}$ *then for any* Γ *with* $\Gamma \cap FV(\mathcal{S}) = \emptyset$ *we can derive* $\mathcal{S} \overset{\Gamma}{\underset{\epsilon}{\rightsquigarrow^*}} \mathcal{S}'$ *such that* $\mathcal{C} \sqsubseteq (\mathcal{S}')^*$

Here we see the relation *SNarr* as a pure rewriting relation: the answer substitution is ϵ. Lemma 4 shows that any substitution θ making reducible a set-expression $\mathcal{S}$ is captured or generalized by a narrowing derivation from $\mathcal{S}$. For example, for the rule $f(z) \twoheadrightarrow \{s(z)\}$, *SNarr* can derive $f(z) \overset{\Gamma}{\underset{\epsilon}{\rightsquigarrow^*}} \{s(z)\}$. Then, there is a derivation $f(X) \overset{\Gamma}{\underset{[X/z]}{\rightsquigarrow^*}} \{s(z)\}$, i.e., the value z for X can be found.

Lemma 4 (Completeness II). *Let $\mathcal{S} \in SetExp$, $\theta \in CSubst$ and Γ such that $\Gamma \cap PV(\mathcal{S}) = \emptyset$. If $\mathcal{S}\theta \overset{\Gamma}{\underset{\epsilon}{\rightsquigarrow}}{}^{*} \mathcal{S}'$ then there exists $\theta', \mu \in CSubst$, with $\theta = \theta'\mu$, such that $\mathcal{S} \overset{\Gamma}{\underset{\theta'}{\rightsquigarrow}}{}^{*} \mathcal{S}''$ with $\mathcal{S}' = \mathcal{S}''\mu$.*

Theorem 3 (Completeness). *If $\mathcal{P} \vdash_{SRL} \mathcal{S}\theta \triangleleft \mathcal{C}$ and $\Gamma \cap FV(\mathcal{S}) = \emptyset$, then there exists a derivation $\mathcal{S} \overset{\Gamma}{\underset{\theta'}{\rightsquigarrow}}{}^{*} \mathcal{S}'$ such that, for some μ, $\theta = \theta'\mu$ (i.e. θ' is more general than θ) and $\mathcal{C} \sqsubseteq (\mathcal{S}'\mu)^{*}$.*

Corollary 1. *If $\mathcal{P} \vdash_{SRL} \mathcal{S}\theta \triangleleft \{F\}$, then $\mathcal{S} \overset{\emptyset}{\underset{\theta'}{\rightsquigarrow}}{}^{*} \{F\}$ for some θ' more general than θ.*

This result shows that we have achieved our goal of devising an operational mechanism for failure, which is *constructive*, in the sense that in presence of variables is able to find appropriate substitutions for them.

5 Conclusions and Future Work

We have defined a narrowing relation for functional logic programs which can make use of failure as programming construct, by means of a function *fails(e)*, which is a computable approximation to the property ' *e cannot be reduced to head normal form*'. Programs are written in a set-oriented syntax where expressions can use unions and indexed unions. Thus the syntax directly reflects non-determinism of functions and call-time choice semantics in a suitable way as to facilitate the definition of narrowing of set-expressions.

The set-narrowing relation serves to the purpose of computing failures, since the failure of an expression e corresponds to the fact that the expression can be narrowed to the set $\{\mathsf{F}\}$, where F is a constant introduced to represent failure. An important feature of our notion of narrowing is that it can operate to compute failures even in presence of variables, for which appropriate bindings are obtained. Using a frequent terminology in the context of logic programming and negation, our narrowing relation realizes *constructive failure* for functional logic programs. Nothing similar can be found in existing *FLP* systems like *Curry* or $\mathcal{TOY}$.

The definition of set-narrowing is quite amenable for implementation, and we have indeed implemented a first prototype for it, with which the examples in the paper have been executed.

In this paper we have not addressed the issue of *strategies* for set-narrowing. Nevertheless, for the implementation we have used some kind of 'demand driven strategy' close to the usual one in existing systems. It this sense it is interesting to remark that, in contrast to other notions of narrowing proposed for non-determinism with call-time choice [7, 8], which has been criticized [3] as being too far from real computations, our definition of set-narrowing seems better suited

to the adoption of strategies. Notice that, in particular, the rule $\mathbf{Nrrw}_1$ performs exactly a narrowing step in the classical sense, just with some conditions imposed about what variables can be bound.

The theoretical investigation of set-narrowing strategies, as well as the integration of our implementation in the system $\mathcal{TOY}$ are the main subjects of future work.

References

[1] M. Abengózar-Carneros et al. $\mathcal{TOY}$: a multiparadigm declarative language, version 2.0. Technical report, Dep. SIP, UCM Madrid, January 2001. 212, 213

[2] S. Antoy. Definitional trees. In *Proc. ALP'92*, pages 143–157. Springer LNCS 632, 1992. 214, 217

[3] S. Antoy. Constructor-based conditional narrowing. In *Proc. PPDP'01*, pages 199–206. ACM Press, 2001. 217, 225

[4] K. R. Apt and R. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19&20:9–71, 1994. 212

[5] D. Chan. Constructive negation based on the completed database. In *Proc. ICSLP'88*, pages 111–125, 1988. 214

[6] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978. 212

[7] J. C. González-Moreno, T. Hortalá-González, F. J. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. ESOP'96*, pages 156–172. Springer LNCS 1058, 1996. 213, 225

[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. *Journal of Logic Programming*, 40(1):47–87, 1999. 212, 213, 218, 225

[9] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994. 212

[10] M. Hanus (ed.). Curry: An integrated functional logic language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry/report.html`, February 2000. 212, 213

[11] F. J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. RTA'99*, Springer LNCS 1631, pages 244–247, 1999. 212, 213

[12] F. J. López-Fraguas and J. Sánchez-Hernández. Proving failure in functional logic programs. In *Proc. CL'00*, Springer LNAI 1861, pages 179–193, 2000. 213, 214, 218

[13] F. J. López-Fraguas and J. Sánchez-Hernández. Functional logic programming with failure: A set-oriented view. In *Proc. LPAR'01*, Springer LNAI 2250, pages 455–469, 2001. 213, 214, 217, 218

[14] F. J. López-Fraguas and J. Sánchez-Hernández. Narrowing failure in functional logic programming (long version). Available at `http://www.ucm.es/info/dsip/jaime/flopsExt.ps`, 2002. 215

[15] F. J. López-Fraguas and J. Sánchez-Hernández. A proof theoretic approach to failure in functional logic programming. Draft available at `http://www.ucm.es/info/dsip/jaime/tplp.ps`, 2002. 213, 214

[16] J. J. Moreno-Navarro. Default rules: An extension of constructive negation for narrowing-based languages. In *Proc. ICLP'94*, pages 535–549. MIT Press, 1994. 214

[17] 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. 214, 216

[18] J. C. Reynolds. *Theories of Programing Languages*. Cambridge Univ. Press, 1998. 216

[19] P. J. Stuckey. Constructive negation for constraint logic programming. In *Proc. LICS'91*, pages 328–339, 1991. 214

[20] P. J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118:12–33, 1995. 214