Functional Logic Programming with Failure and Built-in Equality⋆

F. J. López-Fraguas and J. Sánchez-Hernández

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

Abstract. Constructive failure has been proposed recently as a programming construct useful for functional logic programming, playing a role similar to that of constructive negation in logic programming. On the other hand, almost any functional logic program requires the use of some kind of equality test between expressions. We face in this work in a rigorous way the interaction of failure and equality (even for non-ground expressions), which is a non trivial issue, requiring in particular the use of disequality conditions at the level of the operational mechanism of constructive failure. As an interesting side product, we develop a novel treatment of equality and disequality in functional logic programming, by giving them a functional status, which is better suited for practice than previous proposals.

1 Introduction

Functional logic programming (see [10]) attempts to amalgamate the best features of functional and logic languages. Some works [19, 20, 15, 17, 16, 18] have addressed the subject of *failure of reduction* as a useful programming construct extending naturally to the functional logic setting the notion of negation as failure [7, 4], of great importance in logic programming.

On the other hand, almost any real program needs to use some kind of equality test between expressions. It is known [5, 14, 1] that a good treatment of equality in presence of non-ground expressions requires the use of disequality constraints. In this paper we address the combination of both issues: failure and equality.

Let us clarify our contribution with respect to previous works: in [15, 17] we extended *CRWL* [8, 9] –a well known semantic framework for functional logic programming– to obtain *CRWLF*, a proof calculus given as a semantic basis for failure in functional logic programming; those papers considered equality and disequality constraints, which were lost in [16], where *CRWLF* was reformulated by using a set oriented syntax to reflect more properly the set-valued nature of expressions in presence of non-deterministic functions typical of *CRWL* and *CRWLF*. The purely semantic, non executable approach of [16] was completed in [18] with a narrowing-like procedure able to operate with failure in presence

---

of non-ground expressions (we call this *constructive failure*, due to its similarity with *constructive negation* [23]); equality and disequality as specific features are again absent in [18].

The combination of constructive failure with a good treatment of equality is the **main contribution** of this paper. Equality is realized by means of a three-valued function (==) returning as possible values *true*, *false* or F, (a specific constructor to express the result of a failed computation). As we argue at the end of the next section, this is better suited to practice than to have independent equality and disequality constraints like in [5, 14, 15, 17].

The organization of the paper is as follows: the next section motivates the interest of our work; Section 3 summarizes the basic concepts about the *Set Reduction Logic* of [16] and extends it with the inclusion of the equality function ==; Section 4 is devoted to the operational mechanism, that extends the relation *SNarr* presented in [18] with the inclusion of ==. This extension requires a specialized calculus for ==, the manipulation of disequalities and some other technical notions. We give correctness and completeness results, but proofs are omitted due to lack of space. Finally, in Section 5 we present some conclusions and the future lines of our work.

2 Failure & equality: motivation

We give here a small but not artificial example recalling the interest of constructive failure and showing that the operational treatment of equality of [18] has some drawbacks. We use in this section the concrete syntax of $\mathcal{TOY}$ [13, 1]. The functions `if _ then _` and `if _ then _ else _` are defined as usual. The results indicated in the example correspond, except for some pretty-printing, to the real results obtained with a small prototype implementing the ideas of this work.

The purpose of a computation is to evaluate a possibly non-ground expression, obtaining as answer a pair (S, C) where S is a set of values (constructor terms) and C expresses a condition for the variables in the expression. To consider sets of values instead of single values is the natural thing when dealing with failure and non-deterministic functions, as shown in [15, 17]. When S is a singleton, we sometimes skip the set braces. The condition C, in 'ordinary' functional logic programming, would be simply a substitution. In this paper, as we will see soon, C can have in addition some disequality conditions.

Example: Failure can be used to convert predicates (i.e. `true`-valued functions) into two-valued boolean functions giving values `true` or `false`. The latter are more useful, while the former are simpler to define. As an example, consider the following definition of a predicate `prefix`:

```
prefix [] Ys = true
prefix [X|Xs] [Y|Ys] = if (X==Y) then prefix Xs Ys
```

Here, `==` must be understood as *strict equality* (see e.g. [10]). To obtain from `prefix` a two-valued boolean function `fprefix` is easy using failure:

```
fprefix Xs Ys = if fails (prefix Xs Ys) then false else true
```

The function `fails` is intended to return the value `true` if its argument fails (in finite time) to be reduced to head normal form (variable or constructor rooted expression), and the value `false` if a reduction to hnf exists. Furthermore, failure should be constructive, which means that `fails` should operate soundly in presence of variables[1].

To see what happens with equality, assume first that `==` is seen as any other function, an therefore defined by rules, like in [18] . Assume also, for simplicity, that we are dealing with lists of natural numbers represented by the constructors `0` and `suc`. The rules for `==` would be:

```
      0    == 0 = true          0     == (suc Y) = false
(suc X) == 0 = false       (suc X) == (suc Y) = X == Y
```

With these rules an expression like `fprefix Xs [Y]` produces infinitely many answers binding `Xs` to lists of natural numbers and `Y` to natural numbers.

In contrast, in this paper `==` has a specific evaluation mechanism involving disequality conditions which can be a part of answers. The virtue of disequalities is that one single disequality can encompass the same information as many substitutions. For `fprefix Xs [Y]` only four answers are obtained: `(true,{Xs=[]})`; `(false,{Xs=[Y,Z|Zs]})`; `(true,{Xs=[Y]})`; `(false,{Xs=[Z|Zs],Y/=Z})`. Notice the disequality condition in the last answer.

Some final remarks about our treatment of equality: in [5, 14] we considered equality and disequality as two independent constraints, to be used in conditional rules. One constraint `e == e'` or `e /= e'` might succeed or fail. To use a two-valued boolean equality function, say `eq`, requires to define it by means of two conditional rules: `X eq Y = true <== X==Y;  X eq Y = false <== X /= Y`

But this has a penalty in efficiency: to evaluate an equality `e eq e'` when `e,e'` are in fact unequal (thus `e eq e'` should return `false`), the first rule tries to solve the equality, which requires to evaluate `e` and `e'` (and that might be costly); the failure of the first rule will cause backtracking to the second rule, where a new re-evaluation of `e,e'` will be done. The actual implementation of $\mathcal{TOY}$ replaces the above naive treatment of the function `eq` by a more efficient one where one equality is evaluated as far as possible without guessing in advance the value to obtain; this is done in $\mathcal{TOY}$ in a very ad-hoc way lacking any formal justification with respect to the intended semantics of equality. Here we fill the gap in a rigorous way and in the more complex setting of constructive failure, where a third value, `F`, must be considered for `==`.

3   Set Reduction Logic with Failure and ==

In this section we summarize the basic concepts of [16] about the set oriented syntax, set-expressions and the class of *CIS*-programs that we use to build the

---

[1] From the point of view of implementation, failure is not difficult for ground expressions. For instance, in the system Curry [11] one could use encapsulated search [12]. The problems come with constructive failure, as in the case of logic programs.

Set Reduction Logic (SRL for short) as a semantic basis for constructive failure. In order to extend this proof calculus to deal with equality, we also use some syntactical notions about strict equality, disequality and their failure for constructed terms introduced in [17]. We remark that for the theoretical presentation we use first order notation.

3.1 Technical Preliminaries

We assume a signature $\Sigma = DC_\Sigma \cup FS_\Sigma \cup \{fails, ==\}$, where $DC_\Sigma = \bigcup_{n \in \mathbb{N}} DC_\Sigma^n$ is a set of *constructor* symbols containing at least the usual boolean ones *true* and *false*, $FS_\Sigma = \bigcup_{n \in \mathbb{N}} FS_\Sigma^n$ is a set of *function* symbols, all of them with associated arity and such that $DC_\Sigma \cap FS_\Sigma = \emptyset$. The functions *fails* (with arity 1) and $==$ (infix and with arity 2) do not belong to $DC \cup FS$, and will be defined by specific rules in the SRL-calculus.

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 makes 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 (nullary 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 the result of a failed reduction to hnf. 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 the 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.

The sets of substitutions $CSubst, CSubst_\mathsf{F}$ and $CSubst_{\bot,\mathsf{F}}$ are defined as mappings from $\mathcal{V}$ into $CTerm, CTerm_\mathsf{F}$ and $CTerm_{\bot,\mathsf{F}}$ respectively. We will write θ, σ, μ for general substitutions and ϵ for the identity substitution. The notation $\theta\sigma$ stands for composition of substitutions. All the considered substitutions are idempotent ($\theta\theta = \theta$). We also 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'. Extending $\sqsubseteq$ to sets of terms results in the *Egli-Milner* preordering (see e.g. [21]): given $D, D' \subseteq CTerm_{\perp,\mathsf{F}}$, $D \sqsubseteq D'$ 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 $t \sqsubseteq t'$.

Next we define the relations $\downarrow$, $\uparrow$, $\not\downarrow$ and $\not\uparrow$, expressing, at the level of cterms, strict equality, disequality, failure of equality and failure of disequality, respectively.

Definition 1. *Relations over $CTerm_{\perp,\mathsf{F}}$*
- $t \downarrow t' \Leftrightarrow_{def} t = t', t \in CTerm$. $\blacktriangleright$ $t \uparrow t' \Leftrightarrow_{def} t$ *and* t' *have a DC-clash.*
- $t \not\downarrow t' \Leftrightarrow_{def} t$ *or* t' *contain F as subterm, or they have a DC-clash.*
- $\not\uparrow$ *is defined as the least symmetric relation over $CTerm_{\perp,\mathsf{F}}$ satisfying:*
  - i) $X \not\uparrow X$, *for all* $X \in \mathcal{V}$ ii) $\mathsf{F} \not\uparrow t$, *for all* $t \in CTerm_{\perp,\mathsf{F}}$
  - iii) *if* $t_1 \not\uparrow t'_1, ..., t_n \not\uparrow t'_n$ *then* $c(t_1, ..., t_n) \not\uparrow c(t'_1, ..., t'_n)$, *for* $c \in DC^n$

These relations will be extended to general expressions by means of the function $==$, whose semantic meaning will be fixed in the proof calculus *SRL* of Section 3.3. Notice that $t \uparrow t' \Rightarrow t \not\downarrow t' \Rightarrow \neg(t \downarrow t')$ and $t \downarrow t' \Rightarrow t \not\uparrow t' \Rightarrow \neg(t \uparrow t')$ but the converse implications do not hold in general. Notice also that $t \not\downarrow t'$ and $t \not\uparrow t'$ can be both true for the same t, t', and in this case neither strict equality, $\downarrow$, nor disequality, $\uparrow$, stand for t, t'. This will be used in the *SRL*-calculus (Table 1) in the rule 12, which states when $==$ fails. Such a situation happens, for instance, with $suc(0)$ and $suc(\mathsf{F})$.

3.2 Set Oriented Syntax: Set-expressions and Programs

A set-expression is a syntactical construction designed for manipulating sets of values. We extend the signature with a new countable set of *indexed variables* Γ that will usually be written as $\alpha, \beta, ...$ A (total) set-expression $\mathcal{S}$ is defined as:

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

where $t, t' \in CTerm_{\mathsf{F}}$, $\overline{t} \in CTerm_{\mathsf{F}} \times ... \times CTerm_{\mathsf{F}}$, $f \in FS^n$, $\mathcal{S}_1, \mathcal{S}_2$ are set-expressions and $\alpha \in \Gamma$. We write $SetExp$ for the set of (total) set-expressions. The set $SetExp_\perp$ of *partial* set-expressions has the same syntax except that the cterms $t, t', \overline{t}$ can contain $\perp$. With respect to [16] we have added here the construction $t == t'$ to the syntax of set-expressions.

Given a set-expression $\mathcal{S}$ we distinguish two sets of variables in it: the set $PV(\mathcal{S}) \subset \Gamma$ of *produced variables* (those of indexed unions of $\mathcal{S}$), and the remaining $FV(\mathcal{S}) \subset \mathcal{V}$ or *free variables*. In order to avoid variable renaming and simplify further definitions, we always assume that produced variables of a set-expression are indexed only once in the entire set-expression.

We will use substitutions mapping variables of $\mathcal{V} \cup \Gamma$ into terms built up over $\mathcal{V} \cup \Gamma$. When applying a substitution $[\ldots, X_i/t_i, \ldots, \alpha_j/s_j, \ldots]$ to a set-expression $\mathcal{S}$, it is ensured throughout the paper that it does not bind any

produced variable of $\mathcal{S}$, i.e. $\{\ldots, \alpha_j, \ldots\} \cap PV(\mathcal{S}) = \emptyset$, and also that no produced variable is captured, i.e. $(\ldots \cup var(t_i) \cup \ldots \cup var(s_j) \cup \ldots) \cap PV(\mathcal{S}) = \emptyset$. It is easy to achieve these conditions with an adequate variable renaming, if necessary, of produced variables in $\mathcal{S}$.

We will also use *set-substitutions* for set-expressions: given $D = \{s_1, ..., s_n\} \subseteq CTerm_{\perp,\mathsf{F}}$ 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 $(...(\mathcal{S}[X_1/D_1])...)[X_n/D_n]$.

It is easy to transform any expression $e \in Term_{\perp,\mathsf{F}}$ into its corresponding set-expression $\mathcal{S} \in SetExp_{\perp,\mathsf{F}}$ while preserving the semantics with respect to the appropriate proof calculus (see [16] for details). As an example, if c is a constructor symbol and f and g are function symbols, the set-expression corresponding to $f(c, g(X))$ is $\bigcup_{\alpha \in g(X)} f(c, \alpha)$.

Programs consist of rules of the form: $f(t_1, \ldots, t_n) \twoheadrightarrow \mathcal{S}$, where $f \in FS^n$, $t_i \in CTerm$ (notice that F is not allowed to appear in t_i), the tuple $(t_1, \ldots, t_n)$ is linear (each variable occurs only once), $\mathcal{S} \in SetExp$ and $FV(\mathcal{S}) \subseteq var((t_1, \ldots, t_n))$.

Like in [18], we require programs to be *Complete Inductively Sequential Programs*, or *CIS*-programs for short ([16, 18, 2]). In this kind of programs the heads of the rules must be pairwise non-overlapping and cover all the possible cases of constructor symbols. The interesting point is that for any ground tuple made of terms $t \in CTerm$ there is *exactly one* program rule that can be used for reducing a call $f(\bar{t})$. Notice that this holds for $t \in CTerm$, but not necessarily for $t \in CTerm_{\mathsf{F}}$. If t contains F, which might happen in an intermediate step of a computation, then there could be no applicable rule.

In [3, 16, 18] one can find algorithms to transform general programs into *CIS*-programs while preserving their semantics. This transformation, as well as the transformation to set oriented syntax, can be made transparent to the user in a real implementation, as indeed happens with the small prototype we have developed for this paper.

For instance, the program of the example in Section 2 becomes the following when translated into a *CIS*-program with first order set oriented syntax:

$$prefix([\,], Ys) \twoheadrightarrow \{true\} \qquad prefix([X|Xs], [\,]) \twoheadrightarrow \{\mathsf{F}\}$$

$$prefix([X|Xs], [Y|Ys]) \twoheadrightarrow \bigcup_{\beta \in X==Y} \bigcup_{\gamma \in prefix(Xs, Ys)} ifThen(\beta, \gamma)$$

$$fprefix(Xs, Ys) \twoheadrightarrow \bigcup_{\alpha \in fails(prefix(Xs, Ys))} ifThenElse(\alpha, false, true)$$

The functions `if _ then _` and `if _ then _ else _` are translated as:

$$ifThen(true, X) \ \twoheadrightarrow \{X\} \qquad ifThenElse(true, X, Y) \ \twoheadrightarrow \{X\}$$
$$ifThen(false, X) \twoheadrightarrow \{\mathsf{F}\} \qquad ifThenElse(false, X, Y) \twoheadrightarrow \{Y\}$$

Notice that the original definition of `prefix` is completed in order to cover all the possible cases for the arguments. In particular, the second rule of `prefix` corresponds to a 'missing' case in the original definition, thus giving failure (the value F) in the completed one. Something similar happens with the second rule of *ifThen*.

$$
\begin{array}{ll}
\textbf{(1)} \quad \dfrac{}{\mathcal{S} \lhd \{\bot\}} \quad \mathcal{S} \in SetExp_\bot & \textbf{(2)} \quad \dfrac{}{\{X\} \lhd \{X\}} \quad X \in \mathcal{V}
\end{array}
$$

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

$$
\textbf{(4)} \quad \dfrac{\mathcal{S} \lhd \mathcal{C}}{f(\overline{t}) \lhd \mathcal{C}} \quad (f(\overline{t}) \twoheadrightarrow \mathcal{S}) \in \{R\theta \mid R = (f(\overline{l}) \twoheadrightarrow \mathcal{S}) \in \mathcal{P}, \theta \in CSubst_{\bot,\mathsf{F}}\}
$$

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

$$
\begin{array}{ll}
\textbf{(6)} \quad \dfrac{\mathcal{S} \lhd \{\mathsf{F}\}}{fails(\mathcal{S}) \lhd \{true\}} & \textbf{(7)} \quad \dfrac{\mathcal{S} \lhd \mathcal{C}}{fails(\mathcal{S}) \lhd \{false\}} \quad \begin{array}{l}\text{if there is some } t \in \mathcal{C} \\ \text{with } t \neq \bot,\ t \neq \mathsf{F}\end{array}
\end{array}
$$

$$
\begin{array}{ll}
\textbf{(8)} \quad \dfrac{\mathcal{S}_1 \lhd \mathcal{C}_1 \quad \mathcal{S}_2[\alpha/\mathcal{C}_1] \lhd \mathcal{C}}{\bigcup_{\alpha \in \mathcal{S}_1} \mathcal{S}_2 \lhd \mathcal{C}} & \textbf{(9)} \quad \dfrac{\mathcal{S}_1 \lhd \mathcal{C}_1 \quad \mathcal{S}_2 \lhd \mathcal{C}_2}{\mathcal{S}_1 \cup \mathcal{S}_2 \lhd \mathcal{C}_1 \cup \mathcal{C}_2}
\end{array}
$$

$$
\begin{array}{ll}
\textbf{(10)} \quad \dfrac{}{t == t' \lhd \{true\}} \quad \text{if } t \downarrow t' & \textbf{(11)} \quad \dfrac{}{t == t' \lhd \{false\}} \quad \text{if } t \uparrow t'
\end{array}
$$

$$
\textbf{(12)} \quad \dfrac{}{t == t' \lhd \{\mathsf{F}\}} \quad \text{if } t \not\downarrow t' \text{ and } t \not\uparrow t'
$$

Table 1. Rules for *SRL*-provability

3.3 The proof calculus *SRL*

Following a well established approach in functional logic programming, we fix the semantics of programs by means of a proof calculus determining which logical statements can be derived from a program. The starting point of this semantic approach was the *CRWL* framework [8, 9], which included a calculus 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. We extended *CRWL* to deal with failure, obtaining in [15, 17] the calculus *CRWLF*; the main insight was to replace single reduction statements 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.

The calculus *CRWLF* was adapted to *CIS*-programs with set oriented syntax in [16, 18], and the resulting calculus was called *SRL* (for *Set Reduction Logic*). Here we extend *SRL* to cope with equality. The new calculus, commented below, is presented in Table 1.

Rules 1 to 4 are "classical" in *CRWL(F)* [9, 17, 15, 16]. Notice that rule 4 uses a c-instance of a program rule that is unique if it exists (*CIS*-programs ensure it). If such c-instance does not exist, then by rule 5, the corresponding set-expression reduces to $\{\mathsf{F}\}$. As mentioned before when describing *CIS*-programs, this might happen because of the presence of F at some position in $f(\overline{t})$. Rules 6 and 7 establish the meaning of the function $fails(\mathcal{S})$ and rules 8 and 9 are natural to understand from classical set theory. Finally, rules 10, 11 and 12 define the meaning of the function $==$ as a a three-valued function: $\{true\}$ for the case of

equality, $\{\mathit{false}\}$ for disequality, and $\{\mathsf{F}\}$ if case of failure of both. Notice that given $t, s \in CTerm_{\perp,\mathsf{F}}$ the conjunction of $t \not\!\Downarrow s$ and $t \not\!\Uparrow s$ means that t and s are identical except possibly at the positions (of which there must be at least one) where t or s contain the symbol F.

Given a program $\mathcal{P}$ and $\mathcal{S} \in SetExp_{\perp}$ we write $\mathcal{P} \vdash_{SRL} \mathcal{S} \lhd \mathcal{C}$, or simply $\mathcal{S} \lhd \mathcal{C}$ for sort, if the relation $\mathcal{S} \lhd \mathcal{C}$ is provable with respect to SRL and the program $\mathcal{P}$. The denotation of $\mathcal{S}$ (we also call it the *semantics* of $\mathcal{S}$) is defined as $[\![\mathcal{S}]\!] = \{\mathcal{C} \mid \mathcal{S} \lhd \mathcal{C}\}$. Then the denotation of a set-expression is a set of sets of (possible partial) cterms.

4 Operational Procedure: Set Narrowing with Equality

In this section we enlarge the narrowing relation *SNarr* of [18] with the equality function $==$. First, set-expressions are enriched by adding sets of disequalities to them. These disequalities must be manipulated at the operational level, so we introduce a normalization function *solve* and appropriate semantic notions that will be useful later. Then we fix the operational behavior of the equality function $==$ by means of a specific narrowing relation for it, $\succ\!\!\!\longrightarrow_\theta$, and we give correctness and completeness results for it with respect to *SRL*. Finally, making use of these new capabilities, we integrate the equality function into the general narrowing relation *SNarr*, and we show the corresponding correctness and completeness results, again with respect to *SRL*.

4.1 Disequality Manipulation

In order to introduce the equality function at the operational level we need an explicit manipulation of disequalities. We will work with sets δ of disequalities of the form $t \neq s$, where $t, s \in CTerm$. A first important notion is that of *solution*:

Definition 2. *Given* $\delta = \{t_1 \neq s_1, \ldots, t_n \neq s_n\}$ *we say that* $\sigma \in CSubst_{\perp,\mathsf{F}}$ *is a solution for* δ, *and write* $\sigma \in Sol(\delta)$, *if* $t_1\sigma \uparrow s_1\sigma, \ldots, t_n\sigma \uparrow s_n\sigma$.

We are particularly interested in sets of *solved forms* of disequalities of the form $X \neq t$ (with $X \not\equiv t$), where the variable X and those of t are all in $\mathcal{V}$, i.e., there are not produced variables in them. We introduce a function *solve* to obtain solved forms for sets of disequalities between cterms. As solved forms are not unique in general, *solve* returns the set of solved forms from a set of disequalities:

▶ $solve(\emptyset) = \{\emptyset\}$ ▶ $solve(\{X \neq X\} \cup \delta) = \emptyset$

▶ $solve(\{c \neq c\} \cup \delta) = \emptyset$, for any $c \in DC^0$

▶ $solve(\{c(\overline{t}) \neq d(\overline{t}')\} \cup \delta) = solve(\delta)$, if $c \neq d$

▶ $solve(\{X \neq t\} \cup \delta) = \{\{X \neq t\} \cup \delta' \mid \delta' \in solve(\delta)\}$

▶ $solve(\{c(t_1, \ldots, t_n) \neq c(t'_1, \ldots, t'_n)\} \cup \delta) = \{\delta_i \cup \delta' \mid \delta_i \in solve(\{t_i \neq t'_i\}), \delta' \in solve(\delta)\}$

$$\boxed{\begin{array}{l}
\textbf{(1)}\quad \emptyset \succ\!\!\!\longrightarrow_{\epsilon} true \qquad\qquad \textbf{(2)}\; X == X, C \succ\!\!\!\longrightarrow_{\epsilon} C \qquad \text{if } X \notin \Gamma \\[2mm]
\textbf{(3)}\; X == t, C \succ\!\!\!\longrightarrow_{[X/t]} C[X/t] \qquad \begin{array}{l}\text{if } X \neq t,\, X \notin \Gamma \cup var(t),\, \Gamma \cap var(t) = \emptyset \\ \text{and } t \text{ does not contain } \mathsf{F}\end{array} \\[3mm]
\textbf{(4)}\; c(t_1,\ldots,t_n) == c(s_1,\ldots,s_n), C \succ\!\!\!\longrightarrow_{\epsilon} t_1 == s_1, \ldots, t_n == s_n, C \\[3mm]
\textbf{(5)}\; c(\overline{t}) == d(\overline{s}), C \succ\!\!\!\longrightarrow_{\epsilon} false \\[3mm]
\textbf{(6)}\; X == t, C \succ\!\!\!\longrightarrow_{\epsilon} false \mid_{X \neq t} \qquad \begin{array}{l}\text{if } X \neq t,\, X \notin \Gamma,\, \Gamma \cap var(t) = \emptyset \\ \text{and } t \text{ does not contain } \mathsf{F}\end{array} \\[3mm]
\textbf{(7)}\; X == c(t_1,\ldots,t_n), C \succ\!\!\!\longrightarrow_{[X/c(\overline{Y})]} (Y_1 == t_1, \ldots, Y_n == t_n, C)[X/c(\overline{Y})] \\
\qquad \text{if } X \notin \Gamma \text{ and } var(\overline{t}) \cup \Gamma \neq \emptyset \text{ or } \overline{t} \text{ contains } \mathsf{F},\, \overline{Y} \text{ fresh variables} \\[3mm]
\textbf{(8)}\; X == c(t_1,\ldots,t_n), C \succ\!\!\!\longrightarrow_{[X/d(\overline{Y})]} false \qquad \begin{array}{l}\text{if } X \notin \Gamma \text{ and } var(\overline{t}) \cup \Gamma \neq \emptyset \\ \text{or } \overline{t} \text{ contains } \mathsf{F},\, \overline{Y} \text{ fresh variables}\end{array} \\[3mm]
\textbf{(9)}\; \mathsf{F} == t_1, \ldots, \mathsf{F} == t_n \succ\!\!\!\longrightarrow_{\epsilon} \mathsf{F}
\end{array}}$$

Table 2. Rules for $==$

The interesting property about this function is:

Proposition 1. *If* $solve(\delta) = \{\delta_1, \ldots, \delta_n\}$, *then* $Sol(\delta) = Sol(\delta_1) \cup \ldots \cup Sol(\delta_n)$.

Now, the idea is to associate a set of disequalities δ to any set-expression $\mathcal{S}$, for which we use the notation $\mathcal{S} \square \delta$. For such set-expressions with disequalities we extend the notion of semantics in order to obtain a better way for establishing semantics equivalence between set-expressions. Given $\mathcal{S} \square \delta$, we are interested in the semantics of $\mathcal{S}$ under total substitutions that also are solutions of δ. With this aim we introduce the notion of *hyper-semantics*:

Definition 3. *The hyper-semantics of a set-expression with disequalities* $\mathcal{S} \square \delta$ *(with respect to a program* $\mathcal{P}$*) is defined as:* $[\![\mathcal{S} \square \delta]\!] = \lambda \sigma \in CSubst \cap Sol(\delta).[\![\mathcal{S}\sigma]\!]$

4.2 Operational behavior of $==$

The mechanism for evaluating the function $==$ is defined by means of the relation $\succ\!\!\!\longrightarrow_{\theta}$ of Table 2. The substitution $\theta \in CSubst$ is the *computed substitution*. This relation works on a set of *constraints* of the form $\{t_1 == s_1, \ldots, t_n == s_n\}$, where $t_1, \ldots, t_n, s_1, \ldots s_n \in CTerm_{\mathsf{F}}$. As an abuse of notation we will write $t_1 == s_1, \ldots, t_n == s_n, C$ for representing the set $\{t_1 == s_1, \ldots, t_n == s_n\} \cup C$, where C is a set of constraints; it is not relevant any ordering on the constraints and the relation $==$ is symmetric. Some comments about the rules of Table 2:
Some comments about the rules:

– rule 2 erase a variable identity, 3 is for binding, 4 for decomposition and 7 is an imitation rule. These are general rules that can be used in intermediate steps of a $\succ\!\!\!\longrightarrow$-derivation to obtain any result (*true, false* or F);

- the result *true* (equality) can be obtained by applying the general rules and finally rule 1, that stands for the empty set of constraints;
- *false* (disequality) is reached by checking a clash of constructor symbols in rule 5, or by introducing such a clash in rule 8;
- *false*$|_{X/=u}$ (conditional disequality) can be obtained in rule 6. In this case, the value *false* is conditioned to the disequality $X \neq t$. This is the point where an explicit disequality can be added as part of the solution;
- finally, F is obtained in rule 9 when all the equalities are of the form F $== t$. In such case neither a equality nor a disequality can be proved.

Apart from the possible values that can return the calculus, it is possible that no rule is applicable at a given step in the derivation. This happens when there are produced variables of Γ that block the evaluation. Notice that the evaluation of an equality $t == s$ will be required from the narrowing relation *SNarr*, where this equality is a part of the full set-expression. So, $t == s$ can contain produced variables indexed in the corresponding set-expression. For example, using the function `prefix` of Section 2, the equality `prefix 0 [0,X]`$== $`Y` written as set-expression is $\bigcup_{\alpha \in prefix(0,[0,X])} \alpha == Y$. The equality $\alpha == Y$ cannot be reduced by the rules of $\rightarrowtail$ because α blocks the evaluation, i.e., it is associated to an expression $(prefix(0, [0, X]))$ that requires further evaluation.

We will use the relation $\rightarrowtail^{*}_{\theta}$ for 0 or more steps of $\rightarrowtail$, where θ indicates the composition of θ's in individual steps.

In order to simplify the relation *SNarr* of the next section we consider a uniform shape (ω, δ) for the results provided by the rules for $\rightarrowtail$, where ω is the value and δ a set of disequalities (always empty, except for the case of conditional *false*). So, with this format the possible results are: $(true, \emptyset)$, $(false, \emptyset)$, $(false, \{X \neq u\})$ and $(F, \emptyset)$.

4.3   The operational mechanism

For selecting a redex, *SNarr* uses the notion of *context* defined as:

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

where $\mathcal{S}$ is a set-expression, and C_1 and C_2 are contexts.

For defining the operational behavior of *fails* and also for the completeness result of Section 4.3 we need to consider the *information set* $\mathcal{S}^* \subseteq CTerm_{\perp,F}$ for a set-expression $\mathcal{S}$, defined as:

$$\blacktriangleright (f(\bar{t}))^* = (fails(\mathcal{S}))^* = (t == s)^* = \{\perp\} \qquad \blacktriangleright (\{t\})^* = \{t\}$$
$$\blacktriangleright (\bigcup_{\alpha \in \mathcal{S}'} \mathcal{S})^* = (\mathcal{S}[\alpha/\perp])^* \qquad \blacktriangleright (\mathcal{S}_1 \cup \mathcal{S}_2)^* = \mathcal{S}_1^* \cup \mathcal{S}_2^*$$

Now, we can define *one-narrowing-step relation:* given a program $\mathcal{P}$, two set-expressions $\mathcal{S}, \mathcal{S}' \in SetExp$, $\theta \in CSubst_F$, and δ, δ' sets of disequalities in

Cntx	$C\ [\mathcal{S}]\square\delta \underset{\theta}{\rightsquigarrow} C\theta\ [\mathcal{S}']\square\delta'$	if $\mathcal{S}\square\delta \underset{\theta}{\rightsquigarrow} \mathcal{S}'\square\delta'$
Nrrw$_1$	$f(\overline{t})\square\delta \underset{\theta\mid_{var(\overline{t})}}{\rightsquigarrow} \mathcal{S}\theta\square\delta'$	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 $Dom(\theta) \cap \Gamma = \emptyset$ and $\delta' \in solve(\delta\theta)$
Nrrw$_2$	$f(\overline{t})\square\delta \underset{\epsilon}{\rightsquigarrow} \{\mathsf{F}\}\square\delta$	if for every rule $(f(\overline{s}) \twoheadrightarrow \mathcal{S}) \in \mathcal{P}$, $\overline{s}$ and $\overline{t}$ have a $DC \cup \{\mathsf{F}\}$-clash
Nrrw$_3$	$t == s\square\delta \underset{\theta\mid_{var(t)\cup var(s)}}{\rightsquigarrow} \{\omega\}\square\delta'$	if $t == s \underset{\theta}{\overset{*}{\rightarrowtail}} (\omega, \delta'')$ and $\delta' \in solve(\delta\theta \cup \delta'')$
Fail$_1$	$fails(\mathcal{S})\square\delta \underset{\epsilon}{\rightsquigarrow} \{true\}\square\delta$	if $\mathcal{S}^* = \{\mathsf{F}\}$
Fail$_2$	$fails(\mathcal{S})\square\delta \underset{\epsilon}{\rightsquigarrow} \{false\}\square\delta$	if $\exists t \in \mathcal{S}^*\ t \neq \bot,\ t \neq \mathsf{F}$
Dist	$\bigcup_{\alpha \in \mathcal{S}_1 \cup \mathcal{S}_2} \mathcal{S}_3\square\delta \underset{\epsilon}{\rightsquigarrow} \bigcup_{\alpha \in \mathcal{S}_1} \mathcal{S}_3 \cup \bigcup_{\alpha \in \mathcal{S}_2} \mathcal{S}_3\square\delta$	
Bind	$\bigcup_{\alpha \in \{t\}} \mathcal{S}\square\delta \underset{\epsilon}{\rightsquigarrow} \mathcal{S}[\alpha/t]\square\delta$	
Flat	$\bigcup_{\alpha \in \bigcup_{\beta \in \mathcal{S}_1} \mathcal{S}_2} \mathcal{S}_3\square\delta \underset{\epsilon}{\rightsquigarrow} \bigcup_{\beta \in \mathcal{S}_1} \bigcup_{\alpha \in \mathcal{S}_2} \mathcal{S}_3\square\delta$	
Elim	$\bigcup_{\alpha \in \mathcal{S}'} \mathcal{S}\square\delta \underset{\epsilon}{\rightsquigarrow} \mathcal{S}\square\delta$	if $\alpha \notin FV(\mathcal{S})$

Table 3. Rules for *SNarr*

solved form, a narrowing step is expressed as $\mathcal{S}\square\delta \underset{\theta}{\rightsquigarrow} \mathcal{S}'\square\delta'$ where the relation

$\mathcal{S}\square\delta \underset{\theta}{\rightsquigarrow} \mathcal{S}'\square\delta$ is defined in Table 3. In the following we use *SNarr* as the name for this relation.

Rules are essentially those of [18], except for the disequality sets δ and for the rule **Nrrw$_3$**, specific for ==. This rule performs a subcomputation with the rules for == and integrates the result in the current set-expression. With respect to the other rules: **Cntx** select a redex in the set-expression, **Nrrw$_1$** evaluates a function call by finding the appropriate (unique if it exists) applicable rule; **Nrrw$_2$** returns a failure if such rule does not exist; rules **Fail$_{1,2}$** evaluate the function *fails* according to the information set of the current set-expression. And the rest of rules correspond to easy manipulations from the point of view of sets.

As an example of derivation, consider again the *CIS*-program containing the functions *prefix*, *fprefix*, *ifThen* and *ifThenElse* of Section 3.2. We show one of the four possible derivations for the expression *fprefix(Xs, [Y])* in Table 4 (we write *iT* for *ifThen* and *iTe* for *ifThenElse*). At each step we underline the redex in use, and we point out the rule of *SNarr* used for the step. In the case of **Nrrw$_1$** we also point out the rule of the program used for the step.

Notice that the computed substitution is found at step (2), by applying the second rule of *prefix* to narrow the redex *prefix(Xs, [Y])*. At step (3), the rule **Nrrw$_3$** throws a subcomputation for the function ==. It succeeds with a conditional *false* that inserts the disequality $\{Y \neq B\}$ into the computation. The function *fails* is reduced to *true* by means of **Fail$_1$** at step (7). At the end

$$
\begin{array}{lll}
(1) & \underline{fprefix(Xs,[Y])}\square\emptyset \; \rightsquigarrow_{\epsilon} & (\text{Nrrw}_1\text{-fprefix}) \\[4pt]
(2) & \bigcup_{\alpha\in fails(\underline{prefix(Xs,[Y])})} iTe(\alpha, false, true)\square\emptyset \; \underset{Xs/[B|C]}{\rightsquigarrow} & (\text{Nrrw}_1\text{-prefix}_3) \\[4pt]
(3) & \bigcup_{\alpha\in fails(\bigcup_{\beta\in\underline{B==Y}}\bigcup_{\gamma\in prefix(C,[\;])} iT(\beta,\gamma))} iTe(\alpha, false, true)\square\emptyset \; \rightsquigarrow_{\epsilon} & (\text{Nrrw}_3) \\[4pt]
 & \quad B==Y \; \rightarrowtail_{\epsilon} \; (false, \{Y\neq B\}) \qquad (\text{by rule 6 of ==}) \\[4pt]
(4) & \bigcup_{\alpha\in fails(\bigcup_{\underline{\beta\in\{false\}}}\bigcup_{\gamma\in prefix(C,[\;])} iT(\beta,\gamma))} iTe(\alpha, false, true)\square\{Y\neq B\} \; \rightsquigarrow_{\epsilon} & (\text{Bind}) \\[4pt]
(5) & \bigcup_{\alpha\in fails(\bigcup_{\gamma\in prefix(C,[\;])} \underline{iT(false,\gamma)})} iTe(\alpha, false, true)\square\{Y\neq B\} \; \rightsquigarrow_{\epsilon} & (\text{Nrrw}_1\text{-iT}_2) \\[4pt]
(6) & \bigcup_{\alpha\in fails(\underline{\bigcup_{\gamma\in prefix(C,[\;])}\{\mathtt{F}\}})} iTe(\alpha, false, true)\square\{Y\neq B\} \; \rightsquigarrow_{\epsilon} & (\text{Elim}) \\[4pt]
(7) & \bigcup_{\alpha\in \underline{fails(\{\mathtt{F}\})}} iTe(\alpha, false, true)\square\{Y\neq B\} & (\text{Fail}_1) \\[4pt]
(8) & \bigcup_{\alpha\in\{true\}} iTe(\alpha, false, true)\square\{Y\neq B\} & (\text{Bind}) \\[4pt]
(9) & \underline{iTe(true, false, true)}\square\{Y\neq B\} & (\text{Nrrw}_1\text{-iTe}_1) \\[4pt]
(10) & \{false\}\square\{Y\neq B\} \\[4pt]
& \textbf{Final answer:} \;\; (\{false\}, \{Xs=[B|C], Y\neq B\})
\end{array}
$$

Table 4. Derivation for `fprefix Xs [Y]`

we obtain the answer $(\{false\}, \{Xs=[B|C], Y/=B\})$ as expected. The three remaining answers showed in Section 2 can be obtained in a similar way.

The correctness of *SNarr* with respect to *SRL* is easy to formulate thanks to the notion of hyper-semantics (Definition 3). Essentially it guarantees that the hyper-semantics of a set-expression is preserved under *SNarr* derivations.

Theorem 1 (Correctness of *SNarr*). *Let* $\mathcal{S}, \mathcal{S}' \in SetExp$, $\theta \in CSubst_F$ *and* δ, δ' *sets of solved disequalities. Then:* $\mathcal{S}\square\delta \overset{*}{\underset{\theta}{\rightsquigarrow}} \mathcal{S}'\square\delta' \Rightarrow [\![\mathcal{S}\theta\square\delta']\!] = [\![\mathcal{S}'\square\delta']\!]$

The completeness result is quite technical and ensures that the *SAS*'s obtained by *SRL* are captured by the information provided by *SNarr*, that also finds the appropriate substitutions.

Theorem 2 (Completeness of *SNarr*). *Let* $\mathcal{S}\square\delta$ *be a set-expression with disequalities and* $\theta \in Sol(\delta)$. *If* $\mathcal{P} \vdash_{SRL} \mathcal{S}\theta \triangleleft \mathcal{C}$ *then there exists a derivation* $\mathcal{S}\square\delta \overset{*}{\underset{\theta'}{\rightsquigarrow}} \mathcal{S}'\square\delta'$, *introducing a set of fresh variables* Π, *such that:*

▶ $\theta = (\theta'\mu)\mid_{\mathcal{V}-\Pi}$, *for some* $\mu \in CSubst$ ▶ $\mathcal{C} \sqsubseteq (\mathcal{S}'\mu)^*$ ▶ $\mu \in Sol(\delta')$

As a corollary we obtain the following result that essentially says that we have reached our goal of devising an operational procedure for constructive failure.

Corollary 1. *If* $\mathcal{P} \vdash_{SRL} \mathcal{S}\theta \triangleleft \{\mathtt{F}\}$, *then there exist* $\theta', \mu \in CSubst$ *such that* $\mathcal{S}\square\emptyset \overset{*}{\underset{\theta'}{\rightsquigarrow}} \{\mathtt{F}\}\square\delta$, $\theta = \theta'\mu$ *and* $\mu \in Sol(\delta)$.

5 Conclusions and Future Work

We have addressed in a rigorous way in this paper the problem of how to realize constructive failure in a functional logic language having a built-in equality function. The motivation was that both failure and equality are important expressive resources in functional logic programs, but there was a lack of a convenient combination of them in previous works. We were guided by the following two aims, hopefully achieved:

- We wanted our work to be technically precise from the theoretical point of view, not only at the semantic description level but also at the operational level, with results relating both levels. For the first level we give a proof calculus fixing the semantics of programs and expressions. Failure is used in programs by means of a two-valued function *fails* giving *true* or *false*, while for equality we use a three-valued function == giving as possible values *true*, *false* or F.
- We wanted failure and equality to be well-behaved from a functional logic programming perspective, which implies the ability to operate in presence of non-ground expressions. Thus, following a usual terminology for negation in logic programming, we expect failure to be *constructive*, and this, when combined with equality, required to consider disequality constraints as a part of the operational procedure, which essentially consists in set-narrowing (in the spirit of [18]) combined with a (novel) evaluation mechanism for equality, all proceeding in an ambient of disequality constraints, which can appear in answers.

The operational procedure, although complex in its description, is amenable for a quite direct implementation. We have a small prototype with which all the examples in the paper have been executed. It includes the program transformation needed to convert programs in user-friendly $\mathcal{TOY}$ syntax into *CIS*-programs with set oriented syntax. As a useful tool for experimentation the prototype includes also the possibility of tracing an execution by automatically generating and compiling a TeX file with the sequence of performed steps. The derivation in Table 4 has been obtained with the aid of this tool.

In the future, we plan to embed the prototype into the $\mathcal{TOY}$ system, and to improve its efficiency.

References

1. M. Abengózar-Carneros et al. $\mathcal{TOY}$: a multiparadigm declarative language, version 2.0. Technical report, Dep. SIP, UCM Madrid, January 2001.
2. S. Antoy. Definitional trees. In *Proc. ALP'92*, pages 143–157, Springer LNCS 632, 1992.
3. S. Antoy. Constructor-based conditional narrowing. In *Proc. PPDP'01*, pages 199–206, ACM Press, 2001.
4. K.R. Apt and R. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19&20:9–71, 1994.

5. P. Arenas-Sánchez, A. Gil-Luezas, F. J. López Fraguas, Combining lazy narrowing with disequality constraints. In *Proc. PLILP'94*, pages 385–399, Springer LNCS 844, 1994.

6. D. Chan. Constructive negation based on the completed database. In *Proc. IC-SLP'88*, pages 111–125, 1988.

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

8. 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.

9. 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.

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

11. M. Hanus (ed.). Curry: An integrated functional logic language. Available at `http://www.informatik.uni-kiel.de/~curry/report.html`, 2000.

12. M. Hanus and F. Steiner Controlling search in declarative programs. In *Proc. PLILP/ALP'98*, pages 374–390, Springer LNCS 1490, 1998.

13. 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.

14. F.J. López-Fraguas and J. Sánchez-Hernández. Disequalities may help to narrow. *Proc. APPIA-GULP-PRODE*, pages 89–104, 1999.

15. 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.

16. 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.

17. F.J. López-Fraguas and J. Sánchez-Hernández. A proof theoretic approach to failure in functional logic programming. To appear in *TPLP*.

18. 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.

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

20. 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.

21. J.C. Reynolds. *Theories of programming languages*. Cambridge Univ. Press, 1998.

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

23. P.J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118:12–33, 1995.