Failure and Equality in Functional Logic Programming ⋆

F. J. López-Fraguas and J. Sánchez-Hernández [1,2]

Dep. Sistemas Informáticos y Programación
Univ. Complutense de Madrid
Madrid, Spain

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.

Key words: Constructive Failure, Equality, Narrowing,
Functional Logic Programming

1 Introduction

Functional logic programming (see [10]) attempts to amalgamate into a single paradigm the best features of functional and logic languages. In a functional logic language like *Curry* [11] or $\mathcal{TOY}$ [13,1], one should be able to select for each problem the convenient style (logic, functional or mixed) or feature to use. From this point of view, there is an important expressive resource of logic programming –negation as failure [7,4]– that does not readily extend to the functional logic case and is missed in most works in the field. For this reason, some works [19,20,15,17,16,18] have addressed the subject of *failure of reduction*[3] as a useful programming construct which is the natural extension

⋆ Work partially supported by the Spanish project TIC2002-01167 'MELODIAS'

[1] Email: `fraguas@sip.ucm.es`

[2] Email: `jaime@sip.ucm.es`

[3] To head normal form, see Sect. 2.

of the notion of negation as failure to the functional logic setting.

On the other hand, almost any real program needs to use some kind of equality test between expressions. But equality is another issue not easy to extend properly from logic or functional programming to the functional logic case. Equality is easy to express in logic programming by means of unification. It can be made explicit by a predicate *eq* defined by the clause `eq(X,X) :- true.` Unfortunately, this simple definition is not legal in constructor based lazy functional logic languages like *Curry* or $\mathcal{TOY}$, which require (as functional languages do) heads of rules to be left linear (i.e., variables are not repeated in heads). Functional languages use *strict* equality (two expressions are equal if they can be reduced to the same total constructor term). Although equality is usually built-in, there would be no essential problem, in the functional setting, with programming strict equality by means of program rules, like any other function[4]. However, when moving to functional logic programming, this works fine only for equalities involving ground 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 provides examples motivating the interest of our work. Section 3 summarizes the basic concepts about the *S*et *R*eduction *L*ogic 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 ma-

[4] At least for constructor-based data types, which is our case of interest.

nipulation of disequalities and some other technical notions. The last section summarizes some conclusions.

2 Failure & equality: motivation

We give here a pair of small but not artificial examples 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], which is a mixture of Prolog (upper-cased variables, lower-cased constructor symbols) and Haskell (curried notation). The functions `if _ then _` and `if _ then _ else _` are defined as usual. The results indicated in the examples 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 might have in addition some disequality conditions.

Our first example, taken from [18], has two purposes:

- to recall the interest of constructive failure in a functional logic language, and how it is conceived as a two-valued function in [18].
- to show that the operational treatment of equality of [18] has some drawbacks, and to illustrate how our present work overcomes them.

Example 2.1 A typical way of programming a directed acyclic graph in a functional logic language is by means of a non-deterministic function `next` returning, in different program rules, the adjacents of a given node. A concrete graph could be given by a data type for nodes:

```
data node = a | b | c | d
```

along with a concrete definition of `next`:

```
next a = b
next a = c
next b = c
next b = d
```

Notice that `next` is indeed non-deterministic. An answer for `next X` would be, for example, (`{b,c}`,`{X=a}`). Now, the predicate `path` expresses that two nodes are connected:

```
path X Y = if (X == Y)  then true
                        else path (next X) Y
```

Here, `==` must be understood as the *equality function*. As it is well known (see e.g. [10]), the sensible notion in functional or functional-logic programming is *strict equality*: `e == e'` should be `true` if `e,e'` are reducible to the same constructor term, and `false` if they can be reduced to some extent as to present a constructor clash. In addition, in this paper, we must consider a third value F, corresponding to failure of the other two; for instance, `next c == d` must return F because `next c` fails, and then the given equality cannot be reduced neither to `true` nor to `false`.

As a simple but illustrative example of the use of failure, consider the property: *'a node is safe if it is not connected to the node* `d`*'*. This is very easy to program with the aid of failure:

```
safe X = fails (path X d)
```

The function `fails` is intended to return the value `true` if its argument finitely fails to be reduced to head normal form (variable or constructor rooted expression), and to `false` if a reduction to hnf exists. With this notion, `(safe a)` should be reduced to `false`, while `(safe c)` should be reduced to `true`. Furthermore, failure should be *constructive*, which means that `fails` should operate soundly in presence of variables. For instance, the expression `safe X` should have four answers: `(false,{X=a})`; `(false,{X=b})`; `(true,{X=c})` ; `(false,{X=d})`. To achieve the same behavior without constructive failure is not *that* easy, and requires to re-program the representation of the graph in a different way. Existing systems like *Curry* or $\mathcal{TOY}$ are not able to exhibit such behavior[5] .

Now, to see what happens with equality, assume first that `==` is seen as any other function, an therefore defined by rules, like in [18]. There would be 16 rules defining `==`:

```
a == a = true      a == b = false     ...      a == d = false

b == a = false     b == b = true      ...      b == d = false

...
```

Consider the goal expression `fails (path X Y)`. The rules above produce 16 different answers consisting of a value (`true` or `false`) and a substitution for `X,Y`. We find among them: `(false,{X=a,Y=a})` (and similarly with `X=b,Y=b`, etc.), or `(true,{X=c,Y=a})` (and similarly with `X=c,Y=b` or `X=c,Y=d`). Notice that all solutions are ground. In general, with N nodes we obtain N^2 ground answers.

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

In contrast, in this paper `==` has a specific evaluation mechanism involving disequality conditions which can be a part of answers. The virtue of disequality conditions is that one single disequality can encompass the same information as many substitutions. For instance, `Y/=c` expresses the same as the disjunction of the substitutions `Y=a`, `Y=b`, `Y=d`. Furthermore, the operational procedure described in Section 4 might produce non ground substitutions where the above rules for `==` produced ground ones.

For our goal example `fails (path X Y)` we obtain 10 answers, among them: `(false,{X=Y})` (abbreviating four different ground answers), or `(true, {X=c, Y/=c})` (abbreviating three different ground answers). In the case of N nodes, it can be proved that the number of answers is $1 + k_1 + \ldots + k_N$, where k_i is the number of nodes connected to the ith-node.

In our next example constructive failure and equality coexist again, but in this case the behavior of `==`, if defined by program rules, is even worse than in the previous example.

Example 2.2 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
```

To obtain from it a boolean function `fprefix` is easy using failure:

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

To see what happens with equality in this case, assume first that `==` is defined by rules. Assume also, for simplicity, that we are dealing with lists of natural numbers represented by the constructors `0` and `suc`[6]. The rules for `==` would be:

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

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

With this definition of equality an expression like `fprefix Xs [Y]` produces infinitely many answers binding `Xs` to lists of natural numbers and `Y` to a natural number. With the evaluation of equality of this paper, 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.

[6] We are not cheating with this assumption. The situation becomes worse as the number of constructor symbols grows.

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. A constraint `e == e'` or `e /= e'` might succeed or fail. To use in these works a two-valued boolean equality function, say `eq`, required 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 *S*et *R*eduction *L*ogic (*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 $DC_\Sigma = \bigcup_{n\in\mathbb{N}} DC^n_\Sigma$ 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 $DC_\Sigma \cap FS_\Sigma = \emptyset$. 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 $\perp$, that plays the role of the undefined value. We define $\Sigma_{\perp,\mathsf{F}} = \Sigma \cup \{\perp, \mathsf{F}\}$; the sets $Term_{\perp,\mathsf{F}}$ and $CTerm_{\perp,\mathsf{F}}$ 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 $CSubst_{\perp,\mathsf{F}}$ are defined as mappings from $\mathcal{V}$ into $CTerm, CTerm_{\mathsf{F}}$ and $CTerm_{\perp,\mathsf{F}}$ 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 $Term_{\perp,\mathsf{F}}$ can be defined as the least partial ordering over $Term_{\perp,\mathsf{F}}$ satisfying the following properties:

- $\perp \sqsubseteq e$ for all $e \in Term_{\perp,\mathsf{F}}$,
- $h(e_1, ..., e_n) \sqsubseteq h(e'_1, ..., e'_n)$, 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 3.1 Relations over $CTerm_{\perp,\mathsf{F}}$:

- $t \downarrow t' \Leftrightarrow_{def} t = t', t \in CTerm$
- $t \uparrow t' \Leftrightarrow_{def} t$ 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:

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

where $t, t' \in CTerm_{\mathsf{F}}$, $\bar{t} \in CTerm_{\mathsf{F}} \times ... \times CTerm_{\mathsf{F}}$, $f \in FS^n$, $\mathcal{S}_1, \mathcal{S}_2$ 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 $t, t', \bar{t}$ can contain $\perp$.

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 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 *C*omplete *I*nductively *S*equential 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 $f(\overline{t})$. 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. This corresponds to rule 5 in Table 1.

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:

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

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

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

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

$$\begin{array}{llll} ifThen(true, X) & \twoheadrightarrow \{X\} & \qquad ifThenElse(true, X, Y) & \twoheadrightarrow \{X\} \\ ifThen(false, X) & \twoheadrightarrow \{\mathsf{F}\} & \qquad ifThenElse(false, X, Y) & \twoheadrightarrow \{Y\} \end{array}$$

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*.

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 $e \rightarrow t$, 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 $e \triangleleft \mathcal{C}$, 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. For example, using the program of the Example 1 in Section 2, *CRWLF* could prove $next(a) \triangleleft \{b, c\}$, and also $next(c) \triangleleft \{\mathsf{F}\}$.

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 $f(\bar{t})$. 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 three-valued function: $\{true\}$ for the case of equality, $\{false\}$ 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 $\mathcal{P} \vdash_{SRL} \mathcal{S} \triangleleft \mathcal{C}$, or simply $\mathcal{S} \triangleleft \mathcal{C}$ for sort, if the relation $\mathcal{S} \triangleleft \mathcal{C}$ 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 $[\![\mathcal{S}]\!] = \{\mathcal{C} \mid \mathcal{S} \triangleleft \mathcal{C}\}$. Then the denotation of a set-expression is a set of sets of (possibly 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, $\underset{\theta}{\rightarrowtail}$, 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*.

$$\textbf{(1)}\quad \frac{}{\mathcal{S} \triangleleft \{\bot\}} \qquad \mathcal{S} \in \mathit{SetExp}_\bot$$

$$\textbf{(2)}\quad \frac{}{\{X\} \triangleleft \{X\}} \qquad X \in \mathcal{V}$$

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

$$\textbf{(4)}\quad \frac{\mathcal{S} \triangleleft \mathcal{C}}{f(\overline{t}) \triangleleft \mathcal{C}} \qquad \begin{array}{r}(f(\overline{t}) \twoheadrightarrow \mathcal{S}) \in \{R\theta \mid R = (f(\overline{t}) \twoheadrightarrow \mathcal{S}) \in \mathcal{P},\\ \theta \in CSubst_{\bot,\mathsf{F}}\}\end{array}$$

$$\textbf{(5)}\quad \frac{}{f(\overline{t}) \triangleleft \{\mathsf{F}\}} \qquad \begin{array}{l}\text{for all } (f(\overline{s}) \twoheadrightarrow \mathcal{S}') \in \mathcal{P},\\ \overline{t} \text{ and } \overline{s} \text{ have a } DC \cup \{\mathsf{F}\}\text{-clash}\end{array}$$

$$\textbf{(6)}\quad \frac{\mathcal{S} \triangleleft \{\mathsf{F}\}}{\mathit{fails}(\mathcal{S}) \triangleleft \{\mathit{true}\}}$$

$$\textbf{(7)}\quad \frac{\mathcal{S} \triangleleft \mathcal{C}}{\mathit{fails}(\mathcal{S}) \triangleleft \{\mathit{false}\}} \qquad \text{if there is some } t \in \mathcal{C} \text{ with } t \neq \bot,\ t \neq \mathsf{F}$$

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

$$\textbf{(9)}\quad \frac{\mathcal{S}_1 \triangleleft \mathcal{C}_1 \qquad \mathcal{S}_2 \triangleleft \mathcal{C}_2}{\mathcal{S}_1 \cup \mathcal{S}_2 \triangleleft \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\textbf{(10)}\quad \frac{}{t == t' \triangleleft \{\mathit{true}\}} \qquad \text{if } t \downarrow t'$$

$$\textbf{(11)}\quad \frac{}{t == t' \triangleleft \{\mathit{false}\}} \qquad \text{if } t \uparrow t'$$

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

Table 1
Rules for *SRL*-provability

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

The interesting property about this function is:

Proposition 4.2 *If* $solve(\delta) = \{\delta_1, \ldots, \delta_n\}$, *(*$n \geq 0$*) 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 4.3 The *hyper-semantics of a set-expression with disequalities* $\mathcal{S}\square\delta$ (with respect to a program $\mathcal{P}$) is defined as:

$$\boldsymbol{\llbracket} \mathcal{S}\square\delta \boldsymbol{\rrbracket} = \lambda\sigma \in CSubst \cap Sol(\delta).\llbracket \mathcal{S}\sigma \rrbracket$$

This subtle notion requires some explanations:

- We introduce it because the functional nature of $\boldsymbol{\llbracket}\_\boldsymbol{\rrbracket}$ reflects better than $\llbracket\_\rrbracket$ the idea of semantic equivalence between set-expressions with variables. Consider, for instance, the functions f and g defined as:

$$\begin{array}{ll} f(true) \twoheadrightarrow \{true\} & g(true) \twoheadrightarrow \{true\} \\ f(false) \twoheadrightarrow \{false\} & g(false) \twoheadrightarrow \{true\} \end{array}$$

Then we have $\llbracket f(X) \rrbracket = \llbracket g(X) \rrbracket = \{\perp\}$, despite of the fact that f and g are clearly different. The hyper-semantics captures the difference, since

$[\![f(X)\Box\emptyset]\!] \neq [\![g(X)\Box\emptyset]\!]$: taking the substitution $\sigma = [X/false]$ we have $[\![f(false)]\!] = \{\bot, false\}$ while $[\![g(false)]\!] = \{\bot, true\}$.

- The reason of restricting σ to be a solution of δ can be illustrated by considering $\delta = \{X \neq false\}$: in this case, for the previous program, it is desirable to have $[\![f(X)\Box\delta]\!] = [\![g(X)\Box\delta]\!]$ because in fact f and g behave the same over values different from *false*.
- Finally, the technical reason of restricting σ to be total is to preserve the hyper-semantics of expressions involving the function $==$. For example, it is desirable to have $[\![X == X\Box\emptyset]\!] = \{\bot, true\}$, but if we allow σ to introduce $\bot$ we could not guarantee $X\sigma == X\sigma \lhd \{true\}$.

4.2 Operational behavior of $==$

The mechanism for evaluating the function $==$ is defined by means of the relation $\underset{\theta}{\rightarrowtail}$ 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 $\rightarrowtail$-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 $\mathsf{F} == t$. 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

(1) $\emptyset \underset{\epsilon}{\rightarrowtail} true$	
(2) $X == X, C \underset{\epsilon}{\rightarrowtail} C$	if $X \notin \Gamma$
(3) $X == t, C \underset{[X/t]}{\rightarrowtail} C[X/t]$	if $X \neq t$, $X \notin \Gamma \cup var(t)$, $\Gamma \cap var(t) = \emptyset$ and t does not contain F
(4) $c(t_1, \ldots, t_n) == c(s_1, \ldots, s_n), C \underset{\epsilon}{\rightarrowtail} t_1 == s_1, \ldots, t_n == s_n, C$	
(5) $c(\overline{t}) == d(\overline{s}), C \underset{\epsilon}{\rightarrowtail} false$	
(6) $X == t, C \underset{\epsilon}{\rightarrowtail} false \mid_{X \neq t}$	if $X \neq t$, $X \notin \Gamma$, $\Gamma \cap var(t) = \emptyset$ and t does not contain F
(7) $X == c(t_1, \ldots, t_n), C \underset{[X/c(\overline{Y})]}{\rightarrowtail} (Y_1 == t_1, \ldots, Y_n == t_n, C)[X/c(\overline{Y})]$	if $X \notin \Gamma$ and $var(\overline{t}) \cup \Gamma \neq \emptyset$ or $\overline{t}$ contains F, $\overline{Y}$ fresh vars.
(8) $X == c(t_1, \ldots, t_n), C \underset{[X/d(\overline{Y})]}{\rightarrowtail} false$	if $X \notin \Gamma$ and $var(\overline{t}) \cup \Gamma \neq \emptyset$ or $\overline{t}$ contains F, $\overline{Y}$ fresh vars.
(9) $\mathsf{F} == t_1, \ldots, \mathsf{F} == t_n \underset{\epsilon}{\rightarrowtail} \mathsf{F}$	

Table 2
Rules for ==

`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.

The next results state that the operational mechanism for $==$, $\rightarrowtail$, behaves well with respect to the semantic calculus *SRL* of Section 3.3. We use the relation $\overset{*}{\underset{\theta}{\rightarrowtail}}$ for 0 or more steps of $\rightarrowtail$, where θ indicates the composition of θ's in individual steps.

Theorem 4.4 (Correctness of $\rightarrowtail$) *Assume $t, s \in CTerm_{\mathsf{F}}$. Then:*

- $t == s \overset{*}{\underset{\theta}{\rightarrowtail}} true \Rightarrow t\theta == s\theta \lhd \{true\}$
- $t == s \overset{*}{\underset{\theta}{\rightarrowtail}} false \Rightarrow t\theta == s\theta \lhd \{false\}$

- $t == s \overset{*}{\underset{\theta}{\rightarrowtail}} \mathit{false} \mid_{X \neq u} \Rightarrow t\theta\sigma == s\theta\sigma \lhd \{\mathit{false}\}$, *for all* $\sigma \in \mathit{Sol}(\{X/=u\})$

- $t == s \overset{*}{\underset{\theta}{\rightarrowtail}} \mathsf{F} \Rightarrow t\theta == s\theta \lhd \{\mathsf{F}\}$

With respect to completeness, consider two cterms t and s, and a substitution θ, such that $t\theta == s\theta$ can be reduced to some value with SRL. We must prove that the rules for $==$ are able to get the same value for $t == s$, while generalizing the substitution θ. As a technical detail, we must take care of the set Π of fresh variables introduced by the rules for $==$, in particular by rule 7.

Theorem 4.5 (Completeness with respect to *SRL*) *Let* $t, s \in CTerm_{\mathsf{F}}$ *and* δ *a set of disequalities. Then:*

- *if* $t\theta == s\theta \lhd \{true\}$ *then* $t == s \overset{*}{\underset{\theta'}{\rightarrowtail}} true$, *and* $\exists \mu.\theta = (\theta'\mu) \mid_{\mathcal{V}-\Pi}$
- *if* $t\theta == s\theta \lhd \{false\}$ *then:*
  - · $t == s \overset{*}{\underset{\theta'}{\rightarrowtail}} \mathit{false}$, *and* $\exists \mu.\theta = (\theta'\mu) \mid_{\mathcal{V}-\Pi}$, *or*
  - · $t == s \overset{*}{\underset{\theta'}{\rightarrowtail}} \mathit{false} \mid_{Z \neq w}$, *and* $\exists \mu.\theta = (\theta'\mu) \mid_{\mathcal{V}-\Pi} \wedge \mu \in \mathit{Sol}(\{Z \neq w\})$
- *if* $t\theta == s\theta \lhd \{\mathsf{F}\}$ *then* $t == s \overset{*}{\underset{\theta'}{\rightarrowtail}} \mathsf{F}$, *and* $\exists \mu.\theta = (\theta'\mu) \mid_{\mathcal{V}-\Pi}$

where Π *is the set of fresh variables introduced in the* $\overset{*}{\underset{\theta'}{\rightarrowtail}}$ *derivation*

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: $(true, \emptyset)$, $(\mathit{false}, \emptyset)$, $(\mathit{false}, \{X \neq u\})$ and $(\mathsf{F}, \emptyset)$.

4.3 The operational mechanism

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

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

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

For defining the operational behavior of *fails* and also for the completeness result of Section 4.3 we need the notion of *information set*:

Definition 4.6 Given a set-expression $\mathcal{S}$ its *information set* $\mathcal{S}^* \subseteq CTerm_{\perp,\mathsf{F}}$ is defined as:

- $(f(\overline{t}))^* = (\mathit{fails}(\mathcal{S}))^* = (t == s)^* = \{\perp\}$

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

Table 3
Rules for *SNarr*

- $(\{t\})^* = \{t\}$
- $(\bigcup_{\alpha\in\mathcal{S}'} \mathcal{S})^* = (\mathcal{S}[\alpha/\bot])^*$
- $(\mathcal{S}_1 \cup \mathcal{S}_2)^* = \mathcal{S}_1^* \cup \mathcal{S}_2^*$

Now, we can define *one-narrowing-step relation:* given a program $\mathcal{P}$, two set-expressions $\mathcal{S}, \mathcal{S}' \in SetExp$, $\theta \in CSubst_{\mathsf{F}}$, and δ, δ' sets of disequalities in solved form, a narrowing step is expressed as $\mathcal{S}\Box\delta \underset{\theta}{\rightsquigarrow} \mathcal{S}'\Box\delta'$ where the relation $\mathcal{S}\Box\delta \underset{\theta}{\rightsquigarrow} \mathcal{S}'\Box\delta$ is defined in Table 3. In the following we use *SNarr* as the name for this relation.

(1) $\underline{\mathit{fprefix}(Xs,[Y])}\Box\emptyset \underset{\epsilon}{\rightsquigarrow}$ (Nrrw$_1$-fprefix)

(2) $\bigcup_{\alpha\in \mathit{fails}(\underline{\mathit{prefix}(Xs,[Y])})} iTe(\alpha,\mathit{false},\mathit{true})\Box\emptyset \underset{Xs/[B|C]}{\rightsquigarrow}$ (Nrrw$_1$-prefix$_3$)

(3) $\bigcup_{\alpha\in \mathit{fails}(\bigcup_{\beta\in \underline{B==Y}}\bigcup_{\gamma\in \mathit{prefix}(C,[\ ])} iT(\beta,\gamma))} iTe(\alpha,\mathit{false},\mathit{true})\Box\emptyset \underset{\epsilon}{\rightsquigarrow}$ (Nrrw$_3$)

$B == Y \underset{\epsilon}{\rightarrowtail} (\mathit{false},\{Y\neq B\})$ (by rule 6 of ==)

(4) $\bigcup_{\alpha\in \mathit{fails}(\bigcup_{\beta\in \underline{\{\mathit{false}\}}}\bigcup_{\gamma\in \mathit{prefix}(C,[\ ])} iT(\beta,\gamma))} iTe(\alpha,\mathit{false},\mathit{true})\Box\{Y\neq B\} \underset{\epsilon}{\rightsquigarrow}$ (Bind)

(5) $\bigcup_{\alpha\in \mathit{fails}(\bigcup_{\gamma\in \mathit{prefix}(C,[\ ])} \underline{iT(\mathit{false},\gamma)})} iTe(\alpha,\mathit{false},\mathit{true})\Box\{Y\neq B\} \underset{\epsilon}{\rightsquigarrow}$ (Nrrw$_1$-iT$_2$)

(6) $\bigcup_{\alpha\in \mathit{fails}(\underline{\bigcup_{\gamma\in \mathit{prefix}(C,[\ ])}\{\mathsf{F}\}})} iTe(\alpha,\mathit{false},\mathit{true})\Box\{Y\neq B\} \underset{\epsilon}{\rightsquigarrow}$ (Elim)

(7) $\bigcup_{\alpha\in \underline{\mathit{fails}(\{\mathsf{F}\})}} iTe(\alpha,\mathit{false},\mathit{true})\Box\{Y\neq B\}$ (Fail$_1$)

(8) $\bigcup_{\underline{\alpha\in\{\mathit{true}\}}} iTe(\alpha,\mathit{false},\mathit{true})\Box\{Y\neq B\}$ (Bind)

(9) $\underline{iTe(\mathit{true},\mathit{false},\mathit{true})}\Box\{Y\neq B\}$ (Nrrw$_1$-iTe$_1$)

(10) $\{\mathit{false}\}\Box\{Y\neq B\}$

Final answer: $(\{\mathit{false}\},\{Xs=[B|C],Y\neq B\})$

Table 4
Derivation for `fprefix Xs [Y]`

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 $\mathit{fprefix}(Xs,[Y])$ 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 $\mathit{prefix}(Xs,[Y])$. 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 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 4.3). Essentially it guarantees that the hyper-semantics of a set-expression is preserved under *SNarr* derivations.

Theorem 4.7 (Correctness of *SNarr***)** *Let* $\mathcal{S}$, $\mathcal{S}' \in SetExp$, $\theta \in CSubst_{\mathsf{F}}$ *and* δ,δ' *sets of solved disequalities. Then:* $\mathcal{S}\Box\delta \overset{*}{\underset{\theta}{\leadsto}} \mathcal{S}'\Box\delta' \Rightarrow [\![\mathcal{S}\theta\Box\delta']\!] = [\![\mathcal{S}'\Box\delta']\!]$

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 4.8 (Completeness of *SNarr***)** *Let* $\mathcal{S}\Box\delta$ *be a set-expression with disequalities and* $\theta \in Sol(\delta)$. *If* $\mathcal{P} \vdash_{SRL} \mathcal{S}\theta \lhd \mathcal{C}$ *then there exists a derivation* $\mathcal{S}\Box\delta \overset{*}{\underset{\theta'}{\leadsto}} \mathcal{S}'\Box\delta'$, *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 4.9 *If* $\mathcal{P} \vdash_{SRL} \mathcal{S}\theta \lhd \{\mathsf{F}\}$, *then there exist* $\theta', \mu \in CSubst$ *such that* $\mathcal{S}\Box\emptyset \overset{*}{\underset{\theta'}{\leadsto}} \{\mathsf{F}\}\Box\delta$, $\theta = \theta'\mu$ *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 that we show, 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] Abengózar-Carneros, M. et al. "$\mathcal{TOY}$: a multiparadigm declarative language, version 2.0." Technical report SIP 119/00, UCM Madrid, February 2002. Available at http://titan.sip.ucm.es/toy/toyreport.pdf.

[2] Antoy, S. *Definitional trees.* In Proc. Int. Conf. on Algebraic and Logic Programming (ALP'92), pages 143–157, Springer LNCS 632, 1992.

[3] Antoy, S. *Constructor-based conditional narrowing.* Proc. Int. Conf. on Principles and Practice of Declarative Programming (PPDP'01), pages 199–206, ACM Press, 2001.

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

[5] Arenas-Sánchez, P., A. Gil-Luezas, and F. J. López Fraguas *Combining lazy narrowing with disequality constraints.* In Proc.Int. Symp. on Programming Languages Implementation and Logic Programming (PLILP'94), pages 385–399, Springer LNCS 844, 1994.

[6] Chan, D. *Constructive negation based on the completed database.* In Proc. Int. Conf. and Symp. on Logic Programming (ICSLP'88), pages 111–125, 1988.

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

[8] González-Moreno, J. C., T. Hortalá-González, F. J. López-Fraguas, and M. Rodríguez-Artalejo. *A rewriting logic for declarative programming.* In Proc. European Symp. on Programming (ESOP'96), pages 156–172, Springer LNCS 1058, 1996.

[9] González-Moreno, J. C., 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)** (1999), 47–87.

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

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

[12] Hanus, M., and F. Steiner. *Controlling search in declarative programs.* In Proc.Int. Symp. on Programming Languages Implementation and Logic Programming (PLILP/ALP'98), pages 374–390, Springer LNCS 1490, 1998.

[13] López-Fraguas, F. J., and J. Sánchez-Hernández. *$\mathcal{TOY}$: A multiparadigm declarative system.* In Proc. Int. Con. on Rewriting Techniques and Applications (RTA'99), pages 244–247, Springer LNCS 1631, 1999.

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

[15] López-Fraguas, F. J., and J. Sánchez-Hernández. *Proving failure in functional logic programs.* In Proc. Int. Conf. on Computational Logic (CL'00), pages 179–193, Springer LNAI 1861, 2000.

[16] López-Fraguas, F. J., and J. Sánchez-Hernández. *Functional logic programming with failure: A set-oriented view.* In Proc. Int. Conf. on Logic Programming and Automated Reasoning (LPAR'01), pages 455–469, Springer LNAI 2250, 2001.

[17] López-Fraguas, F. J., and J. Sánchez-Hernández. *A proof theoretic approach to failure in functional logic programming.* To appear in Theory and Practice of Logic Programming.

[18] López-Fraguas, F. J., and J. Sánchez-Hernández. *Narrowing failure in functional logic programming.* In Proc. Int. Symp. on Functional and Logic Programming (FLOPS'02), pages 212–227, Springer LNCS 2441, 2002.

[19] Moreno-Navarro, J. J. *Default rules: An extension of constructive negation for narrowing-based languages.* In Proc. Int. Conf. on Logic Programming (ICLP'94), pages 535–549, The MIT Press, 1994.

[20] Moreno-Navarro, J. J. *Extending constructive negation for partial functions in lazy functional-logic languages.* In Proc. Extensions of Logic Programming (LP'96), pages 213–227, Springer LNAI 1050, 1996.

[21] Reynolds, J. C. "Theories of programming languages". Cambridge Univ. Press, 1998.

[22] Stuckey, P. J. *Constructive negation for constraint logic programming.* In Proc. Int. Conf. on Logic in Computer Science (LICS'91), pages 328–339, 1991.

[23] Stuckey, P. J. *Negation and constraint logic programming.* Information and Computation **118** (1995), 12–33.