Proving Failure in Functional Logic Programs*

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

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

Abstract. How to extract negative information from programs is an important issue in logic programming. Here we address the problem for functional logic programs, from a proof-theoretic perspective. The starting point of our work is *CRWL* (Constructor based ReWriting Logic), a well established theoretical framework for functional logic programming, whose fundamental notion is that of non-strict non-deterministic function. We present a proof calculus, *CRWLF*, which is able to deduce negative information from *CRWL*-programs. In particular, *CRWLF* is able to prove 'finite' failure of reduction within *CRWL*.

1 Introduction

We address in this paper the problem of extracting negative information from functional logic programs. The question of negation is a main topic of research in the logic programming field, and the most common approach is *negation as failure*, as an easy effective approximation to the *CWA* (*closed world assumption*), which is a simple, but uncomputable, way of deducing negative information from positive programs (see e.g. [1] for a survey on negation in logic programming).

On the other hand, functional logic programming (*FLP* for short) is a powerful programming paradigm trying to combine the nicest properties of functional and logic programming (see [4] for a now 'classical' survey on *FLP*). *FLP* subsumes *pure* logic programming: predicates can be defined as functions returning the value 'true', for which definite clauses can be written as conditional rewrite rules. In some simple cases it is enough, to handle negation, just to define predicates as two-valued boolean functions returning the values 'true' or 'false'. But negation as failure is far more expressive, and it is then of clear interest to investigate a similar notion for the case of *FLP*. Failure in logic programs, when seen as functional logic programs, corresponds to failure of reduction to 'true'. This generalizes to a natural notion of failure in *FLP*, which is 'failure of reduction'.

As technical setting for our work we have chosen *CRWL* [3], a well established theoretical framework for *FLP*. The fundamental notion in *CRWL* is that of non-strict non-deterministic function, for which *CRWL* provides a firm logical basis, as mentioned for instance in [5]. Instead of equational logic, *CRWL* considers a Constructor based ReWriting Logic, presented by means of a proof calculus,

* The authors have been partially supported by the Spanish CICYT (project TIC 98-0445-C03-02 'TREND')

which determines what statements can be deduced from a given program. In addition to the proof-theoretic semantics, [3] develop a model theoretic semantics for *CRWL*, with existence of distinguished free term models for programs, and a sound and complete lazy narrowing calculus as operational semantics. The *CRWL* framework (with many extensions related to types, HO and constraints) has been implemented in the system $\mathcal{TOY}$ [8].

Here we are interested in extending the proof-theoretic side of *CRWL* to cope with failure. More concretely, we look for a proof calculus, which will be called *CRWLF* ('*CRWL* with failure'), which is able to prove failure of reduction in *CRWL*. Since reduction in *CRWL* is expressed by proving certain statements, our calculus will provide proofs of unprovability within *CRWL*. As for the case of *CWA*, unprovability is not computable, which means that our calculus can only give an approximation, corresponding to cases which can be intuitively described as 'finite failures'.

There are very few works about negation in *FLP*. In [10] the work of Stuckey about *constructive negation* [12] is adapted to the case of *FLP* with strict functions and innermost narrowing as operational mechanism. In [11] a similar work is done for the case of non-strict functions and lazy narrowing. The approach is very different of the proof-theoretic view of our work. The fact that we also consider non-deterministic functions makes a significant difference.

The proof-theoretic approach, although not very common, has been followed sometimes in the logic programming field, as in [6], which develops for logic programs (with negation) a framework which resembles, in a very general sense, *CRWL*: a program determine a deductive system for which deducibility, validity in a class of models, validity in a distinguished model and derivability by an operational calculus are all equivalent. Our work attempts to be the first step of what could be a similar programme for *FLP* extended with the use of failure as a programming construct.

The rest of the paper is organized as follows. In Section 2 we give the essentials of *CRWL* which are needed for our work. Section 3 presents the *CRWLF*-calculus, preceded by some illustrative examples. Section 4 contains the results about *CRWLF*. Most of the results are technically involved, and their proofs have been skipped because of the lack of space (full details can be found in [9]). Section 5 contains some conclusions.

2 The *CRWL* Framework

We give here a short summary of *CRWL*, in its proof-theoretic face. Model theoretic semantics and lazy narrowing operational semantics are not considered here. Full details can be found in [3].

2.1 Technical Preliminaries

We assume a signature $\Sigma = DC_\Sigma \cup FS_\Sigma$ where $DC_\Sigma = \bigcup_{n\in\mathbb{N}} DC^n_\Sigma$ is a set of *constructor* symbols and $FS_\Sigma = \bigcup_{n\in\mathbb{N}} FS^n_\Sigma$ is a set of *function* symbols, all

of them with associated arity and such that $DC_\Sigma \cap FS_\Sigma = \emptyset$. 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 up with Σ and $\mathcal{V}$ in the usual way, and we distinguish the subset $CTerm_\Sigma$ of (total) constructor terms or (total) *c-terms*, which only make use of DC_Σ and $\mathcal{V}$. The subindex Σ will usually be omitted. Terms intend to represent possibly reducible expressions, while c-terms represent data values, not further reducible.

We will need sometimes to use the signature $\Sigma_\perp$ which is the result of extending Σ with the new constant (0-arity constructor) $\perp$, that plays the role of the undefined value. Over $\Sigma_\perp$, we can build up the sets $Term_\perp$ and $CTerm_\perp$ of (partial) terms and (partial) c-terms respectively. Partial c-terms represent the result of partially evaluated expressions; thus, they can be seen as approximations to the value of expressions.

As usual notations we will write $X, Y, Z, ...$ for variables, $c, d, ...$ for constructor symbols, $f, g, ...$ for functions, $e, e', ...$ for terms and $s, t, ...$ for c-terms.

We will use the sets of substitutions $CSubst = \{\theta : \mathcal{V} \to CTerm\}$ and $CSubst_\perp = \{\theta : \mathcal{V} \to CTerm_\perp\}$. We write $e\theta$ for the result of applying θ to e.

Given a set of constructor symbols S we say that the terms t and t' have an S-clash if they have different constructor symbols of S at the same position.

2.2 The Proof Calculus for *CRWL*

A *CRWL*-program $\mathcal{P}$ is a set of conditional rewrite rules of the form:

$$\underbrace{f(t_1, ..., t_n)}_{head} \to \underbrace{e}_{body} \Leftarrow \underbrace{C_1, ..., C_n}_{condition}$$

where $f \in FS^n$; $(t_1, ..., t_n)$ is a linear tuple (each variable in it occurs only once) with $t_1, ..., t_n \in CTerm$; $e \in Term$ and each C_i is a constraint of the form $e' \bowtie e''$ (*joinability*) or $e' \diamond e''$ (*divergence*) where $e', e'' \in Term$. The reading of the rule is: $f(t_1, ..., t_n)$ reduces to e if the conditions $C_1, ..., C_n$ are satisfied. We write $\mathcal{P}_f$ for the set of defining rules of f in $\mathcal{P}$.

From a given program $\mathcal{P}$, the proof calculus for *CRWLF* can derive three kinds of statements:

- *Reduction or approximation statements*: $e \to t$, with $e \in Term_\perp$ and $t \in CTerm_\perp$. The intended meaning of such statement is that e can be reduced to t, where reduction may be done by applying rewriting rules of $\mathcal{P}$ or by replacing subterms of e by $\perp$. If $e \to t$ can be derived, t represents one of the possible values of the denotation of e.
- *Joinability statements*: $e \bowtie e'$, with $e, e' \in Term_\perp$. The intended meaning in this case is that e and e' can be both reduced to some common totally defined value, that is, we can prove $e \to t$ and $e' \to t$ for some $t \in CTerm$.
- *Divergence statements*: $e \diamond e'$, with $e, e' \in Term_\perp$. The intended meaning now is that e and e' can be reduced to some (possibly partial) c-terms t and t' such that they have a DC-clash.

Table 1. Rules for *CRWL*-provability

(1) $\dfrac{}{e \to \bot}$

(2) $\dfrac{}{X \to X}$ $\quad X \in \mathcal{V}$

(3) $\dfrac{e_1 \to t_1, ..., e_n \to t_n}{c(e_1, ..., e_n) \to c(t_1, ..., t_n)}$ $\quad c \in DC^n, \quad t_i \in CTerm_\bot$

(4) $\dfrac{e_1 \to s_1, ..., e_n \to s_n \quad C \quad e \to t}{f(e_1, ..., e_n) \to t}$ $\quad$ if $t \not\equiv \bot, R \in \mathcal{P}_f$ $(f(s_1, ..., s_n) \to e \Leftarrow C) \in [R]_\bot$

(5) $\dfrac{e \to t \quad e' \to t}{e \bowtie e'}$ $\quad$ if $t \in CTerm$

(6) $\dfrac{e \to t \quad e' \to t'}{e \diamond e'}$ $\quad$ if $t, t' \in CTerm_\bot$ and have a $DC-$clash

It must be mentioned that the *CRWL* framework as presented in [3] does not consider divergence conditions. They have been incorporated to *CRWL* in [7] as a useful and expressive resource for programming.

When using function rules to derive statements, we will need to use what are called *c-instances* of such rules: the set of c-instances of a program rule R is defined as $[R]_\bot = \{R\theta | \theta \in CSubst_\bot\}$. This allows, in particular, to express parameter passing.

Table 1 shows the proof calculus for *CRWL*. We write $\mathcal{P} \vdash_{CRWL} \varphi$ for expressing that the statement φ is provable from the program $\mathcal{P}$.

The rule 4 allows to use c-instances of program rules to prove approximations. These c-instances may contain $\bot$ and rule (1) allows to reduce any expression to $\bot$. This reflects a non-strict semantics.

A distinguished feature of *CRWL* is that functions can be *non-deterministic*. For example, assuming the constructors z (zero) and s (successor) for natural numbers, a non-deterministic function *coin* can defined by the rules $coin \to z$ and $coin \to s(z)$. The use of c-instances in rule (4) instead of general instances corresponds to *call time choice* semantics for non-determinism (see [3]). As an example, if in addition to *coin* we consider the function definition *mkpair(X)* $\to$ *pair(X,X)* (*pair* is a constructor), it is possible to build a *CRWL*-proof for *mkpair(coin)* $\to$ *pair(z,z)* and also for *mkpair(coin)* $\to$ *pair(s(z),s(z))*, but not for *mkpair(coin)* $\to$ *pair(z,s(z))*.

Observe that $\diamond$ is not the logical negation of $\bowtie$. They are not even incompatible: due to non-determinism, two expressions e, e' can satisfy both $e \bowtie e'$ and $e \diamond e'$ (although this cannot happen if e, e' are c-terms). In the 'coin' example, we can derive both $coin \bowtie z$ and $coin \diamond z$.

We can define the *denotation* of an expression e as the set of c-terms to which e can be reduced according to this calculus: $[\![e]\!] = \{t \in CTerm_\bot | \mathcal{P} \vdash_{CRWL} e \to t\}$

3 The *CRWLF* Framework

We now address the problem of failure in *CRWL*. Our primary interest is to obtain a calculus able to prove that a given expression fails to be reduced. Since reduction corresponds in *CRWL* to approximation statements $e \to t$, we can reformulate our aim more precisely: we look for a calculus able to prove that a given expression e has no possible reduction (other than the trivial $e \to \bot$) in *CRWL*, i.e., $[\![e]\!] = \{\bot\}$.

Of course, we cannot expect to achieve that with full generality since, in particular, the reason for having $[\![e]\!] = \{\bot\}$ can be non-termination of the program as rewrite system, which is uncomputable. Instead, we look for a suitable computable approximation to the property $[\![e]\!] = \{\bot\}$, corresponding to cases where failure of reduction is due to 'finite' reasons, which can be constructively detected and managed.

Previous to the formal presentation of the calculus, which will be called *CRWLF* (for '*CRWL* with failure') we give several simple examples for a preliminary understanding of some key aspects of it, and the reasons underlying some of its technicalities.

3.1 Some Illustrative Examples

Consider the following functions, in addition to *coin* , defined in Sect. 2:

$$f(z) \to f(z) \qquad g(s(s(X)) \to z \qquad \begin{array}{l} h \to s(z) \\ h \to s(h) \end{array} \qquad k(X) \to z \Leftarrow X \bowtie s(z)$$

The expressions $f(z)$ and $f(s(z))$ fail to be reduced, but for quite different reasons. In the first case $f(z)$ does not terminate. The only possible proof accordingly to *CRWL* is $f(z) \to \bot$ (by rule 1); any attempt to prove $f(z) \to t$ with $t \neq \bot$ would produce an 'infinite derivation'. In the second case, the only possible proof is again $f(s(z)) \to \bot$, but if we try to prove $f(s(z)) \to t$ with $t \neq \bot$ we have a kind of '*finite failure*': rule 4 needs to solve the parameter passing $s(z) \to z$, that could be finitely checked as failed, since no rule of the *CRWL*-calculus is applicable. The *CRWLF*-calculus does not prove non-termination of $f(z)$, but will be able to detect and manage the failure for $f(s(z))$. In fact it will be able to perform a *constructive proof* of this failure.

Consider now the expression $g(coin)$. Again, the only possible reduction is $g(coin) \to \bot$ and it is intuitively clear that this is another case of finite failure. But this failure is not as simple as in the previous example for $f(s(z))$: in this case the two possible reductions for *coin* to defined values are $coin \to z$ and $coin \to s(z)$. Both of z and $s(z)$ fail to match the pattern $s(s(X))$ in the rule for g, but none of them can be used separately to detect the failure of $g(coin)$. A suitable idea is to collect the set of defined values to which a given expression can be reduced. In the case of *coin* that set is $\{z, s(z)\}$. The fact that $\mathcal{C}$ is the collected set of values of e is expressed in *CRWLF* by means of the statement $e \lhd \mathcal{C}$. In our example, *CRWLF* will prove $coin \lhd \{z, s(z)\}$. Statements $e \lhd \mathcal{C}$

generalize the approximation statements $e \rightarrow t$ of *CRWL*, and in fact can replace them. Thus, *CRWLF* will not need to use explicit $e \rightarrow t$ statements.

How far should we go when collecting values? The idea of collecting all values (and to have them completely evaluated) works fine in the previous example, but there are problems when the collection is infinite. For example, according to its definition above, the expression h can be reduced to any positive natural number, so the corresponding set would be $\{s(z), s(s(z)), s(s(s(z))), ...\}$. Then, what if we try to reduce the expression $f(h)$?. From an intuitive point of view it is clear that the value z will not appear in this set, because all the values in it have the form $s(...)$. We can represent all this values by the set $\{s(\bot)\}$. Here we can understand $\bot$ as an *incomplete information*: we know that all the possible values for h are successor of 'something'; we do not know what is this 'something', but in fact, we do not need to know it. Anyway the set does not contain the value z, so $f(h)$ fails. Notice that all the possible values for h are represented (not present) in the set $\{s(\bot)\}$, and this information is sufficient to prove the failure of $f(h)$. The *CRWLF*-calculus will be able to prove the statement $h \lhd \{s(\bot)\}$, and we say that $\{s(\bot)\}$ is a *Sufficient Approximation Set* (*SAS*) for h.

In general, an expression will have multiple *SAS*'s. Any expression has $\{\bot\}$ as its simplest *SAS*. And, for example, the expression h has an infinite number of *SAS*'s: $\{\bot\}$, $\{s(\bot)\}$, $\{s(z), s(s(\bot))\}$,... The *SAS*'s obtained by the calculus for *coin* are $\{\bot\}$, $\{\bot, s(\bot)\}$,$\{\bot, s(z)\}$, $\{z, \bot\}$, $\{z, s(\bot)\}$ and $\{z, s(z)\}$. The *CRWLF*-calculus provides appropriate rules for working with *SAS*'s. The derivation steps will be guided by these *SAS*'s in the same sense that *CRWL* is guided by approximation statements.

Failure of reduction is due in many cases to failure in proving the conditions in the program rules. The calculus must be able to prove those failures. Consider for instance the expression $k(z)$. In this case we would try to use the c-instance $k(z) \rightarrow z \Leftarrow z \bowtie s(z)$ that allows to perform parameter passing. But the condition $z \bowtie s(z)$ is clearly not provable, so $k(z)$ must fail. For achieving it we must be able to give a proof for *'$z \bowtie s(z)$ cannot be proved with respect to CRWL'*. For this purpose we introduce a new constraint $e \not\bowtie e'$ that will be true if we can build a *proof of non-provability* for $e \bowtie e'$. In our case, $z \not\bowtie s(z)$ is clear simply because of the clash of constructors. In general the proof for a constraint $e \not\bowtie e'$ will be guided by the corresponding *SAS*'s for e and e' as we will see in the next section. As our initial *CRWL* framework also allows constraints of the form $e <> e'$, we need still another constraint $\not<>$ for expressing 'failure of $<>$'.

There is another important question to justify: we use an explicit representation for failure by means of the new constant symbol F. Let us examine some examples involving failures. First, consider the expression $g(s(f(s(z))))$; for reducing it we would need to do parameter passing, i.e., matching $s(f(s(z)))$ with some c-instance of the pattern $s(s(X))$ of the definition of g. As $f(s(z))$ fails to be reduced the parameter passing must also fail. If we take $\{\bot\}$ as an *SAS* for $f(s(z))$ we have not enough information for detecting the failure (nothing can be said about the matching of $s(s(X))$ and $s(\bot)$). But if we take $\{\mathsf{F}\}$ as an *SAS* for $f(s(z))$, this provides enough information to ensure that $s(\mathsf{F})$ cannot match

any c-instance of the pattern $s(s(X))$. Notice that we allow the value F to appear inside the term $s(\mathsf{F})$. It could appear that the information $s(\mathsf{F})$ is essentially the same of F (for instance, F also fails to match any c-instance of $s(s(X))$), but this is not true in general. For instance, the expression $g(s(s(f(s(z)))))$ is reducible to z. But if we take the SAS $\{\mathsf{F}\}$ for $f(s(z))$ and we identify the expression $s(s(f(s(z))))$ with F, matching with the rule for g would not succeed, and the reduction of $g(s(s(f(s(z)))))$ would fail.

We can now proceed with the formal presentation of the *CRWLF*-calculus.

3.2 Technical Preliminaries

We introduce the new constant symbol F into the signature Σ to obtain $\Sigma_{\perp,\mathsf{F}} = \Sigma \cup \{\perp, \mathsf{F}\}$. The sets $Term_{\perp,\mathsf{F}}, CTerm_{\perp,\mathsf{F}}$ are defined in the natural way and we will use the set $CSubst_{\perp,\mathsf{F}} = \{\theta : \mathcal{V} \to CTerm_{\perp,\mathsf{F}}\}$.

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 $i \in \{1, ..., n\}$, $h \in DC \cup FS$

The intended meaning of $e \sqsubseteq e'$ is that e is less defined or has less information than e'. Two expressions $e, e' \in Term_{\perp,\mathsf{F}}$ are *consistent* if they can be refined to obtain the same information, i.e., if there exists $e'' \in Term_{\perp,\mathsf{F}}$ such that $e \sqsubseteq e''$ and $e' \sqsubseteq e''$.

Notice that the only relations satisfied by F are $\perp \sqsubseteq \mathsf{F}$ and $\mathsf{F} \sqsubseteq \mathsf{F}$. In particular, F is maximal. This is reasonable, since F represents 'failure of reduction' and this gives a no further refinable information about the result of the evaluation of an expression. This contrasts with the status given to failure in [11], where F is chosen to verify $\mathsf{F} \sqsubseteq t$ for any t different from $\perp$.

The class of programs that we consider in the following is less general than in the *CRWL* framework. Rules of functions have the same form, but they must not contain *extra variables*, i.e., for any rule $(f(\overline{t}) \to e \Leftarrow \overline{C}) \in \mathcal{P}$ all the variables appearing in e and $\overline{C}$ must also appear in the head $f(\overline{t})$, i.e., $var(e) \cup \mathcal{V}(\overline{C}) \subseteq var(\overline{t})$. In *FLP* with non-deterministic functions this is not *as* restrictive as it could appear: function nesting can replace the use (typical in logic programming) of variables as repositories of intermediate values, and in many other cases where extra variables represent unknown values to be computed by search, they can be successfully replaced by non-deterministic 'lazy generating' functions (see [3] for some examples).

We will frequently use the following notation: given $e \in Term_{\perp,\mathsf{F}}$, $\hat{e}$ stands for the result of replacing by $\perp$ all the occurrences of F in e (notice that $\hat{e} \in Term_{\perp}$, and $e = \hat{e}$ iff $e \in Term_{\perp}$).

3.3 The Proof Calculus for *CRWLF*

In *CRWLF* five kinds of statements can be deduced:

- $e \lhd \mathcal{C}$, intended to mean '$\mathcal{C}$ *is an SAS for* e'.
- $e \bowtie e'$, $e \diamond\!\!> e'$, with the same intended meaning as in *CRWL*.
- $e \not\bowtie e'$, $e \not<\!\!\diamond e'$, intended to mean failure of $e \bowtie e'$ and $e \diamond\!\!> e'$ respectively.

We will sometimes speak of $\bowtie, \diamond\!\!>, \not\bowtie, \not<\!\!\diamond$ as 'constraints', and use the symbol $\diamondsuit$ to refer to any of them. The constraints $\not\bowtie$ and $\bowtie$ are called the *complementary* of each other; the same holds for $\not<\!\!\diamond$ and $\diamond\!\!>$, and we write $\widetilde{\diamondsuit}$ for the complementary of $\diamondsuit$.

When proving a constraint $e \diamondsuit e'$ the calculus *CRWLF* will evaluate an *SAS* for the expressions e and e'. These *SAS*'s will consist of c-terms from $CTerm_{\perp,\mathsf{F}}$, and provability of the constraint $e \diamondsuit e'$ depends on certain syntactic (hence decidable) relations between those c-terms. Actually, the constraints $\bowtie$, $\diamond\!\!>$, $\not\bowtie$ and $\not<\!\!\diamond$ can be seen as the result of generalizing to expressions the relations $\downarrow, \uparrow, \not\downarrow$ and $\not\gamma$ on c-terms, which we define now.

Definition 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*
- $\not\gamma$ *is defined as the least symmetric relation over* $CTerm_{\perp,\mathsf{F}}$ *satisfying:*
  - *i)* $X \not\gamma X$, *for all* $X \in \mathcal{V}$
  - *ii)* $\mathsf{F} \not\gamma t$, *for all* $t \in CTerm_{\perp,\mathsf{F}}$
  - *iii) if* $t_1 \not\gamma t'_1, ..., t_n \not\gamma t'_n$ *then* $c(t_1, ..., t_n) \not\gamma c(t'_1, ..., t'_n)$ *for all* $c \in DC^n$

The relations $\downarrow$ and $\uparrow$ do not take into account the presence of F, which behaves in this case as $\perp$. The relation $\downarrow$ is *strict* equality, i.e., equality restricted to total c-terms. It is the notion of equality used in lazy functional or functional-logic languages as the suitable approximation to 'true' equality ($=$) over $CTerm_{\perp}$. The relation $\uparrow$ is a suitable approximation to '$\neg =$', and hence to '$\neg \downarrow$' (where $\neg$ stands for logical negation). The relation $\not\downarrow$ is also an approximation to '$\neg \downarrow$', but in this case using failure information ($\not\downarrow$ can be read as '$\downarrow$ fails'). Notice that $\not\downarrow$ does not imply '$\neg =$' anymore (we have, for instance, $\mathsf{F} \not\downarrow \mathsf{F}$). Similarly, $\not\gamma$ is also an approximation to '$\neg \uparrow$' which can be read as '$\uparrow$ fails'.

The following proposition reflects these and more good properties of $\downarrow, \uparrow, \not\downarrow, \not\gamma$.

Proposition 1. *The relations* $\downarrow, \uparrow, \not\downarrow, \not\gamma$ *verify*

(a) For all $t, t', s, s' \in CTerm_{\perp,\mathsf{F}}$
  - *(i)* $t \downarrow t' \Leftrightarrow \hat{t} \downarrow \hat{t}'$ *and* $t \uparrow t' \Leftrightarrow \hat{t} \uparrow \hat{t}'$
  - *(ii)* $t \uparrow t' \Rightarrow t \not\downarrow t' \Rightarrow \neg(t \downarrow t')$
  - *(iii)* $t \downarrow t' \Rightarrow t \not\gamma t' \Rightarrow \neg(t \uparrow t')$

(b) $\downarrow, \uparrow, \not\downarrow, \not\gamma$ *are monotonic, i.e., if* $t \sqsubseteq s$ *and* $t' \sqsubseteq s'$ *then:* $t \Re t' \Rightarrow s \Re s'$, *where* $\Re \in \{\downarrow, \uparrow, \not\downarrow, \not\gamma\}$. *Furthermore* $\not\downarrow_B$ *and* $\not\gamma_B$ *are the greatest monotonic approximations to* $\neg \downarrow_B$ *and* $\neg \uparrow_B$ *respectively, where* $\Re_B$ *is the restriction of* $\Re$ *to the set of basic (i.e., without variables) c-terms from* $CTerm_{\perp,\mathsf{F}}$.

(c) $\downarrow$ *and* $\not\gamma$ *are closed under substitutions from* $CSubst$; $\not\downarrow$ *and* $\uparrow$ *are closed under substitutions from* $CSubst_{\perp,\mathsf{F}}$

By *(b)*, we can say that $\downarrow, \uparrow, \not\downarrow, \not\uparrow$ behave well with respect to the information ordering: if they are true for some terms, they remain true if we refine the information contained in the terms. Furthermore, *(b)* states that $\not\downarrow, \not\uparrow$ are defined in the best way, at least for basic c-terms. For c-terms with variables, we must take care: for instance, given the constructor z, we have $\neg(X \downarrow z)$, but not $X \not\downarrow z$. Actually, to have $X \not\downarrow z$ would violate a basic intuition about free variables in logical statements: if the statement is true, it should be true for any value (taken from an appropriate range) substituted for its free variables. The part *(c)* shows that the definitions of $\downarrow, \uparrow, \not\downarrow, \not\uparrow$ respect such principle. Propositions 2 and 3 of the next section show that monotonicity and closedness by substitutions are preserved when generalizing $\downarrow, \uparrow, \not\downarrow, \not\uparrow$ to $\bowtie, \diamond, \not\bowtie, \not\diamond$.

Table 2 contains the *CRWLF*-calculus. Some of the rules use a generalized notion of c-instances of a rule R: $[R]_{\bot,\mathsf{F}} = \{R\theta \mid \theta \in CSubst_{\bot,\mathsf{F}}\}$. We will use the notation $\mathcal{P} \vdash_{CRWLF} \varphi$ ($\mathcal{P} \nvdash_{CRWLF} \varphi$ resp.) for expressing that the statement φ is provable (is not provable resp.) with respect to the calculus *CRWLF* and the program $\mathcal{P}$.

The first three rules are analogous to those of the *CRWL*-calculus, now dealing with *SAS*'s instead of simple approximations (notice the cross product of *SAS*'s in rule 3). In rule 4, for evaluating an expression $f(\overline{e})$ we produce *SAS*'s for the arguments e_i and then, for each combination of values in these *SAS*'s and each program rule for f, a part of the whole *SAS* is produced; all of them are unioned to obtain the final *SAS* for $f(\overline{e})$. This is quite different from rule 4 in *CRWL*: there we could use any c-instance of any rule for f; here we need to consider simultaneously the contribution of each rule to achieve 'complete' information about the values to which the expression can be evaluated. We use the notation $f(\overline{t}) \lhd_R \mathcal{C}$ to indicate that only the rule R is used to produce $\mathcal{C}$.

Rules 5 to 8 consider all the possible ways in which a concrete rule R can contribute to the *SAS* of a call $f(\overline{t})$, where the arguments $\overline{t}$ are all in $CTerm_{\bot,\mathsf{F}}$ (come from the evaluation of the arguments of a previous call $f(\overline{e})$). Rules 5 and 6 can be viewed as *positive* contributions. The first one obtains the trivial *SAS* and 6 works if there is a c-instance of the rule R with a head identical to the head of the call (parameter passing); in this case, if the constraints of this c-instance are provable, then the resulting *SAS* is generated by the body of the c-instance. Rules 7 and 8 consider the *negative* or *failed* contributions. Rule 7 applies when parameter passing can be done, but it is possible to prove the complementary $e_i \tilde{\diamond} e'_i$ of one of the constraints $e_i \diamond e'_i$ in the condition of the used c-instance. In this case the constraint $e_i \diamond e'_i$ (hence the whole condition in the c-instance) fails. Finally, rule 8 considers the case in which parameter passing fails because of a $DC \cup \{\mathsf{F}\}$-clash between one of the arguments in the call and the corresponding pattern in R.

We remark that for given $f(\overline{t})$ and R, the rule 5 and exactly one of rules 6 to 8 are applicable. This fact, although intuitive, is far from being trivial to prove and constitutes in fact an important technical detail in the proofs of the results in the next section. We also remark that, for the sake of a better reading of rule 4, we have written ordinary set union for collecting *SAS*'s. This could

Table 2. Rules for *CRWLF*-provability

(1) $\dfrac{}{e \lhd \{\bot\}}$

(2) $\dfrac{}{X \lhd \{X\}} \qquad X \in \mathcal{V}$

(3) $\dfrac{e_1 \lhd \mathcal{C}_1 \quad ... \quad e_n \lhd \mathcal{C}_n}{c(e_1,...,e_n) \lhd \{c(t_1,...,t_n) \mid \overline{t} \in \mathcal{C}_1 \times ... \times \mathcal{C}_n\}} \qquad c \in DC^n \cup \{\mathsf{F}\}$

(4) $\dfrac{e_1 \lhd \mathcal{C}_1 \quad ... \quad e_n \lhd \mathcal{C}_n \quad ... \quad f(\overline{t}) \lhd_R \mathcal{C}_{R,\overline{t}} \quad ...}{f(e_1,...,e_n) \lhd \bigcup_{R \in \mathcal{P}_f, \overline{t} \in \mathcal{C}_1 \times ... \times \mathcal{C}_n} \mathcal{C}_{R,\overline{t}}} \qquad f \in FS^n$

(5) $\dfrac{}{f(\overline{t}) \lhd_R \{\bot\}}$

(6) $\dfrac{e \lhd \mathcal{C} \quad \overline{C}}{f(\overline{t}) \lhd_R \mathcal{C}} \qquad (f(\overline{t}) \to e \Leftarrow \overline{C}) \in [R]_{\bot,\mathsf{F}}$

(7) $\dfrac{e_i \tilde{\Diamond} e_i'}{f(\overline{t}) \lhd_R \{F\}} \qquad (f(\overline{t}) \to e \Leftarrow ..., e_i \Diamond e_i', ...) \in [R]_{\bot,\mathsf{F}}$, where $i \in \{1,...,n\}$

(8) $\dfrac{}{f(t_1,...,t_n) \lhd_R \{F\}}$ $\quad R \equiv (f(s_1,...,s_n) \to e \Leftarrow \overline{C}), t_i$ and s_i have a $DC \cup \{\mathsf{F}\}$-clash for some $i \in \{1,...,n\}$

(9) $\dfrac{e \lhd \mathcal{C} \quad e' \lhd \mathcal{C}'}{e \bowtie e'} \qquad \exists t \in \mathcal{C}, t' \in \mathcal{C}' \; t \downarrow t'$

(10) $\dfrac{e \lhd \mathcal{C} \quad e' \lhd \mathcal{C}'}{e \diamond e'} \qquad \exists t \in \mathcal{C}, t' \in \mathcal{C}' \; t \uparrow t'$

(11) $\dfrac{e \lhd \mathcal{C} \quad e' \lhd \mathcal{C}'}{e \not\bowtie e'} \qquad \forall t \in \mathcal{C}, t' \in \mathcal{C}' \; t \not\downarrow t'$

(12) $\dfrac{e \lhd \mathcal{C} \quad e' \lhd \mathcal{C}'}{e \not\diamond e'} \qquad \forall t \in \mathcal{C}, t' \in \mathcal{C}' \; t \not\uparrow t'$

be modified in such a way that F is excluded from the union if it contains some other c-term different from F. For example, if we obtain the *SAS*'s $\{z\}$ and $\{\mathsf{F}\}$ from two function rules, we could take $\{z\}$ as the final *SAS* for the call instead of $\{\mathsf{F}, z\}$. All the results of the next section are valid with this modification.

Rules 9 to 12 deal with constraints. With the use of the relations $\downarrow, \uparrow, \not\downarrow, \not\uparrow$ introduced in Sect. 3.3 the rules are easy to formulate. For $e \bowtie e'$ it is sufficient to find two c-terms in the *SAS*'s verifying the relation $\downarrow$, what in fact is equivalent to find a common totally defined c-term such that both expressions e and e' can be reduced to it (observe the analogy with rule 5 of *CRWL*). For the complementary constraint $\not\bowtie$ we need to use all the information of *SAS*'s in order to check the relation $\not\downarrow$ over all the possible pairs. The explanation of rules 11 and 12 is quite similar.

The next example shows a derivation of failure using the *CRWLF*-calculus.

Example 1. Let us consider a program $\mathcal{P}$ with the constructors z, s for natural numbers, [] and ':' for lists (although we use Prolog-like notation for them, that is, $[z, s(z)|L]$ represents the list $(z : (s(z) : L))$) and also the constructors $\mathtt{t}, \mathtt{f}$ that represent the boolean values *true* and *false*. Assume the functions *coin* and h defined in Sect. 2.2 and also the function mb (member) defined as:

$$mb(X, [Y|\, Ys]) \to \mathtt{t} \Leftarrow X \bowtie Y$$
$$mb(X, [Y|\, Ys]) \to \mathtt{t} \Leftarrow mb(X,\, Ys) \bowtie \mathtt{t}$$

If we try to evaluate the expression $mb(coin, [s(h)])$ it will fail. Intuitively, from definition of h the list in the second argument can be reduced to lists of the form $[s(s(...))]$ and the possible values of *coin*, z and $s(z)$, will never belong to those lists. The *CRWLF*-calculus allows to build a proof for this fact, that is, $mb(coin, [s(h)]) \lhd \{\mathtt{F}\}$, in the following way: by application of rule 4 the proof could proceed by generating *SAS*'s for the arguments

$$coin \lhd \{z, s(z)\} \quad (\varphi_1) \qquad [s(h)] \lhd \{[s(s(\bot))]\} \quad (\varphi_2)$$

and then collecting the contributions of rules of mb for each possible combination of values for the arguments; for the pair $(z, [s(s(\bot))])$ the contribution of rules for mb (here we write $\lhd_1$ to refer to the first rule of mb and $\lhd_2$ for the second) will be

$$mb(z, [s(s(\bot))]) \lhd_1 \{\mathtt{F}\} \quad (\varphi_3) \qquad mb(z, [s(s(\bot))]) \lhd_2 \{\mathtt{F}\} \quad (\varphi_4)$$

and for the pair $(s(z), [s(s(\bot)])$ we will have

$$mb(s(z), [s(s(\bot))]) \lhd_1 \{\mathtt{F}\} \quad (\varphi_5) \qquad mb(s(z), [s(s(\bot))]) \lhd_2 \{\mathtt{F}\} \quad (\varphi_6)$$

The following derivation shows the form of the full derivation, but we only give details of the proofs for φ_3 and φ_4. At each step, we indicate by a number on the left the rule of the calculus applied in each case:

$$\dfrac{\varphi_1 \;\; \varphi_2 \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\bot \lhd \{\bot\}}\,1}{s(\bot) \lhd \{s(\bot)\}}\,3}{z \lhd \{z\}}\,3 \quad \dfrac{\dfrac{\dfrac{\dfrac{}{s(\bot) \lhd \{s(\bot)\}}\,3}{z \not\bowtie s(s(\bot))}\,11}{\varphi_3 \equiv mb(z, [s(s(\bot))]) \lhd_1 \{\mathtt{F}\}}\,7 \quad \dfrac{\dfrac{\dfrac{\dfrac{}{z \lhd \{z\}}\,3 \quad \dfrac{}{[] \lhd \{[]\}}\,3 \quad \dfrac{}{mb(z, []) \lhd_{12} \{\mathtt{F}\}}\,8}{mb(z, []) \lhd \{\mathtt{F}\}}\,4 \quad \dfrac{}{\mathtt{t} \lhd \{\mathtt{t}\}}\,3}{mb(z, []) \not\bowtie \{\mathtt{t}\}}\,11}{\varphi_4 \equiv mb(z, [s(s(\bot))]) \lhd_2 \{\mathtt{F}\}}\,7 \quad \varphi_5 \;\; \varphi_6}{mb(coin, [s(h)]) \lhd \{\mathtt{F}\}}\,4$$

In both φ_3 and φ_4 the failure is due to a failure in the constraints of rules, what requires to prove the complementary constraint $\not\bowtie$ by rule 11. In the first case, $z \not\bowtie s(s(\bot))$, there is a clear clash of constructors. But in the second case it involves the failure for the expression $mb(z, [])$ that is proved again by rule 4 of the calculus. The *SAS*'s for the arguments only produce the combination $(z, [])$ and both rules of mb fails over it by rule 8 of the calculus. The notation $mb(z, []) \lhd_{1,2} \{\mathtt{F}\}$ which appears on the top of the proof of φ_4 is an abbreviation for both statements $mb(z, []) \lhd_1 \{\mathtt{F}\}$ and $mb(z, []) \lhd_2 \{\mathtt{F}\}$.

All the contributions of $\varphi_3, \varphi_4, \varphi_5$ and φ_6 are $\{\mathtt{F}\}$, and putting them together we obtain $\{\mathtt{F}\}$ as an *SAS* for the original expression $mb(coin, [s(h)])$ as expected.

4 Properties of *CRWLF*

In this section we explore some properties of the *CRWLF*-calculus and its relation with *CRWL*. In the following we assume a fixed program $\mathcal{P}$.

The non-determinism of the *CRWLF*-calculus allows to obtain different *SAS*'s for the same expression. As the *SAS* for an expression is a finite approximation to the denotation of the expression it is expected some kind of consistency between *SAS*'s for the same expression. Given two of them, we cannot ensure that one *SAS* must be more defined than the other in the sense that all the elements of the first are more defined than all of the second. For instance, two *SAS*'s for *coin* are $\{\bot, s(z)\}$ and $\{z, \bot\}$. The kind of consistency for *SAS*'s that we can expect is the following:

Definition 2 (Consistent Sets of c-terms). *Two sets* $\mathcal{C}, \mathcal{C}' \subset CTerm_{\bot,\mathcal{F}}$ *are* consistent *iff for all* $t \in \mathcal{C}$ *there exists* $t' \in \mathcal{C}'$ *(and vice versa, for all* $t' \in \mathcal{C}'$ *there exists* $t \in \mathcal{C}$*) such that* t *and* t' *are consistent.*

Our first result states that two different *SAS*'s for the same expression must be consistent.

Theorem 1 (Consistency of *SAS*). *Given* $e \in Term_{\bot,\mathcal{F}}$*, if* $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C}$ *and* $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C}'$*, then* $\mathcal{C}$ *and* $\mathcal{C}'$ *are consistent.*

This result is a trivial corollary of part *a*) of the following lemma.

Lemma 1 (Consistency). *For any* $e, e', e_1, e_2, e_1', e_2' \in Term_{\bot,\mathcal{F}}$

a) *If* e, e' *are consistent,* $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C}$ *and* $\mathcal{P} \vdash_{CRWLF} e' \lhd \mathcal{C}'$*, then* $\mathcal{C}$ *and* $\mathcal{C}'$ *are consistent.*
b) *If* e_1, e_1' *are consistent and* e_2, e_2' *are also consistent, then:* $\mathcal{P} \vdash_{CRWLF} e_1 \Diamond e_2 \Rightarrow \mathcal{P} \nvdash_{CRWLF} e_1' \tilde{\Diamond} e_2'$

As a trivial consequence of part *b*) we have:

Corollary 1. $\mathcal{P} \vdash_{CRWLF} e \Diamond e' \Rightarrow \mathcal{P} \nvdash_{CRWLF} e \tilde{\Diamond} e'$*, for all* $e, e' \in Term_{\bot,\mathcal{F}}$

This supports our original idea about $\not\bowtie$ and $\not<>$ as computable approximations to the negations of $\bowtie$ and $<>$.

Another desirable property of our calculus is *monotonicity*, that we can informally understand in this way: the information that can be extracted from an expression can not decrease when we add information to the expression itself. This also reflects in the fact that if we can prove a constraint and we consider more defined terms in both sides of it, the resulting constraint must be also provable. Formally:

Proposition 2 (Monotonicity of *CRWLF*). *For* $e, e', e_1, e_2, e_1', e_2' \in Term_{\bot,\mathcal{F}}$

a) *If* $e \sqsubseteq e'$ *and* $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C}$*, then* $\mathcal{P} \vdash_{CRWLF} e' \lhd \mathcal{C}$
b) *If* $e_1 \sqsubseteq e_1'$*,* $e_2 \sqsubseteq e_2'$ *and* $\mathcal{P} \vdash_{CRWLF} e_1 \Diamond e_2$ *then* $\mathcal{P} \vdash_{CRWLF} e_1' \Diamond e_2'$*, where* $\Diamond \in \{\bowtie, \not\bowtie, <>, \not<>\}$

Monotonicity, as stated here, refers to the degree of evaluation of expression, and does not contradict the well known fact that negation as failure is a non-monotonic reasoning rule. In our setting it is also clearly true that, if we 'define more' the functions (i.e, we refine the program, not the evaluation of a given expression), an expression can become reducible when it was previously failed.

The next property says that what is true for free variables is also true for any possible (totally defined) value, i.e., provability in *CRWLF* is closed under total substitutions.

Proposition 3. *For any* $\theta \in CSubst$, $e, e' \in Term_{\perp,\mathsf{F}}$

a) $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C} \Rightarrow \mathcal{P} \vdash_{CRWLF} e\theta \lhd \mathcal{C}\theta$
b) $\mathcal{P} \vdash_{CRWLF} e \diamondsuit e' \Rightarrow \mathcal{P} \vdash_{CRWLF} e\theta \diamondsuit e'\theta$

4.1 *CRWLF* related to *CRWL*

The *CRWLF*-calculus has been built as an extension of *CRWL* for dealing with failure. Here we show that our aims have been achieved.

We recall that a *CRWLF*-program is a *CRWL*-program not containing extra variables in rules. The following results are all referred to *CRWLF*-programs.

The next result shows that the *CRWLF*-calculus indeed extends *CRWL*. Parts *a)* and *b)* show that statements $e \lhd \mathcal{C}$ generalize approximation statements $e \to t$ of *CRWL*. Parts *c)* and *d)* show that *CRWLF* and *CRWL* are able to prove exactly the same joinabilities and divergences (if F is ignored for the comparison).

Proposition 4. *For any* $e, e' \in Term_{\perp,\mathsf{F}}$

a) $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C} \Rightarrow \forall t \in \mathcal{C}, \mathcal{P} \vdash_{CRWL} \hat{e} \to \hat{t}$
b) $\mathcal{P} \vdash_{CRWL} \hat{e} \to t \Rightarrow \exists \mathcal{C}$ *such that* $t \in \mathcal{C}$ *and* $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C}$
c) $\mathcal{P} \vdash_{CRWLF} e \bowtie e' \Leftrightarrow \mathcal{P} \vdash_{CRWL} \hat{e} \bowtie \hat{e}'$
d) $\mathcal{P} \vdash_{CRWLF} e \mathrel{<>} e' \Leftrightarrow \mathcal{P} \vdash_{CRWL} \hat{e} \mathrel{<>} \hat{e}'$

We can revise within *CRWLF* the notion of denotation of an expression, and define $[\![e]\!]^F = \{t \in CTerm_{\perp,\mathsf{F}} \mid e \lhd \mathcal{C}, t \in \mathcal{C}\}$, for any $e \in Term_{\perp,\mathsf{F}}$. As a consequence of the previous proposition we have $[\![e]\!] \subseteq [\![e]\!]^F$ for any $e \in Term_{\perp}$ and $\widehat{[\![e]\!]^F} = [\![\hat{e}]\!]$ for any $e \in Term_{\perp,\mathsf{F}}$, where, given a set S, $\hat{S}$ is defined in the natural way $\hat{S} = \{\hat{t} \mid t \in S\}$.

All the previous results make easy the task of proving that we have done things right with respect to failure. We will need a result stronger than Prop. 4, which does not provide enough information about the relation between the denotation of an expression and each of its calculable *SAS*'s.

Proposition 5. *Given* $e \in Term_{\perp,\mathsf{F}}$, *if* $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C}$ *and* $t \in [\![\hat{e}]\!]$, *then there exists* $s \in \mathcal{C}$ *such that* s *and* t *are consistent.*

Proof. Assume $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C}$. If we take $t \in CTerm_{\perp}$ such that $\mathcal{P} \vdash_{CRWL} \hat{e} \to t$, then by part *b*) of Prop. 4 there exists $\mathcal{C}'$ such that $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C}'$ with $t \in \mathcal{C}'$.

By Theorem 1 it follows that $\mathcal{C}$ and $\mathcal{C}'$ are consistent. By definition of consistent *SAS*'s, as $t \in \mathcal{C}'$, then there exist $s \in \mathcal{C}$ such that t and s are consistent. □

We easily arrive now at our final result.

Theorem 2. *Given* $e \in Term_{\perp,\mathsf{F}}$, *if* $\mathcal{P} \vdash_{CRWLF} e \lhd \{\mathsf{F}\}$ *then* $[\![\hat{e}]\!] = \{\perp\}$

Proof. If $t \in [\![\hat{e}]\!]$, we know from Prop. 5 that F and t must be consistent. As F is consistent only with $\perp$ and itself, and $t \in CTerm_{\perp}$, we conclude that $t = \perp$. □

5 Conclusions and Future Work

We have proposed the proof calculus *CRWLF* (Constructor based ReWriting Logic with Failure), which allows to deduce negative information from a wide class of functional logic programs. In particular, the calculus provides proofs of failure of reduction, a notion that can be seen as the natural *FLP* counterpart of negation as failure in logic programming.

The starting point for *CRWLF* has been the proof calculus of *CRWL* [3], a well established theoretical framework for *FLP*. The most remarkable insight has been to replace the statements $e \to t$ of *CRWL* (representing a single reduction of e to an approximated value t) by $e \lhd \mathcal{C}$ (representing a whole, somehow complete, set $\mathcal{C}$ of approximations to e). With the aid of $\lhd$ we have been able to cover all the derivations in *CRWL*, as well as to prove failure of reduction and, as auxiliary notions, failure of joinability and divergence, the two other kinds of statements that *CRWL* was able to prove.

It is interesting to remark that $\lhd$ provide, at the level of logical descriptions, a finer control over reduction than $\to$. Two examples: $e \lhd \{t\}, t \in CTerm$ expresses the property that e is reducible to the unique totally defined value t; $e \lhd \mathcal{C}, e' \lhd \mathcal{C}$, with $\mathcal{C}$ consisting only of total c-terms, expresses that e and e' reduce to exactly the same (totally defined) values. The same properties, if expressed by means of $\to$, would require the use of universal quantification, which is out of the scope of *CRWL*. Observe that, although the side conditions '$t \in CTerm$' and '$\mathcal{C}$ *consisting only of total c-terms*' of the examples are not statements of *CRWLF*, they are purely syntactical conditions.

The idea of collecting into an *SAS* values coming from different reductions for a given expression e presents some similarities with abstract interpretation which, within the *FLP* field, has been used in [2] for detecting unsatisfiability of equations $e = e'$ (something similar to failure of our $e \bowtie e'$). We can mention some differences between our work and [2]:

- Programs in [2] are much more restrictive: they must be confluent, terminating, satisfy a property of stratification on conditions, and define strict and total functions.
- In our setting, each *SAS* for an expression e consists of (down) approximations to the denotation of e, and the set of *SAS*'s for e determines in a precise sense (Props. 4 and 5) the *exact* denotation of e. In the abstract interpretation approach one typically obtains, for an expression e, an abstract term representing a *superset* of the denotation of all the instances of e. But some of the rules of the *CRWLF*-calculus (like (9) or (10)) are not valid if we replace SAS's by such supersets. To be more concrete, if we adopt an abstract

interpretation view of our *SAS*'s, it would be natural to see $\bot$ as standing for the set of all constructor terms (since $\bot$ is refinable to any value), and therefore to identify an *SAS* like $\mathcal{C} = \{\bot, z\}$ with $\mathcal{C}' = \{\bot\}$. But from $e \lhd \mathcal{C}$ we can deduce $e \bowtie z$, while it is not correct to do the same from $e \lhd \mathcal{C}'$. Therefore, the good properties of *CRWLF* with respect to *CRWL* are lost.

We see our work as the first step in the research of a whole framework for dealing with failure in *FLP*. Some natural (but not small!) future steps are: to enlarge the class of considered programs by allowing extra variables; to consider 'general' programs which make use of failure information, and to develop model theoretic and operational semantics for them.

Acknowledgements: We thank the anonymous referees for their useful comments.

References

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

[2] D. Bert and R. Echahed. Abstraction of conditional term rewriting systems. In *Proc. ILPS'95*, pages 162–176. MIT Press, 1995.

[3] 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.

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

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

[6] G. Jäger and R.F. Stärk. A proof-theoretic framework for logic programming. In S.R. Buss (ed.), *Handbook of Proof Theory*, pages 639–682. Elsevier, 1998.

[7] F.J. López-Fraguas and J. Sánchez-Hernández. Disequalities may help to narrow. In *Proc. APPIA-GULP-PRODE'99*, pages 89–104, 1999.

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

[9] F.J. López-Fraguas and J. Sánchez-Hernández. Proving failure in functional logic programs (extended version). Tech. Rep. SIP 00/100-00, UCM Madrid, 2000.

[10] J.J. Moreno-Navarro. Default rules: An extension of constructive negation for narrowing-based languages. In *Proc. ICLP'95*, pages 535–549. MIT Press, 1994.

[11] 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.

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