TLP **4** (1 & 2): 41–74, 2004. © Cambridge University Press
DOI: 10.1017/S1471068403001728 Printed in the United Kingdom

A proof theoretic approach to failure in functional logic programming

FRANCISCO J. LÓPEZ-FRAGUAS and JAIME SÁNCHEZ-HERNÁNDEZ
Departamento Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Madrid, Spain
(*e-mail:* {`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*.

KEYWORDS: constructive failure, functional logic programming, proof calculi

1 Introduction

In this paper we address 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* (Clark 1978), as an easy effective approximation to the Closed World Assumption (CWA), which is a simple, but uncomputable, way of deducing negative information from positive programs (e.g. see Apt and Bol (1994) for a survey on negation in logic programming).

On the other hand, Functional Logic Programming (FLP) is a powerful programming paradigm trying to combine the nicest properties of functional and logic programming (see Hanus (194) for a now 'classical' survey on FLP). A mainstream in current *FLP* research considers languages which are biased to the functional programming style, in the sense that programs define functions, but having logic programming capabilities because their operational mechanisms are based on narrowing. Some existing systems of this kind are $\mathcal{TOY}$ (López and Sánchez 1999a; Abengózar *et al.* 2002) or the various implementations of *Curry* (Hanus 2000). In the rest of the paper, we have such an approach in mind when we refer to 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, as we see in the next section, 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 to (partial) data constructor value', or in other terms, 'failure of reduction to head normal form' (hnf).

As technical setting for our work we have chosen *CRWL* (González *et al.* 1996; González *et al.* 1999), 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. Instead of equational logic, which is argued to be unsuitable for FLP in González *et al.* (1999), *CRWL* considers a Constructor based ReWriting Logic, presented by means of a proof calculus, which determines what statements can be deduced from a given program. In addition to the proof-theoretic semantics, González *et al.* (1996, 1999) 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 interest of *CRWL* as a theoretical framework for FLP has been mentioned in Hanus (2000), and is further evidenced by its many extensions incorporating relevant aspects of declarative programming like HO features (González *et al.* 1997), polymorphic and algebraic types (Arenas and Rodríguez 2001), or constraints (Arenas *et al.* 1999). The framework, with many of these extensions (like types, HO and constraints) has been implemented in the system $\mathcal{TOY}$.

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 Moreno (1994), the work of Stuckey about *constructive negation* (Stuckey, 1991, 1995) is adapted to the case of FLP with strict functions and innermost narrowing as operational mechanism. In Moreno (1996), 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 Jäger and Stärk (1998), which develops for logic programs (with negation) a framework which resembles, in a very general sense, *CRWL*: a program determines 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 when writing programs.

The rest of the paper is organized as follows. In section 2 we discuss the interest of using failure as a programming construct in the context of FLP. In section 3 we give the essentials of *CRWL* which are needed for our work. Section 3 presents the *CRWLF*-calculus, preceded by some illustrative examples. Sections 4–6 constitute the technical core of the paper, presenting the properties of *CRWLF* and its relation to *CRWL*. Finally, section 7 outlines some conclusions and possible future work.

2 The interest of failure in FLP

Although this work is devoted only to the theoretical aspects of failure in FLP, in this section we argue some possible applications of this resource from the point of view of writing functional logic programs.

FLP combines some of the main capabilities of the two main streams of declarative programming: Functional Programming (FP) and Logic Programming (LP). Theoretical aspects of FLP are well established (e.g. see González *et al.* (1999)), and there are also practical implementations such as *Curry* or $\mathcal{TOY}$. Disregarding syntax, both pure Prolog and (a wide subset of) Haskell are subsumed by those systems. The usual claim is then that by the use of a FLP system one can choose the style of programming better suited to each occasion.

However there are features related to failure, mainly in LP (but also in FP) yet not available in FLP systems. This poses some problems to FLP: if a logic program uses negation (a very common situation), it cannot be seen as a FLP program. This is not a very serious inconvenience if other features of FLP could easily replace the use of failure. But if the FLP solution (without failure) to a problem is significantly more complex than, say, a LP solution making use of failure, then it is not worth to use FLP for that problem, thus contradicting in practice the claim that FLP can successfully replace LP and FP.

We now give concrete examples of the potential use of a construction to express failure in FLP programs. We assume for the examples below that we incorporate to FLP the following function to express failure of an expression:

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

The sensible notion to consider is *failure of reduction to head normal form*[1], since head normal forms (i.e. variables or expressions $c(\ldots)$, where c is a constructor symbol) are the expressions representing, without the need of further reduction, defined (maybe partial) values.

Example 1 (*Failure to express negation in LP*)
The most widespread approach to negation in the LP paradigm is *negation as failure* (Clark 1978), of which all PROLOG systems provide an implementation. Typically, in a logic program one writes clauses defining the positive cases for a predicate, and

[1] To be technically more precise, we should speak of 'failure to reduction to head normal form with respect to the *CRWL*-calculus', to be recalled in section 3.

the effect of using negation is to 'complete' the definition with the negative cases, which correspond to failure of the given clauses.

For example, in LP the predicate *member* can be defined as:

$$\begin{array}{l} member(X,[X|Ys]). \\ member(X,[Y|Ys]) \leftarrow member(X,Ys). \end{array}$$

This defines *member(X,L)* as a semidecision procedure to check if X is an element of L. If one needs to check that X is not an element of L, then negation can be used, as in the clause

$$add(X,L,[X|L]) :- not\ member(X,L).$$

Predicates like *member* can be defined in FLP as *true*-valued functions, converting clauses into conditional rules returning *true*:

$$\begin{array}{l} member(X,[Y|Ys]) \rightarrow true \Leftarrow X \bowtie Y \\ member(X,[Y|Ys]) \rightarrow true \Leftarrow member(X,Ys) \bowtie true \end{array}$$

To achieve linearity (i.e. no variable repetition) of heads, a usual requirement in FLP, the condition $X \bowtie Y$ is used in the first rule. The symbol $\bowtie$ (taken from González *et al.* (1996, 1999)) is used throughout the paper to express 'joinability', which means that both sides can be reduced to the same data value (for the purpose of this example, $\bowtie$ can be read simply as strict equality).

What cannot be directly translated into FLP (without failure) is a clause like that of *add*, but with failure it is immediate:

$$add(X,L,[X'|L']) \rightarrow true \Leftarrow fails(member(X,L)) \bowtie true, X' \bowtie X, L' \bowtie L$$

In general, any literal of the form *not Goal* in a logic program can be replaced by $fails(Goal) \bowtie true$ in its FLP-translation.

This serves to argue that FLP with failure subsumes LP with negation, but of course this concrete example corresponds to the category of 'dispensable' uses of failure, because there is a natural failure-free FLP counterpart to the predicate *member* in the form of a bivaluated boolean function, where the failure is expressed by the value *false*. The following could be such a definition of *member*:

$$\begin{array}{lcl} member(X,[\,]) & \rightarrow & false \\ member(X,[Y|Ys]) & \rightarrow & true \Leftarrow X \bowtie Y \\ member(X,[Y|Ys]) & \rightarrow & member(X,Ys) \Leftarrow X <> Y \end{array}$$

The symbol $<>$ (corresponding to disequality $\neq$ of (López and Sánchez 1999a; López and Sánchez 1999b)) expresses 'divergence', meaning that both sides can be reduced to some extent as to detect inconsistency, i.e. conflict of constructors at the same position (outside function applications). Now *add* can be easily defined without using failure:

$$add(X,L,[X'|L']) \rightarrow true \Leftarrow member(X,L) \bowtie false, X' \bowtie X, L' \bowtie L$$

The next examples show situations where the use of negation is more 'essential', in the sense that it is the natural way (at least a very natural way) of doing things.

Example 2 (*Failure in search problems I*)
Non-deterministic constructs are a useful way of programming problems involving search. In FLP one can choose to use predicates, as in LP, or non-deterministic functions. In these cases, the use of failure can greatly simplify the task of programming. We see an example with non-deterministic functions, a quite specific FLP feature which is known to be useful for programming (Abengózar *et al.* 2002; Hanus 2000; Antoy 1997) in systems like *Curry* or $\mathcal{TOY}$.

Consider the problem of deciding, for acyclic directed graphs, if there is a path connecting two nodes. A graph can be represented by a non-deterministic function *next*, with rules of the form $next(N) \rightarrow N'$, indicating that there is an arc from N to N'. A concrete graph with nodes a, b, c and d could be given by the rules:

$$\begin{array}{l} next(a) \rightarrow b \\ next(a) \rightarrow c \\ next(b) \rightarrow c \\ next(b) \rightarrow d \end{array}$$

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

$$\begin{array}{l} path(X, Y) \rightarrow true \Leftarrow X \bowtie Y \\ path(X, Y) \rightarrow true \Leftarrow X <> Y,\ path(next(X), Y) \bowtie true \end{array}$$

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

$$safe(X) ::= X \text{ is not connected with } d$$

Using failure this is an easy task:

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

With this definition, $safe(c)$ becomes *true*, while $safe(a)$, $safe(b)$ and $safe(d)$ are all *false*.

Example 3 (*Failure in search problems II*)
We examine now an example mentioned in Apt (2000) as one striking illustration of the power of failure as expressive resource in LP. We want to program a two-person finite game where the players must perform alternate legal moves, until one of them, the loser, cannot move.

We assume that legal moves from a given state are programmed by a non-deterministic function *move(State)* returning the new state after the movement. Using failure it is easy to program a function to perform a winning movement from a given position, if there is one:

$$\begin{array}{lll} winMove(State) \rightarrow State' \Leftarrow & State' \bowtie move(State), \\ & fails(winMove(State')) \bowtie true \end{array}$$

We think it would be difficult to find a simpler coding without using failure.

As a concrete example we consider the well-known game *Nim*, where there are some rows of sticks, and each player in his turn must pick up one or more sticks from one of the rows. A player loses when he cannot make a movement, that is, when there are not more sticks because the other player (the winner) has picked up the last one. Nim states can be defined by a list of natural numbers (represented by 0 and $s(\_)$ as usual), and the non-deterministic function *move* can be programmed as:

$$\begin{array}{l} move([N|Ns]) \to [pick(N)|Ns] \\ move([N|Ns]) \to [N|move(Ns)] \\[4pt] pick(s(N)) \to N \\ pick(s(N)) \to pick(N) \end{array}$$

A winning move from the state $[s(s(z)), s(z)]$ can be obtained by reducing the expression $winMove([s(s(z)), s(z)])$. The proof calculus presented in section 3.2 can prove that it can be reduced to $[s(z), s(z)]$, and it is easy to check that this move guarantees the victory.

Example 4 (*Failure to express default rules*)
Compared to the case of LP, failure is not a so important programming construct in FP. There is still one practical feature of existing FP languages somehow related to failure, which is the possibility of defining functions with *default rules*. In many FP systems pattern matching determines *the* applicable rule for a function call, and as rules are tried from top to bottom, default rules are implicit in the definitions. In fact, the $n+1$th rule in a definition is only applied if the first n rules are not applicable. For example, assume the following definition for the function f:

$$\begin{array}{l} f(0) \to 0 \\ f(X) \to 1 \end{array}$$

The evaluation of the expression $f(0)$ in a functional language like Haskell (Peyton-Jones and Hughes 1999), will produce the value 0 by the first rule. The second rule is not used for evaluating $f(0)$, even if pattern matching would succeed if the rule would be considered in isolation. This sequential treatment of rules is useful in some cases, specially for writing 'last' rules covering default cases whose direct formulation with pattern matching could be complicated. But observe that in systems allowing such sequential trials of pattern matching, rules have not a declarative meaning by themselves; their interpretation also depends upon the previous rules.

This contrasts with functional logic languages which try to preserve the declarative reading of each rule. In such systems the expression $f(0)$ of the example above is reducible, by applying in a non-deterministic way any of the rules, to the values 0 and 1.

To achieve (and generalize) the effect of default rules in FLP, an explicit syntactical construction *'default'* can be introduced, as it has been done in Moreno (1994). The function f could be defined as:

$$\begin{array}{l} f(0) \to 0 \\ default\ f(X) \to 1 \end{array}$$

The intuitive operational meaning is: to reduce a call to f proceed with the first rule for f; if the reduction fails then try the default rule.

The problem now is how to achieve this behavior while preserving the equational reading of each rule. Using conditional rewrite rules and our function *fails*(_), we can transform the definition of a function to eliminate default rules. In the general case we can consider conditional rewrite rules for the original definition. Let h be a function defined as:

$$\begin{array}{l} h(\bar{t}_1) \to e_1 \Leftarrow \overline{C}_1 \\ \ldots \\ h(\bar{t}_n) \to e_n \Leftarrow \overline{C}_n \\ default\ h(\bar{t}_{n+1}) \to e_{n+1} \Leftarrow \overline{C}_{n+1} \end{array}$$

The idea of the transformation is to consider a new function h' defined by the first n rules of h. The original h will be defined as h' if it succeeds and as the default rule if h' fails:

$$\begin{array}{l} h(\overline{X}) \to h'(\overline{X}) \\ h(\bar{t}_{n+1}) \to e_{n+1} \Leftarrow fails(h'(\bar{t}_{n+1})) \bowtie true,\ \overline{C}_{n+1} \\ h'(\bar{t}_1) \to e_1 \Leftarrow \overline{C}_1 \\ \ldots \\ h'(\bar{t}_n) \to e_n \Leftarrow \overline{C}_n \end{array}$$

Applying this transformation to our function example f, we obtain:

$$\begin{array}{l} f(X) \to f'(X) \\ f(X) \to 1 \Leftarrow fails(f'(X)) \bowtie true \\ f'(0) \to 0 \end{array}$$

With this definition we have got the expected behavior for f without losing the declarative reading of rules.

As another example, we can use a default rule to complete the definition of the function *path* in the example 2 above:

$$\begin{array}{l} path(X,Y) \to true \Leftarrow X \bowtie Y \\ path(X,Y) \to true \Leftarrow X <> Y, path(next(X),Y) \bowtie true \\ default\ path(X,Y) \to false \end{array}$$

The function *safe* can now be written as:

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

where *neg* is the boolean function

$$\begin{array}{ll} neg(true) & \to false \\ neg(false) & \to true \end{array}$$

Notice that in this example the (implicit) condition for applying the default rule of *path* is far more complex than a merely syntactical default case expressing failure of pattern matching, a feature recently discussed in the Curry mailing list (2000) as useful for FLP. Of course, default rules in the sense of Moreno (1994) and of this paper also cover such syntactical cases.

3 The *CRWL* framework

We give here a short summary of (a slight variant of) *CRWL*, in its proof-theoretic face. Model theoretic semantics and lazy narrowing operational semantics are not considered here. Full details can be found elsewhere (González *et al.* 1999; López and Sánchez 1999b).

3.1 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 $X, Y, Z, \ldots$ for variables, c, d for constructor symbols, f, g for functions, e for terms and s, t for c-terms. In all cases, primes (') and subindices can be used.

We 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 c-terms t and t' have an S-*clash* if they have different constructor symbols of S at the same position.

3.2 The proof calculus for CRWL

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

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

where $f \in FS^n$, and fulfilling the following conditions:

- $(t_1,\ldots,t_n)$ is a linear tuple (each variable in it occurs only once) with $t_1,\ldots,t_n \in CTerm$;
- $e \in Term$;
- each C_i is a constraint of the form $e' \bowtie e''$ (*joinability*) or $e' <> e''$ (*divergence*) where $e', e'' \in Term$;
- *extra variables* are not allowed, i.e. all the variables appearing in the body e and the condition $\overline{C}$ must also appear in the head $f(\overline{t})$ ($var(e) \cup var(\overline{C}) \subseteq var(\overline{t})$).

This condition is not required in González *et al.* (1996, 1999); see the end of this section for a discussion of this issue.

The reading of the rule is: $f(t_1,\ldots,t_n)$ reduces to e if the conditions $C_1,\ldots,C_n$ are satisfied. We write $\mathscr{P}_f$ for the set of defining rules of f in $\mathscr{P}$.

Given a program $\mathscr{P}$, the proof calculus for *CRWL* can derive from it three kinds of statements:

- *Reduction or approximation statements*: $e \rightarrow t$, 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 $\mathscr{P}$ or by replacing subterms of e by $\perp$. If $e \rightarrow t$ 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 $e \rightarrow t$ and $e' \rightarrow t$ for some $t \in CTerm$.
- *Divergence statements*: $e <> e'$, 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' having a DC-clash. In González *et al.* (1996, 1999), divergence conditions are not considered. They have been incorporated to *CRWL* in López and Sánchez (1999b) as a useful and expressive resource for programming that is implemented in the system $\mathscr{TOY}$.

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:

$$[R]_\perp = \{R\theta | \theta \in CSubst_\perp\}$$

Parameter passing in function calls will be expressed by means of these c-instances in the proof calculus.

Table 1 shows the proof calculus for *CRWL*. We write $\mathscr{P} \vdash_{CRWL} \varphi$ for expressing that the statement φ is provable from the program $\mathscr{P}$ with respect to this calculus. The rule (4) allows to use c-instances of program rules to prove approximations. These c-instances may contain $\perp$ and by rule (1) any expression can be reduced to $\perp$. This reflects a non-strict semantics. A variable X can only be approximated by itself (rule 2) and by $\perp$ (rule 1), so a variable is similar to a constant in derivations with this calculus. Nevertheless, when using function rules of the program a variable of such rule can take any value by taking the appropriate c-instance. Rule (3) is for term decomposition and rules (5) and (6) corresponds to the definition of $\bowtie$ and $<>$ respectively.

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* for expressing the possible results of throwing a coin can defined by the rules:

$$coin \rightarrow z$$
$$coin \rightarrow s(z)$$

Table 1. *Rules for CRWL-provability*

$$(1)\quad \frac{}{e \to \bot}$$

$$(2)\quad \frac{}{X \to X} \qquad X \in \mathscr{V}$$

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

$$(4)\quad \frac{e_1 \to s_1, \ldots, e_n \to s_n \quad C \quad e \to t}{f(e_1, \ldots, e_n) \to t} \qquad \begin{array}{l} \text{if } t \not\equiv \bot, R \in \mathscr{P}_f \\ (f(s_1, \ldots, s_n) \to e \Leftarrow C) \in [R]_\bot \end{array}$$

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

$$(6)\quad \frac{e \to t \quad e' \to t'}{e <> e'} \qquad \text{if } t, t' \in CTerm_\bot \text{ and have a } DC\text{–clash}$$

It is not difficult to see that the previous calculus can derive the statement *coin* $\to z$ and also *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 González *et al.* (1999)). As an example, in addition to *coin* consider the functions *add* and *double* defined as:

$$\begin{array}{ll} add(z, Y) \to Y & \\ add(s(X), Y) \to s(add(X, Y)) & \qquad double(X) \to add(X, X) \end{array}$$

It is possible to build a *CRWL*-proof for the statement $double(coin) \to z$ and also for $double(coin) \to s(s(z))$, but not for $double(coin) \to s(z)$. As an example of derivation, we show a derivation for $double(coin) \to z$; at each step we indicate by a number on the left the rule of the calculus applied:

$$ {}^{4}\frac{{}^{4}\dfrac{{}^{3}\dfrac{}{z \to z}}{coin \to z} \qquad {}^{4}\dfrac{{}^{3}\dfrac{}{z \to z} \quad {}^{3}\dfrac{}{z \to z} \quad {}^{3}\dfrac{}{z \to z}}{add(z, z) \to z}}{double(coin) \to z}$$

Observe that $<>$ 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 $e <> e'$ (although this cannot happen if e, e' are c-terms). In the 'coin' example, we can derive both $coin \bowtie z$ and $coin <> z$.

The *denotation* of an expression e can be defined as the set of c-terms to which e can be reduced according to this calculus:

$$[\![e]\!] = \{t \in CTerm_\bot | \mathscr{P} \vdash_{CRWL} e \to t\}$$

For instance, $[\![coin]\!] = \{\bot, z, s(\bot), s(z)\}$.

To end our presentation of the *CRWL* framework we discuss the issue of extra variables (variables not appearing in the left-hand sides of function rules), which are allowed in González *et al.* (1996, 1999), but not in this paper. This is not *as* restrictive as it could appear: function nesting can replace the use (typical of 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 functions able to compute candidates for such unknown values. A concrete example is given by the function *next* in example 2. More examples can be found in González *et al.* (1999) and Abengózar *et al.* (2002).

The only extra variable we have used in section 2 is Pos' in the definition

$$winMove(Pos) \to Pos' \Leftarrow Pos' \bowtie move(Pos), fails(winMove(Pos')) \bowtie true$$

of example 3. It can be removed by introducing an auxiliary function:

$$\begin{array}{l} winMove(Pos) \to aux(move(Pos)) \\ aux(Pos) \to Pos \Leftarrow Pos \bowtie Pos, fails(winMove(Pos)) \bowtie true \end{array}$$

The effect of the condition $Pos \bowtie Pos$ it to compute a normal form for Pos, which is required in this case to avoid a diverging computation for *winMove(Pos)*.

4 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 $e \to \perp$) in *CRWL*, i.e. $[\![e]\!] = \{\perp\}$.

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

Prior 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.

4.1 Some illustrative examples

Consider the following functions, in addition to *coin*, defined in section 3.2:

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

We discuss several situations involving failure with this program:

- 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 $f(z) \to \perp$ (by rule 1); any attempt to prove $f(z) \to t$

with $t \neq \perp$ would produce an 'infinite derivation'. In the second case, the only possible derivation is again $f(s(z)) \to \perp$, but if we try to prove $f(s(z)) \to t$ with $t \neq \perp$ 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 $g(coin) \to \perp$ 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 $\mathscr{C}$ is the collected set of values of e is expressed in *CRWLF* by means of the statement $e \lhd \mathscr{C}$. In our example, *CRWLF* will prove $coin \lhd \{z, s(z)\}$. Statements $e \lhd \mathscr{C}$ generalize the approximation statements $e \to t$ of *CRWL*, and in fact can replace them. Thus, *CRWLF* will not need to use explicit $e \to t$ 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 $H = \{s(z), s(s(z)), s(s(s(z))), \ldots\}$. 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 H, because all its elements have the form $s(\ldots)$. The partial value $\{s(\perp)\}$ is a common approximation to all the elements of H. Here we can understand $\perp$ as an *incomplete information*: we know that all the values for h are successor of 'something', and this implies that they cannot be z, which suffices for proving the failure of $f(h)$. The *CRWLF*-calculus will be able to prove the statement $h \lhd \{s(\perp)\}$, and we say that $\{s(\perp)\}$ is a Sufficient Approximation Set (SAS) for h.

  In general, an expression will have multiple SASs. Any expression has $\{\perp\}$ as its simplest SAS. And, for example, the expression h has an infinite number of *SAS*'s: $\{\perp\}$, $\{s(\perp)\}$, $\{s(z), s(s(\perp))\}$,. . . The *SAS*'s obtained by the calculus for *coin* are $\{\perp\}$, $\{\perp, s(\perp)\}$,$\{\perp, s(z)\}$, $\{z, \perp\}$, $\{z, s(\perp)\}$ and $\{z, s(z)\}$. The *CRWLF*-calculus provides appropriate rules for working with SASs. The derivation steps will be guided by these SASs 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 $k(z) \to z \Leftarrow z \bowtie s(z)$ 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 because of the clash of constructors. In general the proof for a constraint $e \not\bowtie e'$ will be guided by the corresponding SASs for e and e' as we see in the next section. As our initial *CRWL* framework also allows constraints of the form $e <> e'$, we need also 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 $\{\perp\}$ 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 $s(\perp)$). 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 $s(\mathsf{F})$. One could think 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.

4.2 Technical preliminaries

For dealing with failure we consider two new syntactical elements in *CRWLF*: a function *fails* and a constant F. The first one is directly included into the signature, so we consider $\Sigma = DC \cup FS \cup \{fails\}$, where DC and FS are sets of *constructor* symbols and (user-defined) *functions*, respectively. This symbol, *fails*, stands for a predefined function whose intuitive meaning is:

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

The boolean constants *true* and *false* must belong to DC, as they are needed to define the function *fails*. The formal interpretation of this function will be defined by specific rules at the level of the proof-calculus (Table 2).

The second syntactical element, the constant F, is introduced as an extension of the signature (as it was the element $\perp$ in *CRWL*). So we use the extended signature $\Sigma_{\perp,\mathsf{F}} = \Sigma \cup \{\perp, \mathsf{F}\}$. We do not include it directly in the signature Σ because its role is to express failure of reduction and it is not allowed to appear explicitly in a program. In the case of the function *fails* we want to allow to use it in programs as we have seen in the examples of section 2.

Table 2. *Rules for CRWLF-provability*

(1)	$\dfrac{}{e \lhd \{\bot\}}$	
(2)	$\dfrac{}{X \lhd \{X\}}$	$X \in \mathcal{V}$
(3)	$\dfrac{e_1 \lhd \mathcal{C}_1 \quad \ldots \quad e_n \lhd \mathcal{C}_n}{c(e_1,\ldots,e_n) \lhd \{c(t_1,\ldots,t_n) \mid \bar{t} \in \mathcal{C}_1 \times \ldots \times \mathcal{C}_n\}}$	$c \in DC^n \cup \{\mathsf{F}\}$
(4)	$\dfrac{e_1 \lhd \mathcal{C}_1 \quad \ldots \quad e_n \lhd \mathcal{C}_n \quad \ldots \quad f(\bar{t}) \lhd_R \mathcal{C}_{R,\bar{t}} \quad \ldots}{f(e_1,\ldots,e_n) \lhd \bigcup_{R \in \mathcal{P}_f, \bar{t} \in \mathcal{C}_1 \times \ldots \times \mathcal{C}_n} \mathcal{C}_{R,\bar{t}}}$	$f \in FS^n$
(5)	$\dfrac{}{f(\bar{t}) \lhd_R \{\bot\}}$	
(6)	$\dfrac{e \lhd \mathcal{C} \quad \overline{C}}{f(\bar{t}) \lhd_R \mathcal{C}}$	$(f(\bar{t}) \rightarrow e \Leftarrow \overline{C}) \in [R]_{\bot,\mathsf{F}}$
(7)	$\dfrac{e_i \tilde{\diamond} e'_i}{f(\bar{t}) \lhd_R \{\mathsf{F}\}}$	$(f(\bar{t}) \rightarrow e \Leftarrow \ldots, e_i \diamond e'_i, \ldots) \in [R]_{\bot,\mathsf{F}}$, where $i \in \{1,\ldots,n\}$
(8)	$\dfrac{}{f(t_1,\ldots,t_n) \lhd_R \{\mathsf{F}\}}$	$R \equiv (f(s_1,\ldots,s_n) \rightarrow e \Leftarrow \overline{C}), t_i$ and s_i have a $DC \cup \{\mathsf{F}\}$-clash for some $i \in \{1,\ldots,n\}$
(9)	$\dfrac{e \lhd \mathcal{C} \quad e' \lhd \mathcal{C}'}{e \bowtie e'}$	$\exists t \in \mathcal{C}, t' \in \mathcal{C}' \; t \downarrow t'$
(10)	$\dfrac{e \lhd \mathcal{C} \quad e' \lhd \mathcal{C}'}{e <> e'}$	$\exists t \in \mathcal{C}, t' \in \mathcal{C}' \; t \uparrow t'$
(11)	$\dfrac{e \lhd \mathcal{C} \quad e' \lhd \mathcal{C}'}{e \not\bowtie e'}$	$\forall t \in \mathcal{C}, t' \in \mathcal{C}' \; t \not\downarrow t'$
(12)	$\dfrac{e \lhd \mathcal{C} \quad e' \lhd \mathcal{C}'}{e \not<> e'}$	$\forall t \in \mathcal{C}, t' \in \mathcal{C}' \; t \not\uparrow t'$
(13)	$\dfrac{e \lhd \{\mathsf{F}\}}{\mathit{fails}(e) \lhd \{\mathit{true}\}}$	
(14)	$\dfrac{e \lhd \mathcal{C}}{\mathit{fails}(e) \lhd \{\mathit{false}\}}$	$\exists t \in \mathcal{C}, t \neq \bot, t \neq \mathsf{F}$

The sets $Term_{\bot,\mathsf{F}}, CTerm_{\bot,\mathsf{F}}$ are defined in the natural way, and also the set of substitutions $CSubst_{\bot,\mathsf{F}} = \{\theta : \mathcal{V} \rightarrow CTerm_{\bot,\mathsf{F}}\}$.

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

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

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

Notice that the only relations satisfied by F are $\bot \sqsubseteq \mathsf{F}$ and $\mathsf{F} \sqsubseteq \mathsf{F}$. In particular, F is maximal. This is reasonable, since F represents 'failure of reduction' and this gives no further refinable information about the result of the evaluation of an expression. This contrasts with the status given to failure in Moreno (1996), where F is chosen to verify $\mathsf{F} \sqsubseteq t$ for any t different from $\bot$.

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

4.3 The proof calculus for CRWLF

Programs in *CRWLF* are sets of rules with the same form as in *CRWL*, but now they can make use of the function *fails* in the body and in the condition part, i.e. *CRWLF* extends the class of programs of *CRWL* by allowing the use of *fails* in programs. On the other hand, in *CRWLF* five kinds of statements can be deduced:

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

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

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

Definition 1 (*Relations over $CTerm_{\bot,\mathsf{F}}$*)

- $t \downarrow t' \Leftrightarrow_{def} t = t', t \in CTerm$
- $t \uparrow t' \Leftrightarrow_{def} t$ 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 $CTerm_{\bot,\mathsf{F}}$ satisfying:
  - (i) $X \not\uparrow X$, for all $X \in \mathscr{V}$
  - (ii) $\mathsf{F} \not\uparrow t$, for all $t \in CTerm_{\bot,\mathsf{F}}$
  - (iii) if $t_1 \not\uparrow t'_1, \ldots, t_n \not\uparrow t'_n$ then $c(t_1, \ldots, t_n) \not\uparrow c(t'_1, \ldots, t'_n)$, for $c \in DC^n$

The relations $\downarrow$ and $\uparrow$ do not take into account the presence of F, which behaves in this case as $\bot$. 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 $CTerm_{\bot}$. 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, $\not\Uparrow$ is also an approximation to '$\neg \uparrow$' which can be read as '$\uparrow$ fails'.

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

Proposition 1
The relations $\downarrow, \uparrow, \not\downarrow, \not\Uparrow$ satisfy

(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) $t \downarrow t' \Rightarrow t \not\Uparrow t' \Rightarrow \neg(t \uparrow t')$

(b) $\downarrow, \uparrow, \not\downarrow, \not\Uparrow$ are monotonic, i.e., if $t \sqsubseteq s$ and $t' \sqsubseteq s'$ then: $t\mathfrak{R}t' \Rightarrow s\mathfrak{R}s'$, where $\mathfrak{R} \in \{\downarrow, \uparrow, \not\downarrow, \not\Uparrow\}$. Furthermore $\not\downarrow_G$ and $\not\Uparrow_G$ are the greatest monotonic approximations to $\neg \downarrow_G$ and $\neg \uparrow_G$, respectively, where $\mathfrak{R}_G$ is the restriction of $\mathfrak{R}$ to the set of ground (i.e. without variables) c-terms from $CTerm_{\perp,\mathsf{F}}$.

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

Proof
We prove each property separately.

(a) (i)
- $t \downarrow t' \Leftrightarrow \hat{t} \downarrow \hat{t}'$: two terms satisfying the relation $\downarrow$ cannot contain $\perp$ neither F. Hence $t = \hat{t}$ and $t' = \hat{t}'$, and the equivalence is trivial.
- $t \uparrow t' \Leftrightarrow \hat{t} \uparrow \hat{t}'$: the relation $\uparrow$ is satisfied when the terms have a DC-clash at some position p; since t and $\hat{t}$ (t' and $\hat{t}'$ resp.) have the same constructor symbols at the same positions, the equivalence is clear.

(ii) The implication $t \uparrow t' \Rightarrow t \not\downarrow t'$ is clear from definitions of $\uparrow$ and $\not\downarrow$. For $t \not\downarrow t' \Rightarrow \neg(t \downarrow t')$: if $t \not\downarrow t'$ then either F appears in t or t', or t and t' have a DC-clash. In both cases $t \downarrow t'$ does not hold.

(iii) For $t \downarrow t' \Rightarrow t \not\Uparrow t'$: if $t \downarrow t'$ then $t = t'$ with $t \in CTerm$ and we have $t \not\Uparrow t'$ by applying repeatedly (i) and (iii) of the definition of $\not\Uparrow$. For $t \not\Uparrow t' \Rightarrow \neg(t \uparrow t')$ let us assume $t \not\Uparrow t'$ and proceed by induction on the depth d of t:
$\underline{d = 0}$: if $t = \perp$ or $t = \mathsf{F}$ then t and t' cannot have any DC-clash and then $t \uparrow t'$ is not true. If $t = X$ or $t = c \in DC^0$ then $t \not\Uparrow t'$ implies that $t' = \mathsf{F}$ or $t' = t$; therefore t and t' cannot have any DC-clash and $t \uparrow t'$ is not true.
$\underline{d \Rightarrow d+1}$: if $t = c(t_1, \ldots, t_n)$, then either $t' = \mathsf{F}$ and $t \uparrow t'$ is not true, or $t' = c(t'_1, \ldots, t'_n)$ with $t_i \not\Uparrow t'_i$ for all $i \in \{1, \ldots, n\}$; in this case, by i.h. there is not a pair (t_i, t'_i) with a DC-clash, so neither t and t' have DC-clashes, and therefore $t \uparrow t'$ is not true.

(b) We prove monotonicity for each relation:
- For $\downarrow$: by definition of $\downarrow$, if $t \downarrow t'$ then $t, t' \in CTerm$ (they are maximal with respect to $\sqsubseteq$), hence $s = t$ and $s' = t'$ and then $s \downarrow s'$.
- For $\uparrow$: if $t \uparrow t'$ then t and t' have a DC-clash at some position. As $t \sqsubseteq s$ and $t' \sqsubseteq s'$, then s and s' will have the same DC-clash at the same position, so $s \uparrow s'$.

- For $\not\downarrow$: if t and t' have a DC-clash, s and s' will contain the same DC-clash, as in *ii*). If one of them has F as subterm, by definition of $\sqsubseteq$ it is clear that s or s' will also contain F, so $s \not\downarrow s'$.
- For $\not\uparrow$: Here we proceed by induction on the depth d of the term t:
  $\underline{d = 0}$: let us check the possibilities for t. If $t = X$ or $t = c \in DC^0$, then $t \not\uparrow t'$ implies $t' = t$ or $t' = \mathsf{F}$; since t, t' are maximal with respect to $\sqsubseteq$, then $s = t$ and $s' = t'$, so we will also have $s \not\uparrow s'$. If $t = \mathsf{F}$ then $s = \mathsf{F}$ and then it is clear that $s \not\uparrow s'$. If $t = \perp$ then $t' = \mathsf{F} = s'$ and it is clear that $s \not\uparrow s'$.
  $\underline{d \Rightarrow d+1}$: in this case $t = c(t_1, \ldots, t_n)$ and then either $t' = \mathsf{F}$, what implies $s' = \mathsf{F}$ and then $s \not\uparrow s'$, or $t' = c(t'_1, \ldots, t'_n)$ with $t_i \not\uparrow t'_i$ for all $i \in \{1, \ldots, n\}$. From $t \sqsubseteq s$ and $t' \sqsubseteq s'$ it follows that $s = c(s_1, \ldots, s_n)$ and $s' = c(s'_1, \ldots, s'_n)$, and by i.h. we have $s_i \not\uparrow s'_i$ for all $i \in \{1, \ldots, n\}$, what implies $s \not\uparrow s'$.

Now we prove that $\not\downarrow_G$ and $\not\uparrow_G$ are the greatest monotonic approximations to $\neg \downarrow_G$ and $\neg \uparrow_G$, respectively. We note by $GCTerm_{\perp,\mathsf{F}}$ the set of all ground $t \in CTerm_{\perp,\mathsf{F}}$.

- For $\not\downarrow_G$, assume that a relation $R \subseteq (GCTerm_{\perp,\mathsf{F}} \times GCTerm_{\perp,\mathsf{F}})$ verifies

$$\begin{array}{l} tRt' \Rightarrow \neg(t \downarrow_G t') \\ t \sqsubset s, t' \sqsubseteq s', tRt' \Rightarrow sRs' \end{array}$$

  We must prove that R is included in $\not\downarrow_G$, that is: $(tRt' \Rightarrow t \not\downarrow_G t')$, for any $t, t' \in GCTerm_{\perp,\mathsf{F}}$. We reason by contradiction. Assume tRt' and $\neg(t \not\downarrow_G t')$. Then, by definition of $\not\downarrow_G$, t and t' do not contain F and do not have a DC-clash. Then either $t = t'$, or t and t' differ because at some positions one of them has $\perp$ while the other has not. In both cases, it is easy to see that there exists $s \in GCTerm$ (totally defined) such that $t \sqsubseteq s$ and $t' \sqsubseteq s$. By monotonicity of R we have sRs what implies $\neg(s \downarrow_G s)$, what is a contradiction, since $s \in CTerm$.
- For $\not\uparrow_G$ we proceed in a similar way as in the previous point: assuming that $R \subseteq (GCTerm_{\perp,\mathsf{F}} \times GCTerm_{\perp,\mathsf{F}})$ verifies

$$\begin{array}{l} tRt' \Rightarrow \neg(t \uparrow_G t') \\ t \sqsubset s, t' \sqsubseteq s', tRt' \Rightarrow sRs' \end{array}$$

  we must prove $(tRt' \Rightarrow t \not\uparrow_G t')$. But if tRt' then $\neg(t \uparrow_G t')$, so t and t' cannot have any DC-clash. They could contain F as subterm but then, by (ii) and (iii) of the definition of $\not\uparrow$, we will have $t \not\uparrow_G t'$.

(c) The property is clear for $\downarrow$: if we replace in a c-term all the occurrences of a variable by a totally defined c-term, we will obtain a totally defined c-term. For $\uparrow$, such substitution preserves the DC-clash of the original c-terms.
For $\not\downarrow$, if some of the original c-terms had F as a subterm, the substitution preserves this occurrence of F. On the other hand, if they had a DC-clash, then it is clear that this clash will also be present under the substitution.
For $\not\uparrow$, suppose $t \not\uparrow t'$ and $\theta \in CSusbt_{\perp,\mathsf{F}}$; we proceed by induction on the depth d of the term t:

$\underline{d = 0}$: if $t = \mathsf{F}$, then $t\theta = \mathsf{F}$ and it is clear that $t\theta \not\uparrow t'\theta$. For the cases $t = X$ and $t = c \in DC^0$ we have two possibilities for t': $t' = \mathsf{F}$ or $t' = t$; if $t' = \mathsf{F}$ the result is clear. If we have $t = t' = X$ it is not difficult to prove that $X\theta \not\uparrow X\theta$ by applying repeatedly (i) and (iii) of definition of $\not\uparrow$. The last case, if $t = t' = c \in DC^0$ is trivial because θ does not change the terms.

$\underline{d \Rightarrow d+1}$: in this case $t = c(t_1, \ldots, t_n)$. If $t' = \mathsf{F}$ the proof is as in the base case, otherwise $t' = c(t'_1, \ldots, t'_n)$ with $t_i \not\uparrow t'_i$ for all $i \in \{1, \ldots, n\}$. By i.h. we have $t_i\theta \not\uparrow t'_i\theta$ and then, by (iii) of the definition of $\not\uparrow$ we have $t\theta \not\uparrow t'\theta$. $\square$

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 ground c-terms) as computable approximations to $\neg\downarrow$ and $\neg\uparrow$. 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. 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 closure by substitutions are preserved when generalizing $\downarrow, \uparrow, \not\downarrow, \not\uparrow$ to $\bowtie, \Diamond, \not\bowtie, \not\Diamond$.

We present now the proof rules for the *CRWLF*-calculus, which are shown in Table 2. Rules 6 and 7 use a generalized notion of c-instances of a rule R: $[R]_{\perp,\mathsf{F}} = \{R\theta \mid \theta \in CSubst_{\perp,\mathsf{F}}\}$. We will use the notation $\mathscr{P} \vdash_{CRWLF} \varphi$ ($\mathscr{P} \nvdash_{CRWLF} \varphi$ resp.) for expressing that the statement φ is provable (is not provable resp.) with respect to the calculus *CRWLF* and the program $\mathscr{P}$. *CRWLF*-derivations have a tree structure (e.g. see Example 5); many results in the following sections use induction over the *size* of the derivation, i.e. the number of nodes in the derivation tree, which corresponds to the number of inference steps.

The first three rules are analogous to those of the *CRWL*-calculus, now dealing with SASs instead of simple approximations (notice the cross product of SASs in rule 3). Rule 4 is a complex rule which requires some explanation to make clear its reading and, more importantly, its decidability: to obtain an SAS $\mathscr{C}$ for an expression $f(e_1, \ldots, e_n)$ (that is, to derive $f(e_1, \ldots, e_n) \lhd \mathscr{C}$) we must first obtain *SAS*'s for $e_1, \ldots, e_n$ (that is, we must derive $e_1 \lhd \mathscr{C}_1, \ldots, e_n \lhd \mathscr{C}_n$); then for each combination $\bar{t}$ of values in these SASs (that is, for each $\bar{t} \in \mathscr{C}_1 \times \ldots \times \mathscr{C}_n$) and each program rule R for f, a part $\mathscr{C}_{R,\bar{t}}$ of the whole SAS is produced; the union of all these partial SASs constitutes the final SAS $\mathscr{C}$ for $f(\bar{e})$. Notice that since *SAS*'s are finite sets and programs are finite sets of rules, then there is a finite number of $\mathscr{C}_{R,\bar{t}}$ to be calculated in the premises of the rule, and the union of all of them (the final calculated SAS in the rule) is again a finite set [2].

[2] To be more precise, this reasoning would be the essential part of an inductive proof of finiteness of SASs. But we do not think necessary to burden the reader with such formality.

Rule 4 is quite different from rule 4 in $CRWL$, where 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 $f(\overline{t}) \lhd_R \mathscr{C}$ to indicate that only the rule R is used to produce $\mathscr{C}$.

Rules 5– 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 $CTerm_{\perp,\mathsf{F}}$ (they 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 at most one of rules 6–8 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.

Rules 9–12 deal with constraints. With the use of the relations $\downarrow, \uparrow, \not\downarrow, \not\uparrow$ introduced in section 3.3 the rules are easy to formulate. For $e \bowtie e'$ it is sufficient to find two c-terms in the SASs 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 SASs in order to check the relation $\not\downarrow$ over all the possible pairs. The explanation of rules 10 and 12 is quite similar.

Finally rules 13 and 14 provide together a formal definition of the function *fails* supported by the notion of SAS. Notice that the SAS's $\{\perp\}$ or $\{\perp, \mathsf{F}\}$ do not provide enough information for reducing a call to *fails*. The call $fails(e)$ is only reduced to $\{true\}$ when every possible reduction of the expression e is failed; and it is reduced to $\{false\}$ there is some reduction of e to some (possible partial) c-term of the form $c(\ldots)$ $(c \in DC)$ or X.

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

Example 5

Let us consider a program $\mathscr{P}$ 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 t, f that represent the boolean values *true* and *false*. Assume the functions *coin* and h defined in section 3.2 and section 4.1 respectively and also the function mb (member) defined as:

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

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 $[s(s(\ldots))]$ and the possible values of $coin$, z and $s(z)$, do not belong to those lists. The *CRWLF*-calculus allows to build a proof for this fact, that is, $mb(coin, [s(h)]) \lhd \{\mathsf{F}\}$, in the following way: by application of rule 4 the proof could proceed by generating SASs for the arguments

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

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

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

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

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

The full derivation takes the form:

$$_{4}\dfrac{\varphi_1 \quad \varphi_2 \quad \varphi_3 \quad \varphi_4 \quad \varphi_5 \quad \varphi_6}{mb(coin, [s(h)]) \lhd \{\mathsf{F}\}}$$

The SAS $\{\mathsf{F}\}$ in the conclusion comes from the union of all the contributing SASs of φ_3, φ_4, φ_5 and φ_6. The statements φ_1 to φ_6 require of course their own proof, which we describe now. At each step, we indicate by a number on the left the rule of the calculus applied in each case:

The derivation for φ_1 is not difficult to build, and for φ_2 it is:

$$_{3}\dfrac{_{3}\dfrac{_{4}\dfrac{_{6}\dfrac{_{3}\dfrac{_{1}\dfrac{}{z \lhd \{\perp\}}}{s(z) \lhd \{s(\perp)\}}}{h \lhd_1 \{s(\perp)\}} \qquad _{6}\dfrac{_{3}\dfrac{_{1}\dfrac{}{h \lhd \{\perp\}}}{s(h) \lhd \{s(\perp)\}}}{h \lhd_2 \{s(\perp)\}}}{h \lhd \{s(\perp)\}}}{s(h) \lhd \{s(s(\perp))\}} \qquad _{3}\dfrac{}{[\,] \lhd \{[\,]\}}}{[s(h)] \lhd \{[s(s(\perp))]\}}$$

For φ_3 it can be done as follows:

$$
{}^{7}\dfrac{
  {}^{11}\dfrac{
    {}^{3}\dfrac{}{z \lhd \{z\}}
    \qquad
    {}^{3}\dfrac{
      {}^{3}\dfrac{
        {}^{1}\dfrac{}{\bot \lhd \{\bot\}}
      }{s(\bot) \lhd \{s(\bot)\}}
    }{s(s(\bot)) \lhd \{s(s(\bot))\}}
  }{z \not\bowtie s(s(\bot))}
}{\varphi_3 \equiv mb(z, [s(s(\bot))]) \lhd_1 \{\mathsf{F}\}}
$$

Here, the failure is due to a failure in the constraint $z \bowtie s(s(\bot))$ of the used program rule, what requires to prove the complementary constraint $z \not\bowtie s(s(\bot))$ by rule (11). In this case there is a clear clash of constructors (z and s).

For φ_4 a derivation might be this one:

$$
{}^{7}\dfrac{
  {}^{11}\dfrac{
    {}^{4}\dfrac{
      {}^{3}\dfrac{}{z \lhd \{z\}}
      \quad
      {}^{3}\dfrac{}{[\,] \lhd \{[\,]\}}
      \quad
      {}^{8}\dfrac{}{mb(z,[\,]) \lhd_1 \{\mathsf{F}\}}
      \quad
      {}^{8}\dfrac{}{mb(z,[\,]) \lhd_2 \{\mathsf{F}\}}
    }{mb(z,[\,]) \lhd \{\mathsf{F}\}}
    \qquad
    {}^{3}\dfrac{}{\mathsf{t} \lhd \{\mathsf{t}\}}
  }{mb(z,[\,]) \not\bowtie \{\mathsf{t}\}}
}{\varphi_4 \equiv mb(z, [s(s(\bot))]) \lhd_2 \{\mathsf{F}\}}
$$

The failure is due again to a failure in the constraint of the rule and in this case the complementary constraint is $mb(z,[\,]) \not\bowtie \mathsf{t}$. Now it is involved the failure for the expression $mb(z,[\,])$ that is proved by rule (4) of the calculus. The *SAS*'s for the arguments only produce the combination $(z,[\,])$ and both rules of mb fail over it by rule (8) of the calculus.

The derivations for φ_5 and φ_6 are quite similar to those of φ_3 and φ_4, respectively. All the contributions obtained from $\varphi_3, \varphi_4, \varphi_5$ and φ_6 are $\{\mathsf{F}\}$, and putting them together we obtain $\{\mathsf{F}\}$ as an SAS for the original expression $mb(coin, [s(h)])$, as it was expected.

5 Properties of *CRWLF*

In this section we explore some technical properties of the *CRWLF*-calculus which are the key for proving the results of the next section, where we relate the *CRWLF*-calculus to the *CRWL*-calculus. In the following we assume a fixed program $\mathscr{P}$.

The non-determinism of the *CRWLF*-calculus allows to obtain different SASs for the same expression. As an SAS for an expression is a finite approximation to the denotation of the expression it is expected some kind of consistency between SASs 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 SASs for *coin* are $\{\bot, s(z)\}$ and $\{z, \bot\}$. The kind of consistency for SASs that we can expect is the following:

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

Our first result states that two different SASs for the same expression must be consistent.

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

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

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

a) If e, e' are consistent, $\mathscr{P} \vdash_{CRWLF} e \lhd \mathscr{C}$ and $\mathscr{P} \vdash_{CRWLF} e' \lhd \mathscr{C}'$, then $\mathscr{C}$ and $\mathscr{C}'$ are consistent.

b) If e_1, e_1' are consistent and e_2, e_2' are also consistent, then: $\mathscr{P} \vdash_{CRWLF} e_1 \diamond e_2 \Rightarrow \mathscr{P} \nvdash_{CRWLF} e_1' \tilde{\diamond} e_2'$

Proof
For proving the consistency lemma we will split (b) into (b.1), (b.2) and also strengthen the lemma with a new part (c):

(b) If e_1, e_1' are consistent and e_2, e_2' are also consistent, then:

(b.1) $\mathscr{P} \vdash_{CRWLF} e_1 \bowtie e_2 \Rightarrow \mathscr{P} \nvdash_{CRWLF} e_1' \not\bowtie e_2'$
(b.2) $\mathscr{P} \vdash_{CRWLF} e_1 \Diamond e_2 \Rightarrow \mathscr{P} \nvdash_{CRWLF} e_1' \not\Diamond e_2'$

(c) Given $\bar{t}, \bar{t}' \in CTerm_{\perp,\mathsf{F}} \times \ldots \times CTerm_{\perp,\mathsf{F}}$ pairwise consistent and $R \in \mathscr{P}_f$, if $\mathscr{P} \vdash_{CRWLF} f(\bar{t}) \lhd_{R} \mathscr{C}, f(\bar{t}') \lhd_{R} \mathscr{C}'$, then $\mathscr{C}$ and $\mathscr{C}'$ are consistent.

Now we prove (a), (b) and (c) simultaneously by induction on the size l of the derivation for $e \lhd \mathscr{C}$ in (a), $e_1 \bowtie e_2$ in (b.1), $e_1 \Diamond e_2$ in (b.2) and $f(\bar{t}) \lhd_{R} \mathscr{C}$ in (c).
$\underline{l = 1}$:

(a) The possible derivations in one step are:

- $e \lhd \{\perp\}$. This SAS is consistent with any other;
- $X \lhd \{X\}$. Then either $e' = X$ or $e' = \perp$, so the possibilities for $\mathscr{C}'$ are $\{X\}$ or $\{\perp\}$, both consistent with $\{X\}$;
- $c \lhd \{c\}$, where $c \in DC^0 \cup \{\mathsf{F}\}$. In this case e' must be c or $\perp$, whose possible *SAS*'s are $\{c\}$ and $\{\perp\}$, that are consistent with $\{c\}$.

(b) There is no derivation of the form $e \bowtie e'$ or $e \Diamond e'$ in one step.

(c) The possible derivations of the form $f(\bar{t}) \lhd_{R} \mathscr{C}$ are:

- $f(\bar{t}) \lhd_{R} \{\perp\}$. This *SAS* is consistent with any other;
- $f(\bar{t}) \lhd_{R} \{\mathsf{F}\}$, by means of rule 8, i.e. there exists some $R \equiv (f(\bar{s}) \rightarrow e \Leftarrow \overline{C}) \in \mathscr{P}_f$ and some i such that s_i and t_i have a $DC \cup \{\mathsf{F}\}$-clash at some position p. The *SAS* $\mathscr{C}'$ for $f(\bar{t}')$ using the function rule R must be done by

one of the rules 5 to 8:

– if rule 5 is used then $\mathscr{C}' = \{\bot\}$ that is consistent with $\mathscr{C}$;
– rule 6 is not applicable: t_i and t'_i are consistent because $\bar{t}$ and $\bar{t}'$ are pairwise consistent; then either t'_i at position p has the same constructor symbol as t_i (and then the clash with s_i remains), or t'_i at p or some of its ancestor positions has $\bot$. In both cases it is clear that there is not any c-instance of R for using rule 6;
– by rules 7 or 8 the *SAS* is $\{\mathsf{F}\}$ that is consistent with the initial one $\{\mathsf{F}\}$.

$\underline{l \Rightarrow l+1}$:

a) In $l+1$ steps the possible derivations for $e \lhd \mathscr{C}$ are:

- $\dfrac{e_1 \lhd \mathscr{C}_1 \quad \dots \quad e_n \lhd \mathscr{C}_n}{e = c(e_1,\dots,e_n) \lhd \{c(t_1,\dots,t_n)|\bar{t} \in \mathscr{C}_1 \times \dots \times \mathscr{C}_n\}}$ by rule 3, where $c \in DC^n$ $(n > 0)$. Then either $e' = \bot$, whose only possible *SAS* is $\{\bot\}$, that is consistent with any other, or $e' = c(e'_1,\dots,e'_n)$ with e_i and e'_n being consistent for $i \in \{1,\dots,n\}$ and the *SAS* is produced by rule 3:

$$\frac{e'_1 \lhd \mathscr{C}'_1 \quad \dots \quad e'_n \lhd \mathscr{C}'_n}{e' = c(e'_1,\dots,e'_n) \lhd \{c(t'_1,\dots,t'_n)|\bar{t}' \in \mathscr{C}'_1 \times \dots \times \mathscr{C}'_n\}}$$

By i.h. $\mathscr{C}'_i$ is consistent with $\mathscr{C}_i$ for all $i \in \{1,\dots,n\}$ and then it is clear that $\mathscr{C}$ and $\mathscr{C}'$ are also consistent.

- $\dfrac{e_1 \lhd \mathscr{C}_1 \quad \dots \quad e_n \lhd \mathscr{C}_n \quad f(\bar{t}) \lhd_R \mathscr{C}_{R,\bar{t}}}{e = f(e_1,\dots,e_n) \lhd \bigcup_{R\in\mathscr{P}_f,\bar{t}\in\mathscr{C}_1\times\dots\times\mathscr{C}_n} \mathscr{C}_{R,\bar{t}}}$ by rule 4. Then either $e' = \bot$ whose only possible *SAS* is $\{\bot\}$ that is consistent with any other, or $e' = f(e'_1,\dots,e'_n)$ with e_i, e'_i consistent for all $i \in \{1,\dots,n\}$. If the *SAS* for e' is generated by rule 1 of the calculus, the result would be clear and for rule 4 we have $\dfrac{e'_1 \lhd \mathscr{C}'_1 \quad \dots \quad e'_n \lhd \mathscr{C}'_n \quad f(\bar{t}') \lhd_R \mathscr{C}_{R,\bar{t}'}}{e'=f(e'_1,\dots,e'_n)\lhd \bigcup_{R\in\mathscr{P}_f,\bar{t}'\in\mathscr{C}'_1\times\dots\times\mathscr{C}'_n} \mathscr{C}_{R,\bar{t}'}}$
By i.h. $\mathscr{C}_i$ and $\mathscr{C}'_i$ are consistent for all $i \in \{1,\dots,n\}$, what means that for each $\bar{t} \in \mathscr{C}_1 \times \dots \times \mathscr{C}_n$ there exists $\bar{t}' \in \mathscr{C}'_1 \times \dots \times \mathscr{C}'_n$ consistent with $\bar{t}$. Again by i.h. we have that each *SAS* $\mathscr{C}_{R,\bar{t}}$ is consistent with $\mathscr{C}_{R,\bar{t}'}$ and it can be easily proved that $\mathscr{C} = \bigcup_{R\in\mathscr{P}_f,\bar{t}\in\mathscr{C}_1\times\dots\times\mathscr{C}_n} \mathscr{C}_{R,\bar{t}}$ is then consistent with $\mathscr{C}' = \bigcup_{R\in\mathscr{P}_f,\bar{t}'\in\mathscr{C}'_1\times\dots\times\mathscr{C}'_n} \mathscr{C}_{R,\bar{t}'}$.

- $\dfrac{e_1 \lhd \{\mathsf{F}\}}{e = fails(e_1) \lhd \{true\}}$ by rule 13. If $e' = \bot$ the result is clear, else $e' = fails(e'_1)$. Then the *SAS* for e' requires to obtain an *SAS* $\mathscr{C}'$ for e'_1. By i.h., $\mathscr{C}'$ must be consistent with $\{\mathsf{F}\}$ what means that $\mathscr{C}' = \{\mathsf{F}\}$ or $\mathscr{C}' = \{\bot\}$. Then, the possible *SAS*'s for e' are $\{\bot\}$ and $\{true\}$ (by the same rule 13), both consistent with $\{true\}$.

- $\dfrac{e_1 \lhd \mathscr{C}_1}{e = \mathit{fails}(e_1) \lhd \{\mathit{false}\}}$ by rule 14, such that there exists $t \in \mathscr{C}_1$ with $t \neq \bot$, $t \neq \mathsf{F}$. If $e' = \bot$ the result is clear. Otherwise, if $e' = \mathit{fails}(e'_1)$ the SAS for e' must be obtained by one of the rules 1, 13 or 14. By rule 1 it would be $\{\bot\}$ consistent with any other one; rule 13 would need to obtain the SAS $\{\mathsf{F}\}$ for e'_1, but this is not possible because it must be consistent with $\mathscr{C}_1$ by i.h., so rule 13 is not applicable; and rule 14 would provide the SAS $\{\mathit{false}\}$ for e', consistent with itself.

(b.1) If we have a derivation for $e_1 \bowtie e_2$ by rule 9, there exist two SASs $\mathscr{C}_{e_1}$ and $\mathscr{C}_{e_2}$ such that $e \lhd \mathscr{C}_{e_1}, e_2 \lhd \mathscr{C}_{e_2}$ and there exist $t \in \mathscr{C}_{e_1}, t' \in \mathscr{C}_{e_2}$ with $t \downarrow t'$. Now, let e'_1, e'_2 be consistent with e_1, e_2, respectively, and assume that $e'_1 \not\bowtie e'_2$ can be proved. We reason by contradiction. Since $e'_1 \not\bowtie e'_2$ is provable, we can prove $e'_1 \lhd \mathscr{C}_{e'_1}, e'_2 \lhd \mathscr{C}_{e'_2}$ such that for all $s \in \mathscr{C}_{e'_1}, s' \in \mathscr{C}_{e'_2}$ it will be $s \not\downarrow s'$.

By i.h. $\mathscr{C}_{e_1}$ is consistent with $\mathscr{C}_{e'_1}$, what implies that there exists $u \in \mathscr{C}_{e'_1}$ consistent with t, and then there exists v such that $v \sqsupseteq u, v \sqsupseteq t$. In a similar way, there exists $u' \in \mathscr{C}_{e'_2}$ consistent with t', so there exists v' such that $v' \sqsupseteq u', v' \sqsupseteq t'$.

As $u \in \mathscr{C}_{e'_1}$ and $u' \in \mathscr{C}_{e'_2}$ we would have $u \not\downarrow u'$; by monotonicity of $\not\downarrow$ we have $v \not\downarrow v'$, what implies $\neg(v \downarrow v')$. But monotonicity of $\downarrow$, together with $t \downarrow t', v \sqsupset t, v' \sqsupset t'$, implies $v \downarrow v'$, what is a contradiction.

(b.2) The case of $e_1 \ \diamond\ e_2$ proceeds similarly to $b.1$), using in this case monotonicity of $\uparrow$ and $\not\uparrow$.

(c) In $l+1$ steps the possible derivations for $f(\overline{t}) \lhd_R \mathscr{C}$ where $R \equiv (f(\overline{s}) \to e \Leftarrow \overline{C})$, are:

- $\dfrac{e\theta \lhd \mathscr{C} \quad \overline{C}\theta}{f(\overline{t}) \lhd_R \mathscr{C}}$ by rule 6, using the c-instance $R\theta$ ($\theta \in CSubst_{\bot,\mathsf{F}}$), such that $\overline{t} = \overline{s}\theta$. The derivation $f(\overline{t}') \lhd_R \mathscr{C}'$ must be done by one of the rules 5 to 8:

— if $f(\overline{t}') \lhd_R \{\bot\}$ by rule 5, it is clear that this SAS is consistent with $\mathscr{C}$;

— if the derivation is done by rule 6, it will have the form $\dfrac{e\theta' \lhd \mathscr{C}' \quad \overline{C}\theta'}{f(\overline{t}') \lhd_R \mathscr{C}'}$ using a c-instance $R\theta'$ of R. In particular, we have $\overline{t}' = \overline{s}\theta'$ and we also had $\overline{t} = \overline{s}\theta$. As $\overline{t}$ and $\overline{t}'$ are pairwise consistent, and $var(e) \subseteq var(\overline{s})$ it is not difficult to see that $e\theta$ and $e\theta'$ must be consistent. Then by i.h. (part a)) we deduce that $\mathscr{C}$ and $\mathscr{C}'$ are consistent SASs.

— rule 7 is not applicable: suppose that we have the derivation $\dfrac{\widetilde{C}_i\theta'}{f(\overline{t}') \lhd_R \{\mathsf{F}\}}$ using a c-instance $R\theta'$ of R and $C_i\theta'$ being a constraint of $\overline{C}\theta$. Analogously to the previous case, we have that both members of $C_i\theta'$ are consistent with the corresponding ones of $C_i\theta$; as $C_i\theta$ is provable then by i.h. (part *b*)), $\widetilde{C}_i\theta'$ is not provable, what means that rule 7 cannot be applied.

— rule 8 is not applicable: there cannot be a pair of $\overline{t}'$ and $\overline{s}$ with a $DC \cup \{\mathsf{F}\}$-clash because then the corresponding pair of $\overline{t}'$ and $\overline{s}\theta = \overline{t}$ would have the same clash (the substitution θ cannot make disappear the clash).

- $\dfrac{\widetilde{C}_i\theta}{f(\overline{t}) \lhd_R \{\mathsf{F}\}}$ by rule 7, being $R\theta$ a c-instance of the rule R such that $\overline{t} = \overline{s}\theta$. The derivation $f(\overline{t}') \lhd_R \mathscr{C}'$ can be done by one of the rules 5–8:

— by rule 5, the SAS is $\{\bot\}$ that is consistent with any other;

— it is not possible to use rule 6 because we would need to prove a constraint $C_i\theta'$ of a c-instance $R\theta'$ of R. As $\overline{s}\theta = \overline{t}$ and $\overline{s}\theta' = \overline{t}'$ are pairwise consistent and $var(C_i) \subseteq var(\overline{s})$, both members of $C_i\theta$ and $C_i\theta'$ will be also consistent. Then by i.h. (part *b*)), as $\widetilde{C}_i\theta$ is provable, $C_i\theta'$ will not be provable.

— if 7 or 8 applies we will have $\mathscr{C}' = \{\mathsf{F}\}$ that is consistent with $\mathscr{C} = \{\mathsf{F}\}$ (in fact, 8 would not be applicable). $\square$

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

Corollary 1
$\mathscr{P} \vdash_{CRWLF} e \diamondsuit e' \Rightarrow \mathscr{P} \nvdash_{CRWLF} e \tilde{\diamondsuit} e'$, for all $e, e' \in Term_{\bot,\mathsf{F}}$

This justifies indeed our description of $\not\bowtie$ and $\not\mathrel{<>}$ 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 cannot decrease when we add information to the expression itself. This applies also to the case of constraints: 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 $e, e', e_1, e_2, e_1', e_2' \in Term_{\bot,\mathsf{F}}$

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

Proof
Again we need to strengthen the result with a new part *c*)

c) Given $\overline{t}, \overline{t}' \in CTerm_{\bot,\mathsf{F}} \times \ldots \times CTerm_{\bot,\mathsf{F}}$ such that $t_i \sqsubseteq t_i'$ for all $i \in \{1, \ldots, n\}$ and $R \in \mathscr{P}_f$, if $f(\overline{t}) \lhd_R \mathscr{C}$ then $f(\overline{t}') \lhd_R \mathscr{C}$
We will prove parts (a), (b) and (c) simultaneously by induction on the size l of the derivation for $e \lhd \mathscr{C}$ in (a), $e_1 \diamondsuit e_2$ in (b) and $f(\overline{t}) \lhd_R \mathscr{C}$ in (c):
$\underline{l = 1}$:

(a) The derivation of $e \lhd \mathscr{C}$ in one step can be:

- $e \lhd \{\bot\}$, and it is clear that also $e' \lhd \{\bot\}$
- $X \lhd \{X\}$: then $e = e' = X$
- $c \lhd \{c\}$, $c \in DC^0$: then $e = e' = c$

(b) For $e_1 \diamondsuit e_2$ there are not possible derivations in one steps.

(c) For $f(\bar{t}) \lhd_R \mathscr{C}$ the derivations can be:

- $f(\bar{t}) \lhd_R \{\bot\}$, using rule 5. Then for all $\bar{t}'$ we have $f(\bar{t}') \lhd_R \{\bot\}$
- $f(\bar{t}) \lhd_R \{\mathsf{F}\}$, using rule 8. Then $R \equiv (f(\bar{s}) \rightarrow e \Leftarrow \overline{C})$ and $\bar{t}$ and $\bar{s}$ have a DC-clash at some position. If $\bar{t} \sqsubseteq \bar{t}'$ then $\bar{t}'$ and $\bar{s}$ have the same clash, and rule 8 allows to prove also $f(t') \lhd_R \{\mathsf{F}\}$.

$\underline{l \Rightarrow l+1}$:

(a) We distinguish three cases for the derivation of $e \lhd \mathscr{C}$:

- $e = c(e_1, \dots, e_n)$. Then the derivation of $e \lhd \mathscr{C}$ must use the rule 3 and take the form: $\dfrac{e_1 \lhd \mathscr{C}_1 \ \dots \ e_n \lhd \mathscr{C}_n}{c(e_1,\dots,e_n) \lhd \{c(t_1,\dots,t_n) | \bar{t} \in \mathscr{C}_1 \times \dots \times \mathscr{C}_n\}}$ Since $e \sqsubseteq e'$, e' must take the form $e' = c(e'_1, \dots, e'_n)$ with $e_1 \sqsubseteq e'_1, \dots, e_n \sqsubseteq e'_n$. By i.h. we have $e'_1 \lhd \mathscr{C}_1, \dots, e'_n \lhd \mathscr{C}_n$ and with the same rule 3 we can build a derivation for $c(\bar{e}') \lhd \mathscr{C}$.
- $e = f(e_1, \dots, e_n)$. Then the derivation of $e \lhd \mathscr{C}$ must use rule 4 and take the form: $\dfrac{e_1 \lhd \mathscr{C}_1 \ \dots \ e_n \lhd \mathscr{C}_n \ f(\bar{t}) \lhd_R \mathscr{C}_{R,\bar{t}}}{f(e_1,\dots,e_n) \lhd \bigcup_{R \in \mathscr{P}_f, \bar{t} \in \mathscr{C}_1 \times \dots \times \mathscr{C}_n} \mathscr{C}_{R,\bar{t}}}$ e' must take the form $e' = f(e'_1, \dots, e'_n)$ with $e_i \sqsubseteq e'_i$. By i.h. we will have $e'_1 \lhd \mathscr{C}_1, \dots, e'_n \lhd \mathscr{C}_n$ and then we have the same tuples $\bar{t}$, the same SASs $\mathscr{C}_{R,\bar{t}}$ and finally the same SAS for $f(\bar{e}')$.
- $e = \mathit{fails}(e_1)$. Then $e' = \mathit{fails}(e'_1)$. The derivation $e \lhd \mathscr{C}$ must be done by one of the rules 13 or 14, that require to obtain an SAS for e_1. By i.h. if $e_1 \lhd \mathscr{C}_1$ then $e'_1 \lhd \mathscr{C}_1$ and then the same rule (and only that) is applicable to obtain the same SAS for e', that will be $\{\mathit{true}\}$ if rule 13 is applicable or $\{\mathit{false}\}$ if rule 14 is applied.

(b) The derivation $e_1 \diamondsuit e_2$ with $\diamondsuit \in \{\bowtie, \not\bowtie, <>, \not<>\}$ will be done by generating the SAS's $e_1 \lhd \mathscr{C}_1$ and $e_2 \lhd \mathscr{C}_2$. By i.h. we have $e'_1 \lhd \mathscr{C}_1, e'_2 \lhd \mathscr{C}_2$ and then it is clear that $e'_1 \diamondsuit e'_2$ is also provable.

(c) We distinguish the following cases according to the rule used for the derivation of $f(\bar{t}) \lhd_R \mathscr{C}$:

- By rule 6 the derivation would be: $\dfrac{e\theta \lhd \mathscr{C} \ \overline{C}\theta}{f(t_1,\dots,t_n) \lhd_R \mathscr{C}}$ where the rule R is $R \equiv (f(s_1, \dots, s_n) \rightarrow e \Leftarrow \overline{C})$ and $\theta \in \mathit{Subst}_{\bot,\mathsf{F}}$ such that $\bar{s}\theta = \bar{t}$.
We show that the same rule 6 is applicable for generating an SAS for $f(\bar{t}')$ being $t_i \sqsubseteq t'_i$ for all $i \in \{1, \dots, n\}$. The idea is that if $t_i \sqsubseteq t'_i$ then t'_i is the result of replacing some subterms $\bot$ of t_i by c-terms more defined than $\bot$. As $s_i \in CTerm$ then the corresponding positions or some ancestors must

have variables in s_i. Then we can get a substitution $\theta' \in CSubst_{\perp,\mathsf{F}}$ such that $\theta \sqsubseteq \theta'$ and $s_i\theta' = t_i'$. A formal justification of this fact may be done by induction on the syntactic structure of t_i and, as $\overline{s}$ is a linear tuple, the result can be extended in such a way that $\overline{s_i}\theta' = \overline{t}'$.
We also have that $e\theta \sqsubseteq e\theta'$, so by i.h. we have $e\theta' \lhd \mathscr{C}$. As the constraints $\overline{C}\theta$ are provable and $\theta \sqsubset \theta'$, then by i.h. *b*), the constraints $\overline{C}\theta'$ will also be provable. So we can build a derivation for $f(\overline{t}') \lhd_R \mathscr{C}$ by rule 6.

- By rule 7 the derivation would be: $\dfrac{\widetilde{C_i}\theta}{f(\overline{t}) \lhd_R \{\mathsf{F}\}}$ where the rule R is $R \equiv (f(\overline{s}) \rightarrow e \Leftarrow C_1, \ldots, C_n), i \in \{1, \ldots, n\}$ and $\theta \in CSubst_{\perp,\mathsf{F}}$ is such that $\overline{s}\theta = \overline{t}$.
  As $\overline{t} \sqsubseteq \overline{t}'$, in a similar way as before there exists θ' such that $\overline{s}\theta' = \overline{t}'$ and by i.h. we can prove $\widetilde{C_i}\theta'$, what implies that we can build the derivation for $f(\overline{t}') \lhd_R \{\mathsf{F}\}$, using rule 7. $\square$

Remark
Monotonicity, as stated in Proposition 2, refers to the degree of evaluation of expressions 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, by adding new rules to it), 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) $\mathscr{P} \vdash_{CRWLF} e \lhd \mathscr{C} \Rightarrow \mathscr{P} \vdash_{CRWLF} e\theta \lhd \mathscr{C}\theta$
b) $\mathscr{P} \vdash_{CRWLF} e \Diamond e' \Rightarrow \mathscr{P} \vdash_{CRWLF} e\theta \Diamond e'\theta$

Proof
Again we need to strengthen the result, with a new part (c):

(c) $f(\overline{t}) \lhd_R \mathscr{C} \Rightarrow f(\overline{t})\theta \lhd_R \mathscr{C}\theta$, for any $\overline{t} \in CTerm_{\perp,\mathsf{F}} \times \ldots \times CTerm_{\perp,\mathsf{F}}$

We prove simultaneously the three parts by induction on the size l of the derivations.
$\underline{l = 1}$: in one step we can have the derivations $e \lhd \{\perp\}, c \lhd \{c\}$ ($c \in DC^0 \cup \{\mathsf{F}\}$) and $X \lhd \{X\}$. The property is obvious for the first two and the third follows from the fact that if $t \in CTerm_{\perp,\mathsf{F}}$ then $t \lhd \{t\}$ is provable (this can be proved by induction on the depth of the term t). Notice that $X\theta \in CTerm \subset CTerm_{\perp,\mathsf{F}}$, so $X\theta \lhd \{X\theta\}$.
$\underline{l \Rightarrow l + 1}$: now we can have the following derivations:

- by rule 3 we have $\dfrac{e_1 \lhd \mathscr{C}_1 \quad \ldots \quad e_n \lhd \mathscr{C}_n}{c(e_1, \ldots, e_n) \lhd \{c(t_1, \ldots, t_n) \mid \overline{t} \in \mathscr{C}_1 \times \ldots \times \mathscr{C}_n\}}$ By i.h. we have $e_i\theta \lhd \mathscr{C}_i\theta$ for all $i \in \{1, \ldots, n\}$ and again by rule 3 we can build a derivation for $c(\overline{e})\theta \lhd \{c(t_1, \ldots, t_n)\theta \mid \overline{t}\theta \in \mathscr{C}_1\theta \times \ldots \times \mathscr{C}_n\theta\}$

- by rule 4 we have $\dfrac{e_1 \lhd \mathscr{C}_1 \quad \dots \quad e_n \lhd \mathscr{C}_n \quad \dots \quad f(\overline{t}) \lhd_{\scriptscriptstyle R} \mathscr{C}_{R,\overline{t}} \quad \dots}{f(e_1,\dots,e_n) \lhd \bigcup_{R\in\mathscr{P}_f,\overline{t}\in\mathscr{C}_1\times\ldots\times\mathscr{C}_n} \mathscr{C}_{R,\overline{t}}}$ By i.h. we have $e_i\theta \lhd \mathscr{C}_i\theta$ for all $i \in \{1,\dots,n\}$ and $f(\overline{t})\theta \lhd_{\scriptscriptstyle R} \mathscr{C}_{R,\overline{t}}\theta$ for each $\overline{t}\theta \in \mathscr{C}_1\theta \times \ldots \times \mathscr{C}_n\theta$ and each rule $R \in \mathscr{P}_f$. So we can get a derivation for $f(\overline{e})\theta \lhd \bigcup_{R\in\mathscr{P}_f,\overline{t}\theta\in\mathscr{C}_1\theta\times\ldots\times\mathscr{C}_n\theta} \mathscr{C}_{R,\overline{t}\theta}$
- by rule 5 we have $\dfrac{}{f(\overline{t}) \lhd_{\scriptscriptstyle R} \{\perp\}}$ and it is clear $f(\overline{t})\theta \lhd_{\scriptscriptstyle R} \{\perp\}$
- by rule 6 we have $\dfrac{e\theta' \lhd \mathscr{C} \quad \overline{C}\theta'}{f(\overline{t}) \lhd_{\scriptscriptstyle R} \mathscr{C}}$ where $(f(\overline{s}) \to e \Leftarrow \overline{C}) \in R$ and $\theta' \in CSubst_{\perp,\mathsf{F}}$ is such that $f(\overline{t}) = f(\overline{s})\theta'$. For the call $f(\overline{t})\theta$ we can get the appropriate c-instance by composing θ' and θ, so $f(\overline{t})\theta = f(\overline{s})\theta'\theta$. By i.h. we have $e\theta'\theta \lhd \mathscr{C}\theta$ and $\overline{C}\theta'\theta$, and then $f(\overline{t})\theta \lhd_{\scriptscriptstyle R} \mathscr{C}\theta$ by the same rule 6.
- by rule 7 we have $\dfrac{e_i\theta' \tilde{\diamond} e_i'\theta'}{f(\overline{t}) \lhd_{\scriptscriptstyle R} \{F\}}$ where $(f(\overline{s}) \to e \Leftarrow \overline{C}) \in R$ and $\theta' \in CSubst_{\perp,\mathsf{F}}$ is such that $f(\overline{t}) = f(\overline{s})\theta'$. As before, for the call $f(\overline{t})\theta$ we can get the appropriate c-instance by composing θ' and θ, so $f(\overline{t})\theta = f(\overline{s})\theta'\theta$. By i.h. we have $e_i\theta'\theta\tilde{\diamond}e_i'\theta'\theta$, and then $f(\overline{t})\theta \lhd_{\scriptscriptstyle R} \{F\}$
- by rule 8 we have $\dfrac{}{f(t_1,\dots,t_n) \lhd_{\scriptscriptstyle R} \{F\}}$ where $R \equiv (f(s_1,\dots,s_n) \to e \Leftarrow \overline{C})$ and such that t_i and s_i have a $DC \cup \{\mathsf{F}\}$-clash for some $i \in \{1,\dots,n\}$. It is clear that $t_i\theta$ and s_i will have the same clash so $f(\overline{t})\theta \lhd_r \{\mathsf{F}\}$
- by rules 9 to 12, the derivation would have the form $e\diamond e'$. By i.h. we have $e\theta \lhd \mathscr{C}\theta \quad e'\theta \lhd \mathscr{C}'\theta$. Now, if we take $t \in \mathscr{C}, t' \in \mathscr{C}'$ and $t\mathfrak{R}t'$ holds (where $\mathfrak{R} \in \{\downarrow,\uparrow,\not\downarrow,\not\uparrow\}$), then $t\theta\mathfrak{R}t'\theta$ also holds, by Proposition 1. It follows that $e\theta\diamond e'\theta$.
- by rule 13 (or rule 14), it must be $e = \mathit{fails}(e_1)$. This rule requires to obtain an SAS for e_1, say $e_1 \lhd \mathscr{C}_1$. Then by i.h. $e_1\theta \lhd \mathscr{C}_1\theta$ and it is clear that rule 13 will be applicable to derive $e \lhd \{\mathit{true}\}$ (or $e \lhd \{\mathit{false}\}$ by rule 14). □

6 *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 with respect to these two emphasized aspects. To establish the relations between both calculus we consider in this section the class of programs defined for *CRWL*, i.e. rules cannot use the function *fails*. This means that rules 13 and 14 of the *CRWLF*-calculus are not considered here.

First, we show that the *CRWLF*-calculus indeed extends *CRWL*. Parts (a) and (b) of the next result show that statements $e \lhd \mathscr{C}$ 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) $\mathscr{P} \vdash_{CRWLF} e \lhd \mathscr{C} \Rightarrow \forall t \in \mathscr{C}, \mathscr{P} \vdash_{CRWL} \hat{e} \rightarrow \hat{t}$
b) $\mathscr{P} \vdash_{CRWL} \hat{e} \rightarrow t \Rightarrow \exists \mathscr{C}$ such that $t \in \mathscr{C}$ and $\mathscr{P} \vdash_{CRWLF} e \lhd \mathscr{C}$
c) $\mathscr{P} \vdash_{CRWLF} e \bowtie e' \Leftrightarrow \mathscr{P} \vdash_{CRWL} \hat{e} \bowtie \hat{e}'$
d) $\mathscr{P} \vdash_{CRWLF} e <> e' \Leftrightarrow \mathscr{P} \vdash_{CRWL} \hat{e} <> \hat{e}'$

To prove the property we split it into two separate lemmas. The first one contains (a), the right implication of (c) and (d) and a new part (e):

Lemma 2
Let $\mathscr{P}$ a *CRWLF*-program. Then:

(a) $\mathscr{P} \vdash_{CRWLF} e \lhd \mathscr{C} \Rightarrow \forall t \in \mathscr{C}, \mathscr{P} \vdash_{CRWL} \hat{e} \rightarrow \hat{t}$
(c) $\mathscr{P} \vdash_{CRWLF} e \bowtie e' \Rightarrow \mathscr{P} \vdash_{CRWL} \hat{e} \bowtie \hat{e}'$
(d) $\mathscr{P} \vdash_{CRWLF} e <> e' \Rightarrow \mathscr{P} \vdash_{CRWL} \hat{e} <> \hat{e}'$
(e) Given $\bar{t} \in CTerm_{\perp,\mathsf{F}} \times \ldots \times CTerm_{\perp,\mathsf{F}}$ and $R \in \mathscr{P}_f : \mathscr{P} \vdash_{CRWLF} f(\bar{t}) \lhd_R \mathscr{C} \Rightarrow \forall t \in \mathscr{C}, \mathscr{P} \vdash_{CRWL} \widehat{f(\bar{t})} \rightarrow \hat{t}$

Proof
We prove simultaneously all the parts by induction on the size l of the corresponding derivation:
$\underline{l = 1}$: the derivation can be:

- $\mathscr{P} \vdash_{CRWLF} e \lhd \{\perp\}$, and we have $\mathscr{P} \vdash_{CRWL} \hat{e} \rightarrow \perp$
- $\mathscr{P} \vdash_{CRWLF} X \lhd \{X\}$, we have $\hat{X} = X$ and $\mathscr{P} \vdash_{CRWL} X \rightarrow X$
- $\mathscr{P} \vdash_{CRWLF} c \lhd \{c\}$, where $c \in DC^0$ and we have $\hat{c} = c$ and $\mathscr{P} \vdash_{CRWL} c \rightarrow c$
- $\mathscr{P} \vdash_{CRWLF} \mathsf{F} \lhd \{\mathsf{F}\}$, we have $\hat{\mathsf{F}} = \perp$ and $\mathscr{P} \vdash_{CRWL} \perp \rightarrow \perp$
- $\mathscr{P} \vdash_{CRWLF} f(\bar{t}) \lhd_R \{\perp\}$, and we have $\mathscr{P} \vdash_{CRWL} \widehat{f(\bar{t})} \rightarrow \perp$
- $\mathscr{P} \vdash_{CRWLF} f(\bar{t}) \lhd_R \{\mathsf{F}\}$, and we have $\mathscr{P} \vdash_{CRWL} \widehat{f(\bar{t})} \rightarrow \perp$

$\underline{l \Rightarrow l+1}$: the derivation can be:

- $\mathscr{P} \vdash_{CRWLF} c(e_1, \ldots, e_n) \lhd \{\ldots, c(t_1, \ldots, t_n), \ldots\}$, then by the rule 3 of *CRWLF*, it must be $\mathscr{P} \vdash_{CRWLF} e_i \lhd \{\ldots, t_i, \ldots\}$. By i.h. we have $\mathscr{P} \vdash_{CRWL} \hat{e}_i \rightarrow \hat{t}_i$ and then we can build the derivation $\mathscr{P} \vdash_{CRWL} c(\widehat{e_1, \ldots, e_n}) \rightarrow c(\widehat{t_1, \ldots, t_n})$, by the rule 3 of *CRWL*.
- $\mathscr{P} \vdash_{CRWLF} f(e_1, \ldots, e_n) \lhd \{\ldots, t, \ldots\}$. This derivation must use the rule 4 of *CRWLF*, and then we will have the derivations $\mathscr{P} \vdash_{CRWLF} e_i \lhd \mathscr{C}_i$ for all $i \in \{1, \ldots, n\}$ and $f(\bar{t}) \lhd_R \mathscr{C}_{R,\bar{t}}$. It must be $t \in \mathscr{C}_{R,\bar{t}}$ for some $\bar{t} \in \mathscr{C}_1 \times \ldots \times \mathscr{C}_n$ and $R \in \mathscr{P}_f$. By i.h. we will have $\mathscr{P} \vdash_{CRWL} \widehat{f(\bar{t})} \rightarrow \hat{t}$
- $\mathscr{P} \vdash_{CRWLF} f(\bar{t}) \lhd_R \mathscr{C}$ by rule 6 of *CRWLF*, for which we take $R \equiv (f(\bar{s}) \rightarrow e \Leftarrow \overline{C}), \theta \in CSubst_{\perp,\mathsf{F}}$ such that $\bar{s}\theta = \bar{t}$. We can define $\theta' \in CSubst_{\perp}$ as $X\theta' = \perp$ if $X\theta = \mathsf{F}$ and $X\theta' = X\theta$, in other case. So we have $\bar{s}\theta' = \hat{\bar{t}}$, $e\theta' = \widehat{e\theta}$ and $\overline{C}\theta' = \widehat{\overline{C}\theta}$. Now we can take $(f(\bar{s}) \rightarrow e \Leftarrow \overline{C})\theta' \in [\mathscr{P}]_\perp$. We also have $e\theta \lhd \mathscr{C}$ and if $t \in \mathscr{C}$ by i.h. we have $\widehat{e\theta} \rightarrow \hat{t}$, or what is the same, $e\theta' \rightarrow \hat{t}$. Also by i.h. $\overline{C}\theta' = \widehat{\overline{C}\theta}$ is provable within *CRWL*, and therefore $f(\hat{\bar{t}}) \rightarrow \hat{t}$ by rule 4 of *CRWL*.

- $\mathscr{P} \vdash_{CRWLF} f(\overline{t}) \lhd_R \{\mathsf{F}\}$ and we have $\mathscr{P} \vdash_{CRWL} \widehat{f(\overline{t})} \to \bot$
- $\mathscr{P} \vdash_{CRWLF} e \bowtie e'$ using the rule 9 of $CRWLF$. Then we will have $\mathscr{P} \vdash_{CRWLF} e \lhd \mathscr{C}$, $\mathscr{P} \vdash_{CRWLF} e' \lhd \mathscr{C}'$ and there exist $t \in \mathscr{C}, t' \in \mathscr{C}'$ such that $t \downarrow t'$ (by definition of $\downarrow$ it is easy to see that $t = t'$). By i.h. we have $\mathscr{P} \vdash_{CRWL} \hat{e} \to \hat{t}$ and $\mathscr{P} \vdash_{CRWL} \hat{e}' \to \hat{t}$ and by rule 5 of $CRWL$ we have $\mathscr{P} \vdash_{CRWL} \hat{e} \bowtie \hat{t}$.
- $\mathscr{P} \vdash_{CRWLF} e <> e'$ using the rule 10 of $CRWLF$. Then we will have $\mathscr{P} \vdash_{CRWLF} e \lhd \mathscr{C}$, $\mathscr{P} \vdash_{CRWLF} e' \lhd \mathscr{C}'$ and there exist $t \in \mathscr{C}, t' \in \mathscr{C}'$ such that $t \uparrow t'$. By definition of $\uparrow$, t and t' have a DC-clash. By i.h. we have $\mathscr{P} \vdash_{CRWL} \hat{e} \to \hat{t}$ and $\mathscr{P} \vdash_{CRWL} \hat{e}' \to \hat{t}$ and by rule 5 of $CRWL$ we have $\mathscr{P} \vdash_{CRWL} \hat{e} <> \hat{t}$. □

We now state the second lemma for Proposition 4, in which part (b) and the left implications of parts (c) and (d) will be proved.

Lemma 3
For any $e, e' \in Term_{\bot,\mathsf{F}}$

b) $\mathscr{P} \vdash_{CRWL} \hat{e} \to t \Rightarrow \exists \mathscr{C}$ such that $t \in \mathscr{C}$ and $\mathscr{P} \vdash_{CRWLF} e \lhd \mathscr{C}$
c) $\mathscr{P} \vdash_{CRWL} \hat{e} \bowtie \hat{e}' \Rightarrow \mathscr{P} \vdash_{CRWLF} e \bowtie e'$
d) $\mathscr{P} \vdash_{CRWL} \hat{e} <> \hat{e}' \Rightarrow \mathscr{P} \vdash_{CRWLF} e <> e'$

Proof
We prove the three parts simultaneously by induction on the size l of the derivation:
$\underline{l = 1}$: the derivation can be:

- $\mathscr{P} \vdash_{CRWL} \hat{e} \to \bot$ and it is clear that $\mathscr{P} \vdash_{CRWLF} e \lhd \{\bot\}$
- $\mathscr{P} \vdash_{CRWL} X \to X$ and it is clear that $\mathscr{P} \vdash_{CRWLF} X \lhd \{X\}$
- $\mathscr{P} \vdash_{CRWL} c \to c$ with $c \in DC^0$ and it is clear that $\mathscr{P} \vdash_{CRWLF} c \lhd \{c\}$

$\underline{l \Rightarrow l + 1}$: the derivation can be of the following four forms:

- $\mathscr{P} \vdash_{CRWL} c(\widehat{e_1, \ldots, e_n}) \to c(t_1, \ldots, t_n)$ by rule 3 of $CRWL$ and then we have $\mathscr{P} \vdash_{CRWL} \hat{e}_i \to t_i$ for all $i \in \{1, \ldots n\}$. By i.h. we have $\mathscr{P} \vdash_{CRWLF} e_i \lhd \mathscr{C}_i$ with $t_i \in \mathscr{C}_i$ and by rule 3 of $CRWLF$ we have $\mathscr{P} \vdash_{CRWLF} c(e_1, \ldots, e_n) \lhd \mathscr{C}$ with $c(t_1, \ldots, t_n) \in \mathscr{C}$.
- $\mathscr{P} \vdash_{CRWL} f(\widehat{e_1, \ldots, e_n}) \to t$, then there must exist a rule $R = (f(\overline{s}) \to e \Leftarrow \overline{C}) \in \mathscr{P}$ and $\theta \in CSubst_\bot$ such that by rule 4 of $CRWL$ we will have the derivation
$$\frac{\hat{e}_1 \to s_1\theta \quad \ldots \quad \hat{e}_n \to s_n\theta \quad e\theta \to t \quad \overline{C}\theta}{f(\widehat{e_1, \ldots, e_n}) \to t}$$ By i.h. we have:

  (i) there exists $\mathscr{C}_i$ such that $\mathscr{P} \vdash_{CRWLF} e_i \lhd \mathscr{C}_i$ with $s_i\theta \in \mathscr{C}_i$
  (ii) there exists $\mathscr{C}'$ such that $\mathscr{P} \vdash_{CRWLF} e\theta \lhd \mathscr{C}'$ with $t \in \mathscr{C}'$
  (iii) $\mathscr{P} \vdash_{CRWLF} \overline{C}\theta$

  From (ii) and (iii), by rule 6 of $CRWLF$ we can build the derivation $\mathscr{P} \vdash_{CRWLF} f(s_i\theta) \lhd_R \mathscr{C}'$ using the c-instance $R\theta$. With this derivation and (i) we have $\mathscr{P} \vdash_{CRWLF} f(\overline{e}) \lhd \mathscr{C}$ such that $\mathscr{C}' \subseteq \mathscr{C}$, so $t \in \mathscr{C}$.
- $\mathscr{P} \vdash_{CRWL} \hat{e} \bowtie \hat{e}'$, using the rule 5 of $CRWL$. It follows that $\mathscr{P} \vdash_{CRWL} \hat{e} \to t$ and $\mathscr{P} \vdash_{CRWL} \hat{e}' \to t$ for some $t \in CTerm$. By i.h. $\mathscr{P} \vdash_{CRWLF} e \lhd \mathscr{C}$ and $\mathscr{P} \vdash_{CRWLF} e' \lhd \mathscr{C}'$ where $t \in \mathscr{C} \cap \mathscr{C}'$. Taking into account that $t \downarrow t$ for all $t \in CTerm$, by rule 9 of $CRWLF$ we can build a derivation for $\mathscr{P} \vdash_{CRWLF} e \bowtie e'$.

- $\mathcal{P} \vdash_{CRWL} \hat{e} \lhd\!\rhd \hat{e}'$, using the rule 6 of *CRWL*. It follows that $\mathcal{P} \vdash_{CRWL} \hat{e} \rightarrow t$ and $\mathcal{P} \vdash_{CRWL} \hat{e}' \rightarrow t'$ where $t, t' \in CTerm_\perp$ and have a DC-clash. By i.h. $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C}$ and $\mathcal{P} \vdash_{CRWLF} e' \lhd \mathcal{C}'$ where $t \in \mathcal{C}$ and $t' \in \mathcal{C}'$. By definition of $\uparrow$ and by rule 10 of *CRWLF* we can build a derivation for $\mathcal{P} \vdash_{CRWLF} e \lhd\!\rhd e'$. □

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 Proposition 4, which does not provide enough information about the relation between the denotation of an expression and each of its calculable SASs.

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

Proof
Assume $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C}$ and $\mathcal{P} \vdash_{CRWL} \hat{e} \rightarrow t$. By part *b*) of Proposition 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 SASs, 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
Assume $t \in [\![\hat{e}]\!]$. This means that $\mathcal{P} \vdash_{CRWL} \hat{e} \rightarrow t$, which in particular implies $t \in CTerm_\perp$. On the other hand, since $\mathcal{P} \vdash_{CRWLF} e \lhd \{\mathsf{F}\}$, we know from Proposition 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 $t = \perp$. □

7 Final discussion and future work

We have investigated how to deduce negative information from a wide class of functional logic programs. This is done by considering failure of reduction to head normal form, a notion playing a similar role, in the FLP setting, to that of negation as failure in logic programming, but having quite a different starting point. Negation as failure in LP can be seen mainly as an operational idea (existence of a finite, failed search tree) of which a logical interpretation can be given (successful negated atoms are logical consequences of the completion of the program). The operational view of negation leads to an immediate implementation technique for negation included in all Prolog systems: to solve the negation of a goal, try to solve the goal and succeed if this attempt ends in failure. Unfortunately, as it is well-known, this implementation of negation is logically sound only for ground goals (e.g. see Apt and Bol (1994)).

Our approach has been different: we have given a logical status to failure by proposing the proof calculus *CRWLF* (Constructor based ReWriting Logic with

Failure), which allows to deduce failure of reduction within *CRWL* (González *et al.* 1996; González *et al.* 1999), a well established theoretical framework for FLP.

We must emphasize the fact that *CRWLF* is *not* an operational mechanism for executing programs using failure, but a deduction calculus fixing the logical meaning of such programs. Exactly the same happens in González *et al.* (1996, 1999) with the proof calculus of *CRWL*, which determines the logical meaning of a FLP program, but not its execution. The operational procedure in *CRWL* is given by a narrowing-based goal solving calculus, which is proved to be sound and complete with respect to the proof calculus. Our idea with *CRWLF* is to follow a similar way: with the proof calculus as a guide, develop a narrowing-based operational calculus able to compute failures (even in presence of variables). We are currently working on this issue.

It is nevertheless interesting to comment that the operational approach to failure mentioned at the beginning of the section for the case of Prolog, can be also adopted for FLP, leading to a very easy implementation of failure: to evaluate $fails(e)$, try to compute a head normal form of e; if this fails, return *true*, otherwise return *false*. This is specially easy to be done in systems having a Prolog-based implementation like *Curry* or $\mathcal{TOY}$. We have checked that all the examples in section 2 are executable in $\mathcal{TOY}$ with this implementation of failure, if the function *fails* is only applied to ground expressions. For instance, the goal $safe(c) \bowtie T$ succeeds with answer $T = true$, and $safe(a) \bowtie T$ succeeds with answer $T = false$. If *fails* is applied to expressions with variables, this implementation is unsound. For instance, the goal $safe(X) \bowtie false$ succeeds without binding X, which is incorrect. The relationship between this kind of failure and *CRWLF* is an interesting issue to investigate, but it is outside the scope of this paper.

The most remarkable technical insight in *CRWLF* has been to replace the statements $e \to t$ of *CRWL* (representing a single reduction of e to an approximated value t) by $e \lhd \mathscr{C}$ (representing a whole, somehow complete, set $\mathscr{C}$ 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.

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 Bert and Echahed (1995) 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 Bert and Echahed (1995):

- Programs in Bert and Echahed (1995) 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 SASs for e determines in a precise sense (Propositions 4 and 5) the 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 SASs by such supersets. To be more concrete, if we adopt an abstract interpretation view of our *SAS*'s, it would be natural to see $\perp$ as standing for the set of all constructor terms (since $\perp$ is refinable to any value), and therefore to identify an *SAS* like $\mathscr{C} = \{\perp, z\}$ with $\mathscr{C}' = \{\perp\}$. But from $e \lhd \mathscr{C}$ we can deduce $e \bowtie z$, while it is not correct to do the same from $e \lhd \mathscr{C}'$. Therefore, the good properties of *CRWLF* with respect to *CRWL* are lost.

We see our work as a step in the research of a whole framework for dealing with failure in FLP. Some natural future steps are to develop model theoretic and operational semantics for programs making use of failure information. On the practical side, we are currently working on an implementation of failure for the FLP system $\mathscr{TOY}$ (López and Sánchez 1999a; Abengózar *et al.* 2002).

Acknowledgements

The authors have been partially supported by the Spanish CICYT (project TIC 2002-01167 'MEHLODIAS').

References

Abengózar-Carneros, M., Arenas, P., Caballero, R., Gil, A., González, J. C., Leach, J., López, F. J., Martí, N., Molina, J. M., Pimentel, E., Rodríguez, M., Roldán, M. M., Ruz, J. J. and Sánchez, J. (2002) *$\mathscr{TOY}$: A Multiparadigm Declarative Language. Version 2.0.* Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid.

Antoy, S. (1997) Optimal non-deterministic functional logic computations. *Proceedings 6th International Conference on Algebraic and Logic Programming (ALP'97).* LNCS 1298, pp. 16–30. Springer-Verlag.

Apt, K. R. (2000) *Logic Programming and Prolog.* Unpublished tutorial.

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

Arenas-Sánchez, P., López-Fraguas, F. J. and Rodríguez-Artalejo, M. (1999) Functional plus logic programming with built-in and symbolic constraints. *Proceedings First International Conference onPrinciples and Practice of Declarative Programming (PPDP'99).* LNCS 1702, pp. 152–169. Springer-Verlag.

Arenas-Sánchez, P. and Rodríguez-Artalejo, M. (2001) A general framework for lazy functional logic, programming with algebraic polymorphic types. *Theory and Practice of Logic Programming*, **1**(2), 185–245.

Bert, D. and Echahed, R. (1995) Abstraction of conditional term rewriting systems. *Proceedings International Logic Programming Symposium (ILPS'95).* pp. 162–176. MIT Press.

Clark, K. L. (1978) Negation as failure. In: H. Gallaire and J. Minker (eds.), *Logic and Databases*, pp. 293–322. Plenum Press.

Curry mailing list (2000) `curry@informatik.rwth-aachen.de`, October.

González-Moreno, J. C., Hortalá-González, T., López-Fraguas, F. J. and Rodríguez-Artalejo, M. (1996) A rewriting logic for declarative programming. *Proceedings 6th European Symposium on Programming (ESOP'96).* LNCS 1058, pp. 156–172. Springer-Verlag.

González-Moreno, J. C., Hortalá-González, T., López-Fraguas, F. J. and Rodríguez-Artalejo, M. (1999) An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, **40**(1), 47–87.

González-Moreno, J. C., Hortalá-González, T. and Rodríguez-Artalejo, M. (1997) A higher order rewriting logic for functional logic programming. *Proceedings 14th International Conference on Logic Programming (ICLP'97)*, pp. 153–167. MIT Press.

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

Hanus, M. (ed.) (1999) *Curry: An integrated functional logic language, Version 0.7.1*. Available at `http://www.informatik.uni-kiel.de/~mh/curry/report.html`.

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

López-Fraguas, F. J. and Sánchez-Hernández, J. (1999a) $\mathcal{TOY}$: A multiparadigm declarative system. *Proceedings 10th International Conference on Rewriting Techniques and Applications (RTA'99)*. LNCS 1631, pp. 244–247. Springer-Verlag.

López-Fraguas, F. J. and Sánchez-Hernández, J. (1999b) Disequalities may help to narrow. *Proceedings Joint Conference on Declarative Programming (APPIA-GULP-PRODE'99)*, pp. 89–104.

López-Fraguas, F. J. and Sánchez-Hernández, J. (2000) Proving failure in functional logic programs. *Proceedings 1st International Conference on Computational Logic (CL'2000)*. LNAI 1861, pp. 179–193. Springer-Verlag.

Moreno-Navarro, J. J. (1994) Default rules: An extension of constructive negation for narrowing-based languages. *Proceedings 12th International Conference on Logic Programming (ICLP'95)*, pp. 535–549. MIT Press.

Moreno-Navarro, J. J. (1996) Extending constructive negation for partial functions in lazy functional-logic languages. *Proceedings 5th International Workshop on Extensions of Logic Programming (ELP'96)*. LNAI 1050, pp. 213–227. Springer-Verlag.

Peyton-Jones, S. and Hughes, J. (eds.) (1999) *Haskell 98: A Non-strict, Purely Functional Language*. Available at `http://www.haskell.org`.

Stuckey P. J. (1991) Constructive negation for constraint logic programming. *Proceedings 6th Annual IEEE Symposium on Logic in Computer Science (LICS'91)*, pp. 328–339.

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