Functional Logic Programming with Failure: a Set-oriented View *

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

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

Abstract. Finite failure of computations plays an important role as programming construct in the logic programming paradigm, and it has been shown that this also extends to the case of the functional logic programming paradigm. In particular we have considered *CRWLF*, a previous proof-theoretic semantic framework able to deduce negative (failure) information from functional logic programs. The non-deterministic nature of functions considered in *CRWLF* leads naturally to set-valued semantic description of expressions. Here we reformulate the framework to stress that set flavour, both at syntactic and semantic levels. The given approach, for which we obtain equivalence results with respect to the previous one, increases the expressiveness for writing programs and (hopefully) clarifies the understanding of the semantics given to non-deterministic functions, since classical mathematical notions like union of sets or families of sets are used. An important step in the reformulation is a useful program transformation which is proved to be correct within the framework.

1 Introduction

Functional logic programming (*FLP* for short) [7] is a powerful programming paradigm trying to combine into a single language the nicest features of both functional and logic programming styles. Most of the proposals consider some kind of constructor-based rewrite systems as programs and use some kind of narrowing as operational mechanism. There are practical systems, like Curry [8] or $\mathcal{TOY}$ [11], supporting most of the features of functional and logic languages.

There is nevertheless a major aspect of logic programming still not incorporated to existing *FLP* proposals. It is *negation as failure*, a main topic of research in the logic programming field (see [4] for a survey), and a very useful expressive resource for writing logic programs.

There have been a few works devoted to this issue. In [13, 14] the work of Stuckey [16] about *constructive negation* is adapted to *FLP*, in strict and lazy versions. A different approach has been followed in [12], where a *C*onstructor Based *Re*Writing *L*ogic with *F*ailure (*CRWLF*) is proposed as a proof-theoretic

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

semantic framework for failure in *FLP*. Starting from *CRWL* [5, 6], a well established theoretical framework for *FLP* including a deduction calculus for reduction, *CRWLF* consists of a new proof calculus able to prove (computable cases of) failure of *CRWL*-provability corresponding to 'finite failure' of reduction. The non-deterministic nature of functions considered in *CRWL* and *CRWLF* leads naturally to set-valued semantic description of expressions. In this paper we reformulate the framework to stress that set flavour, both at syntactic and semantic levels.

The organization of the paper is as follows. We first give some motivations and discuss preliminary examples to help the understanding of the paper. Section 3 presents the *CRWLF* framework. In Section 4 we define and give correctness results for a program transformation which is needed for the rest of the paper. In Section 5 we reformulate in a set-oriented manner the *CRWLF* framework: at the syntactic level we introduce set-constructs like unions or indexed unions; we present a proof calculus for the new programs; we explain how to transform *CRWLF*-programs into this new syntax, and give a strong result of semantic equivalence.

2 Preliminary Discussion

• ***CRWLF* and non-deterministic functions:** *CRWL* [5, 6] models reduction by means of a relation $e \to t$, meaning operationally 'the expression e reduces to the term t' or semantically 't is an approximation of e's denotation'. The main technical insight of *CRWLF* was to replace the *CRWL*-statements $e \to t$ by the statements $e \lhd \mathcal{C}$, where $\mathcal{C}$ is what we call a *Sufficient Approximation Set* (*SAS*) for e, i.e., a finite set of approximations collected from all the different ways of reducing e to the extent required for the proof in turn. To prove failure of e corresponds to prove $e \lhd \{\mathsf{F}\}$, where F is a constant introduced in *CRWLF* to represent failure.

While each proof of *CRWL* concentrates on one particular way of reducing e, *CRWLF* obtains proofs related to all the possible ways of doing it. That the two things are not the same is because *CRWL*-programs are not required to be confluent, therefore defining functions which can be non-deterministic, i.e. yielding, for given arguments, different values coming from different possible reductions. The use of lazy non-deterministic functions is now a standard programming technique in systems like Curry or $\mathcal{TOY}$.

Non-determinism induces some kind of set-valued semantics for functions and expressions. As a simple example, assume the constructors z and s, and consider the non-confluent program:

$$\begin{array}{ll} f(X) = X & add(z, Y) = Y \\ f(X) = s(X) & add(s(X), Y) = s(add(X, Y)) \end{array}$$

For each X, $f(X)$ can be reduced to two possible values, X and $s(X)$. The expression $add(f(z), f(z))$ can be reduced in different ways to obtain three possible values: $z, s(z)$ and $s(s(z))$. This set-valued semantics is reflected in the

model semantics of *CRWL*, but not at the level of the proof calculus. *CRWL* is only able to prove separately $add(f(z), f(z)) \to z$, $add(f(z), f(z)) \to s(z)$ and $add(f(z), f(z)) \to s(s(z))$ (and partial approximations like $add(f(z), f(z)) \to s(\bot)$).

In contrast, the calculus of *CRWLF* is designed to collect sets of values. For instance, to prove failure of the expression $g(add(f(z), f(z)))$, when g is defined by $g(s(s(s(X)))) = z$, *CRWLF* needs to prove $add(f(z), f(z)) \triangleleft \{z, s(z), s(s(z))\}$. One of our main interests in the present work has been to reconsider some aspects of *CRWLF* to emphasize this set-minded view of programs.

• **Call-time choice semantics:** The semantics for non-deterministic functions adopted in *CRWL* is *call-time choice* [9]. Roughly speaking it means: to reduce $f(e_1, \ldots, e_n)$ using a rule of f, first choose one of the possible values of $e_1, \ldots, e_n$ and then apply the rule. Consider for instance the function $double(X) = add(X, X)$, and the expression $double(f(z))$, where f is the non-deterministic function defined above. The values for $double(f(z))$ come from picking a value for $f(z)$ and then applying the rule for *double*, obtaining then only two values, z and $s(s(z))$, but not $s(z)$.

To understand the fact that $double(f(z))$ and $add(f(z), f(z))$ are not the same in call-time choice, one must think that in the definition of *double* the variable X ranges over the universe of values (constructor terms), and not over the universe of expressions, which in general represent sets of values. This corresponds nicely to the classical view of functions in mathematics: if we define $double(n) = add(n, n)$ for natural numbers (values), then the equation $double(A) = add(A, A)$ is not valid for sets A of natural numbers, according to the usual definition of application of a function f to a subset of its domain: $f(A) = \{f(x) \mid x \in A\}$. In fact, we have $double(\{0, 1\}) = \{double(x) \mid x \in \{0, 1\}\} = \{0, 2\}$, while $add(\{0, 1\}, \{0, 1\}) = \{add(x, y) \mid x \in \{0, 1\}, y \in \{0, 1\}\} = \{0, 1, 2\}$ That is, mathematical practice follows call-time semantics.

The use of classical set notation can clarify the reading of expressions. For instance, instead of $double(f(z))$ we can write $\bigcup_{X \in f(z)} double(X)$. These kind of set-based changes in syntax is one of our contributions.

• **Overlapping programs:** To write programs in a set-oriented style we find the problem that different values for a function application can be spread out through different rules. This is not the case of non-overlapping rules, and the case of rules with identical (or variant) heads is also not problematic, since the rules can be merged into a single one: for the function f above, we can write $f(X) = \{X\} \cup \{s(X)\}$. The problem comes with definitions like $l(z, z) = z$, $l(z, X) = s(z)$, where the heads overlap but are not variants. To avoid such situations Antoy introduces in [3] the class of *overlapping inductively sequential* programs and proposes in [1] a transformation from general programs to that format. We consider also this class of programs when switching to set-oriented syntax, and propose a transformation with a better behavior than that of [1].

3 The *CRWLF* Framework

The *CRWLF* calculus that we show here is a slightly modified version of that in [12], in two aspects. First, for the sake of simplicity we have only considered programs with unconditional rules. Second, in [12] programs were 'positive', not making use of failure inside them. Here we allow programs to use a 'primitive' function $fails(\_)$ intended to be *true* when its argument fails to be reduced, and *false* otherwise. This behavior of *fails* is determined explicitly in the proof calculus.

The function *fails* is quite an expressive resource. As an application we show by an example how to express *default rules* in function definitions.

Example 1. In many pure functional 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:

$$f(z) = z \qquad f(X) = s(z)$$

The evaluation of the expression $f(z)$ in a functional language (like Haskell [15]), will produce the value z by the first rule. The second rule is not used for evaluating $f(z)$, even if pattern matching would succeed if the rule would be considered individually. This contrasts with functional logic languages (like Curry [8] or $\mathcal{TOY}$ [11]) which try to preserve the declarative reading of each rule. In such systems the expression $f(z)$ would be reduced, by applying in a non-deterministic way any of the rules, to the values z and $s(z)$.

To achieve the effect of default rules in *FLP*, an explicit syntactical construction *'default'* can be introduced, as suggested in [13]. The function f could be defined as:

$$\begin{array}{l} f(z) = z \\ \mathit{default}\ f(X) = s(z) \end{array}$$

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. Using the function *ifThen* (defined as $\mathit{ifThen}(\mathit{true}, X) = X$) and the predefined function *fails*, we can transform the previous definition into:

$$\begin{array}{ll} f(X) = f'(X) & \\ f(X) = \mathit{ifThen}(\mathit{fails}(f'(X)), s(z)) & \qquad f'(z) = z \end{array}$$

This definition achieves the expected behavior for f without losing the equational meaning of rules.

3.1 Technical Preliminaries

We assume a signature $\Sigma = DC_\Sigma \cup FS_\Sigma \cup \{fails\}$ where $DC_\Sigma = \bigcup_{n \in \mathbb{N}} DC^n_\Sigma$ is a set of *constructor* symbols containing at least *true* and $false$, $FS_\Sigma = \bigcup_{n \in \mathbb{N}} FS^n_\Sigma$

is a set of *function* symbols, all of them with associated arity and such that $DC_\Sigma \cap FS_\Sigma = \emptyset$, and *fails* $\notin DC \cup FS$ (with arity 1). We also assume a countable set $\mathcal{V}$ of *variable* symbols. We write $Term_\Sigma$ for the set of (total) *terms* (we say also *expressions*) built over Σ and $\mathcal{V}$ in the usual way, and we distinguish the subset $CTerm_\Sigma$ of (total) constructor terms or (total) *cterms*, which only make use of DC_Σ and $\mathcal{V}$. The subindex Σ will be usually omitted. Terms intend to represent possibly reducible expressions, while cterms 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 the sets $Term_\perp$ and $CTerm_\perp$ of (partial) terms and (partial) cterms respectively. Partial cterms represent the result of partially evaluated expressions; thus, they can be seen as approximations to the value of expressions. The signature $\Sigma_{\perp,\mathsf{F}}$ results of adding to $\Sigma_\perp$ a new constant F, to express failure of reduction. The sets $Term_{\perp,\mathsf{F}}$ and $CTerm_{\perp,\mathsf{F}}$ are defined in the natural way.

We will use three kind of substitutions $CSubst, CSubst_\perp$ and $CSubst_{\perp,\mathsf{F}}$ defined as applications from $\mathcal{V}$ into $CTerm, CTerm_\perp$ and $CTerm_{\perp,\mathsf{F}}$ respectively.

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

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

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

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

The intended meaning of $e \sqsubseteq e'$ is that e is less defined or has less information than e'. Notice that according to this F is maximal. Two expressions $e, e' \in Term_{\perp,\mathsf{F}}$ are *consistent* if there exists $e'' \in Term_{\perp,\mathsf{F}}$ such that $e \sqsubseteq e''$ and $e' \sqsubseteq e''$.

We extend the order $\sqsubseteq$ and the notion of consistency to sets of terms: given $\mathcal{C}, \mathcal{C}' \in CTerm_{\perp,\mathsf{F}}$, $\mathcal{C} \sqsubseteq \mathcal{C}'$ if for all $t \in \mathcal{C}$ there exists $t' \in \mathcal{C}'$ with $t \sqsubseteq t'$ and for all $t' \in \mathcal{C}'$ there exists $t \in \mathcal{C}$ with $t \sqsubseteq t'$. The sets $\mathcal{C}, \mathcal{C}'$ are consistent if there exists $\mathcal{C}''$ such that $\mathcal{C} \sqsubseteq \mathcal{C}''$ and $\mathcal{C}' \sqsubseteq \mathcal{C}''$.

A *CRWLF*-program $\mathcal{P}$ is a set of rewrite rules of the form $f(\overline{t}) \to e$, where $f \in FS^n$; $\overline{t}$ is a linear tuple (each variable in it occurs only once) of cterms; $e \in Term$ and $var(e) \subseteq var(\overline{t})$. We say that $f(\overline{t})$ is the *head* and e is the *body* of the rule. We write $\mathcal{P}_f$ for the set of defining rules of f in $\mathcal{P}$.

To express call-time choice, the calculus of the next section uses the set of c-instances of a rule R, defined as $[R]_{\perp,\mathsf{F}} = \{R\theta \mid \theta \in CSubst_{\perp,\mathsf{F}}\}$.

3.2 The Proof Calculus *CRWLF*

The proof calculus *CRWLF* defines the relation $e \lhd \mathcal{C}$ where $e \in Term_{\bot,\mathsf{F}}$ and $\mathcal{C} \subseteq CTerm_{\bot,\mathsf{F}}$; we say that $\mathcal{C}$ is a *Sufficient Approximation Set* (*SAS*) for the expression e. A *SAS* is a finite approximation to the denotation of an expression. For example, if f is defined as $f(X) \to X$, $f(X) \to s(X)$, then we have the sets $\{\bot\}$, $\{z, \bot\}$, $\{z, s(\bot)\}$, $\{\bot, s(\bot)\}$, $\{\bot, s(z)\}$ and $\{z, s(z)\}$ as finite approximations to the denotation of $f(z)$.

Table 1. Rules for *CRWLF*-provability

(1) $\dfrac{}{e \lhd \{\bot\}} \quad e \in Term_{\bot,\mathsf{F}}$ (2) $\dfrac{}{X \lhd \{X\}} \quad X \in \mathcal{V}$

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

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

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

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

(8) $\dfrac{e \lhd \{\mathsf{F}\}}{fails(e) \lhd \{true\}}$ (9) $\dfrac{e \lhd \mathcal{C}}{fails(e) \lhd \{false\}} \quad t \in \mathcal{C}, t \neq \bot, t \neq \mathsf{F}$

Rules for *CRWLF*-provability are shown in Table 1. Rules 1 to 7 are the restriction of the calculus in [12] to unconditional programs. Rules 8 and 9 define the function *fails* according to the specification given in Sect. 3.

The auxiliary relation $\lhd_R$ used in rule 4 depends on a particular program rule R, and is defined in rules 5 to 7. The function μ in rule 4 is a simplification function for *SAS*'s to delete irrelevant occurrences of F. It is defined as $\mu(\{\mathsf{F}\}) = \{\mathsf{F}\}$; $\mu(\mathcal{C}) = \mathcal{C} - \{\mathsf{F}\}$ otherwise (see [12] for a justification).

Given a program $\mathcal{P}$ and an expression e, we write $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C}$ to express that the relation $e \lhd \mathcal{C}$ is provable with respect to *CRWLF* and the program $\mathcal{P}$. The *denotation* of e is defined as $[\![e]\!]^{CRWLF} = \{\mathcal{C} \mid \mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C}\}$. Notice that the denotation of an expression is a set of sets of partial values. For the function f above we have $[\![f(z)]\!]^{CRWLF} = \{\{\bot\}, \{z, \bot\}, \{z, s(\bot)\}, \{\bot, s(\bot)\}, \{\bot, s(z)\}, \{z, s(z)\}\}$

The calculus *CRWLF* verifies the following properties:

Proposition 1. *Let $\mathcal{P}$ be a CRWLF-program. Then:*

a) Consistency of SAS's: $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C}, e \lhd \mathcal{C}' \Rightarrow \mathcal{C}$ *and* $\mathcal{C}'$ *are consistent. Moreover, there exists* $\mathcal{C}''$ *such that* $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C}''$*, with* $\mathcal{C} \sqsubseteq \mathcal{C}''$ *and* $\mathcal{C}' \sqsubseteq \mathcal{C}''$.
b) Monotonicity: $e \sqsubseteq e'$ *and* $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C} \Rightarrow \mathcal{P} \vdash_{CRWLF} e' \lhd \mathcal{C}$
c) Total Substitutions: $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C} \Rightarrow \mathcal{P} \vdash_{CRWLF} e\theta \lhd \mathcal{C}\theta$*, for* $\theta \in CSubst$.

These properties can be understood in terms of information. As we have seen, in general we can obtain different *SAS*'s for the same expression corresponding to different degrees of evaluation. Nevertheless, **Consistency** ensures that any two *SAS*'s for a given expression can be refined to a common one. **Monotonicity** says that the information that can be extracted from an expression can not decrease when we add information to the expression itself. And **Total Substitutions** shows that provability in *CRWLF* is closed under total substitutions.

4 Overlapping Inductively Sequential Programs

In [3], Antoy introduces the notion of *O*verlapping *I*nductively *S*equential programs (*OIS*-programs) based on the idea of definitional trees [2]. We give here an equivalent but slightly different definition.

Definition 1 (*OIS-CRWLF*-Programs). *A CRWLF-program is called* overlapping inductively sequential *if every pair of rules* $f(\overline{t}_1) \to e_1, f(\overline{t}_2) \to e_2$ *satisfies: the heads* $f(\overline{t}_1)$ *and* $f(\overline{t}_2)$ *are unifiable iff they are the same up to variable renaming.*

We next see that every *CRWLF*-program can be transformed into a semantically equivalent *OIS-CRWLF*-program.

4.1 Transformation of *CRWLF*-Programs into *OIS-CRWLF*-Programs

We need some usual terminologies about positions in terms. A position u in a term e is a sequence of positive integers $p_1 \cdot \ldots \cdot p_m$ that identifies the symbol of e at position u. We write $VP(e)$ for the set of positions in e occupied by variables. We say that a position u is *demanded by a rule* $f(\overline{t}) \to e$ if the head $f(\overline{t})$ has a constructor symbol of DC at position u. Given a set of rules $\mathcal{Q}$ and a position u, we say that u is *demanded* by $\mathcal{Q}$ if u is demanded by some rule of $\mathcal{Q}$, and we say that u is *uniformly demanded* by $\mathcal{Q}$ if it is demanded by all rules of $\mathcal{Q}$.

Definition 2 (Transformation of Sets of Rules). *The transformation algorithm is specified by a function* $\Delta(\mathcal{Q}, f(\overline{s}))$ *where:*

- $\mathcal{Q} = \{(f(\overline{t}_1) \to e_1), ..., (f(\overline{t}_n) \to e_n)\}$
- $f(\overline{s})$ *a* **pattern compatible with** $\mathcal{Q}$*, i.e.,* $\overline{s}$ *is a linear tuple of cterms and for all* $i \in \{1, ..., n\}$*,* $f(\overline{s})$ *is more general than* $f(\overline{t}_i)$ *(i.e.,* $\overline{s}\theta = \overline{t}_i$*, for some* $\theta \in CSubst$*).*

Δ *is defined by the following three cases:*

1. **Some position u in $VP(f(\overline{s}))$ is uniformly demanded by $\mathcal{Q}$** *(if there are several, choose any).*
   Let X be the variable at position u in $f(\overline{s})$. Let $C = \{c_1, ..., c_k\}$ be the set of constructor symbols at position u in the heads of the rules of $\mathcal{Q}$ and $\overline{s}_{c_i} = \overline{s}[X/c_i(\overline{Y})]$, where $\overline{Y}$ is a m-tuple of fresh variables (assuming $c_i \in DC^m$). For each $i \in \{1, ..., k\}$ we define the set $\mathcal{Q}_{c_i}$ as the set of rules of $\mathcal{Q}$ demanding c_i at position u.
   Return $\Delta(\mathcal{Q}_{c_1}, f(\overline{s}_{c_1})) \cup ... \cup \Delta(\mathcal{Q}_{c_k}, f(\overline{s}_{c_k}))$
2. **Some position in $VP(f(\overline{s}))$ is demanded by $\mathcal{Q}$, but none is uniformly demanded.**
   Let $u_1, ..., u_k$ be the demanded positions (ordered by any criterion). Consider the following partition (with renaming of function names in heads) over $\mathcal{Q}$:
   • Let $\mathcal{Q}^{u_1}$ be the subset of rules of $\mathcal{Q}$ demanding position u_1, where the function symbol f of the heads has been replaced by f^{u_1}, and $\overline{\mathcal{Q}^{u_1}} = \mathcal{Q} - \mathcal{Q}^{u_1}$.
   • Let $\mathcal{Q}^{u_2}$ be the subset of rules of $\overline{\mathcal{Q}^{u_1}}$ demanding position u_2, where the function symbol f of the heads have been replaced by f^{u_2} and let $\overline{\mathcal{Q}^{u_2}} = \overline{\mathcal{Q}^{u_1}} - \mathcal{Q}^{u_2}$.
   ...
   • Let $\mathcal{Q}^{u_k}$ be the subset of rules $\overline{\mathcal{Q}^{u_{k-1}}}$ demanding position u_k, where the function symbol f of the heads have been replaced by f^{u_k}.
   • And let $\mathcal{Q}^0$ be the subset of rules of $\mathcal{Q}$ that do not demand any position.
   Return $\mathcal{Q}^0 \cup \{f(\overline{s}) \to f^{u_1}(\overline{s}), ..., f(\overline{s}) \to f^{u_k}(\overline{s})\} \cup \Delta(\mathcal{Q}^{u_1}, f^{u_1}(\overline{s})) \cup ... \cup \Delta(\mathcal{Q}^{u_k}, f^{u_k}(\overline{s}))$

3. **No position in $VP(f(\overline{s}))$ is demanded by $\mathcal{Q}$,** *then* **Return $\mathcal{Q}$**

The initial call for transforming the defining rules of f will be $\Delta(\mathcal{P}_f, f(\overline{X}))$, and a generic call will have the form $\Delta(\mathcal{Q}, f^N(\overline{s}))$, where $\mathcal{Q}$ is a set of rules and $f^N(\overline{s})$ is a pattern compatible with $\mathcal{Q}$. We illustrate this transformation by an example:

Example 2. Consider the constants a, b and c and a function defined by the set of rules $\mathcal{P}_f = \{f(a, X) \to a, f(a, b) \to b, f(b, a) \to a\}$. To obtain the corresponding *OIS*-set of rules the algorithm works in this way:

$$\underbrace{\Delta(\{f(a,X) \to a, f(a,b) \to b, f(b,a) \to a\}, f(Y,Z))}_{\text{by 1}} =$$

$$\underbrace{\Delta(\{f(a,X) \to a, f(a,b) \to b\}, f(a,Z))}_{\text{by 2}} \cup \underbrace{\Delta(\{f(b,a) \to a\}, f(b,Z))}_{\text{by 1}} =$$

$$\{f(a,X) \to a\} \cup \{f(a,Z) \to f^2(a,Z)\} \cup \underbrace{\Delta(\{f^2(a,b) \to b\}, f(a,Z))}_{\text{by 1}} \cup$$

$$\underbrace{\Delta(\{f(b,a) \to a\}, f(b,a))}_{\text{by 3}} =$$

$$\{f(a,X) \to a\} \cup \{f(a,Z) \to f^2(a,Z)\} \cup \underbrace{\Delta(\{f^2(a,b) \to b\}, f(a,b))}_{\text{by 3}} \cup \{f(b,a) \to a\} =$$

$\{f(a,X) \to a\} \cup \{f(a,Z) \to f^2(a,Z)\} \cup \{f^2(a,b) \to b\} \cup \{f(b,a) \to a\} =$
$\{f(a,X) \to a, f(a,Z) \to f^2(a,Z), f(b,a) \to a, f^2(a,b) \to b\}$

Our transformation is quite related to the actual construction of the definitional tree [2, 10] of a function. A different algorithm to obtain an *OIS*-set of rules from a general set of rules is described in [1]. For the example above, such algorithm provides the following set of rules:

$$f(X,Y) \to f_1(X,Y) \mid f_2(X,Y) \qquad \begin{array}{l} f_1(a,Y) \to a \\ f_2(a,b) \ \to b \end{array} \qquad f_2(b,a) \to a$$

where the symbol '|' stands for a choice between two alternatives. This transformed set is worse than the one obtained by our transformation: for evaluating a call to f it begins with a search with two alternatives f_1 and f_2, even when it is not needed. For example, for evaluating $f(b,a)$, it tries both alternatives, but this reduction corresponds to a deterministic computation with the original program and also with our transformed one. The situation is clearly unpleasant if instead of b, we consider an expression e with a costly reduction to b.

Definition 3 (Transformation of Programs). *Given a CRWLF-program $\mathcal{P}$ we define the transformed program $\Delta(\mathcal{P})$ as the union of the transformed sets of defining rules for the functions defined in $\mathcal{P}$.*

It is easy to check that $\Delta(\mathcal{P})$ is indeed an *OIS-CRWLF*-program, and that $\Delta(\mathcal{P}) = \mathcal{P}$ if $\mathcal{P}$ is already an *OIS-CRWLF*-program.

Theorem 1 (Correctness of the Transformation). *For every CRWLF-program $\mathcal{P}$, $\Delta(\mathcal{P})$ is an OIS-CRWLF-program satisfying: for every $e \in Term_{\perp,\mathsf{F}}$ built over the signature of $\mathcal{P}$, $\mathcal{P} \vdash_{CRWLF} e \triangleleft \mathcal{C} \Leftrightarrow \Delta(\mathcal{P}) \vdash_{CRWLF} e \triangleleft \mathcal{C}$.*

5 A Set Oriented View of *CRWLF*: $\widehat{CRWLF}$

In this section we introduce the notion of **sas-expression** as a syntactical construction, close to classical set notation, that provides a clear "intuitive semantics" for the denotation of an expression.

5.1 Sas-Expressions

A sas-expression is intended as a construction for collecting values. These values may either appear explicitly in the construction or they can be eventually obtained by reducing function calls. Formally, a sas-expression $\mathcal{S}$ is defined as:

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

where $t \in CTerm_{\perp,\mathsf{F}}$, $\overline{t} \in CTerm_{\perp,\mathsf{F}} \times ... \times CTerm_{\perp,\mathsf{F}}$, $f \in FS^n$ and $\mathcal{S}_1, \mathcal{S}_2$ are sas-expressions.

The variable X in $\bigcup_{X \in \mathcal{S}_1} \mathcal{S}_2$ is called a produced variable. We can define formally the set $pvar(\mathcal{S})$ of **produced variables** of a sas-expression $\mathcal{S}$ as:

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

A sas-expression $\mathcal{S}$ is called **admissible** if it satisfies the following properties:

- if $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ then it must be $(var(\mathcal{S}_1) - pvar(\mathcal{S}_1)) \cap pvar(\mathcal{S}_2) = \emptyset$, and conversely $(var(\mathcal{S}_2) - pvar(\mathcal{S}_2)) \cap pvar(\mathcal{S}_1) = \emptyset$. The aim of this condition is to express that a variable can not appear in both $\mathcal{S}_1$ and $\mathcal{S}_2$ as produced and as not-produced variable.
- if $\mathcal{S} = \bigcup_{X \in r} \mathcal{S}$ then $X \notin var(r) \cup pvar(\mathcal{S})$ and $var(r) \cap pvar(\mathcal{S}) = \emptyset$

In the following we write $SasExp$ for the set of admissible sas-expressions.

We now define substitutions for non-produced variables.

Definition 4 (Substitutions for Sas-Expressions). *Given $\mathcal{S} \in SasExp$, $Y \notin pvar(\mathcal{S})$ and $s \in CTerm_{\perp,\mathsf{F}}$, the* **substitution** *$\mathcal{S}[Y/s]$ is defined on the structure of $\mathcal{S}$ as:*

- $\{t\}[Y/s] = \{t[Y/s]\}$
- $(\bigcup_{X \in f(\overline{t})} \mathcal{S}_1)[Y/s] = \bigcup_{X \in f(\overline{t})[Y/s]} \mathcal{S}_1[Y/s]$
- $(\bigcup_{X \in fails(\mathcal{S}_1)} \mathcal{S}_2)[Y/s] = \bigcup_{X \in fails(\mathcal{S}_1[Y/s])} \mathcal{S}_2[Y/s]$
- $(\bigcup_{X \in \mathcal{S}_1} \mathcal{S}_2)[Y/s] = \bigcup_{X \in \mathcal{S}_1[Y/s]} \mathcal{S}_2[Y/s]$
- $(\mathcal{S}_1 \cup \mathcal{S}_2)[Y/s] = \mathcal{S}_1[Y/s] \cup \mathcal{S}_2[Y/s]$

The expression $\mathcal{S}\theta$, where $\theta = [Y_1/s_1]...[Y_k/s_k]$, stands for the successive application of the substitutions $[Y_1/s_1], ..., [Y_k/s_k]$ to $\mathcal{S}$.

We will also use **set-substitutions** *for sas-expressions: given a set $\mathcal{C} = \{s_1, ..., s_n\} \in CTerm_{\perp,\mathsf{F}}$ we will write $\mathcal{S}[Y/\mathcal{C}]$ as a shorthand for the distribution $\mathcal{S}[Y/s_1] \cup ... \cup \mathcal{S}[Y/s_n]$.*

In order to simplify some expressions, we also introduce the following notation: given $h \in DC^n \cup FS^n$ and $\mathcal{C} = \{t_1, ..., t_m\} \subseteq CTerm_{\perp,\mathsf{F}}$, we will write $h(e_1, ..., e_{i-1}, \mathcal{C}, e_{i+1}, ..., e_n) \triangleleft \mathcal{C}'$ as a shorthand for $\mathcal{C}' = \mathcal{C}_1 \cup ... \cup \mathcal{C}_m$, where $h(e_1, ..., e_{i-1}, t_1, e_{i+1}, ..., e_n) \triangleleft \mathcal{C}_1$,..., $h(e_1, ..., e_{i-1}, t_m, e_{i+1}, ..., e_n) \triangleleft \mathcal{C}_m$. We will also use a generalized version of this notation and write $h(\mathcal{C}_1, ..., \mathcal{C}_n) \triangleleft \mathcal{C}$, where $\mathcal{C}_1, ..., \mathcal{C}_n \in CTerm_{\perp,\mathsf{F}}$.

5.2 Terms as Sas-Expressions

In this section we precise how to convert expressions into the set-oriented syntax of sas-expressions.

Definition 5 (Conversion into Sas-Expressions). *The sas-expression $\widehat{e}$ corresponding to $e \in Term_{\perp,\mathsf{F}}$ is defined inductively as follows:*

- $\widehat{X} = \{X\}$
- $\widehat{c(e_1, ..., e_n)} = \bigcup_{X_1 \in \widehat{e_1}} ... \bigcup_{X_n \in \widehat{e_n}} \{c(X_1, ..., X_n)\}$, *for every* $c \in DC^n \cup \{\bot, \digamma\}$, *where the variables* $X_1, ..., X_n$ *are fresh.*
- $\widehat{f(e_1, ..., e_n)} = \bigcup_{X_1 \in \widehat{e_1}} ... \bigcup_{X_n \in \widehat{e_n}} \bigcup_{X \in f(X_1,...,X_n)} \{X\}$, *for every* $f \in FS^n$, *where the variables* $X_1, ..., X_n$ *and* X *are fresh.*
- $\widehat{fails(e)} = \bigcup_{X \in fails(\widehat{e})} \{X\}$, *where the variable* X *is fresh.*

As an example of conversion we have

$$\widehat{double(f(X))} = \bigcup_{Y \in \widehat{f(X)}} \bigcup_{Z \in double(Y)} \{Z\} = \bigcup_{Y \in \bigcup_{Y_1 \in \{X\}} \bigcup_{Y_2 \in f(Y_1)} \{Y_2\}} \bigcup_{Z \in double(Y)} \{Z\}$$

This expression could be simplified to the shorter one $\bigcup_{Y \in f(X)} \bigcup_{Z \in double(Y)} \{Z\}$, but this is not needed for our purposes and we do not insist in that issue.

The set-based syntax of sas-expressions results in another benefit from the point of view of expressiveness. The notation $\bigcup_{X \in \mathcal{S}} \mathcal{S}'$ is a construct that binds the variable X and generalizes the sharing-role of (non-recursive) local definitions of functional programs. For instance, an expression like $\bigcup_{Y \in f(X)} \{c(Y, Y)\}$ expresses the sharing of $f(X)$ through the two occurrences of Y in $c(Y, Y)$. The same expression using typical *let* notation would be *let* $Y = f(X)$ *in* $c(Y, Y)$.

5.3 Denotational Semantics for Sas-Expressions: $\widehat{CRWLF}$

In this section we present the proof calculus $\widehat{CRWLF}$ for sas-expressions. This calculus is defined for programs with a set oriented notation. The idea is to start with a *CRWLF*-program $\mathcal{P}$, transform it into an *OIS-CRWLF*-program $\Delta(\mathcal{P})$ and then, transform the last into a $\widehat{CRWLF}$-program $\widehat{\Delta(\mathcal{P})}$, obtained by joining the rules with identical heads into a single rule whose body is a sas-expression obtained from the bodies of the corresponding rules. We have proved in Sect. 4 that the first transformation preserves the semantics. In this section we prove the same for the last one, obtaining then a strong equivalence between *CRWLF* and $\widehat{CRWLF}$.

Definition 6 ($\widehat{CRWLF}$-Programs). *A* $\widehat{CRWLF}$*-Program* $\widehat{\mathcal{P}}$ *is a set of* **non-overlapping** *rules of the form:* $f(\overline{t}) \twoheadrightarrow \mathcal{S}$, *where* $f \in FS^n$; $\overline{t}$ *is a linear tuple (each variable occurs only once) of cterms;* $s \in SasExp$ *and* $(var(\mathcal{S}) - pvar(\mathcal{S})) \subseteq var(\overline{t})$. *Non-overlapping means that there is not any pair of rules with unifiable heads in* $\widehat{\mathcal{P}}$.

According to this definition, it is easy to obtain the corresponding $\widehat{CRWLF}$-program $\widehat{\mathcal{P}}$ from a given *OIS-CRWLF*-program $\mathcal{P}$:

$$\widehat{\mathcal{P}} = \{f(\overline{t}) \twoheadrightarrow \widehat{e_1} \cup ... \cup \widehat{e_n} \mid f(\overline{t}) \to e_1, ..., f(\overline{t}) \to e_n \in \mathcal{P} \text{ and there is not any other rule in } \mathcal{P} \text{ with head } f(\overline{t})\}$$

Table 2. Rules for $\widehat{CRWLF}$-provability

(1) $\dfrac{}{\mathcal{S} \,\hat{\triangleleft}\, \{\bot\}} \quad \mathcal{S} \in SasExp$

(2) $\dfrac{}{\{X\} \,\hat{\triangleleft}\, \{X\}} \quad X \in \mathcal{V}$

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

(4) $\dfrac{\mathcal{S}' \,\hat{\triangleleft}\, \mathcal{C}' \quad \mathcal{S}[X/\mathcal{C}'] \,\hat{\triangleleft}\, \mathcal{C}}{\bigcup_{X \in f(\overline{t})} \mathcal{S} \,\hat{\triangleleft}\, \mathcal{C}} \quad (f(\overline{t}) \twoheadrightarrow \mathcal{S}') \in [\widehat{\mathcal{P}}]_{\bot,\mathsf{F}}$

(5) $\dfrac{\mathcal{S}[X/\mathsf{F}] \,\hat{\triangleleft}\, \mathcal{C}}{\bigcup_{X \in f(\overline{t})} \mathcal{S} \,\hat{\triangleleft}\, \mathcal{C}}$ for all $(f(s_1,...,s_n) \twoheadrightarrow \mathcal{S}') \in \widehat{\mathcal{P}}$, t_i and s_i have a $DC \cup \{\mathsf{F}\}$-clash for some $i \in \{1,...,n\}$

(6) $\dfrac{\mathcal{S}_1 \,\hat{\triangleleft}\, \{\mathsf{F}\} \qquad \mathcal{S}_2[X/true] \,\hat{\triangleleft}\, \mathcal{C}}{\bigcup_{X \in fails(\mathcal{S}_1)} \mathcal{S}_2 \,\hat{\triangleleft}\, \mathcal{C}}$

(7) $\dfrac{\mathcal{S}_1 \,\hat{\triangleleft}\, \mathcal{C}' \qquad \mathcal{S}_2[X/false] \,\hat{\triangleleft}\, \mathcal{C}}{\bigcup_{X \in fails(\mathcal{S}_1)} \mathcal{S}_2 \,\hat{\triangleleft}\, \mathcal{C}}$ there is some $t \in \mathcal{C}', t \neq \bot, t \neq \mathsf{F}$

(8) $\dfrac{\mathcal{S}_1 \,\hat{\triangleleft}\, \mathcal{C}' \qquad \mathcal{S}_2[X/\mathcal{C}'] \,\hat{\triangleleft}\, \mathcal{C}}{\bigcup_{X \in \mathcal{S}_1} \mathcal{S}_2 \,\hat{\triangleleft}\, \mathcal{C}}$

(9) $\dfrac{\mathcal{S}_1 \,\hat{\triangleleft}\, \mathcal{C}_1 \qquad \mathcal{S}_2 \,\hat{\triangleleft}\, \mathcal{C}_2}{\mathcal{S}_1 \cup \mathcal{S}_2 \,\hat{\triangleleft}\, \mathcal{C}_1 \cup \mathcal{C}_2}$

The non-overlapping condition is guaranteed because we join all the rules with the same head (up to renaming) into a single rule.

Table 2 shows the rules for $\widehat{CRWLF}$-provability. Rules 1, 2 and 3 have a natural counterpart in *CRWLF*. For rule 4 we must define the set of c-instances of rules of the program: $[\widehat{\mathcal{P}}]_{\bot,\mathsf{F}} = \{R\theta \mid R = (f(\overline{t}) \twoheadrightarrow \mathcal{S}) \in \widehat{\mathcal{P}}$ and $\theta \in CSubst_{\bot,\mathsf{F}} \mid_{var(\overline{t})}\}$. The notation $CSubst_{\bot,\mathsf{F}} \mid_{var(\overline{t})}$ stands for the set of substitutions $CSubst_{\bot,\mathsf{F}}$ restricted to $var(\overline{t})$. As $var(\overline{t}) \cap pvar(\mathcal{S}) = \emptyset$ the substitution is well defined according to Definition 4.

Notice that rule 4 uses a c-instance of a rule, and this c-instance is unique if it exists (due to the non-overlapping condition imposed to programs). If such c-instance does not exist, then by rule 5, the corresponding expression reduces to F. Rules 6 and 7 are the counterparts of 8 and 9 of *CRWLF*. Finally, rules 8 and 9 are due to the recursive definition of sas-expressions and have a natural reading.

Given a $\widehat{CRWLF}$-program $\widehat{\mathcal{P}}$ and $\mathcal{S} \in SasExp$ we write $\widehat{\mathcal{P}} \vdash_{\widehat{CRWLF}} \mathcal{S} \,\hat{\triangleleft}\, \mathcal{C}$ if the relation $\mathcal{S} \,\hat{\triangleleft}\, \mathcal{C}$ is provable with respect to $\widehat{CRWLF}$ and the program $\widehat{\mathcal{P}}$. The denotation of $\mathcal{S}$ is defined as $[\![\mathcal{S}]\!]^{\widehat{CRWLF}} = \{\mathcal{C} \mid \mathcal{S} \,\hat{\triangleleft}\, \mathcal{C}\}$.

Example 3. Assume the *OIS-CRWLF*-program:

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

The corresponding $\widehat{CRWLF}$-program $\widehat{\mathcal{P}}$ is:

$$
\begin{array}{ll}
add(z,Y) & \twoheadrightarrow \{Y\} \\
add(s(X),Y) & \twoheadrightarrow \bigcup_{A\in\bigcup_{B\in\{X\}}\bigcup_{C\in\{Y\}}\bigcup_{D\in add(B,C)}\{D\}}\{s(A)\} \\
double(X) & \twoheadrightarrow \bigcup_{A\in\{X\}}\bigcup_{B\in\{X\}}\bigcup_{C\in add(A,B)}\{C\} \\
f(X) & \twoheadrightarrow \{X\}\cup\bigcup_{A\in\{X\}}\{s(A)\}
\end{array}
$$

Within *CRWLF* we can prove $double(f(z))\triangleleft\{z,s(s(z))\}$, and within $\widehat{CRWLF}$ we can obtain the same *SAS* by proving $\widehat{double(f(z))} \mathrel{\hat{\triangleleft}} \{z,s(s(z))\}$. Let us sketch the form in which this proof can be done. First, we have:

$$\widehat{double(f(z))} = \bigcup_{A\in\bigcup_{C\in\{z\}}\bigcup_{D\in f(C)}\{D\}}\bigcup_{B\in double(A)}\{B\}$$

By rule 8 of $\widehat{CRWLF}$ this proof is reduced to the proofs:

$$\bigcup_{C\in\{z\}}\bigcup_{D\in f(C)}\{D\} \mathrel{\hat{\triangleleft}} \{z,s(z)\} \qquad (\varphi_1)$$

$$\bigcup_{B\in double(z)}\{B\}\cup\bigcup_{B\in double(s(z))}\{B\} \mathrel{\hat{\triangleleft}} \{z,s(z)\} \qquad (\varphi_2)$$

By rule 8 (φ_1) is reduced to the proofs $\{z\} \mathrel{\hat{\triangleleft}} \{z\}$ and $\bigcup_{D\in f(z)}\{D\} \mathrel{\hat{\triangleleft}} \{z,s(z)\}$. The first is done by rule 3 and the other is reduced by rule 4. On the other hand, by rule 9 the proof (φ_2) can be reduced to the proofs:

$$\bigcup_{B\in double(z)}\{B\} \mathrel{\hat{\triangleleft}} \{z\} \qquad (\varphi_3) \qquad\qquad \bigcup_{B\in double(s(z))}\{B\} \mathrel{\hat{\triangleleft}} \{s(s(z))\} \qquad (\varphi_4)$$

Both (φ_3) and (φ_4) proceed by rule 4 in a similar way. We fix our attention in (φ_4) which, using the rule for *double*, is reduced to:

$$\bigcup_{A\in s(z)}\bigcup_{B\in s(z)}\bigcup_{C\in add(A,B)}\{C\} \mathrel{\hat{\triangleleft}} \{s(s(z))\}$$

and then, by two applications of rule 8 to: $\bigcup_{C\in add(s(z),s(z)}\{C\} \mathrel{\hat{\triangleleft}} \{s(s(z))\}$. Now, rule 4 uses the first defining rule for *add* and the proof is reduced to:

$$\bigcup_{A\in\bigcup_{B\in\{z\}}\bigcup_{C\in\{s(z)\}}\bigcup_{D\in add(B,C)}\{D\}}\{s(A)\} \mathrel{\hat{\triangleleft}} \{s(s(z))\}$$

By rule 8 it is reduced to:

$$\bigcup_{B\in\{z\}}\bigcup_{C\in\{s(z)\}}\bigcup_{D\in add(B,C)}\{D\} \mathrel{\hat{\triangleleft}} \{s(z)\} \quad (\varphi_5) \qquad\qquad \{s(s(z))\} \mathrel{\hat{\triangleleft}} \{s(s(z))\} \quad (\varphi_6)$$

The proof (φ_6) is done by successive applications of rule 3 and (φ_5) is reduced by rule 8 (twice) to $\bigcup_{D\in add(z,s(z))}\{D\} \mathrel{\hat{\triangleleft}} \{s(z)\}$. This last proceeds by applying the first defining rule of *add* by means of rule 4.

5.4 *CRWLF* & $\widehat{CRWLF}$

We show here the strong semantic equivalence between *CRWLF* and $\widehat{CRWLF}$.

Lemma 1 (Semantic Equivalence of *CRWLF* and $\widehat{CRWLF}$). *Let $\mathcal{P}$ be an OIS-CRWLF-program and $\widehat{P}$ be the corresponding $\widehat{CRWLF}$-program. Let $e \in Term_{\perp,\mathcal{F}}$ and $\widehat{e} \in SasExp$ be the corresponding sas-expression. Then*

$$\mathcal{P} \vdash_{CRWLF} e \triangleleft \mathcal{C} \Leftrightarrow \widehat{\mathcal{P}} \vdash_{\widehat{CRWLF}} \widehat{e} \mathrel{\hat{\triangleleft}} \mathcal{C}.$$

As a trivial consequence of this lemma, we arrive at our final result:

Theorem 2. *Let $\mathcal{P}$ be a general CRWLF-program and $\widehat{\Delta(\mathcal{P})}$ be the corresponding $\widehat{CRWLF}$-program. Let e be an expression built over the signature of $\mathcal{P}$ and $\widehat{e}$ be the corresponding sas-expression. Then: $\mathcal{P} \vdash_{CRWLF} e \lhd \mathcal{C} \Leftrightarrow \widehat{\Delta(\mathcal{P})} \vdash_{\widehat{CRWLF}} \widehat{e} \mathrel{\hat{\lhd}} \mathcal{C}$.*

As a consequence, denotations of expressions are preserved in the transformation process, i.e., $[\![e]\!]^{CRWLF} = [\![\widehat{e}]\!]^{\widehat{CRWLF}}$ (referred to $\mathcal{P}$ and $\widehat{\Delta(\mathcal{P})}$, respectively). So, the properties about consistency, monotonicity and substitutions of Prop. 1 are preserved in $\widehat{CRWLF}$ when considering expressions and the corresponding sas-expressions.

6 Conclusions

We have extended and reformulated *CRWLF* [12], a proof-theoretic framework designed to deduce failure information from positive functional logic programs (i.e., programs not making use of failure inside them). To allow programs the use of failure, we have introduced a built-in function *fails*(_), and extended the proof calculus to deal with it.

We have discussed the declarative meaning of functions defined in programs. Since functions can be non-deterministic, they are in general set-valued. Each rule in the program defines (partially, since there can be more rules) a function as a mapping from (tuples of) constructor terms to sets of constructor terms. If we try to re-write the defining rules of a function f to express directly which is the value (set of constructor terms) of applying f to given arguments, we face the problem that this set can be distributed among different overlapping rules. To overcome this problem we have considered the class of overlapping inductively sequential programs [3] in which overlapping rules are always variants. We have defined a transformation of general programs into such kind of programs and proved that the transformation preserves the semantics, which constitutes itself an interesting application of the developed formal framework. Our transformation behaves better than that proposed in [1], if the transformed program is going to be used in existing systems like Curry [8] or $\mathcal{TOY}$ [11].

To stress the set-theoretic reading of programs, we have introduced set-oriented syntactic constructs to be used in right hand sides of rules, like set braces, union of sets, or union of indexed families of sets. This provides a more intuitive reading of programs in terms of classical mathematical notions, close to the intended semantics. As additional interesting point of this new syntax, indexed unions are a binding construct able to express sharing at the syntactic level, playing a role similar to local (*let* or *where*) definitions. As far as we know, this is the first time that some kind of local definitions are incorporated to a formal semantic framework for functional logic programming.

Our last contributions have been a transformation of overlapping inductively sequential programs into programs with set-oriented syntax, and a specific proof calculus for the latter, by means of which we prove that the transformation

preserves the semantics of programs. Apart from any other virtues, we have strong evidence that these new set-oriented syntax and proof calculus are a better basis for an ongoing development of an operational (narrowing based) semantics and subsequent implementation of a functional logic language with failure.

References

[1] S. Antoy Constructor-based Conditional Narrowing. To appear in Proc. PPDP'01, Springer LNCS.

[2] S. Antoy Definitional Trees. In Proc. ALP'92, Springer LNCS 632, pages 143-157, 1992.

[3] S. Antoy Optimal Non-Deterministic Functional Logic Computations. In Proc. ALP'97, Springer LNCS 1298, pages 16-30, 1997.

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

[5] J.C. González-Moreno, T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. A Rewriting Logic for Declarative Programming. In *Proc. ESOP'96*, pages 156–172. Springer LNCS 1058, 1996.

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

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

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

[9] H. Hussman. Non-Determinism in Algebraic Specifications and Algebraic Programs Birkhäuser Verlag, 1993.

[10] R. Loogen, F.J. López, M. Rodríguez. A demand driven computation strategy for lazy narrowing. *Proc. PLILP'3*, Springer LNCS 714, 184–200, 1993.

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

[12] F.J. López-Fraguas and J. Sánchez-Hernández. Proving Failure in Functional Logic Programs In *Proc CL'2000*, Springer LNAI 1861, pages 179–193, 2000.

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

[14] J.J. Moreno-Navarro. Extending constructive negation for partial functions in lazy functional-logic languages. In *Proc. ELP'96*, pages 213–227. Springer LNAI 1050, 1996.

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

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