Disequalities May Help to Narrow

F.J. López Fraguas J. Sánchez Hernández

Dep. Sistemas Informáticos y Programación, Univ. Complutense de Madrid
Fac. Matemáticas, Av. Complutense s/n, 28040 Madrid
email: {fraguas,jaime}@sip.ucm.es Phone: +34 91 3944429 Fax: +34 91 3944602

Abstract

A constructor-based rewriting logic (CRWL) has been recently proposed as an appropriate basis for functional logic programming. In this paper we extend such framework, to cope with disequality constraints, which are a useful and expressive resource from the point of view of programming. As an important application of the enhanced framework, we show how to express within it a sophisticated narrowing strategy which is widely accepted to be appropriate for lazy functional logic languages, but which previously suffered the lack of a formal justification in the original framework. This is achieved by means of a simple, incremental program transformation which can be proved to preserve the semantics of the original program.

Keywords: Functional logic programming, disequality constraints, narrowing, rewriting logic, non-deterministic functions

1 Introduction

Multiparadigm declarative programming has been subject of active research, specially for the case of functional logic programming (FLP in short; see [12] for a survey). Recently [10, 11], an approach to FLP has been proposed which takes possibly non-deterministic non-strict functions as the fundamental notion. Programs are theories in a constructor-based rewriting logic (CRWL) for which proof and model theories are given, as well as a sound and complete lazy narrowing calculus. The interest of non-deterministic functions for programming has been demonstrated by, e.g., Hussmann [16], and the particular usefulness of the CRWL-approach to that issue is shown for instance in [5, 6], where practical programs and programming methodologies have been developed using $\mathcal{TOY}$ [21], a system which implements (an enhancement of) the CRWL framework.

In this work we extend the CRWL framework by allowing disequality constraints – which are a very expressive resource – to appear both in programs and answers.

The authors have been partially supported by the Spanish CICYT (project TIC 98-0445-C03-02 'TREND') and the ESPRIT Working Group 22457 (CCL-II).

Although there exists deep theoretical work on unification and disunification problems (see [17, 7] for surveys), it is valid for equational logic and therefore does not readily apply to our setting.

In the next two sections we motivate the interest of our work, discuss some related one and clarify our contributions. Sect. 4 contains some preliminaries and notations. Sect. 5 presents the rewriting logic with disequalities which constitutes the semantic foundation of our proposal. Sect. 6 is devoted to the operational semantics in the form of a sound and complete calculus for goal solving which combines lazy narrowing with constraint solving. In Sect. 7 we discuss some important drawbacks of our narrowing calculus from the point of view of efficiency. Then we propose and prove the correctness of a program transformation which makes a simple use of disequalities and that, without the need of modifying the calculus, achieves the effect of an efficient (previously known) strategy. In Sect. 8 we give some conclusions. Due to space limitations, proofs are not included or simply sketched; complete proofs, together with other auxiliary technical definitions and results, can be found in a longer version [22].

2 The problem of equality in FLP

A prominent feature of narrowing-based FLP, when compared to functional programming, is the possibility of performing reductions over expressions containing variables, which may become instantiated in the process. This is roughly what narrowing does. It is usual to speak of *reversibility* or *multiples modes of use* of functions when referring to this ability, one of the nicest that FLP shares with logic programming. But reversibility is not easy to be kept in FLP programs requiring to make equality tests, which are very usual in practice.

Consider, for instance, the following standard definition:

$$\begin{array}{ll} member(X, [\,]) & = false \\ member(X, [Y|Ys]) & = if\ (X\ eq\ Y)\ then\ true\ else\ member(X, Ys) \end{array}$$

The symbol eq expresses here some suitable notion for equality. In presence of lazy languages allowing non terminating but still meaningful programs, the sensible notion (see e.g. [12]) for eq is that of *strict equality*, which means that e eq e' reduces to *true* if both e and e' are reducible to the same data term, and to $false$ if e and e' can be reduced to some extent as to detect inconsistency, i.e., different constructor symbols at the same position in e and e'.

Now, what if we consider an expression containing variables like $member(X, [Y])$? One possible reduction for it gives the value *true* together with the substitution X/Y which can be also seen as a constraint $X = Y$. But $member(X, [Y])$ can also be reducible to $false$ if X and Y are given inconsistent values. An attempt of covering such values by means of substitutions results in infinitely many answers. For instance, if the signature includes the constructors *0* and *s*, among the answers we would find

$$X = 0, Y = s(U);\ X = s(U), Y = 0;\ X = s(0), Y = s(s(U));\ X = s(s(U)), Y = s(0); \ldots$$

This family of answers cannot be replaced by any equivalent finite set of substitutions. The situation changes drastically if we consider disequality constraints, since *one single* disequality, namely $X \neq Y$, collects all the information embodied in all those substitutions.

It seems then interesting to be able to explicitly handle disequality constraints (it is recognized so, for instance, in [15]). This has not been done in most of the works related to FLP which use the notion of strict equality. In some cases, like in [2, 9, 23, 24], the situation is even worse, since the treatment given there to equality implies that even the positive solution $X = Y$ is split into an infinite number of ground substitutions. Other approaches, like [10, 19], are only able to deal properly with the positive case.

Thus, our **first contribution** has been to enhance the CRWL framework of [10, 11], by extending both the logic and the lazy narrowing calculus there with new rules for dealing with disequalities, and obtaining correctness and completeness results for the narrowing calculus.

This is not the first time that the problem of disequalities has been addressed in the context of lazy FLP. [18] incorporated disequality constraints to a restricted class of Babel programs [24] and proposed an abstract machine for implementing the resulting language. That work focused on implementation and lacked any formal theoretical treatment. In [4] the approach to disequality was closer to our present work, but there are still many differences: the framework assumed by [4] was CFLP(X) [20], a general scheme conceived for constraint functional logic programming, more general than CRWL in some aspects (CFLP(X) serves for many constraint systems) but less general in others (CRWL functions can be non-deterministic, CRWL programs have a proof theoretic semantics in addition to model theoretic semantics). Furthermore, we obtain here stronger completeness results for the operational semantics, both with respect to the generality of solutions (in [4] only completeness with respect to ground solutions was obtained), and with respect to the narrowing strategy ([4] only dealt with a naive strategy).

3 The problem of the narrowing strategy

Our narrowing calculus, as happens with the case of [10, 11], is not intended to be directly applied as operational model, since it implies a 'naive' narrowing strategy in which many computations may be wasted (see Section 7 for further explanations). In [10, 11] this drawback is pointed out but not technically solved. In practice we must use more efficient strategies, like the *Demand Driven Strategy* (*DDS*) of [19], which is very close to *needed narrowing* [2, 3]; both are based on the notion of *definitional tree* [1]. Experimental data related to the impact of adopting such a strategy can be found in [13]. Real implementations, like the systems $\mathcal{TOY}$ [21] or *Curry* [15], use *DDS* (combined with *residuation* for the case of *Curry*; see [14]).

Our **second contribution** has been to address the question of how to accommodate *DDS* within the CRWL framework. For this purpose, instead of reformulating the narrowing calculus from the scratch, we have followed a different approach: we define a very simple program transformation (that makes use of disequalities even

if they were absent in the original program) which has the effect of splitting into small steps the 'demand driving' effect of definitional trees. We prove that the transformed program is semantically equivalent to the original one. We then propose an algorithm for iterating such transformation in an appropriate ordering as to achieve the global effect of definitional trees. Therefore, there is no need of complicating the simple 'naive' calculus, it is the program instead which is transformed in such a way that, under the naive strategy, it behaves as if the efficient demand driven strategy would have been used.

We compare now our work with [19, 2, 3] in relation to the strategy. In [19] there are no formal justifications, such as soundness and completeness results. With respect to [2, 3], they refer to less general classes of rewriting systems (unconditional, inductively sequential [1], and for the case of [2], confluent). As a counterpart, they obtain some optimality results which are missing here. Anyway, we think that our 'program transformation approach' is a valuable alternative to their proofs of soundness and completeness of the strategy, based on a sophisticated redefinition of the narrowing rule. Finally, let us mention that our transformation presents some similarities with *unification factoring* [8] for logic programs. Obviously, our setting is very different; in particular, we must take care of functions and lazy evaluation.

4 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 write sometimes c/n, f/n for indicating that $c \in DC^n_\Sigma$, $f \in FS^n_\Sigma$, i.e., c, f have arity n. We also assume a countable set $\mathcal{V}$ of *variable* symbols. We write $Term_\Sigma$ for the set of (total) terms 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.

We will frequently work with the extended signature $\Sigma_\perp$ which is the result of incorporating the new constant (0-arity constructor) $\perp$ to Σ (this constant plays the role of the undefined value). Over this signature we can built up the sets $Term_\perp$ and $CTerm_\perp$ of *partial terms* and *partial c-terms* respectively. (Possibly partial) c-terms are meant to represent (possibly partial) data values, while terms represent possibly reducible expressions.

We will say that two partial c-terms t and t' are *inconsistent* if there exists a position in which t and t' have two constructor symbols c and d such that $c \not\equiv d$ and $c \not\equiv \perp \not\equiv d$.

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

The set of *partial c-substitutions* (substitutions, in short) is defined as $CSubst_\perp = \{\theta : \mathcal{V} \to CTerm_\perp\}$. We write $t\theta$ as the result of applying the substitution θ to the term t and $\mu\theta$ as the composition such that $t(\mu\theta) = (t\mu)\theta$. As usual $\theta = [X_1/t_1, ..., X_n/t_n]$ stands for the substitution that satisfies $X_i\theta = t_i$ for all $i = 1..n$ and $Y\theta \equiv Y$ for all $Y \in \mathcal{V} - \{X_1, ..., X_n\}$.

5 A Constructor-based Rewriting Logic with Disequality ($CRWL_{\neq}$)

The Constructor-based ReWriting Logic (CRWL) of [10, 11] defines a semantics embodying *angelic non-determinism* with *call-time choice* (see e.g. [25]) for *non-strict* functions. We now add disequality constraints to obtain a new logic $CRWL_{\neq}$. We have only addressed the 'proof-theoretic face' of the CRWL-framework, but to extend also the model-theoretic semantics of [10, 11] would not be difficult.

Programs are theories in $CRWL_{\neq}$. More precisely, assuming a given $\Sigma = DC \cup FS$, a *$CRWL_{\neq}$-program* is a set $\mathcal{R}$ of conditional rewriting rules of the form:

$$f(\overline{t}) = e \ <== \ C$$

where $f \in FS^n$, $\overline{t}$ is a linear tuple of c-terms and $C \equiv e_1 \diamond e'_1, ..., e_m \diamond e'_m$ with $e_i, e'_i \in Term$ and $\diamond \in \{==, \neq\}$. When C is empty we omit the symbol $<==$.

From a given $\mathcal{R}$ we want to derive statements of the following kinds: *approximation statements* of the form $e \to t$ ($e \in Term_\bot, t \in CTerm_\bot$), intended to mean that t is an approximation to a value resulting of reducing e; *equality (or joinability) statements* of the form $e == e'$ ($e, e' \in Term_\bot$), intended to mean that e, e' are both reducible to a common total c-term t; and *disequality (or divergence) statements* of the form $e \neq e'$ ($e, e' \in Term_\bot$), intended to mean that e, e' are reducible to inconsistent partial c-terms t, t'.

Formal $CRWL_{\neq}$-provability is governed by the rules of Table 1. They use the set of *partial c-instances* of $\mathcal{R}$ defined as $[\mathcal{R}]_\bot \equiv \{(l = r <== C)\theta \mid (l = r <== C) \in \mathcal{R}, \theta \in CSubst_\bot\}$. To consider *partial c-instances* is needed because of *call-time choice non-strict* semantics, whose intuitive meaning is: for reducing $f(e_1, \ldots, e_n)$, one first chooses some fixed (possibly partial) value for each argument e_i before applying a rule for f. The fixed value is then *shared* in all occurrences of e_i in the right hand side of the rule. We will write $\mathcal{R} \vdash \varphi$ for indicating that the statement φ is $CRWL_{\neq}$-provable using the program $\mathcal{R}$.

(1)	$\dfrac{}{e \to \bot}$	
(2)	$\dfrac{}{X \to X}$	$for\ X \in \mathcal{V}$
(3)	$\dfrac{e_1 \to t_1, ..., e_n \to t_n}{c(e_1, ..., e_n) \to c(t_1, ..., t_n)}$	$for\ c \in DC^n, \quad t_i \in CTerm_\bot$
(4)	$\dfrac{e_1 \to s_1, ..., e_n \to s_n \quad C \quad e \to t}{f(e_1, ..., e_n) \to t}$	$if\ t \not\equiv \bot,\ (f(s_1, ..., s_n) = e <== C) \in [R]_\bot$
(5)	$\dfrac{a \to t \quad b \to t}{a == b}$	$if\ t \in CTerm$
(6)	$\dfrac{a \to t \quad b \to t'}{a \neq b}$	$if\ t, t' \in CTerm_\bot\ are\ inconsistent$

Table 1: Rules for $CRWL_{\neq}$-provability

Example 1 *Let* $\Sigma = \{0/0, s/1, [\,]/0, :/2\} \cup \{f/1, coin/0\}$. *We will use '*$[\,]$*' and ':' as constructors for lists, but we will write lists in Prolog-like notation, that is, the expression* $[0, s(0)|L]$ *represents the list* $(0 : (s(0) : L))$. *Let* $\mathcal{R}$ *be a program over* Σ *with the following three rules:* $f(N) = [N|f(s(N))]$ *and* $coin = 0$; $coin = s(0)$. *Notice that, as a rewrite system, this program is not confluent (due to the rules for coin) nor terminating (due to the rule for* f*).*

As a first remark, notice that the semantics given to $==$ *and* $\neq$*, which is reflected in the rules (5) and (6) for* $CRWL_{\neq}$*, enables to prove both* $coin == 0$ *and* $coin \neq 0$. *This is not a nonsense, it simply reflects the fact that* $==$ *and* $\neq$ *are meant to operate over single values picked up from the set of possible values of both sides (*$\{0, 1\}$ *for the case of coin). If only deterministic functions are involved,* $==$ *and* $\neq$ *behave as usual.*

Notice also that call-time choice, reflected in rule (4) of the calculus, allows to prove $f(coin) \rightarrow [0, s(0) \mid \bot]$ *and also* $f(coin) \rightarrow [s(0), s(s(0)) \mid \bot]$, *but not* $f(coin) \rightarrow [0, s(s(0)) \mid \bot]$.

We next show a complete $CRWL_{\neq}$*-proof of a disequality,* $f(coin) \neq [0, 0]$, *that involves the non-deterministic function coin and a potentially infinite list produced by the function* f. *The proof should be read from the bottom. At each step, the applied* $CRWL_{\neq}$*-rule has been indicated.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{0 \rightarrow 0}(3)}{coin \rightarrow 0}(4)}{f(coin) \rightarrow [0, s(0)|\bot]}\boxed{(4)}}{f(coin) \neq [0,0]}(6) \quad \dfrac{\dfrac{\dfrac{\dfrac{}{0 \rightarrow 0}(3)}{s(0) \rightarrow s(0)}(3)}{[0|f(s(0))] \rightarrow [0, s(0)|\bot]}(4)}{f(s(0)) \rightarrow [s(0)|\bot]}(3) \quad \dfrac{\dfrac{\dfrac{}{0 \rightarrow 0}(3)}{s(0) \rightarrow s(0)}(3)}{[s(0)|f(s(0))] \rightarrow [s(0)|\bot]}(3) \quad \dfrac{}{f(s(0)) \rightarrow \bot}\boxed{(1)}}{[0,0] \rightarrow [0,0]}(3) \quad \dfrac{}{\ldots}$$

Notice that sharing is implicit in step $\boxed{(4)}$ *and coin is reduced only once when evaluating* $f(coin)$. *Step* $\boxed{(1)}$ *captures laziness by allowing to evaluate only a fragment of the list. The approximation statement* $[0, 0] \rightarrow [0, 0]$ *on the bottom right side can be proved by successive application of rule (3).*

6 A Lazy Narrowing Calculus with Disequalities ($LNC_{\neq}$)

In the spirit of logic languages, computing in our framework means *solving goals*, which in turn means obtaining values (in the form of constraints) for the variables in the goal, such that with these values substituted for the variables the goal becomes $CRWL_{\neq}$ provable. We first define goals and solutions.

Definition 1 (Goals) *A* goal *is an expression* $G \equiv \exists \overline{U}.P \square R \square \sigma \square \delta$ *where:*

- P *is a multiset of approximation statements (also called* productions*) of the form* $e \rightarrow t$ *with* $e \in Term, t \in CTerm$.
- R *is a multiset of constraints* $e \diamond e'$, *where* $e, e' \in Term, \diamond \in \{==, \neq\}$.

- σ *is a set of equalities of the form* $X = s$ *where* $s \in CTerm$ *and* X *does not occur elsewhere in* G.
- δ *is a multiset of disequalities of the form* $X \neq t$, *where* $t \in CTerm$.

The form for an **initial goal** *is* $G \equiv \Box R \Box\Box$ *and for a* **solved goal** *is* $G \equiv \exists \overline{U}.\Box\Box\sigma\Box\delta$.

Definition 2 (Solutions) *Let* $\mathcal{R}$ *be a program. We say that* $\theta \in CSubst_\bot$ *is* a solution *for a goal* $G \equiv \exists \overline{U}.P\Box R\Box\sigma\Box\delta$, *and write* $\theta \in Sol(G)$ *if:*

$$i)\, \mathcal{R} \vdash (P, R, \delta)\theta, \quad ii)\, \sigma\theta \text{ consists of identities}, \quad iii)\, X\theta \in CTerm\, \forall X \notin \overline{U}$$

Table 2. contains the rules for a calculus, $LNC_{\neq}$, specifying how to perform one step $G \rightsquigarrow G'$ of goal solving. Here, the symbols $==$ and $\neq$ are seen as symmetric. The main difference with the calculus presented in [10] is the addition of disequality constraints. For reasons of space, some failure rules have been omitted in Table 2.

Some distinguished sets of goal variables are used in the calculus: the set of *produced variables* of G is $pvar(G) = \{X \mid X \in var(t) \text{ for some } e \to t \in P\}$, and the set of *demanded variables* is $dvar(G) = \{X \mid (X \diamond e) \in R, \diamond \in \{==, \neq\}\}$. The set of *secure variables* of a term e, $svar(e)$, consists of all variables occurring in e at some position whose ancestor positions are all occupied by constructors.

Goals that are in fact reachable in $LNC_{\neq}$-computations starting from an initial goal satisfy a number of technical conditions, related to produced variables, which are needed for proving soundness and completeness of $LNC_{\neq}$. In a short summary, they are: produced variables appear as such only once, are existential, do not occur in solved parts σ and δ, and do not present a cycle of dependencies. We skip the technical details.

Some remarks about the rules on Table 2 follow. Two notations used in the calculus can be illustrated by means of the rule (**Obind**) (*output binding*), which solves approximations $X \to c(\overline{t})$ by applying the substitution $[X/c(\overline{t})]$. If X has not been introduced by some derivation step, but appeared in the initial goal, then the substitution must take part of the answer, which means that it must be added to σ in the form $X = c(\overline{t})$. Otherwise, X is an existential variable which can simply be deleted, and there is no need of recording its binding. For this distinction we use the following notation: if $G \equiv \exists \overline{U}.P\Box R\Box\sigma\Box\delta$ then

$$(G \uplus X = t) \equiv \begin{cases} \exists \overline{U}.P\Box R\Box X = t, \sigma\Box\delta & \text{if } X \notin \overline{U} \\ \exists (\overline{U} - \{X\}).P\Box R\Box\sigma\Box\delta & \text{if } X \in \overline{U} \end{cases}$$

The other notation is justified by the following fact: all the disequalities in δ have been introduced by **Store** in *solved form* $Y \neq s$, with $Y \in \mathcal{V}, s \in CTerm$. In particular, δ may contain disequalities like $X \neq s$ that miss their solved form when the substitution $[X/c(\overline{t})]$ is applied. The resulting disequalities will take the form $t \neq s$ and must be returned back to the constraints R for further processing. In order to extract them from δ we will introduce the notation δ_X to represent all disequalities associated to the variable X, $\delta_X = \{X \neq t \mid (X \neq t) \in \delta\}$.

These notations are also used in the rule **Bind** and in two rules of *term structure imitation*, $\mathbf{Imit}_{==}$ and $\mathbf{Imit}_{\neq}$. In other rules, like $\mathbf{Ibind}_{\rightarrow}$ (*input binding*) and

▶ Rules for $\rightarrow$

Decomp$_\rightarrow$ $\exists\overline{U}.c(\overline{e}) \rightarrow c(\overline{t}), P\Box R\Box\sigma\Box\delta \rightsquigarrow \exists\overline{U}.\overline{e} \rightarrow \overline{t}, P\Box R\Box\sigma\Box\delta$

Obind $\exists\overline{U}.X \rightarrow c(\overline{t}), P\Box R\Box\sigma\Box\delta, \delta_X \rightsquigarrow \exists\overline{U}.(P\Box R, \delta_X\Box\sigma\Box\delta)[X/c(\overline{t})] \uplus X = c(\overline{t})$

Ibind $\exists X, \overline{U}.t \rightarrow X, P\Box R\Box\sigma\Box\delta \rightsquigarrow \exists\overline{U}.(P\Box R\Box\sigma\Box\delta)[X/t]$ if $t \in CTerm$

Imit$_\rightarrow$ $\exists X, \overline{U}.c(\overline{e}) \rightarrow X, P\Box R\Box\sigma\Box\delta \rightsquigarrow \exists\overline{Y}, \overline{U}.(\overline{e} \rightarrow \overline{Y}, P\Box R\Box\sigma\Box\delta)[X/c(\overline{Y})]$
if $c(\overline{e}) \notin CTerm, X \in dvar(G), \overline{Y}$ new vars.

Elim $\exists X, \overline{U}.e \rightarrow X, P\Box R\Box\sigma\Box\delta \rightsquigarrow \exists\overline{U}.P\Box R\Box\sigma\Box\delta$ if $X \notin var(P \cup R)$

Narrow$_\rightarrow$ $\exists\overline{U}.f(\overline{e}) \rightarrow t, P\Box R\Box\sigma\Box\delta \rightsquigarrow \exists\overline{Y}, \overline{U}.\overline{e} \rightarrow \overline{s}, e \rightarrow t, P\Box C, R\Box\sigma\Box\delta$
if $(t \notin \mathcal{V} \vee t \in dvar(G))$,
$f(\overline{s}) = e <== C$ is a variant of a rule in $\mathcal{R}$ with new variables $\overline{Y}$

▶ Rules for $==, \neq$

Narrow$_\diamond$ $\exists\overline{U}.P\Box f(\overline{e})\diamond e', R\Box\sigma\Box\delta \rightsquigarrow \exists\overline{Y}, \overline{U}.\overline{e} \rightarrow \overline{s}, P\Box e\diamond e', C, R\Box\sigma\Box\delta$
if $f(\overline{s}) = e <== C$ is a variant of a rule in $\mathcal{R}$ with new variables $\overline{Y}$

▶ Rules for $==$

Decomp$_{==}$ $\exists\overline{U}.P\Box c(\overline{e}) == c(\overline{e}'), R\Box\sigma\Box\delta \rightsquigarrow \exists\overline{U}.P\Box\overline{e} == \overline{e}', R\Box\sigma\Box\delta$

Id $\exists\overline{U}.P\Box X == X, R\Box\sigma\Box\delta \rightsquigarrow \exists\overline{U}.P\Box R\Box\sigma\Box\delta$ if $X \notin pvar(G)$

Bind $\exists\overline{U}.P\Box X == t, R\Box\sigma\Box\delta, \delta_X \rightsquigarrow \exists\overline{U}.(P\Box\delta_X, R\Box\sigma\Box\delta)[X/t] \uplus X = t$
if $t \in CTerm, var(t) \cap pvar(G) = \emptyset, X \notin var(t) \cup pvar(G)$

Imit$_{==}$ $\exists\overline{U}.P\Box X == c(\overline{e}), R\Box\sigma\Box\delta, \delta_X \rightsquigarrow$
$\exists\overline{Y}, \overline{U}.(P\Box\delta_X, \overline{Y} == \overline{e}, R\Box\sigma\Box\delta)[X/c(\overline{Y})] \uplus X = c(\overline{Y})$
if $(c(\overline{e}) \notin CTerm \vee var(c(\overline{e})) \cap pvar(G) \neq \emptyset)$,
$X \notin pvar(G), X \notin svar(c(\overline{e})), \overline{Y}$ new vars.

▶ Rules for $\neq$

Clash $\exists\overline{U}.P\Box c(\overline{e}) \neq d(\overline{e}'), R\Box\sigma\Box\delta \rightsquigarrow \exists\overline{U}.P\Box R\Box\sigma\Box\delta$

Store $\exists\overline{U}.P\Box X \neq t, R\Box\sigma\Box\delta \rightsquigarrow \exists\overline{U}.P\Box R\Box\sigma\Box\delta, X \neq t$
if $X \notin pvar(G), t \in CTerm, X$ is not $t, pvar(G) \cap var(t) = \emptyset$

Decomp$_\neq$ $\exists\overline{U}.P\Box c(\overline{e}) \neq c(\overline{e}'), R\Box\sigma\Box\delta \rightsquigarrow \exists\overline{U}.P\Box e_i \neq e_i', R\Box\sigma\Box\delta \quad i \in \{1, ..., n\}$

Imit$_\neq$ $\exists\overline{U}.P\Box X \neq c(\overline{e}), R\Box\sigma\Box\delta, \delta_X \rightsquigarrow$ *(choose one of the following two)*
$\begin{cases} \exists\overline{U}, \overline{Y}.(P\Box R, \delta_X\Box\sigma\Box\delta)[X/d(\overline{Y})] \uplus X = d(\overline{Y}) & d \in DC, d \neq c \\ \exists\overline{U}, \overline{Y}.(P\Box Y_i \neq e_i, R, \delta_X\Box\sigma\Box\delta)[X/c(\overline{Y})] \uplus X = c(\overline{Y}) \end{cases}$
if $X \notin pvar(G), (c(\overline{e}) \notin CTerm \vee var(c(\overline{e})) \cap pvar(G) \neq \emptyset), \overline{Y}$ new vars.

Table 2: Rules for $LNC_\neq$

Imit$_{\rightarrow}$, there is no need of such notations, since in that cases the variable X is a produced one, and then it does not appear in δ and does not need to take part in the answer.

Notice also that solving disequalities might involve don't know choices. For the case $c(\overline{e}) \neq c(\overline{e}')$ the rule **Decomp**$_{\neq}$ chooses one argument, while for $X \neq c(\overline{e})$, **Imit**$_{\neq}$ guesses a possible substitution $[X/d(\overline{Y})]$. If d is not c, the disequality has been solved without further reduction of $c(\overline{e})$; otherwise, a decomposition is needed, which implies a new choice.

The rules **Narrow**$_{\diamond}$ and **Narrow**$_{\rightarrow}$ work when a functional expression must be reduced, for solving a constraint in the first case and for solving an approximation statement of the form $f(\overline{e}) \rightarrow t$ (where t is a non-variable c-term or a demanded variable X) in the second one. Both two try to apply a rule of the program by making parameter passing in the form of new approximation statements, and solving the constraints in the condition of the rule. The rule **Elim** cleans the goal by deleting unnecessary productions in P.

Example 2 *Consider $\mathcal{R}$ as in example 1. One solution for $f(coin) \neq [X|Xs]$ is given by $X\theta = s(Y)$ and one $LNC_{\neq}$-computation capturing θ is the following:*

$$\Box \underline{f(coin) \neq [X|Xs]} \Box\Box \overset{Narrow_{\diamond}}{\leadsto} \exists Y.coin \rightarrow Y \Box \underline{[Y|f(s(Y))] \neq [X|Xs]} \Box\Box \overset{Decomp_{\neq}}{\leadsto}$$
$$\exists Y.\underline{coin \rightarrow Y} \Box Y \neq X \Box\Box \overset{Narrow_{\rightarrow}}{\leadsto} \exists Y.\underline{0 \rightarrow Y} \Box Y \neq X \Box\Box \overset{Ibind}{\leadsto} \Box 0 \neq X \Box\Box \overset{Store}{\leadsto}$$
$$\Box\Box\Box X \neq 0$$

The last goal is a solved one that yields the answer $X \neq 0$.

Example 3 *The definition of member in Sect. 2 can be completed with definitions for the functions (written in infix and mixfix notations) _ eq _ and if _ then _ else _*

$$X\ eq\ Y = true <== X == Y \qquad if\ true\ then\ X\ else\ Y = X$$
$$X\ eq\ Y = false <== X \neq Y \qquad if\ false\ then\ X\ else\ Y = Y$$

It can be checked that the only two solved goals obtained by $LNC_{\neq}$ for the initial goal $G \equiv \Box\ member(X, [Y]) == T\ \Box\Box$ are $G' \equiv \Box\Box T = true, X = Y \Box$ and $G'' \equiv \Box\Box T = false\ \Box\ X \neq Y$.

The relationship between the logic $CRWL_{\neq}$ and the goal solving mechanism $LNC_{\neq}$ is established in the following results:

Theorem 1 (Correctness of $LNC_{\neq}$) *If G is an initial goal, $G \overset{*}{\leadsto} G'$ and $\theta' \in Sol(G')$, then there exists $\theta \in Sol(G)$ coinciding with θ' over $var(G)$.*

Theorem 2 (Completeness of $LNC_{\neq}$) *If G is an initial goal and $\theta \in Sol(G)$, then there exist a solved form G' and $\theta' \in Sol(G')$ coinciding with θ over $var(G)$ such that $G \overset{*}{\leadsto} G'$.*

The proofs of these results are non-trivial modifications of those in [11] and are technically involved. The intermediate results which are needed allow to characterize the kind of indeterminism that can appear when proceeding with a computation: if several $LNC_{\neq}$-rules are applicable to a given goal, the choice among them is *don't care* (this is sometimes called *strong completeness* [23]); when the don't care selected rule implies a choice (this is the case of **Narrow**$_{\diamond}$, **Narrow**$_{\rightarrow}$, **Decomp**$_{\neq}$ and **Imit**$_{\neq}$), it is *don't know*, thus determining a search space.

7 Demand Driven Strategy

Our calculus $LNC_{\neq}$, if directly adopted as operational model, implies a 'naive' and inefficient way, with respect to backtracking, of evaluating functions. For example, consider the following definitions for functions $leq/2$ (less or equal for natural numbers) and $add/2$:

$$\begin{array}{lll} (\mathsf{leq}_1) & leq(0,Y) & = true \\ (\mathsf{leq}_2) & leq(s(X),0) & = false \\ (\mathsf{leq}_3) & leq(s(X),s(Y)) & = leq(X,Y) \end{array} \qquad \begin{array}{ll} add(0,Y) & = Y \\ add(s(X),Y) & = s(add(X,Y)) \end{array}$$

If we want to solve the goal $leq(add(0,s(0)),add(s(0),s(0))) == B$, trying the rules of functions in textual order and with chronological backtracking in case of failure, the computation would have the following structure:

$$\xrightarrow{(\mathsf{leq}_1)} \begin{cases} \underline{add(0,s(0))} \to 0 \ ... \ \boxed{\text{Fail}} \\ add(s(0),s(0)) \to Y \end{cases} \qquad \xrightarrow{(\mathsf{leq}_2)} \begin{cases} \underline{add(0,s(0))} \to s(X) \\ \underline{add(s(0),s(0))} \to 0 \ ... \ \boxed{\text{Fail}} \end{cases}$$

$$\xrightarrow{(\mathsf{leq}_3)} \begin{cases} \underline{add(0,s(0))} \to s(X) \\ \underline{add(s(0),s(0))} \to s(Y) \\ ... \end{cases}$$

By using (leq_1) our calculus will try to solve $add(0,s(0)) \to 0$. It will perform some steps for evaluating $add(0,s(0))$ until failure is detected. Then, it will try (leq_2) and $add(0,s(0))$ will be evaluated again from the beginning because previous computation steps have been lost. A new failure is detected and then it will try the last rule (leq_3) for leq. The expression $add(0,s(0)) \to 0$ has been evaluated three times.

One can proceed in a different way: notice that all of the rules of leq have a constructor as first argument, what means that the first parameter of any call to leq must be reduced to one of these constructors. So, there is no forfeit if we perform some reduction steps on the first argument $add(0,s(0))$ to obtain a *head normal form* $s(\ldots)$ before applying any rule of leq. Doing so, the first rule for leq will fail, but we can reuse the head normal form computed before for trying the next rules of leq. This is the idea in which the *Demand Driven Strategy* [19] is based.

Now, how can we suit this idea into our formal framework? We have the problem that, although there is freedom for the ordering in which the $LNC_{\neq}$ rules are applied, yet there is no way of evaluating an argument of a function call before a program rule for the call has been selected. In a real implementation we may use control mechanisms for evaluating head normal forms. Here we do not have control, but we have the expressive power of disequalities instead and they will suffice. Notice that a disequality $X \neq e$, where e is in head normal form, can be solved in $LNC_{\neq}$ (either by **Store** or by $\mathbf{Imit}_{\neq}$), without needing to perform any further reduction in e. An example may help to understand this: if we consider the disequality $X \neq add(s(0),0)$, we could have the following $LNC_{\neq}$-derivation:

$$\begin{array}{l} \Box X \neq add(s(0),0)\Box\Box \overset{Narrow_{\diamond}}{\rightsquigarrow} (\textit{with the second rule of } add) \\ \exists Y_1,Y_2.s(0) \to Y_1, 0 \to Y_2 \Box X \neq s(add(Y_1,Y_2))\Box\Box \overset{Decomp_{\to},Ibind,Ibind}{\rightsquigarrow} \\ \Box X \neq s(add(0,0))\Box\Box \overset{Imit_{\neq}}{\rightsquigarrow} \Box\Box X = 0\Box \end{array}$$

This suggests the following idea: to get a head normal form for an expression we can solve a disequality between this expression and a new variable; our calculus will reduce the expression only until it can solve the disequality, i.e., until reaching a head normal form.

The idea of using disequalities to force the computation of *hnf* for arguments of a function call can be achieved by means of an incremental program transformation. For the case of *leq* this would be:

$$\begin{array}{l} leq(\boxed{0},Y) = true \\ leq(\boxed{s}(X),0) = false \\ leq(\boxed{s}(X),s(Y)) = leq(X,Y) \end{array} \overset{(1)}{\leadsto} \begin{array}{l} leq(A,B) = leq_1(A,B) <== U \neq A \\ leq_1(0,Y) = true \\ leq_1(s(X),\boxed{0}) = false \\ leq_1(s(X),\boxed{s}(Y)) = leq(X,Y) \end{array} \overset{(2)}{\leadsto}$$

$$\begin{array}{l} leq(A,B) = leq_1(A,B) <== U \neq A \\ leq_1(0,B) = true \\ leq_1(s(A),B) = leq_{1_2}(s(A),B) <== V \neq B \\ leq_{1_2}(s(A),0) = false \\ leq_{1_2}(s(A),s(B)) = leq(A,B) \end{array}$$

In step (1) we handle the first argument of *leq* and in (2) a similar transformation is done for the second argument of the last two rules of leq_1. Now there is only one rule for *leq* with a condition $U \neq A$ (U is a new variable) in its condition whose effect is to compute a head normal form for the first argument and then call to leq_1, which distinguishes if this head normal form is 0 or $s(\_)$. In the first case the value *true* is returned and in the second one a head normal form for the second argument is evaluated through the solving of the disequality $V \neq B$. If it is 0 then $false$ is returned; otherwise *leq* is invoked again. We show below how to use $LNC_{\neq}$ to solve the goal $leq(add(0,s(0)),add(s(0),s(0))) == X$, using the transformed program. At each $LNC_{\neq}$-step the (don't care) selected condition for applying a $LNC_{\neq}$-rule has been underlined.

$$\begin{array}{l} \Box\underline{leq(add(0,s(0)),add(s(0),s(0))) == X}\Box\Box \overset{Narrow_{\diamond}}{\leadsto} \\ \exists Y_1,Y_2,U.\underline{add(0,s(0)) \to Y_1}, add(s(0),s(0)) \to Y_2\Box leq_1(Y_1,Y_2) == X, U \neq Y_1\Box\Box \overset{Narrow_{\to}}{\leadsto} \\ \exists Y_1,Y_2,U,Z.\underline{0 \to 0},\underline{s(0) \to Z},\underline{Z \to Y_1}, add(s(0),s(0)) \to Y_2\Box \\ \qquad\qquad leq_1(Y_1,Y_2) == X, U \neq Y_1\Box\Box \overset{Decomp_{\to},Ibind,IBind}{\leadsto} \\ \exists Y_2,U.add(s(0),s(0)) \to Y_2\Box leq_1(s(0),Y_2) == X, U \neq s(0)\Box\Box \overset{Store}{\leadsto} \\ \exists Y_2,U.add(s(0),s(0)) \to Y_2\Box leq_1(s(0),Y_2) == X\Box\Box U \neq s(0) \overset{Narrow_{\diamond}}{\leadsto} \\ ... \end{array}$$

Now, before applying any rule of leq_1 we have got the head normal form $s(0)$ for the first argument $add(0,s(0))$. Then the first rule of leq_1 will fail in a few steps, but the computations steps for $add(0,s(0))$ are not lost when trying the other rule for leq_1.

We have reached our purpose, but also introduced an undesirable effect: we have to solve disequalities that do not appear using the initial program. They can suppose an overloading of the constraint store δ and, in fact, we are not really interested in

solving them, they are only used to get head normal forms. Fortunately, it is not difficult to solve this problem; we can add a new rule to our calculus $LNC_{\neq}$:

Elim$_{\neq}$ $\exists X, \overline{U}.P \Box X \neq e, R \Box \sigma \Box \delta \ \leadsto \ \exists \overline{U}.P \Box R \Box \sigma \Box \delta$
if X does not appear elsewhere in G and ($e \equiv c(\overline{e})$ or $e \in \mathcal{V} - pvar(G)$).

This rule gets rid of this kind of disequalities once we have got the desired head normal form. It is easy to see that this rule cannot destroy the completeness of $LNC_{\neq}$ and preserves also its correctness if there are at least two constructor symbols in the signature (since in this case, independently of the form of e in **Elim**$_{\neq}$, there is a possible value of X satisfying $X \neq e$). We remark that, although the rule **Elim**$_{\neq}$ overlaps with **Imit**$_{\neq}$ (and the same for **Store**), to use one or the other for a given $X \neq e$ is a don't care choice. The use of **Elim**$_{\neq}$ can be then restricted to those artificial disequalities introduced by our program transformation; in such cases the test 'X *does not appear elsewhere in* G' is unnecessary since its success is guaranteed.

We now provide a formal definition of the transformations and prove that they preserve the semantics.

7.1 Program Transformation

We need in this section some usual terminologies about positions in terms: positions are sequences of positive integers $p_1 \cdot ... \cdot p_m$. A position u in a term e identifies both the subterm of e at u and the symbol of e at u. We write $VP(e)$ for the set of positions in e occupied by variables. We will also need the subsumption ordering $\leq$ over $CTerm_{\perp}$: $s \leq t$ iff $\exists \theta \in CSubst_{\perp}$ such that $s\theta = t$

Given a program $\mathcal{R}$ and a function symbol f, we write $\mathcal{R}_f$ for the set of defining rules for f. A position u is *demanded by a rule* $f(t_1, ..., t_n) = e <== C$ if the left hand side $f(t_1, ..., t_n)$ has a constructor symbol at position u. A position u is *demanded by a set of rules* $\mathcal{S}$ if u is demanded by some rule of $\mathcal{S}$; if u is demanded by all the rules of $\mathcal{S}$, we say that u is *uniformly demanded* by $\mathcal{S}$.

Definition 3 (Transformation of Sets of Rules) *Let $\mathcal{R}$ be a program over Σ, f a function with rules $\mathcal{R}_f$, and*

- $\mathcal{S} = \{(f(\overline{t}_1) = e_1 <== C_1), ..., (f(\overline{t}_n) = e_n <== C_n)\}$ *a subset of* $\mathcal{R}_f$,
- $f(\overline{s})$ *a* **pattern compatible with** $\mathcal{S}$, *i.e., $\overline{s}$ is a linear tuple of c-terms and $f(\overline{s}) \leq f(\overline{t}_i)$, for all $i = 1..n$,*
- $u \in VP(f(\overline{s}))$ *a position uniformly demanded by $\mathcal{S}$. Let X the variable in $f(\overline{s})$ at position u.*

The transformed set of $\mathcal{S}$ using $f(\overline{s})$ and u *is* $\mathcal{T}(\mathcal{S}, f(\overline{s}), u) =_{def} \{R\} \cup S_u$, *where*

$R \equiv f(\overline{s}) = f_u(\overline{s}) <== U \neq X$
where f_u is a new function symbol and U is a new variable.

$S_u \equiv \{f_u(\overline{t}_1) = e_1 <== C_1, ..., f_u(\overline{t}_n) = e_n <== C_n\}$

Now we want to prove that the semantics of a program is preserved under transformations, that is: given a program $\mathcal{R}$ and a program $\mathcal{R}'$ which is the result of transforming some function definition in $\mathcal{R}$ then the same statements are provable using $\mathcal{R}$ or $\mathcal{R}'$. Of course, as in $\mathcal{R}'$ there is a new function symbol, statements that use this symbol make no sense for $\mathcal{R}$, but they have no interest for us.

To prove this result we use an additional hypothesis: the signature must contain at least two constructor symbols. This condition guarantees that the disequalities introduced by the transformation can be proved by rule (6) of $CRWL_{\neq}$, taking an appropriate c-instance of the rule R produced by the transformation[1]. Nevertheless, in practice this condition may be dropped because these disequalities are not really solved, but are eliminated by the new rule $\mathbf{Elim}_{\neq}$ that we have incorporated into our calculus.

Lemma 1 (Equivalence under Transformations) *Let Σ be a signature with at least two constructor symbols, $\mathcal{R}$ a program over Σ, f a function of Σ, $\mathcal{S} \subseteq \mathcal{R}_f$, $f(\overline{s})$ a pattern compatible with $\mathcal{S}$, and u a position in it. Assume $\mathcal{R}' = (\mathcal{R} - \mathcal{S}) \cup \mathcal{T}(\mathcal{S}, f(\overline{s}), u)$ and φ a condition of the form $e \to t, e == e'$ or $e \neq e'$ involving only symbols of $\Sigma_\bot$. Then $\mathcal{R} \vdash \varphi \Leftrightarrow \mathcal{R}' \vdash \varphi$*

Proof sketch.

$\Rightarrow$) We proceed by induction over the length l of the proof for $\mathcal{R} \vdash \varphi$ (number of steps of the $CRWL_{\neq}$-derivation). The base case $l = 0$ is immediate. For the case $l + 1$ things are also easy except when φ takes the form $f(\overline{e}) \to t$, with $t \not\equiv \bot$. In this case, there exists a c-instance $Q \equiv (f(\overline{t}) = e <== C) \in [\mathcal{R}]_\bot$, so that the proof $\mathcal{R} \vdash f(\overline{e}) \to t$ is reducible by rule (4) of $CRWL_{\neq}$ to $\mathcal{R} \vdash \overline{e} \to \overline{t}, C, e \to t$. With respect to $\mathcal{R}'$ we can take a c-instance $R\mu$ of the rule $R \equiv f(\overline{s}) = f_u(\overline{s}) <== U \neq X$ introduced by the transformation in order to reduce the proof for φ, also by rule (4), to $\mathcal{R} \vdash \overline{e} \to \overline{s}\mu, (U \neq X)\mu, f_u(\overline{s}\mu) \to t$. Choosing the c-substitution μ appropriately we can get: $(\overline{e} \to \overline{s}\mu) \equiv (\overline{e} \to \overline{t})$ which is provable with respect to $\mathcal{R}'$ by i.h.; using the fact that Σ contains at least two constructor symbols it is easy to make $X\mu$ and $U\mu$ inconsistent; and for the approximation $f_u(\overline{s}\mu) \to t$ we can use the same c-instance Q that appears also in $\mathcal{R}'$, but with name f_u and then apply i.h.

$\Leftarrow$) We proceed also by induction over the length l of the proof for $\mathcal{R}' \vdash \varphi$. Now, the critical case is when $\varphi \equiv f(\overline{e}) \to t$ and its proof with respect $\mathcal{R}'$ is reducible by rule (4) of $CRWL_{\neq}$, using a c-instance of the rule $R\mu$ introduced by the transformation. The result of this step will be $\mathcal{R}' \vdash \overline{e} \to \overline{s}\mu, (U \neq X)\mu, f_u(\overline{s}\mu) \to t$. The last statement must be reducible also by rule (4) making use of a c-instance $Q' \equiv (f_u(\overline{t}) = e <== C) \in [\mathcal{R}']_\bot$ to $\mathcal{R}' \vdash \overline{s}\mu \to \overline{t}, C, e \to t$. It can be proved that the relation $\to$ is transitive, therefore, from $\mathcal{R}' \vdash \overline{e} \to \overline{s}\mu, \overline{s}\mu \to \overline{t}$ we can conclude $\mathcal{R}' \vdash \overline{e} \to \overline{t}$. With respect to $\mathcal{R}$ we can use a c-instance Q identical to Q' except for the function name, that will be f, in order to reduce the proof $\mathcal{R} \vdash f(\overline{e}) \to t$ to $\mathcal{R} \vdash \overline{e} \to \overline{t}, C, e \to t$, which are all provable by i.h. and then whe have $\mathcal{R} \vdash f(\overline{e}) \to t$. ■

[1]There is another possibility: the transformation can also introduce a new constructor; in this case the rules of $LNC_{\neq}$ should prevent the appearance of such fictitious constructor in σ and δ in solved forms.

7.2 Transformation Algorithm

By iterating in a convenient way the transformation of the previous section, we obtain an algorithm for transforming function definitions which works in a similar form to the process of translation of functional logic programs into Prolog programs described in [19] as a specification of the *Demand Driven Strategy*. The essential difference is that in our case the result of the transformation lies inside the same formal $CRWL_{\neq}$-framework, thus making more feasible to obtain correctness results like Lemma 1 or Theorem 3.

The transformation is performed by the function *trans* defined in Table 3 that takes a set of rules $\mathcal{S}$ and a pattern compatible with $\mathcal{S}$, $f(\overline{s})$, and returns the **transformed set of rules of** $\mathcal{S}$. For transforming full programs we simply need to apply this algorithm to each function f of the program whit the **initial call:** $trans(\mathcal{R}_f, f(\overline{X}))$. The recursive calls will have the form $trans(\mathcal{S}, f(\overline{s}))$, where $\mathcal{S}$ is a set of function rules and $f(\overline{s})$ a pattern.

If $\mathcal{S}$ is empty or a singleton then **return** $\mathcal{S}$, otherwise, apply one of the following (mutually incompatible) alternatives:

a) Some position u in $VP(f(\overline{s}))$ is uniformly demanded by $\mathcal{S}$ (if there are several, choose any).
Let $\mathcal{T}(\mathcal{S}, f(\overline{s}), u) = \{R\} \cup S_u$ and let $c_1, ..., c_n$ be the constructors at position u in left hand sides of rules of $\mathcal{S}$. Consider the following partition over $\mathcal{S}_u$:
Let $\mathcal{S}_u^1$ be the subset of $\mathcal{S}_u$ demanding c_1 at position u.
...
Let $\mathcal{S}_u^n$ be the subset of $\mathcal{S}_u$ demanding c_n at position u.

Let X be the variable at position u in $f(\overline{s})$. For each constructor c_i build the pattern $p_i = f_u(\overline{s})[X/c_i(Y_1, ..., Y_m)]$ where $\overline{Y}$ are new variables.

Return $\{R\} \cup trans(\mathcal{S}_u^1, p_1) \cup ... \cup trans(\mathcal{S}_u^n, p_n)$

b) Some position in $VP(f(\overline{s}))$ is demanded by $\mathcal{S}$, but none is uniformly demanded.
Let $u_1, ..., u_k$ be the demanded positions (ordered by any criterion). Consider the following partition over $\mathcal{S}$:
Let $\mathcal{S}_{u_1}$ be the subset of $\mathcal{S}$ demanding position u_1, $Q_1 = \mathcal{S} - \mathcal{S}_{u_1}$.
Let $\mathcal{S}_{u_2}$ be the subset of Q_1 demanding position u_2, $Q_2 = Q_1 - \mathcal{S}_{u_2}$.
...
Let S_{u_k} be the subset of Q_{k-1} demanding position u_k.
And let S_0 be the set of rules of $\mathcal{S}$ that do not demand any position.

Return $\mathcal{S}_0 \cup trans(\mathcal{S}_{u_1}, f(\overline{s})) \cup ... \cup trans(\mathcal{S}_{u_n}, f(\overline{s}))$

c) No position in $VP(f(\overline{s}))$ is demanded by $\mathcal{S}$, then **return** $\mathcal{S}$

Table 3: Function *trans*

Theorem 3 (Correctness of the Algorithm) *Let Σ be a signature with at least two constructor symbols, $\mathcal{R}$ a program over Σ, f a function of $\mathcal{R}$. Then:*

i) The transformation algorithm terminates, for any initial call $trans(\mathcal{R}_f, f(\overline{X}))$.

ii) The program $\mathcal{R}' = (\mathcal{R} - \mathcal{R}_f) \cup trans(\mathcal{R}_f, f(\overline{X}))$ is equivalent to $\mathcal{R}$ in the sense of Lemma 1.

Proof sketch. We first remark that all calls to $trans(\mathcal{U}, f(\overline{u}))$ produced by the algorithm satisfy the invariant condition: $f(\overline{u}) \leq f(\overline{t})$ for all rules $(f(\overline{t}) = t <== C) \in \mathcal{U}$. Therefore in the alternative **a)** the transformed set $\mathcal{T}(\mathcal{S}, f(\overline{s}), u)$ is always calculable according to Definition 3. For termination, note that when a call $trans(\mathcal{U}, f(\overline{u}))$ generates another call $trans(\mathcal{U}', f(\overline{u}'))$ then one of the following conditions yields:

- the cardinal of $\mathcal{U}$ is greater than the cardinal of $\mathcal{U}'$ (if first **a)** applies), or
- the number of constructor positions in $f(\overline{u}')$ is greater than the number of constructors positions in $f(\overline{u})$, and this number is bounded by the maximum number of constructor positions in the rules of $\mathcal{R}_f$ (because of the invariant).

With these two ideas it is not difficult to define the *complexity of a call to trans* and to prove that it decreases in each call. Part ii) can be proved using Lemma 1.■

8 Conclusions

We have improved CRWL [10, 11], a previously known framework for functional logic programming, in two directions: first, we have enhanced the expressive power of framework by allowing disequality constraints in programs and answers. Second, we have defined an incremental program transformation under which programs behave as is an efficient demand driven strategy in the sense of [19] would have been followed. Disequalities play a very simple, but important, role in the transformation; the main interest lies in the fact that the transformed programs are still in the same formal framework, thus allowing us to easily obtain correctness results for the transformation.

All these improvements are formal counterparts of different aspects of the functional logic system $\mathcal{TOY}$ [21], which is a concrete implementation of the CRWL framework enriched with HO features, types, demand driven strategy, disequality constraints and real arithmetic constraints. Therefore, $\mathcal{TOY}$ has now more solid formal foundations. Nevertheless, there are still many non trivial and interesting optimizations related to equality and disequality which are done in $\mathcal{TOY}$ but are not covered by this work. Their formal justification is subject of future work.

Acknowledgements We thank Juan C. González and Mario Rodríguez for many valuable discussions, and Eva Ullán for her comments.

References

[1] S. Antoy: *Definitional Trees*, Proc. ALP'92, Springer LNCS 632, 1992, pp. 143-157.

[2] S. Antoy, R. Echahed, and M. Hanus: *A Needed Narrowing Strategy*, Proc. POPL'94, ACM Press 1994, 268-279.

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

[4] P. Arenas-Sánchez, A. Gil-Luezas, F. J. López Fraguas: *Combining Lazy Narrowing with Disequality Constraints*, Proc. PLILP'94, Springer LNCS 844, 1994, 385-399.

[5] R. Caballero-Roldán, F.J. López-Fraguas: *Parsing with Non-Deterministic Functions*, Proc. APPIA-GULP-PRODE'98, Pisa 1998, 1-16.

[6] R. Caballero-Roldán, F.J. López-Fraguas: *A Functional-Logic Alternative to Monads*, Proc. Workshop on Component-Based Software Development in Computer Logic, 87-100, 1998.

[7] H. Comon: *Disunification: A Survey*, in : J.L.Lassez,G. Plotkin (eds.) *Computational Logic, Essays in Honor of Alan Robinson*, MIT Press, 1991, 322-359.

[8] S. Dawson, C.R. Ramakrishnan, S. Skiena, T. Swift: *Principles and Practice of Unification Factoring*, *ACM TOPLAS*, 18(5), 1996, 528-563.

[9] E. Giovannetti, G. Levi, C. Moiso, C. Palamidessi: *Kernel LEAF: A Logic plus Functional Language*, Journal of Computer and System Sciences, Vol. 42, No. 2, Academic Press 1991, 139-185.

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

[11] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, M. Rodríguez-Artalejo: *An Approach to Declarative Programming Based on a Rewriting Logic* (to appear in J. of Logic Programming).

[12] M. Hanus: *The Integration of Functions into Logic Programming: A Survey*, J. of Logic Programming 19&20, 1994, 583-628.

[13] M. Hanus: *Efficient Translation of Lazy Functional Logic Programs into Prolog*, Proc. LOPSTR'95, Springer LNCS 1048, 1995, 252-266

[14] M. Hanus: *A Unified Computation Model for Functional and Logic Programming*, Proc. POPL'97 ACM Press 1997, 80-93.

[15] M. Hanus (ed.): *Curry: An integrated functional logic language*. Available at `http://www-i2.informatik.rwth-aachen.de/ hanus/curry`, Jan. 1999.

[16] H. Hussmann: *Non-determinisms in Algebraic Specifications and Algebraic Programs*, Birkhäuser, 1993.

[17] J.P. Jouannaud, C. Kirchner: *Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification*, in J.L.Lassez,G. Plotkin (eds.) *Computational Logic, Essays in Honor of Alan Robinson*, The MIT Press, 1991, 357-321.

[18] H. Kuchen, F.J. López-Fraguas, J.J. Moreno-Navarro, M. Rodríguez-Artalejo: *Implementing Disequality in a Functional Logic Language*, Proc. JICSLP'92, The MIT Press 1992, 207-221.

[19] R. Loogen, F. J. López Fraguas, M. Rodríguez Artalejo: *A Demand Driven Computation Strategy for Lazy Narrowing*, Proc. PLILP'93, Springer LNCS 714, 1993, 184-200.

[20] F.J. López-Fraguas: *A General Scheme for Constraint Functional Logic Programming*, Proc. ALP'92, Springer LNCS 632, 1992, 213-217.

[21] F.J. López-Fraguas and J. Sánchez-Hernández: *TOY: A Multiparadigm Declarative System*, to appear in Proc. RTA'99, Springer LNCS. System available at `http://mozart.sip.ucm.es/toy`

[22] F.J. López-Fraguas, J. Sánchez-Hernandez: *Disequalities May Help to Narrow*, draft version avaliable at `ftp://147.96.25.167/pub/csl99draft.ps`

[23] A. Middeldorp, S. Okui and T. Ida: *Lazy Narrowing: Strong Completeness and Eager Variable Elimination*, Theoretical Computer Science 167, 1996, 95-130.

[24] J.J. Moreno-Navarro, M. Rodríguez-Artalejo: *Logic Programming with Functions and Predicates: The Language BABEL*, J. of Logic Programming 12, 1992, 189-223.

[25] H. Søndergaard and P. Sestoft: *Non-determinism in Functional Languages*, The Computer Journal 35(5), 1992, 514-523.