Combining Lazy Narrowing with Disequality Constraints*

Puri Arenas-Sánchez, Ana Gil-Luezas, Francisco J. López-Fraguas

Universidad Complutense de Madrid, Departamento de Informática y Automática
Facultad de C.C. Matemáticas, Av. Complutense s/n, 28040 Madrid, Spain
email: {puri,anagil,fraguas}@dia.ucm.es

Abstract. We investigate an extension of a lazy functional logic language, which uses term disequations both in programs and in computed answers. The semantic properties of the language are derived from the fact that it can be viewed as an instance of the $CFLP(X)$-scheme proposed in [Lo92] for constraint functional logic programming. In particular, the operational semantics for $CFLP(X)$ – so called lazy constrained narrowing – is a computation mechanism parameterized by a constraint solver over the underlying domain. We define a constraint solver for our language, whose properties ensure completeness. As a further step towards implementation, we present an executable Prolog specification of the operational semantics, which has been implemented on top of a Prolog system. Experimental results have shown good performance.

1 Introduction

Many approaches to the integration of functional and logic programming (FLP) (see [Ha94] for a survey) use conditional term rewriting systems as programs, and narrowing as goal solving mechanism. In particular, so called *(lazy) functional logic languages* use *(lazy) narrowing* [Re85] as operational semantics. Narrowing is able to solve goals by producing answer substitutions – which can be seen as sets of equations $X = t$ – , thus having the drawback of obtaining an infinite number of solutions in some cases where the use of some negative information could reduce the set of solutions to a finite size. For instance, the disequation $X \neq Y$ cannot be replaced by any equivalent and finite set of equations. Hence, to incorporate disequality constraints to a narrowing-based language substantially increases the expressivity of programs and answers.

In [KLMR92] a lazy functional logic language enhanced with disequality constraints is presented, and an implementation is proposed by means of an abstract machine, based on an operational semantics which combines lazy narrowing and disequality constraint solving. There is no claim of completeness, because no declarative semantics is given for the language.

* This research has been partially supported by the Spanish National Project TIC92–0793 "PDR" and the Esprit BRA Working Group Nr. 6028 "CCL".

The idea of complementing unification – regarded as equation solving over some tree domain– with disequations as constraints appears from the very beginning [Co82, Co84] in the works about PROLOG II, which is now accepted as the first *constraint logic programming* (CLP) language. After that, Jaffar and Lassez proposed [JL86, JL87] the CLP(X) scheme as a framework for constraint logic programming (for a recent survey on CLP see [JM94]). CLP languages using symbolic equations and disequations may profit from deep theoretical work on unification and disunification (see [JK92, Co92] for surveys).

In [Lo92] a scheme – CFLP(X) – for constraint functional logic languages is proposed, playing a similar role with respect to FLP to that of the scheme CLP(X) with respect to LP. It is shown in this work how, in particular, LP,FLP and any instance of CLP(X), can be realized as instances of CFLP(X).

In this paper we present the language $SFL_{\neq}$ (Simple Functional Logic language with disequations), which is essentially equivalent to that in [KLMR92]. Programs are conditional rewrite rules with equations and disequations as conditions. We show how to realize the language as an instance of the CFLP(X) scheme, thus inheriting its semantic properties. We present also some improvements of the scheme, allowing us to obtain for our language a clear and proved complete operational semantics. Proofs of the new results about CFLP(X), and of their applicability to $SFL_{\neq}$, can be found in [Lo94].

In a further step towards implementation, we give a Prolog-like specification of the operational semantics, close to that presented in [LLR93] for specifying lazy narrowing strategies. The specification converts into a real executable Prolog implementation by a careful design of the representation of expressions and constraints.

The organization of the paper is as follows: in Sec. 2 we describe the language $SFL_{\neq}$. Sec. 3 is devoted to summarize the CFLP(X) scheme and to overview the improvements with respect to the presentation in [Lo92]. In Sec. 4 we realize $SFL_{\neq}$ as an instance of the scheme, and propose a constraint solver to be combined with lazy narrowing. Sec. 5 and 6 cover the specification and implementation issues. In Sec. 7 we further discuss related work.

2 The language $SFL_{\neq}$

We assume a first order signature $\langle CS, \Delta \rangle$ with the ranked alphabet $CS = \bigcup_{n \in \mathbb{N}} CS^n$ of *constructor symbols* and the disjoint ranked alphabet $\Delta = \bigcup_{n \in \mathbb{N}} \Delta^n$ of *defined function symbols*. Given a countably infinite set of *variables* $X, Y, Z \ldots \in Var$, we build *terms* $s, t \ldots \in Term$ (using only variables and constructors) and *expressions* $e, l, r \ldots \in Exp$ (using variables, constructors and function symbols). For function symbols $f \in \Delta^n$ we consider *defining rules*, which must be *left linear* conditional equations of the form

$$f(t_1, \ldots, t_n) = e \Leftarrow l_1 == r_1, \ldots, l_m == r_m, l'_1 =/= r'_1, \ldots, l'_s =/= r'_s$$

where $t_i \in Term$, $e, l_i, r_i, l'_i, r'_i \in Exp$. Operationally, such equations will be used as *conditional rewrite rules*. The sign '==' in conditions stands for *strict equal-*

ity, meaning that a condition $l_i == r_i$ must be satisfied by narrowing l_i, r_i into unifiable terms. The sign '$=/=$' in conditions stands for *disequality*, meaning that a condition $l_i =/= r_i$ must be satisfied by narrowing l_i, r_i to a sufficient extent for detecting a disagreement of constructor symbols at the same position. An $SFL_{\neq}$*-program* is any finite set of defining rules obeying certain natural conditions which ensure the functionality of definitions; see [GHR92]. *Goals* for $SFL_{\neq}$-programs are systems of strict equations and disequations of the form

$$\Leftarrow l_1 == r_1, \ldots, l_m == r_m, l'_1 =/= r'_1, \ldots, l'_s =/= r'_s$$

to be solved by a combination of narrowing and constraint solving.

As an example, let $CS^0 = \{true, false, 0, [\,]\}$, $CS^1 = \{s\}$, $CS^2 = \{[.|.]\}$ and $\Delta^1 = \{size\}$, $\Delta^2 = \{member\}$.
A legal SFL-program is given by the following defining rules:

$$\begin{array}{lll}
member(X, [\,]) & = false. & (mb_1) \\
member(X, [Y \mid Ys]) & = true \Leftarrow X == Y. & (mb_2) \\
member(X, [Y \mid Ys]) & = member(X, Ys) \Leftarrow X =/= Y. & (mb_2) \\
size([\,]) & = 0. & (sz_1) \\
size([X \mid Xs]) & = size(Xs) \Leftarrow member(X, Xs) == true. & (sz_2) \\
size([X \mid Xs]) & = s(size(Xs)) \Leftarrow member(X, Xs) == false. & (sz_3)
\end{array}$$

A simple goal for this example is $\Leftarrow size([X, Y]) == N$, for which we expect $X == Y, N == s(0)$ and $X =/= Y, N == s(s(0))$ as computed answers.

3 The $CFLP(X)$ Scheme: A Summary

We assume a given *base signature* $\Sigma = F \cup \Pi$, where F and Π are sets of symbols (with fixed arities) for *primitive functions* and *primitive predicates* respectively. The signature Σ, together with a numerable set of variables V, determine as usual the syntactic domains $\mathcal{T}_F(V), \mathcal{B}_{\Pi,F}(V)$ of terms (or *primitive expressions*) and atoms (or *primitive atomic constraints*). Primitive constraints $\varphi \in \mathcal{CON}_\Sigma$ are conjunctions of primitive atomic constraints, possibly with some of their variables existentially quantified.

A Σ*-constraint structure* $\mathcal{R}$ is a triple $\langle (D, \sqsubseteq_D), \{f^R\}_{f \in F}, \{p^R\}_{p \in \Pi} \rangle$, where $(D, \sqsubseteq_D)$ is a Scott domain, and for each $f \in F^n$ ($p \in \Pi^n$ resp.), $f^R : D^n \longrightarrow D$ ($p^R : D^n \longrightarrow \{true, \bot_{bool}\}$ resp.) is a continuous function (in the sense of domain theory). To consider Scott domains [Sc82] as carriers is natural, since we are looking for languages with the ability of computing with infinite objects defined as limits of finite approximations. In each instance of the scheme, a fixed signature and structure are given, and references to them will be omitted in many cases.

A *valuation* $\alpha \in \mathcal{VAL}$ is an application $\alpha : V \longrightarrow D$. Every valuation α can be extended to $\mathcal{T}_F(V)$ and $\mathcal{B}_{\Pi,F}(V)$. If $\varphi \in \mathcal{CON}_\Sigma$ is a quantifier free constraint and $\alpha(b) = true$ for each component b of φ, we say that α *satisfies* φ, and write

$\alpha \models \varphi$. We say that α satisfies $\exists \overline{U} \varphi$ if there is α' which satisfies φ and coincides with α over $V - \overline{U}$.

Programs will be sets of defining rules for a new set Δ of *defined* (or *non primitive*) functions. With Δ the syntactic domains are enlarged to $\mathcal{T}_{F \cup \Delta}(V)$ (*expressions*), $\mathcal{B}_{\Pi, F \cup \Delta}(V)$ (*atoms*) and $\mathcal{CON}_{\Sigma \cup \Delta}$ (*constraints*). Notice that atoms in constraints $\in \mathcal{CON}_{\Sigma \cup \Delta}$ are built with primitive predicates and arguments possibly with non primitive subexpressions. Every $f \in \Delta$ is defined through a non empty set of *constrained conditional rewrite rules* of the form

$$\underbrace{f(X_1, \ldots, X_n)}_{head} = \overbrace{\underbrace{e}_{body} \Leftarrow \underbrace{\varphi}_{constraint}}^{right\ hand\ side}$$

where X_1,...,X_n are distinct variables, $e \in \mathcal{T}_{F \cup \Delta}(V)$ and $\varphi \in \mathcal{CON}_{\Sigma \cup \Delta}$ (and can be omitted if $\varphi = true$). Note that non primitive expressions may appear in the rhs of a rule.

An *interpretation* $I \in \mathcal{INT}$ assigns to each $f \in \Delta^n$ a continuous function $f^I : D^n \longrightarrow D$. Given $I \in \mathcal{INT}$, every valuation α can be extended to $\mathcal{T}_{F \cup \Delta}(V)$ and $\mathcal{B}_{\Pi, F \cup \Delta}(V)$. We write $\alpha[\![e]\!]_I$ for the value of e under α and I, and $\alpha \models^I \varphi$ for indicating that α satisfies $\varphi \in \mathcal{CON}_{\Sigma \cup \Delta}$ in I.

There is a natural notion of *model* of rules and programs, for which it can be proved that every semantically non ambiguous program P has a *least model* $I_{\mathcal{P}}$; see [Lo92, Lo94].

We account next for the operational semantics. A *goal* is simply a constraint $\varphi \in \mathcal{CON}_{\Sigma \cup \Delta}$. A *solution* for a goal φ is a valuation α such that $\alpha \models^{I_{\mathcal{P}}} \varphi$. We say that a goal φ is *semantically finished* wrt α if α is a solution of $\mid \varphi \mid$. By $\mid \varphi \mid$, the *shell* of φ, we mean the result of replacing in φ all the outermost subexpressions of the form $f(e_1, \ldots, e_m)$ by $\perp$. We use the word 'semantically' because φ may still contain non primitive subexpressions – so in a 'syntactical' sense we have not yet finished the evaluation of defined functions – but their values are irrelevant to the fact that α is a solution of the goal.

In the following, by $[\varphi]_u$ and $\varphi[u \leftarrow e]$ we mean the subexpression of φ at position u, and the replacement in φ of $[\varphi]_u$ by e, respectively.

The computation mechanism proposed in [Lo92] for solving goals consists of a single rule, specifying the *one-step constrained narrowing* relation:

$\varphi \leadsto_n \exists \overline{Y}(\varphi[u \leftarrow e\sigma] \cup \psi\sigma)$ if
(i) $[\varphi]_u = f(e_1, \ldots, e_n)$, with $f \in \Delta$, for some position u in φ
(ii) $f(X_1, \ldots, X_n) = e \Leftarrow \psi$ is a variant (with fresh variables $\overline{X} \cup \overline{Y}$) of a rule
(iii) σ is the substitution $\{X_i \leftarrow e_i\}$.

Syntactically, this notion just expresses a rewrite step; we call it 'narrowing' because the space of solutions is possibly 'narrowed' by means of the added constraint ψ of the rule. With the above definition, constrained narrowing – although sound (see [Lo92]) and even complete, as a corollary of the results below – is a very unrestricted computation rule, since at each step we may choose among different positions u and rules R. In general, we can only expect don't know indeterminism for the rules, but it is desirable to have don't care

indeterminism for the positions, and this is not true for unrestricted narrowing. In [Lo92] an abstract notion of *demanded positions* (with respect to a given solution α) is introduced, so that if we restrict narrowing steps to be performed in those positions, don't care indeterminism wrt the positions is achieved, while preserving completeness, as stated in Theorem 1 below. A computation is called *lazy* (wrt α) if all its steps are performed in demanded positions (wrt α).

We have the following completeness results ([Lo92], also [Lo94]).

Theorem 1. *(Completeness of lazy constrained narrowing) Let α be a solution of φ. Then, for any position u in φ which is demanded wrt α, there is a lazy computation $\varphi \leadsto_n{}^* \psi$, starting at position u, such that α is also a solution of ψ and ψ is semantically finished wrt α.*

Lazy constrained narrowing is not enough as computation mechanism, if we want to design real programming languages as instances of the scheme. Some of the obvious shortcomings are:

- Constrained narrowing is simply rewriting. No notion of constraint solving and simplification mechanism is considered, and hence computed answers would consist of an unreadable mess of constraints.
- Furthermore, the notion of finished computation is semantic (it refers to a particular solution, and finished goals may contain non primitive parts). When solving a goal in practice, no solution is known in advance, and we would expect to obtain as answers a (possibly infinite) collection of primitive constraints in some kind of solved form.

As an improvement of the scheme (proofs are omitted; the interested reader is referred to [Lo94]), we postulate the existence, over the base structure, of a *constraint solving* mechanism, formalized through a *one step constraint solving* relation $\varphi \leadsto_{cs} \psi$ among (possibly non primitive) constraints $\varphi, \psi \in \mathcal{CON}_{\Sigma \cup \Delta}$. We require of $\leadsto_{cs}$ to satisfy the following properties [2]:

- *Soundness*: For any α, I, if $\varphi \leadsto_{cs} \psi$ and $\alpha \models^I \psi$ then $\alpha \models^I \varphi$.
- *Completeness*: For any α, I, if φ is $\leadsto_{cs}$-reducible and $\alpha \models^I \varphi$ then $\varphi \leadsto_{cs} \psi$ and $\alpha \models^I \psi$, for some ψ.
- *Termination*: There is no infinite chain of $\leadsto_{cs}$-reductions.
- *Laziness*: If $\varphi \in \mathcal{CON}_{\Sigma \cup \Delta}$ is non primitive and $\mid \varphi \mid$ is satisfiable, then φ is $\leadsto_{cs}$-reducible.

If we consider now $\leadsto_{ncs} = \leadsto_n \cup \leadsto_{cs}$ as our improved operational semantics, we can prove (see [Lo94]) the following completeness result.

Theorem 2. *(Completeness of lazy constrained narrowing, improved version) Let α be a solution of φ. Then there exist a lazy computation $\varphi \leadsto_{ncs}{}^* \psi$ such that ψ is a primitive $\leadsto_{cs}$-irreducible constraint, and α is also a solution of ψ.*

[2] As our results will refer to successful computations, we do not consider in this general setting *failure rules* for detecting unsatisfiable constraints. For practical purposes, we will introduce them in the next section, where this general framework is to be applied to our language.

4 $SFL_{\neq}$ as an instance of $CFLP(X)$

Given a set CS of constructor symbols, $SFL_{\neq}$-programs can be realized as programs over the following instance $CFLP(\mathcal{IH}_{CS})$ of our scheme (we define the structure, the signature becoming apparent from it):

The base domain is the infinitary Herbrand domain [GLMP91, MR92] $\mathcal{IH}_{CS}$ determined by CS [3] together with an added $\perp$ element, i.e., the set of all finite or infinite, totally or partially defined trees, built with aid of $CS \cup \{\perp\}$. The ordering is defined by stating that $\perp$ is the bottom element and constructors in CS are monotonous in all their arguments. The finite and total (in the sense of domain theory) elements are the finite and totally defined (with no $\perp$ in the leaves) trees.

Among the primitive functions we have the constructors in CS, interpreted as *free non strict functions*. Among the primitive predicates we have $==$ (*strict equality*) and $=/=$ (*inconsistency* in $\mathcal{IH}_{CS}$), defined as
$(t_1 == t_2) = true$ iff t_1, t_2 are equal, finite and total.
$(t_1 =/= t_2) = true$ if t_1, t_2 have different constructors at the same position.
It can be proved that $==$ is the maximal continuous approximation to 'true' equality $=$ in $\mathcal{IH}_{CS}$ [4]. The same happens with $=/=$ with respect to $\neq$ (the logical negation of $=$). We remark that $=/=$ is not the logical negation of $==$.

As $SFL_{\neq}$-rules have linear patterns in their heads, we must also get unification of an expression e and a pattern t by means of primitive operations. We cannot use $e == t$ for this purpose, because this would preclude any successful unification when e denotes an infinite tree, destroying the laziness of the language. What we need is indeed $e = t$, facing at the problem that $=$ is not continuous. We remark at this point that unification $e = t$ plays two roles: imposing conditions on the 'shape' of e, and accessing to its components (through the variables in t, possibly used in the rhs of the rule). These two aspects can be expressed by introducing new continuous primitive operations:

- *Recognizer* predicates is_c, for each $c \in CS$: $is_c(t) = true$ if $t = c(t_1, \ldots, t_n)$ ($\perp$ otherwise)
- *Selector* functions c_k, for each $c \in CS^n, 1 \leq k \leq n$: $c_k(t) = t_k$ if $t = c(t_1, \ldots, t_n)$ ($\perp$ otherwise)

Rules with patterns can be now considered as a complex syntactic sugar for $CFLP$-rules using recognizers and selectors [5]. It is not difficult to formalize the 'unsugaring', which we explain here with an example.

For the rule (mb_3) of the example in Sec. 2 we would have

$$member(X, L) = member(X, Ys) \Leftarrow L = [Y \mid Ys], X =/= Y.$$

[3] We assume at least one non nullary constructor in CS; otherwise $\mathcal{IH}_{CS}$ is a finite domain, for which our later developments would not serve.

[4] Equality $=$ in $\mathcal{IH}_{CS}$ is not continuous (and hence not computable), since $\perp = \perp$ and monotonicity would imply $t = s$ for all trees $t, s \in \mathcal{IH}_{CS}$.

[5] The linearity condition for patterns is essential for this. Unification (understood as 'true' equality) with non linear patterns is *not* continuous.

when expressing unification by means of $=$-equations, and

$$member(X, L) = member(X, [.|.]_2(L)) \Leftarrow is_{[]}(L), X =/= [.|.]_1(L).$$

when using 'pure' CFLP-syntax.

For the rest of the paper, we restrict our attention to $CFLP(\mathcal{IH}_{CS})$-programs obtained by unsugaring rules with patterns, and to initial $SFL_{\neq}$-goals consisting of a system of strict equations and disequalities (which are also valid $CFLP$-goals). Furthermore, although recognizers and selectors have been introduced in order to realize $SFL_{\neq}$ as an instance of $CFLP(X)$, we will prefer to formulate the operational semantics in terms of the intermediate level of translation, where unification is expressed through equality $=$, hence avoiding the explicit use of is_c and c_k primitives. As a consequence, primitive expressions occurring during computations are all constructor terms.

We propose next a constraint solver for our language, which can be proved [Lo94] to fulfil the conditions required in the previous section, for the class of goals reachable by $\leadsto_{ncs}$ steps. Therefore, its combination with constrained narrowing results on a complete operational semantics for our language.

4.1 Rules for constraint solving

In the following rules t, t_i are terms $\in \mathcal{T}_F(V)$, and e, e_i, l_i, r_i are expressions $\in \mathcal{T}_{F \cup \Delta}(V)$.

- **Rules for** $=$

  $=_1$) $c(e_1, \ldots, e_n) = c(t_1, \ldots, t_n), \varphi \leadsto e_1 = t_1, \ldots, e_n = t_n, \varphi$
  if $c \in CS^n$

  $=_2$) $c(e_1, \ldots, e_n) = d(t_1, \ldots, t_m), \varphi \leadsto FAIL$
  if $c \not\equiv d$.

  $=_3$) $X = t, \varphi \leadsto X = t, \varphi\sigma$
  if t is not a variable, X occurs in φ, $\sigma \equiv \{X/t\}$.

  $=_4$) $e = X, \varphi \leadsto \varphi\sigma$
  where $\sigma \equiv \{X/e\}$.

  These constitute a set of rules for lazy (rule $=_4$) occur-check free (rules $=_3$ and $=_4$) unification. Remark that, due to the conditions imposed on program rules, only linear terms may appear in the right hand side of $=$-equations.

- **Rules for** $==$

  $==_1$) $c(l_1, \ldots, l_n) == c(r_1, \ldots, r_n), \varphi \leadsto l_1 == r_1, \ldots, l_n == r_n, \varphi$
  if $c \in CS^n$.

  $==_2$) $c(l_1, \ldots, l_n) == d(r_1, \ldots, r_m), \varphi \leadsto FAIL$
  if $c \not\equiv d$.

  $==_3$) $Unif, X == e, \varphi \leadsto Unif, X == sk(e), (U_1 == e_1, \ldots, U_k == e_k, \varphi)\sigma$
  if X occurs in e, φ, X not occurs in $\mid e \mid$, $\sigma \equiv \{X/sk(e)\}$.
  $Unif$ is the set of equations $=$.
  $sk(e)$ is the result of replacing all the outermost non primitive subexpressions $e_1, \ldots, e_k$ of e by fresh variables $U_1, \ldots, U_k$.

This is an improved 'Imitate' rule which performs the binding X/e if e is a primitive term. Otherwise, it performs a partial binding for X. To leave $Unif$ unaffected by substitutions coming from strict disequalities is required for maintaining unification free of occur-check, hence making possible don't care selection of $\leadsto_{cs}$-steps. If for practical purposes substitutions are globally applied, a restriction on the ordering of $\leadsto_{cs}$-steps must be imposed: $=$-equations coming from a program rule in a narrowing step must be completely solved before rule $==_3$ could be applied to strict equations of the same program rule. This will be done so in the specification of the next section.

$==_4$) $X == e, \varphi \leadsto FAIL$
if $X \not\equiv e$, X occurs in $\mid e \mid$.
$==_5$) $e == X, \varphi \leadsto X == e, \varphi$
if e is not a variable.

– **Rules for** $=/=$

$=/=_1$) $c(l_1, \ldots, l_n) =/= c(r_1, \ldots, r_n), \varphi \leadsto l_i =/= r_i, \varphi$
if $c \in CS^n$, $1 \leq i \leq n$.
$=/=_2$) $c(l_1, \ldots, l_n) =/= d(r_1, \ldots, r_m), \varphi \leadsto \varphi$
if $c \not\equiv d$.
$=/=_3$) $X =/= X, \varphi \leadsto FAIL$
$=/=_4$) $e =/= X, \varphi \leadsto X =/= e, \varphi$
if e is not a variable.
$=/=_5$) $X =/= c(e_1, \ldots, e_n), \varphi \leadsto X = d(U_1, \ldots, U_m), \varphi$
if $c(e_1, \ldots, e_n)$ is not a term, $c \neq d$, $U_1, \ldots, U_m$ new variables.
$=/=_6$) $X =/= c(e_1, \ldots, e_n), \varphi \leadsto X = c(U_1, \ldots, U_n), U_i =/= e_i, \varphi$
if $c(e_1, \ldots, e_n)$ is not a term, $U_1, \ldots, U_m$ new variables, $1 \leq i \leq n$.

The last two rules are needed for ensuring the laziness condition of $\leadsto_{cs}$.

Notice that the information given by solved equations of the form $X = t$ is different to the case $X == t$. The first accepts as solutions finite or infinite, total or partial values of X, whereas the second only admits finite and total solutions. Also, equations $X == X$ are not trivially satisfiable since they express that X must be finite and total. Something similar happens with disequations $X =/= t$, where X occurs in t, because some infinite trees do not satisfy that disequation. Nevertheless, if we restrict our interest only to finite and total solutions, we can ignore the difference among $X = t$ and $X == t$ (and treat both as substitutions) and remove equations $X == X$ or disequations $X =/= t$, with X occurring in t. This does not affect the lazy nature of the language, and will be done in our implementation, because successful terminating computations involving infinite values are possible.

5 An executable Prolog specification for lazy narrowing with disequalities

In this section we describe an executable Prolog specification for lazy narrowing with disequalities. The specification is presented as a compilation to Prolog which converts any given $SFL_{\neq}$ program P, into a set of clauses $PTC(P)$. The PTC translation can be chosen to realize different strategies for lazy constrained narrowing. This approach extends previous work for lazy narrowing [LLR93], where the reader may find more details on lazy strategies.

The set of clauses $PTC(P)$ which specifies lazy narrowing with disequalities depends on the sets CS,Δ and the set of rules which define a $SFL_{\neq}$ program P. The variables and expressions of the language are represented as Prolog variables and terms respectively. However, at runtime disequalities generate constrained variables, whose representation is a little more complex, in order to be able to keep their constraints. In order to abstract from the representation details, we postpone them until section 5.3 and we develop our specification using the following auxiliary predicates, whose concrete definitions depend on the representation of variables.

- *is_var*(E): checks if E is a variable with or without constraints.
- *is_c_apply*$(E, c(E_1, \ldots, E_n))$: succeeds if E represents an expression $c(e_1, \ldots, e_n)$ for $c \in CS^n$ and E_i represents e_i for each $i \in \{1, \ldots, n\}$.
- *is_f_apply*$(E, f(E_1, \ldots, E_n))$: succeeds if E represents an expression $f(e_1, \ldots, e_n)$ for $f \in \Delta^n$ and E_i represents e_i for each $i \in \{1, \ldots, n\}$.
- *same_var*(X, Y): determines if X,Y represent the same variable.
- *constrained_to_be_different*(X, Y): checks if X,Y represent variables RX, RY respectively, such that $RX =/= RY$ is one of the constraints of RX and vice versa.
- *constraints*(X, C): returns in C the constraints of X in such a way that if X is a constraint free variable C satisfies the predicate *is_empty*(C).
- *add_constraints*(C, X): adds to X the constraints given in C.
- *add_term_as_const*(T, X): adds to X the constraint $X =/= T$.
- *bind*(X, T): binds the variable X to T. The effect of such binding is global, that is, affects to all the occurrences of the variable X.

The set of clauses $PTC(P)$ executed under Prolog, computes the same solutions that lazy narrowing with disequalities applied to P. Note that the substitutions computed by narrowing are subsumed by Prolog unification.

5.1 Solving a goal with incremental occur check

We first specify how a constraint consisting of strict equations and disequations is solved. The constraint solver *solve* is defined by the following clauses:

solve$((Cond, Rest))$:– *solve*$(Cond)$, *solve*$(Rest)$.
solve$(L == R)$:– *equal*(L, R).
solve$(L =/= R)$:– *not_equal*(L, R).

The specification of equality is an optimized version of the given one in [LLR93] for SFL, but now an equation $X == Y$ between variables is not resoluble if the variables are constrained to be different. In other case, X is bound to Y after adding the constraints of X to Y.

$equal(L, R) :- hnf(L, HL), hnf(R, HR), eq\_hnf(HL, HR).$

$eq\_hnf(X, H) :- is\_var(X), !, eq\_var(X, H).$
$eq\_hnf(H, X) :- is\_var(X), !, eq\_var(X, H).$
$eq\_hnf(L, R) :- is\_c\_apply(L, c(L_1, \ldots, L_n)), is\_c\_apply(R, c(R_1, \ldots, R_n)),$
$equal(L_1, R_1), \ldots, equal(L_n, R_n).$ % for each $c \in CS^n$ $(n \geq 0)$

$eq\_var(X, Y) :- is\_var(Y), same\_var(X, Y), !.$
$eq\_var(X, Y) :- is\_var(Y), !, not\ constrained\_to\_be\_different(X, Y),$
$constraints(X, C), add\_constraints(C, Y), bind(X, Y).$
$eq\_var(X, E) :- is\_c\_apply(E, c(E_1, \ldots, E_n)), occursnot(X, E, ShE, Con),$
$unify(X, ShE), cont\_eq\_var(Con).$ % for each $c \in CS^n$ $(n \geq 0)$

$occursnot(X, Y, Y, L/L) :- is\_var(Y), !, not\ same\_var(X, Y).$
% the skeleton of Y is Y.
$occursnot(X, E, c(Sk_1, \ldots, Sk_n), L_0/L_n) :- is\_c\_apply(E, c(E_1, \ldots, E_n)), !,$
$occursnot(X, E_1, Sk_1, L_0/L_1), \ldots,$
$occursnot(X, E_n, Sk_n, L_{n-1}/L_n).$
% The skeleton of E is $c(sk(E_1), \ldots, sk(E_n))$, for each $c \in CS^n$ $(n \geq 0)$
$occursnot(X, E, U, [U == E|L]/L).$ % The skeleton of E is a new variable

$cont\_eq\_var([]/[]) :- !.$
$cont\_eq\_var([E_1 == E_2|L_1]/L_2) :- equal(E_1, E_2), cont\_eq\_var(L_1/L_2).$

The predicate $occursnot(X, E, Sk, Con)$ simultaneously checks that X does not occur in the *skeleton* of E, builds the skeleton of E in Sk and collects in Con the constraints which are still to be solved.

Disequations are solved by:

$not\_equal(L, R) :- hnf(L, HL), hnf(R, HR), neq\_hnf(HL, HR).$

$neq\_hnf(X, Y) :- is\_var(X), !, neq\_var(X, Y).$
$neq\_hnf(X, Y) :- is\_var(Y), !, neq\_var(Y, X).$
$neq\_hnf(L, R) :- is\_c\_apply(L, c(L_1, \ldots, L_n)), is\_c\_apply(R, d(R_1, \ldots, R_m)).$
% for each $c \in CS^n$ $(n \geq 0)$ and $d \in CS^m$ $(m \geq 0)$, $c \neq d$
$neq\_hnf(L, R) :- is\_c\_apply(L, c(L_1, \ldots, L_n)), is\_c\_apply(R, c(R_1, \ldots, R_n)),$
$(not\_equal(L_1, R_1)\ ;\ \ldots\ ;\ not\_equal(L_n, R_n)).$
% for each $c \in CS^n$ $(n \geq 0)$

$neq\_var(X, Y) :- is\_var(Y), !, not\ same\_var(X, Y),$
$add\_term\_as\_const(Y, X), add\_term\_as\_const(X, Y).$
$neq\_var(X, E) :- is\_term(E), !, add\_term\_as\_constraint(E, X).$
$neq\_var(X, E) :- is\_c\_apply(E, c(E_1, \ldots, E_n)), unify(X, d(U_1, \ldots, U_m)).$
% for each $c \in CS^n$ $(n \geq 0)$ and $d \in CS^m$ $(m \geq 0)$, $c \neq d$
$neq\_var(X, E) :- is\_c\_apply(E, c(E_1, \ldots, E_n)), unify(X, c(U_1, \ldots, U_n)),$
$not\_equal(X, E).$ % for each $c \in CS^n$ $(n > 0)$

$is\_term(X) :- is\_var(X), !.$
$is\_term(T) :- is\_c\_apply(T, c(T_1, \ldots, T_n)), is\_term(T_1), \ldots, is\_term(T_n).$
% for each $c \in CS^n$ $(n \geq 0)$

The specification of $=/=$ establishes that a disequality $X =/= Y$ is not solvable if X and Y are the same variable. In other case, the constraint $X =/= Y$ must be added to both variables. On the other side, all the constraints added to variables are finite and total terms. So, for $X =/= e$ where e is not such a term, a partial binding is guessed to unify e with X, solving the disequality lazily.

5.2 Rule application and computation of head normal forms

The predicate $hnf(E, H)$ returns in H one of the possible head normal forms of the expression E, resulting of narrowing E. The following specification of *hnf* is analogous to that given for the 'naïve strategy' for SFL in [LLR93] [6]. To compute the head normal form of $f(E_1, \ldots, E_n)$ a call to the predicate $\#f$ is performed. There is an $(n+1) - ary$ predicate $\#f$ for each $n - ary$ function f in the program.

$hnf(E, H) :- is\_var(E), !, H = E.$
$hnf(E, H) :- is\_c\_apply(E, c(E_1, \ldots, E_n)), !, H = c(E_1, \ldots, E_n).$
% for each $c \in CS^n$ $(n \geq 0)$
$hnf(E, H) :- is\_f\_apply(E, f(E_1, \ldots, E_n)), !, \#f(E_1, \ldots, E_n, H).$
% for each $f \in \Delta^n$ $(n \geq 0)$

$\#f(E_1, \ldots, E_n, H) :- unify(E_1, t_1), \ldots, unify(E_n, t_n), solve(b), hnf(e, H).$
% for each defining rule $f(t_1, \ldots, t_n) = e \Longleftarrow b$ in P

The predicate $unify(E, T)$, unifies the expression E and the linear term T, reducing E by lazy narrowing as much as demanded by the constructors occurring in T. Note that the first clause for *unify_hnf* establishes that to unify a variable X with constraints $X =/= t_1, \ldots, X =/= t_n$ and a term t, first the constrained variable X is bound to the term t, and afterwards the new generated constraints $t =/= t_1, \ldots, t =/= t_n$ must be solved.

$unify(E, T) :- var(T), !, T = E.$
$unify(E, T) :- hnf(E, H), unify\_hnf(H, T).$

$unify\_hnf(E, T) :- is\_var(E), !, bind(E, T),$
$constraints(E, C), propagate(T, C).$
$unify\_hnf(E, c(T_1, \ldots, T_n)) :- is\_c\_apply(E, c(E_1 \ldots, E_n)),$
$unify(E_1, T_1), \ldots, unify(E_n, T_n).$
% for each $c \in CS^n$ $(n \geq 0)$

$propagate(T, C) :- is\_empty(C).$
$propagate(T, C) :- select(S, C, Rest), solve(T =/= S), propagate(T, Rest).$
% $selec(S, C, Rest)$ returns in S a constraint of C and the rest of them in $Rest$

[6] An specification of the 'demand driven strategy' given in [LLR93] is also possible.

5.3 Representation of the computation expressions

An important point in our specification for lazy narrowing with disequalities is the Prolog representation of the constrained variables. Such representation must be accurated to be able to define the auxiliary predicates pending of specification.

The *constraint free variables* do not present any problem, as they can be represented as Prolog variables. However, for a *constrained variable* X, it seems natural to use a representation as a Prolog term *neq(RX,C)* where: RX is a Prolog variable which will possibly hold instances of the variable X in future computations; and C will keep the constraints for X. C can be represented as a Prolog list $[t_1, \ldots, t_n]$, where $t_1, \ldots, t_n$ are the constraints for X.

Let us discuss the effect of the predicates *bind* and *add_term_as_const* when acting on constrained variables under this representation. To bind a constrained variable X represented by a Prolog term $neq(RX, C)$, the Prolog variable RX would be instantiated. Hence, sometimes it will be necessary to dereference the Prolog term $neq(RX, C)$. To add a constraint $X =/= t$ to a constrained variable X represented by $neq(RX, C)$, RX could also be instantiated to the term $neq(RZ, [t|C])$, but again it would be needed to dereference this expression to access to the real value of the Prolog term. Instead of that, we use a *partial list* of the form $[t_1, \ldots, t_k|R]$ (R variable) to represent the list of constraints of the variables. In this way, to add $X =/= t$ to $neq(RX, C)$ we can instantiate C instead of RX, avoiding dereferenciation.

Following this representation for variables, the specification for the auxiliary predicates is straightforward and due to lack of space it is omitted but can be found in [Lo94].

Let us note that according to our specification, disequality $X =/= Y$ between different variables is solved by adding the constraint $X =/= Y$ to both variables, producing circular references from our representation of constrained variables. For instance, if we consider the simplest case in which both variables are constraint free, then the following Prolog unification problem comes up:

$$X = neq(RX, [Y|LX]), Y = neq(RY, [X|LY])$$

Prolog solves this problem by producing cyclic terms. We propose to exploit the lack of "occur check" in most Prolog systems by accepting the cyclic terms as a part of our representation. In the context of our specification, it is guaranteed that the arising cyclic structures are harmless for the computation. From another point of view, it may be argued that we are using a subset of Prolog II, supported by Prolog.

Although for the particular disequality $X =/= Y$, this problem could be avoided by adding the constraint only to one of both variables, this is not true in general.

6 Implementation issues

The given Prolog specification can be viewed as an executable Prolog program, but for an efficient implementation, several optimizations are required. The currently existing developed environment has been implemented to support sharing

-following a technique introduced by Cheong [Ch90]- and the two lazy strategies from [LLR93], extended to $SFL_{\neq}$. Moreover, we have incorporated several optimizations which improve the efficiency in solving goals.

The environment has been implemented in BimProlog, and can be executed on a SPARC Station 1+ under SUNOS 4.1.1 or higher. The environment works as a translator of $SFL_{\neq}$ programs into Prolog, following one of the mentioned lazy strategies.

To illustrate the performance of our system we use the $SFL_{\neq}$ program presented in section 2. The results have been computed under the naïve strategy. The table 1 shows the number of computed solutions for the corresponding goals, and the time consumed to obtain them, including the needed time to detect that a goal has no more solutions. Time is measured in seconds.

The goals -which are scheme goals over the parameter N- and some of their computed answers are the following:

- $\mathbf{G_1}$: $size\ [A1, \ldots, AN] == S$. Answers:
  $\{S == s(0), A1 == A2 == \ldots == AN\}$,
  $\{S == s^2(0), A1 =/= A2, A2 == \ldots == AN\}$, and so on.
- $\mathbf{G_2}$: $size\ [A1, \ldots, AN] =/= size[B1, \ldots, BN]$. Answers:
  $\{A1 == A2 == \ldots == AN, B1 =/= B2, B2 == \ldots == BN\}$,
  $\{A1 == A2 == \ldots == AN, B1 =/= B2, B1 =/= B3, B2 =/= B3,$
  $B3 == \ldots == BN\}$, and so on.
- $\mathbf{G_3}$: $size\ [1, \ldots, M] == S$ asks the size of a list of different numbers $1 \ldots M$, where $M = N * 10$. Answer: $\{S == s^M(0)\}$.

N	3		5		7		10	
Goal	Time	N. sols	Time	N. sols	Time	N. sols	Time	N. sols
$\mathbf{G_1}$	0.05	5	0.70	52	15.31	877	2590	115975
$\mathbf{G_2}$	0.22	14	32.20	1704	11442	505408	-	-
$\mathbf{G_3}$	0.34	1	0.84	1	1.59	1	3.09	1

Table 1. Performance results

7 Conclusions and related work

We have presented an extension of a lazy functional logic language which allows for the use of disequations both in programs and answers, thus increasing substantially the expressivity. The language has well-founded semantics, derived from general results about the CFLP(X) scheme from which it is an instance, Also an implementation, based on a Prolog specification, is reported.

Our language differs from those designed as CLP(X)-instances where some term (tree) algebra is chosen as underlying domain and the constraint language uses $=$ and $\neq$ for primitive atomic constraints (see f.i. [Sm91, DR93]). Firstly, in most of the cases the underlying universe consists only of *finite* trees (classes

of finite trees, if a quotient algebra). Secondly, in *all* these CLP-languages $=$ is interpreted as *true* equality, and $\neq$ as true disequality, i.e. the logical negation of $=$, while in $SFL_{\neq}$ we are enforced, because of the lazy functional component of the language, to work with approximations $==$, $=/=$ of $=, \neq$ (coinciding with them for finite and totally defined trees). This makes the difference, at the constraint system level, between our language and Prolog II, whose universe also has (rational) infinite trees [7]. As a consequence, at the programming level, the way of defining infinite trees differs greatly in both languages: In Prolog II we can 'define' an infinite tree by means of constraints, eg. X=s(X), while the strict equation $X == s(X)$ in $SFL_{\neq}$ would fail. As a counterpart, we can define in $SFL_{\neq}$ a constant function f as $f = s(f)$ denoting the same infinite tree (and also non rational trees can be defined). This is not possible in Prolog II, of course, since the possibility of defining functions is not allowed. Furthermore, as argued in [Lo92, Lo94], a reasonable functional extension of Prolog II would leave the above function f as undefined.

Our work is mostly related to [KLMR92], where an abstract machine is proposed for implementing a similar language, but there are significant departure points. First, in [KLMR92] programs are restricted to be *uniform programs* [MKLR90], and both the operational semantics and the design of the abstract machine rely upon this assumption. Although every program can be converted into an equivalent, uniform one (cfr. [MKLR90]), this is an unnecessary restriction. Most importantly, no completeness result is given in [KLMR92] with respect to any suitable declarative semantics. Finally, it seems to us that our Prolog specification is a simpler way to obtain an implementation than the abstract machine approach of [KLMR92], for which we do not know any implementation. To implement the language on top of an efficient Prolog system results in acceptable performance, while discharging the implementor of many hard tasks. We think also that future improvements would be easier using Prolog as target language.

Acknowledgement

We thank Mario Rodríguez-Artalejo for many valuable discussions.

References

[Ch90] P.H. Cheong: *Compiling lazy narrowing into Prolog*, Technical Report 25, LIENS, 1990, to appear in: Journal of New Generation Computing.

[Co82] A. Colmerauer: *Prolog and infinite trees*, in K.L. Clark, S.A. Tarnlund (eds.) Logic Programming, Academic Press, 1982, 231-251.

[Co84] A. Colmerauer: *Equations and inequations on finite and infinite trees*, Procs. FGCS'84, 1984, 85-99.

[7] To think about extending Prolog II constraint language to the universe of all infinite trees makes really no sense, since the corresponding theories are elementarily equivalent [Ma88].

[Co92] 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.

[DR93] A. Dovier, G.F. Rossi: *Embedding extensional finite sets in CLP*, Procs. Int. Logic Progr. Symp., MIT Press, 1993.

[GLMP91] 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.

[GHR92] J.C. González Moreno, M.T. Hortalá González, M. Rodríguez Artalejo: *On the Completeness of Narrowing as the Operational Semantics of Functional Logic Programming*, Procs. Computer Science Logic CSL'92, LNCS 702,1993, 216-230.

[Ha94] M. Hanus: *The Integration of Functions into Logic: From Theory to Practice*, to appear in J. of Logic Progr. Also available as Tech. Rep. MPI-I-94,201, 1994.

[JL87] J. Jaffar, J.L. Lassez: *Constraint Logic Programming*, Procs. 14th ACM Symp. on Princ. of Prog. Lang., 1987, 114-119.

[JL86] J. Jaffar, J.L. Lassez: *Constraint Logic Programming*, Tech. Rep. 86/73, Dep. of Computer Science, Monash University, 1986.

[JM94] J. Jaffar,M.J. Maher: *Constraint Logic Programming: A Survey*, To appear in Journal of Logic Programming.

[JK92] 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.

[KLMR92] H. Kuchen, F.J. López-Fraguas, J.J. Moreno-Navarro, M. Rodríguez-Artalejo: *Implementing Disequality in a Functional Logic Language*, Procs. Joint Int. Conf and Symp. on Logic Programming, MIT Press, 1992, 207-221.

[LLR93] R. Loogen, F. J. López Fraguas, M. Rodríguez Artalejo: *A Demand Driven Computation Strategy for Lazy Narrowing*, Int. Symp. on Programming Language Implementation and Logic Programming (PLILP) 93, LNCS 714, 184-200.

[Lo94] F.J. López-Fraguas: *Constraint Functional Logic Programming*, PhD. dissertation (forthcoming).

[Lo92] F.J. López-Fraguas: A General Scheme for *Constraint Functional Logic Programming*, Procs. ALP, LNCS 632, 1992, 213-217.

[Ma88] M. Maher: *Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees*, Procs. 3rd IEEE Symp. Logic in Computer Science, 1988, 348-357.

[MKLR90] J. J. Moreno-Navarro, H. Kuchen, R. Loogen, M. Rodríguez-Artalejo: *Lazy Narrowing in a Graph Machine*, ALP, LNCS 463, 1990, 298-317.

[MR92] J.J. Moreno-Navarro, M. Rodríguez-Artalejo: *Logic Programming with Functions and Predicates: The Language BABEL*, J. Logic Programming, 12, 1992, 189-223.

[Re85] U.S. Reddy: *Narrowing as the Operational Semantics of Functional Languages*, Procs. Int. Symp. on Logic Programming, 1985, 138-151.

[Sc82] D.S. Scott: *Domains for Denotational Semantics*, Procs. ICALP, LNCS 140,1892, 577-613.

[Sm91] D.A. Smith: *Constraint Operations for CLP(FT)*, Procs. 8th Int. Conf. on Logic Programming, MIT Press, 1991, 760-774.