Designing an Efficient Computation Strategy in $CFLP(\mathcal{FD})$ Using Definitional Trees

Sonia Estévez Martín   Rafael del Vado Vírseda *

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
C/ Profesor José García Santesmases s/n, 28040 Madrid
{s.estevez,rdelvado}@sip.ucm.es

Abstract

This paper proposes the integration of *finite domain* ($\mathcal{FD}$) constraints into a general purpose lazy *functional logic programming* language by means of a concrete instance of the generic scheme $CFLP(\mathcal{D})$, proposed in [19] for lazy *Constraint Functional Logic Programming* over a parametrically given constraint domain $\mathcal{D}$. We sketch in this $CFLP(\mathcal{FD})$ language the basis of an efficient computation strategy for solving goals for programs by using *definitional trees* [1] in order to efficiently control the computation and maintain the good properties shown for *needed* and *demand-driven narrowing strategies* [4, 15, 25] in functional logic programming. This convenient computation mechanism is obtained as an optimization of the generic *Constrained Lazy Narrowing Calculus* $CLNC(\mathcal{D})$ presented in [20], which has been proved sound and strongly complete w.r.t. a suitable $CFLP(\mathcal{D})$ semantics, and provides a formal foundation for efficient implementations in existing systems such as $Curry$ [11] and $\mathcal{TOY}$ [17]. Finally, we describe the execution of an example implemented in the $CFLP(\mathcal{FD})$ system called $\mathcal{TOY}(\mathcal{FD})$ [9], which is based on the theoretical ideas introduced in this paper, following our computation strategy.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.6 [*Programming Techniques*]: Logic Programming; D.3.2 [*Programming Languages*]: Language Classifications—Multiparadigm languages; D.3.2 [*Programming Languages*]: Language Classifications—Constraint and logic languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Constraints; G.2.2 [*Discrete Mathematics*]: Graph Theory—Trees

General Terms Algorithms, Languages, Performance, Theory.

Keywords Functional Logic Languages, Constraint Logic Programming, Finite Domains, Narrowing, Definitional Trees.

* The work of this author has been partially supported by the Spanish National Project MELODIAS (TIC2002-01167).

WCFLP'05 September 29, 2005, Tallinn, Estonia.

1. Introduction

The effort to combine the main lines of research in multiparadigm declarative programming, namely *Constraint Logic Programming* (CLP) [27, 13, 14] and *Functional Logic Programming* (FLP) [10], in a unified and suitable framework called *Constrained Functional Logic Programming* ($CFLP$), arose around 1990 and has grown in the last years. Recently, a new generic scheme called $CFLP(\mathcal{D})$ has been proposed in [19] as a logical and semantic framework for lazy Constraint Functional Logic Programming over a parametrically given constraint domain $\mathcal{D}$, which provides a clean and rigorous declarative semantics for $CFLP$ languages like in the $CLP(\mathcal{D})$ scheme, but overcoming some limitations of older $CFLP$ schemes [16, 21, 22]. In this setting, $CFLP(\mathcal{D})$-programs are presented as sets of constrained rewrite rules that define the behavior of possible higher-order and/or non-deterministic lazy functions over $\mathcal{D}$. The main novelties in [19] were a new formalization of constraint domains for $CFLP$ and a new *Constraint ReWriting Logic* $CRWL(\mathcal{D})$ parameterized by a constraint domain $\mathcal{D}$, which provides a logical characterization of program semantics. Further, [20] has extended [19] with a suitable operational semantics, which relies on a new formal notion of constraint solver and a new *Constrained Lazy Narrowing Calculus* $CLNC(\mathcal{D})$ for solving goals for $CFLP(\mathcal{D})$-programs, which can be proved sound and strongly complete w.r.t. $CRWL(\mathcal{D})$'s semantics. These properties qualify $CLNC(\mathcal{D})$ as a convenient computation mechanism for declarative constraint programming languages.

However, efficiency is a major concern for the implementation of $CFLP(\mathcal{D})$ systems, since non-deterministic computations often generate huge search spaces with their associated overheads both in terms of time and space. In the field of functional logic programming languages using lazy narrowing as operational model, *needed narrowing strategies* [4, 2, 12] and *demand-driven narrowing strategies* [15, 25] are known to provide a sound and complete goal solving mechanism while avoiding unneeded computation steps. These strategies are based on *definitional trees*, first introduced in [1], and they have led to efficient implementations of lazy narrowing in existing systems such as $Curry$ [11] and $\mathcal{TOY}$ [17, 9].

Although $Curry$ and $\mathcal{TOY}$ support constraint programming over a few specific domains, general results on the application of a suitable demand/needed constrained narrowing strategy (sound and complete w.r.t. a suitable $CFLP$ semantics), into an efficient implementation of goal and constraint solving are still missing. Among the interesting constraint domains known for their practical value in constraint programming, *finite domain* ($\mathcal{FD}$) are widely used because they allow to naturally model many real life problems [23] (e.g. scheduling, routing and timetabling). The aim of the present paper is to provide the basis of an efficient computation strategy in the concrete instance $CFLP(\mathcal{FD})$. More precisely, this paper uses definitional trees with finite domain constraints to sketch a suitable strategy in the generic sound and strongly complete calculus $CLNC(\mathcal{D})$ over $\mathcal{FD}$ which contracts only needed positions and maintains the efficiency properties shown for existing demand/ needed narrowing strategies. This convenient strategy is implemented in the $CFLP(\mathcal{FD})$ system $\mathcal{TOY}(\mathcal{FD})$ [9] that integrates, as a host language, the higher-order lazy functional logic language $\mathcal{TOY}$ and, as constraint solver, the efficient $\mathcal{FD}$ constraint solver of SICStus Prolog.

The organization of this paper is as follows: In section 2 we describe the $CFLP(\mathcal{FD})$ language as a concrete instance of the generic $CFLP(\mathcal{D})$ scheme [19, 20] over the finite constraint domain $\mathcal{FD}$. We also define the class of $CFLP(\mathcal{FD})$-programs and goals using a refined representation of definitional trees that deals properly with constraints. In Section 3, we give a description of our computation strategy with definitional trees together with an example that demonstrates the usefulness of combining lazy functions with constraint solving over finite domains, exploiting lazy evaluation over infinite data structures. Section 4 describes the execution of a programming example implemented in $\mathcal{TOY}(\mathcal{FD})$. Finally, some conclusions and plans for future work are drawn in section 5.

2. The *CFLP*($\mathcal{FD}$) Language

In this section, we introduce some needed technical preliminaries regarding our instantiation over finite domains of the generic $CFLP(\mathcal{D})$ scheme presented in [19, 20] for lazy Constraint Functional Logic Programming over a parametrically given constraint domain $\mathcal{D}$. We will use this scheme as the logical and semantic framework to define our declarative constraint programs and our computation strategy with definitional trees for goal solving. First, we briefly introduce the syntax of applicative expressions and patterns, which is needed for understanding the construction of constraint finite domains.

2.1 Expressions, Patterns and Substitutions

We assume a *universal signature* $\Sigma = \langle DC, FS \rangle$, where $DC = \bigcup_{n\in\mathbb{N}} DC^n$ and $FS = \bigcup_{n\in\mathbb{N}} FS^n$ are families of countably infinite and mutually disjoint sets of *data constructors* resp. *evaluable function symbols*, each one with an associated arity. Evaluable functions can be further classified into domain dependent *primitive functions* $PF^n \subseteq FS^n$ and user *defined functions* $DF^n = FS^n \setminus PF^n$ for each $n \in \mathbb{N}$. We assume that DC^0 includes the special symbol $\perp$, intended to denote an undefined data value, and the three constants $true$, $false$ and $success$, which are useful for representing the results returned by various primitive functions. Next we assume a countably infinite set $\mathcal{V}$ of *variables* $X, Y, \ldots$ and the integer set $\mathbb{Z}$ of primitive elements 0,1,−1,2,−2,. . ., mutually disjoint and disjoint from Σ. Integer *partial expressions* $e \in Exp_\perp(\mathbb{Z})$ have the following syntax:

$$e ::= \perp \mid u \mid X \mid h \mid (e\, e_1)$$

where $u \in \mathbb{Z}$, $X \in \mathcal{V}$, $h \in DC \cup FS$. The set of variables occurring in e is written $var(e)$. An expression e is called *linear* iff there is no $X \in var(e)$ having more than one occurrence in e. Some interesting subsets of $Exp_\perp(\mathbb{Z})$ are: $GExp_\perp(\mathbb{Z})$, the set of the *ground* expressions e such that $var(e) = \emptyset$ and $Exp(\mathbb{Z})$, the set of the *total* expressions e with no occurrences of $\perp$. Another important subclass of expressions is the set of integer *partial patterns* $s, t \in Pat_\perp(\mathbb{Z})$, whose syntax is defined as follows:

$$t ::= \perp \mid u \mid X \mid c\,\bar{t}_m \mid f\,\bar{t}_m$$

where $u \in \mathbb{Z}$, $X \in \mathcal{V}$, $c \in DC^n$, $m \leq n$, $f \in FS^n$, $m < n$. A *passive symbol* is an integer primitive element $u \in \mathbb{Z}$ or the root symbol h of a pattern of the form $(h\,\bar{e}_m)$ where $h \in DC \cup FS$. We define the *information ordering* $\sqsubseteq$ as the least partial ordering over $Pat_\perp(\mathbb{Z})$ satisfying the following properties: $\perp \sqsubseteq t$ for all $t \in Pat_\perp(\mathbb{Z})$, and $(h\bar{t}_m) \sqsubseteq (h\overline{t'}_m)$ whenever these two expressions are patterns and $t_i \sqsubseteq t'_i$ for all $1 \leq i \leq m$.

As usual, we define integer *substitutions* $\sigma \in Sub_\perp(\mathbb{Z})$ as mappings $\sigma : \mathcal{V} \rightarrow Pat_\perp(\mathbb{Z})$ extended to $\sigma : Exp_\perp(\mathbb{Z}) \rightarrow Exp_\perp(\mathbb{Z})$ in the natural way. We write ε for the identity substitution, $e\sigma$ instead of $\sigma(e)$, and $\sigma\theta$ for the composition of σ and θ, such that $e(\sigma\theta) = (e\sigma)\theta$ for any $e \in Exp_\perp(\mathbb{Z})$. We define the domain $dom(\sigma)$ and the range $ran(\sigma)$ of a substitution σ in the usual way. Finally, a substitution σ such that $dom(\sigma) \cap ran(\sigma) = \emptyset$ is called *idempotent*.

2.2 The Constraint Finite Domain $\mathcal{FD}$

Adopting the general approach of [19, 20], a *constraint finite domain* $\mathcal{FD}$ can be formalized as a structure with carrier set $D_\mathbb{Z}$, consisting of ground patterns built from the symbols in a signature Σ and the set of primitive elements $\mathbb{Z}$. Symbols in Σ are intended to represent data constructors (e.g. the list constructor), domain specific primitive functions (e.g. addition and multiplication over $\mathbb{Z}$) satisfying *monotonicity*, *antimonotonicity* and *radicality* properties (see [19] for details), and user defined functions. Requiring primitives to be *radical* is more novel and just means that for given arguments, they are expected to return a total result, unless the arguments bear too few information for returning any result different of $\perp$.

Assuming then a constraint finite domain $\mathcal{FD}$, we define the syntax of constraints over $\mathcal{FD}$ used in this work. In contrast to $CLP(\mathcal{FD})$, our constraints can include now occurrences of user defined functions.

- *Primitive Constraints* have the syntactic form $p\bar{t}_n \rightarrow!\ t$, with $p \in PF^n$ a primitive function symbol and $t_1, \ldots, t_n, t \in Pat(\mathbb{Z})$ with t total.

- *Constraints* have the syntactic form $p\,\bar{e}_n \rightarrow!\, t$, with $p \in PF^n$, $e_1, \ldots, e_n \in Exp_\perp(\mathbb{Z})$ and $t \in Pat(\mathbb{Z})$ total.

As a concrete example, consider the primitive operators $\otimes^\mathbb{Z}$ with $\otimes \in \{+, -, *, /\}$ and relations $\asymp^\mathbb{Z}$ with $\asymp \in \{=, \neq, <, \leq, >, \geq\}$ defined over $\mathbb{Z}$ in the usual way. *Figure 1* shows a minimum set of primitive functions $p \in PF^n$ and their declarative interpretation $p^{\mathcal{FD}} \subseteq D^n_\mathbb{Z} \times D_\mathbb{Z}$ (we use the notation $p^{\mathcal{FD}}\bar{t}_n \rightarrow t$ to indicate that $(\bar{t}_n, t) \in p^{\mathcal{FD}}$) considered in our constraint finite domain $\mathcal{FD}$. The function $indomain$ covers a primitive *labeling* (*enumeration* or *search*) strategy which is crucial in constraint solving to assign values to each variable. We note that in our framework, various *labeling strategies* have all the same declarative semantics, but they may differ in operational behavior and therefore in efficiency (more details about different labeling strategies can be found in [9]).

In the rest of the paper, when opportune, we use the following notations:

- $t == s$ abbreviates $seq\ t\ s \rightarrow!\ true$ and $t \setminus = s$ abbreviates $seq\ t\ s \rightarrow!\ false$ (the notations $=$ and $\neq$ can be understood as a particular case of the notations $==$ and $\setminus =$ when these are applied to integers and/or variables that may be instantiated to an integer value).
- $a \leq b$ abbreviates $leq\ a\ b \rightarrow!\ true$, $a > b$ abbreviates $leq\ a\ b \rightarrow!\ false$ (and analogously for the other comparison primitives $<$, $>$ and $\geq$).
- $a \otimes b \asymp c$ abbreviates $a \otimes b \rightarrow!\ r, r \asymp c$.

Strict Equality (on patterns)	$seq^{\mathcal{FD}} : D_{\mathbb{Z}} \times D_{\mathbb{Z}} \rightarrow \{true, false, \bot\}$ $seq^{\mathcal{FD}}\ t\ t \rightarrow\ true, \forall t \in D_{\mathbb{Z}}$ total $seq^{\mathcal{FD}}\ t_1\ t_2 \rightarrow\ false, \forall t_1, t_2 \in D_{\mathbb{Z}}.\ t_1, t_2$ have no common upper bound w.r.t. the information ordering $\sqsubseteq$ $seq^{\mathcal{FD}}\ t_1\ t_2 \rightarrow\ \bot,$ otherwise
Less or Equal (on integers)	$leq^{\mathcal{FD}} : D_{\mathbb{Z}} \times D_{\mathbb{Z}} \rightarrow \{true, false, \bot\}$ $leq^{\mathcal{FD}}\ u_1\ u_2 \rightarrow\ true,$ if $u_1, u_2 \in \mathbb{Z}$ and $u_1 \leq^{\mathbb{Z}} u_2$ $leq^{\mathcal{FD}}\ u_1\ u_2 \rightarrow\ false,$ if $u_1, u_2 \in \mathbb{Z}$ and $u_1 >^{\mathbb{Z}} u_2$ $leq^{\mathcal{FD}}\ u_1\ u_2 \rightarrow\ \bot,$ otherwise
Operators (on integers)	$\otimes^{\mathcal{FD}} : D_{\mathbb{Z}} \times D_{\mathbb{Z}} \rightarrow D_{\mathbb{Z}}$ $\otimes^{\mathcal{FD}}\ u_1\ u_2 \rightarrow\ u_1 \otimes^{\mathbb{Z}} u_2,$ if $u_1, u_2 \in \mathbb{Z}$ $\otimes^{\mathcal{FD}}\ u_1\ u_2 \rightarrow\ \bot,$ otherwise
Finite Domains	$domain^{\mathcal{FD}} : D_{\mathbb{Z}} \times D_{\mathbb{Z}} \rightarrow \{true, false, \bot\}$ $domain^{\mathcal{FD}}\ u\ [u_1, \ldots, u_n] \rightarrow\ true,$ if $\forall i \in \{1, \ldots, n{-}1\}.u_i \leq^{\mathbb{Z}} u_{i+1}$ and $\exists i.u =^{\mathbb{Z}} u_i$ $domain^{\mathcal{FD}}\ u\ [u_1, \ldots, u_n] \rightarrow\ false,$ if $\exists i \in \{1, \ldots, n{-}1\}.u_i >^{\mathbb{Z}} u_{i+1}$ or $\forall i.u \neq^{\mathbb{Z}} u_i$ $domain^{\mathcal{FD}}\ u\ [u_1, \ldots, u_n] \rightarrow \bot$, otherwise
Variable Labeling	$indomain^{\mathcal{FD}} : D_{\mathbb{Z}} \rightarrow \{success, \bot\}$ $indomain^{\mathcal{FD}}\ u\ \rightarrow\ success,$ if $u \in \mathbb{Z}$ $indomain^{\mathcal{FD}}\ u\ \rightarrow\ \bot,$ otherwise

Figure 1. Primitive function symbols in $\mathcal{FD}$

- u *in* D abbreviates $domain\ u\ D \rightarrow!\ true$ and $u_1, \ldots, u_n$ *in* D abbreviates u_1 *in* $D \wedge \ldots \wedge u_n$ *in* D.
- $domain\ [u_1, \ldots, u_n]\ a\ b$ with $a, b \in \mathbb{Z}$ $(a \leq b)$ abbreviates $u_1, \ldots, u_n\ in\ [a\ ..\ b]$, where $[a\ ..\ b]$ represents the integer interval $[a, a+1, \ldots, b-1, b]$.
- $labeling\ l\ [u_1, \ldots, u_n]$ abbreviates and extends $indomain\ u_1 \rightarrow!\ success \wedge \ldots \wedge indomain\ u_n \rightarrow!\ success$ with a list l of items that allow to specify different labelling strategies.

For instance, the following usual finite domain constraints can be expressed in our $CFLP(\mathcal{FD})$ language: $domain\ [X, Y]\ 1\ 10$ constraints variables X and Y to have values in the range [1..10], $labeling$ [ff] $[X, Y]$ assigns each value to a variable (in this case, the option ff specifies the variable with the smallest domain), and $X + Y \leq 5$ involves relational and arithmetic constraints between X and Y. Moreover, one can also write the primitive constraint *seq* $X\ Y \rightarrow!\ R$ where R is a variable, whose use in programs can lead to greater expressivity.

2.3 Programs, Goals and Answers over $\mathcal{FD}$

In the $CFLP(\mathcal{FD})$ language, *programs* are presented as sets of constrained rewrite rules that define the behavior of possibly higher-order and/or non-deterministic lazy functions over $\mathcal{FD}$, called *program rules*. More precisely, a program rule R for $f \in DF^n$ has the form $R : f\,\overline{t}_n = r \Leftarrow C$ (abbreviated as $f\,\overline{t}_n = r$ if C is empty) and is required to satisfy the three conditions listed below:

1. The *left-hand side* $f\,\overline{t}_n$ is a linear expression, and for all $1 \leq i \leq n$, $t_i \in Pat(\mathbb{Z})$ are total integer patterns.
2. The *right-hand side* $r \in Exp(\mathbb{Z})$ is a total expression.
3. C is a finite set of total finite domain constraints, intended to be interpreted as conjunction, and possibly including occurrences of defined function symbols.

EXAMPLE 1. *In the following $CFLP(\mathcal{FD})$-program we use the list constructors ([ ] denotes the empty list and $[X|Xs]$ denotes a non-empty list consisting of a first element X and a tail Xs), the integer primitive elements (0,1,2,3,4,...), the addition primitive function (+) over $\mathbb{Z}$, the primitive constraint $domain$ introduced in the previous subsection, a function $from$ to define an infinite list starting at a particular value, and a function $check$ that constraints the first element of a list and returns different values depending on the integer interval.*

$$
\begin{array}{llcccl}
from & N & = & [N \mid from\ (N+1)] & & \\
check & [\,] & = & 0 & & \\
check & [X|\mathrm{Xs}] & = & 1 & \Leftarrow & domain\ [X]\ 1\ 2 \\
check & [X|\mathrm{Xs}] & = & 2 & \Leftarrow & domain\ [X]\ 3\ 4 \\
check & [X|\mathrm{Xs}] & = & 4 & \Leftarrow & domain\ [X]\ 5\ 7
\end{array}
$$

The class of programs used in this work to describe our computation strategy is defined as a proper subclass of the $CFLP(\mathcal{FD})$-programs whose defining rules can be organized in a hierarchical structure called *definitional tree* [1]. More precisely, we choose to reformulate and extend the notions presented in [2, 3, 25] about *overlapping definitional trees* and *conditional overlapping inductively sequential systems*, including now constrained rules over the finite constraint domain $\mathcal{FD}$.

1. $\mathcal{T}$ is a *constrained Definitional Tree over* $\mathcal{FD}$ ($cDT(\mathcal{FD})$ for short), whose call pattern τ is a linear pattern of the form $f\bar{t}_n$ with $f \in DF^n$ and $t_1, \ldots, t_n \in Pat_{\bot}(\mathbb{Z})$, iff its depth is finite and one of the following cases holds:

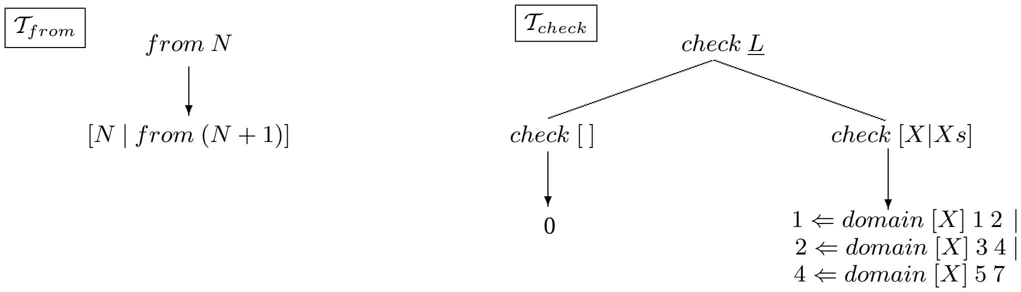

Figure 2. Constrained definitional trees for $from$ and $check$

- $\mathcal{T} \equiv \underline{rule}(\tau = r_1 \Leftarrow C_1|\ldots|r_m \Leftarrow C_m)$, where $\tau = r_i \Leftarrow C_i$ for all $1 \leq i \leq m$ is a fresh variant of a constrained program rule in $\mathcal{P}$.
- $\mathcal{T} \equiv \underline{case}(\tau, X, [\mathcal{T}_1, \ldots, \mathcal{T}_k])$, where X is a *case distinction variable* in τ, $h_1, \ldots, h_k$ $(k > 0)$ are pairwise different passive symbols of $\mathcal{P}$, and for all $1 \leq i \leq k$, $\mathcal{T}_i$ is a $cDT(\mathcal{FD})$ whose call pattern is $\tau\sigma_i$, where $\sigma_i = \{X \mapsto h_i\overline{Y}_{m_i}\}$ with $\overline{Y}_{m_i}$ fresh variables such that $h_i\overline{Y}_{m_i} \in Pat_\perp(\mathbb{Z})$.

A $cDT(\mathcal{FD})$ *of a defined function symbol* $f \in DF^n$ defined by $\mathcal{P}$ is a $cDT(\mathcal{FD})$ $\mathcal{T}$ whose call pattern is $f\overline{X}_n$ with $\overline{X}_n$ new variables. We represent it using the notation $\mathcal{T}_f$.

2. A $CFLP(\mathcal{FD})$-program $\mathcal{P}$ is a *Constrained Overlapping Inductively Sequential System over* $\mathcal{FD}$ (shortly, $COISS(\mathcal{FD})$) iff for each function $f \in DF^n$ defined by $\mathcal{P}$ a $cDT(\mathcal{FD})$ $\mathcal{T}_f$ of f exists such that the collection of all the program rules $\tau = r_i \Leftarrow C_i$ $(1 \leq i \leq m)$ obtained from the different nodes $\underline{rule}(\tau = r_1 \Leftarrow C_1|\ldots|r_m \Leftarrow C_m)$ occurring in $\mathcal{T}_f$ equals, up to variants, the collection of all the constrained program rules in $\mathcal{P}$ whose left hand side has the root symbol f.

As a concrete example, we consider the $CFLP(\mathcal{FD})$-program given in *Example 1*. From the definitional trees illustrated by the pictures given in *Figure 2*, it is easy to check that this program is a $COISS(\mathcal{FD})$. For instance, the defined function symbols $from$ and $check$ have the following constrained definitional trees $\mathcal{T}_{from}$ and $\mathcal{T}_{check}$, respectively:

$$\mathcal{T}_{from} \equiv \underline{rule}\ (from\ N\ =\ [N \mid from\ (N+1)])$$

$$\begin{array}{l} \mathcal{T}_{check} \equiv \underline{case}\ (check\ L, L, [\\ \qquad \underline{rule}\ (check\ [\,]\ =\ 0), \\ \qquad \underline{rule}\ (check\ [X|Xs]\ =\ 1 \Leftarrow domain\ [X]\ 1\ 2\ | \\ \qquad\qquad\qquad\qquad\qquad\quad 2 \Leftarrow domain\ [X]\ 3\ 4\ | \\ \qquad\qquad\qquad\qquad\qquad\quad 4 \Leftarrow domain\ [X]\ 5\ 7\ )\ ]\ ) \end{array}$$

A generic *goal* for a given $COISS(\mathcal{D})$-program must have the form $G \equiv \exists\overline{U}.\ P \,\square\, C \,\square\, S \,\square\, \sigma$, where the symbol $\square$ must be interpreted as conjunction, and:

- $\overline{U}$ is the set of so-called *existential variables* of the goal G. These are intermediate variables used in the computation, whose bindings may be partial patterns.
- $P \equiv e_1 \to R_1, \ldots, e_n \to R_n$ is a finite conjunction of so-called *productions* where each R_i is a distinct variable and e_i is an expression or a pair of the form $< \tau, \mathcal{T} >$, where τ is an instance of the pattern in the root of an $cDT(\mathcal{FD})$ $\mathcal{T}$. Those productions $e \to R$ whose left hand side e is simply an expression are called *suspensions*, while those whose left hand side is of the form $< \tau, \mathcal{T} >$ are called *demanded productions*. The set of *produced variables* of G is defined as $pvar(P) =_{def} \{R_1, \ldots, R_n\}$.
- $C \equiv \delta_1, \ldots, \delta_k$ is a finite conjunction of total finite domain constraints (possibly including occurrences of defined function symbols).
- $S \equiv \pi_1, \ldots, \pi_l$ is a finite conjunction of total finite domain primitive constraints, called *constraint store*.
- σ is an idempotent substitution called *answer substitution* such that $dom(\sigma) \cap var(P \,\square\, C \,\square\, S) = \emptyset$.

An *initial goal* can be any admissible goal, but for practical use in programming, P and S are usually empty and σ is the identity substitution ε.

Similarly to [20, 25], we use a notion of *demanded variable* to deal with lazy evaluation, but now in this work w.r.t. a constraint store, higher-order variables and definitional trees. We say that $X \in var(G)$ is a *demanded variable* in a goal G iff one of the following cases holds:

1. X is *demanded by the constraint store* S of G, i.e. $\mu(X) \neq \perp$ holds for every possible solution substitution μ of S (see [20] for more details and for practical use).
2. X is *demanded by a suspension* $(X\overline{a}_k \to R) \in P$ such that $k > 0$ and R is a demanded variable in G.
3. X is *demanded by a production with definitional tree* $(< e, \underline{case}(\tau, Y, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R) \in P$ such that X occurs in e at the same position that the case-distinction variable Y in τ.

The distinction between the two possible kinds of productions is useful in order to define our computation strategy for solving goals and to efficiently control the computation:

- *Demanded productions* $< \tau, \mathcal{T} > \to R$ are used to compute a value for the demanded variable R. This value will be shared by all occurrences of R in the goal. Note that τ is always an instance of the call pattern in the root of the tree $\mathcal{T}$.
- *Suspensions* $e \to R$ eventually become demanded productions if R becomes demanded in the computation, or else disappear if R becomes absent from the rest of the goal.

Finally, an *answer* for a goal G and a given $COISS(\mathcal{FD})$-program $\mathcal{P}$, must have the form $\Pi \,\square\, \theta$, where Π is a finite conjunction of total primitive constraints and θ is an idempotent substitution such that $dom(\theta) \cap var(\Pi) = \emptyset$.

Additional results relating all of these notions with respect to a suitable declarative semantic are given in [19, 20] by means of a *Constraint ReWriting Logic* $CRWL(\mathcal{D})$, which can be also directly instantiates with our finite domain $\mathcal{FD}$.

3. Goal Solving over $\mathcal{FD}$ using Definitional Trees

Our computation strategy with definitional trees over $\mathcal{FD}$ is presented as an optimization of the goal solving calculus $CLNC(\mathcal{D})$ introduced in [20], which consists of a set of transformation rules for goals, where each transformation takes the form $G \Vdash G'$, specifying one of the possible ways of performing one step of goal solving. Then, derivations are sequences of $\Vdash$-steps, and as in the case of constrained SLD derivations for $CLP(\mathcal{D})$ programs [14], successful derivations will eventually end with a *solved goal* (with P and C empty) identifying a *computed answer* $S \ \square\ \sigma$. Failing derivations (ending with an obviously inconsistent goal $\blacksquare$) and infinite derivations are also possible.

Similarly to other narrowing strategies [20, 25] and from a theoretical viewpoint, all the goal transformations are applied by viewing P and C as sets, rather than sequences (of course, our concrete $\mathcal{TOY}(\mathcal{FD})$ implementation given in *Section 4*, which is based on backtracking and compilation to Prolog, considers sequences in a particular order). The transformations concerning suspensions are designed with the aim of modelling the behavior of constrained lazy narrowing with *sharing* as in the $CLNC(\mathcal{D})$ calculus, but now using more simple productions of the form $e \rightarrow R$ (with only a variable R in the right hand side) involving primitive functions, possibly higher-order defined functions and functional variables over $\mathcal{FD}$. The main novelty w.r.t. the previous calculus is now the following: if e has a defined function symbol in the root and R is a demanded variable, we can awake the suspension decorating e with an appropriate $cDT(\mathcal{FD})$ $\mathcal{T}$ and introducing a new demanded production $< e, \mathcal{T} > \rightarrow R$ in the goal in order to perform a more convenient and efficient narrowing strategy by means of the definitional tree $\mathcal{T}$ (instead of perform directly a non-deterministic and inefficient rewriting step over e, as it occurs with the goal transformation rule **Defined Function** in the $CLNC(\mathcal{D})$ calculus).

The goal transformation rules corresponding to demanded productions $< e, \mathcal{T} > \rightarrow R$ (we note that the variable R is always demanded and therefore needed in the computation) are used to encode the efficient *needed narrowing strategy* over the expression e guided by the definitional tree $\mathcal{T}$, in a vein similar to [12, 25]:

- If $\mathcal{T}$ is a $\underline{rule}$ tree, then we can choose one of the available program rules for rewriting e, introducing appropriate suspensions and the finite domain constraints associated to the rule in the new goal so that lazy evaluation is ensured.

- If $\mathcal{T}$ is a $\underline{case}$ tree, one of the following distinct transformation can be applied, according to the kind of symbol occurring in e at the case-distinction position: if this symbol is a passive symbol h_i, then we can select the appropriate subtree $\mathcal{T}_i$ associated to h_i (if possible; otherwise we fail). If the symbol is a non-produced variable Y, then we can select a subtree $\mathcal{T}_i$, generating an appropriate binding $\{Y \mapsto h_i\overline{Y}_{m_i}\}$ with $\overline{Y}_{m_i}$ fresh variables such that $h_i\overline{Y}_{m_i} \in Pat_\perp(\mathbb{Z})$. Finally, if the symbol is a (non-passive) primitive or defined function symbol, we introduce a new demanded suspension in the goal, in order to evaluate this active argument. In any other case, selection of the subgoal must be delayed until a further stage of the computation.

Finally, we can use the same goal transformation rules concerning constraints presented in the $CLNC(\mathcal{D})$ calculus (see [20]), which are designed to combine (primitive or user defined) constraints with the action of a constraint solver.

The following example of goal solving is intended to illustrate the useful combination of lazy functions with constraint solving over finite domains, exploiting lazy evaluation over infinite data structures. At each goal transformation step, we underline which subgoal is selected.

EXAMPLE 2. *We compute all the answers from the user defined constraint* $check\ (from\ M) < 3$ *using the* $COISS(\mathcal{FD})$ *program given in* Example 1 *and the definitional trees given in* Figure 2. *We use suspensions to achieve the effect of a* demand-driven *evaluation with infinite lists, and we use demanded productions for ensuring the efficient choice of demand/needed redexes. Definitional trees* $\mathcal{T}_{from}$ *and* $\mathcal{T}_{check}$ *are used to guide and avoid* don't know *choices of program rules and failure computations.*

(1) $\square\ \underline{check\ (from\ M) <\ 3}\ \square\ \square\ \varepsilon \Vdash$
(we evaluate the non-primitive arguments of the $\mathcal{FD}$ constraint)
(2) $\exists R.\ \underline{check\ (from\ M) \rightarrow R}\ \square\ \square\ R < 3\ \square\ \varepsilon \Vdash$
(R is a demanded variable by the constraint store)
(3) $\exists R.\ \underline{< check\ (from\ M), \mathcal{T}_{check} > \rightarrow R}\ \square\ \square\ R < 3\ \square\ \varepsilon \Vdash$
(defined function '$from$' in the case-distinction position)
(4) $\exists R', R.\ \underline{< from\ M, \mathcal{T}_{from} > \rightarrow R'}, < check\ R', \mathcal{T}_{check} > \rightarrow R\ \square\ \square\ R < 3\ \square\ \varepsilon \Vdash$ ('$from$' rule application with R' demanded)
(5) $\exists R', R.\ \underline{[M|from\ (M+1)] \rightarrow R'}, < check\ R', \mathcal{T}_{check} > \rightarrow R\ \square\ \square\ R < 3\ \square\ \varepsilon \Vdash_{\{R' \mapsto [M|As]\}}$
(syntactic unification by imitation in the narrowing process)
(6) $\exists As, R.\ from\ (M+1) \rightarrow As, \underline{< check\ [M|As], \mathcal{T}_{check} > \rightarrow R}\ \square\ \square\ R < 3\ \square\ \varepsilon \Vdash$

At this point, As *is not a demanded variable and we have three possible alternatives according to the application of the program rules defining* $check$ *in the* rule *subtree. We note that the first rule of* $check$ *given in* Example 1 *cannot be applied because the argument is evaluated to a non-empty list.*

(7) $\exists As, R.\ from\ (M+1) \rightarrow As, \underline{1 \rightarrow R}\ \square\ \square\ domain\ [M]\ 1\ 2, R < 3\ \square\ \varepsilon \Vdash_{\{\mathbf{R \mapsto 1}\}}$
(we obtain a binding for the first argument of our constraint)
(8) $\exists As.\ from\ (M+1) \rightarrow As\ \square\ \square\ domain\ [M]\ 1\ 2, \underline{1 < 3}\ \square\ \varepsilon \Vdash$
(constraint satisfaction in the constraint store)
(9) $\exists As.\ \underline{from\ (M+1) \rightarrow As}\ \square\ \square\ domain\ [M]\ 1\ 2\ \square\ \varepsilon \Vdash$
(As is unneeded in the computation: lazy evaluation of an infinite list!)
(10) $\square\ \square\ \underline{domain\ [M]\ 1\ 2}\ \square\ \varepsilon \Vdash$
(finally, the constraint solver shows the first computed answer)
computed answer: $M\ in\ 1 \ldots 2$

By applying the third '$check$*' program rule and repeating the same steps (7)–(10), we can obtain the second computed answer. We have only to check the satisfiability of the new constraint '*$domain$*' and we don't have to rebuild again the derivation.*

(11) $\exists As, R.\ from\ (M+1) \rightarrow As, \underline{2 \rightarrow R}\ \square\ \square\ domain\ [M]\ 3\ 4, R < 3\ \square\ \varepsilon \Vdash_{\{\mathbf{R \mapsto 2}\}}$
(12) $\exists As.\ from\ (M+1) \rightarrow As\ \square\ \square\ domain\ [M]\ 3\ 4, \underline{2 < 3}\ \square\ \varepsilon \Vdash^*$
(13) $\square\ \square\ \underline{domain\ [M]\ 3\ 4}\ \square\ \varepsilon \Vdash$ **computed answer**: $M\ in\ 3 \ldots 4$

Finally, by using the fourth '$check$*' program rule, we obtain a failure in the constraint solving process and no more answers can be obtained.*

(14) $\exists As, R.\ from\ (M+1) \rightarrow As, \underline{4 \rightarrow R}\ \square\ \square\ domain\ [M]\ 5\ 7, R < 3\ \square\ \varepsilon \Vdash_{\{\mathbf{R \mapsto 4}\}}$
(15) $\exists As.\ from\ (M+1) \rightarrow As\ \square\ \square\ domain\ [M]\ 5\ 7, \underline{4 < 3}\ \square\ \varepsilon \Vdash$
$\blacksquare$ (failure detection in $\mathcal{FD}$ constraint solving)

The main properties of our computation strategy with definitional trees, *soundness* and *completeness* with respect to $CFLP(\mathcal{FD})$ semantics, can be obtained by using techniques similar to those used for the $CLNC(\mathcal{D})$ calculus in [20] (with generic constraints but no definitional trees) and the DNC calculus in [25] (with definitional trees but no generic constraints), and can be found in [26].

From the viewpoint of efficiency, definitional trees in demanded productions are used for ensuring only needed narrowing steps in the line of [4, 2, 25]. Then, computations in $CDNC(\mathcal{FD})$ are in essence needed narrowing derivations modulo non-deterministic choices between overlapping and constrained program rules over $\mathcal{FD}$. Therefore, our efficient mechanism maintains the optimality properties shown in [4, 2, 25] guiding (and avoiding) *don't know* choices of constrained program rules by means of definitional trees.

4. Example of Our Computation Strategy in the $\mathcal{TOY}(\mathcal{FD})$ System

$\mathcal{TOY}(\mathcal{FD})$ [9] is a $CFLP(\mathcal{FD})$ implementation that extends the $\mathcal{TOY}$ system [17] to deal with $\mathcal{FD}$ constraints for solving constraint satisfaction problems and typical combinatorial problems [8]. $\mathcal{TOY}(\mathcal{FD})$ integrates, as a host language, the higher-order lazy functional logic language $\mathcal{TOY}$ and, as a constraint solver, the efficient $\mathcal{FD}$ constraint solver of SICStus. Therefore, $\mathcal{TOY}(\mathcal{FD})$ programs are essentially $\mathcal{TOY}$ programs, where $\mathcal{FD}$ constraints are defined as functions that are solved by the $\mathcal{FD}$ constraint solver connected to $\mathcal{TOY}$. Moreover, $\mathcal{TOY}(\mathcal{FD})$ is implemented on top on SICStus Prolog.

In this section, we show how the execution in $\mathcal{TOY}(\mathcal{FD})$ of the goal introduced in *Example 2* matches our computation strategy with definitional trees for goal solving described in the previous section. More precisely, we show how our computation strategy with definitional trees matches with the trace in the system $\mathcal{TOY}(\mathcal{FD})$ corresponding to the goal $check\ (from\ M) < 3$. For this purpose we show each step of the previous constrained narrowing derivation with its corresponding exact code lines in the trace.

Each step of the trace contains the name of the active module in each moment. We briefly describe the modules that appear in the trace steps.

- **initToy:** contains the interface with the user and recognizes at the prompt level, goals, commands and expressions to evaluate. It includes the lexical and syntactic analysis.
- **primitivCod:** contains the original set of primitives of $\mathcal{TOY}$.
- **plgenerated:** the compilation of a $\mathcal{TOY}(\mathcal{FD})$ program generates a SICStus Prolog program. This program is defined in the `plgenerated` module, which contains predefined definitions for types and functions. The module is created in the translation process. After the compilation, the Prolog file generated can be loaded and the user is ready to execute goals.
- **toycomm:** This module is necessary for executing programs in $\mathcal{TOY}(\mathcal{FD})$. The system loads it automatically at the beginning. It contains all the common predicates to all the $\mathcal{TOY}$ programs.

We begin the computation

(1) $\square\ \underline{check\ (from\ M) <\ 3}\ \square\ \square\ \varepsilon \Vdash$

Initially $\mathcal{TOY}(\mathcal{FD})$ does a lexical and syntactical analysis of the goal. If this process is correct, then types are checked. If the types are correct, then the goal is translated into Prolog code. Each constraint that is part of the goal is translated and sent to the $\mathcal{FD}$ constraint solver of SICStus.

The following trace step is analogous to the step (1) in our narrowing derivation.

Call: initToy:**$<**('$$susp'('$**check**',['$$susp'('$**from**',[_**23825**],
_24738,_24736)],_24561,_24559),**3**,true,[],_20384) ?

This trace step is the `Call` to the inequality predicate defined bellow. The inequality predicate has a prefix symbol `'$'` in order to avoid name clashes, e.g. The symbol `'$'` prefixes $<$ obtaining `$<` in order to distinguish between our primitive operator and the Prolog inequality predicate. The definition of the $\mathcal{TOY}(\mathcal{FD})$ inequality predicate contained in the module `primitivCod` is the following Prolog clause

```
$<(X,Y,H,Cin,Cout):-
    hnf(X,HX,Cin,Cout1),
    hnf(Y,HY,Cout1,Cout),
    (number(HX),number(HY) ->
        (HX<HY,H=true;HX>=HY,H=false);errPrim).
```

Before the invocation of the $\mathcal{FD}$ solver by means of `HX<HY`, the operators, $check\ (from\ M)$ and 3, are transformed into head normal form (hnf for short) in an analogous way to the evaluation of the arguments of the $\mathcal{FD}$ constraint in our strategy with definitional trees. As 3 is a number, then it is already in hnf. Since $check\ (from\ M)$ is a non-primitive argument, the system computes its hnf.

Once the arguments are in hnf, the inequality predicate verify if both of them are numbers. If so then they are compared by means of the Prolog relational operator ($<$) in `HX<HY`.

(2) $\exists R.\ \underline{check\ (from\ M) \to R}\ \square\ \square\ R < 3\ \square\ \varepsilon \Vdash$

This step of our constrained narrowing derivation evaluates the non-primitive argument of the $\mathcal{FD}$ constraint. It is similar to compute the hnf of $check\ (from\ M)$.

The next trace step is the `Call` to the predicate that computes the hnf of $check\ (from\ M)$

Call: primitivCod:**hnf**('$$susp'('$check',['$$susp'
('$from',[_23825],_24738,_24736)],_24561,_24559),
_28441,[ ],_28443) ?

Following our narrowing derivation, $check\ (from\ M)$ has a defined function symbol in the root and R is a demanded variable. Then we can awake the suspension $check\ (from\ M)$ and introducing a new demanded production $< check\ (from\ M), \mathcal{T}_{check} > \to R$

(3) $\exists R.\ \underline{< check\ (from\ M), \mathcal{T}_{check} > \to R}\ \square\ \square\ R < 3\ \square\ \varepsilon \Vdash$

The $\mathcal{TOY}(\mathcal{FD})$ system awake the suspension by means of the `hnf_susp` predicate. It is contained in the body of the `hnf` predicate. We briefly describe the behavior of the predicate `hnf`: if the argument is a variable or an expression with a passive symbol in the root, then it is a hnf. If the expression is a defined function, then it can appear in *suspended form*. In this case, `hnf` checks that the expression has not been evaluated (otherwise, it would be returned). Then, it calls the predicate `hnf_susp`.

Call: toycomm:**hnf_susp**('$check',['$$susp'
('$from',[_23825],_24738,_24736)],_24561,[ ],_28443) ?

The definition of `hnf_susp` for the `$check` function is the following

```
hnf_susp('$check', '.'(_A, [ ]), _B, _C, _D) :-
    '$check'(_A, _B, _C, _D).
```

And the corresponding call is

Call: plgenerated:'$check'('$$susp'('$from'
,[_23825],_24738,_24736),_24561,[ ],_28443) ?

The definition of the `$check` predicate is

'$check'(_A, _B, _C, _D):- hnf(_A, _E, _C, _F),
'$check_1'(_E, _B, _F, _D).

The `$check` predicate first calculates the `hnf` of its argument $from\ M$

Call: plgenerated:hnf('$$susp'(**'$from'**,**[_23825]**,
_24738,_24736),_34510,[ ],_34512) ?

In our strategy, $\mathcal{T}_{check}$ is a <u>*case*</u> tree and the symbol occurring in $check\ (from\ M)$ at the case-distinction position is a defined function symbol. Next we introduce a new demanded suspension in the goal, in order to evaluate this active argument

(4) $\exists R', R.\ \underline{< from\ M, \mathcal{T}_{from} > \rightarrow R'}, < check\ R', \mathcal{T}_{check} > \rightarrow R\ \Box$
$\Box\ R < 3\ \Box\ \varepsilon \Vdash$

Returning to the trace, If the argument ($from\ M$) to evaluate is neither a variable nor an expression with a passive symbol in the root, it has to call again the predicate `hnf_susp`

Call: toycomm:**hnf_susp**('**$from**',[**_23825**],_24738, [ ],_34512) ?

The function `hnf_susp` for `$from` is defined as

hnf_susp('$from', '.'(_A, [ ]), _B, _C, _D) :-
'$from'(_A, _B, _C, _D).

And the corresponding call is

Call: plgenerated:'$from' (_23825,_24738,[ ],_34512) ?

The function `$from` is defined as

'$from'(_A, :(_A, '$$susp'('$from',['$$susp'
($+, [_A, 1], _B, _C)], _D, _E)), _F, _F).

Since this is a Prolog fact, the hnf computation of `$from` is finished.

(5) $\exists R', R.\ \underline{[M|from\ (M+1)] \rightarrow R'}, < check\ R', \mathcal{T}_{check} > \rightarrow R\ \Box$
$\Box\ R < 3\ \Box\ \varepsilon \Vdash_{\{R' \mapsto [M|As]\}}$

Our strategy achieve syntactic unification in the narrowing process. In the trace we show the `Exit` of the `hnf` predicate with a bold part that correspond with $[M|from\ (M+1)]$

Exit: plgenerated:hnf('$$susp'('$from',[_23825],_23825:
'$$susp'('$from',['$$susp'($+,[_23825,1],_40354,_40355)],
_40347,_40348),hnf),**_23825:**'$$susp'('**$from**',['$$susp'
(**$+**,[**_23825**,**1**],_40354,_40355)],_40347,_40348),[ ],[ ]) ?

When the computation of the hnf of the argument `$from` of the `$check` predicate is concluded, a case-distinction by means of the predicate `$check_1` arises according to the *check* function defined in *Example 1*.

(6) $\exists As, R.\ from\ (M+1) \rightarrow As, \underline{< check\ [M|As], \mathcal{T}_{check} > \rightarrow R}\ \Box$
$\Box\ R < 3\ \Box\ \varepsilon \Vdash$

We have four possible alternatives according to the application of the program rules defining $check$ in the <u>*rule*</u> subtree.

'$check_1'(_A, **0**, _B, _C):-
unifyHnfs(_A, **[ ]**, _B, _C).
'$check_1'(_A, **1**, _B, _C):-
unifyHnfs(_A, **:(_D, _E)**, _B, _F),
'$domain'(:(_D, [ ]), 1, 2, true, _F, _C).
'$check_1'(_A, **2**, _B, _C):-
unifyHnfs(_A, **:(_D, _E)**, _B, _F),
'$domain'(:(_D, [ ]), 3, 4, true, _F, _C).
'$check_1'(_A, **4**, _B, _C):-
unifyHnfs(_A, **:(_D, _E)**, _B, _F),
'$domain'(:(_D, [ ]), 5, 7, true, _F, _C).

The call to the predicate `$check_1` is

Call: plgenerated:'**$check_1**'(**_23825**:'$$susp'(
'$from',['$$susp'($+,[_23825,1],_40354,_40355)],
_40347,_40348),_24561,[ ],_28443) ?

It begins trying to unify its arguments with the empty list in the first clause.

Call: plgenerated:unifyHnfs(_23825:'$$susp'
('$from',['$$susp'($+,[_23825,1],_40354,
_40355)],_40347,_40348),**[ ]**,[ ],_28443) ?

Since is fails, it continues with the next clause

Call: plgenerated:unifyHnfs(_23825:'$$susp'(
'$from',['$$susp'($+,[_23825,1],_40354,_40355)],
_40347,_40348),**_42798:_42799**,[ ],_42804) ?

As this one is successful, it continues with `$domain'`

Call: plgenerated:
'**$domain**'(_23825:[ ],1,2,true, [ ],_28443) ?

The execution of `$domain` calculates the hnf of its first three arguments (list $[M]$ and numbers 1 and 2 in *Example 2*). Later, the Prolog `domain` constraint is executed.

Call: plgenerated:domain([_23825],1,2) ?

The `call` to `domain` returns success and, consequently, the `$check_1` function also succeeds and returns 1 as the hnf of the first argument, (i.e. $check\ (from\ M)$).

(7) $\exists As, R.\ from\ (M+1) \rightarrow As, \underline{1 \rightarrow R}\ \Box\ \Box\ domain\ [M]\ 1\ 2,$
$R < 3\ \Box\ \varepsilon \Vdash_{\{\mathbf{R} \mapsto \mathbf{1}\}}$

We obtain a binding for the first argument of our constraint; in the trace we put the number 1 in bold

Exit:primitivCod:hnf('$$susp'('$check',['$$susp'('$from'
,[_58819],_58819:'$$susp'('$from',['$$susp'($+,[_58819|...]
,_40354,_40355)],_40347,_40348),hnf)],1,hnf), **1** ,[ ],[ ]) ?

(8) $\exists As.\ from\ (M+1) \rightarrow As\ \Box\ \Box\ domain\ [M]\ 1\ 2,\ \underline{1 < 3}\ \Box\ \varepsilon \Vdash$

Once the calculation of the hnfs in the inequality predicate (`$<`) is finished, where these hnfs are primitive elements (1 and 3 respectively), the Prolog predicate $1 < 3$ is evaluated to `true`, and then our inequality (`$<`) becomes `true` (as it also occurs in the step (8) of the narrowing derivation). In the trace this step is

Exit: 1<3 ?

(9) $\exists As.\ \underline{from\ (M+1) \rightarrow As}\ \Box\ \Box\ domain\ [M]\ 1\ 2\ \Box\ \varepsilon \Vdash$

As is absent from the rest of the goal and therefore this production disappears

(10) $\Box\ \Box\ \underline{domain\ [M]\ 1\ 2}\ \Box\ \varepsilon \Vdash$

Afterwards, the first answer of the $\mathcal{FD}$ constraint solver of SICStus is picked up and treated by $\mathcal{TOY}(\mathcal{FD})$ in order to show it:

M in 1..2

Now, the system backtracks and searches alternative answers in an analogous way to the steps (7)-(10) in the narrowing derivation.

Redo: initToy: $<('$$susp'('$check',['$$susp'('$from',[_58819]
,_58819:'$$susp'('$from',['$$susp'($+,[_58819|...],
_40354,_40355)],_40347,_40348),hnf)],1,hnf),3,true,[ ],[ ]) ?

Following this process, the hnfs are now 2 and 3 respectively, and, as above, the Prolog predicate $2 < 3$ is evaluated to `true`. The inequality (`$<`) becomes also `true`, as it occurs in the step (11)-(13). Therefore, the second answer is shown.

M in 3..4

The system does backtracking again and searches more alternative answers (analogous to the derivation step (7) in *Example 2*).

Redo: initToy:$<('$$susp'('$check',['$$susp'('$from'
,[_58819],_58819:'$$susp'('$from',['$$susp'($+,[_58819|...]
,_40354,_40355)],_40347,_40348),hnf)],2,hnf),3,true,[ ],[ ])?

Finally, the hnfs are now 4 and 3. Since 4 is not less than 3, more hnfs of the expression are searched.

Redo:primitivCod:hnf('$$susp'('$check',['$$susp'
('$from',[_58819],_58819:'$$susp'('$from',['$$susp'
($+,[_58819|...],_40354,_40355)],
_40347,_40348),hnf)],4,hnf),4,[ ],[ ]) ?

No more hnfs can be found. Therefore, no more alternatives are searched, finishing the trace of the goal (analogously to the derivation step (15) in the narrowing derivation).

Fail: initToy:$<('$$susp'('$check',['$$susp'('$from'
,[_23825],_24738,_24736)],_24561,_24559),3,true,[ ],_20384)?

5. Conclusions and Future Work

In this paper, we have presented a functional logic programming approach to finite domain ($\mathcal{FD}$) constraint solving by means of a particular instance over $\mathcal{FD}$ of the generic scheme $CFLP(\mathcal{D})$ [19], giving rise to a suitable $CFLP(\mathcal{FD})$ language for Constraint Functional Logic Programming over Finite Domains. Taking this language as the basis of our work, we have sketch an effective computational strategy for the integration of constraint goal solving for $CFLP(\mathcal{FD})$-programs by means of an optimization of the generic *Constrained Lazy Narrowing Calculus* $CLNC(\mathcal{D})$ [20] over $\mathcal{FD}$, using definitional trees to guide the choice of demand/needed redexes. Moreover, we have described how this strategy can be integrated in the $CFLP(\mathcal{FD})$ system $\mathcal{TOY}(\mathcal{FD})$ [9] by means of an example that combines lazy functions with constraint solving over finite domains using the efficient $\mathcal{FD}$ constraint solver of SICStus Prolog and exploiting lazy evaluation over infinite data structures.

In the near future, we plan to investigate both improvements and applications of the $CFLP(\mathcal{FD})$ language. Since $CFLP(\mathcal{FD})$ assumes only free data constructors, planned improvements include enriching the language with algebraic data constructors in the vein of [5].

Planned future work will include further theoretical investigation about optimality results for our computation strategy extending the good properties known for needed narrowing [4, 2] in functional logic programming, a formal comparison between [20, 25] and our new framework over $\mathcal{FD}$, and a way to quantify the efficiency improvements of our $\mathcal{TOY}(\mathcal{FD})$ implementation by using definitional trees with respect to $Curry$ [11] and other $CFLP(\mathcal{FD})$ or $CLP(\mathcal{FD})$ implementations.

Last but not least, we are working on declarative debugging techniques for $CFLP(\mathcal{FD})$ programs in $\mathcal{TOY}(\mathcal{FD})$, following previous work for FLP in $\mathcal{TOY}$ [6, 7] as well as related work for $CLP(\mathcal{D})$-programs in [24]. The *Constraint ReWriting Logic* $CRWL(\mathcal{FD})$ already provides a formal framework for the declarative debugging of *wrong answers*. We are designing an extension of $CRWL(\mathcal{FD})$ which will serve as a formal framework for the declarative debugging of *missing answers*. As a byproduct of this research, we expect to obtain a formal characterization of *finite failure* in $CFLP(\mathcal{FD})$ programming, generalizing some of the already known results on the finite failure semantics of functional logic programs with disequality constraints [18].

Acknowledgments

The authors are thankful to Teresa Hortalá González, Mario Rodríguez Artalejo and Fernando Sáenz Pérez for their collaboration, comments and contributions during the development of this work and for the help in preparing the final version of this paper.

References

[1] S. Antoy. *Definitional trees*. In Proc. Int. Conf. on Algebraic and Logic Programming (ALP'92), volume 632 of Springer LNCS, pp. 143–157, 1992.

[2] S. Antoy. *Optimal non-deterministic functional logic computations*. In Proc. of ALP'97, pages 16-30. Springer LNCS 1298, 1997.

[3] S. Antoy. *Constructor-based conditional narrowing*. In Proc. PPDP'01, ACM Press, pp. 199-206, 2001.

[4] S. Antoy, R. Echahed, M. Hanus. *A needed narrowing strategy*. Journal of the ACM, 47(4): 776-822, 2000.

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

[6] R. Caballero and M. Rodríguez-Artalejo. *A Declarative Debugging System for Lazy Functional Logic Programs.* Electronic Notes in Theoretical Computer Science 64, 63 pages, 2002.

[7] R. Caballero and M. Rodríguez-Artalejo. $\mathcal{DDT}$: *A Declarative Debugging Tool for Functional Logic Languages.* Proc. of the 7th International Symposium on Functional and Logic Programming (FLOPS'04), volume 2998 of Springer LNCS, pp. 70–84, 2004.

[8] A.J. Fernández, M.T. Hortalá-González and F. Sáenz Pérez. *Solving Combinatorial Problems with a Constraint Functional Logic Language.* Proc. of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL'2003), Springer LNCS 2562, pp. 320–338, 2003.

[9] A.J. Fernández, M.T. Hortalá-González and F. Sáenz Pérez. $\mathcal{TOY}(\mathcal{FD})$. *User's Manual*, October 27, 2003. System available at `http://www.lcc.uma.es/~afdez/cflpfd/`.

[10] M. Hanus. *The Integration of Functions into Logic Programming: From Theory to Practice.* Journal of Logic Programming 19&20, pp. 583–628, 1994.

[11] M. Hanus (ed.), *Curry: an Integrated Functional Logic Language*, Version 0.8, April 15, 2003. System available at `http://www-i2.informatik.uni-kiel.de/~curry/`.

[12] M. Hanus, C. Prehofer. *Higher-Order Narrowing with Definitional Trees.* Journal of Functional Programming, 9(1):33-75, 1999.

[13] J. Jaffar and M.J. Maher. *Constraint Logic Programming: A Survey*. The Journal of Logic Programming 19&20, pp. 503–581, 1994.

[14] J. Jaffar, M.J. Maher, K. Marriott and P.J. Stuckey. *The Semantics of Constraint Logic Programs.* Journal of Logic Programming, 37 (1-3) pp. 1–46, 1998.

[15] R. Loogen, F.J. López-Fraguas, M. Rodríguez-Artalejo. *A demand driven computation strategy for lazy narrowing.* In Proc. Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'93), volume 714 of Springer LNCS pp. 184–200, 1993.

[16] F.J. López-Fraguas. *A General Scheme for Constraint Functional Logic Programming.* In Proc. Int. Conf. on Algebraic and Logic Programming (ALP'92), Springer LNCS 632, pp. 213–227, 1992.

[17] F.J. López-Fraguas, J. Sánchez-Hernández. $\mathcal{TOY}$*: A Multiparadigm Declarative System.* Proc. RTA'99, Springer LNCS 1631, pp 244–247, 1999. System available at `http://toy.sourceforge.net`.

[18] F.J. López-Fraguas, J. Sánchez-Hernández. *A Proof Theoretic Approach to Failure in Functional Logic Programming.* Theory and Practice of Logic Programming 4(1), pp. 41–74, 2004.

[19] F.J. López-Fraguas, M. Rodríguez-Artalejo and R. del Vado-Vírseda. *Constraint Functional Logic Programming Revisited.* WRLA'2004, Elsevier ENTCS series, vol 117, pp. 5–50, 2005.

[20] F.J. López-Fraguas, M. Rodríguez-Artalejo and R. del Vado-Vírseda. *A lazy narrowing calculus for declarative constraint programming.* 6th International Conference on PPDP'04, ACM Press, pp. 43–54, 2004.

[21] M. Marin, T. Ida and W. Schreiner. *CFLP: a Mathematica Implementation of a Distributed Constraint Solving System.* In Third International Mathematical Symposium (IMS'99), Hagenberg, Austria, August 23–25, 10 pages, 1999.

[22] M. Marin, T. Ida and T. Suzuki. *Cooperative Constraint Functional Logic Programming.* In IPSE'2000, pp. 223–230, Nov. 1–2, 2000.

[23] K. Marriott and P.J. Stuckey. *Programming with Constraints, An Introduction.* The MIT Press, 1998.

[24] A. Tessier and G. Ferrand. *Declarative Diagnosis in the CLP Scheme.* In P. Deransart, M. Hermenegildo and J. Małuszynski (eds.), *Analysis and Visualization Tools for Constraint Programming*, Chapter 5, pp. 151–174. Springer LNCS 1870, 2000.

[25] R. del Vado-Vírseda. *A Demand-driven Narrowing Calculus with Overlapping Definitional Trees.* 5th International Conference on Principles and Practice of Declarative Programming (PPDP'03), ACM Press, pp. 213–227, 2003.

[26] R. del Vado-Vírseda. *Declarative Constraint Programming with Definitional Trees.* To appear in Proc. of the 5th International Workshop on Frontiers of Combining Systems (FroCoS 2005), Springer LNCS, 2005.

[27] P. Van Hentenryck. *Constraint logic programming.* The Knowledge Engineering Review, Vol. 6:3, pp. 151–194, 1991.