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 Santestebán 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 $CFLP(D)$, proposed in [19] for lazy Constraint Functional Logic Programming over a parametrically given constraint domain $D$. 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 $CLNC(D)$ presented in [20], which has been proved sound and strongly complete w.r.t. a suitable $CFLP(D)$ semantics, and provides a formal foundation for efficient implementations in existing systems such as Curry [11] and TOY [17]. Finally, we describe the execution of an example implemented in the $CFLP(\mathcal{FD})$ system called $T\Omega Y(\mathcal{FD})$ [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).

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 $CFLP(D)$ has been proposed in [19] as a logical and semantic framework for lazy Constraint Functional Logic Programming over a parametrically given constraint domain $D$, which provides a clean and rigorous declarative semantics for $CFLP_D$ languages like $CLP(D)$, but overcoming some limitations of older $CFLP$ schemes [16, 21, 22]. In this setting, $CFLP(D)$-programs are presented as sets of constrained rewrite rules that define the behavior of possible higher-order and/or non-deterministic lazy functions over $D$. The main novelties in [19] were a new formalization of constraint domains for $CFLP$ and a new Constraint ReWrit-ting Logic $CRW_L(D)$ parameterized by a constraint domain $D$, 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 $CLNC(D)$ for solving goals for $CFLP(D)$-programs, which can be proved sound and strongly complete w.r.t. $CRW_L(D)$’s semantics. These properties qualify $CLNC(D)$ as a convenient computation mechanism for declarative constraint programming languages.

However, efficiency is a major concern for the implementation of $CFLP(D)$ 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 TOY [17, 9].

Although Curry and 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\((\mathcal{F}D)\). 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 \(\mathcal{F}D\) which contracts only needed positions and maintains the efficiency properties shown for existing demand/needed narrowing strategies. This convenient strategy is implemented in the system TOY\((\mathcal{F}D)\) [9] that integrates, as a host language, the higher-order lazy functional logic language TOY and, as constraint solver, the efficient \(\mathcal{F}D\) constraint solver of SICStus Prolog.

The organization of this paper is as follows: In section 2 we describe the CFLP\((\mathcal{F}D)\) language as a concrete instance of the generic CFLP\((\mathcal{D})\) scheme presented in [19, 20] over the finite constraint domain \(\mathcal{F}D\). We also define the class of CFLP\((\mathcal{F}D)\)-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 TOY\((\mathcal{F}D)\). Finally, some conclusions and plans for future work are drawn in section 5.

2. The CFLP\((\mathcal{F}D)\) 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 applicable 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 \(\Sigma = \langle \mathcal{D}C, \mathcal{F}S\rangle\), where \(\mathcal{D}C = \bigcup_{n \in \mathbb{N}} \mathcal{D}C^n\) and \(\mathcal{F}S = \bigcup_{n \in \mathbb{N}} \mathcal{F}S^n\) 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 \(\mathcal{F}P^n \subseteq \mathcal{F}S^n\) and user defined functions \(\mathcal{D}E^n = \mathcal{F}S^n \setminus \mathcal{F}P^n\) for each \(n \in \mathbb{N}\). We assume that \(\mathcal{D}C^n\) includes the special symbol \(\bot\), 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 \(\{1, -1, 2, -2, \ldots\}\) mutually disjoint and disjoint from \(\Sigma\). Integer partial expressions \(e \in \mathcal{E}XP(\mathcal{Z})\) have the following syntax:

\[
e ::= \bot \mid a \mid X \mid h \mid (ee_1)
\]

where \(a \in \mathbb{Z}, X \in \mathcal{V}, h \in \mathcal{D}C \cup \mathcal{F}S\). The set of variables occurring in \(e\) is written \(\text{var}(e)\). An expression \(e\) is called linear iff there is no \(X \in \text{var}(e)\) having more than one occurrence in \(e\). Some interesting subsets of \(\mathcal{E}XP(\mathcal{Z})\) are: \(G\mathcal{E}XP(\mathcal{Z})\), the set of the ground expressions \(e\) such that \(\text{var}(e) = \emptyset\) and \(\mathcal{E}XP(\mathcal{Z})\), the set of the total expressions \(e\) with no occurrences of \(\bot\).

Another important subclass of expressions is the set of integer partial patterns \(s, t \in \mathcal{P}at(\mathcal{Z})\), whose syntax is defined as follows:

\[
t ::= \bot \mid u \mid \langle c \mathcal{I}_m \mid \mathcal{I}_m \rangle
\]

where \(u \in \mathbb{Z}, X \in \mathcal{V}, c \in \mathcal{D}C^n, m \leq n, f \in \mathcal{F}S^n, m < n\).

A passive symbol is an integer primitive element \(u \in \mathbb{Z}\) or the root symbol \(h\) of a pattern of the form \(\langle h \mathcal{I}_m \rangle\) where \(h \in \mathcal{D}C \cup \mathcal{F}S\). We define the information ordering \(\subseteq\) as the least partial ordering over \(\mathcal{P}at(\mathcal{Z})\) satisfying the following properties: \(\bot \subseteq t\) for all \(t \in \mathcal{P}at(\mathcal{Z})\), and \((h\mathcal{I}_m) \subseteq (h\mathcal{I}_n)\) whenever these two expressions are patterns and \(t_i \subseteq t'_{i}\) for all \(1 \leq i \leq m\).

As usual, we define integer substitutions \(\sigma \in \mathcal{S}ub(\mathcal{Z})\) as mappings \(\mathcal{V} \rightarrow \mathcal{P}at(\mathcal{Z})\) extended to \(\mathcal{E}XP(\mathcal{Z}) \rightarrow \mathcal{E}XP(\mathcal{Z})\) in the natural way. We write \(\mathcal{E}\) for the identity substitution, \(\mathcal{E}\) instead of \(\sigma(e)\) and \(\mathcal{E}\sigma\) for the composition of \(\sigma\) and \(\mathcal{E}\), such that \(\mathcal{E}\sigma(h) = (\mathcal{E}\sigma)h\) for any \(h \in \mathcal{E}XP(\mathcal{Z})\). We define the domain \(\mathcal{D}\) of \(\sigma\) and the range \(\mathcal{R}\) of \(\sigma\) as a substitution \(\sigma\) in the usual way. Finally, a substitution \(\sigma\) such that \(\mathcal{D}\sigma \cap \mathcal{R}\sigma = \emptyset\) is called idempotent.

2.2 The Constraint Finite Domain \(\mathcal{F}D\)

Adopting the general approach of [19, 20], a constraint finite domain \(\mathcal{F}D\) can be formalized as a structure with carrier set \(\mathcal{D}_2\), consisting of ground patterns built from the symbols in a signature \(\Sigma\) and the set of primitive elements \(\mathcal{E}\). Symbols in \(\Sigma\) 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 \(\bot\).

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

- **Primitive Constraints** have the syntactic form \(p \mathcal{I}_n \rightarrow !t\), with \(p \in \mathcal{P}F^n\) a primitive function symbol and \(t_1, \ldots, t_n, t \in \mathcal{P}at(\mathcal{Z})\) with total.

- **Constraints** have the syntactic form \(p \mathcal{I}_n \rightarrow !t\), with \(p \in \mathcal{P}F^n\), \(e_1, \ldots, e_n \in \mathcal{E}XP(\mathcal{Z})\) and \(t \in \mathcal{P}at(\mathcal{Z})\) total.

As a concrete example, consider the primitive operators \(\otimes^\mathcal{E}\) with \(\otimes \in \{\cdot, -, \cdot, /\}\) and relations \(\otimes^\mathcal{E}\) with \(\otimes \in \{\cdot, \neq, <\leq, >\geq\}\). We further define over \(\mathcal{E}\) in the usual way. Figure 1 shows a minimum set of primitive functions \(p \in \mathcal{P}F^n\) and their declarative interpretation \(p^\mathcal{D} \subseteq \mathcal{D}_2 \times \mathcal{D}_2\) (we use the notation \(p^\mathcal{D}\mathcal{I}_m \rightarrow t\) to indicate that \((\mathcal{T}_m, t) \in p^\mathcal{D}\mathcal{I}_m\) considered in our constraint finite domain \(\mathcal{F}D\). The function \(\mathcal{I}domain\) 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 \(t \sim s\) abbreviates \(seq t s \rightarrow !false\) (the notations \(\sim\) and \(\mathcal{E}\) can be understood as a particular case of the notations \(==\) and \(\bot\) when these are applied to integers and/or variables that may be instantiated to an integer value).

- \(a \leq b\) abbreviates \(\text{leq} a b \rightarrow !true, a > b\) abbreviates \(\text{leq} a b \rightarrow !false\) (and analogously for the other comparison primitives \(<, \leq, >, \geq\)).

- \(a \otimes b \leq c\) abbreviates \(a \otimes b \rightarrow !r, r \times c\).
The class of programs used in this work to describe our computation strategy is defined as a proper subclass of the CFLP(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 FD.

1. T is a constrained Definitional Tree over FD (cDT(FD)) for short, whose call pattern τ is a linear pattern of the form fTn with f ∈ DF0 and t1, . . . , tn ∈ Pat(ℤ), if its depth is finite and one of the following cases holds:

2.3 Programs, Goals and Answers over FD

In the CFLP(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 FD, called program rules. More precisely, a program rule R for f ∈ DF0 has the form R: fTn = r ≡ C (abbreviated as fTn = r if C is empty) and is required to satisfy the three conditions listed below:

1. The left-hand side fTn is a linear expression, and for all 1 ≤ i ≤ n, ti ∈ Pat(ℤ) are total integer patterns.
2. The right-hand side r ∈ Exp(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(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 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.

| Figure 1. Primitive function symbols in FD |
A cDT(\mathcal{F}D) of a defined function symbol \( f \in \mathcal{D}F^n \) defined by \( \mathcal{P} \) is a cDT(\mathcal{F}D) \( T \) whose call pattern is \( fX_n \) with \( X_n \) new variables. We represent it using the notation \( T_f \).

2. A CFLP(\mathcal{F}D)-program \( \mathcal{P} \) is a Constrained Overlapping Inductively Sequential System over \( \mathcal{F}D \) (shortly, \textit{COISS(\mathcal{F}D)}) iff for each function \( f \in \mathcal{D}F^n \) defined by \( \mathcal{P} \) a cDT(\mathcal{F}D) \( T_f \) 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 \text{rule}(\tau = r_i \leftarrow C_i | r_m \leftarrow C_m)\) occurring in \( T_f \) 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{F}D)-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{F}D). For instance, the defined function symbols \textit{from} and \textit{check} have the following constrained definitional trees \( T_{\text{from}} \) and \( T_{\text{check}} \), respectively:

\[
T_{\text{from}} \equiv \text{rule}(\text{from} N = |N| \text{ from } (N + 1))
\]

\[
T_{\text{check}} \equiv \text{case}(\text{check} L, L, [\text{rule}(\text{check} | \text{check}()) = 0], \text{rule}(\text{check} | X|Xs) = 1 \in \text{domain} |X| 1 2 | 2 \in \text{domain} |X| 3 4 | 4 \in \text{domain} |X| 5 7)
\]

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

- \( \mathcal{U} \) 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.
- \( P \equiv e_1 \rightarrow R_1, \ldots, e_n \rightarrow R_n \) is a finite conjunction of so-called productions, whose each \( e_i \) is a distinct variable and \( e_i \) is an expression or a pair of the form \( < \tau, T > \), where \( \tau \) is an instance of the pattern in the root of an cDT(\mathcal{F}D) \( T \). Those productions \( e \rightarrow R \) whose left hand side \( e \) is simply an expression are called suspensions, while those whose left hand side is of the form \( < \tau, T > \) are called demanded productions. The set of produced variables of \( G \) is defined as \( \text{pvar}(P) =_{\text{def}} \{ R_1, \ldots, R_n \} \).
- \( C \equiv \delta_1, \ldots, \delta_k \) 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.
- \( \sigma \) is an idempotent substitution called answer substitution such that \( \text{dom}(\sigma) \cap \text{var}(P \circ C \circ S) = \emptyset \).

An initial goal can be any admissible goal, but for practical use in programming, \( P \) and \( S \) are usually empty and \( \sigma \) is the identity substitution \( \varepsilon \).

Similarly to [20, 25], we use a notion of \textit{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 \( X \in \text{var}(G) \) is a \textit{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 \( \mu \) of \( S \) (see [20] for more details and for practical use).
2. \( X \) is demanded by a suspension \( (X \pi_k \rightarrow R) \in P \) such that \( k > 0 \) and \( R \) is a demanded variable in \( G \).
3. \( X \) is demanded by a production with definitional tree \( (< \varepsilon, \text{case}(\tau, Y, \{T_1, \ldots, T_k \}) \rightarrow R) \in P \) such that \( X \) occurs in \( e \) at the same position that the case-distinction variable \( Y \) in \( \tau \).

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:

- \textit{Demanded productions} \( < \tau, T > \rightarrow R \) 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 \( \tau \) is always an instance of the call pattern in the root of the tree \( T \).
- \textit{Suspensions} \( e \rightarrow R \) 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 \textit{answer} for a goal \( G \) and a given COISS(\mathcal{D})-program \( \mathcal{P} \), must have the form \( \Pi \circ \theta \), where \( \Pi \) is a finite conjunction of total primitive constraints and \( \theta \) is an idempotent substitution such that \( \text{dom}(\theta) \cap \text{var}(\Pi) = \emptyset \).

Additional results relating all of these notions with respect to a suitable declarative semantic are given in [19, 20] by means of a \textit{Constraint Rewriting Logic CRWL(\mathcal{D})}, which can be also directly instantiates with our finite domain \( \mathcal{F}D \).
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 $\text{CLNC}(\mathcal{D})$ 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 $\text{CLP}(\mathcal{D})$ programs [14], successful derivations will eventually end with a solved goal (with $P$ and $C$ empty) identifying a computed answer $\sigma$. Failing derivations (ending with an obviously inconsistent goal $\Box$) 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 $\text{TOY}(\mathcal{FD})$ 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 $\text{CLNC}(\mathcal{D})$ 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 $cDT(\mathcal{FD}) \; T$ and introducing a new demanded production $e. T \rightarrow R$ in the goal in order to perform a more convenient and efficient narrowing strategy by means of the definitional tree $T$ (instead of perform directly a non-deterministic and inefficient rewriting step over $e$, as it occurs with the goal transformation rule $\text{Defined Function}$ in the $\text{CLNC}(\mathcal{D})$ calculus).

The goal transformation rules corresponding to demanded productions $e. T \rightarrow R$ (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 $T$, in a vein similar to [12, 25]:

- If $T$ is a 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 $T$ is a 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 $T_i$ associated to $h_i$ (if possible; otherwise we fail). If the symbol is a non-processed variable $Y$, then we can select a subtree $T_i$, generating an appropriate binding $\{ Y \leftarrow h_i \; Y_{m_i} \}$ with $Y_{m_i}$ fresh variables such that $h_i \; Y_{m_i} \in Pat_2(\mathcal{Z})$. 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 $\text{CLNC}(\mathcal{D})$ 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 $(\text{from} \; M) < 3$ using the $\text{COISSL}(\mathcal{FD})$ 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 $T_{\text{from}}$ and $T_{\text{check}}$ are used to guide and avoid not knowing choices of program rules and failure computations.

Example 2}

\begin{enumerate}
  \item $\Box \; \text{check} \; (\text{from} \; M) < 3 \; \Box \; e \vdash$ (we evaluate the non-primitive arguments of the $\mathcal{FD}$ constraint)
  \item $\exists R. \; \text{check} \; (\text{from} \; M) \rightarrow R \; \Box \; R < 3 \; \Box \; e \vdash$ (if $R$ is a demanded variable by the constraint store)
  \item $\exists R. \; \text{check} \; (\text{from} \; M), T_{\text{check}} \rightarrow \rightarrow R \; \Box \; R < 3 \; \Box \; e \vdash$ (defined function ‘from’ in the case-distinction position)
  \item $\exists R', R. \; \text{check} \; (\text{from} \; M), T_{\text{from}} \rightarrow \rightarrow R' \; \Box \; R < 3 \; \Box \; e \vdash$ (‘from’ rule application with $R'$ demanded)
  \item $\exists R', R. \; \text{from} \; (M + 1) \rightarrow R' \; \Box \; e \vdash$ (needed narrowing strategy)
  \item $\exists R', R. \; \text{from} \; (M + 1) \rightarrow R' \; \Box \; e \vdash$ (syntactic unification by imitation in the narrowing process)
  \item $\exists As, R. \; \text{from} \; (M + 1) \rightarrow As \; \Box \; R \; \Box \; e \vdash$
\end{enumerate}

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.

\begin{enumerate}
  \item $\exists R. \; \text{from} \; (M + 1) \rightarrow As \; \Box \; \text{check} \; (M[A]), T_{\text{check}} \rightarrow \rightarrow R \; \Box \; R < 3 \; \Box \; e \vdash$
\end{enumerate}

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.

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

\begin{enumerate}
  \item $\exists As, R. \; \text{from} \; (M + 1) \rightarrow As \; \Box \; \text{domain} \; (M) \; 3, 4 \; R < 3 \; \Box \; e \vdash$ (finally, the constraint solver shows the first computed answer)
  \item $\exists As, R. \; \text{from} \; (M + 1) \rightarrow As \; \Box \; \text{domain} \; (M) \; 3, 4 \; 2 < 3 \; \Box \; e \vdash$
  \item $\exists As, R. \; \text{from} \; (M + 1) \rightarrow As \; \Box \; \text{domain} \; (M) \; 3, 4 \; e \vdash$
  \item $\Box \; \text{domain} \; (M) \; 3, 4 \; e \vdash$
\end{enumerate}

The main properties of our computation strategy with definitional trees, soundness and completeness with respect to $\text{CF LP}(\mathcal{FD})$ semantics, can be obtained by using techniques similar to those used for the $\text{CLNC}(\mathcal{D})$ 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 $CDNC(FD)$ are in essence needed narrowing derivations modulo non-deterministic choices between overlapping and constrained program rules over $FD$. 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 $TOY(FD)$ System

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

In this section, we show how the execution in $TOY(FD)$ 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 $TOY(FD)$ 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 $TOY$.
- $plgenerated$: the compilation of a $TOY(FD)$ 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 $TOY(FD)$. The system loads it automatically at the beginning. It contains all the common predicates to all the $TOY$ programs.

We begin the computation

\[(l) \quad \text{check (from } M) < 3 \quad \epsilon \epsilon \epsilon \]

Initially $TOY(FD)$ 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 $FD$ constraint solver of SICStus.

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

```
Call: initToy:$\text{\textasciitilde}$('S$\text{\textasciitilde}$sus$\text{\textasciitilde}$'Check', '[S$\text{\textasciitilde}$sus$\text{\textasciitilde}$'Sfrom', '[23825], 24738, 24736], 24561, 24559), 3, true, [], 20384) ?
```

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

\[
\ll <(X,Y,H,Cin,Cout) :-
\ll \quad \text{hnf}(X,H,Cin,Cout),
\ll \quad \text{hnf}(Y,H,Cin,Cout),
\ll \quad \underline{\text{number}}(HX,\text{number}(HY)) :-
\ll \quad (HX < HY,H=true;HX>=HY,H=false);\text{errPrim}).
\]

Before the invocation of the $FD$ 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 $FD$ 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 (\textless) in $HX<HY$.

\[(2) \quad \exists R. \text{check (from } M) \rightarrow R \square \square \square R < \epsilon \square \epsilon \epsilon \epsilon \]

This step of our constrained narrowing derivation evaluates the non-primitive argument of the $FD$ 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('S$\text{\textasciitilde}$sus$\text{\textasciitilde}$'Check', '[S$\text{\textasciitilde}$sus$\text{\textasciitilde}$'Sfrom', '[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 \textless check (from $M$), $T_{\text{check}} > \rightarrow R$

\[(3) \quad \exists R. \text{check (from } M), T_{\text{check}} \rightarrow \rightarrow R \square \square \square R < \epsilon \epsilon \epsilon \epsilon \epsilon \epsilon \]

The $TOY(FD)$ system awake the suspension by means of the hnf$_{sus}$ 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 \textit{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$_{sus}$. 

```
Call: toycomm:hnf_sus('Check', '[S$\text{\textasciitilde}$sus$\text{\textasciitilde}$'Sfrom', '[23825], 24738, 24736], 24561, [], 28443) ?
```

The definition of hnf$_{sus}$ for the check function is the following

```
```
And the corresponding call is

\[ \text{Call: plgenerated:hnf('$$susp'('$$from', \ldots ))} \]

The definition of the \$check predicate is

\[ '\text{check}(\langle A, B, C, D \rangle) : = \text{hnf}(\langle A, \ldots \rangle). \]

\[ '\text{check}_1(\langle E, B, F, D \rangle). \]

The \$check predicate first calculates the \text{hnf} of its argument \textit{from} \textit{M}

\[ \text{Call: plgenerated:hnf('$$susp'('$$from', \ldots ))} \]

In our strategy, \text{check} is a case tree and the symbol occurring in \text{check} \text{(from} \text{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.

\[ (4) \exists R', R < \text{from} \text{M}, \text{T}_{\text{from}} \rightarrow R', \text{T}_{\text{check}} \rightarrow R \]

Returning to the trace, If the argument \text{(from} \text{M}) to evaluate is neither a variable nor an expression with a passive symbol in the root, it has to call again the predicate \text{hnf}_\text{susp}

\[ \text{Call: toycomm:hnf}_\text{susp('$$from', \ldots ))} \]

The function \text{hnf}_\text{susp} for \textit{from} is defined as

\[ \text{hnf}_\text{susp}('\text{from}', \langle A \ldots \rangle) : = \langle A, \ldots \rangle. \]

And the corresponding call is

\[ \text{Call: plgenerated:$$from', \ldots ))} \]

The function \$from is defined as

\[ '\text{from}(\langle A : \ldots \rangle, \ldots \rangle) : = \langle A : \ldots \rangle. \]

Since this is a Prolog fact, the \text{hnf} computation of \textit{from} is finished.

\[ (5) \exists R', R < [M \text{from} (M + 1)] \rightarrow R', \text{T}_{\text{check}} \rightarrow R \]

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

\[ \text{Exit: plgenerated:hnf('$$susp'('$$from', \ldots ))} \]

When the computation of the \text{hnf} of the argument \$from of the \$check predicate is concluded, a case-distinction by means of the predicate \$check, \text{check}_1 arises according to the \text{check} function defined in Example 1.

\[ (6) \exists R, \text{from} (M + 1) \rightarrow R, \text{T}_{\text{check}} \rightarrow R \]

We have four possible alternatives according to the application of the program rules defining \text{check} in the \text{rule} subtree.

\[ '\text{check}_1(\langle A, B, C, D \rangle) : = \text{unifyHnfs}(\langle A, \ldots \rangle). \]

\[ '\text{from}(\langle A : \ldots \rangle, \ldots \rangle) : = \langle A : \ldots \rangle. \]

\[ '\text{check}_1(\langle A, B, C, D \rangle) : = \text{unifyHnfs}(\langle A, \ldots \rangle). \]

\[ '\text{check}_1(\langle A, B, C, D \rangle) : = \text{unifyHnfs}(\langle A, \ldots \rangle). \]

The call to the predicate \$check, \text{check}_1 is

\[ \text{Call: plgenerated:$$check', \ldots ))} \]

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

\[ \text{Call: plgenerated:unifyHnfs('$$from', \ldots ))} \]

Since is fails, it continues with the next clause

\[ \text{Call: plgenerated:unifyHnfs('$$from', \ldots ))} \]

As this one is successful, it continues with \$domain

\[ \text{Call: plgenerated:$$domain', \ldots ))} \]

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

\[ \text{Call: plgenerated:domain([\ldots ])} \]

The call to \text{domain} returns success and, consequently, the \text{check, \text{check}_1 function also succeeds and returns 1 as the \text{hnf} of the first argument, (i.e. \text{check} \text{(from} \text{M}))

\[ (7) \exists R, \text{from} (M + 1) \rightarrow R, \text{domain} \rightarrow R \]

We obtain a binding for the first argument of our constraint; in the trace we put the number 1 in bold
5. Conclusions and Future Work

In this paper, we have presented a functional logic programming approach to finite domain (FD) constraint solving by means of a particular instance over FD of the generic scheme CFLP(D) [19], giving rise to a suitable CFLP(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(FD)-programs by means of an optimization of the generic Constrained Lazy Narrowing Calculus CLNC(D) [20] over FD, using definitional trees to guide the choice of demand/needed restructures. Moreover, we have described how this strategy can be integrated in the CFLP(FD) system TOY(FD) [9] by means of an example that combines lazy functions with constraint solving over finite domains using the efficient 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(FD) language. Since CFLP(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 FD, and a way to quantify the efficiency improvements of our TOY(FD) implementation by using definitional trees with respect to Curry [11] and other CFLP(FD) or CLP(FD) implementations.

Last but not least, we are working on declarative debugging techniques for CFLP(FD) programs in TOY(FD), following previous work for FLP in TOY [6, 7] as well as related work for CLP(D)-programs in [24]. The Constraint ReWriting Logic CWL(FD) already provides a formal framework for the declarative debugging of wrong answers. We are designing an extension of CWL(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(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


