Constraint Functional Logic Programming over Finite Domains

ANTONIO J. FERNÁNDEZ [*]
Dpto. de Lenguajes y Ciencias de la Computación,
Universidad de Málaga, Spain
E-mail: afdez@lcc.uma.es

TERESA HORTALÁ-GONZÁLEZ[†],

FERNANDO SÁENZ-PÉREZ [†] and RAFAEL DEL VADO-VÍRSEDA [‡]
Dpto. de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
E-mails: {teresa,fernan,rdelvado}@sip.ucm.es

submitted 15 July 2004; revised 1 August 2005; accepted 13 January 2006

Abstract

In this paper, we present our proposal to Constraint Functional Logic Programming over Finite Domains ($CFLP(\mathcal{FD})$) with a lazy functional logic programming language which seamlessly embodies finite domain ($\mathcal{FD}$) constraints. This proposal increases the expressiveness and power of constraint logic programming over finite domains ($CLP(\mathcal{FD})$) by combining functional and relational notation, curried expressions, higher-order functions, patterns, partial applications, non-determinism, lazy evaluation, logical variables, types, domain variables, constraint composition, and finite domain constraints.

We describe the syntax of the language, its type discipline, and its declarative and operational semantics. We also describe $\mathcal{TOY}(\mathcal{FD})$, an implementation for $CFLP(\mathcal{FD})$, and a comparison of our approach with respect to $CLP(\mathcal{FD})$ from a programming point of view, showing the new features we introduce. And, finally, we show a performance analysis which demonstrates that our implementation is competitive with respect to existing $CLP(\mathcal{FD})$ systems and that clearly outperforms the closer approach to $CFLP(\mathcal{FD})$.

KEYWORDS: Constraint Logic Programming, Functional Logic Programming, Finite Domains.

1 Introduction

Constraint logic programming (CLP) (Jaffar and Maher 1994) was born from a desire to have both a better problem expression and performance than logic pro-

[*] This author was partially supported by Spanish MCyT under contracts TIN2004-7943-C04-01 and TIN2005-08818-C04-01.
[†] The work of this author has been supported by the Spanish project PR 48/01-9901 funded by UCM.
[‡] The work of this author has been partially supported by the Spanish National Project MELODIAS (TIC2002-01167).

gramming (LP). Its success lies in that it combines the declarativeness of LP with the efficiency of the constraint programming (CP) paradigm. The essential component of the CLP schema is that it can be parameterized by a computational domain so that different domains determine different instances of the schema. Constraint Programming over finite domains ($CP(\mathcal{FD})$) (Marriot and Stuckey 1998; Henz and Müller 2000) has emerged as one of the most exciting paradigms of programming in recent decades. There are several reasons for the interest that $CP(\mathcal{FD})$ has raised: (1) the strong theoretical foundations (Tsang 1993; Apt 2003; Frühwirth and Abdennadher 2003) that make CP a sound programming paradigm; (2) $CP(\mathcal{FD})$ is a heterogeneous field of research ranging from theoretical topics in mathematical logic to practical applications in industry (particularly, problems involving heterogeneous constraints and combinatorial search) and (3) CP is based on posing constraints, which are basically true relations among domain variables. For this last reason, the declarative languages seem to be more appropriate than imperative languages to formulate $\mathcal{FD}$ constraint problems. In particular, one of the most successful instances of $CP(\mathcal{FD})$ is $CLP(\mathcal{FD})$.

Another well-known instance of declarative programming (DP) is functional programming (FP). The basic operations in functional languages are defined using functions which are invoked using function applications and put together using function composition. FP gives great flexibility, different from that provided by $(C)LP$, to the programmer, because functions are first-class citizens, that is, they can be used as any other object in the language (i.e., results, arguments, elements of data structures, etc). Functional languages provide evident advantages such as declarativeness, higher-order functions, polymorphism and lazy evaluation, among others. To increase the performance, one may think of integrating $\mathcal{FD}$ constraints into FP (as already done in LP). However, literature lacks proposals in this sense and the reason seems to lie in the relational nature of $\mathcal{FD}$ constraints, which do not fit well in FP. In spite of this limitation, it seems clear that the integration of $\mathcal{FD}$ constraints into FP is interesting not only for FP but also for discrete constraint programming, as the constraint community may benefit from the multiple advantages of FP.

More recently, functional logic programming (FLP) emerges with the aim to integrate the declarative advantages from both FP and LP. FLP gives rise to new features which cannot be found neither in FP nor in LP (Hanus 1994). FLP has not the inherent limitations of FP and thus it is an adequate framework for the integration of functions and constraints. To our best knowledge, the first proposal for a constraint functional logic programming scheme ($CFLP$) that attempts to combine constraint logic programming and functional logic programming is (Darlington et al. 1992). The idea behind this approach can be roughly described by the equation $CFLP(\mathcal{D}) = CLP(FP(\mathcal{D}))$, intended to mean that a $CFLP$ language over the constraint domain $\mathcal{D}$ is viewed as a CLP language over an extended constraint domain $FP(\mathcal{D})$ whose constraints include equations between expressions involving user defined functions, which will be solved via narrowing. Further, the $CFLP$ scheme proposed by F.J. López-Fraguas in (López-Fraguas 1992) tried to provide a declarative semantics such that $CLP(\mathcal{D})$ programs could be formally understood as a particular case of $CFLP(\mathcal{D})$ programs. In the classical approach to CLP semantics, a constraint domain is viewed as a first-order structure $\mathcal{D}$, and constraints are viewed as first-order formulas that can be interpreted in $\mathcal{D}$. In (López-Fraguas 1992), $CFLP(\mathcal{D})$ programs were built as sets of constrained rewriting rules. In order to support a lazy semantics for the user defined functions, constraint domains $\mathcal{D}$ were formalized as continuous structures, with a Scott domain (Gunter and Scott 1990) as a carrier, and a continuous interpretation of function and predicate symbols. The resulting semantics had many pleasant properties, but also some limitations. In particular, defined functions had to be first-order and deterministic, and the use of patterns in function definitions had to be simulated by means of special constraints.

In a recent work (López-Fraguas et al. 2004a), a new generic scheme $CFLP(\mathcal{D})$ has been proposed, intended 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 $CFLP(\mathcal{D})$ languages as in the $CLP(\mathcal{D})$ scheme, overcoming the limitations of the older $CFLP(\mathcal{D})$ scheme (López-Fraguas 1992). $CFLP(\mathcal{D})$ programs are presented as sets of constrained rewriting rules that define the behavior of possibly higher-order and/or non-deterministic lazy functions over $\mathcal{D}$. The main novelties in (López-Fraguas et al. 2004a) were a new formalization of constraint domains for $CFLP$, a new notion of interpretation for $CFLP(\mathcal{D})$ programs, and a new Constraint Rewriting Logic $CRWL(\mathcal{D})$ parameterized by a constraint domain, which provides a logical characterization of program semantics. Further, (López-Fraguas et al. 2004b) has formalized an operational semantics for the new generic scheme $CFLP(\mathcal{D})$ proposed in (López-Fraguas et al. 2004a). This work presented a constrained lazy narrowing calculus $CLNC(\mathcal{D})$ for solving goals for $CFLP(\mathcal{D})$ programs, which was proved sound and strongly complete with respect to $CRWL(\mathcal{D})$'s semantics. These properties qualified $CLNC(\mathcal{D})$ as a convenient computation mechanism for declarative constraint programming languages. More recently, (del Vado-Vírseda 2005) presented an optimization of the $CLNC(\mathcal{D})$ calculus by means of definitional trees (Antoy 1992) to efficiently control the computation. This new constrained demanded narrowing calculus $CDNC(\mathcal{D})$ preserves the soundness and completeness properties of $CLNC(\mathcal{D})$ and maintains the good properties shown for needed and demand-driven narrowing strategies (Loogen et al. 1993; Antoy et al. 2000; del Vado-Vírseda 2003). These properties adequately render $CDNC(\mathcal{D})$ as a concrete specification for the implementation of the computational behavior in existing $CFLP(\mathcal{D})$ systems such as $\mathcal{TOY}$ (Caballero et al. 1997) and *Curry* (Hanus 1999).

The main contributions of the paper are listed below:

- The paper describes the theoretical foundations for the $CFLP(\mathcal{FD})$ language, i.e., a concrete instance of the general scheme $CFLP(\mathcal{D})$ presented in (López-Fraguas et al. 2004a; López-Fraguas et al. 2004b). First, this instance includes an explicit treatment of a type system for constraints as well as for programs, goals and answers. Second, it also presents a new formalization of the constraint finite domain $\mathcal{FD}$ for $CFLP$ that includes a succinct declarative semantics (similarly as done for CLP) for an enough-expressive primitive constraints set. Finally, it provides the formal specification of a finite domain constraint solver defined over those primitive constraints that constitutes the theoretical basis for the implementation of correct propagation solvers.

- The paper presents an operational semantics for finite domain constraint solving on declarative languages using a new constraint lazy narrowing calculus $CLNC(\mathcal{FD})$, consisting of a natural and novel combination of lazy evaluation and $\mathcal{FD}$ constraint solving that does not exist, to our knowledge, in any declarative constraint solver (see (López-Fraguas et al. 2004b) and Section 5). This operational semantics is defined with respect to a constraint rewriting logic over a $\mathcal{FD}$ setting that makes it very different from the operational behavior of $CLP(\mathcal{FD})$ languages.
- The paper presents $\mathcal{TOY}(\mathcal{FD})$ from a programming point of view, which is the first $CFLP(\mathcal{FD})$ system that provides a wide set of $\mathcal{FD}$ constraints comparable to existing $CLP(\mathcal{FD})$ systems and which is competitive with them, as shown with performance results. Also, the advantages of combining $\mathcal{FD}$ constraints into FLP are highlighted via examples. Our system is therefore a contribution for increasing the expressiveness and efficiency of FLP by using $\mathcal{FD}$ constraints and a state-of-the-art propagation solver.

The structure of the paper is as follows. Section 2 presents our $CFLP(\mathcal{FD})$ language by describing its type discipline, patterns and expressions, finite domains, and constraint solvers. In Section 3, we provide a constraint lazy narrowing calculus over $\mathcal{FD}$ domains ($CLNC(\mathcal{FD})$) along with the notions of well-typed programs, admissible goals, and correct answers. Next, Section 4 describes our implementation of $CFLP(\mathcal{FD})$: $\mathcal{TOY}(\mathcal{FD})$, highlighting the advantages obtained from embodying constraints into a functional logic language with respect to $CLP(\mathcal{FD})$, and comparing the performance of our $CFLP(\mathcal{FD})$ system with other declarative constraint systems. Section 5 discusses related work and, finally, Section 6 summarizes some conclusions and points out future work.

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

We propose a constraint functional logic programming language over finite domains which is a pure declarative language, typed, lazy, and higher-order, and that provides support for constraint solving over finite domains. $CFLP(\mathcal{FD})$ aims to the integration of the best features of existing functional and logic languages into $\mathcal{FD}$ constraint solving.

In this section, we present the basis of our $CFLP(\mathcal{FD})$ language about syntax, type discipline, and declarative semantics. We also formalize the notion of a constraint finite domain and specify the expected behavior that a $\mathcal{FD}$ constraint solver attached to our $CFLP(\mathcal{FD})$ language must hold.

2.1 Polymorphic Signatures

We assume a countable set $\mathcal{TV}ar$ of *type variables* $\alpha,\ \beta,\ \dots$ and a countable ranked alphabet $TC = \bigcup_{n\in\mathbb{N}} TC^n$ of *type constructors* $C \in TC^n$. Types $\tau \in Type$ have the syntax $\tau ::= \alpha \quad | \ C\ \tau_1 \dots \tau_n \quad | \ \tau \to \tau' \ | \ (\tau_1, \dots, \tau_n)$.

By convention, $C\ \overline{\tau}_n$ abbreviates $C\ \tau_1 \dots \tau_n$, "$\to$" associates to the right, $\overline{\tau}_n \to \tau$ abbreviates $\tau_1 \to \cdots \to \tau_n \to \tau$, and the set of type variables occurring in τ is written $tvar(\tau)$. A type τ is called *monomorphic* iff $tvar(\tau) = \emptyset$, and *polymorphic* otherwise. $(\tau_1, \dots, \tau_n)$ is a type intended to denote n-tuples. A type without any occurrence of "$\to$" is called a *datatype*. *Datatype* definitions declare new (possibly polymorphic) *constructed types* and determine a set of *data constructors* for each type. Our $CFLP(\mathcal{FD})$ language provides predefined types such as $[A]$ (the type of polymorphic lists, for which Prolog notation is used), *bool* (with constants *true* and $false$), *int* for integer numbers, *char* (with constants a, b, ...) or *success* (with constant $\top$).
A *polymorphic signature* over TC is a triple $\Sigma = \langle TC,\ DC,\ FS\rangle$, where $DC = \bigcup_{n\in\mathbb{N}} DC^n$ and $FS = \bigcup_{n\in\mathbb{N}} FS^n$ are ranked sets of *data constructors* and *evaluable function symbols*. 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 $n \in \mathbb{N}$.

Each n-ary $c \in DC^n$ comes with a principal type declaration $c\ ::\ \overline{\tau}_n \to C\ \overline{\alpha}_k$, where $n, k \geq 0, \alpha_1, \dots, \alpha_k$ are pairwise different, τ_i are datatypes, and $tvar(\tau_i) \subseteq \{\alpha_1, \dots,\ \alpha_k\}$ for all $1 \leq i \leq n$. Also, every n-ary $f \in FS^n$ comes with a principal type declaration $f\ ::\ \overline{\tau}_n \to \tau$, where τ_i, τ are arbitrary types. For any polymorphic signature Σ, we write $\Sigma_\perp$ for the result of extending DC^0 in Σ with a special data constructor $\perp$, intended to represent an undefined value that belongs to every type. We also assume that DC^0 includes the three constants mentioned above *true*, $false\ ::\ bool$, and $\top\ ::\ success$, which are useful for representing the results returned by various primitive functions. As notational conventions, in the rest of the paper, we use $c, d \in DC$, $f, g \in FS$ and $h \in DC \cup FS$, and we define the arity of $h \in DC^n \cup FS^n$ as $ar(h) = n$.

2.2 Expressions, Patterns and Substitutions

In the sequel, we always assume a given polymorphic signature Σ, often not made explicit in the notation. We introduce the syntax of applicative expressions and patterns, which is needed for understanding the construction of constraint finite domains and constraint finite domain solvers. We assume a countably infinite set $\mathcal{V}ar$ of *(data) variables* $X, Y, \dots$ and the integer set $\mathbb{Z}$ of primitive elements 0, 1, -1, 2, -2, ..., mutually disjoint and disjoint from $\mathcal{TV}ar$ and Σ. Primitive elements are intended to represent the finite domain specific set of integer values.

An *expression* $e \in Exp_\perp(\mathbb{Z})$ has the syntax $e ::= \perp \mid\ u \ \mid\ X \ \mid\ h \ \mid\ (e\,e') \ \mid (e_1,\ \dots,\ e_n)$ where $u \in \mathbb{Z}$, $X \in \mathcal{V}ar$, $h \in DC \cup FS$, $(e\,e')$ stands for the *application* operation which applies the function denoted by e to the argument denoted by e' and $(e_1, \dots, e_n)$ represents *tuples* with n components. The set of data variables occurring in e is written $var(e)$. Moreover, e is called *linear* iff every $X \in var(e)$ has a single occurrence in e, *ground* iff $var(e) = \emptyset$ and *total* iff is an expression with no occurrences of $\perp$. *Partial patterns* $t \in Pat_{\perp}(\mathbb{Z}) \subset Exp_{\perp}(\mathbb{Z})$ are built as $t ::= \perp \mid u \mid X \mid c\,t_1 \ldots t_m \mid f\,t_1 \ldots t_n$ where $u \in \mathbb{Z}$, $X \in \mathcal{V}ar$, $c \in DC$ with $m \leq ar(c)$, and $f \in FS$ with $n < ar(f)$. Notice that partial applications (i.e., application to less arguments than indicated by the arity) of c and f are allowed as patterns, which are then called *higher-order patterns* (González-Moreno et al. 99b), because they have a functional type. 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 $h\ \overline{t_m} \sqsubseteq h\ \overline{t'_m}$ whenever these two expressions are patterns and $t_i \sqsubseteq t'_i$ for all $i \in \{1, \ldots, m\}$.

As usual, we define *(data) substitutions* $\sigma \in Sub_{\perp}(\mathbb{Z})$ as mappings $\sigma : \mathcal{V}ar \rightarrow Pat_{\perp}(\mathbb{Z})$ extended to $\sigma : Exp_{\perp}(\mathbb{Z}) \rightarrow Exp_{\perp}(\mathbb{Z})$ in the natural way. By convention, we write ε for the identity substitution, $e\sigma$ instead of $\sigma(e)$ and $\sigma\theta$ for the composition of σ and θ. We define the domain $dom(\sigma)$ of a substitution σ in the usual way. A substitution σ such that $\sigma\sigma = \sigma$ is called *idempotent*. For any set of variables $\mathcal{X} \subseteq \mathcal{V}ar$ we define the *restriction* $\sigma \upharpoonright \mathcal{X}$ as the substitution σ' such that $dom(\sigma') = \mathcal{X}$ and $\sigma'(X) = \sigma(X)$ for all $X \in \mathcal{X}$. We use the notation $\sigma =_{\mathcal{X}} \theta$ to indicate that $\sigma \upharpoonright \mathcal{X} = \theta \upharpoonright \mathcal{X}$, and we abbreviate $\sigma =_{\mathcal{V}ar \setminus \mathcal{X}} \theta$ as $\sigma =_{\setminus \mathcal{X}} \theta$. *Type substitutions* can be defined similarly, as mappings $\sigma_t : \mathcal{T}\mathcal{V}ar \rightarrow Type$ with a unique extension $\hat{\sigma_t} : Type \rightarrow Type$, denoted also as σ_t. The set of all type substitutions is denoted as *TSubst*. Most of the concepts and notations for data substitutions (such as domain, composition, etc.) make sense also for type substitutions, and we will freely use them when needed.

2.3 Well-typed Expressions

Inspired by Milner's type system (Damas and Milner 1982) we now introduce the notion of well-typed expression. We define a *type environment* as any set T of type assumptions $X :: \tau$ for data variables s.t. T does not include two different assumptions for the same variable. The domain $dom(T)$ of a type environment is the set of all the data variables that occur in T. For any variable $X \in dom(T)$, the unique type τ s.t. $(X :: \tau) \in T$ is denoted as $T(X)$. The notation $(h :: \tau) \in_{var} \Sigma$ is used to indicate that Σ includes the type declaration $h :: \tau$ up to a renaming of type variables. *Type judgements* $(\Sigma, T) \vdash_{WT} e :: \tau$ with $e \in Exp_{\perp}(\mathbb{Z})$ are derived by means of the following *type inference* rules:

$(\Sigma, T) \vdash_{WT} u :: int$, for every $u \in \mathbb{Z}$.
$(\Sigma, T) \vdash_{WT} X :: \tau$, if $T(X) = \tau$.
$(\Sigma, T) \vdash_{WT} h :: \tau\sigma_t$, if $(h :: \tau) \in_{var} \Sigma_{\perp},\ \sigma_t \in TSubst$.
$(\Sigma, T) \vdash_{WT} (e\ e') :: \tau$, if $(\Sigma, T) \vdash_{WT} e :: (\tau' \rightarrow \tau)$, $(\Sigma, T) \vdash_{WT} e' :: \tau'$,
for some $\tau' \in Type$.
$(\Sigma, T) \vdash_{WT} (e_1, \ldots, e_n) :: (\tau_1, \ldots, \tau_n)$, if $\forall i \in \{1, \ldots, n\} : (\Sigma, T) \vdash_{WT} e_i :: \tau_i$.

An expression $e \in Exp_{\perp}$ is called *well-typed* iff there exist some *type environment* T and some type τ, such that the *type judgement* $(\Sigma, T) \vdash_{WT} e :: \tau$ can be derived.

Expressions that admit more than one type are called *polymorphic*. A well-typed expression always admits a so-called *principal type* (PT) that is more general than any other. A pattern whose PT determines the PTs of its subpatterns is called *transparent*.

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

Adopting the general approach of (López-Fraguas et al. 2004a; López-Fraguas et al. 2004b), 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 polymorphic 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 (López-Fraguas et al. 2004a) for details), and user defined functions. Requiring primitives to be *radical*, which 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 from $\perp$. As we will see in the next subsection, it is also possible to instantiate this notion of constraint finite domain by adding a new formal specification of a constraint finite domain solver $Solve^{\mathcal{FD}}$ as the theoretical basis of our operational semantics and implementation.

Assuming then a constraint finite domain $\mathcal{FD}$, we define first the syntax and semantics 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 and can return any value of the *Type* set.

- *Primitive constraints* have the syntactic form $p\,\overline{t}_n \;\to!\, t$, being $p \in PF^n$ a primitive function symbol and $t_1, \ldots, t_n, t \in Pat_\perp(\mathbb{Z})$ with t total.
- *Constraints* have the syntactic form $p\,\overline{e}_n \;\to!\, t$, with $p \in PF^n$, $e_1, \ldots, e_n \in Exp_\perp(\mathbb{Z})$ and $t \in Pat_\perp(\mathbb{Z})$ total.

In the sequel, we use the notation $PCon(\mathcal{FD})$ for the set of all the primitive constraints π over $\mathcal{FD}$. We reserve the capital letter Π for sets of primitive constraints, often interpreted as conjunctions. The semantics of primitive constraints depends on the notion of *valuation* $Val(\mathcal{FD})$ over $\mathcal{FD}$, defined as the set of substitutions of ground patterns for variables. The set of *solutions* of $\pi \in PCon(\mathcal{FD})$ is a subset $Sol_{\mathcal{FD}}(\pi) \subseteq Val(\mathcal{FD})$ that satisfy π in $\mathcal{FD}$ in the sense of (López-Fraguas et al. 2004a). Analogously, the set of solutions of $\Pi \subseteq PCon(\mathcal{FD})$ is defined as $Sol_{\mathcal{FD}}(\Pi) = \bigcap_{\pi \in \Pi} Sol_{\mathcal{FD}}(\pi)$. Moreover, we define the set of solutions of a pair $\Pi \;\square\; \sigma$ with $\sigma \in Sub_\perp(\mathbb{Z})$ as $Sol_{\mathcal{FD}}(\Pi \;\square\; \sigma) = Sol_{\mathcal{FD}}(\Pi) \cap Sol(\sigma)$, where $Sol(\sigma)$ is the set of valuations η such that $X\eta \equiv t\eta$ for each $X \mapsto t \in \sigma$.

The next definition presents a possible specific polymorphic signature with finite domain constraints constituted by a minimum set of primitive function symbols with their corresponding declarative semantics. By means of this signature, our $CFLP(\mathcal{FD})$ language allows the management of the usual finite domain constraints defined over $\mathbb{Z}$ in $CLP(\mathcal{FD})$ and also equality and disequality constraints defined over $Pat_{\perp}(\mathbb{Z})$ in a similar way as done in (González-Moreno et al. 99b).

Definition 1
Consider the following usual primitive operators and relations defined over $\mathbb{Z}$:

$$\otimes^{\mathbb{Z}} :: int \rightarrow int \rightarrow int, \quad \text{where } \otimes \in \{+,-,*,/\}$$
$$\asymp^{\mathbb{Z}} :: int \rightarrow int \rightarrow bool, \quad \text{where } \asymp \in \{=,\neq,<,\leq,>,\geq\}.$$

Table 1. Primitive Function Symbols in $\mathcal{FD}$ and their Declarative Interpretation

Strict Equality (on patterns)	$seq :: \alpha \rightarrow \alpha \rightarrow bool$ $seq^{\mathcal{FD}} : D_{\mathbb{Z}} \times D_{\mathbb{Z}} \rightarrow \{true, false, \perp\}$ $seq^{\mathcal{FD}}\ t\ \ t \rightarrow\ true, \forall t \in D_{\mathbb{Z}}$ 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 $\sqsubseteq$ $seq^{\mathcal{FD}}\ t_1\ t_2 \rightarrow\ \perp$, otherwise
Less or Equal (on integers)	$leq :: int \rightarrow int \rightarrow bool$ $leq^{\mathcal{FD}} : D_{\mathbb{Z}} \times D_{\mathbb{Z}} \rightarrow \{true, false, \perp\}$ $leq^{\mathcal{FD}}\ u_1\ u_2 \rightarrow\ true$, 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 $u_1 >^{\mathbb{Z}} u_2$ $leq^{\mathcal{FD}}\ u_1\ u_2 \rightarrow\ \perp$, otherwise
Operators (on integers)	$\otimes :: int \rightarrow int \rightarrow int$ $\otimes^{\mathcal{FD}} : D_{\mathbb{Z}} \times D_{\mathbb{Z}} \rightarrow D_{\mathbb{Z}}$ $\otimes^{\mathcal{FD}}\ u_1\ u_2 \rightarrow\ u_1 \otimes^{\mathbb{Z}} u_2$, if $u_1, u_2 \in \mathbb{Z}$ $\otimes^{\mathcal{FD}}\ u_1\ u_2 \rightarrow\ \perp$, otherwise
Finite Domains	$domain :: int \rightarrow [int] \rightarrow bool$ $domain^{\mathcal{FD}} : D_{\mathbb{Z}} \times D_{\mathbb{Z}} \rightarrow \{true, false, \perp\}$ $domain^{\mathcal{FD}}\ u\ [u_1, \ldots, u_n] \rightarrow\ true$, if $\forall i \in \{1, \ldots, n-1\}.u_i \leq^{\mathbb{Z}} u_{i+1}$ and $\exists i \in \{1, \ldots, n\}.u =^{\mathbb{Z}} u_i$ $domain^{\mathcal{FD}}\ u\ [u_1, \ldots, u_n] \rightarrow\ false$, if $\exists i \in \{1, \ldots, n-1\}.u_i >^{\mathbb{Z}} u_{i+1}$ or $\forall i \in \{1, \ldots, n\}.u \neq^{\mathbb{Z}} u_i$ $domain^{\mathcal{FD}}\ u\ [u_1, \ldots, u_n] \rightarrow \perp$, otherwise
Variable Labeling	$indomain :: int \rightarrow\ success$ $indomain^{\mathcal{FD}} : D_{\mathbb{Z}} \rightarrow \{\top, \perp\}$ $indomain^{\mathcal{FD}}\ u\ \rightarrow\ \top$, if $u \in \mathbb{Z}$ $indomain^{\mathcal{FD}}\ u\ \rightarrow\ \perp$, otherwise

Table 1 shows the set of primitive functions $p \in PF^n$ with their corresponding type declarations and their declarative interpretation $p^{\mathcal{FD}} \subseteq D^n_{\mathbb{Z}} \times D_{\mathbb{Z}}$ considered in our constraint finite domain $\mathcal{FD}$ (we use the notation $p^{\mathcal{FD}}\overline{t}_n \rightarrow t$ to indicate that $(\overline{t}_n, t) \in p^{\mathcal{FD}}$). We note that all our primitive functions satisfy the aforementioned properties. ♦

The function *indomain* is the basis for a *labeling* (*enumeration* or *search*) strategy, which is crucial in constraint solving efficiency. *labeling* is applied when no more constraint propagation is possible, and its basic step consists of selecting a variable X with a non-empty, non-singleton domain, selecting a value u of this domain, and assigning u to X. We note that in our framework, various labeling strategies (variable and value selection) have all the same declarative semantics, but they may differ in their operational behavior and therefore in efficiency as it happens in the $CLP(\mathcal{FD})$ setting (more details can be found in Section 4.5.1). 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 integer variables).
- $a \leq b$ abbreviates $leq\ a\ b \rightarrow!\ true$ (and analogously for the other comparison primitives $<$, $>$ and $\geq$).
- $a \otimes b \asymp c$ abbreviates $a \otimes b \rightarrow!\ r, r \asymp c$.
- $u \in D$ abbreviates $domain\ u\ D \rightarrow!\ true$ and $u_1, \ldots, u_n \in D$ abbreviates $u_1 \in D \wedge \ldots \wedge u_n \in D$. Analogously, $u \notin D$ abbreviates $domain\ u\ D \rightarrow!\ false$ and $u_1, \ldots, u_n \notin D$ abbreviates $u_1 \notin D \wedge \ldots \wedge u_n \notin D$.
- $domain\ [u_1, \ldots, u_n]\ a\ b$ with $a, b \in \mathbb{Z}$ $(a \leq b)$ abbreviates $u_1, \ldots, u_n \in [a\ ..\ b]$, where $[a\ ..\ b]$ represents the integer list $[a, a+1, \ldots, b-1, b]$ that intuitively represents the integer interval [a,b].
- $labeling\ L\ [u_1, \ldots, u_n]$ abbreviates and extends $indomain\ u_1 \rightarrow!\ \top \wedge \ldots \wedge indomain\ u_n \rightarrow!\ \top$ with a list L of predefined constants that allow to specify different labeling strategies.

Using these notations, a *primitive constraint store* $S \subseteq PCon(\mathcal{FD})$ can be expressed as a finite conjunction of primitive constraints of the form $t == s$, $t \setminus = s$, $u \in D$, $u \notin D$, $a \otimes b \asymp c$, $domain\ [u_1, \ldots, u_n]\ a\ b$ and/or $labeling\ L\ [u_1, \ldots, u_n]$ where t, s are total patterns, $u_i, u, a, b, c \in \mathbb{Z} \cup \mathcal{V}ar$, and L, D are total patterns representing a variable or a list.

2.5 Constraint Solvers over $\mathcal{FD}$

The design of a suitable operational semantics over finite domains for goal solving in $CFLP(\mathcal{FD})$ combines constrained lazy narrowing with constraint solving over a given constraint finite domain $\mathcal{FD}$. The central notion of lazy narrowing can be found in the literature, e.g., (Middeldorp and Okui 1998; Middeldorp et al. 2002). In this subsection, we describe the expected behavior of a constraint solver over the finite constraint domain $\mathcal{FD}$ w.r.t. the semantics given in the previous subsection, as the basis of our goal solving mechanism.

Definition 2
A *constraint solver* over a given constraint domain $\mathcal{FD}$ is a function named $Solve^{\mathcal{FD}}$ expecting as parameters a finite primitive constraint store $S \subseteq PCon(\mathcal{FD})$ in the sense of Definition 1 and a finite set of variables $\chi \subseteq \mathcal{V}ar$ called the set of *protected variables*. The constraint solver is expected to return a finite disjunction of

k alternatives: $Solve^{\mathcal{FD}}(S,\chi) = \bigvee_{i=1}^{k}(S_i \Box \sigma_i)$, where each $S_i \subseteq PCon(\mathcal{FD})$ and each $\sigma_i \in Sub_{\perp}(\mathbb{Z})$ is a total idempotent substitution satisfying the following requirements: no alternative can bind protected variables, for each alternative either all the protected variables disappear or some protected variable becomes *demanded* (i.e., no solution can bind these variables to an undefined value), no solution is lost by the constraint solver, and the solution space associated to each alternative is included in one of the input constraint stores (i.e., $Sol_{\mathcal{FD}}(S) = \bigcup_{i=1}^{k} Sol_{\mathcal{FD}}(S_i \Box \sigma_i)$). In the case $k = 0$, the disjunction is understood as failure and $Sol_{\mathcal{FD}}(S) = \emptyset$ (that means failure detection). ♦

(López-Fraguas et al. 2004b) describes a constraint solver defined on the domain $\mathcal{H}_{seq}$ in which the constraints considered are just those for the strict (dis)equality on pure patterns (i.e. those patterns constructed over an empty set of primitive elements). Now, in this paper, we extend this solver to the constraint domain $\mathcal{FD}$ in which we consider $\mathbb{Z}$ as the set of primitive elements. We follow this approach and assume that the solver $Solve^{\mathcal{FD}}$ will behave as follows: $Solve^{\mathcal{FD}}(S,\chi) = \bigvee_{i=1}^{k}(S_i \Box \sigma_i)$ iff $S \Box \varepsilon \Vdash_{\chi}^{*} \bigvee_{i=1}^{k}(S_i \Box \sigma_i) \not\Vdash_{\chi}$, where the relation $\Vdash_{\chi}$ expresses a solver resolution step, and $S \Box \varepsilon \not\Vdash_{\chi}$ indicates that S is in solved form w.r.t. the action of the constraint solver in the sense of Definition 2. Moreover, the relation $\Vdash_{\chi}$ manipulates disjunctions by combining them as follows:

$$\ldots \vee S_i \Box \sigma_i \vee \ldots \Vdash_{\chi} \ldots \vee \bigvee_j (S_j \Box \sigma_j) \vee \ldots \qquad \text{if } S_i \Box \sigma_i \Vdash_{\chi} \bigvee_j (S_j \Box \sigma_j)$$

Tables 2-5 show the sets of rules that define the relation $\Vdash_{\chi}$. These rules extend and complement those presented in (López-Fraguas et al. 2004b) specifically to work with finite domain constraints defined on the set of integer patterns. For clarity, we omit the corresponding failure rules, which can be easily deduced from our tables.

Table 2 captures the operational behavior of the constraint solver $Solve^{\mathcal{FD}}$ to manage constraints of the form *seq*, *leq* or *domain* when these return a variable as a result. The result variable is instantiated to each of its possible values (i.e., *true* and *false*) giving rise to different alternatives for each of the possibilities and propagating the corresponding bind to the remaining alternatives.

Table 2. General Rules for the Constraint Solver

$seq\ t\ s \rightarrow!\ R,\ S \ \Box\ \sigma \Vdash_{\chi} (t == s,\ S\theta_1 \ \Box\ \sigma\theta_1) \vee (t \setminus = s,\ S\theta_2 \ \Box\ \sigma\theta_2)$
$leq\ a\ b \rightarrow!\ R,\ S \ \Box\ \sigma \Vdash_{\chi} (a \leq b,\ S\theta_1 \ \Box\ \sigma\theta_1) \vee (a > b,\ S\theta_2 \ \Box\ \sigma\theta_2)$
$domain\ u\ [u_1,\ldots,u_n] \rightarrow!\ R,\ S \ \Box \sigma \Vdash_{\chi} (u \in [u_1,\ldots,u_n],\ S\theta_1 \ \Box\ \sigma\theta_1) \vee$ $(u \notin [u_1,\ldots,u_n],\ S\theta_2 \ \Box\ \sigma\theta_2)$ (For the 3 rules) only if $R \notin \chi$; with $\theta_1 = \{R \mapsto true\}$ and $\theta_2 = \{R \mapsto false\}$

Observe that, by applying the rules shown in Table 2, all the constraints based on the primitive *seq* are proposed as explicit constraints in form of strict equality or strict disequality. Then, the solver distinguishes several cases depending on the syntactic structure of the (integer) patterns used as arguments. Table 3 shows the rules to cover these cases that reproduce the process of syntactic unification between equalities and disequalities as it is done in the classical syntactic unification algorithms (see for example (Fernández 1992)).

Table 3. Rules for Strict (Dis)-Equality

$$u == u,\, S \,\Box\, \sigma \Vdash_\chi S \,\Box\, \sigma, \ \text{ if } u \in \mathbb{Z}$$
$$X == t,\, S \,\Box\, \sigma \Vdash_\chi t == t,\, S\theta \,\Box\, \sigma\theta, \ \ \text{ if } X \notin \chi \cup var(t),\, var(t) \cap \chi = \emptyset,\, \theta = \{X \mapsto t\}$$
$$(h\ t_1 \ldots t_n) == (h\ s_1 \ldots s_n),\, S \,\Box\, \sigma \ \Vdash_\chi t_1 == s_1,\, \ldots,\, t_n == s_n,\, S \Box\ \sigma$$

$$u \setminus = u',\, S \,\Box\, \sigma \Vdash_\chi S \,\Box\, \sigma, \ \text{ if } u, u' \in \mathbb{Z}, \text{ and } u \neq^{\mathbb{Z}} u'$$
$$X \setminus = (h\ t_1 \ldots t_n),\, S \,\Box\, \sigma \Vdash_\chi (\textstyle\bigvee_i (S\theta_i \,\Box\, \sigma\theta_i)) \vee (\bigvee_{k=1}^{n} (U_k \setminus = t_k\theta,\, S\theta \,\Box\, \sigma\theta))$$
$$\text{if } X \notin \chi,\, var(h\ t_1 \ldots t_n) \cap \chi \neq \emptyset,\, \theta_i = \{X \mapsto h_i\ \overline{Y}_{m_i}\}, \text{ with } h_i \neq h, \text{ and}$$
$$\theta = \{X \mapsto h\ \overline{U}_n\},\, \overline{Y}_{m_i},\, \overline{U}_n \text{ are new fresh variables.}$$
$$(h\ t_1 \ldots t_n) \setminus = u,\, S \,\Box\, \sigma \Vdash_\chi S \,\Box\, \sigma \ \text{ if } u \in \mathbb{Z}$$
$$(h\ t_1 \ldots t_n) \setminus = (h\ s_1 \ldots s_n),\, S \,\Box\, \sigma \Vdash_\chi (t_1 \setminus = s_1,\, S \,\Box\, \sigma) \vee \ldots \vee (t_n \setminus = s_n,\, S \,\Box\, \sigma)$$
$$(h\ t_1 \ldots t_n) \setminus = (h' s_1 \ldots s_m),\, S \,\Box\, \sigma \Vdash_\chi S \,\Box\, \sigma, \ \ \text{ if } h \neq h' \text{ or } m \neq n$$

In addition to the rules for the strict (dis)equality over integer patterns, the solver has also to consider, by contrast to the solver given in (López-Fraguas et al. 2004b), new rules for the particular treatment of the primitive constraints (specific for $\mathcal{FD}$) defined over the primitive elements in $\mathbb{Z}$. These rules are shown in Table 4.

Table 4. Rules for the Specific Primitive Constraints of $\mathcal{FD}$

$$u \leq u',\, S \,\Box\, \sigma \Vdash_\chi S \,\Box\, \sigma, \ \text{ if } u, u' \in \mathbb{Z}, \text{ and } u \leq^{\mathbb{Z}} u'$$
$$u > u',\, S \,\Box\, \sigma \Vdash_\chi S \,\Box\, \sigma, \ \text{ if } u, u' \in \mathbb{Z}, \text{ and } u >^{\mathbb{Z}} u'$$

$$a \otimes b \asymp c,\, S \,\Box\, \sigma \Vdash_\chi S \,\Box\, \sigma, \ \text{ if } a, b, c \in \mathbb{Z} \text{ and } a \otimes^{\mathbb{Z}} b \asymp^{\mathbb{Z}} c$$
$$a \otimes b = X,\, S \,\Box\, \sigma \Vdash_\chi S\theta \,\Box\, \sigma\theta, \ \text{ if } X \notin \chi,\, a, b \in \mathbb{Z} \text{ and } \theta = \{X \mapsto a \otimes^{\mathbb{Z}} b\}$$

Moreover, the solver also has to cover the *domain* and *indomain* classical constraints in finite domain constraint programming languages, to respectively fix the domain of the constrained variables and label them according to their corresponding domain (Dechter 2003). Table 5 shows the new rules that consider these cases.

After applying the constraint solver $Solve^{\mathcal{FD}}$, a primitive constraint store $S \subseteq PCon(\mathcal{FD})$ is expressed in *solved form* as a finite conjunction of primitive constraints of the form (we use the notations given in Section 2.4) $X == t$, $X \setminus = t$,

Table 5. Rules for Finite Domain and Variable Labeling

$$u \in [u_1,\ldots,u_n],\ S \ \Box\ \sigma \Vdash_\chi S \ \Box\ \sigma, \ \text{ if } u, u_i \in \mathbb{Z} \cup \mathcal{V}ar \text{ and } \exists i \in \{1,\ldots,n\}.\ u_i \equiv u.$$
$$u \notin [u_1,\ldots,u_n],\ S \ \Box\ \sigma \Vdash_\chi S \ \Box\ \sigma, \ \text{ if } u, u_i \in \mathbb{Z} \text{ and } \forall i \in \{1,\ldots,n\}.\ u_i \neq^{\mathbb{Z}} u.$$

$$\mathit{labeling}\ [\ldots]\ [X],\ X \in [u_1,\ldots,u_n],\ S \ \Box\ \sigma \Vdash_\chi \bigvee_{i=1}^{n} (S\theta_i \ \Box\ \sigma\theta_i),$$
$$\text{if } X \notin \chi, \text{ and } \forall i \in \{1,\ldots,n\},\ u_i \in \mathbb{Z} \text{ and } \theta_i = \{X \mapsto u_i\}$$
$$\mathit{labeling}\ [\ldots]\ [u],\ S \ \Box\ \sigma \Vdash_\chi S \ \Box\ \sigma, \ \text{ if } u \in \mathbb{Z}$$

$u \in D$ and $a \otimes b \asymp c$ where $X \in \mathcal{V}ar$, t is a total pattern, $u, a, b, c \in \mathbb{Z} \cup \mathcal{V}ar$ and D is a total pattern defining a list of variables and/or integers.

Example 1
We illustrate the operational semantics of our finite domain constraint solver providing a constraint solver derivation from the initial constraint store $\{seq\ X\ (s\ K) \rightarrow!\ R,\ A + B < Z\}$ and taking into account the set of protected variables $\{Z\}$. We describe in detail the rules applied by the constraint solver and, at each goal transformation step, we underline which subgoal is selected:

$$\underline{seq\ X\ (s\ K) \rightarrow!\ R},\ A + B < Z \ \Box\ \varepsilon \Vdash_{\{Z\}}$$
$$(\{X == s\ K,\ A + B < Z\} \ \Box\ \{R \mapsto true\}) \vee$$
$$(\{X \setminus = s\ K,\ A + B < Z\} \ \Box\ \{R \mapsto false\}) \Vdash_{\{Z\}}$$

- $(\{\underline{X == s\ K},\ A + B < Z\} \ \Box\ \{R \mapsto true\}) \Vdash_{\{Z\}}$
  $(\{\underline{s\ K == s\ K},\ A + B < Z\} \ \Box\ \{R \mapsto true, X \mapsto s\ K\}) \Vdash_{\{Z\}}$
  $(\{K == K,\ A + B < Z\} \ \Box\ \{R \mapsto true, X \mapsto s\ K\}) \not\Vdash_{\{Z\}}$
- $(\{\underline{X \setminus = s\ K},\ A + B < Z\} \ \Box\ \{R \mapsto false\}) \Vdash_{\{Z\}}$
  $(\{A + B < Z\} \ \Box\ \{R \mapsto false,\ X \mapsto 0\}) \vee$
  $(\{A + B < Z,\ M \setminus = K\} \ \Box\ \{R \mapsto false,\ X \mapsto s\ M\}) \not\Vdash_{\{Z\}}$

Therefore, the constraint solver returns the following solved forms:

$$Solve^{\mathcal{FD}}(\{\ seq\ X\ (s\ K) \rightarrow!\ R,\ A + B < Z\ \},\ \{Z\}\ ) =$$
$$(\{A + B < Z,\ K == K\} \ \Box\ \{R \mapsto true,\ X \mapsto s\ K\}) \vee$$
$$(\{A + B < Z\} \ \Box\ \{R \mapsto false,\ X \mapsto 0\}) \vee$$
$$(\{A + B < Z,\ M \setminus = K\} \ \Box\ \{R \mapsto false,\ X \mapsto s\ M\})$$

As shown in Tables 2-5, our new constraint solver for the finite domain $\mathcal{FD}$ with strict equality and disequality has been designed to hold all the initial assumptions required in the general framework $CFLP$ for constraint solvers (see Definition 2). It can be formally proved by means of the following result.

Theorem 1
Let $S \subseteq PCon(\mathcal{FD})$ be a primitive constraint store, $\sigma \in Sub_\perp(\mathbb{Z})$ an idempotent total substitution and $\chi \subseteq \mathcal{V}ar$ a set of protected variables. If $S \ \Box\ \sigma$ satisfies the requirements of Definition 2 and $S \ \Box\ \sigma \Vdash_\chi \bigvee_{i=1}^{k} (S_i \ \Box\ \sigma_i)$, then $Sol_{\mathcal{FD}}(S \ \Box\ \sigma) =$

$\bigcup_{i=1}^{k} Sol_{\mathcal{FD}}(S_i \ \Box\ \sigma_i)$, where $dom(\sigma_i) \cap var(S_i) \ = \ \emptyset$ and $\chi \cap (dom(\sigma_i) \cup ran(\sigma_i))$ $= \ \emptyset$ for all $1 \leq i \leq k$. Moreover, if $S \ \Box\ \sigma \Vdash_{\chi}$ fails then $Sol_{\mathcal{FD}}(S \ \Box\ \sigma) \ = \ \emptyset$.

The proof of this theorem (see Appendix A) can be done distinguishing several cases from the declarative semantics of each primitive function symbol given in Table 1 and the requirements of each constraint solver rule in Tables 2-5. According to this result, the relation $\Vdash_{\chi}$ preserves the requirements of a constraint solver and the constraint solver steps fail only in case of an unsatisfiable constraint store. Therefore, if we repeatedly apply this result from an initial constraint store and a set of protected variables in order to compute a constraint store in solved form, we directly obtain the correctness of our finite domain constraint solver w.r.t. $CFLP(\mathcal{FD})$'s semantics.

3 The $CLNC(\mathcal{FD})$ Calculus

This section describes a lazy narrowing calculus with constraints defined on the finite domain $\mathcal{FD}$ (the Constraint Lazy Narrowing Calculus $CLNC(\mathcal{FD})$ for short) for the solving of goals from programs. Since we have proved in the previous section that our finite domain constraint solver holds the properties required in the general framework, this calculus can be obtained as a simplified instantiation of the general scheme for $CFLP$ described in (López-Fraguas et al. 2004b), and used in this work as the formal foundation of the operational semantics in $\mathcal{TOY}(\mathcal{FD})$.

In order to understand the main ideas of our operational semantics, we first give a precise definition for the class of well-typed programs, admissible goals and correct answers we are going to work with.

3.1 Programs, Goals and Answers

Our well-typed $CFLP(\mathcal{FD})$*-programs* are sets of constrained rewriting 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 a defined function symbol $f \in DF^n$ with an associated principal type $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$ has the form $f\, t_1 \ldots t_n \ = r \ \Leftarrow C$ and is required to satisfy the conditions listed below:

1. $t_1 \ldots t_n$ is a linear sequence of transparent patterns and $r \in Exp_{\perp}(\mathbb{Z})$ is a total expression.
2. C is a finite set of total constraints (in the form of Definition 1), intended to be interpreted as conjunction, and possibly including occurrences of defined function symbols.
3. There exists some type environment T with domain $var(R)$ which well-types the defining rule in the following sense:
   (a) For $1 \leq i \leq n$: $(\Sigma, T) \vdash_{WT} t_i \ :: \ \tau_i$.
   (b) $(\Sigma, T) \vdash_{WT} r \ :: \ \tau$.
   (c) For each $(e == e') \in C$: $\exists \mu \in$ *Type* s.t. $(\Sigma, T) \vdash_{WT} e \ :: \ \mu \ :: \ e'$.
   (d) For each $(e \setminus = e') \in C$: $\exists \mu \in$ *Type* s.t. $(\Sigma, T) \vdash_{WT} e \ :: \ \mu \ :: \ e'$.

(e) For each $(u \in D) \in C$: $(\Sigma, T) \vdash_{WT} u :: int, D :: [int]$.
(f) For each $(a \otimes b \asymp c) \in C$: $(\Sigma, T) \vdash_{WT} a, b, c :: int$

where $(\Sigma, T) \vdash_{WT} e :: \tau :: e'$ denotes $(\Sigma, T) \vdash_{WT} e :: \tau$, $(\Sigma, T) \vdash_{WT} e' :: \tau$.

The left-linearity condition required in item 1 is quite common in functional and functional logic programming. As in constraint logic programming, the conditional part of a program rule needs no explicit occurrences of existential quantifiers. Another distinguished feature of our language is that no confluence properties are required for the programs, and therefore functions can be *non-deterministic*, i.e. return several values for given (even ground) arguments.

Example 2
The following example illustrates the previous definition of typed $CFLP(\mathcal{FD})$-programs by showing some constrained program rules which will be used for lazy evaluation of infinite lists in the next subsections.

take :: int $\rightarrow [\alpha] \rightarrow [\alpha]$
(T1) *take* 0 *Xs* $= [\,]$
(T2) *take N* $[\,]$ $= [\,]$ $\Leftarrow N > 0$
(T3) *take N* $[X \mid Xs]$ $= [X|$*take (N - 1) Xs*$]$ $\Leftarrow N > 0$

check_list :: $[int] \rightarrow int$
(CL1) *check_list* $[\,]$ $= 0$
(CL2) *check_list* $[X|Xs]$ $= 1$ $\Leftarrow$ *domain* $[X]$ *1 2*
(CL3) *check_list* $[X|Xs]$ $= 2$ $\Leftarrow$ *domain* $[X]$ *3 4*
(CL4) *check_list* $[X|Xs]$ $= 4$ $\Leftarrow$ *domain* $[X]$ *5 7*

generateFD :: int $\rightarrow [int]$
(G1) *generateFD* 0 $= [\ ]$
(G2) *generateFD N* $= [X \mid$ *generateFD N*$]$ $\Leftarrow N > 0,$ *domain* $[X]$ *0 N-1*

from :: int $\rightarrow [int]$
(F) *from N* $= [N \mid$ *from (N+1)*$]$

According to (López-Fraguas et al. 2004b), we define *goals* for this kind of programs in the general form $G \equiv \exists \overline{U}.\ P \ \square\ C \ \square\ S \ \square\ \sigma$, where the symbol $\square$ must be interpreted as conjunction, $\overline{U}$ is the finite set of so-called *existential variables* of the goal G, P is a multiset of so-called *productions* of the form $e_1 \rightarrow t_1, \ldots, e_n \rightarrow t_n$, where $e_i \in Exp_\perp(\mathbb{Z})$ and $t_i \in Pat_\perp(\mathbb{Z})$ are totals for all $1 \leq i \leq n$ (the set of *produced variables* of G is defined as the set of variables occurring in $t_1 \ldots t_n$), C is a finite conjunction of constraints (possibly including occurrences of defined function symbols), S is a finite conjunction of primitive constraints in the form of Definition 1, called *constraint store*, and σ is an idempotent substitution called *answer substitution* such that $dom(\sigma) \cap var(P \ \square\ C \ \square\ S) = \emptyset$.

Additionally, we say that a goal G is an *admissible goal* iff it is well-typed: satisfies the same admissibility criteria given above for programs for each constraint in C

and S, and the same conditions of compatible types for each production in P and each binding in σ given in (González-Moreno et al. 99b). Moreover, it must hold the so-called *goal invariants* given in (López-Fraguas et al. 2004b): each produced variable is produced only once, all the produced variables must be existential, the transitive closure of the relation between produced variables must be irreflexive, and no produced variable enters the answer substitution. An admissible goal is called a *solved goal* iff P and C are empty and S is in solved form w.r.t. the action of the constraint solver in the sense of Definition 2.

Similarly to (González-Moreno et al. 99a; González-Moreno et al. 99b; del Vado-Vírseda 2003), the $CLNC(\mathcal{FD})$ calculus uses a notion of *demanded variable* to deal with lazy evaluation.

Definition 3
Let G be an admissible goal. We say that $X \in var(G)$ is a *demanded variable* iff

1. X is *demanded by the constraint store* S of G, i.e. $\mu(X) \neq \bot$ holds for every $\mu \in Sol_{\mathcal{FD}}(S)$ (for practical use, the calculation of this kind of demanded variables in $\mathcal{FD}$ can be easily done extending the rules given in the appendix of (López-Fraguas et al. 2004b) in the line of our rules shown in Tables 2-5).
2. X is *demanded by a production* $(X\overline{a}_k \rightarrow t) \in P$ such that, $t \notin \mathcal{V}ar$ or $k > 0$ and t is a demanded variable in G.

Example 3
We suppose an admissible goal with only the primitive constraint $seq\ X\ (s\ K) \rightarrow!\ R$ in the associated constraint store S. We note that K is not a demanded variable by S, because $\mu = \{X \mapsto 0,\ K \mapsto \bot,\ R \mapsto false\} \in Sol_{\mathcal{FD}}(S)$ (clearly, $seq^{\mathcal{FD}}\ \mu(X)\ (s\ K)\mu \rightarrow \mu(R) = false$ where $\mu(X) = 0$ and $(s\ K)\mu = s\ (\mu(K)) = s\ (\bot)$ have no common upper bound w.r.t. the information ordering $\sqsubseteq$, according to Table 1) but $\mu(K) = \bot$. However, X and R are both demanded variables by S (according to the *radicality* property, any $\mu \in Sol_{\mathcal{FD}}(S)$ must satisfy $\mu(R)$ total and then $\mu(R) \neq \bot$ and consequently $\mu(X) \neq \bot$). In this situation, if we have also a production $F\ 1 \rightarrow X$ in the produced part of the goal involving a higher-order variable F, automatically F is also a demanded variable (by a production but not by the constraint store S). Moreover, we note that it is also possible to have a variable F demanded by both the constraint store (for example, if we add the primitive constraint $F == \oplus\ 2$ to S) and a production (for example, $F\ 1 \rightarrow 3$ instead of $F\ 1 \rightarrow X$). In this case, F is demanded twice, supplying more relevant and precise information for goal solving in the produced part and the constraint store of the goal.

Finally, we describe the notion of correct answer that we want to compute from goals and programs in our $CFLP(\mathcal{FD})$-framework. Since the calculus $CLNC(\mathcal{FD})$ is semantically based on the *Constraint ReWriting Calculus* $CRWL(\mathcal{FD})$, that represents a concrete instance over the constraint domain $\mathcal{FD}$ of the constraint rewriting logic described in (López-Fraguas et al. 2004a), this logic can be also used as a logical characterization of our program semantics. On the basis of this logic, we define our concept of correct *answer* with respect to an admissible goal G and a given $CFLP(\mathcal{FD})$-program as a pair of the form $\Pi \,\Box\, \theta$, where $\Pi \subseteq PCon(\mathcal{FD})$ and $\theta \in Sub_{\perp}(\mathbb{Z})$ is an idempotent substitution such that $dom(\theta) \cap var(\Pi) = \emptyset$, fulfilling the same semantic conditions given in (López-Fraguas et al. 2004b) w.r.t. $CRWL(\mathcal{D})$'s semantics.

The following example shows a correct answer for the admissible goal with only a strict equality $\Box\ take\ 3\ (generateFD\ 10) == List\ \Box\ \Box$ and the $CFLP(\mathcal{FD})$-program given in Example 2:

$$\{X_1, X_2, X_3 \in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]\} \,\Box\, \{List \mapsto [X_1, X_2, X_3]\}$$

Analogously, it is also possible to prove that $M \in [1..2]$ and $M \in [3..4]$ (both of them with an empty substitution) are correct answers for the admissible goal with only a user defined finite domain constraint $\Box\ check\_list\ (from\ M) < 3\ \Box\ \Box$. We will see in the next subsection how to compute all of these answers by means of the constrained lazy narrowing over $\mathcal{FD}$.

3.2 Constrained Lazy Narrowing over $\mathcal{FD}$

The calculus $CLNC(\mathcal{FD})$ can be obtained as a particular instantiation from the general $CLNC(\mathcal{D})$ calculus because we have proved that our finite domain constraint solver satisfies the requirements given in the general framework. Therefore, the calculus $CLNC(\mathcal{FD})$ can be described as a set of transformation rules for admissible goals of the form $G \Vdash G'$, specifying one of the possible ways of performing one step of goal solving. In this sense, *derivations* are sequences of $\Vdash$-steps where successful derivations will eventually end with a solved goal and failing derivations end with an inconsistent goal $\blacksquare$. We have two classes of goal transformation rules: rules for constrained lazy narrowing by means of productions, and rules for constraint solving and failure detection.

The goal transformation rules concerning productions are the same rules given in (López-Fraguas et al. 2004b) for general productions and are designed with the aim of modeling the behavior of constrained lazy narrowing with *sharing*, but now involving only the primitive functions over finite domains given in Definition 1, possibly higher-order defined functions and functional variables.

The goal transformation rules concerning constraints can also be used to combine (primitive or used defined) finite domain constraints with the action of our constraint finite domain solver. As the main novelty, we note that only primitive constraints are sent to the $\mathcal{FD}$ constraint solver. This is because non-primitive constraints are first translated to primitive ones by replacing the non-primitives arguments by new fresh variables before executing constraint solving and by registering new productions between the non-primitive arguments and the new variables for lazy evaluation. Moreover, the constraint solver must protect all the produced variables of the goal in order to respect the constrained lazy evaluation and the admissibility conditions of goals. Additionally, the usual failure rules can also be used for failure detection in constraint solving and failure detection in the syntactic unification of the produced part of the goal.

Finally, we note that since Theorem 1 proves the correctness of our finite domain constraint solver w.r.t. the general framework, the main properties of the lazy narrowing calculus $CLNC(\mathcal{FD})$, *soundness* and *completeness* w.r.t. the declarative semantics of $CRWL(\mathcal{FD})$, follows directly from the general results of (López-Fraguas et al. 2004b). Obviously, these properties qualify $CLNC(\mathcal{FD})$ as a convenient computation mechanism for constraint functional logic programming over finite domains and provide a formal foundation for our $CFLP(\mathcal{FD})$ implementation $\mathcal{TOY}(\mathcal{FD})$. From the viewpoint of efficiency, a computation strategy for $CLNC(\mathcal{FD})$ using *definitional trees* (Antoy 1992) has been proposed recently in (del Vado-Vírseda 2005) and (Estévez-Martín and del Vado-Vírseda 2005) for ensuring only needed narrowing steps and extend the efficient properties shown in (Loogen et al. 1993; Antoy et al. 2000; del Vado-Vírseda 2003) guiding and avoiding *don't know* choices of constrained program rules over $\mathcal{FD}$.

3.3 Example of Goal Resolution by Using $CLNC(\mathcal{FD})$

This section is closed with a simple example which illustrates the process of goal solving via the narrowing calculus $CLNC(\mathcal{FD})$ and our finite domain constraint solver $Solve^{\mathcal{FD}}$. We compute all the answers from the goal $\Box$ *check_list* $(from\ M)$ $< 3\ \Box\ \Box$ using the $CFLP(\mathcal{FD})$-programs given in Example 2. Its resolution corresponds to the following sequence of goal transformation rules in (López-Fraguas et al. 2004b) where, at each goal transformation step, we underline which subgoal is selected. $\Vdash_{\mathbf{R}}$ indicates that the rule $\mathbf{R}$ in that work is applied.

$\Box\ \underline{check\_list\ (from\ M)\ <\ 3}\ \Box\ \Box\ \varepsilon \Vdash_{\mathbf{AC}}$
$\exists X.\ \underline{check\_list\ (from\ M) \to X}\ \Box\ \Box\ X < 3\ \Box\ \varepsilon \Vdash_{\mathbf{DF}}$

At this point, we note that X is a variable demanded by the constraint store and we have several alternatives due to *don't know* choice of the program rule *check_list*:

$\exists X.\ \underline{check\_list\ (from\ M) \to X}\ \Box\ \Box\ X < 3\ \Box\ \varepsilon \Vdash_{\mathbf{DF(CL1)}}$
$\exists X.\ from\ M \to [\,],\ \underline{0 \to X}\ \Box\ \Box\ X < 3\ \Box\ \varepsilon \Vdash_{\mathbf{SP\{X\mapsto 0\}}}$
$from\ M \to [\,]\ \Box\ \Box\ \underline{0 < 3}\ \Box\ \varepsilon \Vdash_{\mathbf{CS\{\emptyset\}}}$ $\quad (Solve^{\mathcal{FD}}(\{0 < 3\}, \emptyset) = \emptyset\ \Box\ \varepsilon)$
$\underline{from\ M \to [\,]}\ \Box\ \Box\ \Box\ \varepsilon \Vdash_{\mathbf{DF(F)}}$
$\underline{[M \mid from\ (M + 1)] \to [\,]}\ \Box\ \Box\ \Box\ \varepsilon \Vdash_{\mathbf{CF}}\ \blacksquare$

The application of the first program rule for *check_list* leads to a failure derivation without answer. We apply now the second program rule of *check_list*:

$\exists X.\ \underline{check\_list\ (from\ M) \to X}\ \Box\ \Box\ X < 3\ \Box\ \varepsilon \Vdash_{\mathbf{DF(CL2)}}$
$\exists X', Xs', X.\ from\ M \to [X'|Xs'],\ \underline{1 \to X}\ \Box$
$\qquad domain\ [X']\ 1\ 2\ \Box\ X < 3\ \Box\ \varepsilon \Vdash_{\mathbf{SP\{X\mapsto 1\}}}$
$\exists X', Xs'.\ from\ M \to [X'|Xs']\ \Box\ \underline{domain\ [X']\ 1\ 2}\ \Box\ 1 < 3\ \Box\ \varepsilon \Vdash_{\mathbf{AC}}$
$\exists X', Xs'.\ \underline{from\ M \to [X'|Xs']}\ \Box\ \Box\ 1 < 3,\ domain\ [X']\ 1\ 2\ \Box\ \varepsilon \Vdash_{\mathbf{DF(F)}}$
$\exists X', Xs'.\ \underline{[M \mid from\ (M + 1)] \to [X'|Xs']}\ \Box\ \Box\ 1 < 3,\ domain\ [X']\ 1\ 2\ \Box\ \varepsilon \Vdash_{\mathbf{DC}}$

$\exists X', Xs'.\ \underline{M \to X'},\ from\ (M+1) \to Xs' \ \Box\ \Box$
$1 < 3,\ domain\ [X']\ 1\ 2\ \Box\ \varepsilon \Vdash_{\mathbf{SP\{X' \mapsto M\}}}$
$\exists Xs'.\ \underline{from\ (M+1) \to Xs'}\ \Box\ \Box\ 1 < 3,\ domain\ [M]\ 1\ 2\ \Box\ \varepsilon \Vdash_{\mathbf{EL}}$
$\Box\ \Box\ \underline{1 < 3, domain\ [M]\ 1\ 2}\ \Box\ \varepsilon \Vdash_{\mathbf{CS(\emptyset)}}\ \Box\ \Box\ M \in [1..2]\ \Box\ \varepsilon$, because
$Solve^{\mathcal{FD}}(\{1 < 3, domain\ [M]\ 1\ 2\}, \emptyset) = \{M \in [1..2]\}\ \Box\ \varepsilon$

Therefore, we obtain the first computed answer $\Pi_1\ \Box\ \theta_1 \equiv \{M \in [1..2]\}\ \Box\ \varepsilon$. Analogously, we can apply the third program rule of *check_list*:

$\exists X.\ \underline{check\_list\ (from\ M) \to X}\ \Box\ \Box\ X < 3\ \Box\ \varepsilon \Vdash^*_{\mathbf{DF(CL3)}}$
$\Box\ \Box\ M \in [3..4]\ \Box\ \varepsilon$

and we obtain the second computed answer $\Pi_2\ \Box\ \theta_2 \equiv \{M \in [3..4]\}\ \Box\ \varepsilon$. No more answers can be computed, because if we apply the fourth program rule of *check_list* we have again a failing derivation:

$\exists X.\ \underline{check\_list\ (from\ M) \to X}\ \Box\ \Box\ X < 3\ \Box\ \varepsilon \Vdash_{\mathbf{DF(CL4)}}$
$\exists X', Xs', X.\ from\ M \to [X'|Xs'],\ \underline{4 \to X}\ \Box\ domain\ [X']\ 5\ 7\ \Box\ X < 3\ \Box\ \varepsilon$
$\Vdash_{\mathbf{SP\{X \mapsto 4\}}}$
$\exists X', Xs'.\ from\ M \to [X'|Xs']\ \Box\ \underline{domain\ [X']\ 5\ 7}\ \Box\ 4 < 3\ \Box\ \varepsilon \Vdash_{\mathbf{AC}}$
$\exists X', Xs'.\ from\ M \to [X'|Xs']\ \Box\ \Box\ \underline{4 < 3, domain\ [X']\ 5\ 7}\ \Box\ \varepsilon \Vdash_{\mathbf{SF\{X', Xs'\}}} \blacksquare$
because $Solve^{\mathcal{FD}}(\{4 < 3, domain\ [X']\ 5\ 7\}, \{X', Xs'\}) = \emptyset$

A detailed explanation of the computation of these answers using definitional trees in $CLNC(\mathcal{FD})$ to efficiently guide and avoid *don't know* choices of constrained program rules can be found in (Estévez-Martín and del Vado-Vírseda 2005). Moreover, we will see in the next section that these are exactly the same answers computed by our $CFLP(\mathcal{FD})$ implementation $\mathcal{TOY}(\mathcal{FD})$.

4 $\mathcal{TOY}(\mathcal{FD})$

So far, we have introduced the theoretical framework. Now, in this section we introduce $\mathcal{TOY}(\mathcal{FD})$, a $CFLP(\mathcal{FD})$ implementation that extends the $\mathcal{TOY}$ system to deal with $\mathcal{FD}$ constraints, highlight its advantages, and show its performance.

4.1 Introducing $\mathcal{TOY}(\mathcal{FD})$

In this section, we describe $\mathcal{TOY}(\mathcal{FD})$ from a programming point of view, briefly describing its concrete syntax and some predefined $\mathcal{FD}$ constraints.

4.1.1 An Overview of $\mathcal{TOY}(\mathcal{FD})$

$\mathcal{TOY}(\mathcal{FD})$ programs consist of *datatypes*, *type alias*, *infix operator* definitions, and rules for defining *functions*. The syntax is mostly borrowed from Haskell with the remarkable exception that variables and type variables begin with upper-case letters, whereas constructor symbols and type symbols begin with lower-case. In particular, functions are *curried* and the usual conventions about associativity of application hold. As usual in functional programming, types are inferred, checked and, optionally, can be declared in the program. To illustrate the datatype definitions, we present the following examples using the concrete syntax of $\mathcal{TOY}$:

- `data nat = zero | suc nat`, to define the naturals, and
- the Boolean predefined type as `data bool = false | true`;

A $\mathcal{TOY}(\mathcal{FD})$ program P is a set of *defining rules* for the function symbols in its signature. Defining rules for a function f have the syntactic basic form $f\ t_1\ \ldots\ t_n = r <== C$ and, informally, its intended meaning is that a call to f can be reduced to r whenever the actual parameters match the patterns t_i, and the conditions in C are satisfied. $\mathcal{TOY}(\mathcal{FD})$ also allows predicates (defined similarly as in logic programming) where predicates are viewed as a particular kind of functions, with type $p\ ::\ \overline{\tau}_n \rightarrow$ `bool`. As a syntactic facility, we can use *clauses* as a shorthand for defining rules whose right-hand side is *true*. This allows to write Prolog-like predicate definitions, so that a clause $p\ t_1\ \ldots\ t_n\ :-\ C$ abbreviates a defining rule of the form $p\ t_1\ \ldots\ t_n = true <== C$. With this sugaring in mind and some obvious changes (like currying elimination) it should be clear that (pure) $CLP(\mathcal{FD})$-programs can be straightforwardly translated to $CFLP(\mathcal{FD})$-programs.

4.1.2 Simple Programming Examples

Table 6 shows introductory programming examples in $\mathcal{TOY}$ that do not make use of the extension over $\mathcal{FD}$, together with some goals and their outcomes (López-Fraguas and Sánchez-Hernández 1999). Note that infix constraint operators are allowed in $\mathcal{TOY}(\mathcal{FD})$, such as `//` to build the expression `X // Y`, which is understood as `// X Y`. The goal *(a)* in the table sorts a list, in a pure functional computation. The answer for the goal *(b)* involves a syntactic disequality. In goal *(c)*, `F` is a higher-order logic variable, and the obtained values for this variable are higher-order patterns (`permut`, `sort`,...).

4.1.3 $\mathcal{FD}$ Constraints in $\mathcal{TOY}(\mathcal{FD})$

Table 7 shows a small subset of the $\mathcal{FD}$ constraints supported by $\mathcal{TOY}(\mathcal{FD})$, which are typical instances found in CP systems, and covers adequately the primitive constraints summarized in Table 1. In Table 7, `int` is a predefined type for integers, and `[`τ`]` is the type 'list of τ'. The datatype `labelType` is a predefined type which is used to define many search strategies for finite domain variable labeling (Fernández et al. 2004).

Relational constraint operators are applied to integers and return a Boolean value. Arithmetical constraint operators are applied to and return integer values (the set of primitive elements). They can be combined with relational constraint operators to build (non)linear (dis)equations as constraints. Moreover, reified constraints[1] can be implemented by equating a Boolean variable to a Boolean constraint, for all of the constraints built from the operators in this table and the

[1] *Reified constraints* reflect the entailment of a constraint in a Boolean variable. In general,

Table 6. $\mathcal{TOY}$ Programming Basic Examples

% Non-deterministic choice of one of two values
```
infixr 40 //
X // Y = X
X // Y = Y
```
% Non-deterministic insertion of an element into a list
```
insert X [] = [X]
insert X [Y|Ys] = [X,Y|Ys] // [Y|insert X Ys]
```
% Non-deterministic generation of list permutations
```
permut [] = []
permut [X|Xs] = insert X (permut Xs)
```
% Testing whether a list of numbers is sorted
```
sorted [] = true
sorted [X] = true
sorted [X,Y|Ys] = sorted [Y|Ys] <== X <= Y
```
% Lazy 'generate-and-test' permutation sort. 'check' calls 'sorted' which demands its
% argument, which is lazily, non-deterministically generated by 'permut'. As soon as
% the test fails, 'permut' stops the generation and tries another alternative
```
sort Xs = check (permut Xs)
check Xs = Xs <== sorted Xs == true
```

Goal	*Answers*
(a) `sort [4,2,5,1,3] == L`	`L == [1,2,3,4,5]; no more solutions`
(b) `sort [3,2,1] /= L`	`L /= [1,2,3] ; no more solutions`
(c) `F [2,1,3] == [1,2,3]`	`F == permut; F == sort; ...`

contraint `domain` (see Example 4). Due to the functional component, we can apply this technique to equate Boolean expressions to Boolean constraints, as well. Both relational and arithmetical constraint operators are syntactically distinguished (by prefixing them with #) from standard relational operators in order to denote its different operational behavior. Whereas a standard arithmetical operator demands its arguments, an arithmetical constraint does not. The membership constraint `domain` restricts a list of variables (its first argument) to have values in an integer interval (defined by its two next integer arguments) whenever its return value is `true`, whereas it restricts these variables to have values different from the interval when its return value is `false`. The enumeration constraint `labeling` assigns values to the variables in its input integer list according to the options specified with the argument of type list of `labelType`. In this list, search strategies, such as *first-*

constraints in *reified* form allow their fulfillment to be reflected back in an $\mathcal{FD}$ variable. For example, $X = (Y + Z > V)$ constrains X to *true* as soon as the disequation is known to be true and to $false$ as soon as the disequation is known to be false. On the other hand, constraining X to *true* imposes the disequation, and constraining X to $false$ imposes its negation. Usually, in $CLP(\mathcal{FD})$ languages, the Boolean values $false$ and *true* directly correspond to the numerical values 0 and 1, respectively.

Table 7. A Subset of Predefined $\mathcal{FD}$ Constraints in $\mathcal{TOY}(\mathcal{FD})$

RELATIONAL CONSTRAINT OPERATORS	
(#=), (#\=) :: int → int → bool	**(Strict Equality)**
(#<), (#<=), (#>), (#>=) :: int → int → bool	**(Less or Equal)**
ARITHMETICAL CONSTRAINT OPERATORS	
(#+), (#−), (#*), (#/) :: int → int → int	**(Operators)**
MEMBERSHIP CONSTRAINTS	
domain :: [int] → int → int → bool	**(Finite Domains)**
ENUMERATION CONSTRAINTS	
labeling :: [labelType] → [int] → bool	**(Variable Labeling)**
COMBINATORIAL CONSTRAINTS	
all_different :: [int] → bool	**(Global Constraints)**

fail (see Section 4.5.1), as well as optimization options for finding minimum and maximum values for cost functions can be specified. The combinatorial constraint `all_different` ensures different values for the elements in its list argument and is an example of the set of global constraints (for which an efficient propagation algorithm has been developed) supported by $\mathcal{TOY}(\mathcal{FD})$.

We do neither mention nor explain all the predefined constraints in detail and encourage the interested reader to visit the link proposed in (Fernández et al. 2004) for a more detailed explanation. We emphasize that all the pieces of code in this paper are executable in $\mathcal{TOY}(\mathcal{FD})$ and the answers for example goals correspond to actual executions of the programs.

4.1.4 Simple Examples with $\mathcal{FD}$ Constraints

Example 4
Below, we show the resolution at the $\mathcal{TOY}(\mathcal{FD})$ command line level of a simple goal that does not involve labeling.

```
TOY(FD)> domain [X, Y] 10 20, X #<= Y == L
         yes     L == true, X in 10..20, Y in 10..20;
         yes     L == false, X in 11..20, Y in 10..19;
         no
```

Also note that this $CFLP(\mathcal{FD})$ implementation only inform about a limited outcome, which consists of: (1) substitutions of the form `Variable == Pattern`, (2) disequality constraints `Variable /= Pattern`, (3) disjunctions D of constraints `Variable in IntegerRange` (these constraints denote the possible values a variable *might* take, as in common constraint systems; i.e., they do not state D, but negated D), and (4) success information: `yes` and `no` stand for *success* and *failure*, respectively. Finally, '`;`' separates the solutions which has been explicitly requested by the user. Primitive constraints in the finite domain constraint store are not shown.

Example 5
We show a $\mathcal{TOY}(\mathcal{FD})$ program involving labeling to solve the classical N-queens problem whose objective is to place N queens on an $N \times N$ chessboard so that there are no threatening queens.

```
include "misc.toy"
include "cflpfd.toy"

queens :: int -> [labelType] -> [int]
queens N Label = L <== length L == N, domain L 1 N,
                       constrain_all L, labeling Label L

constrain_all :: [int] ->  bool
constrain_all [] = true
constrain_all [X|Xs] = true <== constrain_between X Xs 1,
                       constrain_all Xs

constrain_between :: int -> [int] -> int -> bool
constrain_between X [] N = true
constrain_between X [Y|Ys] N = true <== no_threat X Y N,
                       constrain_between X Ys (N+1)

no_threat:: int -> int -> int -> bool
no_threat X Y I = true <== X #\= Y, X #+ I #\= Y, X #- I #\= Y
```

The intended meaning of the functions should be clear from their names and definitions, provided that `length L` returns the length of the list `L`. The first two lines are needed to include predefined functions such as `length` and `domain`. An example of solving at the command prompt, where `ff` stands for the first-fail enumeration strategy (see Section 4.5.1), is

```
TOY(FD)> queens 15 [ff] == L
         yes    L ==  [1,3,5,14,11,4,10,7,13,15,2,8,6,9,12]
```

Example 6
We present a $\mathcal{TOY}(\mathcal{FD})$ program using syntactic sugaring for predicate-like functions that solves the well-known $CLP(\mathcal{FD})$ program *Send+More=Money*.

```
smm :: int -> int -> int -> int -> int -> int -> int -> int
       -> [labelType] -> bool

smm S E N D M O R Y Label :-
       domain [S,E,N,D,M,O,R,Y] 0 9, S #> 0, M #> 0,
       all_different [S,E,N,D,M,O,R,Y], add S E N D M O R Y,
       labeling Label [S,E,N,D,M,O,R,Y]

add :: int -> int -> int -> int -> int -> int -> int -> int -> bool
add S E N D M O R Y :-  1000#*S #+ 100#*E #+ 10#*N #+ D
                     #+ 1000#*M #+ 100#*O #+ 10#*R #+ E
        #=  10000#*M #+ 1000#*O #+ 100#*N #+ 10#*E #+ Y
```

For our simple $\mathcal{TOY}(\mathcal{FD})$ programs, some examples of goals and answers which can be computed by $\mathcal{TOY}(\mathcal{FD})$ are shown in Table 8.

Table 8. Examples of Goal Solving

Goal	*Answers*
`domain [A,B] 1 (1+2), A#>B,` `all_different [A,B], labeling [ ] [A,B]`	`A==2,B==1; A==3,B==1;` `A==3,B==2; no more solutions`
`domain [X,Y,Z] 1 10,` `2 #* X #+ 3 #* Y #+ 2 #< Z`	`X in 1..2,Y==1,Z in 8..10;` `no more solutions`
`domain [X,Y,Z] 1 5, X #> Y,` `2 #* Y #> Z #+ 4, X #>= Z`	`X in 4..5,Y in 3..4,Z in 1..3;` `no more solutions`
`smm S E N D M O R Y [] == T`	`S==9,E==5,N==6,D==7,M==1,O==0,` `R==8,Y==2,T==true;` `no more solutions`
`queens 5 [] == [M,A,E,Y,B],` `smm S E N D M O R Y []`	`M==1,A==3,E==5,Y==2,B==4,S==9,` `N==6,D==7,O==0,R==8;` `no more solutions`

4.2 $CFLP(\mathcal{FD})$ ***vs.*** *$CLP(\mathcal{FD})$*

It is commonly acknowledged that $CLP(\mathcal{FD})$ is a successful declarative approach; hence, we discuss the advantages of $CFLP(\mathcal{FD})$, focusing on the $\mathcal{TOY}(\mathcal{FD})$ implementation, with respect to $CLP(\mathcal{FD})$. This section explains why the addition of FP features enhances the CLP setting. When necessary, we illustrate different features of $CFLP(\mathcal{FD})$ by means of examples. Further programming examples in pure functional logic programming and $CFLP(\mathcal{FD})$ can be found, respectively, in (López-Fraguas and Sánchez-Hernández 1999) and (Fernández et al. 2004).

4.2.1 $CFLP(\mathcal{FD}) \supset CLP(\mathcal{FD})$

As already pointed out, besides other features, $CFLP(\mathcal{FD})$ provides the main characteristics of $CLP(\mathcal{FD})$, i.e., $\mathcal{FD}$ constraint solving, non-determinism and relational form. Moreover, $CFLP(\mathcal{FD})$ provides a sugaring syntax for LP predicates and thus, as already commented, any pure $CLP(\mathcal{FD})$-program can be straightforwardly translated into a $CFLP(\mathcal{FD})$-program. In this sense, $CLP(\mathcal{FD})$ may be considered as a strict subset of $CFLP(\mathcal{FD})$ with respect to problem formulation. As a direct consequence, our language is able to cope with a wide range of applications (at least with all those applications that can be formulated with a $CLP(\mathcal{FD})$ language). We will not insist here on this matter, but prefer to concentrate on the extra capabilities of $CFLP(\mathcal{FD})$ with respect to $CLP(\mathcal{FD})$.

4.2.2 $CFLP(\mathcal{FD}) \setminus CLP(\mathcal{FD})$

Due to its functional component, $CFLP(\mathcal{FD})$ adds further expressiveness to $CLP(\mathcal{FD})$ as allows the declaration of functions and their evaluation in the FP style. In the following, we enumerate and discuss other features not present (or unusual) in the $CLP(\mathcal{FD})$ paradigm.

Types. Our language is strongly typed and thus involves all the well-known advantages of a type checking process, enhancing program development and maintenance. Each $\mathcal{FD}$ constraint has associated, like any function, a type declaration, which means that a wrong use can be straightforwardly detected in the typical type checking process.

Functional Notation. It is well-known that functional notation reduces the number of variables with respect to relational notation, and thus, $CFLP(\mathcal{FD})$ increases the expressiveness of $CLP(\mathcal{FD})$ as it combines relational and functional notation. For instance, in $CLP(\mathcal{FD})$ the constraint conjunction `N=2, X ∈ [1,10-N]` cannot be expressed directly and must be written adding a third component, as `N=2, Max is 10-N, domain([X],1,Max)` that uses an extra variable. However, $\mathcal{TOY}(\mathcal{FD})$ expresses that constraint directly as `N==2, domain [X] 1 (10-N)`.

Currying. Again, due to its functional component, $\mathcal{TOY}(\mathcal{FD})$ allows curried functions (and thus constraints); for instance, see the application of curried $\mathcal{FD}$ constraint `(3#<)`/1 in Example 7 later in this section.

Higher-Order and Polymorphism. In $\mathcal{TOY}(\mathcal{FD})$ functions are first-class citizens, which means that a function (and thus an $\mathcal{FD}$ constraint) can appear in any place where data do. As a direct consequence, an $\mathcal{FD}$ constraint may appear as an argument (or even as a result) of another function or constraint. The functions managing other functions are called higher-order (HO) functions. Also, polymorphic arguments are allowed in $CFLP(\mathcal{FD})$.

Example 7
A traditional example of a polymorphic HO function is

```
map :: (A -> B) -> [A] -> [B]
map F [] = []
map F [X|Xs] = [(F X) | (map F Xs)]
```

that receives both a function F and a list as arguments and produces a list resulting from applying the function to each element in the list. Now, suppose that X and Y are $\mathcal{FD}$ variables ranging in the domain [0..100] (expressed, for instance, via the constraint `domain [X,Y] 0 100`). Then, the goal `map (3#<) [X,Y]` returns the Boolean list `[true,true]` resulting from evaluating the list `[3#<X,3#<Y]`, and `X` and `Y` are also restricted to have values in the range `[4,100]` as the constraints `3#<X` and `3#<Y` are sent to the constraint solver. Note also the use of the curried function `(3#<)`.

Laziness. In contrast to logic languages, functional languages support *lazy evaluation*, where function arguments are evaluated to the required extend (the *call-by-value* used in LP vs. the call-by-need used in FP). Strictly speaking, lazy evaluation may also correspond to the notion of *only once evaluated* in addition to *only required extent* (Peyton-Jones 1987). $\mathcal{TOY}(\mathcal{FD})$ increases the power of $CLP(\mathcal{FD})$ by incorporating a novel mechanism that combines *lazy evaluation* and $\mathcal{FD}$ constraint solving, in such a way that only the demanded constraints are sent to the solver. This is a powerful mechanism that opens new possibilities for $\mathcal{FD}$ constraint solving. For example, in contrast to $CLP(\mathcal{FD})$, it is possible to manage infinite structures.

Example 8
Consider the recursive functions `take` and `generateFD` from Example 2. An eager evaluation of the following goal does not terminate as it tries to completely evaluate the second argument, yielding to an infinite computation. However, a lazy evaluation generates just the first 3 elements of the list, as shown below:

```
TOY(FD)> take 3 (generateFD 10) == List
         yes    List ==  [ _A, _B, _C ]   _A, _B, _C in 0..9
```

In general, lazy narrowing avoids computations which are not demanded, therefore saving computation time. Example 9 contains a formulation of the typical magic series (or sequences) problem (Van Hentenryck 1989). This example highlights the expressive power of $\mathcal{TOY}(\mathcal{FD})$ by solving multiple problem instances that can be described and solved via lazy evaluation of infinite lists.

Example 9
Let `S = (`s_0`, `s_1`, ...,`s_{N-1}`)` be a non-empty finite series of non-negative integers. The series `S` is said *N-magic* if and only if there are s_i occurrences of `i` in `S`, for all `i` $\in$ `{0,...,N-1}`. Below, we propose a $\mathcal{TOY}(\mathcal{FD})$ program to calculate magic series where the function `generateFD` is as defined in Example 2.

```
lazymagic :: int -> [int]
lazymagic N = L <== take N (generateFD  N) == L,
                    constrain L  L  0  Cs, sum L (#=) N,
                    scalar_product Cs  L (#=) N, labeling [ff] L

constrain :: [int] -> [int] -> int -> [int] -> bool
constrain [] A B  [] = true
constrain [X|Xs] L I [J|Js] = true <== I==J, count I L (#=) X,
                                        constrain Xs L (I+1) Js
```

`sum/3`, `scalar_product/4` and `count/4` are predefined HO constraints (Fernández et al. 2004), that accept a relational $\mathcal{FD}$ constraint operator with type `int` → `int` → `bool` as argument (e.g., the constraint `#=`). `sum L C N` means that the summation of the elements in the list `L` is related through `C` with the integer `N` (in the example, the summation is constrained to be equal to `N`). `scalar_product` and `count` stand for scalar product and element counting under the same parameters as `sum`.

A goal `lazymagic N`, for some natural N, returns the N-magic series where the condition `take N (generateFD N)` is evaluated lazily as `(generateFD N)` produces an infinite list. More interesting is to return a list of different solutions starting from N. This can be done using a recursive definition to produce the infinite list of magic series `(from N)` as shown below.

```
magicfrom :: int -> [[int]]
magicfrom N = [lazymagic N | magicfrom (N+1)]
```

Now, it is easy to generate a list of magic series by lazy evaluation. For example, the following goal generates a 3-element list containing, respectively, the solution to the problems of 7-magic, 8-magic and 9-magic series.

```
TOY(FD)> take 3 (magicfrom 7) == L
          yes      L ==  [ [ 3, 2, 1, 1, 0, 0, 0 ],
                           [ 4, 2, 1, 0, 1, 0, 0, 0 ],
                           [ 5, 2, 1, 0, 0, 1, 0, 0, 0 ] ]
```

More expressiveness is shown by mixing curried functions, HO functions and function composition (another nice feature from the functional component of $\mathcal{TOY}(\mathcal{FD})$). For example, consider the $\mathcal{TOY}(\mathcal{FD})$ code shown below:

```
from :: int -> [int]
from N = [N | from (N+1)]

(.):: (B -> C) -> (A -> B) -> (A -> C)
(F . G) X = F (G X)

lazyseries :: int -> [[int]]
lazyseries = map lazymagic . from
```

where `(.)/2` defines the composition of functions. Observe that *lazyseries* curries the composition `(map lazymagic) . from`. Then, it is easy to generate the 3-element list shown above by just typing the goal

```
TOY(FD)> take 3 (lazyseries 7) == L
```

This simple example gives an idea of the nice features of $CFLP(\mathcal{FD})$ that combines $\mathcal{FD}$ constraint solving, management of infinite lists and lazy evaluation, curried notation of functions, polymorphism, HO functions (and thus HO constraints), composition of functions and a number of other characteristics that increase the potentialities with respect to $CLP(\mathcal{FD})$.

4.3 Correctness of the CFLP(FD) Implementation

In this section, we briefly discuss the correctness of our $\mathcal{TOY}(\mathcal{FD})$ implementation with respect to our $CFLP(\mathcal{FD})$ framework.

$\mathcal{TOY}(\mathcal{FD})$ integrates, as a host language, the higher-order lazy functional logic language $\mathcal{TOY}$ (López-Fraguas and Sánchez-Hernández 1999) and, as constraint solver, the efficient $\mathcal{FD}$ constraint solver of SICStus Prolog (Carlsson et al. 1997). Under the condition of considering just an empty set of protected variables, the SICStus Prolog finite domain solver always satisfies the conditions for constraint solvers required in Section 2.5. Since the $CLNC(\mathcal{FD})$ calculus is *strongly complete* (see (López-Fraguas et al. 2004b)) in the sense that the choice of goal transformation rules can be a *don't care* choice, in practice, we can choose a suitable demand-driven strategy: our $\mathcal{FD}$ constraint solver is only applied at the end of the process of goal solving, when we have an empty set of protected variables (as we have done in the example in Section 3.3) or when protected variables are not relevant. This strategy can be performed in the $CLNC(\mathcal{FD})$ calculus in the line of (del Vado-Vírseda 2005) as well as in $\mathcal{TOY}(\mathcal{FD})$ in the line of (Estévez-Martín and del Vado-Vírseda 2005). Therefore, we can conclude that our operational semantics with this strategy covers adequately the $\mathcal{TOY}(\mathcal{FD})$ implementation.

Additionally, we have run a number of tests in the implementation and have compared the derivations produced by the calculus $CLNC(\mathcal{FD})$ to the traces obtained from debugging in $\mathcal{TOY}(\mathcal{FD})$, and the results show that these are effectively identical by following an adequate demand-driven strategy in $CLNC(\mathcal{FD})$. For instance, the $CFLP(\mathcal{FD})$ program shown in Example 2 corresponds almost directly to a $\mathcal{TOY}(\mathcal{FD})$ program, and the solving of the goal $check\_list\ (from\ M)\ <\ 3$ in $\mathcal{TOY}(\mathcal{FD})$ is shown below (see (Estévez-Martín and del Vado-Vírseda 2005) for more details).

```
Toy(FD)> check_list (from M) < 3
      yes
      M in 1..2
      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
      yes
      M in 3..4
      Elapsed time: 0 ms.

more solutions (y/n/d) [y]?
      no.
      Elapsed time: 0 ms.
```

Note that the computed answers correspond exactly to those obtained in the goal solving process described in Section 3.3 via the narrowing calculus $CLNC(\mathcal{FD})$.

4.4 Notes about the Implementation

In $\mathcal{TOY}(\mathcal{FD})$, $\mathcal{FD}$ constraints are evaluated internally by using mainly two predicates: `hnf(E,H)`, which specifies that `H` is one of the possible results of narrowing the expression `E` into head normal form, and `solve`/1, which checks the satisfiability of constraints (of rules and goals) before the evaluation of a given rule. This predicate is, basically, defined as follows[2]:

$$
\begin{array}{lll}
(1)\ \texttt{solve}((\varphi,\varphi')) & :- & \texttt{solve}(\varphi), \texttt{solve}(\varphi'). \\
(2)\ \texttt{solve}(\texttt{L} == \texttt{R}) & :- & \texttt{hnf}(\texttt{L},\texttt{L}'), \texttt{hnf}(\texttt{R},\texttt{R}'), \texttt{equal}(\texttt{L}',\texttt{R}'). \\
(3)\ \texttt{solve}(\texttt{L} \;/= \texttt{R}) & :- & \texttt{hnf}(\texttt{L},\texttt{L}'), \texttt{hnf}(\texttt{R},\texttt{R}'), \texttt{notequal}(\texttt{L}',\texttt{R}'). \\
(4)\ \texttt{solve}(\texttt{L}\#\Diamond\ \texttt{R}) & :- & \texttt{hnf}(\texttt{L},\texttt{L}'), \texttt{hnf}(\texttt{R},\texttt{R}'), \{\texttt{L}'\#\Diamond\texttt{R}'\}. \\
 & & \quad \text{where } \Diamond \in \{=, \backslash=, <, <=, >, >=\}. \\
(5)\ \texttt{solve}(\texttt{C}\ \texttt{A}_1 \ldots \texttt{A}_n) & :- & \texttt{hnf}(\texttt{A}_1,\texttt{A}_1'), \ldots, \texttt{hnf}(\texttt{A}_n,\texttt{A}_n'), \{\texttt{C}(\texttt{A}_1', \ldots, \texttt{A}_n')\}. \\
 & & \quad \text{where } \texttt{C} \text{ is any constraint returning a Boolean.}
\end{array}
$$

The interaction with the constraint solver (i.e., SICStus $\mathcal{FD}$ constraint solver in the current $\mathcal{TOY}(\mathcal{FD})$ version) is reflected in the last two clauses: every time an $\mathcal{FD}$ constraint appears, the solver is eventually invoked with a goal $\{G\}$ where G is the translation of the $\mathcal{FD}$ constraint from $\mathcal{TOY}(\mathcal{FD})$ to SICStus Prolog. Head normal forms are required for constraint arguments in order to allow the solver to solve the constraint.

4.5 Performance

As far as we know, $\mathcal{TOY}(\mathcal{FD})$ was the first *FLP* system integrating a $\mathcal{FD}$ constraint system. However, we know about the existence of an implementation of the *FLP* language Curry (Hanus 1999) that supports a limited set of $\mathcal{FD}$ constraints (Hanus M. (editor) 2005). This implementation, called PAKCS, provides the following constraints:

(1) a set of arithmetical operations {`*#`,`+#`,`-#`,`=#`,`/=#`,`<#`,`<=#`,`>#`,`>=#`},
(2) a membership constraint `domain` /3,
(3) some global constraints[3] and
(4) an enumeration constraint `labeling` /1 that also provides searching options.

In this section, we compare the performance of $\mathcal{TOY}(\mathcal{FD})$ with that of the *Curry2Prolog* compiler, which is the most efficient implementation of Curry inside PAKCS (version 1.7.1 of December 2005).

In addition, for evaluating if $\mathcal{TOY}(\mathcal{FD})$ is competitive with respect to existing $CLP(\mathcal{FD})$ systems, we have also considered four well-known $CLP(\mathcal{FD})$ systems:

1. The version 3.12.1 of April 2005 of the $\mathcal{FD}$ constraint solver of SICStus Prolog (Carlsson et al. 1997; SICStus Prolog 2005). This solver was included in order to measure the overhead due to the management of functional logic

[2] The code does not correspond exactly to the implementation, which is the result of many transformations and optimizations.
[3] Exactly, those named in this paper, i.e. `all_different`/1, `count`/4, `scalar_product`/4 and `sum`/3.

expressions, which are compiled to SICStus Prolog in $\mathcal{TOY}(\mathcal{FD})$, and, therefore, including all the stuff needed to handle the FLP characteristics such as laziness and higher-order functions.

2. The GNU Prolog system (version 1.2.16) (Diaz and Codognet 2001; GNU Prolog 2005), which is a free Prolog compiler that includes one of the most efficient finite domain constraint solver. This solver is based on the concept of *indexicals* (Codognet and Diaz 1996) and it has been demonstrated that it has a performance comparable to commercial systems.
3. SWI-Prolog (version 5.4.x) (Wielemaker 2003; SWI-Prolog 2005) that it is an emergent and very promising Prolog system that provides an integer domain constraint solver implemented with attributed variables.
4. Ciao Prolog (version 1.10#5 of August 2004) which is a full multi-paradigm programming environment for developing programs in the Prolog language and in several other languages which are extensions and modifications of Prolog in several interesting and useful directions. Ciao Prolog provides a package, based upon the indexical concept, to write and evaluate constraint programming expressions over finite domains in a Ciao program.

4.5.1 Labeling

Constraint solving can be implemented with a combination of two processes: constraint propagation and labeling (i.e., search) (Dechter 2003). The labeling process consists of (1) choosing a variable (variable ordering) and (2) assigning to the variable a value which belongs to its domain (value ordering). The variable ordering and the value ordering used for the labeling can considerably influence the efficiency of the constraint solving when only one solution to the problem is required. It has little effect when the search is for all solutions. In this study, we consider two labelings, the naïve labeling that chooses the leftmost variable of a list of variables and then selects the smallest value in its domain, and the *first-fail* labeling that uses a principle (Haralick and Elliot 1980) which says that *to succeed, try first where you are the most likely to fail.* This principle recommends the choice of the most constrained variable, which often means (for the finite domain) choosing a variable with the smallest domain. The naïve labeling assures that both variable and value ordering are the same for all the systems and hence (although less efficient) is better for comparing the different systems when only one solution is required.

4.5.2 The Benchmarks

We have used a wide set of benchmarks[4] and, for the sake of fairness, whenever it was possible, we used exactly the same formulation of the problems for all systems as well as the same $\mathcal{FD}$ constraints. The benchmarks used are:

- **cars**: solve a car sequencing problem with 10 cars (Dincbas et al. 1988). This benchmark deals with 100 Boolean variables (i.e., finite domain variables

[4] All the programs used in the comparison are available at *http://www.lcc.uma.es/∼afdez/cflpfd/*.

ranging over [0,1]), 10 finite domain variables ranging over [1,6], 6 *atmost* constraints, 50 *element* constraints, and 49 linear disequations.

- **equation 10**: a system of 10 linear equations with 7 variables ranging over [0,10].
- **equation 20**: a system of 20 linear equations with 7 variables ranging over [0,10].
- **magic series (N)**: calculate a series of N numbers such that each of them is the number of occurrences in the series of its position in the series (Codognet and Diaz 1996).
- **optimal Golomb ruler (N)**: find an ordered set of n distinct non-negative integers, called *marks*, $a_1 < ... < a_n$, such that all the differences $a_i - a_j$ $(i > j)$ are distinct and a_n is minimum (Shearer 1990).
- **queens (N)**: place N queens on a $N \times N$ chessboard such that no queen attacks each other (Van Hentenryck 1989).
- **pythagoras**: calculate the proportions of a triangle by using the Pythagorean theorem. This problem involves 3 variables ranging over [1,1000], and 7 disequality (non-linear) equations.
- **sendmore**: a cryptoarithmethic problem with 8 variables ranging over [0,9], with one linear equation, 2 disequations and 28 inequality constraints (or alternatively one *all_different* constraint imposed over the whole set of constrained variables). It consists of solving the equation $SEND + MORE = MONEY$.
- **suudoku**: the problem is to fill partially filled 9x9 squares of 81 squares such that each row and column are permutations of [1,...,9], and each 3x3 square, where the leftmost column modulo 3 is 0, is a permutation of [1,...,9].

The programs **equation 10**, **equation 20** and **sendmore** test the efficiency of the systems to solve linear equation problems. The programs **cars** and **suudoku** check the efficiency of specialized constraints such as the *all_different* constraint. The **pythagoras** problem deals with non-linear equations.

The **queens** and **magic series** programs are scalable and therefore useful to test how the systems work for bigger instances of the same problem. Note that both the number of variables and the number of values for each variable grow linearly with the parameter N in the examples. That is, given a value N, at least N $\mathcal{FD}$ variables must be declared with domains that range between 0 or 1 and N.

The search for **optimal Golomb rulers** is an extremely difficult task as it is a combinatorial problem whose bounds grow geometrically with respect to the solution size (Shearer 1990). This (also scalable) benchmark allows us to check the optimization capabilities of the system.

4.5.3 Results

All the benchmarks were performed on the same Linux machine (under Fedora Core system, 2.69-1667) with an Intel(R) Pentium 4 processor running at 2.40 GHz and with a RAM memory of 512 Mb. For the sake of brevity, we only provide the results for first solution search.

Table 9. $\mathcal{TOY}(\mathcal{FD})$ vs. C(F)LP Systems: Naïve Labeling

Benchmark	$\mathcal{TOY}(\mathcal{FD})$	PAKCS	*SICStus*	SWI	GNU	Ciao
cars	5	N	5	N	1	N
equation10	20	50	10	590	2	-
equation20	35	60	10	1185	4	-
magic(64)	265	340	(430) 260	N (OGS)	134	N
magic(100)	910	980	(1520) 900	N (OGS)	901	N
magic(150)	2700	3180	(4770) 2560	N (OGS)	(SO) 4894	N
magic(200)	5970	6540	(10870) 5690	N (OGS)	(SO) 14570	N
magic(300)	18365	22750	(RE) 17780	N (OGS)	(SO) 68020	N
pythagoras	50	80	20	940	10	902
queens(8)	10	20	10	110	1	31
queens(16)	180	200	170	38720	11	6873
queens(20)	4030	4200	3930	1064130	216	190435
queens(24)	8330	8400	8120	??	460	576625
queens(30)	1141760	1141940	1069750	??	67745	??
sendmore	0	5	0	15	0	14
suudoku	10	20	10	60	1	51

Table 10. Speed-Up of $\mathcal{TOY}(\mathcal{FD})$ wrt. other C(F)LP Systems for Naïve Labeling

Benchmark	PAKCS	*SICStus*	SWI	GNU	Ciao
cars	∞	1 00	∞	0.20	∞
equation10	2.50	0.50	29.50	0.10	∞
equation20	1.71	0.28	33.85	0.11	∞
magic (64)	1.28	(1.62) 0.98	∞	0.50	∞
magic (100)	1.07	(1.67) 0.98	∞	0.99	∞
magic (150)	1.17	(1.76) 0.98	∞	(∞) 1.81	∞
magic (200)	1.09	(1.82) 0.99	∞	(∞) 2.44	∞
magic (300)	1.23	(∞) 0.96	∞	(∞) 3.70	∞
pythagoras	1.60	0.40	18.80	0.20	18.04
queens (8)	2.00	1.00	11.00	0.10	3.12
queens (16)	1.11	0.94	215.11	0.06	38.18
queens (20)	1.04	0.97	264.05	0.05	42.25
queens (24)	1.00	0.97	(?)	0.05	69.22
queens (30)	1.00	0.93	(?)	0.05	(?)
sendmore	$\geq$ 5.00	$\geq$ 1.00	$\geq$ 15.00	$\geq$ 1.00	$\geq$ 14.00
suudoku	2.00	1.00	6.00	0.10	5.10

Table 9 shows the results using naïve labeling. The meaning for the columns is as follows. The first column gives the name of the benchmark used in the comparison, and the next six columns show the running (elapsed) time (measured in milliseconds) to find the first answer of the benchmark for each system.

Table 10 shows the results shown in Table 9 in terms of the speed-up of $\mathcal{TOY}(\mathcal{FD})$

Table 11. $\mathcal{TOY}(\mathcal{FD})$ vs. C(F)LP systems: First-Fail Labeling

Benchmark	$\mathcal{TOY}(\mathcal{FD})$	PAKCS	*SICStus*	SWI	GNU	Ciao
cars	0	N	0	N	0	N
equation10	20	50	10	N	3	N
equation20	30	55	15	N	4	N
magic (64)	90	150	(320) 80	N	18	N
magic (100)	220	310	(1090) 195	N	53	N
magic (150)	470	690	(3440) 465	N	(SO) 52	N
magic (200)	870	1480	(7950) 850	N	(SO) 125	N
magic (300)	1835	3610	(RE) 1820	N	(SO) 568	N
magic (400)	3420	10050	(RE) 3370	N	(SO) 1088	N
magic (500)	5510	13100	(RE) 5250	N	(SO) 1830	N
pythagoras	50	80	10	N	10	N
queens (8)	10	15	5	N	1	N
queens (16)	20	50	8	N	2	N
queens (20)	45	75	10	N	3	N
queens (24)	40	80	15	N	4	N
queens (30)	150	190	25	N	6	N
sendmore	0	5	0	N	0	N
suudoku	10	20	10	N	1	N

with respect to the rest of the systems (that is, the result of dividing the time of a given system by the time of $\mathcal{TOY}(\mathcal{FD})$).

Table 11 shows the results of solving the same benchmarks by using first-fail labeling. Note that the current versions of SWI Prolog and Ciao Prolog do not provide first-fail labeling. Also, Table 12 shows the speed-up corresponding to the results in Table 11 and again displays the performance of $\mathcal{TOY}(\mathcal{FD})$ with respect to the rest of the systems. The meaning for the columns is as in Table 10, but a last column is added in order to show the speed-up of $\mathcal{TOY}(\mathcal{FD})$ using first-fail labeling with respect to the same system with naïve labeling.

Tables 13 and 14 display corresponding results for optimization. Particularly, Table 13 shows the (elapsed) time measured in milliseconds to solve the optimization problem considered in the benchmarking process, whereas Table 14 shows the speed-up of our system with respect to the rest of the systems.

In these tables, all numbers represent the average of ten runs. The symbol ?? means that we did not receive a solution for the benchmark in a reasonable time and (?) indicates a non-determined value. The symbol **N** in the PAKCS, SWI Prolog and Ciao Prolog columns mean that we could not formulate the benchmark because of insufficient provision for constraints.

Also the notation *OGS* in the SWI column indicates that we received an error of *Out Of Global Stack* and, consequently, no answer was returned. In the GNU Prolog column, the notation *(SO) number* means that, in the first execution of the program no answer was calculated because a *Stack Overflow* error was raised, and that, after increasing significantly the corresponding (cstr and trail) environment variables, in

Table 12. Speed-Up of $\mathcal{TOY}(\mathcal{FD})$ wrt. other C(F)LP Systems for First-Fail Labeling

Benchmark	PAKCS	*SICStus*	SWI	GNU	Ciao	$\mathcal{TOY}(\mathcal{FD})$ (naïve)
cars	∞	$\geq$ 1.00	∞	$\geq$ 1.00	∞	$\geq$ 5.00
equation10	2.50	0.50	∞	0.15	∞	1.00
equation20	1.83	0.50	∞	0.13	∞	1.16
magic (64)	1.66	(3.55) 0.88	∞	0.20	∞	2.94
magic (100)	1.40	(4.95) 0.88	∞	0.24	∞	4.13
magic (150)	1.46	(7.31) 0.98	∞	(∞) 0.11	∞	5.74
magic (200)	1.70	(9.13) 0.97	∞	(∞) 0.14	∞	6.86
magic (300)	1.96	(∞) 0.99	∞	(∞) 0.30	∞	10.00
magic (400)	2.93	(∞) 0.98	∞	(∞) 0.31	∞	(?)
magic (500)	2.37	(∞) 0.95	∞	(∞) 0.33	∞	(?)
pythagoras	1.60	0.20	∞	0.20	∞	1.00
queens (8)	1.50	0.50	∞	0.10	∞	1.00
queens (16)	2.50	0.40	∞	0.10	∞	9.00
queens (20)	1.66	0.22	∞	0.06	∞	89.55
queens (24)	2.00	0.37	∞	0.10	∞	208.25
queens (30)	1.26	0.16	∞	0.04	∞	7611.73
sendmore	$\geq$ 5.0	$\geq$ 1.00	∞	$\geq$ 1.00	∞	$\geq$ 1.00
suudoku	2.00	1.00	∞	0.10	∞	1.00

Table 13. $\mathcal{TOY}(\mathcal{FD})$ vs. C(F)LP Systems: Optimization Benchmarks

Benchmark	$\mathcal{TOY}(\mathcal{FD})$	PAKCS	*SICStus*	SWI	GNU	Ciao
golomb(8)	360	350	280	N	86	N
golomb(10)	26230	27500	25730	N	8595	N
golomb(12)	5280170	5453220	5208760	N	2162863	N

further executions we obtained an answer in the (average) time indicated by *number*. The notation *RE* in the SICStus Prolog column indicates that we also did not compute an answer because a *Resource Error by Insufficient Memory* was returned. The dash (-) in the Ciao Prolog column means that we received an incorrect answer for this benchmark[5].

As already declared, whenever possible we maintained the same formulation for all the benchmarks in each system. However, this was not always possible in the magic series benchmark. In the $\mathcal{TOY}(\mathcal{FD})$, PAKCS and SICStus Prolog systems, this problem was coded by using specific constraints (i.e., *count*/4, *sum*/3 and *scalar_product*/4 - see formulation in Example 9). However, the GNU Prolog system

[5] This event seems to be caused by a bug existing in the $\mathcal{FD}$ constraint package.

Table 14. Speed-Up of $\mathcal{TOY}(\mathcal{FD})$ wrt. other C(F)LP Systems for Optimization Benchmarks

Benchmark	*SICStus*	PAKCS	SWI	GNU	Ciao
golomb(8)	0.77	0.97	∞	0.23	∞
golomb(10)	0.98	1.04	∞	0.32	∞
golomb(12)	0.98	1.03	∞	0.40	∞

lacks these constraints and, therefore, we used a classical formulation that requires to use reified constraints (Codognet and Diaz 1996). This classical formulation is somewhat different in $\mathcal{TOY}(\mathcal{FD})$ since reification applies to Boolean types (whilst in GNU Prolog, as in general in $CLP(\mathcal{FD})$ languages, the Boolean values $false$ and $true$ correspond to the numerical values 0 and 1 respectively). On the other hand, it was not possible in PAKCS as reified constraints are not available in this system. However, since SICStus Prolog allows reified constraints, the two formulations were considered in this system. Then, in the SICStus column and for the magic series benchmark row, we show between parentheses the (elapsed solving) time associated with the reified constraints-based formulation followed by the time associated to the alternative formulation based on the use of specific constraints.

In the speed-up tables, in those cases in which for a particular system either a problem could not be expressed (e.g., for PAKCS, SWI Prolog or Ciao Prolog), or an error was returned avoiding to compute a first answer, or an incorrect answer was returned, we use the symbol ∞ to express that our system clearly outperforms that system since our system provides constraint support to formulate a solution for the benchmark and compute an answer. Also, a result $\geq$ *x.00* indicates that $\mathcal{TOY}(\mathcal{FD})$ computed an answer in 0.0 milliseconds and thus no speed-up can be calculated; in these cases, *x.00* indicates that $\mathcal{TOY}(\mathcal{FD})$ is, at least, x times faster than the compared system.

4.5.4 Analysis of the Results

The third column in Tables 10 and 12, and column 2 in Table 14 show that, in general, our implementation behaves closely to that of SICStus Prolog in both constraint satisfaction and constraint optimization (in fact, this is not surprising as current version of $\mathcal{TOY}(\mathcal{FD})$ uses SICStus Prolog $\mathcal{FD}$ solver) except for solving linear equations (in these cases it is between two and four times slower). The reason seems to be in the transformation process previous to the invocation of the $\mathcal{FD}$ solver. Expressions have to be transformed into head normal form, which means that their arguments are also transformed into head normal form (see Section 4.4). Thus, there seems to be an overhead when expressions (such as those for linear equations) involve a high number of arguments and sub-expressions. This may be the same reason argued to explain the slow-down of $\mathcal{TOY}(\mathcal{FD})$ in the solving of the queens benchmark via first-fail labeling, although no appreciable slow-down was shown in the solving via naïve labeling.

PAKCS is between one and three times slower than our implementation. This is quite interesting as the PAKCS implementation is fairly efficient and is also based on the SICStus Prolog $\mathcal{FD}$ library. Perhaps the reason of this slowdown with respect to $\mathcal{TOY}(\mathcal{FD})$ is that PAKCS implements an alternative operational model that also supports concurrency, and this model introduces some kind of overhead in the solving of goals.

$\mathcal{TOY}(\mathcal{FD})$ also performs reasonably well compared to the other $CLP(\mathcal{FD})$ systems. It clearly outperforms both Ciao Prolog's and SWI Prolog's constraint solvers which are far, in their current versions, from the efficiency of $\mathcal{TOY}(\mathcal{FD})$ in the solving of constraint satisfaction problems (for fairness, we have to say that these results cannot be extrapolated to the whole Ciao Prolog and SWI Prolog systems which are quite efficient; in fact, the integer bounds constraint solver of SWI Prolog seems to be a rather non-optimized simple integer constraint solver that probably will be largely improved in future versions. This same argument can be applied to the finite domain constraint solving package currently existing in the Ciao Prolog system that seems to be non-mature yet). With respect to GNU Prolog's constraint solver, our system behaves acceptably well if we take into account that this solver has shown an efficiency comparable to commercial systems. Except for the N-queens benchmark (that seems to be particularly optimized for GNU solver) our system is in the same order of efficiency. Moreover, it even behaves better on scalable problems when the size of the problem increases (e.g., in the magic series problem with naïve labeling). In this sense, again with the exception of the N-queens problem, as the instance of the problem increases, the performance of $\mathcal{TOY}(\mathcal{FD})$ becomes closer to that of GNU Prolog (this result is confirmed for both constraint satisfaction and constraint optimization).

Further, with regard to the comparison to the other $CFLP(\mathcal{FD})$ system, we have to say that PAKCS provides a small set of global constraints (i.e., exactly four) as mentioned in Section 4.5, whereas $\mathcal{TOY}(\mathcal{FD})$ also gives support to specialized constraints for particular problems such as scheduling and placements problems. Moreover, PAKCS does not provide $\mathcal{FD}$ constraints that help users to recover statistics of the constraint solving process (e.g., number of domain prunings, entailments detected by a constraint, backtracks due to inconsistencies, constraint resumptions, etc) which is very useful in practice, as $\mathcal{TOY}(\mathcal{FD})$ does. (For the sake of fairness, we mention again that PAKCS supports the concurrent evaluation of constraints which is also very convenient in practice.)

Based on the results shown in this Section, we can assure that $\mathcal{TOY}(\mathcal{FD})$ is the first pure $CFLP(\mathcal{FD})$ system that provides a wide set of $\mathcal{FD}$ constraints that makes it really competitive compared to existing $CLP(\mathcal{FD})$ systems. These results encourage us to continue working on our approach, and we hope to further improve the results in a close future by means of introducing further optimizations.

5 Related Work

In addition to already cited related work, in this section we discuss some more related work. As already said, most of the work to integrate constraints in the declarative programming paradigm has been developed on LP (Codognet and Diaz 1996; Carlsson et al. 1997). However, there exist some attempts to integrate constraints in the functional logic framework. For instance, (Arenas et al. 1996; López-Fraguas and Sánchez-Hernández 1999) show how to integrate both linear constraints over real numbers and disequality constraints in the FLP language $\mathcal{TOY}$. Also, (Lux 2001) describes the addition of linear constraints over real numbers to the FLP language Curry (Hanus 1999). Our work is guided to the $\mathcal{FD}$ constraint, instead of real constraints (although they are preserved), which allows to use non-linear constraints and adapts better to a range of $\mathcal{FD}$ applications.

With respect to $\mathcal{FD}$, the closer proposal to ours is that described in (Antoy and Hanus 2000) that indicated how the integration of $\mathcal{FD}$ constraints in FLP could be carried out. As already indicated, PAKCS is an implementation that follows these indications.

$\mathcal{TOY}(\mathcal{FD})$ may also be considered from a multiparadigmatic view that means to combine constraint programming with another paradigms in one setting. In this context, there are some similarities with the language Oz (Van Roy et al. 2003; Van Roy and Haridi 2004) as this provides salient features of FP such as compositional syntax and first-class functions, and features of LP and constraint programming including logic variables, constraints, and programmable search mechanisms. However, Oz is quite different to $\mathcal{TOY}(\mathcal{FD})$ because of a number or reasons: (1) Oz does not provide main features of classical functional languages such as explicit types or curried notation; (2) functional notation is provided in Oz as a syntactic convenience; (3) The Oz computation mechanism is not based on rewriting logic as that of $\mathcal{TOY}(\mathcal{FD})$; (4) Oz supports a class of lazy functions based on a demand-driven computation but this is not an inherent feature of the language (as in $\mathcal{TOY}(\mathcal{FD})$) and functions have to be made lazy explicitly (e.g., via the concept of *futures*); (5) functions and constraints are not really integrated, that is to say, they do not have the same category as in $\mathcal{TOY}(\mathcal{FD})$ (i.e., constraints are functions) and both coexist in a concurrent setting, and (6) Oz programs follow a far less concise program syntax than $\mathcal{TOY}(\mathcal{FD})$. In fact, Oz generalizes the CLP and concurrent constraint programming paradigms to provide a very flexible approach to constraint programming very different to our proposal for $CFLP(\mathcal{FD})$.

Also, LIFE (Aït-kaci and Podelski 1993) is an experimental language aiming to integrate logic programming and functional programming but, as Oz, the proposal is quite different to $\mathcal{TOY}(\mathcal{FD})$ as firstly, it is considered in the framework of object-oriented programming, and, secondly, LIFE enables the computation over an order-sorted domain of feature trees by allowing the equality (i.e., unification) and entailment (i.e., matching) constraints over order-sorted feature terms.

There exist other constraint systems that share some aspects with $\mathcal{TOY}(\mathcal{FD})$ although they are very different. One of those systems is FaCiLe (Barnier and Brisset 2001), a constraint programming library that provides constraint solving over integer finite domains, HO functions, type inference, strong typing, and user-defined constraints. However, despite these similarities, FaCiLe is very different to $\mathcal{TOY}(\mathcal{FD})$ as it is built on top of the functional language OCaml that provides full imperative capabilities and does not have a logical component; also OCaml is a strict language, as opposed to lazy ones. In fact, as Oz , it allows the manipulation of potentially infinite data structures by *explicit* delayed expressions, but laziness is not an inherent characteristic of the resolution mechanism as in $\mathcal{TOY}(\mathcal{FD})$. Moreover, FaCiLe is a library and thus it lacks programming language features.

Other interesting systems are OPL (Van Hentenryck 1999) and AMPL (Fourer et al. 1993) that cannot be compared to our work because they are algebraic languages which therefore are not general programming languages. Moreover, these languages do not benefit neither from complex terms and patterns nor from non-determinism as $\mathcal{TOY}(\mathcal{FD})$ does.

Finally, we mention here another $CFLP$ scheme proposed in the Phd Thesis of M. Marin (Marin 2000). This approach introduces $CFLP\,(\mathcal{D},\,\mathcal{S},\,\mathcal{L})$, a family of languages parameterized by a constraint domain $\mathcal{D}$, a strategy $\mathcal{S}$ which defines the cooperation of several constraint solvers over $\mathcal{D}$, and a constraint lazy narrowing calculus $\mathcal{L}$ for solving constraints involving functions defined by user given constrained rewriting rules. This approach relies on solid work on higher-order lazy narrowing calculi and has been implemented on top of Mathematica (Marin et al. 1999; Marin et al. 2000). Its main limitation from our viewpoint is the lack of declarative semantics.

Generally speaking, $\mathcal{TOY}(\mathcal{FD})$ is, from its nature, different to all the constraints systems discussed above since $\mathcal{TOY}(\mathcal{FD})$ is a *pure* FLP language that combines characteristics of *pure* LP and *pure* FP paradigms, and its operational mechanism is the result of combining the operational methods of logic languages (i.e., unification and resolution) and functional languages (i.e., rewriting).

6 Conclusions and Future Work

In this paper we have presented $CFLP(\mathcal{FD})$, a functional logic programming approach to $\mathcal{FD}$ constraint solving. $CFLP(\mathcal{FD})$ is not only a declarative alternative to $CLP(\mathcal{FD})$ but also extends its capabilities with new characteristics unusual or not existing in $CLP(\mathcal{FD})$ such as functional and curried notation, types, curried and higher-order functions (e.g., higher-order constraints), constraint composition, higher-order patterns, lazy evaluation and polymorphism, among others. As a consequence, $CFLP(\mathcal{FD})$ provides better tools, when compared to $CLP(\mathcal{FD})$, for a productive declarative programming as it implicitly enables more expressivity, due to the combination of functional, relational and curried notation as well as type system. Moreover, lazy evaluation allows the use of structures hard to manage in $CLP(\mathcal{FD})$, such as infinite lists.

A $CFLP(\mathcal{FD})$ language is also presented by describing its syntax, type discipline and both declarative and operational semantics. $\mathcal{FD}$ constraints are integrated as functions to make them first-class citizens and allow their use in any place where a data can (e.g., as arguments of functions). This provides a powerful mechanism to define higher-order constraints.

We have also reported an implementation of the $CFLP(\mathcal{FD})$ proposal which connects a FLP language to a $\mathcal{FD}$ constraint solver, that provides both lazy computation and $\mathcal{FD}$ constraint solving. The $\mathcal{FD}$ solver is required to hold termination, soundness and completeness properties. $\mathcal{TOY}(\mathcal{FD})$ is our implementation of the $CFLP(\mathcal{FD})$ language previously described, that connects the functional logic language $\mathcal{TOY}$ to the efficient $\mathcal{FD}$ constraint solver of SICStus Prolog. The result is that $\mathcal{TOY}(\mathcal{FD})$ is a lazy functional logic system with support for $\mathcal{FD}$ constraint solving.

We have also explained the most important contributions by showing the extra capabilities of $CFLP(\mathcal{FD})$ with respect to $CLP(\mathcal{FD})$. This comparison points out the main benefits of integrating FLP and $\mathcal{FD}$ in a declarative language.

Moreover, we have also shown that constraint solving in $\mathcal{TOY}(\mathcal{FD})$ is fairly efficient as, in general, behaves closely to SICStus Prolog, which means that the wrapping of SICStus Prolog by $\mathcal{TOY}$ does not increase significantly the computation time. In addition, $\mathcal{TOY}(\mathcal{FD})$ clearly outperforms existing $CLP(\mathcal{FD})$ systems such as SWI Prolog and Ciao Prolog and also is competitive with respect to GNU Prolog, one of the most efficient $CLP(\mathcal{FD})$ systems. Furthermore, $\mathcal{TOY}(\mathcal{FD})$ is around one and three times faster than PAKCS, its closer $CFLP(\mathcal{FD})$ implementation. Practical applications of $\mathcal{TOY}(\mathcal{FD})$ can be found in (Fernández et al. 2002; Fernández et al. 2003).

Throughout the paper it should have been clear that one inherent advantage of the $CFLP(\mathcal{FD})$ approach is that it enables to solve all the $CLP(\mathcal{FD})$ applications as well as other problems closer to the functional setting.

We claim that the integration of $\mathcal{FD}$ constraints into a FLP language receive benefits from both worlds, i.e., taking functions, higher-order patterns, partial applications, non-determinism, lazy evaluation, logical variables, and types from FLP and domain variables, constraints, and propagators from the $\mathcal{FD}$ constraint programming.

In addition, we claim that the idea of interfacing a FLP language and constraint solvers can be extended to other kind of interesting constraint systems, such as non-linear constraints, constraints over sets, or Boolean constraints, to name a few. Observe that $\mathcal{TOY}(\mathcal{FD})$ can be thought of as a constraint solving procedure integrated into a sophisticated, state-of-the-art execution mechanism for lazy narrowing. Operationally speaking, $\mathcal{TOY}(\mathcal{FD})$ compiles $CFLP(\mathcal{FD})$-programs into Prolog-programs in a system equipped with a constraint solver. This makes both lazy evaluation and constraint solving be inherent features of the system.

Acknowledgment

We thank the anonymous reviewers for their useful comments and suggestions that helped us to improve this paper.

References

Aït-kaci, H. and Podelski, A. 1993. Towards a meaning of LIFE. *Journal of Logic Programming 16,* 3, 195–234. A preliminary version appeared in (Maluszynski and Wirsing 1991), pp:255-274.

Antoy, S. 1992. Definitional trees. In *3rd International Conference on Algebraic and Logic Programming (ALP'92).* Number 632 in LNCS. Springer-Verlag, Volterra, Italy, 143–157.

Antoy, S., Echahed, R., and Hanus, M. 2000. A needed narrowing strategy. *J. ACM 47,* 4, 776–822.

Antoy, S. and Hanus, M. 2000. Compiling multi-paradigm declarative programs into prolog. In *3rd International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, H. Kirchner and C. Ringeissen, Eds. Number 1794 in LNCS. Springer, Nancy, France, 171–185.

Apt, K. 2003. *Principles of constraint programming.* Cambridge University Press.

Arenas, P., Hortalá, M., López-Fraguas, F., and Ullán, E. 1996. Real constraints within a functional logic language. In *Joint Conference on Declarative Programming (APPIA-GULP-PRODE'96)*, P. Lucio, M. Martelli, and M. Navarro, Eds. Donostia-San Sebastian, Spain.

Barnier, N. and Brisset, P. 2001. FaCiLe: a functional constraint library. *ALP Newsletter 14,* 2 (May).

Caballero, R., López-Fraguas, F., and Sánchez, J. 1997. User's manual for $\mathcal{TOY}$. Technical report SIP-5797, Universidad Complutense de Madrid, Dpto. Lenguajes, Sistemas Informáticos y Programación.

Carlsson, M., Ottosson, G., and Carlson, B. 1997. An open-ended finite domain constraint solver. In *9th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'97)*, U. Montanari and F. Rossi, Eds. Number 1292 in LNCS. Springer-Verlag, Southampton, UK, 191–206.

Codognet, P. and Diaz, D. 1996. Compiling constraints in clp(FD). *The Journal of Logic Programming 27,* 3, 185–226.

Damas, L. and Milner, R. 1982. Principal type-schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages (POPL'82).* ACM Press, Albuquerque, New Mexico, 207–212.

Darlington, J., Guo, Y., and Pull, H. 1992. A new perspective on the integration of functional and logic languages. In *International Conference on Fifth Generation Computer Systems (FGCS'92)*, I. Staff, Ed. IOS Press, Tokyo, Japan, 682–693.

Dechter, R. 2003. *Constraint processing.* Morgan Kaufmann.

del Vado-Vírseda, R. 2003. A demand-driven narrowing calculus with overlapping definitional trees. In *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'03).* ACM Press, Uppsala, Sweden, 213–227.

del Vado-Vírseda, R. 2005. Declarative constraint programming with definitional trees. In *5th International Workshop on Frontiers of Combining Systems (FroCoS'05).* Lecture Notes in Computer Science, vol. 3717. Springer, Vienna, Austria, 184–199.

Diaz, D. and Codognet, P. 2001. Design and implementation of the GNU prolog system. *Journal of Functional and Logic Programming 2001,* 6 (October).

Dincbas, M., Simonis, H., and Van Hentenryck, P. 1988. Solving the car-sequencing problem in constraint logic programming. In *8th European Conference on Artificial Intelligence (ECAI'88)*, Y. Kodratoff, Ed. Pitmann Publishing, London, Munich, Germany, 290–295.

Estévez-Martín, S. and del Vado-Vírseda, R. 2005. Designing an efficient computation strategy in $cflp(\mathcal{FD})$ using definitional trees. In *2005 ACM SIGPLAN workshop* *on Curry and functional logic programming (WCFLP '05)*. ACM Press, New York, NY, USA, 23–31.

Fernández, A. J., Hortalá-González, M. T., and Sáenz-Pérez, F. 2002. A constraint functional logic language for solving combinatorial problems. In *Research and Development in Intelligent Systems XIX*, A. P. M. Bramer and F. Coenen, Eds. BCS Conference Series. Springer-Verlag, Cambridge, England, 337–350.

Fernández, A. J., Hortalá-González, M. T., and Sáenz-Pérez, F. 2003. Solving combinatorial problems with a constraint functional logic language. In *5th International Symposium on Practical Aspects of Declarative Languages (PADL'2003)*, P. Wadler and V. Dahl, Eds. Number 2562 in LNCS. Springer-Verlag, New Orleans, Louisiana, USA, 320–338.

Fernández, A. J., Hortalá-González, T., and Sáenz-Pérez, F. 2004. TOY(FD): System, Sources and user manual. Available at `http://toy.sourceforge.net/`.

Fernández, M. 1992. Narrowing based procedures for equational disunification. *Applicable Algebra in Engineering Communication and Computing 3*, 1–26.

Fourer, R., Gay, D., and Kernighan, B. 1993. AMPL: A modeling language for mathematical programming. Scientific Press.

Frühwirth, T. and Abdennadher, S. 2003. *Essentials of constraint programming.* Cognitive Technologies Series. Springer.

GNU Prolog. 2005. `http://pauillac.inria.fr/∼diaz/gnu-prolog/`.

González-Moreno, J., Hortalá-González, M., López-Fraguas, F., and Rodríguez-Artalejo, M. 99a. An approach to declarative programming based on a rewriting logic. *The Journal of Logic Programming 40,* 1 (July), 47–87.

González-Moreno, J., Hortalá-González, M., and Rodríguez-Artalejo, M. 99b. Polymorphic types in functional logic programming. In *4th International Symposium on Functional and Logic Programming (FLOPS'99)*, A. Middeldorp and T. Sato, Eds. Number 1722 in LNCS. Springer-Verlag, Tsukuba, Japan, 1–20. Also published in a special issue of the Journal of Functional and Logic Programming, 2001. See `http://danae.uni-muenster.de/lehre/kuchen/JFLP`.

Gunter, C. and Scott, D. 1990. Semantic domains. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, J. van Leeuwen, Ed. Elsevier and The MIT Press, 633–674.

Hanus, M. 1994. The integration of functions into logic programming: a survey. *The Journal of Logic Programming 19-20*, 583–628. Special issue: Ten Years of Logic Programming.

Hanus, M. 1999. Curry: a truly integrated functional logic language. `http://www.informatik.uni-kiel.de/∼curry/`.

Hanus M. (editor). 2005. PAKCS 1.7.1, User manual (version of December 2005). The Portland Aachen Kiel Curry System. Available from `http://www.informatik.uni-kiel.de/∼pakcs/`.

Haralick, R. and Elliot, G. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence 14*, 263–313.

Henz, M. and Müller, T. 2000. An overview of finite domain constraint programming. In *5th Conference of the Association of Asia-Pacific Operational Research Societies.* Singapore.

Jaffar, J. and Maher, M. 1994. Constraint logic programming: a survey. *The Journal of Logic Programming 19-20*, 503–581.

Loogen, R., López-Fraguas, F. J., and Rodríguez-Artalejo, M. 1993. A demand driven computation strategy for lazy narrowing. In *5th International Symposium on Programming Language Implementation and Logic Programming PLILP'93*, M. Bruynooghe and J. Penjam, Eds. Lecture Notes in Computer Science, vol. 714. Springer, Tallinn, Estonia, 184–200.

López-Fraguas, F. 1992. A general scheme for constraint functional logic programming. In *3rd International Conference on Algebraic and Logic Programming (ALP'92)*, H. Kirchner and G. Levi, Eds. Number 632 in LNCS. Springer-Verlag, Volterra, Italy, 213–227.

López-Fraguas, F., Rodríguez-Artalejo, M., and del Vado-Vírseda, R. 2004a. Constraint functional logic programming revisited. In *5th International Workshop on Rewriting Logic and its Applications (WRLA'2004)*. Barcelona, Spain. An extended and revised version has also been published in Elsevier ENTCS series 117:5–50, 2005.

López-Fraguas, F., Rodríguez-Artalejo, M., and del Vado-Vírseda, R. 2004b. A lazy narrowing calculus for declarative constraint programming. In *6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'04)*. ACM Press, Verona, Italy, 43–54.

López-Fraguas, F. and Sánchez-Hernández, J. 1999. $\mathcal{TOY}$: A multiparadigm declarative system. In *10th International Conference on Rewriting Techniques and Applications*, P. Narendran and M. Rusinowitch, Eds. Number 1631 in LNCS. Springer-Verlag, Trento, Italy, 244–247.

Lux, W. 2001. Adding linear constraints over real numbers to Curry. In *5th International Symposium on Functional and Logic Programming (FLOPS'2001)*, A. Middeldorp, H. Kuchen, and K. Ueda, Eds. Number 2024 in LNCS. Springer-Verlag, Tokyo, Japan, 185–200.

Maluszynski, J. and Wirsing, M., Eds. 1991. *Programming Language Implementation and Logic Programming, 3rd International Symposium, PLILP'91, Passau, Germany, August 26-28, 1991, Proceedings.* Lecture Notes in Computer Science, vol. 528. Springer.

Marin, M. 2000. Functional logic programming with distributed constraint solving. Ph.D. thesis, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz.

Marin, M., Ida, T., and Schreiner, W. 1999. CFLP: a mathematica implementation of a distributed constraint solving system. In *3rd International Mathematical Symposium (IMS'99)*. Computational Mechanics Publications. WIT Press, Hagenberg, Austria, 23–25.

Marin, M., Ida, T., and Suzuki, T. 2000. Cooperative constraint functional logic programming. In *International Symposium on Principles of Software Evolution (IPSE'2000)*, T. Katayama, T. Tamai, and N. Yonezaki, Eds. IEEE, Kanazawa, Japan, 223–230.

Marriot, K. and Stuckey, P. J. 1998. *Programming with constraints.* The MIT Press, Cambridge, Massachusetts.

Middeldorp, A. and Okui, S. 1998. Deterministic lazy narrowing calculus. *Journal of Symbolic Computation 25,* 6, 733–757.

Middeldorp, A., Suzuki, T., and Hamada, M. 2002. Complete selection functions for a lazy conditional narrowing calculus. *Journal of Functional and Logic Programming 3.*

Peyton-Jones, S. 1987. *The implementation of functional programming languages.* Prentice Hall, Englewood Cliffs, N.J.

Shearer, J. 1990. Some new optimum golomb rulers. *IEEE Transactions on Information Theory 36*, 183–184.

SICStus Prolog. 2005. `http://www.sics.se/isl/sicstus`.

SWI-Prolog. 2005. `http://www.swi-prolog.org/`.

Tsang, E. 1993. *Foundations of constraint satisfaction.* Academic Press, London and San Diego.

Van Hentenryck, P. 1989. *Constraint satisfaction in logic programming.* The MIT Press, Cambridge, MA.

Van Hentenryck, P. 1999. *The OPL optimization programming language.* The MIT Press, Cambridge, MA.

Van Roy, P., Brand, P., Duchier, D., Haridi, S., Henz, M., and Schulte, C. 2003. Logic programming in the context of multiparadigm programming: the Oz experience. *Theory and Practice of Logic Programming 3,* 6 (November), 717–763.

Van Roy, P. and Haridi, S. 2004. *Concepts, techniques and models of computer programming.* The MIT Press, Cambridge, MA.

Wielemaker, J. 2003. An overview of the SWI-Prolog programming environment. In *Proceedings of the 13th International Workshop on Logic Programming Environments*, F. Mesnard and A. Serebenik, Eds. Katholieke Universiteit Leuven, Heverlee, Belgium, 1–16. CW 371.

Appendix A [Proof of Theorem 1] in Page 12

The proof of theorem 1 can be done distinguishing several cases from the declarative semantics of each primitive function symbol given in Table 1 and the requirements of each constraint solver rule or failure rule in Tables 2-5:

Rules of Table 2

We examine for example the first rule in Table 2: $seq\ t\ s \to!\ R,\ S\ \square\ \sigma \Vdash_\chi (t == s,\ S\theta_1\ \square\ \sigma\theta_1) \vee (t \setminus = s,\ S\theta_2\ \square\ \sigma\theta_2)$ with $R \notin \chi$, $\theta_1 = \{R \mapsto true\}$ and $\theta_2 = \{R \mapsto false\}$ (the rest of rules in Table 2 are analogous). We prove that $Sol_{\mathcal{FD}}(seq\ t\ s \to!\ R,\ S\ \square\ \sigma) = Sol_{\mathcal{FD}}(t == s,\ S\theta_1\ \square\ \sigma\theta_1) \cup Sol_{\mathcal{FD}}(t \setminus = s,\ S\theta_2\ \square\ \sigma\theta_2)$:

$\subseteq$) Let $\eta \in Sol_{\mathcal{FD}}(seq\ t\ s \to!\ R,\ S\ \square\ \sigma)$. By definition of $Sol_{\mathcal{FD}}$ we have $\eta \in Sol_{\mathcal{FD}}(seq\ t\ s \to!\ R)$ and $\eta \in Sol_{\mathcal{FD}}(S\ \square\ \sigma)$. Since $\eta \in Sol_{\mathcal{FD}}(seq\ t\ s \to!\ R)$ we obtain $seq^{\mathcal{FD}}\ t\eta\ s\eta \to \eta(R)$ with $\eta(R)$ total. According to Table 1, $\eta(R)$ must be only $true$ or $false$. We distinguish two cases:

- If $\eta(R) = true$ then trivially $\eta \in Sol_{\mathcal{FD}}(seq\ t\ s \to!\ true)$ or equivalently $\eta \in Sol_{\mathcal{FD}}(t == s)$. Moreover, since $\eta(R) = true = (true)\eta$ we have $\eta \in Sol(\theta_1)$ and then $\theta_1\eta = \eta$ (because $\eta(\theta_1(R)) = (true)\eta = \eta(R)$ and $\eta(\theta_1(X)) = \eta(X)$ for all $X \neq R$). Then, since $\eta \in Sol_{\mathcal{FD}}(S\ \square\ \sigma)$ we also have $\theta_1\eta \in Sol_{\mathcal{FD}}(S\ \square\ \sigma)$, or equivalently $\eta \in Sol_{\mathcal{FD}}(S\theta_1\ \square\ \sigma\theta_1)$. We can conclude $\eta \in Sol_{\mathcal{FD}}(t == s,\ S\theta_1\ \square\ \sigma\theta_1)$.
- If $\eta(R) = false$, using an analogous reasoning, we can also conclude $\eta \in Sol_{\mathcal{FD}}(t \setminus = s,\ S\theta_2\ \square\ \sigma\theta_2)$.

Therefore, $\eta \in Sol_{\mathcal{FD}}(t == s,\ S\theta_1\ \square\ \sigma\theta_1) \cup Sol_{\mathcal{FD}}(t \setminus = s,\ S\theta_2\ \square\ \sigma\theta_2)$.

$\supseteq$) Let $\eta \in Sol_{\mathcal{FD}}(t == s,\ S\theta_1\ \square\ \sigma\theta_1) \cup Sol_{\mathcal{FD}}(t \setminus = s,\ S\theta_2\ \square\ \sigma\theta_2)$. We distinguish again two cases:

- If $\eta \in Sol_{\mathcal{FD}}(t == s,\ S\theta_1\ \square\ \sigma\theta_1)$ then, by definition of $Sol_{\mathcal{FD}}$ we have $\eta \in$

$Sol_{\mathcal{FD}}(t == s)$ and $\eta \in Sol_{\mathcal{FD}}(S\theta_1 \;\Box\; \sigma\theta_1)$ (or equivalently, $\eta \in Sol_{\mathcal{FD}}(S\theta_1)$ and $\eta \in Sol_{\mathcal{FD}}(\sigma\theta_1)$). Since $\eta \in Sol_{\mathcal{FD}}(\sigma\theta_1)$ and $R \notin dom(\sigma)$ (by initial hypothesis, $seq\ t\ s \to!\ R,\ S \;\Box\; \sigma$ satisfy the requirements of Definition 2) we deduce $\eta \in Sol(\theta_1)$ and then $\eta(R) = (true)\eta = true$. But then, $\eta \in Sol_{\mathcal{FD}}(seq\ t\ s \to!\ R)$ because $seq^{\mathcal{FD}}\ t\eta\ s\eta \to \eta(R) = true$ and we have $\eta \in Sol_{\mathcal{FD}}(t == s)$. Moreover, $\theta_1\eta = \eta$ (because $\eta(\theta_1(R)) = (true)\eta = \eta(R)$ and $\eta(\theta_1(X)) = \eta(X)$ for all $X \neq R$) and we can also obtain $\eta \in Sol_{\mathcal{FD}}(S \;\Box\; \sigma)$ because $\eta \in Sol_{\mathcal{FD}}(S\theta_1 \;\Box\; \sigma\theta_1)$, or equivalently, $\theta_1\eta \in Sol_{\mathcal{FD}}(S \;\Box\; \sigma)$. Therefore, $\eta \in Sol_{\mathcal{FD}}(seq\ t\ s \to!\ R,\ S \;\Box\; \sigma)$.

- If $\eta \in Sol_{\mathcal{FD}}(t \setminus = s,\ S\theta_2 \;\Box\; \sigma\theta_2)$, using an analogous reasoning, we can also conclude $\eta \in Sol_{\mathcal{FD}}(seq\ t\ s \to!\ R,\ S \;\Box\; \sigma)$.

The remaining conditions of the theorem for this rule trivially hold because of the initial hypothesis $seq\ t\ s \to!\ R,\ S \;\Box\; \sigma$ satisfies the requirements of Definition 2, and because of the conditions of the rule $R \notin \chi$.

Rules of Table 3

We examine the first rule in Table 3: $u == u,\ S \;\Box\; \sigma \Vdash_\chi S \;\Box\; \sigma$ with $u \in \mathbb{Z}$. In this case, trivially $Sol_{\mathcal{FD}}(u == u,\ S \;\Box\; \sigma) = Sol_{\mathcal{FD}}(u == u) \cap Sol_{\mathcal{FD}}(S \;\Box\; \sigma) = Val(\mathcal{FD}) \cap Sol_{\mathcal{FD}}(S \;\Box\; \sigma) = Sol_{\mathcal{FD}}(S \;\Box\; \sigma)$. The remaining conditions of the theorem trivially holds by initial hypothesis. We examine now the second rule in Table 3: $X == t,\ S \;\Box\; \sigma \Vdash_\chi t == t,\ S\theta \;\Box\; \sigma\theta$ with $X \notin \chi \cup var(t)$, $var(t) \cap \chi = \emptyset$ and $\theta = \{X \mapsto t\}$. We prove that $Sol_{\mathcal{FD}}(X == t,\ S \;\Box\; \sigma) = Sol_{\mathcal{FD}}(t == t,\ S\theta \;\Box\; \sigma\theta)$:

$\subseteq$) Let $\eta \in Sol_{\mathcal{FD}}(X == t, S \;\Box\; \sigma)$. By definition of $Sol_{\mathcal{FD}}$ we have $\eta \in Sol_{\mathcal{FD}}(X == t)$ and $\eta \in Sol_{\mathcal{FD}}(S \;\Box\; \sigma)$. Since $\eta \in Sol_{\mathcal{FD}}(X == t)$ we obtain $seq^{\mathcal{FD}}\ \eta(X)\ t\eta \to true$. According to Table 1 we obtain $\eta(X) = t\eta$ with $t\eta$ total and then $\eta \in Sol(\theta)$. In this situation, trivially $\eta \in Sol_{\mathcal{FD}}(t == t)$. Moreover, since $\eta \in Sol(\theta)$, we deduce $\theta\eta = \eta$ (because $\eta(\theta(X)) = t\eta = \eta(X)$ and $\eta(\theta(Y)) = \eta(Y)$ for all $Y \neq X$). Then, since $\eta \in Sol_{\mathcal{FD}}(S \;\Box\; \sigma)$, we also have $\theta\eta \in Sol_{\mathcal{FD}}(S \;\Box\; \sigma)$, or equivalently $\eta \in Sol_{\mathcal{FD}}(S\theta \;\Box\; \sigma\theta)$. Therefore, we can conclude $\eta \in Sol_{\mathcal{FD}}(t == t,\ S\theta \;\Box\; \sigma\theta)$.

$\supseteq$) Let $\eta \in Sol_{\mathcal{FD}}(t == t,\ S\theta \;\Box\; \sigma\theta)$. By definition of $Sol_{\mathcal{FD}}$ we have $\eta \in Sol_{\mathcal{FD}}(t == t)$ and $\eta \in Sol_{\mathcal{FD}}(S\theta \;\Box\; \sigma\theta)$ (or equivalently, $\eta \in Sol_{\mathcal{FD}}(S\theta)$ and $\eta \in Sol_{\mathcal{FD}}(\sigma\theta)$). Since $\eta \in Sol_{\mathcal{FD}}(\sigma\theta)$ and $X \notin dom(\sigma)$ (by initial hypothesis, $X == t,\ S \;\Box\; \sigma$ satisfies the requirements of Definition 2) we deduce $\eta \in Sol(\theta)$ and then $\eta(X) = t\eta$. But then, $\eta \in Sol_{\mathcal{FD}}(X == t)$ because $\eta \in Sol_{\mathcal{FD}}(t == t)$ and $seq^{\mathcal{FD}}\ \eta(X)\ t\eta \to true$ with $\eta(X) = t\eta$ total. Moreover, $\theta\eta = \eta$ (because $\eta(\theta(X)) = t\eta = \eta(X)$ and $\eta(\theta(Y)) = \eta(Y)$ for all $Y \neq X$) and we can obtain $\eta \in Sol_{\mathcal{FD}}(S \;\Box\; \sigma)$ because $\eta \in Sol_{\mathcal{FD}}(S\theta \;\Box\; \sigma\theta)$, or equivalently, $\theta\eta \in Sol_{\mathcal{FD}}(S \;\Box\; \sigma)$. Therefore $\eta \in Sol_{\mathcal{FD}}(X == t,\ S \;\Box\; \sigma)$.

The remaining conditions of the theorem for this rule trivially hold because of the initial hypothesis $X == t,\ S \ \square\ \sigma$ satisfies the requirements of Definition 2, and because of the conditions of the rule $X \notin \chi \cup var(t)$ and $var(t) \cap \chi = \emptyset$. Finally, we examine the main rule in Table 3 for strict disequality (the rest of rules in Table 3 are analogous or more simples): $X \setminus = h\ \overline{t}_n,\ S \ \square\ \sigma \Vdash_\chi (\bigvee_i (S\theta_i \ \square\ \sigma\theta_i)) \vee (\bigvee_{k=1}^{n} (U_k \setminus = t_k\theta,\ S\theta \ \square\ \sigma\theta))$ with $X \notin \chi$, $var(h\ \overline{t}_n) \cap \chi \neq \emptyset$, $\theta_i = \{X \mapsto h_i\ \overline{Y}_{m_i}\}$ with $h_i \neq h$, and $\theta = \{X \mapsto h\ \overline{U}_n\}$ with $\overline{Y}_{m_i}$, $\overline{U}_n$ new fresh variables. We prove that $Sol_{\mathcal{FD}}(X \setminus = h\ \overline{t}_n,\ S \ \square\ \sigma) = (\ \bigcup_i Sol_{\mathcal{FD}}(\exists \overline{Y}_{m_i}.\ (S\theta_i \ \square\ \sigma\theta_i))\ ) \cup (\ \bigcup_{k=1}^{n} Sol_{\mathcal{FD}}(\exists \overline{U}_n.\ (U_k \setminus = t_k\theta,\ S\theta \ \square\ \sigma\theta))\ )$:

$\subseteq$) Let $\eta \in Sol_{\mathcal{FD}}(X \setminus = h\ \overline{t}_n,\ S \ \square\ \sigma)$. By definition of $Sol_{\mathcal{FD}}$ we have $\eta \in Sol_{\mathcal{FD}}(X \setminus = h\ \overline{t}_n)$ and $\eta \in Sol_{\mathcal{FD}}(S \ \square\ \sigma)$. Since $\eta \in Sol_{\mathcal{FD}}(X \setminus = h\ \overline{t}_n)$ we obtain $seq^{\mathcal{FD}}\ \eta(X)\ (h\ \overline{t}_n)\eta \rightarrow false$. According to Table 1, $\eta(X)$ and $(h\ \overline{t}_n)\eta = h\ \overline{t_n\eta}$ have no common upper bound w.r.t. the information ordering $\sqsubseteq$, and we can distinguish two cases:

- $\eta(X) = h_i\ \overline{s}_{m_i}$ with $h_i \neq h$. Since $\overline{Y}_{m_i}$ are new variables, we can define $\eta' =_{\setminus \overline{Y}_{m_i}} \eta$ such that $\eta'(Y_k) = s_k$ for all $1 \leq k \leq m_i$ and $\eta'(Z) = \eta(Z)$ for all $Z \notin \overline{Y}_{m_i}$. Clearly, $\eta'(X) = \eta(X) = h_i\ \overline{s}_{m_i} = h_i\ \overline{\eta'(Y_{m_i})} = (h_i\ \overline{Y}_{m_i})\eta'$ and then $\eta' \in Sol(\theta_i)$. Moreover, $\theta_i\eta' =_{\setminus \overline{Y}_{m_i}} \eta$ because $\eta'(\theta_i(X)) = (h_i\ \overline{Y}_{m_i})\eta' = h_i\ \overline{\eta'(Y_{m_i})} = h_i\ \overline{s}_{m_i} = \eta(X)$ and $\eta'(\theta_i(Z)) = \eta'(Z) = \eta(Z)$ for all $Z \notin \{X\} \cup \overline{Y}_{m_i}$. Since $\eta \in Sol_{\mathcal{FD}}(S \ \square\ \sigma)$ and $\overline{Y}_{m_i}$ are new variables in $S \ \square\ \sigma$, we also have $\theta_i\eta' \in Sol_{\mathcal{FD}}(S \ \square\ \sigma)$, or equivalently, $\eta' \in Sol_{\mathcal{FD}}(S\theta_i \ \square\ \sigma\theta_i)$. Finally, since there exists $\eta' =_{\setminus \overline{Y}_{m_i}} \eta$ with $\overline{Y}_{m_i}$ new variables such that $\eta' \in Sol_{\mathcal{FD}}(S\theta_i \ \square\ \sigma\theta_i)$ we can deduce $\eta \in Sol_{\mathcal{FD}}(\exists \overline{Y}_{m_i}.\ (S\theta_i \ \square\ \sigma\theta_i))$ for any i such that $h_i \neq h$.
- $\eta(X) = h\ \overline{s}_n$ with a pattern s_k $(1 \leq k \leq n)$ such that s_k and $t_k\eta$ have no common upper bound w.r.t. the information ordering $\sqsubseteq$ (i.e., $seq^{\mathcal{FD}}\ s_k\ t_k\eta \rightarrow false$). Since $\overline{U}_n$ are new variables, we can define $\eta' =_{\setminus \overline{U}_n} \eta$ such that $\eta'(U_k) = s_k$ for all $1 \leq k \leq n$ and $\eta'(Y) = \eta(Y)$ for all $Y \notin \overline{U}_n$. Clearly, $\eta'(X) = \eta(X) = h\ \overline{s}_n = h\ \overline{\eta'(U_n)} = (h\ \overline{U}_n)\eta'$ and then $\eta' \in Sol(\theta)$. Moreover, $\theta\eta' =_{\setminus \overline{U}_n} \eta$ because $\eta'(\theta(X)) = (h\ \overline{U}_n)\eta' = h\ \overline{\eta'(U_n)} = h\ \overline{s}_n = \eta(X)$ and $\eta'(\theta(Y)) = \eta'(Y) = \eta(Y)$ for all $Y \notin \{X\} \cup \overline{U}_n$. Therefore, there exists $1 \leq k \leq n$ such that $seq^{\mathcal{FD}}\ \eta'(U_k)\ t_k\theta\eta' \rightarrow false$ because $\eta'(U_k) = s_k$ and $t_k\theta\eta' = t_k\eta$ (since $\overline{U}_n$ are new variables, $var(t_k) \cap \overline{U}_n = \emptyset$) and we can deduce $\eta' \in Sol_{\mathcal{FD}}(U_k \setminus = t_k\theta)$. On the other hand, $\eta \in Sol_{\mathcal{FD}}(S \ \square\ \sigma)$, or equivalently $\theta\eta' \in Sol_{\mathcal{FD}}(S \ \square\ \sigma)$, because $\overline{U}_n$ are again new variables in $S \ \square\ \sigma$. We can also conclude $\eta' \in Sol_{\mathcal{FD}}(S\theta \ \square\ \sigma\theta)$. Finally, since there exists $\eta' =_{\setminus \overline{U}_n} \eta$ with $\overline{U}_n$ new variables such that $\eta' \in Sol_{\mathcal{FD}}(U_k \setminus = t_k\theta,\ S\theta \ \square\ \sigma\theta)$, we obtain $\eta \in Sol_{\mathcal{FD}}(\exists \overline{U}_n.\ (U_k \setminus = t_k\theta,\ S\theta \ \square\ \sigma\theta))$ $(1 \leq k \leq n)$.

$\supseteq$) Let $\eta \in (\ \bigcup_i Sol_{\mathcal{FD}}(\exists \overline{Y}_{m_i}.\ (S\theta_i \ \square\ \sigma\theta_i))\ ) \cup (\ \bigcup_{k=1}^{n} Sol_{\mathcal{FD}}(\exists \overline{U}_n.\ (U_k \setminus = t_k\theta,\ S\theta \ \square\ \sigma\theta))\ )$. We distinguish again two cases:

- $\eta \in Sol_{\mathcal{FD}}(\exists \overline{Y}_{m_i}.\ (S\theta_i \ \square\ \sigma\theta_i))$ for any i such that $h_i \neq h$. By definition of $Sol_{\mathcal{FD}}$, there exists $\eta' =_{\setminus \overline{Y}_{m_i}} \eta$ such that $\eta' \in Sol_{\mathcal{FD}}(S\theta_i \ \square\ \sigma\theta_i)$ (or equivalently, $\eta' \in Sol_{\mathcal{FD}}(S\theta_i)$ and $\eta' \in Sol_{\mathcal{FD}}(\sigma\theta_i)$). Since $\eta' \in Sol_{\mathcal{FD}}(\sigma\theta_i)$ and X

$\notin dom(\sigma)$ (by initial hypothesis, $X \setminus = h\ \overline{t}_n,\ S \ \square\ \sigma$ satisfies the requirements of Definition 2), we deduce $\eta' \in Sol(\theta_i)$ and then $\eta'(X) = (h_i\ \overline{Y}_{m_i})\eta' = h_i\ \overline{\eta'(Y_{m_i})}$. Moreover, since $\eta' =_{\setminus \overline{Y}_{m_i}} \eta$, we also deduce $\theta_i\eta' =_{\setminus \overline{Y}_{m_i}} \eta$ because $\eta'(\theta_i(X)) = (h_i\ \overline{Y}_{m_i})\eta' = \eta'(X) = \eta(X)$ and $\eta'(\theta_i(Z)) = \eta'(Z) = \eta(Z)$ for all $Z \notin \{X\} \cup \overline{Y}_{m_i}$. In this situation, $seq^{\mathcal{FD}}\ \eta'(X)\ (h\ \overline{t}_n)\eta' \rightarrow false$, because $\eta'(X) = (h_i\ \overline{Y}_{m_i})\eta' = h_i\ \overline{\eta'(Y_{m_i})}$ and $(h\ \overline{t}_n)\eta' = h\ \overline{t_n\eta'}$ with $h_i \neq h$ have no common upper bound w.r.t. the information ordering $\sqsubseteq$. Therefore, $\eta' \in Sol_{\mathcal{FD}}(X \setminus = h\ \overline{t}_n)$, and we also have $\eta \in Sol_{\mathcal{FD}}(X \setminus = h\ \overline{t}_n)$ because $\overline{Y}_{m_i}$ are new variables in $X \setminus = h\ \overline{t}_n$ and $\eta' =_{\setminus \overline{Y}_{m_i}} \eta$. On the other hand, since $\eta' \in Sol_{\mathcal{FD}}(S\theta_i\ \square\ \sigma\theta_i)$, or equivalently $\theta_i\eta' \in Sol_{\mathcal{FD}}(S\ \square\ \sigma)$, and $\overline{Y}_{m_i}$ are new variables in $S\ \square\ \sigma$, we obtain $\eta \in Sol_{\mathcal{FD}}(S\ \square\ \sigma)$ because $\theta_i\eta' =_{\setminus \overline{Y}_{m_i}} \eta$. Therefore, $\eta \in Sol_{\mathcal{FD}}(X \setminus = h\ \overline{t}_n,\ S\ \square\ \sigma)$.

- $\eta \in Sol_{\mathcal{FD}}(\exists \overline{U}_n.\ (U_k \setminus = t_k\theta,\ S\theta\ \square\ \sigma\theta))$ $(1 \leq k \leq n)$. By definition of $Sol_{\mathcal{FD}}$, there exists $\eta' =_{\setminus \overline{U}_n} \eta$ such that $\eta' \in Sol_{\mathcal{FD}}(U_k \setminus = t_k\theta,\ S\theta\ \square\ \sigma\theta)$ $(1 \leq k \leq n)$. By definition of $Sol_{\mathcal{FD}}$ again we have $\eta' \in Sol_{\mathcal{FD}}(U_k \setminus = t_k\theta)$ and $\eta' \in Sol_{\mathcal{FD}}(S\theta\ \square\ \sigma\theta)$ (or equivalently, $\eta' \in Sol_{\mathcal{FD}}(S\theta)$ and $\eta' \in Sol_{\mathcal{FD}}(\sigma\theta)$). Since $\eta' \in Sol_{\mathcal{FD}}(\sigma\theta)$ and $X \notin dom(\sigma)$ (by initial hypothesis, $X \setminus = h\ \overline{t}_n$, $S\ \square\ \sigma$ satisfies the requirements of Definition 2) we deduce $\eta' \in Sol(\theta)$ and then $\eta'(X) = (h\ \overline{U}_n)\eta' = h\ \overline{\eta'(U_n)}$. Moreover, since $\eta' =_{\setminus \overline{U}_n} \eta$, we also deduce $\theta\eta' =_{\setminus \overline{U}_n} \eta$ because $\eta'(\theta(X)) = (h\ \overline{U}_n)\eta' = \eta'(X) = \eta(X)$ and $\eta'(\theta(Z)) = \eta'(Z) = \eta(Z)$ for all $Z \notin \{X\} \cup \overline{U}_n$. Since $\eta' \in Sol_{\mathcal{FD}}(U_k \setminus = t_k\theta)$, and according to Table 1, we have $seq^{\mathcal{FD}}\ \eta'(U_k)\ t_k\theta\eta' \rightarrow false$ where $\eta'(U_k)$ and $t_k\theta\eta' = t_k\eta$ (because $var(t_k) \cap \overline{U}_n = \emptyset$) have no common upper bound w.r.t. the information ordering $\sqsubseteq$. In this situation, we also have $seq^{\mathcal{FD}}\ \eta(X)\ (h\ \overline{t}_n)\eta \rightarrow false$ because $\eta(X) = (h\ \overline{U}_n)\eta' = h\ \overline{\eta'(U_n)}$, $(h\ \overline{t}_n)\eta = h\ \overline{t_n\eta}$, and clearly $\eta(X)$ and $(h\ \overline{t}_n)\eta$ have no common upper bound w.r.t. the information ordering $\sqsubseteq$ (there exists $1 \leq k \leq n$ such that $\eta'(U_k)$ and $t_k\eta$ have no common upper bound w.r.t. the information ordering $\sqsubseteq$). Therefore, $\eta \in Sol_{\mathcal{FD}}(X \setminus = h\ \overline{t}_n)$. On the other hand, since $\eta' \in Sol_{\mathcal{FD}}(S\theta\ \square\ \sigma\theta)$, or equivalently $\theta\eta' \in Sol_{\mathcal{FD}}(S\ \square\ \sigma)$, and $\overline{U}_n$ are new variables in $S\ \square\ \sigma$, we obtain $\eta \in Sol_{\mathcal{FD}}(S\ \square\ \sigma)$ because $\theta\eta' =_{\setminus \overline{U}_n} \eta$. Therefore, $\eta \in Sol_{\mathcal{FD}}(X \setminus = h\ \overline{t}_n,\ S\ \square\ \sigma)$.

The remaining conditions of the theorem for this rule trivially hold because of the initial hypothesis $X \setminus = h\ \overline{t}_n,\ S\ \square\ \sigma$ satisfies the requirements of Definition 2, and because of the conditions of the rule $X \notin \chi$, $var(h\ \overline{t}_n) \cap \chi \neq \emptyset$, and $\overline{Y}_{m_i}$, $\overline{U}_n$ are new fresh variables.

Rules of Table 4

We examine the first rule in Table 4: $u \leq u',\ S\ \square\ \sigma \Vdash_\chi S\ \square\ \sigma$ with $u, u' \in \mathbb{Z}$ and $u \leq^{\mathbb{Z}} u'$. In this case, trivially $Sol_{\mathcal{FD}}(u \leq u',\ S\ \square\ \sigma) = Sol_{\mathcal{FD}}(u \leq u') \cap Sol_{\mathcal{FD}}(S\ \square\ \sigma) = Val(\mathcal{FD}) \cap Sol_{\mathcal{FD}}(S\ \square\ \sigma) = Sol_{\mathcal{FD}}(S\ \square\ \sigma)$. The remaining conditions of the theorem trivially hold by the initial hypothesis. We examine now the main rule in Table 4 (the rest of rules are analogous or more simples): $a \otimes b = X,\ S\ \square\ \sigma \Vdash_\chi S\theta\ \square\ \sigma\theta$ with $X \notin \chi$, $a, b \in \mathbb{Z}$ and $\theta = \{X \mapsto a \otimes^{\mathbb{Z}} b\}$. We prove that $Sol_{\mathcal{FD}}(a \otimes b = X,\ S \ \Box\ \sigma) = Sol_{\mathcal{FD}}(S\theta \ \Box\ \sigma\theta)$:

$\subseteq$) Let $\eta \in Sol_{\mathcal{FD}}(a \otimes b = X,\ S \ \Box\ \sigma)$. By definition of $Sol_{\mathcal{FD}}$ we have $\eta \in Sol_{\mathcal{FD}}(a \otimes b = X)$ and $\eta \in Sol_{\mathcal{FD}}(S \ \Box\ \sigma)$. Since $\eta \in Sol_{\mathcal{FD}}(a \otimes b = X)$ we obtain $\otimes^{\mathcal{FD}}\ a\ b \to a \otimes^{\mathbb{Z}} b$, $seq^{\mathcal{FD}}\ (a \otimes^{\mathbb{Z}} b)\ \eta(X) \to true$ where $a, b, a \otimes^{\mathbb{Z}} b \in \mathbb{Z}$. According to Table 1, we obtain $\eta(X) = a \otimes^{\mathbb{Z}} b = (a \otimes^{\mathbb{Z}} b)\eta$ and then $\eta \in Sol(\theta)$. Moreover, we deduce $\theta\eta = \eta$ because $\eta(\theta(X)) = (a \otimes^{\mathbb{Z}} b)\eta = a \otimes^{\mathbb{Z}} b = \eta(X)$ and $\eta(\theta(Y)) = \eta(Y)$ for all $Y \neq X$. Since $\eta \in Sol_{\mathcal{FD}}(S \ \Box\ \sigma)$ we also have $\theta\eta \in Sol_{\mathcal{FD}}(S \ \Box\ \sigma)$, or equivalently, $\eta \in Sol_{\mathcal{FD}}(S\theta \ \Box\ \sigma\theta)$.

$\supseteq$) Let $\eta \in Sol_{\mathcal{FD}}(S\theta \ \Box\ \sigma\theta)$. By definition of $Sol_{\mathcal{FD}}$ we have $\eta \in Sol_{\mathcal{FD}}(S\theta)$ and $\eta \in Sol_{\mathcal{FD}}(\sigma\theta)$. Since by initial hypothesis $a \otimes b = X,\ S \ \Box\ \sigma$ satisfies the requirements of Definition 2, we have $X \notin dom(\sigma)$ and then $\eta \in Sol(\theta)$ (i.e., $\eta(X) = (a \otimes^{\mathbb{Z}} b)\eta = (a \otimes^{\mathbb{Z}} b) \in \mathbb{Z}$, where $a, b \in \mathbb{Z}$). But then $\otimes^{\mathcal{FD}}\ a\ b \to a \otimes^{\mathbb{Z}} b$, $seq^{\mathcal{FD}}$ $(a \otimes^{\mathbb{Z}} b)\ \eta(X) \to true$, and therefore $\eta \in Sol_{\mathcal{FD}}(a \otimes b = X)$. Moreover, $\theta\eta = \eta$ because $\eta(\theta(X)) = (a \otimes^{\mathbb{Z}} b)\eta = a \otimes^{\mathbb{Z}} b = \eta(X)$ and $\eta(\theta(Y)) = \eta(Y)$ for all $Y \neq X$. Since $\eta \in Sol_{\mathcal{FD}}(S\theta \ \Box\ \sigma\theta)$, or equivalently $\theta\eta \in Sol_{\mathcal{FD}}(S \ \Box\ \sigma)$, we obtain $\eta \in Sol_{\mathcal{FD}}(S \ \Box\ \sigma)$. Therefore, $\eta \in Sol_{\mathcal{FD}}(a \otimes b = X,\ S \ \Box\ \sigma)$.

The remaining conditions of the theorem for this rule trivially hold because of the initial hypothesis $a \otimes b = X,\ S \ \Box\ \sigma$ satisfies the requirements of Definition 2, and because of the conditions of the rule $X \notin \chi$.

Rules of Table 5

We examine the first rule in Table 5: $u \in [u_1, \ldots, u_n],\ S \ \Box\ \sigma \Vdash_{\chi} S \ \Box\ \sigma$ with $u, u_i \in \mathbb{Z} \cup \mathcal{V}ar$ and $\exists i \in \{1, \ldots, n\}.\ u_i \equiv u$. In this situation, and according to Table 1, we have $Sol_{\mathcal{FD}}(u \in [u_1, \ldots, u_n]) = Val(\mathcal{FD})$: $\eta \in Sol_{\mathcal{FD}}(u \in [u_1, \ldots, u_n])$ implies that $domain^{\mathcal{FD}}\ u\eta\ [u_1\eta, \ldots, u_n\eta] \to true$ where $\forall i \in \{1, \ldots, n{-}1\}.\ u_i\eta \leq^{\mathbb{Z}} u_{i+1}\eta$ and $\exists i \in \{1, \ldots, n\}.\ u\eta =^{\mathbb{Z}} u_i\eta$. It holds for all $\eta \in Val(\mathcal{FD})$ because of the initial hypothesis $u \in [u_1, \ldots, u_n],\ S \ \Box\ \sigma$ satisfies the requirements of Definition 2 (i.e., $[u_1, \ldots, u_n]$ represents an increasing integer list), and because of the conditions of this rule (i.e., $\exists i \in \{1, \ldots, n\}.\ u_i \equiv u$). Then, trivially $Sol_{\mathcal{FD}}(u \in [u_1, \ldots, u_n],\ S \ \Box\ \sigma) = Sol_{\mathcal{FD}}(u \in [u_1, \ldots, u_n]) \cap Sol_{\mathcal{FD}}(S \ \Box\ \sigma) = Val(\mathcal{FD}) \cap Sol_{\mathcal{FD}}(S \ \Box\ \sigma) = Sol_{\mathcal{FD}}(S \ \Box\ \sigma)$. The remaining conditions of the theorem for this rule trivially hold by the initial hypothesis. The second rule in Table 5 is completely analogous: $Sol_{\mathcal{FD}}(u \notin [u_1, \ldots, u_n]) = Val(\mathcal{FD})$ because $u, u_i \in \mathbb{Z}$, $\forall i \in \{1, \ldots, n\}.\ u_i \neq^{\mathbb{Z}} u$, and according to Table 1, $domain^{\mathcal{FD}}\ u\eta\ [u_1\eta, \ldots, u_n\eta] \to false$ holds for all $\eta \in Val(\mathcal{FD})$.

Finally, we examine the main rule for *labeling* in Table 5: *labeling* $[\ldots]\ [X]$, $X \in [u_1, \ldots, u_n],\ S \ \Box\ \sigma \Vdash_{\chi} \bigvee_{i=1}^{n} (S\theta_i \ \Box\ \sigma\theta_i)$ with $X \notin \chi$, and $\forall i \in \{1, \ldots, n\}$, $u_i \in \mathbb{Z}$, $\theta_i = \{X \mapsto u_i\}$. We prove that $Sol_{\mathcal{FD}}(labeling\ [\ldots]\ [X],\ X \in [u_1, \ldots, u_n],\ S \ \Box\ \sigma) = \bigcup_{i=1}^{n} Sol_{\mathcal{FD}}(S\theta_i \ \Box\ \sigma\theta_i)$:

$\subseteq$) Let $\eta \in Sol_{\mathcal{FD}}(labeling\ [\ldots]\ [X],\ X \in [u_1,\ \ldots,\ u_n],\ S\ \square\ \sigma)$. By definition of $Sol_{\mathcal{FD}}$ we have $\eta \in Sol_{\mathcal{FD}}(labeling\ [\ldots]\ [X], X \in [u_1, \ldots, u_n])$ and $\eta \in Sol_{\mathcal{FD}}(S\ \square\ \sigma)$. Then, $indomain^{\mathcal{FD}}\ \eta(X) \rightarrow \top$, $domain^{\mathcal{FD}}\ \eta(X)\ [u_1, \ldots, u_n] \rightarrow true$ because $u_i \in \mathbb{Z}$ for all $1 \leq i \leq n$. According to Table 1, we deduce $\eta(X) \in \mathbb{Z}$, $\forall i \in \{1, \ldots, n{-}1\}$. $u_i \leq^{\mathbb{Z}} u_{i+1}$ and $\exists i \in \{1, \ldots, n\}$. $\eta(X) =^{\mathbb{Z}} u_i$. Therefore, $\eta(X) = u_i = u_i\eta$ and then $\eta \in Sol(\theta_i)$ $(1 \leq i \leq n)$. Moreover, we have $\theta_i\eta = \eta$ because $\eta(\theta_i(X)) = u_i\eta = u_i = \eta(X)$ and $\eta(\theta_i(Y)) = \eta(Y)$ for all $Y \neq X$. Finally, since $\eta \in Sol_{\mathcal{FD}}(S\ \square\ \sigma)$ we can conclude $\theta_i\eta \in Sol_{\mathcal{FD}}(S\ \square\ \sigma)$ or equivalently $\eta \in Sol_{\mathcal{FD}}(S\theta_i\ \square\ \sigma\theta_i)$ $(1 \leq i \leq n)$.

$\supseteq$) Let $\eta \in Sol_{\mathcal{FD}}(S\theta_i\ \square\ \sigma\theta_i)$ $(1 \leq i \leq n)$. By definition of $Sol_{\mathcal{FD}}$ we have $\eta \in Sol_{\mathcal{FD}}(S\theta_i)$ and $\eta \in Sol_{\mathcal{FD}}(\sigma\theta_i)$. By the initial hypothesis $labeling\ [\ldots]\ [X]$, $X \in [u_1, \ldots, u_n]$, $S\ \square\ \sigma$ satisfies the requirements of Definition 2, we have $X \notin dom(\sigma)$ and then $\eta \in Sol(\theta_i)$ (i.e., $\eta(X) = u_i\eta = u_i$ due to $u_i \in \mathbb{Z}$). Moreover, we have $\theta_i\eta = \eta$ because $\eta(\theta_i(X)) = u_i\eta = u_i = \eta(X)$ and $\eta(\theta_i(Y)) = \eta(Y)$ for all $Y \neq X$. Then, since $\eta \in Sol_{\mathcal{FD}}(S\theta_i\ \square\ \sigma\theta_i)$, or equivalently, $\theta_i\eta \in Sol_{\mathcal{FD}}(S\ \square\ \sigma)$, we deduce $\eta \in Sol_{\mathcal{FD}}(S\ \square\ \sigma)$. Finally, we prove that $\eta \in Sol_{\mathcal{FD}}(labeling\ [\ldots]\ [X],\ X \in [u_1,\ \ldots,\ u_n])$. Since $\eta(X)$, $u_i \in \mathbb{Z}$ for all $1 \leq i \leq n$, $[u_1, \ldots, u_n]$ is an increasing integer list by the initial hypothesis, and there exists $1 \leq i \leq n$ such that $\eta(X) = u_i \in \mathbb{Z}$, according to Table 1 we can deduce $domain^{\mathcal{FD}}\ \eta(X)\ [u_1, \ldots, u_n] \rightarrow true$. Moreover, since $\eta(X) \in \mathbb{Z}$, trivially $indomain^{\mathcal{FD}}\ \eta(X) \rightarrow \top$ according again to Table 1. Then, $indomain^{\mathcal{FD}}\ \eta(X) \rightarrow \top$, $domain^{\mathcal{FD}}\ \eta(X)\ [u_1, \ldots, u_n] \rightarrow true$, and we can conclude that $\eta \in Sol_{\mathcal{FD}}(labeling\ [\ldots]\ [X],\ X \in [u_1,\ \ldots,\ u_n])$. Therefore, $\eta \in Sol_{\mathcal{FD}}(labeling\ [\ldots]\ [X],\ X \in [u_1,\ \ldots,\ u_n],\ S\ \square\ \sigma)$.

The remaining conditions of the theorem for this rule trivially hold because of the initial hypothesis $labeling\ [\ldots]\ [X]$, $X \in [u_1,\ \ldots,\ u_n]$, $S\ \square\ \sigma$ satisfies the requirements of Definition 2, and because of the conditions of the rule $X \notin \chi$. The last rule for *labeling* follows a trivial reasoning because $Sol_{\mathcal{FD}}(labeling\ [\ldots]\ [u]) = Val(\mathcal{FD})$ if $u \in \mathbb{Z}$. According to Table 1, $indomain^{\mathcal{FD}}\ u\eta \rightarrow \top$ for all $\eta \in Val(\mathcal{FD})$. Therefore, $Sol_{\mathcal{FD}}(labeling\ [\ldots]\ [u],\ S\ \square\ \sigma) = Sol_{\mathcal{FD}}(labeling\ [\ldots]\ [u]) \cap Sol_{\mathcal{FD}}(S\ \square\ \sigma) = Val(\mathcal{FD}) \cap Sol_{\mathcal{FD}}(S\ \square\ \sigma) = Sol_{\mathcal{FD}}(S\ \square\ \sigma)$. The remaining conditions of the theorem are also trivial by initial hypothesis.

Failure Rules

Finally, we suppose any arbitrary failure rule such that $S\ \square\ \sigma \Vdash_{\chi} fail$ and we prove that $Sol_{\mathcal{FD}}(S\ \square\ \sigma) = \emptyset$. First, we note that any failure rule must have the following syntactic form: $S_1,\ S_2\ \square\ \sigma \Vdash_{\chi} fail$ with conditions such that $Sol_{\mathcal{FD}}(S_1) = \emptyset$. For example, consider the failure rule associated to Table 4: $u \leq u'$, $S\ \square\ \sigma \Vdash_{\chi} fail$ with $u,\ u' \in \mathbb{Z}$ and $u >^{\mathbb{Z}} u'$. Clearly, $Sol_{\mathcal{FD}}(u \leq u') = \emptyset$. In this situation, $Sol_{\mathcal{FD}}(S_1,\ S_2\ \square\ \sigma) = Sol_{\mathcal{FD}}(S_1) \cap Sol_{\mathcal{FD}}(S_2\ \square\ \sigma) = \emptyset \cap Sol_{\mathcal{FD}}(S_2\ \square\ \sigma) = \emptyset$. ♦