COMPILING CONSTRAINTS IN `clp(FD)`

PHILIPPE CODOGNET AND DANIEL DIAZ

▷ We present the `clp(FD)` system: a Constraint Logic Programming language with finite domain constraints. We detail its implementation and present an abstract instruction set for the constraint solver that can be smoothly integrated into the WAM architecture. It is based on the use of a single *primitive constraint* X `in` r that embeds the core propagation mechanism. Complex user constraints such as linear equations or inequations are compiled into X `in` r expressions that encode the propagation scheme chosen to solve the constraint. The uniform treatment of a single primitive constraint leads to a better understanding of the overall constraint solving process and allows three main general optimizations that encompass many previous particular optimizations of "black-box" finite domain solvers. Implementation results show that this approach combines both simplicity and efficiency. Our `clp(FD)` system is about four times faster than CHIP on average, with peak speedup reaching eight. We also show that, following the "glass-box" approach, `clp(FD)` can be naturally enhanced with various new constraints such as constructive disjunction, boolean constraints, non-linear constraints and symbolic constraints. ◁

1. INTRODUCTION

Constraint Logic Programming (CLP) has shown to be a very active field of research over recent years, and languages such as CHIP [21, 44, 2], CLP($\mathcal{R}$) [26, 28] and PrologIII [17] have proved that this approach opens Logic Programming (LP) up to a wide range of real-life applications.

*Address correspondence to*Philippe Codognet and Daniel Diaz, INRIA-Rocquencourt, BP 105, 78153 Le Chesnay, FRANCE. email: {Philippe.Codognet,Daniel.Diaz}@inria.fr

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$3.50

The basic idea of CLP [26] is to replace unification by constraint solving over a particular domain of interest, considering the constraint solver as a "black-box" that is responsible for checking the consistency of a set of constraints and, possibly, for reducing it into some normal form. Although this dichotomy is very important from the theoretical point of view, and makes it possible to import many results from LP semantics into CLP, it is not very satisfactory from the practical point of view. It may be noted that there is a curious lack of literature about the practical side of CLP...

One of the major breakthroughs of the last decade in LP has arguably been the definition of the Warren Abstract Machine (WAM) [50] that became a de facto standard for the compilation of Prolog and has helped many researchers to gain a better understanding of Prolog's execution and to develop efficient LP systems. Moreover the WAM proved to be flexible enough to remain the backbone of various extensions such as Higher-Order, parallel or concurrent LP. To return to CLP, we could but deplore the fact that the black-box approach does not give much information about the architecture of a real CLP system, and does not lead to the design of an abstract machine for constraints. One of the main issues is that there should be as many abstract machines as constraint domains and solvers.

We chose to focus on Finite Domains (FD), as introduced in LP by the CHIP language, where constraint solving is done by propagation and consistency techniques originating from Constraint Satisfaction Problems [48, 30, 33]. Very close to those methods are the interval arithmetic constraints of BNR-Prolog [4]. Luckily, a recent paper [46] broke the black-box monopoly by unveiling a "glass-box" for FD constraints. The basic idea is to have a *single constraint* X `in` r, where r denotes a *range* (e.g. $t1..t2$). More complex constraints such as linear equations and inequations are then defined in terms of this primitive constraint. The X `in` r constraint can be seen as embedding the *core* propagation mechanism for constraint solving over FD, and should be a good basis for an abstract machine for CLP(FD)[1].

We have thus developed an extension of the WAM for FD based on the X `in` r constraint, and we propose an instruction set to implement this constraint that is very much more in the spirit of the WAM. It is also worth noticing that the basic WAM architecture and data structures are left untouched, e.g. the representation of choice-points, environments and non-FD terms is not changed. Complex FD constraints are translated at compile-time into a set of X `in` r constraints that really encodes the propagation scheme chosen to solve the constraint. This makes it possible to express at a high level the constraint solving scheme and to change it very simply if desired. Indeed the X `in` r expressions give us a *language* to express propagation methods, which is obviously not the case with the black-box approach of CHIP or BNR where one has to get down to C for any change. Also the uniform treatment of a single primitive for all complex "user" constraints leads to a better understanding of the overall constraint solving process and allows for (a few) general optimizations, as opposed to the many local and particular optimizations hidden inside the black-box. Hence, we have designed three simple but powerful optimizations for the X `in` r constraint that encompass many previous particular optimizations for FD constraints. Implementation results show that this approach was sound and can be competitive in terms of efficiency. On a traditional set of

[1] although the authors introduced it in the context of concurrent constraint languages [40].

benchmark programs, our `clp(FD)` engine is about four times faster than the CHIP system, with peak speedup reaching eight.

The rest of this paper is organized as follows. Section 2 presents the FD constraint system and Section 3 explains the use of X `in` r to define high-level constraints. Section 4 describes the integration of X `in` r into the WAM, presents the compilation scheme and details the work performed when telling a constraint, while Section 5 presents implementation results of the `clp(FD)` system. Results of a basic implementation are first presented and analyzed before we propose three optimizations whose impact is then assessed. Section 6 shows the ability of `clp(FD)` to deal with disjunctive constraints in an active manner. Section 7 presents `clp(B/FD)` an efficient boolean solver built on top of `clp(FD)`. Section 8 shows how a generalization of X `in` r allows the user to define in a declarative way many symbolic constraints usually "wired" in black-box solvers. A short conclusion and perspectives end the paper.

2. THE FD CONSTRAINT SYSTEM

The FD constraint system is a general purpose constraint framework for solving discrete constraint satisfaction problems (CSP). It was originally proposed by Pascal Van Hentenryck in a concurrent constraint setting [46]. FD is based on a single *primitive constraint* by which complex constraints are defined, so for example constraints such as $X = Y$ or $X \leq 2Y$ are *defined* by FD constraints, instead of being built into the theory. This constraint is thought of as propagation rules, i.e. rules for describing node and arc consistency propagation (see [48, 30, 33] for more details on CSPs and consistency algorithms). We present here the basic notions underlying the FD constraint system.

A *domain* in FD is a (non empty) finite set of natural numbers (i.e. a *range*). More precisely a range is a subset of $\{0, 1, ..., infinity\}$ where $infinity$ is a particular integer denoting the greateast value that a variable can take[2]. *Dom* is the set of all domains. We use the interval notation $k_1..k_2$ as shorthand for the set $\{k_1, k_1 + 1, ..., k_2\}$. From a range r we define $min(r)$ (resp. $max(r)$) as the lower (resp. upper) bound of r. In addition to standard operations on sets (e.g. union, intersection, etc.) we define *pointwise operations* $(+, -, *, /)$ between a range r and an integer i as the set obtained by applying the corresponding operation on each element of d. Finally, $\mathcal{V}_d$ is the set of FD variables, i.e. variables constrained to take a value in a given domain.

2.1. Syntax of X `in` r

As mentioned above, the FD constraint system is based on a unique constraint X `in` r. There are three kinds of syntactic objects: constraints (c), ranges (r) and arithmetic terms (t and ct for constant terms). $Constr$ is the set of syntactic constraints, $SynDom$ is the set of syntactic domains and $SynTerm$ is the set of syntactic terms.

[2] from the implementation point of view this value depends on the machine.

Definition 2.1. A *constraint* is a formula of the form X `in` r where $X \in \mathcal{V}_d$ and $r \in SynDom$ (cf. syntax in Table 1).

$c ::=$	X `in` r	
$r ::=$	$t_1..t_2$	(interval)
	$\{t\}$	(singleton)
	R	(range parameter)
	`dom(`Y`)`	(indexical domain)
	r_1 `:` r_2	(union)
	r_1 `&` r_2	(intersection)
	`-`r	(complementation)
	r `+` ct	(pointwise addition)
	r `-` ct	(pointwise subtraction)
	r `*` ct	(pointwise multiplication)
	r `/` ct	(pointwise division)
$t ::=$	`min(`Y`)`	(indexical term min)
	`max(`Y`)`	(indexical term max)
	ct	(constant term)
	t_1`+`t_2 | t_1`-`t_2 | t_1`*`t_2 | t_1`/<`t_2 | t_1`/>`t_2	(integer operations)
$ct ::=$	C	(term parameter)
	n | `infinity`	(greatest value)
	ct_1`+`ct_2 | ct_1`-`ct_2 | ct_1`*`ct_2 | ct_1`/<`ct_2 | ct_1`/>`ct_2	

Table 1. Syntax of X `in` r constraints

We will use $X = n$ as shorthand for X `in` $n..n$. Intuitively, a constraint X `in` r enforces X to belong to the range denoted by r that can be not only a *constant range* (e.g. 1..10) but also an *indexical range* using:

- `dom(`Y`)` representing the whole current domain of Y.
- `min(`Y`)` representing the minimum value of the current domain of Y.
- `max(`Y`)` representing the maximum value of the current domain of Y.

When an X `in` r constraint uses an indexical on another variable Y it becomes *store sensitive* and must be checked each time the domain of Y is updated. A constraint can also use *parameters* which are run-time constant values. Let us remark that the FD constraint system is closed under negation since the constraint $\neg X$ `in` r is just X `in` `-`r.

A *store* is a finite set of constraints. A store is in *normal form* iff it contains exactly one constraint X `in` r for each variable $X \in \mathcal{V}_d$. From any store S we obtain a store in normal form by replacing all constraints X `in` r_1, X `in` r_2, ... , X `in` r_n on X by a single constraint of the form X `in` r_1 `&` r_2 `&` `...` `&` r_n. These sets are obviously equivalent since they have the same tuples of solutions. In

the following we always consider stores in normal form, in particular the result of $S \cup \{c\}$ is a store in normal form obtained by the addition of c to S (see section 2.2 for details about this operation). *Store* is the set of all stores.

During computation, a constraint can *succeed*, *fail* or *suspend*. For example, let us consider the store {`X in 3..20`, `Y in 5..7:10..100`}:

- `X in 10..50` succeeds and the new store is:
  {`X in 3..20 & 10..50`, `Y in 5..7:10..100`}, i.e.:
  {`X in 10..20`, `Y in 5..7:10..100`}.
- `X in 30..50` fails.
- `X in min(Y)..40` suspends and the new store is:
  {`X in 3..20 & 5..40 & min(Y)..40`, `Y in 5..7:10..100`}, i.e.:
  {`X in 5..20 & min(Y)..40`, `Y in 5..7:10..100`}.
  Let us remark that the (indexical) constraint `X in min(Y)..40` provides a constraint evaluated in the current store (i.e. `X in 5..40`) and *remains* in the store to propagate future reductions of the domain of `Y`.
- `X in dom(Y)+1` suspends and the new store is:
  {`X in 3..20 & 6..8:11..101 & dom(Y)+1`, `Y in 5..7:10..100`}, i.e.:
  {`X in 6..8:11..20 & dom(Y)+1`, `Y in 5..7:10..100`}.

A constraint c can be removed from the current store only if it succeeds. If c suspends, it must remain in the store. Hence in the third example, `X in min(Y)..40` must remain in the store as long as $min(Y)$ is greater than $min(X)$. Indeed, at each modification of $min(Y)$, that constraint will be triggered to check consistency with the domain of X, reducing it if necessary.

2.2. Semantics of X `in` r

The addition of a constraint in a store is a *tell* operation. We present a denotational semantics for this operation in Table 2. The function $\mathcal{T}\ [\![X \texttt{ in } r]\!]\ S$ gives the semantics of the $Tell$ operation consisting in adding the constraint X `in` r to the store S. This consists in updating X (wrt r evaluated in S) and in reactivating all constraints depending on X (i.e. propagation). The first phase is ensured by the semantic function $\mathcal{T}'\ [\![X \texttt{ in } r]\!]$ and the propagation is simply modeled using a fixpoint operator on the result of $\mathcal{T}\ [\![X \texttt{ in } r]\!]$ that re-evaluates all constraints in $S \cup \{X \texttt{ in } r\}$ (via $\mathcal{T}'$ until quiescence). It is worth noticing that the intermediate funcion $\mathcal{T}'\ [\![X \texttt{ in } r]\!]$ adds two instances of the constraint X `in` r to the store:

(a) one where r is evaluated in S (see the semantic function $\mathcal{E}_r\ [\![r]\!]$ where $\lceil \mathcal{E}_r\ [\![r]\!]\ s \rceil$ represents the syntactic domain associated to the evaluation of r).

(b) one where r is unchanged to allow future reconsiderations of this constraint in the presence of an indexical range.

To evaluate an indexical (e.g. `dom(`X`)`) it is necessary to get the current domain of a variable. Thanks to point **(a)**, this comes down to obtaining the constraint X `in` r associated to X (cf. `lookup_store` function) and to evaluating r in an

$SynDom$	: syntactic domains	$\mathcal{T}$	: $Constr \rightarrow Store \rightarrow Store$
$SynTerm$	: syntactic terms	$\mathcal{T}'$	: $Constr \rightarrow Store \rightarrow Store$
Dom	: domains	$\mathcal{E}_r$	: $SynDom \rightarrow Store \rightarrow Dom$
$\mathcal{N}$	: natural numbers	$\mathcal{E}_t$	: $SynTerm \rightarrow Store \rightarrow \mathcal{N}$
$Constr$	: X `in` r constraints		
$Store$	: stores		

$$
\begin{array}{ll}
\mathcal{T}\ [\![c]\!]\ s & = \texttt{fix}\ (\lambda\ s\ .\ \bigcup_{c' \in s \cup \{c\}} \mathcal{T}'\ [\![c']\!]\ s) \\
\mathcal{T}'\ [\![x\ \texttt{in}\ r]\!]\ s & = \textbf{let}\ \texttt{d} = \lceil \mathcal{E}_r\ [\![r]\!]\ s \rceil\ \textbf{in}\ s \cup \{\ x\ \texttt{in}\ d\ \} \cup \{\ x\ \texttt{in}\ r\ \} \\
 & \\
\mathcal{E}_r\ [\![t_1 .. t_2]\!]\ s & = \mathcal{E}_t\ [\![t_1]\!]\ s\ ..\ \mathcal{E}_t\ [\![t_2]\!]\ s \\
\mathcal{E}_r\ [\![\{t\}]\!]\ s & = \{\mathcal{E}_t\ [\![t]\!]\ s\} \\
\mathcal{E}_r\ [\![R]\!]\ s & = \texttt{lookup\_range}(R) \\
\mathcal{E}_r\ [\![\texttt{dom}(Y)]\!]\ s & = \texttt{cur\_domain}(X, s) \\
\mathcal{E}_r\ [\![r_1\ \texttt{:}\ r_2]\!]\ s & = \mathcal{E}_r\ [\![r_1]\!]\ s\ \cup\ \mathcal{E}_r\ [\![r_2]\!]\ s \\
\mathcal{E}_r\ [\![r_1\ \texttt{\&}\ r_2]\!]\ s & = \mathcal{E}_r\ [\![r_1]\!]\ s\ \cap\ \mathcal{E}_r\ [\![r_2]\!]\ s \\
\mathcal{E}_r\ [\![\texttt{-}r]\!]\ s & = 0..infinity\ \setminus\ \mathcal{E}_r\ [\![r]\!]\ s \\
\mathcal{E}_r\ [\![r\ \texttt{+}\ ct]\!]\ s & = \mathcal{E}_r\ [\![r]\!]\ s\ +\ \mathcal{E}_t\ [\![ct]\!]\ s \\
\mathcal{E}_r\ [\![r\ \texttt{-}\ ct]\!]\ s & = \mathcal{E}_r\ [\![r]\!]\ s\ -\ \mathcal{E}_t\ [\![ct]\!]\ s \\
\mathcal{E}_r\ [\![r\ \texttt{*}\ ct]\!]\ s & = \mathcal{E}_r\ [\![r]\!]\ s\ *\ \mathcal{E}_t\ [\![ct]\!]\ s \\
\mathcal{E}_r\ [\![r\ \texttt{/}\ ct]\!]\ s & = \mathcal{E}_r\ [\![r]\!]\ s\ /\ \mathcal{E}_t\ [\![ct]\!]\ s \\
 & \\
\mathcal{E}_t\ [\![n]\!]\ s & = n \\
\mathcal{E}_t\ [\![\texttt{infinity}]\!]\ s & = infinity \\
\mathcal{E}_t\ [\![C]\!]\ s & = \texttt{lookup\_term}(C) \\
\mathcal{E}_t\ [\![\texttt{min}(Y)]\!]\ s & = min(\texttt{cur\_domain}(X, s)) \\
\mathcal{E}_t\ [\![\texttt{max}(Y)]\!]\ s & = max(\texttt{cur\_domain}(X, s)) \\
\mathcal{E}_t\ [\![t_1\ \texttt{+}\ t_2]\!]\ s & = \mathcal{E}_t\ [\![t_1]\!]\ s\ +\ \mathcal{E}_t\ [\![t_2]\!]\ s \\
\mathcal{E}_t\ [\![t_1\ \texttt{-}\ t_2]\!]\ s & = \mathcal{E}_t\ [\![t_1]\!]\ s\ -\ \mathcal{E}_t\ [\![t_2]\!]\ s \\
\mathcal{E}_t\ [\![t_1\ \texttt{*}\ t_2]\!]\ s & = \mathcal{E}_t\ [\![t_1]\!]\ s\ *\ \mathcal{E}_t\ [\![t_2]\!]\ s \\
\mathcal{E}_t\ [\![t_1\ \texttt{/<}\ t_2]\!]\ s & = \lfloor \mathcal{E}_t\ [\![t_1]\!]\ s\ /\ \mathcal{E}_t\ [\![t_2]\!]\ s \rfloor \\
\mathcal{E}_t\ [\![t_1\ \texttt{/>}\ t_2]\!]\ s & = \lceil \mathcal{E}_t\ [\![t_1]\!]\ s\ /\ \mathcal{E}_t\ [\![t_2]\!]\ s \rceil \\
 & \\
\texttt{cur\_domain}(X, s) & = \mathcal{E}_r\ \texttt{lookup\_store}(X, s)\ \emptyset \\
\texttt{lookup\_store}(X, s) & = \textbf{if}\ \exists\ X\ \texttt{in}\ r \in s\ \textbf{then}\ r\ \textbf{else}\ \texttt{0..infinity} \\
 & \\
\texttt{lookup\_range}(R) & \text{returns the domain bound to } R \\
\texttt{lookup\_term}(C) & \text{returns the integer bound to } C
\end{array}
$$

Table 2. Denotational semantics of the *Tell* operation

empty store (cf. `cur_domain` function) to avoid the evaluation of the indexicals of r.

We will use the following shorthands to avoid cumbersome notations:

- $X_S =$ `cur_domain(`X`,`S`)` (i.e. the value of the domain of X in S).
- $min(X)_S = min(X_S)$.
- $max(X)_S = max(X_S)$.
- $r_S = \mathcal{E}_r \ [\![r]\!] \ S$ (i.e. domain denoted by r in S).
- $t_S = \mathcal{E}_t \ [\![t]\!] \ S$ (i.e. integer denoted by t in S).

Definition 2.2. A store S is *consistent* iff it does not contain any empty domain, i.e. $\forall X \in \mathcal{V}_d \quad X_S \neq \emptyset$.

Definition 2.3. A variable X is *instantiated* to n in the store S iff $X_S = \{n\}$.

Definition 2.4. Let S and S' be two sets of constraints, S' is *stronger than* S ($S' \sqsubseteq S$) iff $\forall X \in \mathcal{V}_d \quad X_{S'} \subseteq X_S$.

One important requirement is that telling a constraint has to be a *monotonic* operation. From a theoretical point of view this ensures the existence of a fixpoint (since all domains are finite). From a practical point of view this makes it possible to remove impossible values as soon as they appear and to avoid reconsidering of accumulated information when a constraint is told. The tell of a constraint X `in` r is monotonic if the range denoted by r is monotonic, i.e. can only decrease when more constraints are added [46].

Definition 2.5. A range r is *monotonic* (resp. *anti-monotonic*) iff
$\forall S, S' \quad S' \sqsubseteq S \ \Rightarrow \ r_{S'} \subseteq r_S$ (resp. $r_S \subseteq r_{S'}$).
A constraint $c \equiv X$ `in` r is (anti-)monotonic iff r is (anti-)monotonic.

In that case, the tell of a constraint X `in` r simply consists in removing impossible values of X, which do not belong to r (see Figure 1). This comes down to an intersection operation between X and r (see also the above definition of a store in normal form). Figure 1 schematize this operation.

To ensure that r is monotonic, some syntactic restrictions have to be made on the use of the indexicals. Intuitively, a "positive" occurrence of $min(X)$ (resp. $max(X)$) must be used in the lower (resp. upper) part of the range and conversely for a "negative" occurrence. The indexical term $dom(X)$ must be used in "positive" occurrences. An incorrect use of $dom(X)$ could be X `in -dom(`Y`)`. Indeed, as the domain of Y decreases, its complementary increases. The simplest way to deal with an incorrect indexical $min/max/dom(X)$ is to wait until X becomes instantiated, in such a case the indexical is constant and thus monotonic. So, the tell of a constraint containing an incorrect indexical term on X should be delayed until X has been

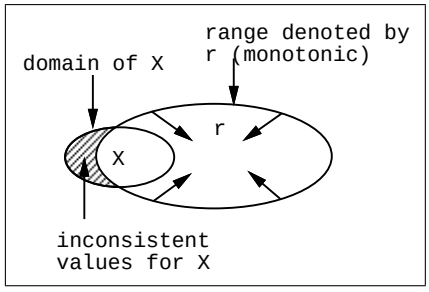

Figure 1. Telling a constraint X `in` r

instantiated. An elegant solution to achieve such a suspension is to move to the concurrent constraint framework and to use an *ask* mechanism [40], but staying in the CLP approach a simple solution is to use some well-known delay mechanism (`freeze`, `wait`, etc.) [27]. In our approach this is achieved using a new indexical term $val(X)$ that delays the activation of a constraint in which it occurs until X has been instantiated (see for example the below definition of $X \neq Y$).

2.3. Constraint Systems

The simplest way to define constraints is to consider them as first-order formulas interpreted in some non-Herbrand structure [26], in order to take into account the particular semantics of the constraint system. Such *declarative* semantics is adequate when a non-Herbrand structure exists beforehand and suitably fits the constraint system (e.g. $\mathcal{R}$ for arithmetic constraints), but it does not work very well for more practical constraint systems (e.g. finite domains). Obviously, it cannot address any operational issues related to the constraint solver itself. Recently, another formalization has been proposed by [41], which can be seen as a first-order generalization of Scott's *information sytems* [42]. The emphasis is put on the definition of an *entailment* relation (noted $\vdash$) between constraints, which suffices to define the overall constraint system. Such an approach is of prime importance in the framework of concurrent constraint languages, but is also useful for pure CLP, as it makes it possible to define a constraint system ex nihilo by giving the entailment relation and verifying some basic properties. The entailment relation is given by rules, and we can therefore define a kind of *operational semantics* of the entailment between constraints. This will be particularly useful when defining our propagation-based boolean constraint system, as the entailment relation will accurately represent how information is propagated between constraints.

Definition 2.6. [41]
A *constraint system* is a pair $(D, \vdash)$ satisfying the following conditions:

1. D is a set of first-order formulas closed under conjunction and existential quantification.
2. $\vdash$ is an *entailment* relation between a finite set of formulas and a single formula satisfying the following inference rules:

$$S, d \vdash d \ \ (\textbf{Struct}) \qquad \frac{S_1 \vdash d \quad S_2, d \vdash e}{S_1, S_2 \vdash e} \ \ (\textbf{Cut})$$

$$\frac{S, d, e \vdash f}{S, d \wedge e \vdash f} \ (\wedge \vdash) \qquad \frac{S \vdash d \quad S \vdash e}{S \vdash d \wedge e} \ (\vdash \wedge)$$

$$\frac{S, d \vdash e}{S, \exists X.\, d \vdash e} \ (\exists \vdash) \qquad \frac{S \vdash d[t/X]}{S \vdash \exists X.\, d} \ (\vdash \exists)$$

In $(\exists \vdash)$, X is assumed not free in S, e.

3. $\vdash$ is *generic*: that is $S[t/X] \vdash d[t/X]$ whenever $S \vdash d$, for any term t.

In order to build constraint systems, it suffices to define a *pre-constraint system* $(D, \vdash)$ satisfying only **(Struct)**, **(Cut)** and the genericity condition. Existential quantification and conjunction can be added in a straightforward way, as stated by the following theorem.

Theorem 2.1. [41] Let $(D', \vdash')$ be a pre-constraint system. Let D be the closure of D' under existential quantification and conjunction, and $\vdash$ the closure of $\vdash'$ under the basic inference rules. Then $(D, \vdash)$ is a constraint system.

As an important corollary, a constraint system can be constructed even more simply from any first-order theory, i.e. any set of first-order formulas. Consider a theory T and take for D the closure of the subset of formulas in the vocabulary of T under existential quantification and conjunction. Then one defines the entailment relation $\vdash_T$ as follows. $S \vdash_T d$ iff S entails d in the logic, *with the extra non-logical axioms of* T. Then $(D, \vdash_T)$ can be easily verified to be a constraint system.

Observe that this definition of constraint systems thus naturally encompasses the traditional view of constraints as interpreted formulas.

2.4. Entailment Relation

Here we define the entailment relation and prove that FD is a constraint system.

Definition 2.7. A store S *entails* a constraint $c \equiv X$ `in` r iff c is true in any store S' stronger than S, i.e.

$$S \vdash c \ \ \textit{iff} \ \ \forall S' \ \ S' \sqsubseteq S \ \Rightarrow \ X_{S'} \subseteq r_{S'}$$

A store S *disentails* a constraint $c \equiv X$ `in` r iff S entails $\neg c$, i.e. $S \vdash X$ `in -`r.

The following proposition states that $(Constr, \vdash)$ is a pre-constraint system (the proof can be found in [19]).

Proposition 2.1. the relation $\vdash$ (as defined in Definition 7) satisfies (**Struct**), (**Cut**) *and is generic.*

Thanks to Theorem 2.1, we can define D as the closure of $Constr$ under existential quantification and conjunction, and $\vdash^3$ as the closure of the entailment relation under the basic inference rules, then FD=$(D, \vdash)$ is a constraint system.

Finally, we can also define an equivalence between constraints to capture the fact that two constraints provide the same tuples of solutions.

Definition 2.8. two constraints c_1 and c_2 are *equivalent* iff $\forall S \;\; S \vdash c_1 \;\Leftrightarrow\; S \vdash c_2$.

3. HIGH-LEVEL CONSTRAINTS

From the basic X `in` r constraints, it is possible to define high-level constraints, called *user constraints*, as Prolog predicates. Each constraint specifies how the *constrained variable* must be updated when the domains of other variables change. In the `clp(FD)` system, basic user constraints are already defined via a library. CHIP-like constraints such as equations, inequations and disequations can be used directly by the programmer. A preprocessor will translate them at compile time. So, from a user point of view, `clp(FD)` offers the usual constraints over finite domains as proposed by CHIP together with the possibility to define new constraints in a declarative way and, from an implementation point of view, only the basic X `in` r has to be implemented.

3.1. Basic User Constraints

Let us now see how to define simple high-level constraints. In the following examples X, Y, Z are *FD variables* and C is a *term parameter* (runtime constant value).

Example 3.1.

```
'x=y+c'(X,Y,C):- X in min(Y)+C..max(Y)+C,
                 Y in min(X)-C..max(X)-C.

'x+y=z'(X,Y,Z):- X in min(Z)-max(Y)..max(Z)-min(Y),
                 Y in min(Z)-max(X)..max(Z)-min(X),
                 Z in min(X)+min(Y)..max(X)+max(Y).

'x≥y'(X,Y):-     X in min(Y)..infinity,
                 Y in 0..max(X).

'x≠y'(X,Y):-     X in -{val(Y)},
                 Y in -{val(X)}.
```

In the user constraint `'x≠y'(X,Y)` the constraint `X in -{val(Y)}` is delayed until Y is bound (in a *forward checking* manner, cf. [24]) as explained above.

[3] we reuse the same notation for this entailment relation to avoid heavy notations.

The propagation scheme used in the user constraint `'x=y+c'` is a *partial lookahead*, namely only changes on *min* and *max* of X and Y are propagated. A *full lookahead* scheme could be specified by:

Example 3.2.

```
'x=y+c'(X,Y,C):- X in dom(Y)+C,
                 Y in dom(X)-C.
```

3.2. Linear Arithmetic Constraints

We here describe how arithmetical constraints (equations, disequations, etc.) are managed in `clp(FD)`.

Definition 3.1. A *linear arithmetic constraint* is an expression $E.F$ where E and F are linear arithmetic expressions and $. \in \{=, \neq, <, \leq, >, \geq\}$.

The first step when compiling an arithmetic constraint consists in *normalizing* the constraint. Namely, a constraint $E.F$ is transformed in an equivalent form $S.T$ where $S = a_1 * x_1 + \ldots + a_k * x_k + c$ and $T = a_{k+1} * x_{k+1} + \ldots + a_n * x_n + d$. Each x_i is a distinct variable, each a_i is an integer > 0 (since FD only deals with natural numbers), c and d are two positive constants such that either c or d is equal to 0.

For instance the equation $2 * F + 2 * H - 20 = F + 3 * H - G - 10$ becomes $F + G = H + 10$. The normalization phase groups the terms according to variables. Doing this we obtain approximations (i.e. intervals *min..max*) which are much more accurate than those obtained when dealing with several occurrences of a same variable independently. Indeed, in this case *arc-consistency* would give rise to an approximation for each occurrence of a variable X (since it is not complete). The resulting approximation will then encompass the approximation associated to each occurrence of X. Performing a normalization step we obtain only one, more precise, approximation for X.

Each normalized term S and T is then sorted on its coefficients in descending way in order to add the constraints that achieve the larger pruning before the others. From a normal form, there are two main solutions to compiling a linear constraint: compilation to *inline code* or compilation to *library calls*.

3.2.1. Inline Code. An X `in` r expression is generated for every variable x_i (i.e. each variable is defined from the $n - 1$ others).

Example 3.3. The equation $F + G = H + 10$ will be translated as follows:

$F = H + 10 - G$	`F in min(H)+10-max(G)..max(H)+10-min(G)`	(c_F)
$G = H + 10 - F$	`G in min(H)+10-max(F)..max(H)+10-min(F)`	(c_G)
$H = F + G - 10$	`H in min(F)+min(G)-10..max(F)+max(G)-10`	(c_H)

The major drawback of such a method is that the code size generated is quadratic in the size of the input [8]. Another drawback is that a lot of redundant evaluations are made by all constraints. For instance, in $A + B + D = F + G + H + T$ if D is modified then $F + G + H + T$ is evaluated twice (to update A and B) and $A + B + D$ is evaluated four times (to update F, G, H and T). The last drawback is that each time a variable is modified, all constraints are triggered involving a reconsideration of all other variables. However, the modification of a variable often has no impact on other variables (due to the incompleteness of the arc-consistency). Since this compilation scheme cannot detect this situation it will awake uselessly $n - 1$ constraints. Consider again Example 3 in the following store:

$\{$`F in 0..15`, `G in 0..15`$\}$

giving:

$\{$`F in 0..15`, `G in 0..15`, `H in 0..20`, c_F, c_G, $c_H\}$

Suppose now that the constraint `F in 5..15` is added to the store. c_G is re-evaluated and the resulting range is $-5..30$ which already includes the current domain of G (which is thus not reduced). c_H is also re-evaluated and provides the range $-10..20$ which includes the current domain of H (which is not reduced). And so on for all other variables.

Because of all these drawbacks we have chosen the next alternative in `clp(FD)`.

3.2.2. Library calls. The main idea is to split the linear constraint into several linear constraints introducing intermediate variables. Each linear constraint gives rise to a call to a specific user constraint defined in a library. These basic constraints are very similar to those presented in Example 1.

Example 3.4. The equation $F + G = H + 10$ could be translated as follows:

$F + G = I$	`'x+y=z'(F,G,I)`
$I = H + 10$	`'x=y+c'(I,H,10)`

The code produced in this scheme is thus compact since it only consists of calls to user functions (i.e. predicates). However, the major advantage of this method stems from the introduction of intermediate variables which is a good way to both factorize computations and help propagation to focus on the relevant part of a constraint. Consider again Example 3 in the (same) store:

$\{$`F in 0..15`, `G in 0..15`$\}$

giving:

$\{$`F in 0..15`, `G in 0..15`, `I in 10..30`, `H in 0..20`,
$F + G = I$, $I = H + 10\}$

When the constraint `F in 5..15` is added to the store G is reconsidered from `min(I)-max(F)..max(I)-min(F)` = 0..25 that already includes the current domain of G (which is not reduced). I is checked from `min(F)+min(G)..max(F)+max(G)` = 5..30 and is not reduced since its current domain is included in 5..30. The propagation step is now finished and H has not been (uselessly) reconsidered. The impact of this optimization can be very great on complex arithmetical constraints.

Obviously there are many ways to decompose an expression [8]. Intuitively, if the decomposition is too fine there are too many intermediate variables (giving rise to an important overhead) and if the decomposition is too large we lose too many sources of factorization. On the other hand the larger the decomposition, the larger the library needed. In `clp(FD)` the decomposition is done by groups of three variables. Experiments have shown that this decomposition has good performance and requires a limited library.

4. INTEGRATION OF X `in` r INTO THE WAM

Let us now show how to implement the FD constraint system (i.e. the single constraint X `in` r) into the WAM. The reader is assumed to be familiar with basic notions of the WAM (see [50, 3] for a detailed presentation of the WAM). We first explain the extension of the WAM to deal with FD variables and after we detail how X `in` r constraints are managed.

4.1. Modifying the WAM for FD Variables

Here, we explain the necessary modifications of the WAM to manage a new data type: FD variables. They will be located in the heap, and an appropriate tag (`FDV`) is introduced to distinguish them from Prolog variables (see section 4.2.4 for the description of the data structure of an FD variable). Dealing with FD variables slightly affects data manipulation, unification, indexing and trailing instructions.

4.1.1. Data Manipulation. FD variables, like standard WAM unbound variables, cannot be duplicated (unlike for terms due to structure-copy). For example, loading an unbound variable into a register consists of creating a binding to the variable whereas loading a constant consists of really copying it. In standard WAM, thanks to self-reference representation for unbound variables, the same copy instruction can be used for both of these kinds of loading. There are two solutions to solving the problem of loading an FD variable:

- reuse the same scheme as in standard WAM (i.e. same copy instruction + self-reference). In this case the dereferentiation algorithm has to consider a tagged word `<FDV,`α`>` as a word `<REF,`α`>`. This slows down the dereferentiation algorithm since there are now two kinds of words that can record references.
- do not modify the dereferentiation algorithm and take care not to copy FD variables. Namely, when a source word W_s must be loaded into a destination word W_d, if W_s is an FD variable then W_d is bound to W_s or else W_s is physically copied into W_d.

In `clp(FD)`, we have chosen the second alternative to avoid penalizing portions of pure Prolog code since the dereferentiation is an operation performed very often. Also suppose we want to extend our engine with new constraint systems (e.g reals, sets, lists, etc.). Since all these new data types will require taking care of the loading, the first approach will really slow down the dereferentiation since there will be too many possible referencing words.

Note that a tagged word `<FDV,`*value*`>` of an FD variable X will be never dissociated from the other information of X (e.g. domain, etc.). Thus the *value* part is useless (or can be used to encode a part of the information needed for X). In `clp(FD)` we continue to use a self-reference since the presence of `<FDV,`α`>` *alone* allows us to determine the address of X (i.e. α). This simplifies the extension of all functions handling Prolog terms (e.g. display) which accept a tagged word as input.

4.1.2. Unification. An FD variable X can be unified with:

- an unbound variable Y: Y is just bound to X,
- an integer n: equivalent to X `in` $n..n$[4],
- another FD variable Y: equivalent to X `in dom(`Y`)` and Y `in dom(`X`)`.

4.1.3. Indexing. The simplest way to manage an FD variable is to consider it as an ordinary unbound variable and thus try all clauses. Obviously, doing more complex indexing based on the actual values of the domain would be useful, e.g. for optimizing the declarative definition of piecewise functions.

4.1.4. Trailing. In the WAM, unbound variables only need one word (whose value is fully defined by their address thanks to self-references), and can only be bound once, thus trailed at most once. These key properties make it possible to use a simple-entry trail. With FD variables these two properties no longer hold and a multiple-entry trail is needed.

Multiple-entry trail. A tagged trail is used to record the multiple values for FD variables (*min*, *max*, etc.). Hence we have three types of objects in the trail: one-word entry for standard Prolog variables, two-word entry for trailing one previous value, (n+2)-word entry for trailing n previous values (see Figure 1).

Avoiding useless trailing. As the domain of an FD variable is gradually reduced (in many intermediate steps), the standard (WAM) criterion for trailing would lead to much useless trailing. Indeed, only one trailing is necessary per choice point for an FD variable. We thus use the time stamp method of [1] that consists of adding a new register `STAMP` to number the choice points[5] and an extra cell per FD variable that refers to the choice point of its last trailing. Then, an FD variable X (including its stamp) needs to be trailed if `Stamp(`X`)` $\neq$ `STAMP`.

Note that another alternative consists in recording, in the frame associated to an FD variable, the address of its domain. When this domain is updated, if its address is below the last choice point it is copied on the top of the heap, the old address is

[4] we will describe later how constraints are managed.

[5] i.e. `STAMP` is incremented at choice point creation and decremented at choice point deletion.

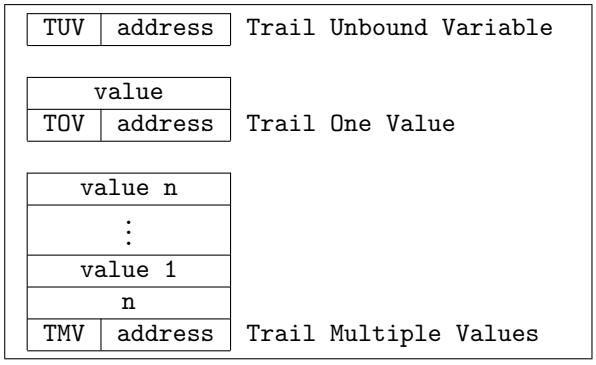

Figure 1. Trail frames

trailed and updated to point to this new copy. This solution has two advantages: it avoids using a new register (`STAMP`) and it only needs a one-value trail. Its drawback is that an indirection is needed to access the domain of a variable. However these two alternatives have more or less the same performance both in terms of memory space and execution time.

4.2. Implementing X `in` r Constraints

In this section we study how to implement X `in` r constraints. We describe the necessary new data stuctures and present an instruction set to compile X `in` r constraints. Let us detail the necessary operations to execute a constraint X `in` r and determine the data structures required:

evaluation of r: this concerns computing the range denoted by r, which implies recording the address of the (compiled) code that performs this evaluation. Since a range can depend on some arguments (i.e. indexical terms or parameters) it is necessary to record the context in which r must be evaluated. In other words we need to record a pointer to the *environment* where the code associated to r will take the values of the arguments it uses.

modification of X: this consists of updating X from the (previous) evaluation of r. This obviously implies recording a pointer to X.

propagation of the changes: reexecute all constraints depending on X. This implies recording, together with the domain of X, the list of constraints depending on X.

Figure 2 schematizes the data structures used to manage X `in` r constraints. All these data structures reside in the heap.

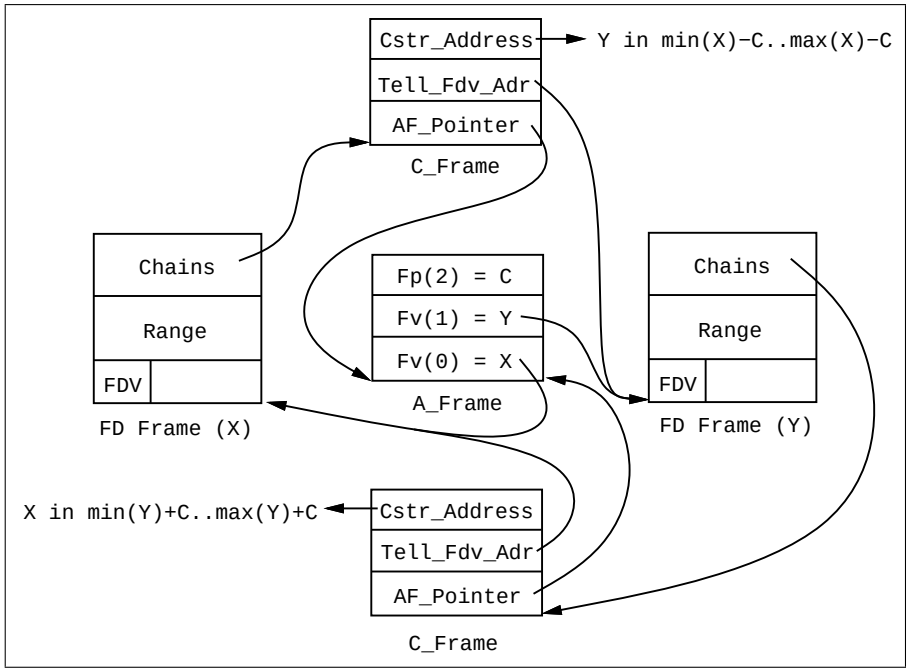

Figure 2. Data structures for X `in` r constraints

4.2.1. Representation of Environments. An argument frame (A_Frame) represents the environment in which the constraint is called, it records addresses of FD variables and values of parameters (see Figure 3). All the constraints defined in the same clause share the same A_Frame. A new register `AF` will point to the current A_Frame. In the following, FD variables will be referred to as $fv(i)$ (Frame Variable) and parameters as $fp(j)$ (Frame Parameter) where i and j are indices in the environment. For example, `'x=y+c'` (cf. Example 1) will be translated into the following pseudo-code:

Example 4.1.

```
'x=y+c'(X,Y,C):-
          create a 3 elements A_Frame,
          put X into A_Frame (fv(0)),
          put Y into A_Frame (fv(1)),
          put C into A_Frame (fp(2)),
          fv(0) in min(fv(1))+fp(2)..max(fv(1))+fp(2),
          fv(1) in min(fv(0))-fp(2)..max(fv(0))-fp(2).
```

4.2.2. Representation of Constraints A constraint frame (C_Frame) is created for every constraint. A new register `CF` will point to the current constraint. The information recorded in a C_Frame is as follows (see Figure 4):

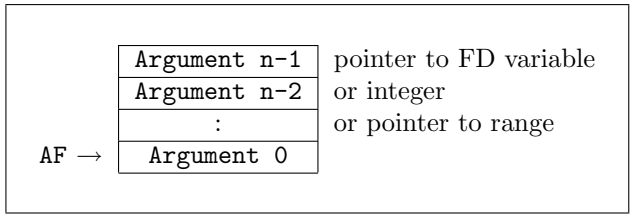

Figure 3. Argument frame (A_Frame)

- the pointer to the associated A_Frame,
- the address of the FD variable that is constrained[6],
- the address of the associated code.

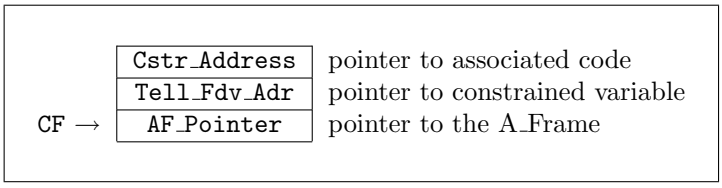

Figure 4. Constraint frame (C_Frame)

4.2.3. Representation of Ranges. There are two representations for a range:

- *Min-Max*: only the *min* and the *max* are recorded. This representation is then used to encode intervals (included in $0..infinity$).
- *Sparse*: in addition to the *min* and the *max* a bit-vector is used to record each value of the range (in the range $0..vector\_max$). By default *vector_max* equals 127 and can be redefined via an environment variable or via a built-in predicate. However, bit-vectors are not dynamic (all bit-vectors have the same size).

The initial representation for a range is always a *Min-Max* representation and is switched to a *Sparse* representation when a "hole" appears in the range (e.g. due to union, complementation, etc.). When a range R becomes *Sparse*, some values may be lost since *vector_max* is less than $infinity$. To detect these sources of imcompleteness, `clp(FD)` manages a flag for R which indicates that this range has been *extra constrained* by the solver (via an imaginary constraint X *in* $0..vector\_max$). The flag *extra_cstr* associated to each range is updated by all operations. For instance, the union of two ranges is extra-constrained if at least one range is extra

[6] for readers with short memories: that is X in the constraint X `in` r.

constrained, thus the resulting flag is the logical *or* between the two flags. When a failure occurs on a variable whose domain is extra constrained a message is displayed to inform the user that some solutions can be lost since bit-vectors are too small. This solution has been adopted since it is simple, correct and because the underlying algorithms can be implemented efficiently. It would be possible to use a representation with lists of intervals or to use bit-vectors with a base β to encode a $Sparse$ range in $\beta..\beta + vector\_max$. However, this would penalize the performance since in many cases the domains are compact and near to 0. In any case it is possible to state the problem with a translation.

Finally en empty range is represented with $min > max$. This makes it possible to perform an intersection between R_1 and R_2 in *Min-Max* mode simply with $max(min(R_1), min(R_2))..min(max(R_1), max(R_2))$ which returns $min > max$ if either R_1 or R_2 is empty. Figure 5 shows all possible representations of a range.

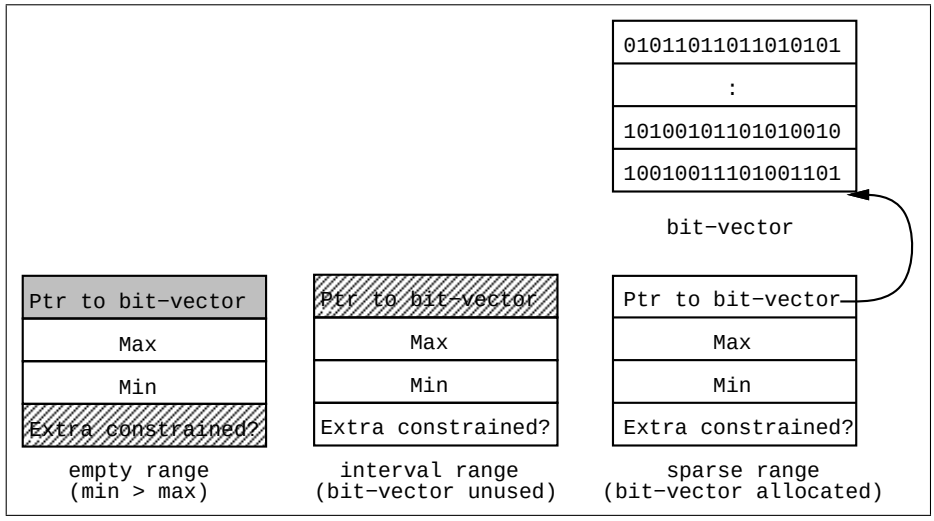

Figure 5. Representations of a range

4.2.4. FD Variable Frame. The frame associated to an FD variable X is divided into two main parts:

- the domain recording the range (see above) and the size of the range (number of elements currently present).
- the constraints depending on X (i.e. several lists of pointers to C_Frames).

These two parts are not modified at the same time. Chains are created when the constraints are *installed* whereas the domain can be updated during execution. Each part has its own stamp and can thus be trailed independently. Several distinct chains are used, in order to avoid useless propagation[7]:

[7]for instance, it is useless to reexecute a constraint depending on $min(X)$ when only $max(X)$ is changed.

- `Chain_Min`: list of constraints depending on `min(`X`)` and not on `max(`X`)`.
- `Chain_Max`: list of constraints depending on `max(`X`)` and not on `min(`X`)`.
- `Chain_Min_Max`: list of constraints depending on `min(`X`)` and on `max(`X`)`.
- `Chain_Dom`: list of constraints depending on `dom(`X`)`.
- `Chain_Val`: list of constraints depending on `val(`X`)`.

Figure 6 summarizes the information recorded for an FD variable. Figure 7 shows the data structures involved in `'x=y+c'`.

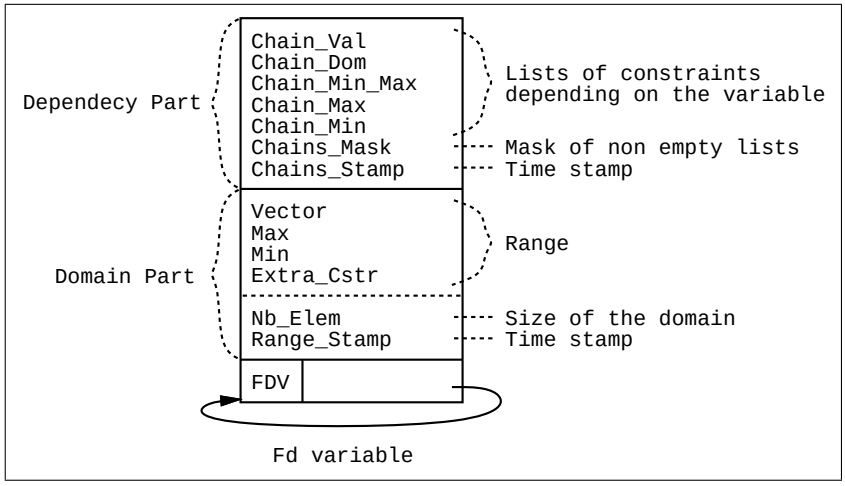

Figure 6. FD variable frame

4.2.5. Propagation Queue. As seen in Section 4.2, the propagation phase consists of awaking and executing a set of constraints that could themselves enrich this set by new constraints. As the overall order of execution is obviously irrelevant for correctness, we could thus either manage an explicit propagation queue (or stack, bag, heap, etc.) or handle an implicit continuation-based execution. This is very similar to the execution of goals in logic programs where one can choose between a Prolog depth-first search and a more complex handling of (active) goals in the resolvent, as in concurrent logic languages. We have chosen the latter and maintain an explicit propagation queue for reasons of flexibility (i.e. breadth-first search). The small overhead induced by this scheme is largely counterbalanced by the potential for order heuristics and optimizations (see below). Moreover, our experiments show that we reach the solution in this way more quickly than with a depth-first search. Two registers `BP` and `TP` point to the base and the top of the queue. A simple optimization consists in avoiding to enqueue all constraints but only a pair $< X, mask >$ where X is the updated variable (which has caused the propagation) and $mask$ is a bit-mask of dependency lists to awake (see Section 4.2.4).

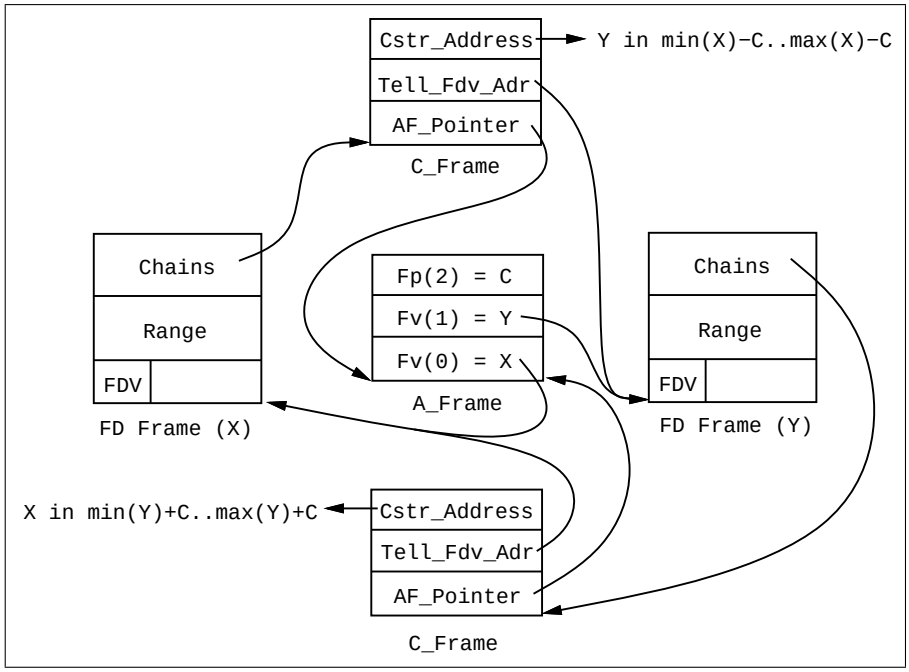

Figure 7. Data structures involved in `'x=y+c'`

4.2.6. Registers. In order to manage the previous data structures, we need to introduce new registers:

`BP`	: Base Pointer to the propagation queue.
`TP`	: Top Pointer to the propagation queue.
`AF`	: pointer to the current A_Frame.
`CF`	: pointer to the current C_Frame.
`CC`	: Continuation after Constraint.
`STAMP`	: choice point number.
`T(t)`	: Term registers.
`R(r)`	: Range registers.

As just mentioned, the registers `BP` and `TP` point the base and the top of the propagation queue. The `CC` register points to the next instruction to execute after the call constraint.

There is a bank of term registers (`T(t)`) and a bank of range registers (`R(r)`) used by the code evaluating the domain denoted by a range.

Remark that the standard WAM `CP` and `Ai` registers could be used instead of `CC`, `T(t)` and `R(r)`. But in that case they should be saved in the standard way (`allocate` and `deallocate` for `CP`, `try` and `retry` for `Ai` registers), while this can be avoided with these special registers and thus X `in` r constraints are compiled as *inline predicates*.

4.3. Compilation Scheme

In this section we give an overview of the compilation scheme. A complete description of the instruction set can be found in Appendix A.1.

The compilation of a clause which contains at least one X `in` r constraint gives rise to three groups of instructions:

Interface with Prolog clauses. Creates and loads an A_Frame. The space is reserved at the top of the heap and addresses of FD variables and values of parameters are loaded into this new A_Frame.

Installation code. Creates and loads a C_Frame. It also initializes the appropriate chain lists for all FD variables used by this constraint. For example, in the constraint $c \equiv X$ `in min(`Y`)..infinity`, the installation code will add a pointer to c in the list of constraints depending on the *min* of Y.

Constraint code. This code is decomposed into four parts:

- loading parameters, indexical terms and ranges into appropriate registers. Useless loading is avoided: for instance, if a constraint uses `dom(`X`)` and `min(`X`)`, only the domain of X needs to be loaded because it contains the *min* of X.
- evaluating the range r. The compilation of r is very easy, driven by the constraint syntax. The syntactical tree of r is traversed bottom-up: each leaf and each node give rise to a specific instruction. For leaves corresponding to indexical terms (or parameters), copy instructions are produced to set the appropriate registers from those loaded in the previous part. The final code can benefit from register optimization[8].
- telling X `in` r (i.e. updating X w.r.t. r). See section 4.4.
- returning.

Table 1 shows a fragment of code generated for our typical example `'x=y+c'`.

4.4. Telling the Constraint X `in` r

Here, we detail the work done to tell a constraint. For a constraint X `in` r we have the following possible behaviors:

If X is an integer, there are two possibilities:

- $X \in r$: success (*)
- $X \notin r$: failure

or else (X is an FD variable whose range is r_X) let r' be $r \cap r_X$:

- $r' = \emptyset$: failure
- $r' = r_X$ (i.e. $r_X \subseteq r$): success (*)

[8] in our compiler we reused the register allocation written for the WAM.

```
'x=y+c'/3:
    fd_set_AF(3,X(3))                    environment for X, Y, C
    fd_value_in_A_frame(X(0))            X is fv(0)
    fd_value_in_A_frame(X(1))            Y is fv(1)
    fd_term_parameter_in_A_frame(X(2))   C is fp(2)
    fd_install_constraint(inst_1,X(3))   install cstr_1
    fd_call_constraint                   call cstr_1
    fd_install_constraint(inst_2,X(3))   install cstr_2
    fd_call_constraint                   call cstr_2
    proceed                              Prolog return

inst_1:
    fd_create_C_frame(cstr_1,0)
    fd_install_ind_min_max(fv(1))        uses min(Y) and max(Y)
    fd_proceed                           install return

cstr_1:
    fd_ind_min_max(T(0),T(1),fv(1))      T(0)=min(Y), T(1)=max(Y)
    fd_term_parameter(T(2),fp(2))        T(2)=C
    fd_add(T(0),T(2))                    T(0)=min(Y)+C
    fd_add(T(1),T(2))                    T(1)=max(Y)+C
    fd_tell_interval(T(0),T(1))          X in min(Y)+C..max(Y)+C
    fd_proceed                           constraint return

inst_2:
    (...)
```

Table 1. Fragment of code generated for `'x=y+c'`

- otherwise: the domain of X is replaced by r' (X possibly becomes instantiated) and propagation occurs. Namely, as the domain of X has been modified, some constraints should be reexecuted. Here, we take advantage of having separate constraint chains (cf. Section 4.2.4). The current `CC` must be pushed onto the stack (local or global) to restore it after propagation.

It is worth noticing that the issues marked with (*) correspond to cases where the tell is useless (i.e. it neither fails nor reduces the domain of X). Thus, if we know how to detect such a tell, it becomes possible to avoid it, as we will see later.

5. IMPLEMENTATION RESULTS

5.1. Basic Implementation

`clp(FD)` is based on the `wamcc` Prolog compiler developed at INRIA [16]. Its novelty is to translate Prolog to C via the WAM. Predicates give rise to C functions, WAM instructions to C macros. Its performances are similar to Quintus Prolog

(halfway between Sicstus 2.1 emulated and Sicstus 2.1 native code). The extension to `clp(FD)` gives rise to C boolean functions for X `in` r constraints.

Several traditional benchmark programs have been used (the sources can be obtained by anonymous `ftp`, see Section 10):

- `crypta`: a cryptarithmetic problem on 10 variables ranging over 0..9, 2 over 0..1, 3 linear equations and 45 disequations [44].
- `eq10`: a system of 10 linear equations with 7 variables over 0..10.
- `eq20`: a system of 20 linear equations with 7 variables over 0..10.
- `alpha`: a cipher problem involving 26 variables over 1..26, 20 equations and 325 disequations.
- `queens`: the well-known N-queens problem [44] with N variables over 1..N and $3 * N * (N-1)/2$ disequations.
- `five`: the five houses puzzle [44] that involves 25 variables over 1..5, 11 linear equations, 50 disequations and 3 disjuctions of 2 linear equations.
- `cars`: the car sequencing problem of [20] with 10 variables over 1..6, 50 over 0..1, 49 inequations and 56 symbolic constraints (`element`, `atmost` [44]).

The programs `crypta`, `eq10`, `eq20` and `alpha` make it possible to test the efficiency of `clp(FD)` to solve linear equation problems. The other programs display the ability of `clp(FD)` to deal with forward checking constraints (`queens`), with disjunctions (`five`) and with symbolic constraints like `element` and `atmost` (`cars`). In all programs only the first solution is computed and the labeling is done in the standard way unless `ff` is stated which stands for *first fail principle* (i.e. favoring labeling on variable with smallest domain [44]).

We can compare this basic implemention of `clp(FD)` with other CLP systems over finite domains, such as the CHIP 3.2 system first developed at ECRC and then at COSYTEC. Exactly the same programs were run on both systems. The machine used for both `clp(FD)` and CHIP 3.2 was a Sparc station 2 (28.5 Mips). Performances of the architecture above presented are fairly good. The average speedup w.r.t. CHIP is around 1.5 for the linear equation programs and 3 for the other programs. Full measurements for this implementation can be found in Table 1.

However, if we analyze the decomposition of tells, we note that on average 75 % of the total number of tells are useless (the best case being `five` with 57 % and the worst case being `queens 70 ff` with 91 %). So we have studied where these useless tells originate from and we have designed three general optimizations to avoid some of them. We will evaluate the impact of each optimization on both the total number of tells and on the execution times. These statistics are shown in Table 2 and in Table 3.

5.2. Optimization 1

Many of useless tells stem from the fact that many constraints are *equivalent*, so it is not necessary to reexecute them. Intuitively, all constraints inside a single user constraint have the same declarative meaning and would lead to such a phenomenon, shown in the following example.

Program	CHIP 3.2	clp(FD) 2.1	speedup factor
crypta	0.120	0.090	1.33
eq10	0.170	0.110	1.54
eq20	0.300	0.170	1.76
alpha	61.800	9.290	6.65
alpha ff	0.280	0.160	1.75
queens 16	2.830	1.620	1.74
queens 64 ff	0.990	0.220	4.50
queens 70 ff	42.150	47.960	↓ 1.13
queens 81 ff	1.620	0.430	3.76
five	0.030	0.010	3.00
cars	0.120	0.040	3.00

Table 1. Basic version of `clp(FD)` vs. CHIP (in sec. on a Sparc 2)

Consider the constraint $X = Y + 5$, (`'x=y+c'(X,Y,5)`) with a current store:

{`X in 5..15`, `Y in 0..10`}

giving:

{`X in 5..15`, `Y in 0..10`
`X in min(Y)+5..max(Y)+5` (C_X),
`Y in min(X)-5..max(X)-5` (C_Y)}

Let us now show in detail what happens if the constraint `X in 12..100` is told. X is set to 12..15 and thus its *min* is propagated to Y via C_Y (`Y in 7..10`). Now, as the *min* of Y has been modified, C_X (`X in 12..15`) will be reexecuted giving rise to a useless tell (i.e. it does not modify the domain of X). Is is obviously useless to evaluate X from Y because Y has just been computed from X. C_X and C_Y are equivalent.

Consider a constraint $c \equiv X\ in\ r$ and an equivalent constraint $c' \equiv Y\ in\ r'$. Let $V \cup \{Y\}$ be the set of variables on which r depends, then obviously r' depends on $V \cup \{X\}$. If c has been executed due to a modification of Y then it is useless to call c' as it cannot reduce the domain of Y, because c and c' are equivalent. If not, (c has been executed due to some $Z \in V$) c' has also been enqueued in the propagation queue (due to Z). In both cases it is useless to enqueue c' once c has been executed.

Optimization 1: *telling c, it is useless to reexecute c' if c' is* equivalent *to c.*

In `clp(FD)`, we have designed all user constraints such as linear equations, inequations and disequations such that all constraints in the body of a user constraint definition are equivalent. We recall that all constraints defined in the same clause share the same A_Frame, and therefore to implement this optimization we only have to compare the current `AF` with the one used by the constraint to be called.

When this optimization is effective it makes it possible to save on average 18 % of tells and 12 % of the execution times (linear equations). In the worst cases, this optimization is ineffective (`queens`).

5.3. Optimization 2

Another source of useless tell stems from entailed constraints whose (re)executions are obviously useless since the tell operation is monotonic. Consider for example the constraint $X \neq Y$ (`'x≠y'(X,Y)`) with a store:

{`X in 1..10`, `Y in 1..10`}

giving:

{`X in 1..10`, `Y in 1..10`,
`X in -{val(Y)}` (C_X),
`Y in -{val(X)}` (C_Y)}

When X is set to 5, C_Y is awoken and 5 is removed from the domain of Y. Thus, the new store is:

{`X=5`, `Y in 1..4:6..10`, C_X, C_Y}

Suppose now that a constraint `Y=8` is told. After modification of the domain of Y, the propagation reexecutes C_X giving rise to a useless tell. Indeed C_X is now entailed (5 no longer belongs to the domain of Y).

Since an FD solver based on local propagation is not complete, it is not realistic to try to detect as soon as possible whether a constraint c is entailed (it would often imply enumerating the domain of the variables at each tell). So the best way consists in using an approximation of the entailment check that is stronger than the actual entailment condition. In `clp(FD)`, there is only one approximation condition to detect the entailment of X `in` r which is a *groundness* check, i.e. $S \vdash X$ `in` r whenever X is instantiated in S. For instance, in our previous example, when `Y=8` is told `X in -{val(Y)}` is detected to be entailed because X is instantiated. Obviously, we have to take care that X has become instantiated before calling the top-level constraint and not in the current propagation (i.e. all propagations due to the reduction of the domain of X have been done).

Optimization 2: *it is useless to reexecute* X `in` r *if* X *became instantiated before the top-level call constraint.*

To do this we use a new register (`DATE`) that is incremented at each constraint call. When a variable becomes instantiated it is dated with the current date. For this purpose a new cell is added to the FD variable frame.

Important remarks:

- for linear equations, the approximation X *instantiated* seems the best one since we cannot decide about the entailment of $X = Y$ before X (and thus Y) are instantiated.

- for inequations there are better approximations. For example, $X \geq Y$ is entailed as soon as the domain of X is fully greater than the domain of Y. Thus a better approximation would be $min(X) \geq max(Y)$.

  For disequations too there are better approximations. Indeed, $X \neq Y$ is entailed as soon as the domains of X and Y are disjoined. So $(min(X) > max(Y)) \vee (max(X) < min(Y))$ is a more accurate approximation.

  In these cases it is not necessary to wait until the constrained variable is instantiated to detect entailment.

- The better approximation conditions shown above can be produced automatically from the syntax of the constraint [9, 19]. We will integrate this facility when implementing the ask operator [40, 46].

- for some constraints the groundness approximation does not work (see for example the definition of `max(X,Y,Z)` in Section 6.3) . These constraints are then always reexecuted.

- it is possible to delete (and trail) an entailed constraint instead of dating it and testing its date each time it is invoked. In `clp(FD)` we prefer to keep it because we plan to implement some non-monotonic operations like the dynamic deletion of a constraint in order to be able to deal with reactivity.

In the best cases it makes it possible to save as many as 85 % of tells and 72 % of the execution time (`queens 70 ff`). However, for linear equations this optimization only makes it possible to save 18 % of tells while the impact on the execution time is limited to 8 %.

5.4. Optimization 3

Many useless tells result from the fact that we have in the propagation queue multiple occurrences of a single constraint awoken from several variables. Since the order in which constraints are executed is irrelevant, this leads to many redundant executions: only one instance of a constraint has to be present in the propagation queue at any one time.

Optimization 3: *if a constraint is already present in the propagation queue, it is useless to add it again.*

This can be achieved efficiently without scanning the whole propagation queue by using some simple dating technique reusing the `DATE` register introduced for optimization 2. A new cell is added to the constraint frame for dating the last call to a constraint.

Linear equation problems benefit from this optimization as follows: 26 % of tells and 17 % of the execution time saved. In the worst case (`queens`) this optimization is ineffective.

5.5. Final Results

Table 2 shows the impact of all the optimizations together on the number of tells. They make it possible to save on average 50 % of the total number of tells (45 % for

linear equation problems, 55 % for the other problems). Table 3 shows the impact on the execution times. These optimizations save on average 30 % of the execution time. Figure 1 shows the decomposition of the total time and of the total number of tells for both the basic and the final versions. In conclusion, there are some important remarks to make about our optimizations:

- The proportion of useless tells avoided is 66 % (see Figure 1). The remaining 33 % correspond to 50 % of the total number of tells in the final version (we recall that this proportion was around 75 % in the basic implementation). Among these 50 % we think that many of them could be avoided by a more precise entailment detection than the current optimization 2 (see remarks in Section 5.3).

- The execution time saved is on average 30 % to avoid 66 % of the total number of useless tells. So we can conclude that, in the basic version, the useless tells (75 %) correspond to 45 % of the total execution time (see Figure 1). We can explain this by the fact that the time spent in a useless tell of X `in` r is only the time necessary to evaluate the range r (since there are neither updating of X nor propagation). For constraints submitted to partial lookahead this evaluation only involves arithmetic operations on integers that are usually very fast. So the upper bound of the ability of the optimizations is 45 % on average. We think that our 30 % could be improved with a more precise entailment detection. It is worth noticing that for some problems the impact of the optimizations is very great. For example `queens ff 70` is around 4 times faster with the final version w.r.t the basic implementation.

- The overhead introduced by these optimizations is very small (around 1 %). See, for example, Table 3 where the `queens` program does not benefit from the optimization 3.

- These optimizations are general (vs. ad-hoc optimizations of black-box solvers). So any user constraint can benefit from them.

- In `clp(FD)`, the ratio between the time spent for a useful and for a useless tell is on average 3.5, i.e. a useless tell is 3.5 times faster than a useful tell on average. However, in other architectures (e.g. concurrent constraint), this ratio is closer to 1 since the evaluation of a constraint is often much more expensive due to context copy, garbage collection, etc. Thus, in these architectures the theoretical upper bound should be close to 75 % on average.

Table 4 shows the execution times for both CHIP 3.2 an `clp(FD)` on a Sparc station 2. We also include the `bridge` benchmark [44] (whose source was provided by COSYTEC) that involves 46 variables over 0..200, 91 linear inequations and 77 disjunctions of 2 linear inequations. On the linear equation problems the lowest speedup factor is 2 with peaks reaching 8. On the other programs `clp(FD)` is around 4 times faster than CHIP.

We can also compare `clp(FD)` with the CHIP compiler system. This compiler is not publicly available but [2] presents execution times for the `queens` and the `bridge` programs on a Sparc 1+ (18 Mips). We have thus normalized this times by a factor 1/1.6 to obtain a fair comparison with our timings measured on a Sparc

				Gain Analysis		
Program	Basic	Final	Gain	Opt. 1	Opt. 2	Opt. 3
`crypta`	8919	5047	43.41 %	22.67 %	14.80 %	26.93 %
`eq10`	15746	8739	44.50 %	20.18 %	12.99 %	29.42 %
`eq20`	24546	14483	41.00 %	11.90 %	14.29 %	26.12 %
`alpha`	904936	518793	42.67 %	14.97 %	26.32 %	17.63 %
`alpha ff`	15124	6978	53.86 %	21.81 %	21.43 %	33.54 %
`queens 16`	64619	31980	50.51 %	2.32 %	48.19 %	0.00 %
`queens 64 ff`	4556	2133	53.18 %	0.22 %	52.96 %	0.00 %
`queens 70 ff`	2009404	278826	86.12 %	0.67 %	85.45 %	0.00 %
`queens 81 ff`	10633	3748	64.75 %	0.37 %	64.38 %	0.00 %
`five`	566	345	39.05 %	13.60 %	29.15 %	3.36 %
`cars`	2483	1546	37.74 %	27.67 %	4.79 %	21.51 %

Table 2. Impact of the optimizations on the number of tells

				Gain Analysis		
Program	Basic	Final	Gain	Opt. 1	Opt. 2	Opt. 3
`crypta`	0.090	0.060	33.33 %	18.18 %	0.00 %	20.15 %
`eq10`	0.110	0.080	27.27 %	12.50 %	6.25 %	25.00 %
`eq20`	0.170	0.130	23.53 %	7.69 %	7.69 %	15.38 %
`alpha`	9.290	7.870	15.29 %	9.33 %	7.97 %	10.06 %
`alpha ff`	0.160	0.110	31.25 %	18.75 %	6.25 %	18.75 %
`queens 16`	1.620	1.010	37.65 %	0.71 %	33.33 %	↓ 1.40 %
`queens 64 ff`	0.220	0.170	22.73 %	0.00 %	22.22 %	0.00 %
`queens 70 ff`	47.960	12.650	73.62 %	0.50 %	71.88 %	↓ 1.10 %
`queens 81 ff`	0.430	0.290	32.56 %	0.00 %	31.43 %	0.00 %
`five`	0.010	0.010	0.00 %	0.00 %	0.00 %	0.00 %
`cars`	0.040	0.030	25.00 %	20.00 %	0.00 %	20.00 %

Table 3. Impact of the optimizations on the execution times

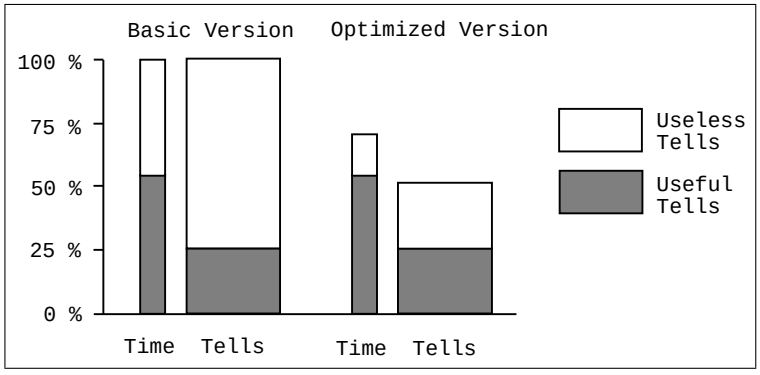

Figure 1. Comparison between the basic and the final version

Program	CHIP 3.2	clp(FD) 2.1	speedup factor
crypta	0.120	0.060	2.00
eq10	0.170	0.080	2.12
eq20	0.300	0.130	2.30
alpha	61.800	7.870	7.85
alpha ff	0.280	0.110	2.54
queens 16	2.830	1.010	2.80
queens 64 ff	0.990	0.170	5.82
queens 70 ff	42.150	12.650	3.33
queens 81 ff	1.620	0.290	5.58
five	0.030	0.010	3.00
cars	0.120	0.030	4.00
bridge	2.750	0.640	4.29

Table 4. clp(FD) vs. CHIP (in sec. on a Sparc 2)

2 (28.5 Mips). On these examples `clp(FD)` is still 3 times faster on average (see Table 5).

Program	CHIP compiler	`clp(FD)` 2.1	speedup factor
`queens 16 ff`	0.040	0.010	4.00
`queens 64 ff`	0.490	0.170	2.88
`queens 256 ff`	14.560	6.930	2.10
`bridge`	2.068	0.640	3.23

Table 5. `clp(FD)` vs. CHIP compiler (in sec. on a Sparc 2)

6. HANDLING DISJUNCTIVE CONSTRAINTS

The handling of disjunctive constraints is currently one of the major issues in CLP or CSP approaches, as disjunctive constraints appear in many real-life problems such as disjunctive scheduling, job-shop, bin-packing or spatial planning applications. The most simple and traditional approach for handling a disjunction of constraints in CLP is to use the non-determinism of the underlying logical engine, cf. [44], and therefore relies on choice-point creation. This is certainly very convenient from the programming point of view but leads to inefficiency and thrashing behavior because of the naive backtracking scheme of Prolog. Intelligent backtracking can provide improvement in some cases, see [10], but will be useless when the constraint network is strongly connected and constrained variables are all inter-linked to one-another, because in that case all choice-points are considered as pertinent.

Thus the most promising approach is to avoid creating choice-points and, when necessary, create them in a dynamic (data-driven) way rather than in a static (program driven) way. Such an approach is exemplified by the *Andorra principle*, proposed in [51], which is at the core of languages such as Andorra-I [18] and AKL [25], and favors deterministic computations by delaying non-determinate goals as long as some determinate one exists in the resolvent. Indeed, the roots of such a concept can be traced further back to the early developments of Prolog, as for instance in the "sidetracking" search procedure of [35] that favors the development of goals with the least alternatives. This is indeed nothing more than another variant of the well-known *first-fail* principle.

These ideas have been pushed a bit further in CLP by the definition of the *constructive disjunction* operator of [46]. The basic concept underlying this notion is to factorize the constraints entailed by all alternative branches and to add them to the store as soon as possible, without creating a choice point. This can be formalized, when constraint systems are defined as lattices [41], by considering a glb (greatest lower bound) operator between constraints in addition to the usual lub corresponding to conjunction: $glb(c_1, c_2) = \{c \,/\, c_1 \vdash c \wedge c_2 \vdash c\}$. Thus disjunctive constraints are used actively, even without creating choice-points. The power of this approach has been shown on real-life applications in [29]. However this mechanism can be quite costly in finite domain solvers, because disjunctive constraints have to be reconsidered after each propagation step that modifies some constrained

variable in order to extract some new commonly entailed constraint to tell. Obviously such a scheme could be exported back into logic programming to handle non-determinism, but it seems sensible only when branches of the disjunction are *flat*, i.e. are reduced to constraints and not to logical predicates (that could lead to unbound computations).

We will see that the FD constraint system provides a means of encoding a particular case of constructive disjunction for which the same pruning can be achieved in a much simpler and more efficient way. We will also see that nearly all current uses of constructive disjunction fit into that case.

6.1. A Simple Example

Lewis Carroll's well-known "five houses" or "zebra" puzzle has long been used for as a benchmark problem in the Prolog and CLP communities. The problem statement involves five men living in five houses. The men each have a different profession, nationality, favorite animal and favorite drink. There are fourteen facts describing various aspects of the situation and the problem consists of assigning everyone (and everything) to the right house and identifying the home of the zebra, etc.

The formulation of this problem in CLP [44] uses, for each of the five houses, a variable for the nationality, profession, animal and drink. The facts will be formulated as equality or disequality constraints between these variables. Among the fourteen facts, three lead to disjunctive constraints. For instance a fact like: "the Norwegian's house is next to the blue one" means that it can be either on the left or on the right. This will lead to a constraint of the form:

```
N5 = C4 + 1  or  N5 = C4 - 1
```

Therefore one has to introduce a predicate `plus_or_minus` defined by:

Example 6.1.

```
plus_or_minus(X,Y,C):- X = Y-C.
plus_or_minus(X,Y,C):- X = Y+C.
```

Such a predicate will therefore create a choice-point for each invocation. However a determinate predicate with identical declarative behavior but more efficient operational behavior can be defined in `clp(FD)` thanks to the *union* operation between ranges provided in the FD constraint system:

Example 6.2.

```
plus_or_minus(X,Y,C):- X in dom(Y)-C : dom(Y)+C,
                       Y in dom(X)+C : dom(X)-C.
```

Let us consider the constraint `plus_or_minus(X,Y,1)` with a store:

$\{$`X in 1..3, Y in 1..5`$\}$

The predicate defined in `clp(FD)` will remove the impossible value 5 from the domain of Y (and will never create any choice point), whereas the first definition will not. This corresponds to the behavior of a constructive disjunction. For the complete "five house puzzle", the second formulation is more than twice as fast as the first one (used in the above comparison with CHIP, see Table 4).

6.2. "United we stand, divided we fall"

The idea consists in defining a formula F from a formula $E \equiv c_1 \vee c_2 \vee ... \vee c_n$ so that there is no disjunction in F. Two cases are interesting:

(a) $E \Leftrightarrow F$: the addition of F to the store suffices and no choice point is needed.

(b) $E \Rightarrow F$: the addition of F to the store is not enough to ensure the correctness which will be then ensured by a choice point.

Let us study a case of **(a)** that occurs frequently. Consider a disjunction $E = c_1 \vee c_2 \vee ... \vee c_n$ where all constraints c_i have the form: X_1 `in` $r_1^i \wedge \ldots \wedge X_k$ `in` r_k^i so that 1) all constraints X_j `in` r_j^i are equivalent and 2) all constraints c_i contain exactly the same variables, say $\{X_1, ..., X_k\}$. Intuitively this corresponds to a disjunction of user constraints having all variables in common and where each user constraint is expressed as a conjunction of equivalent X `in` r constraints (e.g. above example `plus_or_minus`). Let us then define F from E as follows:

$$E \equiv \quad c_1 \vee \ldots \vee c_n \quad \equiv \bigvee \begin{array}{l} X_1 \texttt{ in } r_1^1 \wedge \ldots \wedge X_k \texttt{ in } r_k^1 \\ \ldots \\ X_1 \texttt{ in } r_1^n \wedge \ldots \wedge X_k \texttt{ in } r_k^n \end{array}$$

$$F \equiv \bigwedge \begin{array}{l} X_1 \texttt{ in } r_1^1 \vee \ldots \vee X_1 \texttt{ in } r_1^n \\ \ldots \\ X_k \texttt{ in } r_k^1 \vee \ldots \vee X_k \texttt{ in } r_k^n \end{array} \equiv \bigvee \begin{array}{l} X_1 \texttt{ in } r_1^1 : \ldots : r_1^n \\ \ldots \\ X_k \texttt{ in } r_k^1 : \ldots : r_k^n \end{array}$$

E and F are equivalent since in E all constraints of a conjunction are equivalent. We thus have a determinate formulation, as the disjunctive aspect is tackled at the range level by the union operator. Indeed, for a variable X_j the range associated to each alternative is evaluated and X_j is constrained to belong in the union of each of these ranges. Observe that, thanks to the usual propagation mechanism used in the FD system, ranges are recomputed as soon as one of their components is modified and therefore in our case a union will be recomputed as soon as one of its elements is modified, leading to a behavior identical to that of the constructive disjunction. However this treatment is much simpler and more efficient as it does not require telling the different alternative constraints (possibly performing a full propagation step) and taking the common instantiations.

Nevertheless such a decomposition is not possible for disjunctions between constraints that do not satisfy the above syntactic restrictions, i.e. involving different sets of variables, and we need a full constructive disjunction or rely on choice-point creation in those cases. Let us finally note that a compromise can be used that consists in deriving an approximation F from E such that $E \Rightarrow F$. In this case the

addition of F to the store is not enough to ensure the correctness and we will need to create a choice point. However the constraint F can be used to obtain an initial pruning and the choice point creation can be delayed (e.g. until the enumeration phase). For instance, from $E =$ `(X=4 ∧ Y=3) ∨ (X=8 ∧ Y=6)` we can define $F =$ `(X=4 ∨ X=8) ∧ (Y=3 ∨ Y=6)` so that $E \Rightarrow F$. The addition of F to the store will reduce the domain of X to $\{4, 8\}$ and the domain of Y to $\{3, 6\}$. The pruning obtained is great and makes it possible to delay the creation of a choice point.

6.3. Further Examples

Let us review the high-level constraints for which constructive disjunctions have been proposed and see whether they all satisfy the syntactic restrictions proposed above, so that they can be efficiently implemented in the FD constraint system.

6.3.1. Maximum Value. [46] proposes a constraint `max(X,Y,Z)` that holds iff Z is the maximum of X and Y. This can be expressed in the following way:

Example 6.3.

```
'max(x,y)=z'(X,Y,Z):- Z in min(X)..infinity,
                      Z in min(Y)..infinity,
                      Z in dom(X) : dom(Y).
```

The two first constraints ensure that Z is never less than X or Y and the third constraint ensures that Z is either X or Z. Such a constraint in the store:

$\{$`X in 5..10`, `Y in 7..11`, `Z in 1..12`$\}$

will reduce the domain of Z to 7..11.

6.3.2. Disjunctive Scheduling. In scheduling problems it is necessary to state that two tasks sharing the same resource cannot overlap, i.e that one is strictly before or after the other. Consider a task 1 starting at $T1$ with duration $D1$ and a task 2 starting at $T2$ with duration $D2$, the constraint to be expressed is:

$T1 + D1 \leq T2 \quad \vee \; T2 + D2 \leq T1.$

This translates in the FD constraint system as:

```
T1 in 0..max(T2)-D1 ∧ T2 in min(T1)+D1..infinity ∨
T2 in 0..max(T1)-D2 ∧ T1 in min(T2)+D2..infinity.
```

This can be defined in `clp(FD)` as follows:

Example 6.4.

```
disjunction(T1,D1,T2,D2):-
    T1 in 0..max(T2)-D1 : min(T2)+D2..infinity,
    T2 in 0..max(T1)-D2 : min(T1)+D1..infinity.
```

Let us consider the addition of the constraint `no_overlap(T1,4,T2,8)` in the store:

{`T1 in 1..10`, `T2 in 1..10`}

reducing the domain of $T1$ to $1..6 \cup 9..10$ and the domain of $T2$ to $1..2 \cup 5..10$.

6.3.3. Absolute Distance. Spatial planning problems often require dealing with absolute values in order to state distance constraints. For instance [47] defines a constraint $|X - Y| \geq C$ and CHIP proposes a `distance` built-in predicate. This can be expressed in `clp(FD)` in the following way, recalling the decomposition of the $\geq$ constraint:

Example 6.5.

```
'|x-y|>=c'(X,Y,C):- X in min(Y)+C..infinity : 0..max(Y)-C,
                    Y in min(X)+C..infinity : 0..max(X)-C.
```

Let us see in a simple example that this constraint achieves the same pruning as the definition of [47]. Consider the constraint `'|x-y|>=c'(X,Y,8)` with the store:

{`X in 1..10`, `Y in 1..10`}

The distance constraint will reduce the domains of X and Y to $\{1, 2, 9, 10\}$.

7. `clp(B/FD)`: A BOOLEAN CONSTRAINT LANGUAGE

The idea of considering booleans as a particular case of finite domains ({0,1}!) and of reusing local consistency techniques to solve boolean constraints was first introduced in the CHIP language. In fact, this approach was quite successful and it has become the standard tool in the commercial version of CHIP, whereas the special-purpose boolean solver of CHIP (based on boolean unification) is optional. An important by-product of this approach is that many extensions such as multi-valued logics [49] or pseudo-booleans (linear equations over booleans) [5] are available for free.

In CHIP, the particular propagation scheme of the boolean *and*, *or* and *not* constraints is, following the black-box approach, "wired" inside the solver and distinct from the finite domain part, although it uses some low-level routines. In `clp(FD)`, we can directly encode a boolean solver at the "user" level, thanks to the primitive constraint, and decompose boolean constraints such as *and*, *or*, and *not* in X `in` r expressions. In this way, we obtain a boolean solver that is obviously more efficient than the encoding of booleans with arithmetic constraints (at a lower-lever), and obviously more readable than the "wired" primitives of CHIP since the propagation scheme is coded in a constraint language and not in C. Worth noticing is that this boolean extension, called `clp(B/FD)`, is very simple; the overall solver (coding of boolean constraints into X `in` r expressions) being about ten lines long, the glass-box is very clear indeed. Moreover, this solver is surprisingly very efficient, being on average an order of magnitude faster than the CHIP solver, which was nonetheless reckoned to be efficient. Remark in passing that this is one more argument for the glass-box approach versus the black-box approach.

7.1. Boolean Solvers

Boolean problems have been tackled from a long time in various research areas, e.g. theorem proving or hardware circuit verification, and many boolean solvers have been developed, which are based on very different methods, such as SL-resolution, Davis/Putman-like enumeration algorithms, BDD-based methods, Operational Research-based approaches and more recently based on local propagation schemes, see [12] or [14] for a general review. It is also worth distinguishing between stand-alone solvers intended to take a set of boolean formulas as input, and CLP languages that offer much more flexibility by providing a full logic language to state the problem and generate the boolean formulas. Only PrologIII, CHIP and `clp(B/FD)` fall in the latter category.

As we will see later, `clp(B/FD)` is more efficient than CHIP and, surprizingly, it is also more efficient (several times faster) than such special-purpose solvers (see [12] or [14] for a comprehensive comparision).

7.2. Boolean Constraints

A *boolean constraint* on a set $\mathcal{V}$ of variables is one of the following formulas: $and(X,Y,Z)$, $or(X,Y,Z)$, $not(X,Y)$, $X = Y$, for $X,Y,Z \in \mathcal{V}$. The intuitive meaning of these constraints is: $X \wedge Y \equiv Z$, $X \vee Y \equiv Z$, $X \equiv \neg Y$, and $X \equiv Y$.

Observe that it is easy to enhance, if desired, this constraint system by other boolean constraints such as *xor* (exclusive or), *nand* (not and), *nor* (not or), $\Leftrightarrow$ (equivalence), or $\Rightarrow$ (implication) by giving the corresponding rules, but they can also be decomposed into the basic boolean constraints.

Therefore designing the boolean solver comes down to finding a user constraint for each boolean constraint. As the constraint X `in` r makes it possible to use arithmetic operations on the bounds of a domain, we use some mathematical relations satisfied by the boolean constraints:

$$
\begin{array}{lll}
and(X,Y,Z) & \text{satisfies} & Z = X \times Y \\
 & & Z \leq X \leq Z \times Y + 1 - Y \\
 & & Z \leq Y \leq Z \times X + 1 - X \\
 & & \\
or(X,Y,Z) & \text{satisfies} & Z = X + Y - X \times Y \\
 & & Z \times (1 - Y) \leq X \leq Z \\
 & & Z \times (1 - X) \leq Y \leq Z \\
 & & \\
not(X,Y) & \text{satisfies} & X = 1 - Y \\
 & & Y = 1 - X
\end{array}
$$

The definition of the solver is then quite obvious and presented in Table 1. It simply encodes the above relations.

It is easy to prove that such *and*, *or*, and *not* user constraints are correct and complete w.r.t. corresponding boolean operations by a simple case-analysis on truth-tables.

7.3. Performance Evaluations

In order to test the performances of `clp(B/FD)` we have tried a set of traditional boolean benchmarks:

```
and(X,Y,Z):-   Z in min(X)*min(Y)..max(X)*max(Y),
               X in min(Z)..max(Z)*max(Y)+1-min(Y),
               Y in min(Z)..max(Z)*max(X)+1-min(X).

or(X,Y,Z):-    Z in min(X)+min(Y)-min(X)*min(Y)..
                    max(X)+max(Y)-max(X)*max(Y),
               X in min(Z)*(1-max(Y))..max(Z),
               Y in min(Z)*(1-max(X))..max(Z).

not(X,Y):-     X in {1-val(Y)},
               Y in {1-val(X)}.
```

Table 1. The boolean solver definition

- `schur`: Schur's lemma. The problem consists of finding a 3-coloring of the integers $\{1 \ldots n\}$ such that there is no monochrome triplet (x, y, z) where $x + y = z$. The formulation uses $3 \times n$ variables to indicate, for each integer, its color. This problem has a solution iff $n \leq 13$.
- `pigeon`: the pigeon-hole problem consists of putting n pigeons in m pigeon-holes (at most 1 pigeon per hole). The boolean formulation uses $n \times m$ variables to indicate, for each pigeon, its hole number. Obviously, there is a solution iff $n \leq m$.
- `queens`: place n queens on a $n \times n$ chessboard such that there are no queens threatening each other. The boolean formulation uses $n \times n$ variables to indicate, for each square, if there is a queen on it.
- `ramsey`: find a 3-coloring of a complete graph with n vertices such that there are no monochrome triangles. The formulation uses 3 variables per edge to indicate its color. There is a solution iff $n \leq 16$.

Table 2 compares `clp(B/FD)` with the commercial version of CHIP (version 3.2) using propagation-based boolean constraints [9]. The same programs were run for both systems on a Sparc station 2. All solutions are computed unless if `first` is stated.

The average speedup of `clp(B/FD)` w.r.t. CHIP is around a factor of eight, i.e. an order of magnitude. This factor can be compared with the factor of four that we have on the traditional FD benchmarks. The main reason for this difference could be that in `clp(B/FD)` booleans are encoded at a lower level, thanks to the X `in` r primitive. Also a marginal gain can be attributed to the fact that the boolean constraints benefit from the general optimizations for primitive constraints, but this gain is limited to roughly 30 %.

Nevertheless, performances can be improved by simplifying the data-structures used in `clp(FD)`, which are designed for full finite domain constraints, and specializing them for booleans by explicitly introducing a new type and new instructions

[9] the other solver of CHIP, based on boolean unification, quickly became unpracticable: none of the benchmarks presented here could even run with it, due to memory limitations.

Program	CHIP 3.2	clp(B/FD) 2.1	speedup factor
schur 13	0.830	0.100	8.30
schur 14	0.880	0.100	8.80
schur 30	9.370	0.250	37.48
schur 100	200.160	1.174	170.49
pigeon 6/5	0.300	0.050	6.00
pigeon 6/6	1.800	0.360	5.00
pigeon 7/6	1.700	0.310	5.48
pigeon 7/7	13.450	2.660	5.05
pigeon 8/7	12.740	2.220	5.73
pigeon 8/8	117.800	24.240	4.85
queens 8	4.410	0.540	8.16
queens 9	16.660	2.140	7.78
queens 10	66.820	8.270	8.07
queens 14 first	6.280	0.870	7.21
queens 16 first	26.380	3.280	8.04
queens 18 first	90.230	10.470	8.61
queens 20 first	392.960	43.110	9.11
ramsey 12 first	1.370	0.190	7.21
ramsey 13 first	7.680	1.500	5.12
ramsey 14 first	33.180	2.420	13.71
ramsey 15 first	9381.430	701.106	13.38
ramsey 16 first	31877.520	1822.220	17.49

Table 2. clp(B/FD) vs. CHIP (in sec. on a Sparc 2)

for boolean variables. For instance, it is possible to reduce the variable frame representing the domain of a variable and its associated constraints to only two words: one pointing to the chain of constraints to awaken when the variable is bound to 0 and the other when it is bound to 1. Obviously time-stamps also become useless for boolean variables. Such a specialized solver is about twice as fast as `clp(B/FD)` and is described in [14]. Observe however that this gain is not really drastic when considering the level of encoding of the constraints: one expresses the propagation scheme in a "high-level" constraint language (X `in` r expressions) while the other requires the definiton of a new solver.

8. GENERALIZING THE CONSTRAINT X `in` r

8.1. Motivation

The X `in` r constraint gives us the possibility to define a range by functions over ranges (e.g. intersection, union, etc.) and functions over terms (e.g. addition, subtraction, etc.). It is worth noting that the definition of this set of allowed functions is mainly based on P. van Hentenryck's (great) experience of finite domain constraints. However, it seems natural to generalize the syntax of X `in` r constraints to allow any other functions over ranges or terms (see Table 1). Such functions are called *user functions* and their arguments are either ranges or terms. In `clp(FD)`, user functions are written in C for reasons of efficiency and because the underlying Prolog compiler already supports external functions in C. The following examples show the expressive power of user functions.

8.2. Magic Series

The magic series problem consists of finding a sequence of integers $\{x_0, \ldots, x_{n-1}\}$ such that each x_i is the number of occurrences of the integer i in the series [44]. The original formulation [44] used a `freeze` on each X_i. As presented in [34], we would like to simply encode the following relation:

$$x_i = \sum_{j=0}^{n-1} (x_j = i)$$

where $(x = y)$ is 1 if $x = y$ and 0 if $x \neq y$. This comes down to defining a user constraint `'x=a ⇔ b'(X,A,B)` where X is a domain variable, A a term parameter and B a boolean variable. The semantics of this user constraint is: $X = A$ *iff* B *is true* (i.e. $B = 1$). Operationally this constraint is active in two ways:

- as soon as $X \neq A$ (resp. $X = A$) B is set to 0 (resp. 1),
- as soon as $B = 0$ the value A is removed from the domain of X (constraining X to be different from A) and as soon as $B = 1$ X is set to A.

Obviously, the most elegant way of defining such a constraint is to use the (concurrent) ask primitive to write the four propagation rules. However, even in a CLP scheme we can define this constraint via two user functions as follows:

Example 8.1.

$c ::=$	X `in` r	
$r ::=$	$t_1..t_2$	(interval)
	$\{t\}$	(singleton)
	R	(range parameter)
	`dom(`Y`)`	(indexical domain)
	r_1 `:` r_2	(union)
	r_1 `&` r_2	(intersection)
	`-`r	(complementation)
	r `+` ct	(pointwise addition)
	r `-` ct	(pointwise subtraction)
	r `*` ct	(pointwise multiplication)
	r `/` ct	(pointwise division)
	f_r`(`a_1`,...,`a_k`)`	(user function)
$a ::=$	$r \mid t$	(user function argument)
$t ::=$	`min(`Y`)`	(indexical term min)
	`max(`Y`)`	(indexical term max)
	ct	(constant term)
	t_1`+`t_2 $\mid$ t_1`-`t_2 $\mid$ t_1`*`t_2 $\mid$ t_1`/<`t_2 $\mid$ t_1`/>`t_2	(integer operations)
	f_t`(`a_1`,...,`a_k`)`	(user function)
$ct ::=$	C	(term parameter)
	$n \mid$ `infinity`	(greatest value)
	ct_1`+`ct_2 $\mid$ ct_1`-`ct_2 $\mid$ ct_1`*`ct_2 $\mid$ ct_1`/<`ct_2 $\mid$ ct_1`/>`ct_2	

Table 1. Syntax of the generalized constraint X `in` r

```
'x=a  ⇔  b'(X,A,B):- B in x_to_b(dom(X),A),
                      X in b_to_x(val(B),A).
```

The user function `x_to_b` returns 1 if $X = A$, 0 if $X \neq A$ and 0..1 otherwise. The user function `b_to_x` is triggered as soon as B is instantiated and yields A if $B = 1$ or else the range $0..\infty \setminus A$. Note that the propagation scheme used here is a full-lookahead (any change of X or B involves the re-evaluation of this constraint). Obviously, a partial lookahead could use specifying `min(`X`)` and `max(`X`)` instead of `dom(`X`)`.

To show the power of such a constraint, let us compare this definition with that of CHIP which uses a `freeze` on each x_i. `clp(FD)` starts at about 4 times as fast as CHIP for small n and then grows up to a factor 460 (see Table 2). Obviously, there is less pruning in the CHIP definition.

Program	CHIP 3.2	clp(FD) 2.1	speedup factor
magic 10 ff	0.180	0.040	4.50
magic 20 ff	1.510	0.130	11.61
magic 30 ff	11.200	0.270	41.48
magic 40 ff	66.750	0.470	142.02
magic 50 ff	334.870	0.720	465.09

Table 2. Magic series problem (in sec. on a Sparc 2)

8.3. Atmost

The symbolic constraint `atmost(N,[X`$_1$`,...,X`$_m$`],V)` holds iff at most N variables X_i are equal to V. This constraint can be defined via the relation:

$$\sum_{j=0}^{n} (x_j = V) \ \leq \ N$$

Here too, a boolean B_i is associated to each variable x_i and is set to 1 if $x_i = V$ and to 0 if $x_i \neq V$ (via the constraint `'x=a` $\Leftrightarrow$ `b'` defined above). The sum of all Bi must be less than or equal to N. It is worth noticing that this constraint is "wired" in CHIP when it is defined as a user constraint (i.e. declaratively) in `clp(FD)` according to the glass-box paradigm.

More generally, it is possible to use a domain variable N to count the number of constraints that are true (or false) and then to constrain N by some FD constraints. The most interesting possibilities of the cardinality operator [45, 46] are then available in `clp(FD)`.

8.4. Non-Linear Equations

Usually non-linear equations are not supported directly by the solver since it is possible to delay the resolution of $X * Y = Z$ until either X or Y is instantiated to simply solve a linear equation. However, the only problem with $X * Y = Z$ is that X should be updated at each modification of Y or Z by the evaluation of Z/Y and thus we have to prevent the case $Y = 0$ (similarly for Z/X). Hence, it is possible to deal with non-linear equations in both a more declarative and efficient manner in executing $X = Z/Y$ only if $Y \neq 0$. So, let us define the constraint $X * Y = Z$:

Example 8.2.

```
'xy=z'(X,Y,Z):- X in div_e(min(Z),max(Y))..div_d(max(Z),min(Y)),
                Y in div_e(min(Z),max(X))..div_d(max(Z),min(X)),
                Z in min(X)*min(Y)..max(X)*max(Y).
```

The function $div\_e(x, y)$ (resp. $div\_d(x, y)$) returns $x /_{excess} y$ (resp. $x /_{default} y$) if $y \neq 0$ and 0 (resp. ∞) otherwise.

The pruning performed is much greater than the pruning obtained by delaying the evaluation until the equation is linear since $Y \neq 0$ is a weaker condition than $ground(Y)$. For example let us consider the constraint $X * Y = 110$ with the store: {`X in 1..40`, `Y in 6..30`}. As neither X nor Y is instantiated the constraint is delayed and thus the domains of X and Y are not reduced when the constraint `'xy=z'(X,Y,110)` would reduce the domain of X to 5..11 and the domain of Y to 10..22. Finding all solutions to this equation would involve the labeling phase to try 40 values for X with the delay version when only 7 would be tried with the partial-lookahead constraint `'xy=z'(X,Y,Z)`.

To show a bit more the flexibility of the generalization of X `in` r, let us consider the particular case $X = Y$ (i.e. $X^2 = Z$). The constraint `'xy=z'(X,X,Z)` with the store {`X in 1..100`, `Z in 5..24`} reduces the domain of X to 1..24 and does not modify the domain of Z. However, it is possible to improve this pruning since $X = \sqrt{Z}$. This is very similar to what occurs if the normalization step is not performed for linear equations. Indeed, since the underlying arc-consistency gives rise to approximations for domains, if all occurrences of a same variable X are handled separately each of them gives rise to an approximation for the domain of X. The result is a large approximation A that encompasses all sub-approximations. On the other hand, if all occurrences are "regrouped" (in a mathematical sense) then only one approximation is generated which is more accurate than A. Then, we define the user constraint $X^2 = Z$ as follows:

Example 8.3.

```
'xx=z'(X,Z):- X in sqrt_e(min(Z))..sqrt_d(max(Z)),
              Z in min(X)*min(X)..max(X)*max(X).
```

the function $sqrt\_e(x)$ (resp. $sqrt\_d(x)$) returns the integer square root of x rounded by excess (resp. default). In this case the constraint `'xx=z'(X,Y)` with the store {`X in 1..100`, `Z in 5..24`} reduces the domain of X to 3..4 and the domain of Z to 9..16.

9. CONCLUSION AND PERSPECTIVES

We have presented an abstract instruction set for a constraint solver over finite domains, which can be smoothly integrated into the WAM architecture. It is based on the idea of [46] of using a single primitive constraint X `in` r that embeds the core propagation mechanism, while complex constraints are compiled into X `in` r expressions.

Implementation results show that this approach is sound, combining as it does both simplicity and efficiency. Our `clp(FD)` system is about four times faster than CHIP on average, with peak speedup reaching eight. We have also shown that, following the glass-box approach, `clp(FD)` can be naturally enriched with various new constraints such as constructive disjunction, boolean constraints, non-linear constraints and symbolic constraints by using X `in` r decompositions. The boolean solver, for instance, performs fairly well, being eight times faster than the CHIP propagation-based solver and infinitely better than the CHIP boolean unification on usual boolean benchmarks [12].

Future work will involve integrating more complex constraints such as cardinality, full constructive disjunction, and a simple intelligent backtracking scheme on FD constraints [10].

Perspectives also include moving to the concurrent constraint framework [40] by defining a simple and efficient ask mechanism, and extending the constraint solver for incremental solving in reactive systems, i.e. for an intelligent handling of addition or deletion of constraints "from the outside" with minimal recomputation.

10. HOW TO GET `clp(FD)`

The `clp(FD)` system is available by anonymous `ftp` at `ftp.inria.fr` in the directory `/INRIA/Projects/ChLoE/LOGIC_PROGRAMMING/clp_fd`. The standard distribution also includes the boolean solver presented above and the sources of all benchmarks used in this paper.

ACKNOWLEDGEMENTS

We would like to thank Pascal Van Hentenryck for the initial idea of the glass-box and his helpful answers to our questions. We also thank Bjorn Carlsson, Mats Carlsson and Gregory Sidebottom for many fruitful discussions.

REFERENCES

1. A. Aggoun and N. Beldiceanu. Time Stamps Techniques for the Trailed Data in CLP Systems. In *Actes du Séminaire 1990 - Programmation en Logique*, Tregastel, France, CNET 1990.

2. A. Aggoun and N. Beldiceanu. Overview of the CHIP Compiler System. In *8th International Conference of Logic Programming*, Paris, France, MIT Press, 1991. Also in *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). MIT Press, 1993.

3. H. Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. Logic Programming Series, MIT Press, 1991.

4. BNR-Prolog User's Manual. Bell Northern Research. Ottawa, Canada, 1988.

5. A. Bockmayr. Logic Programming with Pseudo-Boolean Constraints. Research report MPI-I-91-227, Max Planck Institut, Saarbrucken, Germany, 1991.

6. R.E. Bryant, Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on computers*, no. 35 (8), 1986, pp 677–691.

7. W. Büttner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, no. 4 (1987), pp 191-205.

8. B. Carlsson, M. Carlsson. Constraint Solving and Entailment Algorithms for `cc(FD)`. Research Report, SICS, Sweden, 1993.

9. B. Carlsson, M. Carlsson, D. Diaz Entailment of Finite Domain Constraints. In *11th International Conference on Logic Programming*, Santa Margherita, Italy, MIT Press, 1994.

10. P. Codognet, F. Fages and T. Sola. A metalevel compiler for CLP(FD) and its combination with intelligent backtracking. In *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). MIT Press, 1993.

11. P. Codognet and D. Diaz. A Minimal Extension of the WAM for `clp(FD)`. In *10th International Conference on Logic Programming*, Budapest, Hungary, MIT Press, 1993.

12. P. Codognet and D. Diaz. Boolean Constraint Solving Using `clp(FD)`. In *International Logic Programming Symposium*, Vancouver, British Columbia, Canada, MIT Press, 1993.

13. P. Codognet and D. Diaz. `clp(B)`: Combining Simplicity and Efficiency in Boolean Constraint Solving. In *Programming Language Implementation and Logic Programming* Madrid, Spain, Springer-Verlag, 1994.

14. P. Codognet and D. Diaz. A Simple and Efficient Boolean Solver for Constraint Logic Programming. To appear in *Journal of Automated Reasoning*.

15. D. Chemla, D. Diaz, P. Kerlirzin and S. Manchon. Using `clp(FD)` to Support Air Traffic Flow Management. In *International Logic Programming Symposium Post-Conference Workshop on Constraint Languages/Systems and Their Use in Problem Modelling*, Ithaca, New-York, P. Lim and J. Jourdan (Eds.), 1994.

16. P. Codognet and D. Diaz. `wamcc`: Compiling Prolog to C. In *12th International Conference on Logic Programming*, Tokyo, Japan, MIT Press, 1995.

17. A. Colmerauer. An introduction to Prolog-III. *communications of the ACM*, 33 (7), July 1990.

18. Vítor Santos Costa, D. H. D. Warren, and Rong Yang. The Andorra-I engine: A parallel implementation of the basic andorra model. In *8th International Conference of Logic Programming*, Paris, France, MIT Press, 1991.

19. D. Diaz. *Etude de la compilation des langages logiques de programmation par contraintes sur les domaines finis : le système* `clp(FD)`. PhD thesis, University of Orleans, France, January 1995.

20. M. Dincbas, H. Simonis and P. Van Hentenryck. Solving the Car-Sequencing Problem in Constraint Logic Programming. In *ECAI-88*, Munich, W. Germany, August 1988.

21. M. Dincbas, H. Simonis and P. Van Hentenryck. Solving large combinatorial problems in Logic Programming. *Journal of Logic Programming*, 8 (1,2), 1990.

22. G. Dore and P. Codognet. A Prototype Compiler for Prolog with Boolean Constraints. In *GULP'93, Italian Conference on Logic Programming*, Gizzeria Lido, Italy, 1993.

23. G. Gallo, G. Urbani, Algorithms for Testing the Satisfiability of Propositional Formulae. *Journal of Logic Programming*, no. 7 (1989), pp 45-61.

24. R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence 14 (1980)*, pp 263-313

25. S. Haridi and S. Janson. Kernel Andorra Prolog and its computation model. In *7th International Conference of Logic Programming*, Jerusalem, Israel, MIT Press, 1990.

26. J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *Principles Of Programming Languages*, Munich, Germany, January 1987.

27. J. Jaffar and S. Michaylov. A Methodology for Managing Hard Constraints in CLP Systems. In *proceedings of Sigplan PLDI*, Toronto, Canada, ACM Press 1991.

28. J. Jaffar, S. Michaylov, P. J. Stuckey and R. Yap. An Abstract Machine for CLP($\mathcal{R}$). In *proceedings of Sigplan PLDI*, San Francisco, USA, ACM Press 1992.

29. J. Jourdan and T. Sola. The Versatility of Handling Disjunctions as Constraints In *Programming Language Implementation and Logic Programming*, Talin, Estonia, 1993.

30. A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence 8 (1977)*, pp 99-118.

31. U. Martin, T. Nipkow, Boolean Unification – The story so far. *Journal of Symbolic Computation*, no. 7 (1989), pp 191-205.

32. J-L. Massat. Using Local Consistency Techniques to Solve Boolean Constraints. In *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). MIT Press, 1993.

33. B. A. Nadel. Constraint Satisfaction Algorithms. *Computational Intelligence 5 (1989)*, pp 188-224.

34. W.J. Older, F. Benhamou Programming in clp(BNR). In *position Papers of 1st PPCP*, Newport, Rhode Island, 1993.

35. L. M. Pereira and A. Porto. Intelligent backtracking and sidetracking in horn clause programs. Technical Report CIUNL 2/79, Universitade Nova de Lisboa, 1979.

36. A. Rauzy. *L'Evaluation Sémantique en Calcul Propositionnel*. PhD thesis, University of Aix-Marseille II, Marseille, France, January 1989.

37. A. Rauzy. Adia. Technical report, LaBRI, Université Bordeaux I, 1991.

38. A. Rauzy. Using Enumerative Methods for Boolean Unification. In *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). MIT Press, 1993.

39. A. Rauzy. Some Practical Results on the SAT Problem. Draft, 1993.

40. V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Research Report CMU-CS-89-108, Carnegie-Mellon University, 1989. Also (revised) MIT Press, 1993.

41. V. Saraswat. The Category of Constraint Systems is Cartesian-Closed. In *Logic In Computer Science*, IEEE Press 1992.

42. D. S. Scott. Domains for Denotational Semantics. In *ICALP'82, International Colloquium on Automata Languages and Programming*, 1982.

43. H. Simonis, M. Dincbas. Propositional Calculus Problems in CHIP. ECRC, Technical Report TR-LP-48, 1990.

44. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, The MIT Press, 1989.

45. P. Van Hentenryck and Y. Deville. The Cardinality Operator: A new Logical Connective for Constraint Logic Programming. In *8th International Conference of Logic Programming*, Paris, France, MIT Press, 1991.

46. P. Van Hentenryck, V. Saraswat and Y. Deville. Constraint processing in cc(FD). Draft, 1991.

47. P. Van Hentenryck, V. Saraswat and Y. Deville. Design, Implementation and Evaluation of the Constraint language cc(FD). Draft, 1993.

48. P. Van Hentenryck, Y. Deville and C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence 57 (1992)*, pp 291-321.

49. P. Van Hentenryck, H. Simonis and M. Dincbas. Constraint Satisfaction Using Constraint Logic Programming. *Artificial Intelligence* no 58, pp 113-159, 1992.

50. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Oct. 1983.

51. D. H. D. Warren. The Andorra Principle. Internal Report, Gigalips Group, 1987.

A. INSTRUCTION SET

A.1. Interfacing with Prolog Clause

These instructions are responsible for creating and loading the A_Frame. Mainly, the space is reserved at the top of the heap, the addresses of FD variables and values of parameters are loaded into this new A_Frame.

`fd_set_AF(nb_arg,Vi)` reserves space, on the top of the heap, for an A_Frame whose size is `nb_arg`. `AF` and the `Vi` variable point to the start of the A_Frame.

`fd_variable_in_A_frame(Vj)` binds `Vj` to an FD variable created on top of the heap (whose range is $0..\infty$). Puts its address into the cell pointed by `AF`. `AF` is incremented.

`fd_value_in_A_frame(Vj)` let w be the dereferenced word of `Vj`, if it is:

- an unbound variable: similar to `fd_variable_in_A_frame(`w`)`.

- an integer: it is pushed on the heap and its address is stored into the cell pointed by `AF`. `AF` is incremented.
- an FD variable: its address is stored into the cell pointed by `AF`. `AF` is incremented.

`fd_range_parameter_in_A_frame(Vj)` the dereference of `Vj` must be a list of integers and a corresponding range is created on the top of the heap whose address is copied into the cell pointed by `AF`. `AF` is incremented.

`fd_term_parameter_in_A_frame(Vj)` the dereference of `Vj` must be an integer and its value is copied into the cell pointed by `AF`. `AF` is incremented.

Then, for every constraint, the following instructions are produced:

`fd_install_constraint(install_proc,Vi)` restores `AF` with `Vi`, sets `CC` to the next instruction and gives the control to the install procedure (described in the following section).

`fd_call_constraint` sets `CC` to the next instruction and gives the control to the code of the constraint pointed by `CF`.

A.2. Installing Constraints

For every constraint, an installation procedure is generated. It is responsible for creating and loading the C_Frame. It also initializes the appropriate chain lists for all FD variables used by this constraint.

`fd_create_C_frame(constraint_proc,tell_fv)` creates on the heap a C_Frame associated to the constraint whose code is located at `constraint_proc` and whose constrained variable is `tell_fv`. `CF` points to this C_Frame.

`fd_install_`{ `ind_min` | `ind_max` | `ind_min_max` | `ind_dom` | `dly_val` }`(fv)`

These instructions are used when the constraint (currently pointed by `CF`) uses the *min* (or max, or both *min* and max, etc.) of the `fv`*th* variable. So a new element is added to the appropriate chain list of the `fv`*th* variable.

`fd_proceed` gives the control to the address pointed by `CC`.

A.3. Computing Constraints

For every constraint X `in` r a constraint procedure is generated which is decomposed into four parts:

- loading parameters, indexical terms and ranges into appropriate registers,
- computing the range r,
- telling the constraint X `in` r,
- returning (by `fd_proceed` as above).

A.3.1. Loading parameters, indexical terms and ranges

`fd_range_parameter(R(r),fp)` loads the range pointed by the `fp`*th* parameter into `R(r)`.

`fd_term_parameter(T(t),fp)` loads the value of the `fp`*th* parameter into `T(t)`.

$\texttt{fd\_ind\_}\left\{\begin{array}{l}\texttt{min}\\ \texttt{max}\end{array}\right\}\texttt{(T(t),fv)}$ loads the $\left\{\begin{array}{l}\text{min}\\ \text{max}\end{array}\right\}$ of the `fv`*th* variable into `T(t)`.

`fd_ind_min_max(T(t1),T(t2),fv)` loads the *min* and the *max* of the `fv`*th* variable in `T(t1)` and `T(t2)`.

`fd_ind_dom(R(r),fv)` loads the domain (a range) of the `fv`*th* variable into `R(r)`.

`fd_dly_val(T(t),fv,lab_else)` if the `fv`*th* variable is an integer, it is copied in `T(t)`, or else the control is given to the label `lab_else`.

A.3.2. Computing the Range.

`fd_interval_range(R(r),T(t1),T(t2))` executes `R(r)`$\leftarrow$`T(t1)..T(t2)`.

$\texttt{fd\_}\left\{\begin{array}{l}\texttt{union}\\ \texttt{inter}\end{array}\right\}\texttt{(R(r),R(r1))}$ executes $\texttt{R(r)}\leftarrow\texttt{R(r)}\left\{\begin{array}{c}\cup\\ \cap\end{array}\right\}\texttt{R(r1)}$.

`fd_compl(R(r))` executes `R(r)`$\leftarrow 0..\infty \setminus$ `R(r)`.

`fd_compl_of_singleton(R(r),T(t))` executes `R(r)`$\leftarrow$`0..`$\infty \setminus \{$`T[t]`$\}$.

$\texttt{fd\_}\left\{\begin{array}{l}\texttt{add}\\ \texttt{sub}\\ \texttt{mul}\\ \texttt{div}\end{array}\right\}\texttt{\_in\_range(R(r),T(t))}$ executes $\texttt{R(r)}\leftarrow\texttt{R(r)}\left\{\begin{array}{c}+_{pointwise}\\ -_{pointwise}\\ *_{pointwise}\\ /_{pointwise}\end{array}\right\}\texttt{T(t)}$.

`fd_range_copy(R(r),R(r1))` executes `R(r)`$\leftarrow$`R(r1)`.

`fd_integer(T(t),n)` executes `T(t)`$\leftarrow$`n`.

$\texttt{fd\_}\left\{\begin{array}{l}\texttt{add}\\ \texttt{sub}\\ \texttt{mul}\\ \texttt{floor\_div}\\ \texttt{ceil\_div}\end{array}\right\}\texttt{(T(t),T(t1))}$ executes $\texttt{T(t)}\leftarrow\texttt{T(t)}\left\{\begin{array}{c}+\\ -\\ *\\ \lfloor/\rfloor\\ \lceil/\rceil\end{array}\right\}\texttt{T(t1)}$.

`fd_term_copy(T(t),T(t1)` executes `T(t)`$\leftarrow$`T(t1)`.

Telling the constraint X `in` r**.** The current constraint is pointed by `CF` and X can be reached from the C_Frame. So only r must be provided to `tell`. In order to optimize the execution we distinguish the case X *in* $t_1..t_2$ and the case X `in` r (with any r). A complete description of the tell operation is given in Section 4.4.

`fd_tell_range(R(r))` tells `X in r` where `r` is a range.

`fd_tell_interval(T(t1),T(t2))` tells `X in t1..t2` (i.e. `r` is an interval).