Compiling and Executing Disjunctions of Finite Domain Constraints

Björn Carlson
Computing Science Department, Uppsala University
Box 311
751 05 Uppsala
e-mail: `bjornc@csd.uu.se`

Mats Carlsson
Swedish Institute of Computer Science
Box 1263, S-164 28 KISTA, Sweden
e-mail: `matsc@sics.se`

Abstract We present two schemes for compiling disjunctions of finite domain constraints, where disjunction is treated as constructive. In the first scheme each disjunction is compiled to a set of indexicals, i.e. a set of range functions computing domain restrictions, such that the evaluation of the indexicals maintains a weak form of consistency of the disjunction. The second scheme is based on constraint lifting, i.e. constructive disjunction applied to the set of constraint stores given by executing a disjunction of goals, for which we provide an algorithm for lifting finite domain constraints. This scheme maintains stronger consistency than the first with a penalty in efficiency. We compare the two schemes with speculative disjunction, i.e. disjunction executed nondeterministically, and with disjunction via cardinality. Our conclusions are that the indexical scheme implements the most efficient pruning for many disjunctive constraints, such as resource and maximum/minimum constraints, and that the lifting scheme can be used for implementing lookahead pruning.

1 Introduction

Disjunctions of finite domain constraints can be used for pruning search [VHSD91, VHSD92, JS93]. However, instead of speculatively exploiting disjunctions as search directives, they should be handled as constraints proper. The basic idea is to propagate information common to each disjunct, where the problem is to compute what is common.

From a constraint such as $(x = 2 \wedge y = 1) \vee (x = 50 \wedge y = 25)$ it follows that $2y = x$. However, in the general case such inferences are intractable to compute. Rather, we focus on deriving domain constraints, e.g. from the above example we would like to infer $x \in \{2, 50\}$ and $y \in \{1, 25\}$.

Even with this restriction, we are still concerned with a multitude of approaches. Consider a disjunction $c_1 \vee c_2$ executed in conjunction with a constraint c in a store σ. Either, the domain constraints generated by considering the consistency of each constraint in c_1 and c_2 in σ can be combined, or the domain constraints given by enforcing the consistency of $c_1 \wedge c$ and $c_2 \wedge c$ in σ respectively can be combined. We say that either the disjunction is executed *locally* or *globally*.

In this paper we present two schemes for compiling disjunctions of finite domain constraints, one which generates code for maintaining local consistency, and one which maintains global consistency.

The first scheme is based on FD [VHSD91], a constraint language of indexicals. An indexical is a range function, indexed by the domains of variables, which computes restrictions on a domain variable. We use an extension of the original proposal of FD with conditional reasoning [CJH94] as the target language for the compilation of disjunctions of finite domain constraints. The extension is crucial for our purposes.

Hence, disjunctions of finite domain constraints are transformed into sets of (conditional) indexicals which by computing unions of ranges maintain consistency locally. This does not result in maximal propagation, but instead the indexicals can be evaluated efficiently. Another advantage of this technique is that it is based on indexicals with no extra support for disjunction, e.g. there is no need for managing local states which is necessary to implement global consistency.

The second scheme uses a concurrent language with deep guards, such as AKL [Jan94, CJH94], to execute disjunctions globally. Given a disjunction of constraints, each constraint is executed in a private local store, thus generating local domain constraints by maintaining consistency. Henceforth, domain constraints common to each local store are computed (lifted) and added to the embedding store. In this paper, we give an algorithm for lifting domain constraints, and briefly describe the implementation of constraint lifting in AKL.

The compilation schemes presented here have been implemented in the AGENTS-system, developed at SICS [JH91, Jan94, CJH94].

Our initial performance evaluations indicate that the indexical scheme implements sufficient pruning for many disjunctive constraints, such as resource and maximum/minimum constraints, extending the pruning given by cardinality-based disjunction. For our benchmarks the overhead of the lifting scheme for such constraints does not pay off, since it gives no more pruning than the indexical scheme.

On the other hand, the lifting scheme can be exploited for lookahead pruning [VH89], which decreases the number of nondeterminate steps dramatically for highly constrained problems such as graph-coloring and n-queens problems. This technique is not applicable using the indexical scheme.

The paper is structured as follows. In Section 2 we introduce indexicals and different types of disjunction. Section 3 deals with the compilation of arithmetical finite domain constraints to indexicals. In Section 4 the rules for rewriting disjunctions of constraints into sets of indexicals are given, and

$$\begin{array}{lcl}
N & ::= & x \mid i, \text{ where } i \in \mathcal{N} \mid \infty \\[6pt]
T & ::= & N \mid T+T \mid T-T \mid T*T \mid \lceil T/T \rceil \mid \lfloor T/T \rfloor \mid T \textbf{ mod } T \\
 & \mid & \textbf{min}(R) \mid \textbf{max}(R) \\[6pt]
R & ::= & T..T \mid R \cap R \mid R \cup R \mid R \supset R \mid -R \\
 & \mid & R+T \mid R-T \mid R \textbf{ mod } T \\
 & \mid & \textbf{dom}(N)
\end{array}$$

Figure 1: Syntax of FD range expressions

in Section 5 constraint lifting and lifting of domain constraints is explained. Section 6 contains a performance evaluation of the two schemes compared with speculative and cardinality-based disjunction.

2 Background

We now briefly explain FD, the different types of disjunction we consider, and constraint lifting in AKL.

2.1 FD: a theory of finite domain indexicals

The constraint system FD is based on *domain constraints* and functional rules called *indexicals* [VHSD91]. These rules may be thought of as rules for maintaining arc-consistency [Mac77]. We have extended FD with a conditional range operator which is necessary to treat disjunctions properly.

A domain constraint is an expression $x \in I$, where I is a set of integers. The sets that are considered will always be finite unions of intervals. A set σ of domain constraints is called a *store*. The expression x_σ denotes the intersection $I_1 \cap \cdots \cap I_n$ for all constraints $x \in I_k$ in σ, $1 \leq k \leq n$. If σ does not contain a constraint $x \in I$, x_σ is the set $\mathcal{Z}$ of integers. A variable x is *determined in* σ if x_σ is a singleton set. Let $\sigma_1 \sqsubseteq \sigma_2$, for stores σ_1 and σ_2, if for all variables x, $x_{\sigma_2} \subseteq x_{\sigma_1}$. It follows trivially that $\sqsubseteq$ defines a lattice.

An indexical has the form x **in** r, where r is a *range* (generated by R in Figure 1). The *value* of x **in** r in σ is $x \in r_\sigma$, where r_σ is the value of r in σ (see below). The *value of a range* r in σ, r_σ, is a set of integers computed as follows. The expression $\textbf{dom}(y)$ evaluates to y_σ. The expression $t_1..t_2$ is interpreted as the set $\{i \in \mathcal{Z}\colon {t_1}_\sigma \leq i \leq {t_2}_\sigma\}$. The operators $\cup$ and $\cap$ denote union and intersection respectively. The (new) conditional range $r \supset r'$ equals r'_σ if $r_\sigma \neq \emptyset$ and $\emptyset$ otherwise. The expressions $r + t$, $r - t$, and r **mod** t denote the integer operators applied pointwise, where t cannot contain **max** or **min** terms. Finally, the value of $-r$ in σ is the set $\mathcal{Z} \backslash r_\sigma$.

The *value of a term* t in σ, t_σ, is an integer computed as follows. A number is itself. A variable is evaluated to its assignment, if it is determined in σ. The interpretation of the arithmetical operators is as usual. The expressions $\mathbf{min}(r)$ and $\mathbf{max}(r)$ evaluate to the infimum and supremum values of r_σ, possibly $-\infty$ (∞).

In the following we use $t..$ and $..t$ as shorthand for $t..\infty$ and $-\infty..t$, $\mathbf{min}(x)$ and $\mathbf{max}(x))$ as shorthand for $\mathbf{min}(\mathbf{dom}(x))$ and $\mathbf{max}(\mathbf{dom}(x))$. Where nonambiguous, we use t instead of $t..t$.

A *linear finite domain constraint* $x \cdot y$, an *arithmetic* constraint for short, is an equation $(x = y)$, an inequation $(\cdot \in \{\leq, \geq, <, >\})$, or a disequation $(x \neq y)$ over linear expressions x and y, where a linear expression is either of the form $n_1 * x_1 \pm \ldots \pm n_k * x_k \pm n_0$, where n_i is a positive integer and x_i an integer variable $(0 \leq i \leq k)$, or of the form t/n, where t is linear and n a positive integer.

A *constraint* is thus an indexical, an arithmetic constraint a, or a disjunction (conjunction) thereof. The *value* of a constraint c in a store σ is defined by $\mathbf{eval}(c, \sigma)$ as follows:

- $\mathbf{eval}(x \ \mathbf{in} \ r, \sigma) = \sigma \sqcup \{x \in r_\sigma\}$.
- $\mathbf{eval}(a, \sigma) = \sigma \sqcup \{x_1 \in I_1, \ldots, x_k \in I_k\}$, for some integer sets $I_1, \ldots, I_k$, where $x_1, \ldots, x_k$ are the variables that occur in a. Each I_i can be computed through a translation of a as in Section 3.
- $\mathbf{eval}(c_1 \wedge c_2, \sigma) = \mathbf{eval}(c_1, \sigma) \sqcup \mathbf{eval}(c_2, \sigma)$.
- $\mathbf{eval}(c_1 \vee c_2, \sigma) = \mathbf{eval}(c_1, \sigma) \sqcap \mathbf{eval}(c_2, \sigma)$.

Furthermore, we define two fixed-point functions $\mathbf{L}$ and $\mathbf{G}$ for constraints in disjunctive normal form (*dnf*) as follows. Suppose c is in dnf, i.e. $c \equiv c_1 \vee \cdots \vee c_k$, for some k, where c_i is a conjunction of arithmetic constraints and indexicals.

- $\mathbf{L}(c_1 \vee \cdots \vee c_k, \sigma) = \sigma'$, where σ' is the smallest store extending σ such that $\mathbf{eval}(c_1, \sigma') \sqcap \cdots \sqcap \mathbf{eval}(c_k, \sigma') = \sigma'$.
- $\mathbf{G}(c_1 \vee \cdots \vee c_k, \sigma) = \sigma'$, where σ' is the smallest store extending σ such that $\mathbf{eval}(c_i, \sigma') = \sigma'$, $1 \leq i \leq k$.

Obviously, for any store σ and constraint c it follows that $\mathbf{L}(c, \sigma) \sqsubseteq \mathbf{G}(c, \sigma)$. The intuitive understanding of $\mathbf{L}$ and $\mathbf{G}$ is that $\mathbf{L}$ evaluates a disjunction without the need for local states to keep the results of propagating each disjunct separately, whereas this is necessary when evaluating $\mathbf{G}$. Furthermore, $\mathbf{G}$ suffers from the fact that σ' must be a fixed-point to each disjunct, which requires $\mathbf{G}$ to iterate more than $\mathbf{L}$.

Example 2.1. Consider the disjunction $c \equiv (x = y \wedge x = z \wedge y = 1) \vee (x = y \wedge x = z \wedge z = 1)$ in $\sigma = \{x \in \{1,2\}, y \in \{1,2\}, z \in \{1,2\}\}$. It follows that $\mathbf{G}(c, \sigma) = \{x \in \{1\}, y \in \{1\}, z \in \{1\}\}$, and $\mathbf{L}(c, \sigma) = \{x \in \{1,2\}, y \in \{1,2\}, z \in \{1,2\}\}$. □

Let σ be a store, and let c be a constraint. Entailment of c from σ is defined as:

- σ *entails* x **in** r if r_σ is defined, and $x_{\sigma'} \subseteq r_{\sigma'}$, for any σ' such that $\sigma \sqsubseteq \sigma'$.
- σ *entails* a if for each σ' such that $\sigma \sqsubseteq \sigma'$ and σ' determines the variables in a, a is true in σ'.
- σ *entails* $c_1 \wedge c_2$ if σ entails c_1 and c_2.
- σ *entails* $c_1 \vee c_2$ if σ entails either c_1 or c_2.
- c is *consistent* in σ if for some σ', $\sigma \sqsubseteq \sigma'$, σ' entails c.
- c is *inconsistent* in σ if c is not consistent in σ.
- c and d are *equivalent* if c is entailed iff d is entailed.

A range r is *monotone* if for every pair of stores σ_1 and σ_2 such that $\sigma_1 \sqsubseteq \sigma_2$, $r_{\sigma_2} \subseteq r_{\sigma_1}$. We define x **in** r as monotone if r is monotone.

2.2 Disjunction

We consider three types of disjunction; speculative, cardinality and constructive.

2.2.1 Speculative

Speculative disjunction, i.e. nondeterminate disjunction, is what is used in Prolog. Executing $c_1 \vee c_2$ in σ speculatively thus means to first execute c_1 in σ, and if failure later occurs, execute c_2 in σ instead. The problem with this scheme is that choices are made prematurely and that backtracking is needed to undo the effects of choices.

2.2.2 Cardinality

Cardinality-based disjunction is disjunction defined as

$$c_1 \vee c_2 \equiv \#(1, [c_1, c_2], 2)$$

[VHD91], i.e. at least one of c_1 or c_2 must be true. Hence, given a store σ, neither c_1 nor c_2 is executed in σ until the other is inconsistent in σ. The cardinality-operator is not speculative, but achieves insufficient propagation in many cases, typically for disjunctive scheduling problems.

2.2.3 Constructive

Constructive disjunction was proposed as a way to treat a disjunction of constraints as a constraint to avoid the speculative behavior, and to utilize the inherent propagation of disjunctions [VHSD91, VHSD92, JS93]. We only consider propagating domain constraints from a disjunction in the following.

t	$\mathit{inf}(t)$	$\mathit{sup}(t)$
n	n	n
x	$\mathbf{min}(x)$	$\mathbf{max}(x)$
$t_1 + t_2$	$\mathit{inf}(t_1) + \mathit{inf}(t_2)$	$\mathit{sup}(t_1) + \mathit{sup}(t_2)$
$t_1 - t_2$	$\mathit{inf}(t_1) - \mathit{sup}(t_2)$	$\mathit{sup}(t_1) - \mathit{inf}(t_2)$
$n * x$	$n * \mathit{inf}(x)$	$n * \mathit{sup}(x)$

Table 1: Upper and lower bounds of linear expressions

We distinguish between constructive disjunction executed locally and globally. Let c be in dnf. Then, executing c *globally* (*locally*) in a store σ is equivalent to evaluating $\mathbf{G}(c, \sigma)$ ($\mathbf{L}(c, \sigma)$).

3 Compilation of Arithmetic Constraints

In this section, we describe the compilation of linear finite domain constraints to monotone indexicals for constraint propagation.

Let *inf* (*sup*) be a function from linear expressions to values which increases (decreases) as the computation progresses. That is, $\mathit{inf}(t)$ ($\mathit{sup}(t)$) is the smallest (largest) value that t can ever get (see Table 1).

The *lower (upper) bound* of a linear expression E is thus computed by $\mathit{inf}(E)$ ($\mathit{sup}(E)$) (see Table 1).

3.1 Compilation of constraints

We give a simple-minded compilation of arithmetic constraints into indexicals, used as a basis for our compilation of disjunctions in later sections (see Section 4). The compilation of an arithmetic constraint c is based on deriving necessary conditions for c expressed as monotone indexicals. A constraint over k variables is compiled into k monotone indexicals over $k-1$ variables, which approximate the constraint by interval arithmetic reasoning, i.e. they maintain partial arc-consistency. This is similar to the distinction made between interval and domain reasoning of constraints in cc(FD) [VHSD92]. The scheme can be modified to provide full consistency by allowing arbitrary range arithmetics such as $r + r'$, where r and r' are ranges, as is done in clp(FD) for example [DC93b, DC93a]. The method is best explained by an example.

Example 3.1. The constraint

$$2x = 3y + 5$$

is rewritten twice to equivalent equations, each expressing a single variable as a function of the others:

$$\begin{aligned} 2x &= 3y + 5, \\ 3y &= 2x - 5 \end{aligned}$$

$\cdot$	r_i
$=$	$\lceil inf(E_i)/n_i \rceil .. \lfloor sup(E_i)/n_i \rfloor$
$\leq$	$.. \lfloor sup(E_i)/n_i \rfloor$
$\geq$	$\lceil inf(E_i)/n_i \rceil ..$
$<$	$.. \lfloor sup(E_i - 1)/n_i \rfloor$
$>$	$\lceil inf(E_i + 1)/n_i \rceil ..$
$\neq$	$-(\lfloor sup(E_i)/n_i \rfloor .. \lceil inf(E_i)/n_i \rceil)$

Table 2: Translation of arithmetic constraints

These equations are approximated by the indexicals:

$$x \ \textbf{in} \ \lceil (3 * \textbf{min}(y) + 5)/2 \rceil .. \lfloor (3 * \textbf{max}(y) + 5)/2 \rfloor,$$
$$y \ \textbf{in} \ \lceil (2 * \textbf{min}(x) - 5)/3 \rceil .. \lfloor (2 * \textbf{max}(x) - 5)/3 \rfloor$$

$\square$

The compilation of inequations and disequations is completely analogous to that of equations. In general, the idea is to rewrite a constraint over the variables $x_1 \ldots x_k$ into the equivalent constraints

$$n_i * x_i \cdot E_i$$

where $\cdot$ is the relation symbol, $1 \leq i \leq k$, and then to translate them into a conjunction of monotone indexicals

$$x_i \ \textbf{in} \ r_i$$

that propagate information whenever the **min** or the **max** of a variable changes, where $1 \leq i \leq k$ and r_i is defined from E_i and n_i (see Table 2).

The translation rules (see Table 2) are obtained as follows: a necessary condition for $n * x \leq E$ is the indexical x **in** $..\lfloor sup(E)/n \rfloor$; a necessary condition for $n * x \geq E$ is the indexical x **in** $\lceil inf(E)/n \rceil..$; and the following equivalences hold:

$$\begin{aligned}
x = y &\equiv x \leq y \wedge x \geq y \\
x < y &\equiv x \leq y - 1 \\
x > y &\equiv x \geq y + 1 \\
x \neq y &\equiv x < y \vee x > y \\
x \ \textbf{in} \ r_1 \wedge x \ \textbf{in} \ r_2 &\equiv x \ \textbf{in} \ r_1 \cap r_2 \\
x \ \textbf{in} \ r_1 \vee x \ \textbf{in} \ r_2 &\equiv x \ \textbf{in} \ r_1 \cup r_2 \\
x \ \textbf{in} \ ..(i-1) \cup (j+1).. &\equiv x \ \textbf{in} \ -(i..j)
\end{aligned}$$

Note that the proposed translation produces indexicals such that any pair of indexicals generated from an arithmetic constraint are equivalent, i.e. one of the indexicals is entailed iff the other one is.

This compilation scheme has one major drawback: the code size is quadratic in the size of the input. This property is probably unacceptable except in toy programs or for binary constraints, and can be removed by using conjunctions of library calls instead [DC93a].

4 Disjunctions executed locally

Let c be a constraint in dnf. Furthermore, let $x_1, \ldots, x_k$ be the variables that occur in c. *Indexing* c amounts to computing a set of monotone indexicals that evaluates $\lambda\sigma.\mathbf{L}(c, \sigma)$ (see Section 2.1).

Let A denote the set of indexicals generated by compiling a (see Section 3), where a is an arithmetic constraint.

We proceed stepwise as follows:

1. Assume c is equal to $c_1 \vee \ldots \vee c_n$, where each c_i is a conjunction of arithmetic constraints. Let $A_i = \bigcup\{A{:}\, a \in c_i\}$.

2. Define Y_i as A_i where any two indexicals x **in** r and x **in** r' have been replaced by x **in** $r \cap r'$. Hence, $Y_i = \{y_1 \textbf{ in } r_{i1}, \ldots, y_l \textbf{ in } r_{il}\}$, for some l and $y_1, \ldots, y_l \in \{x_1, \ldots, x_k\}$.

3. Let $V_i = \{x_1, \ldots, x_k\} \backslash \{y_1, \ldots, y_l\}$, and thus $X_i = Y_i \cup \{x \textbf{ in } -\infty..\infty{:}\, x \in V_i\}$.

Hence, X_i is the result of indexing conjunct c_i in c. We now turn to how the disjunction of $c_1, \ldots, c_n$ can be removed. Again, we proceed stepwise.

1. Let X_i be as above, i.e. $X_i = \{x_1 \textbf{ in } r_{i1}, \ldots, x_k \textbf{ in } r_{ik}\}$, $1 \leq i \leq n$. We define s_i as

$$s_i = (\mathbf{dom}(x_1) \cap r_{i1}) \supset \ldots \supset (\mathbf{dom}(x_k) \cap r_{ik}),$$

   and r_i as

$$r_i = (s_1 \supset r_{1i}) \cup \ldots \cup (s_n \supset r_{ni}).$$

2. Consequently, let X_c be the set $\{x_1 \textbf{ in } r_1, \ldots, x_k \textbf{ in } r_k\}$.

It can be proven that X_c is entailed if c is entailed, and that the inverse is not true. Furthermore, it follows that $\mathbf{L}(c, \sigma) = \mathbf{L}(x_1 \textbf{ in } r_1 \wedge \cdots \wedge x_k \textbf{ in } r_k, \sigma)$, where $X_c = \{x_1 \textbf{ in } r_1, \ldots, x_k \textbf{ in } r_k\}$.

When compiling arithmetic constraints, indexicals are generated which are equivalent to each other (Section 3). This can be used for optimizing the ranges generated by the compilation.

Let X_c be as above. X_c is optimized by removing redundant conditional ranges. There are several cases to apply:

s_i For each s_i, $1 \leq i \leq n$, replace any two ranges $\mathbf{dom}(x) \cap r$ and $\mathbf{dom}(y) \cap r'$, such that x **in** r and y **in** r' are generated from the same arithmetic constraint a, by $\mathbf{dom}(x) \cap r$.

r_i For any indexical x_i **in** $r_0 \cup ((\mathbf{dom}(x) \cap r) \supset r') \cup r_1$, $1 \leq i \leq k$, replace $(\mathbf{dom}(x) \cap r) \supset r'$ with r' if x_i **in** r' and x **in** r are generated from the same arithmetic constraint a.

$-\infty..\infty$ A conditional range such as $(\mathbf{dom}(x) \cap -\infty..\infty) \supset r'$ is replaced by r', for any x, since $\mathbf{dom}(x) \cap -\infty..\infty$ is nonempty in any consistent store.

In the examples below, the reductions are applied beforehand to reduce the complexity of the indexicals. There are other optimizations concerned with using intermediate variables for storing range values used multiple times, and for optimizing the evaluation of conditional ranges that we do not go into here.

Let us now consider a few examples of disjunctive constraints compiled into indexicals.

Example 4.1. Let c be the constraint $x + i \leq y \vee y + j \leq x$, for some constants i and j. The constraint is indexed into the two sets

$$\{x \textbf{ in } ..(\mathbf{max}(y) - i), y \textbf{ in } (\mathbf{min}(x) + i)..\}, \text{ and}$$

$$\{x \textbf{ in } (\mathbf{min}(y) + j).., y \textbf{ in } ..(\mathbf{max}(x) - j)\}.$$

These sets are conditioned and removed of redundant conditions computing X_c as

$$\begin{array}{l} x \textbf{ in } ..(\mathbf{max}(y) - i) \cup (\mathbf{min}(y) + j).. \\ y \textbf{ in } (\mathbf{min}(x) + i).. \cup ..(\mathbf{max}(x) - j) \end{array}$$

Note that c is a typical scheduling constraint, stating that either x preceeds y by some constant, or y preceeds x by some constant. The intended behavior is to exploit the disjunction constructively, which the indexicals do. The domain of x is effectively pruned of any integer in the interval $[\mathbf{max}(y) - i - 1, \mathbf{min}(y) + j + 1]$, and similarly the domain of y is pruned of any integer in the interval $[\mathbf{max}(x) - j - 1, \mathbf{min}(x) + i + 1]$ □

Example 4.2. Let c be the constraint $(x = 1 \wedge y = i_1) \vee (x = 2 \wedge y = i_2)$. The constraint is indexed into

$$\{x \textbf{ in } 1, y \textbf{ in } \mathbf{dom}(i_1)\} \text{ and } \{x \textbf{ in } 2, y \textbf{ in } \mathbf{dom}(i_2)\},$$

where the indexicals for i_1 and i_2 are ignored. Thus, $X_c =$

$$\begin{array}{l} x \textbf{ in } ((\mathbf{dom}(y) \cap \mathbf{dom}(i_1)) \supset 1) \cup ((\mathbf{dom}(y) \cap \mathbf{dom}(i_2)) \supset 2), \\ y \textbf{ in } ((\mathbf{dom}(x) \cap 1) \supset \mathbf{dom}(i_1)) \cup ((\mathbf{dom}(x) \cap 2) \supset \mathbf{dom}(i_2)) \end{array}$$

where no optimization rules are applicable.

Consider for a moment the element(x, l, y) constraint which is true iff the xth element in l is equal to y [DSH88, CJH94]. The constraint is equivalent to

$$\bigvee_j (x = j \wedge y = i_j)$$

where $l = [i_1, \ldots, i_k]$ and i_j is assumed to be determined, $1 \leq j \leq k$, of which c is a particular case where $k = 2$. It was previously shown that element/3 can be defined in terms of cardinality disjunction [HSD92], however, under the assumption that i_j is determined. Thus, our approach is slightly more general since i_j need not be determined. □

Example 4.3. Let c be the constraint $x = z \vee y = z$. Hence, $X_c =$

$$\begin{array}{l} x \textbf{ in } ((\textbf{dom}(y) \cap \textbf{dom}(z)) \supset -\infty..\infty) \cup \textbf{dom}(z), \\ y \textbf{ in } ((\textbf{dom}(x) \cap \textbf{dom}(z)) \supset -\infty..\infty) \cup \textbf{dom}(z), \\ z \textbf{ in } \textbf{dom}(x) \cup \textbf{dom}(y) \end{array}$$

which for example in the store $\{x \in \{1, 2\}, y \in \{3, \ldots, 6\}, z \in \{6\}\}$ propagates $y \in \{6\}$. Conjoining X_c with $x \leq z$ and $y \leq z$ compiled to indexicals (see Section 3) thus gives a more powerful max/3 constraint than in clp(FD) [DC93a]. □

5 Disjunctions executed globally

First we define constraint lifting, as implemented in AKL, and then we give an algorithm for lifting domain constraints in FD.

5.1 Constraint lifting

Constraint lifting is constructive disjunction applied to a set of constraint stores generated by running a disjunction of goals in separate local stores, where constraints true in each local store are lifted and added to the embedding store. The notion is generic in choice of constraint system, and computing approximations of disjunctions generally requires domain specific knowledge. However, for some constraint systems, such as boolean equalities, where the constraint language supports disjunction, lifting becomes trivial.

In AKL, a deep concurrent constraint language [Jan94], we have introduced constraint lifting through a deep guard operator ||, explained below. Any computation in AKL is done in a *local* constraint store. A hierarchy of stores is created by running goals in guards. Each local store is associated with all the constraints generated by the local execution, that constrain or depend on variables in external stores. Thus, AKL supports directly the structures that are necessary to implement constraint lifting, since a representation is kept which gives access both to constraint stores and to the constraints visible externally in each store.

The operator || is defined as an adaption of the guard mechanism of AKL [Jan94]. Thus; the *lifting statement* in AKL

$$\begin{array}{ll} & G_1 \parallel B_1 \\ ; & \ldots \\ ; & G_n \parallel B_n \end{array}$$

is used for expressing constraint lifting. Its components are called *(guarded) clauses* and the components of a clause *guard* (G_i) and *body* (B_i), where G_i and B_i contain procedure calls which include constraints.

Now, in the following let α be a function such that, given the constraint stores $\sigma_1, \ldots, \sigma_k$, $\alpha(\sigma_1, \ldots, \sigma_k) \sqsubseteq \sigma_i$ holds, for any i between 1 and k. For an instance of α see Section 5.2.

Suppose a lifting statement is executed in a store σ. Its guards are executed separately and the execution of the statement proceeds as follows.

- Let σ_i be the store resulting from the execution of guard G_i in σ.
- If σ_i is unsatisfiable, the guard fails, and the corresponding clause is deleted. If all clauses are deleted, the lifting statement fails.
- If only one nonfailed clause remains, i say, σ is replaced with σ_i, and the lifting statement is replaced with the body B_i.
- If σ_i is entailed by σ, the lifting statement is replaced with B_i.
- Otherwise, let $\sigma_1, \ldots, \sigma_k$ be all remaining local stores which are neither unsatisfiable nor entailed by σ, $k > 1$. Hence, add $\alpha(\sigma_1, \ldots, \sigma_k)$ to σ.
- Finally, the lifting statement suspends until more constraints are added to σ which may affect the execution of G_i, for some i, $1 \leq i \leq k$, and thereby the statement is reexecuted (incrementally, of course).

Using the lifting statement, disjunctions of finite domain constraints can be executed globally through a lifting function α for domain constraints (next section), and by encoding a disjunction $c_1 \vee \cdots \vee c_n$ as $c_1 \parallel$ **true**; $\cdots$; $c_n \parallel$ **true**.

5.2 Lifting domain constraints

In the following we assume there exists a lexicographic ordering of variables.

Given the constraint stores $\sigma_1, \ldots, \sigma_k$, $\alpha(\sigma_1, \ldots, \sigma_k)$ equals σ', where σ' is defined as:

1. Let σ' initially be empty.
2. Sort each σ_i by the ordering of variables.
3. For each x such that $x_{\sigma_i} \neq \mathcal{Z}$ for each i between 1 and k, generate $x \in x_{\sigma_1} \cup \cdots \cup x_{\sigma_k}$.
4. For each constraint $x \in I$ computed as in step 3, if $x_\sigma \subseteq I$, ignore the constraint. Otherwise, add $x \in I$ to σ'.

Since the local stores are kept sorted, generating $x \in x_{\sigma_1} \cup \cdots \cup x_{\sigma_k}$ incrementally for n variables in step 3 can be done in $O(k * n)$ time. Furthermore, in AKL, local stores are associated with a DIRTY-bit which is set initially and whenever a local store can no longer be guaranteed to be sorted (such as after a garbage collection, or after the addition of new constraints). Hence, before lifting is performed for FD constraints, each local store σ_i is checked if dirty, $1 \leq i \leq k$. If dirty, the store is sorted and the bit is reset. This improves the incremental behavior of lifting.

Example 5.1. Consider the disjunction $y = 1 \vee z = 1$ conjoined with $x = y \wedge x = z$ in the store $\sigma = \{x \in \{1,2\}, y \in \{1,2\}, z \in \{1,2\}\}$. By indexing $y = 1 \vee z = 1$ the indexicals y **in** $(\mathbf{dom}(z) \cap 1) \supset -\infty..\infty \cup 1$ and z **in** $(\mathbf{dom}(y) \cap 1) \supset -\infty..\infty \cup 1$ are generated which will not produce any further domain constraints in σ.

However, instead running $y = 1 \| \ \mathbf{true}$; $z = 1 \| \ \mathbf{true}$ will produce the stores $\sigma_y = \{x \in \{1\}, y \in \{1\}, z \in \{1\}\}$ and $\sigma_z = \{x \in \{1\}, y \in \{1\}, z \in \{1\}\}$. Hence, $x \in \{1\}$ is lifted and σ is updated to $\{x \in \{1\}, y \in \{1\}, z \in \{1\}\}$. □

6 Performance Evaluation

We now compare our two approaches for constructive disjunction with speculative and cardinality disjunction. As benchmarks we use two problems for scheduling and planning, the bridge-project problem [VH89] and the perfect squares problem [VHSD92], together with the n-queens problem.

The bridge and squares problems are concerned with shared resources, where the disjunctions are thus resource constraints. In the bridge example the disjunction

$$x_1 + s_1 \leq x_2 \vee x_2 + s_2 \leq x_1$$

is used, where x_1 and x_2 are domain variables, and s_1 and s_2 are constants.

In the perfect squares example two disjunctions are used:

$$x_1 + s_1 \leq x_2 \vee x_2 + s_2 \leq x_1 \vee y_1 + s_1 \leq y_2 \vee y_2 + s_2 \leq y_1$$

where x_1, x_2, y_1, and y_2 are domain variables, and s_1 and s_2 are constants, and the disjunction

$$(b = 1 \wedge x \in \{p - s + 1, \ldots, p\}) \vee (b = 0 \wedge x \in \{0, \ldots, p - s\} \cup \{p + 1, \ldots\})$$

where x is a domain variable, and p and s are constants.

For the n-queens problem we consider the effect of applying lookahead pruning [VH89] through disjunctive reasoning on the number of nondeterminate steps. Lookahead is applied by adding member$(i, [x_1, \ldots, x_n])$ for each i between 1 and n, where x_i represents queen i and member$(i, [x_1, \ldots, x_k])$ is interpreted as $i = x_1 \vee \cdots \vee i = x_k$.

We have run the programs in AGENTS, the implementation of AKL, currently under development at SICS.[1] The timings are in milliseconds computed on a SPARC-10 system. If no answer was computed within one minute,

[1] For more information on this system please contact `agents-request@sics.se`.

bridge	spec	card	local	global
Time (ms)	30290	406	80	3790
Non-det. steps	51	243	36	34

Table 3: BRIDGE-PROJECT

perfect squares	spec	card	local	global
Time (ms)	620	310	180	3390
Non-det. steps	25	20	8	8

Table 4: PERFECT SQUARES 8

or when the memory consumption became too large, "?" is used in the tables. We have used first-fail labeling throughout [VH89].

In tables 3, 4, and 5 we have included the runtime and number of nondeterminate steps for planning a bridge-project with about 30 jobs and 70 constraints, for packing a square with 8 squares, and packing a square with 17 squares, using speculative disjunction (spec), cardinality-based disjunction (card), and disjunctions executed locally (local) and globally (global).

As can be seen disjunction executed locally or globally prunes the number of nondeterminate steps more than do speculative and cardinality disjunction, however, the local scheme outperforms the global in runtime. This is because the global scheme does not produce sufficiently more pruning than the local, while being more expensive in time and space.

As we see it, the most problematic aspect of executing disjunctions globally is the reactivity of the disjunction. Each disjunct may affect, or be constrained by, many other variables. Hence, for any update of any one of those variables in the embedding store, the disjuncts must be reconsidered, and lifting retried. This should be controlled somehow, e.g. by only executing disjunctions globally at certain stages in the computation, and not necessarily at each propagation.

The reason why speculative disjunction needs fewer nondeterminate steps than cardinality in the bridge-example is that the solution happens to be found early in the speculative search. However, the execution of the program using speculative disjunction is heavily burdened by expensive deep guard propagations in AGENTS.

In Table 6 we give the timings for running the n-queens program with the extra member/2 disjunctions added and their four different interpretations, together with the version of n-queens with no extra disjunctions added (no).

As seen from the table, the member/2 disjunction executed globally prunes the number of nondeterminate steps dramatically, however, at a large performance cost. The speculative, local and cardinality disjunctions prunes the number of nondeterminate steps somewhat, however, with no obvious performance gain. We have also experimented with adding the redundant constraint $x_i = 1 \vee \cdots \vee x_i = n$, for each x_i, which for this example was less

perfect squares	spec	card	local	global
Time (ms)	?	?	1170	?
Non-det. steps	?	?	33	?

Table 5: Perfect squares 17

8-queens	no	spec	card	local	global
Time (ms)	45	200	130	400	330
Non-det. steps	25	24	22	22	4

Table 6: 8-queens with member constraint

efficient than the member/2 constraint.

7 Conclusion

We have presented two schemes for compiling and executing disjunctions of finite domain constraints such that various degrees of constructive disjunction is maintained.

The first scheme is solely based on conditional indexicals and implements constructive disjunction through local reasoning which ignores the effects of propagating each disjunct respectively.

The second scheme executes each disjunct in a private constraint store, propagating the consequences of the constraints in each disjunct respectively, and henceforth lifts constraints implied by the disjunction of the stores and adds them to the embedding store. This gives stronger pruning than the first scheme, however, a heavy implementation machinery is needed.

Furthermore, our initial performance evaluations indicate that the indexical scheme in fact implements sufficient pruning for many disjunctive constraints, such as resource and maximum/minimum constraints, extending the pruning given by cardinality-based disjunction. For our benchmarks the overhead of the lifting scheme for such constraints does not pay off, since it gives no more pruning than the indexical scheme.

On the other hand, the lifting scheme can be exploited for lookahead pruning, which decreases the number of nondeterminate steps dramatically for highly constrained problems such as graph-coloring and n-queens problems. This technique is not applicable using the indexical scheme.

Acknowledgments: This work has partly been financed by the ESPRIT-project ACCLAIM (# 7195). As always, running the article through the Torkel Franzén-filter has been rewarding. We also want to give a special thanks to the reviewers whose comments were most valuable.

References

[CJH94] B. Carlson, S. Janson, and S. Haridi. AKL(FD): a concurrent language for finite domain programming. In *Logic Programming: Proceedings of the 1994 International Symposium.* MIT Press, 1994.

[DC93a] D. Diaz and P. Codognet. Compiling constraints in clp(FD). Research report, INRIA, 1993.

[DC93b] D. Diaz and P. Codognet. A Minimal Extension of the WAM for clp(FD). In *Proceedings of the International Conference on Logic Programming.* MIT Press, 1993.

[DSH88] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the car sequencing problem in constraint logic programming. In *European Conference on Artificial Intelligence*, 1988.

[HSD92] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58:113–159, 1992.

[Jan94] Sverker Janson. *AKL—a multiparadigm programming language.* Uppsala theses in computing science 19, Uppsala University, June 1994.

[JH91] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Symposium.* MIT Press, 1991.

[JS93] J. Jourdan and T. Sola. The versatility of handling disjunctions as constraints. In *Proceedings of the Programming Language Implementation and Logic Programming Conference, LNCS 714.* Springer Verlag, 1993.

[Mac77] A. Mackworth. Consistency in Networks of Relations. *Journal of Artifical Intelligence*, 8:99–118, 1977.

[VH89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[VHD91] Pascal Van Hentenryck and Yves Deville. The cardinality operator: a new logical connective in constraint logic programming. In *International Conference on Logic Programming.* MIT Press, 1991.

[VHSD91] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint processing in cc(FD). Unpublished manuscript, 1991.

[VHSD92] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint Logic Programming over Finite Domains: the Design, Implementation, and Applications of cc(FD). Technical report, Computer Science Department, Brown University, 1992.