Constraint Processing in cc(FD)

Pascal Van Hentenryck[1], **Vijay Saraswat**[2], **Yves Deville**[3]

Abstract

Constraint logic programming languages such as CHIP [26,5] have demonstrated the practicality of declarative languages supporting consistency techniques and nondeterminism. Nevertheless they suffer from the *black-box effect*: the programmer must work with a monolithic, unmodifiable, inextensible constraint-solver.

This problem can be overcome within the logically and computationally richer concurrent constraint (cc) programming paradigm [17]. We show that some basic constraint-operations currently hardwired into constraint-solvers can be abstracted and made available as combinators in the programming language. This allows complex constraint-solvers to be decomposed into logically clean and efficiently implementable cc programs over a much simpler constraint system. In particular, we show that the CHIP constraint-solver can be simply programmed in cc(FD), a cc language with an extremely simple built-in constraint solver for finite domains.

1 Introduction

The purpose of our research is to design programming languages that support the solution of various combinatorial search problems arising in areas like combinatorial optimization, artificial intelligence, natural language processing, and hardware design. Our goal is to extract some of the fundamental techniques used in those areas and to support them inside programming languages in order to speed development time, allow rapid prototyping, and ease modifiability.

One aspect of our work has focused on the support of consistency techniques, a paradigm emerging from Artificial Intelligence, inside constraint logic programming. Consistency techniques have been used in a number of systems (e.g. [12,6,14,15] in order to reduce the search space by removing combinations of values that cannot appear together in a solution. Constraint Logic Programming (CLP) is a new class of declarative languages which generalizes Logic Programming by replacing unification with constraint solving [10]. Such languages are attractive for solving combinatorial search problems since their nondeterminism makes them adequate for stating various search procedures and their relational form makes it easy to state constraints. The support of consistency techniques in conjunction with nondeterminism is particularly appealing (as noted by Mackworth in [14]) as it frees the programmer from implementing both tree-search and constraint propagation.

The feasibility and practicality of a declarative language supporting both consistency techniques and nondeterminism was demonstrated in the CHIP system [26,5]. Solutions for a large variety of combinatorial problems — test-generation [22,29], car-sequencing [3], microcode labelling [4], scheduling and cutting stock [26] — were developed quickly, and were comparable in efficiency (within a constant factor) to specialized algorithms using the same approach. CHIP supported consistency techniques mainly by providing a number of primitive constraints, such as equations, inequalities, and disequations involving variables ranging over finite subsets of natural numbers.

Though successful, CHIP has a number of limitations, the most important being its lack of extensibility. The programmer has to recast non-primitive constraints in terms of existing constraints, is unable to define new primitive constraints and is unable to cope adequately with disjunctions of constraints or the definition of incomplete constraint solvers. As a result, non-primitive constraints are often recast in terms of more basic variables (e.g. Boolean variables) and/or are used to make choices. Both alternatives decrease pruning, increase the search space, and entail a substantial loss in efficiency compared to procedural languages. The

[1] Brown University, Box 1910, Providence, RI 02912, Email: pvh@cs.brown.edu
[2] Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, Email: saraswat@parc.xerox.com
[3] Université Catholique de Louvain, Pl. Ste Barbe, 2 B-1348 Louvain-La-Neuve, Belgium, Email: yde@info.ucl.ac.be

problem is not specific to CHIP but present in all CLP languages which are all based on a black-box constraint solver that the programmer cannot extend or modify.

The glass-box approach. We propose to remove the above limitation by taking a different approach. Our thesis is that consistency techniques can be supported in a much more fundamental way by providing general-purpose, efficiently implementable, declarative operations in the programming language that capture essential algorithmic techniques embedded in the implementation of such systems. To support this thesis, we introduce four new constraint language ideas: (1) *indexical constraints*, related to a restricted use of universal quantification, (2) *cardinality operators* with indexical bounds, related to the threshold operator of propositional logic [28], (3) *constructive disjunction*, related to intuitionistic disjunction, and (4) *extended Asks*, related to intuitionistic implication. We instantiate these ideas in the programming language `cc(FD)`, incorporating a very simple constraint system over finite systems, `FD`. Through these combinators, we show that `cc(FD)` supports the basic reasoning principles implemented in systems such as ALICE [12], REF-ARF [6], CHIP in a very simple and flexible way. Further, we briefly discuss a prototype implementation of `cc(FD)` that supports our contention that the language can be implemented extremely efficiently.

The main advantages of our approach are generality, extensibility, efficiency and semantic simplicity.

The constructs introduced are *general-purpose*: they are definable on very general notions of constraint systems [19], including rational arithmetic, Booleans and first-order terms.

They are powerful enough that the underlying constraint-solver can be extremely simple (e.g., it may provide only unary constraints). The additional functionality provided by more powerful constraint-solvers can be obtained by means of user-level programs. This significantly reduces the black-box effect. The programmer can define his own constraints (to the extent of the reasoning principles supported) without sacrificing the declarative reading of the programs.

Since the embedded constraint-solver is extremely simple, it can be very efficiently implemented. The combinators themselves are sufficiently "low-level" as not to introduce overhead (in time or space) compared to procedural implementation of the reasoning principles supported.

Finally, the abstract semantics of these combinators can be presented in a very simple way, within the concurrent constraint (`cc`) programming [17,18,19]. In this framework, computation progresses through the interaction of agents that communicate with each other via a shared store that is itself a constraint. The basic operations are *Tell* (add a constraint to the store) and *Ask* (check that a constraint is entailed by the store). [17,18,19] also discuss several other combinators such as parallel composition, hiding, and backtracking. Logically, the Ask operation corresponds to a primitive form of intuitionistic implication, parallel composition to conjunction, hiding to existential quantification and backtracking to disjunction in a very precise way [13]. Denotationally, agents can be thought of as *closure operators* over the underlying constraint system[4]. The four new combinators we introduce in this paper can readily be integrated into the framework as new operations on closure operators.

Rest of this paper. In summary, `cc(FD)` offers a new alternative to the traditional CLP model for constraint programming by systematically exploiting the `cc` framework. It proposes to design constraint languages by abstracting the computational principles behind a class of applications and by supporting them at the language level via constraint operations. The combinators are based on procedural interpretations of logical connectives; hence the approach combines the advantages of referential transparency, flexibility, and computational efficiency. This idea would bring similar advantages for other solvers (e.g. first-order terms, rationals) and application areas exploiting constraint technology (e.g. natural language processing, artificial intelligence). Although languages of this type would be more basic, it is natural to imagine that constraint solvers would be built in a modular, incremental, and hierarchical manner recovering the ease of programming of existing CLP languages while adding flexibility and efficiency.

The rest of this paper is devoted to an overview of the constraint processing facilities of `cc(FD)`. First we present the `FD` constraint system underlying `cc(FD)`— this is used in several examples throughout the paper.

[4] A closure operator over a lattice is a monotone, idempotent and extensive operator.

Next, we introduce the four combinators and discuss their operational and denotational semantics. Finally we consider the design of the `cc` language, which is an instantiation of the `cc` language with these combinators, over the `FD` constraint system. In particular, we sketch the implementation of the CHIP constraint-solver in `cc(FD)`.

To ease reading, only informal presentation are given in the text. The precise operational and denotational semantics of the new constructs will be given in the full version of the paper.

Note for the reviewer 1.1 This technical abstract assumes some familiarity with the notions of constraint systems, concurrent constraint programming languages and closure operators. For more information on these, the reviewer may consult [19]. Appendix A briefly summarizes the notions of closure operators. [15] is a good source for information on local consistency techniques (e.g., arc-consistency). □

2 The `FD` constraint system

We briefly present the very simple constraint system underlying `cc(FD)`. (In reality, the constraint system in `cc(FD)` also provides Herbrand trees, but for the purposes of this paper we concentrate on novel aspects.)

There are three kinds of syntactic objects, (t) arithmetic terms, (r) ranges, and (c) constraints. Their syntax is given by the grammar in Table 1. Ranges denote a finite union of intervals on the number-line — the given operations are interpreted in the obvious way. The entailment relation on constraints is induced in the obvious way from the interpretation: a valuation θ satisfies a constraint $X\ in\ r$ exactly when $\theta(X)$ lies in the range denoted by r. Thus the constraint

$$X\ in\, t_1\, ..\, t_2$$

is inconsistent if $t_2 < t_1$ — this is the only way inconsistency can arise in this system. Note that the constraint system is closed under negation: the constraint $\neg X\ in\, r$ is just $X\ in\ -r$.

Incremental algorithms for checking consistency of constraints, and for entailment are completely straightforward and very efficiently implementable.

The following proposition is easy to establish.

Proposition 2.1 *Any finite conjunction of constraints in this system in one variable can be represented by a single constraint of one of the following kinds:*

$$X \text{ in } (m_1\, ..\, n_1 : \ldots : m_k\, ..\, n_k)$$
$$X \text{ in } (m_1\, ..\, n_1 : \ldots : m_k\, ..)$$

where either $(k = 1, m_1 = 1, n_1 = 0)$ *or* $(k \geq 1, m_i \leq n_i < m_{i+1})$.

Such a constraint is said to be in normal form. A set of constraints $\{X_i\ in\ s_i \mid i \in I\}$, for some index-set I, is in normal form if all the variables are distinct and each constraint is in normal form.

The construction in the following theorem is the completion by ideals used in the definition of information systems [21].

Theorem 2.2 *For any finite set of variables V, let $\mathtt{FD}_V$ be the family of all entailment-closed sets of constraints generated from a set of constraints with free variables in V.*

1. *FD_V is a complete distributive lattice, with glbs given by intersection, and lubs by closure of the union.*
2. *The finite elements of $\mathtt{FD}_V$ are exactly those generated from a finite set of constraints.*[5]
3. *$\mathtt{FD}_V$ is not finitary: that is, there exists a finite constraint which has infinitely many finite elements below it.*

[5] u is a finite element of a lattice L iff for every directed subset S of L with lub above u, there is a finite subset of S with lub above u.

In the following, n ranges over natural numbers.

$$
\begin{array}{lll}
t ::= & X \mid n \mid t+t \mid t-t \mid t*t \mid t \;\mathtt{mod}\; t \mid t \div t & \\
r ::= & [t_1] \,..\, [t_2] & \textit{(from } t_1\textit{, default } 0\textit{, to } t_2\textit{, default infinity)} \\
 & \mid t & \textit{(shorthand for } t\,..\,t\textit{)} \\
 & \mid r_1 : r_2 & \textit{(union)} \\
 & \mid r_1 \,\&\, r_2 & \textit{(intersection)} \\
 & \mid -r & \textit{(complementation)} \\
 & \mid r+t & \textit{(pointwise addition)} \\
 & \mid r \;\mathtt{mod}\; t & \textit{(pointwise mod)} \\
c ::= & X \; in \; r &
\end{array}
$$

Each constraint is associated with an *implicit ask restriction* [17] — a constraint $X \; in \; r$ can be added to the store only when r is ground.

Some handy abbreviations:

$$
\begin{array}{ll}
X \; in \; \{n_1, \ldots, n_k\} & \triangleq X \; in \; n_1 : \ldots : n_k \\
X \leq t & \triangleq X \; in \; ..t \\
X < t & \triangleq X \; in \; ..(t-1) \\
X \geq t & \triangleq X \; in \; t\,.. \\
X > t & \triangleq X \; in \; (t+1)\,.. \\
X \neq t & \triangleq X \; in \; ..(t-1) : (t+1)\,.. \\
X = t & \triangleq X \; in \; t
\end{array}
$$

Table 1: The constraint system FD

4. The glb of two finite elements of $\mathtt{FD}_V$ *is itself finite.*

In particular, the glb of two normal-form sets of constraints $\{X_i \; in \; r_i \mid i \in I\}$ and $\{Y_j \; in \; s_j \mid j \in J\}$ is just the set of constraints generated by $Z \; in \; r_i : s_j$ where $X_i = Y_j$.

Note that the choice of the constraints represent one more step towards simplicity in the constraint-solver and hence reduces the black-box effect of usual CLP languages. In previous work [27], primitive constraints were limited to at most two variables. Such constraints can be solved completely with arc-consistency. In this paper, we can achieve the same effect while assuming that the implementation supports just unary constraints, using indexical constraints.

3 The Extended cc language framework

The basic Ask-and-Tell cc languages provide several combinators: Tell, Ask, Parallel composition, Hiding and Backtracking. These are provided in cc(FD) as well. The discussion of these combinators is omitted because they are standard — some information about Ask, Tell and Parallel composition can be obtained from the operational semantics in Table 3.

Indexical Constraints. Suppose we would like to write a program on top of the FD constraint system to implement reasoning with variation intervals for the constraint $X \geq Y + c$, for X, Y variables and c a constant. That is, we would like this program to have the following effect:

1. whenever it is true that $Y \geq e$ (for e a constant), the program should impose the constraint $X \geq e + c$.

2. whenever it is true that $e \geq X$, the program should impose the constraint $(e - c \geq Y)$.

In other words, we would like the program to behave as if it were the infinite conjunction of rules of the form

$$(Y \geq \alpha) \;\rightarrow\; (X \geq \alpha + c)$$

Basic Ask-and-Tell language.

$$
\begin{array}{rrll}
\textit{Programs} & P ::= & H :: A \mid P \wedge P \mid \forall X\, P & \\
\textit{Agents} & A ::= & B & \text{(Tell)} \\
 & & \mid B \rightarrow A & \text{(Ask)} \\
 & & \mid A, A & \text{(Conjunction)} \\
 & & \mid A; A & \text{(Disjunction)} \\
 & & \mid \exists X\, A & \text{(Local variables)} \\
 & & \mid H & \text{(Procedure Call)} \\
\textit{Basic constraints} & B ::= & c \mid B, B & \\
\textit{Procedure Call} & H ::= & p(X_1, \ldots, X_n) &
\end{array}
$$

The Extended language. The language allows Basic agents to be defined using cardinality, constructive disjunction, and indexical constraints. It allows users to specify new primitive constraints (the atomic formulas D), provided **tell**, **ask**, declarations are provided. It adds to the grammar for the basic Ask-and-Tell language the productions:

$$
\begin{array}{rrll}
\textit{Basic agents} & B ::= & \#(l, [B, \ldots, B]) \mid \#([c, \ldots, c], u) & \text{(Cardinality)} \\
 & & \mid\ \nabla\ (B, \ldots, B) & \text{(Constructive Disjunction)} \\
 & & \mid ic & \text{(Indexical constraints)} \\
 & & \mid D & \text{(User-defined constraints)} \\
\textit{Programs} & P ::= & \textbf{tell}\ D :: B \mid \textbf{ask}\ D :: B &
\end{array}
$$

Basic agents are required to be monotone in their indexical arguments in the Tell part, and anti-monotone in their Ask part. The definitions for user-defined constraints are required to be recursion-free.

Table 2: Abstract Syntax for `cc(FD)`

for all values of α.

For this purpose we introduce the notion of *indexical terms* and *indexical constraints*. An indexical term is syntactically an ordinary term (e.g., $max(X)$), but semantically it denotes a *mapping* from constraints to terms. As an example, for X a variable, we may define $max(X)$ to be the indexical term which in store s denotes n, where n is the least upper bound on X given s. Thus, in store $s = \{X\ in\ 1 : 3 : 4\ ..\ 20\}$, the value $max(X)_s$ of the indexical term $max(X)$ would be 20. The indexical terms $min(X)$ (returns the greatest lower bound on X) and $dom(X)$ (returns the domain of X) can similarly be defined in `FD`.

We allow indexical terms to be used within constraints just as ordinary terms. A constraint containing an indexical sub-term is called an *indexical constraint*. It denotes a mapping from constraints to constraints obtained in the obvious way: given an indexical constraint d and input constraint s, the mapping returns the constraint obtained from d by replacing each indexical sub-term t by t_s. For example, in the store $s = \{X\ in\ 3\ ..,\ Z\ in\ 2\}$, the indexical constraint $(Y\ in\ min(X)\ ..)$ denotes the constraint $(Y\ in\ 3\ ..)$.

Example 3.1 (greatereqc/3) The constraint `greatereqc/3` may now be programmed as follows:

`greatereqc(X, Y, C)` $::$ `(X` in `(`min`(Y) + C)..), (Y` in `..(`max`(X)` $-$ `C)).`

In any store s, both constraints may fire, producing information about `X` and `Y` respectively, exactly as desired. □

Logically, a constraint $(X\ \ in\ \ (min(Y) + c)\ \ ..)$ can be thought of as the constraint: $\forall k.(Y \geq k) \Rightarrow X \geq k + c$. However, only very restricted kinds of universal quantifications can be obtained by using $min/1$, $max/1$, $dom/1$ indexical terms.

Example 3.2 (mod/4) As another example, consider the constraint `mod(X,Y,C,Base)` which holds iff `X = (Y + C) mod Base` assuming that `X,Y` are variables ranging over `{0..Base-1}` and `C` is a integer in `{0..Base-1}` and `Base` is a positive number. We can get this effect with:

Configurations are pairs of the form $\langle \Gamma, s\rangle$ where s is a set of constraints, and Γ is a multiset of agents. (The empty multiset is denoted ().) Below, we define a binary relation $\longrightarrow$, the *transition* relation which specifies how configurations evolve. Note that only *fair* execution sequences (e.s.) are considered legal. (A fair e.s. is not unfair; an *unfair* e.s. is one in which some given transition is enabled at every configuration in the sequence, and never taken.)

Basic Ask-and-Tell language. We omit the rules for existential quantification, backtracking and procedure calls, since they are standard. (See for example, [11].) Below, $\vdash$ is the entailment relation of the underlying constraint system.

$$\begin{array}{rll} \text{(Tell)} & \langle(\Gamma, c), s\rangle \longrightarrow \langle \Gamma, s \cup \{c\}\rangle & \\ \text{(Ask)} & \langle(\Gamma, d \rightarrow A), s\rangle \longrightarrow \langle(\Gamma, A), s\rangle & s \vdash d \\ \text{(Parallel Composition)} & \langle(\Gamma, (A_1, A_2)), s\rangle \longrightarrow \langle(\Gamma, A_1, A_2), s\rangle & \end{array}$$

Indexical Constraints.

$$\langle(\Gamma, ic), s\rangle \longrightarrow \langle(\Gamma, ic), s \cup \{ic_s\}\rangle$$

Cardinality. To simplify presentation we assume that the collection of agents of a cardinality constraint is a multiset of the form $[(B_1, d_1), \ldots, (B_n, d_n)]$, where the d_i are (possibly empty) sets of constraints. (The d_i denote local stores for each basic agent.)

$$\frac{\langle B, d \cup s\rangle \longrightarrow \langle B', d'\rangle}{\langle(\Gamma, \#(l, [(B, d) \mid \sigma])), s\rangle \longrightarrow \langle(\Gamma, \#(l, [(B', d') \mid \sigma])), s\rangle}$$

$$\begin{array}{ll} \langle(\Gamma, \#(l, [(B, \texttt{false}) \mid \sigma])), s\rangle \longrightarrow \langle(\Gamma, \#(l, \sigma)), s\rangle & \\ \langle(\Gamma, \#(l, [((), d) \mid \sigma])), s\rangle \longrightarrow \langle(\Gamma, \#(l-1, \sigma)), s\rangle & \text{if } s \vdash d \\ \langle(\Gamma, \#(l, [(B_1, d_1), \ldots, (B_l, d_l)])), s\rangle \longrightarrow \langle(\Gamma, B_1, \ldots, B_l), s \cup d_1 \cup \ldots \cup d_l\rangle & \\ \langle(\Gamma, \#(l, \sigma)), s\rangle \longrightarrow \langle(), \texttt{false}\rangle & \text{if } |\sigma| < l \end{array}$$

The rules for upper-cardinality agents are similar, and omitted.

Constructive Disjunction. To simplify presentation we assume that the argument of a disjunctive agent is an unordered collection of agents of the form (B, d), where d is a possibly empty set of constraints.

$$\frac{\langle B, d \cup s\rangle \longrightarrow \langle B', d'\rangle}{\langle(\Gamma, \nabla\ ((B, d), E)), s\rangle \longrightarrow \langle(\Gamma, \nabla\ ((B', d'), E)), s\rangle}$$

$$\langle(\Gamma, \nabla\ ((B_1, d_1), \ldots, (B_n, d_n))), s\rangle \longrightarrow \langle(\Gamma, \nabla\ ((B_1, d_1), \ldots, (B_n, d_n))), s \cup glb(d_1, \ldots, d_n)\rangle$$

Extended Asks. To simplify presentation, we assume that the ask part is an agent of the form (B, d), where d is a possibly empty set of constraints.

$$\frac{\langle B, d \cup s\rangle \longrightarrow \langle B', d'\rangle}{\langle(\Gamma, (B, d) \rightarrow A), s\rangle \longrightarrow \langle(\Gamma, (B', d') \rightarrow A), s\rangle}$$

$$\langle(\Gamma, ((), d) \rightarrow A), s\rangle \longrightarrow \langle(\Gamma, d \rightarrow A), s\rangle$$

Table 3: Transition system for Extended cc language

$$\begin{array}{rl} it ::= & X \mid n \mid it + it \mid it - it \mid it \times it \mid it \ \texttt{mod}\ t \mid it \div it \mid min(X) \mid max(X) \\ ir ::= & [it_1]..[it_2] \mid it \mid ir_1 : ir_2 \mid ir_1 \& ir_2 \mid -ir \mid ir + t \mid ir \ \texttt{mod}\ t \mid dom(X) \\ ic ::= & X\ in\ ir \end{array}$$

The indexical terms $min(X)$, $max(X)$, and $dom(X)$ are considered ground, for purposes of the implicit Ask restriction. Also, constraints satisfy the condition that terms $min(X)$ must occur "negatively" in the range, and terms $max(X)$ must occur "positively" in the range.

Table 4: Indexical constraints in cc(FD)

mod(X, Y, C, Base) :: X in (dom(Y) + C) mod Base, Y in (dom(X) − C) mod Base.

The first expression makes sure that, whenever a value is removed from the domain of Y, the corresponding value is removed from the domain of X. Similarly for the second. □

For indexical constraints to generate closure operators, (and hence to be well-defined) they must denote *monotone* functions: that is, as the information content of the store increases, the strength of the constraint yielded by the indexical constraint should increase.[6] This is true, for example, for $(Y\ in\ min(X)\ ..)$. It is not true for $(Y\ in\ ..\ min(X))$. Therefore, additional syntactic restrictions have to be placed on occurrences of indexical terms to ensure that the overall constraints built using them are monotone. These conditions are easy to state: intuitively all (positive) occurrences of terms $max(X)$ should be in the "upper" portions of range-restrictions, and all (positive) occurrences of terms $min(X)$ should be in "lower" portions of range-restrictions.

Use. Indexical arithmetic constraints provide more effective algorithms for arc-consistency. The new algorithms are $O(ed)$ (where e is the number of constraints and d the size of the domains), improving on the traditional quadratic algorithms. More important, the space requirement is $O(e)$ — each constraint takes constant space. An example of an application which can benefit from this extension is the microcode labeling problem [26].

Implementation. It should be clear that for the FD constraint system, the implementation of indexical terms and constraints is totally straightforward. The implementation already maintains max, min and dom information for each variable: we are now simply providing a mechanism for the user to access this information in a "safe" way.

Cardinality. The *cardinality* combinator [28] is a generalization of the threshold operators for propositional calculus and provides a general form of disjunction. Related to the Andorra model of computation [30], it allows arc-consistency of any finite domain constraint to be implemented within the complexity bounds of the optimal algorithm of [16]. Moreover, it provides a general way of implementing the *DMA/DMT* options as well as the *conditional summation* of **ALICE** [12].

There are two cardinality combinators, the *lower* and the *upper*, of the form $\#(it, [B_1, \ldots, B_n])$ and $\#([c_1, \ldots c_n], it)$ respectively, where it is an indexical term. Intuitively, the constraint $\#(l, [B_1, \ldots, B_n])$ holds in store s iff at least l_s of the given agents are true, and the constraint $\#([c_1, \ldots, c_n], u)$ holds iff at most u_s of the given constraints are true. Note that for this to be well-defined it must be the case that l_s increases as s becomes stronger (e.g. $l = min(X)$), and u_s decreases as s becomes stronger (e.g. $u = max(X)$). In the following, we will consider the expression $c \Rightarrow d$ as shorthand for $\#(1, [\neg c, d])$, and the expression $c \iff d$ as shorthand for $(c \Rightarrow d), (d \Rightarrow c)$.

The agent $\#(l, [c_1, \ldots, c_n])$ is executed by spawning n Ask agents, one for each argument and maintaining a counter each for the number of entailed constraints (a), and the number of constraints whose negation is not yet entailed (t). If in any store $a \geq l$, the agent may terminate without adding any new information to the store. On the other hand, if $l = t$, then the agent may terminate, adding those constraints $c \in [c_1, \ldots, c_n]$ such that neither c nor its negation is entailed by the store. If $l > t$, the agent may terminate, causing the store to become inconsistent. The evaluation rules for $\#([c_1, \ldots, c_n], u)$ are similar — note just that if $u = a$, the agent may terminate, adding the *negations* of all the constraints in $[c_1, \ldots, c_n]$ not entailed by the store. The set of equational laws underlying this procedural description are given in Table 5.

[6] Every monotone function f generates a closure operator $\hat{f} = \lambda s.\mathbf{fix}(\lambda a.s \wedge (fa)))$ by augmentation and iteration to quiescence.

Use.

Example 3.3 (Arc-consistency.) We show how to enforce directional arc-consistency on a binary constraint p(X,Y). First, the domains of X and Y must be reduced respectively to the projections of the constraint on the first and second arguments, say D_1, D_2. Next, for each value v in D_1, we generate the constraint

$$Y \neq w_1, \ldots, Y \neq w_p \quad \Rightarrow \quad X \neq v$$

where $w_1, \ldots, w_p$ are all values w such that p(v,w). Thus, as soon as Y is constrained to be different from all values in $\{w_1, \ldots, w_p\}$, X is constrained to be different from v. Conversely, if X is equated to v, then Y will be required to take on one of the given values. □

[28] showed that, on simple examples, cardinality can bring an exponential improvement over traditional approaches. Most CHIP applications can be solved more naturally using cardinality removing the need for ad-hoc built-in constraints. For these applications, the overhead may be as low as 6% (e.g. car-sequencing [3]) while not exceeding 30%. A new interesting application is the allocation of jobs to pipeline processors to minimize total delay. A cc(FD) program was developed for the task and is comparable (sometimes faster) in efficiency to a specific branch and bound algorithm. Cardinality was used to implement conditional summation (e.g. the summation of the computation times of all jobs assigned to the same processor must not exceed it capacity in a cycle time) as well as the constraint for the transmission time which is: if either job is run on the master processor (processor 1), then transmission time is 0; and if the two jobs are run on adjacent machines (neither of which is the master processor) then transmission time is 1; otherwise transmission time is 2. This can be expressed directly in cc(FD):

$$\#(1, [P_1 \neq 1, P_2 \neq 1, T_{12} = 0])$$
$$\#(1, [P_1 = 1, P_2 = 1, \#([X \; in \; dom(Y) + 1, Y \; in \; dom(X) + 1], 0), T_{12} = 1])$$
$$\#(1, [P_1 = 1, P_2 = 1, \#([\min(P_1) > max(P_2) + 1, min(P_2) > max(P_2) + 1], 0), T_{12} = 2])$$

where P_1, P_2 are the processors associated with jobs 1 and 2 and T_{12} is the transmission time between them.

Constructive Disjunction. The basic idea behind constructive disjunction is to add to the store constraints entailed by all possible alternatives, i.e. the conjunctions made up from the store and one of the disjunctions. By factoring common information from the disjuncts constructive disjunction produces more pruning than cardinality, albeit at a greater cost. Constructive disjunction can be used for many constraints involving (explicitly or implicitly) disjunctions.

Example 3.4 (max/3) A typical example is the max/3 constraint, especially useful in scheduling problems [26]. The constraint max(X,Y,Z) holds iff Z is the maximum of X and Y. It can be expressed, using cardinality, in the following way:

$$\max(X, Y, Z) :: Z \; in \; min(X) .. \; \& \; min(Y) .., \; \#(1, [Z \; in \; dom(X), Z \; in \; dom(Y)]).$$

The first two constraints ensure that Z is always not smaller than X and Y, while the last constraint ensures that Z is equal to one of them. The constraint however does not achieve as much pruning as we would like. For instance, if X,Y,Z range over {5..10}, {7..11}, and {1..20} respectively, the above program would reduce the domain of Z to to {7..20} instead of the expected {7..11}, because cardinality handles the two equations locally and concludes that neither is entailed.

□

Constructive disjunction adds to the store the constraints common to the disjuncts, e.g. `Z in {5..11}` in the above example. Syntactically, a constructive disjunction is of the form $\nabla\ (B_1, \ldots, B_n)$ and can be read declaratively as a disjunction. Operationally it performs *constraint generalization*, i.e. it computes the greatest lower bound (glb) in the constraint lattice of the constraints $B_1(s), \ldots, B_n(s)$ (where we are thinking of each B_i as a closure operator). If a glb algorithm is not available for the constraint system, any approximation (i.e. any set of constraints which is implied by all constraints in the *glb*) would do as well.

Example 3.5 (max/3 contd.) The `max/3` constraint can be expressed as

$$\texttt{max(X, Y, Z)} :: \texttt{Z}\ \mathit{in}\ \texttt{min(X)}..\ \&\ \mathit{min}(\texttt{Y})..,\ \ \nabla\ (\texttt{Z}\ \mathit{in}\ \mathit{dom}(\texttt{X}),\ \texttt{Z}\ \mathit{in}\ \mathit{dom}(\texttt{Y})).$$

This program achieves the expected pruning since information from both equations can be combined during propagation. □

Use. Constructive disjunction is mainly motivated by disjunctive scheduling applications although it may be used in many other applications (e.g. biology). In the general problem, tasks have to be scheduled on different machines in presence of precedence and other constraints. For simplicity, we assume that the machine on which a task has to be executed has already been determined. At this point, the key problem is to find an ordering of the tasks on a machine so as to minimize an objective function (e.g. the makespan or the total delay). To obtain reasonable efficiency, it becomes necessary to exploit the disjunctions to prune the search space. Consider n tasks in such a disjunction and assume that their starting dates, end dates and durations are denoted by S_1, E_i, D_i respectively. A typical pruning rule [2] is as follows:

> If the starting date of task i plus the summation of all durations is greater than the maximum of all end dates but the one of task i, then task i cannot be scheduled first.

This rule can be implemented using implication and constructive disjunction as

$$\begin{array}{l}
\texttt{Max}\ \mathit{in}\ \mathit{min}(\texttt{E}_\texttt{1})..\ \&\ \ldots\ \&\ \mathit{min}(\texttt{E}_\texttt{n}),\\
\quad \nabla\ (\texttt{Max}\ \mathit{in}\ \mathit{dom}(\texttt{E}_\texttt{1}),\ \ldots,\ \texttt{Max}\ \mathit{in}\ \mathit{dom}(\texttt{E}_\texttt{n})),\\
\quad \texttt{Max}\ \mathit{in}\ ..(\texttt{min}(\texttt{S}_\texttt{i}) + \texttt{D}_\texttt{1} + \ldots + \texttt{D}_\texttt{n} - \texttt{1}) \ \rightarrow\ \texttt{P}_\texttt{i}\ \mathit{in}\ \texttt{0 : 2}..).
\end{array}$$

where $\texttt{P}_1$ represents the position of task i in the disjunction.

The branch & bound algorithm of [2] was implemented using the above constraint and additional ones in the same spirit. The resulting program is short, yet it provides reasonable efficiency (between 5-10 time slower). As constraint programming languages are still rather recent and `cc(FD)` is a prototype, the result is encouraging.

Implementation. In our current implementation constraint generalization is done pairwise and changes are propagated only when the result has been modified and a subsequent operation may change the result. Successive glbs are maintained in a persistent data structure (somewhat similar to a RETE network [7]) to avoid restarting the whole computation from scratch when the store is updated. Similar implementations are possible for other domains (e.g. first-order terms).

Extended Asks. To obtain a flexible, extensible, and efficient system, it is desirable to ask non-primitive constraints as well as primitive constraints. In general, a non-primitive constraint will be a closure operator. Asking whether a primitive constraint c holds is just checking to see if the store has enough information to entail c. Semantically, the closure operator corresponding to $c \ \rightarrow\ f$ (for c a constraint, and f a closure operator) is just the operator whose fixed-point set is $\{s \mid s \not\vdash c\} \cup f$ [18] (see the Appendix for the connection between closure operators and their fixed-point sets). What should it mean to Ask a closure operator?

One natural answer is as follows: say that a store s solves a closure operator f if f *terminates* when run in s, without producing any more information. In such a case, f cannot produce any more information in s, or *in any strengthening of* s. More abstractly, we can say that s solves f (and write $s \vdash f$) if s is a stable fixed-point of f, that is, every $t \geq s$ is a fixed-point of f (notationally: $\uparrow s \subseteq f$). Now it becomes easy to define an *extended* ask operator: given a closure operators f and g, $f \;\rightarrow\; g$ is the closure operator whose fixed-point set is given by

$$\{s \mid s \not\vdash f\} \cup g$$

It is easy to see that this set is closed under glbs and hence is the fixed-point set of a closure operator.

The operational semantics of Extended Asks is given in more detail in Table 3. The correspondence between the operational and denotational semantics is not difficult to establish. We note that this operation satisfies a number of nice properties, summarized in Table 5. Note, however, that the operation is anti-monotone in its first argument — hence we cannot allow the class of agents that appear in this position (namely, Basic Agents) to be closed under recursion.

We also note in passing that (monotone) indexical constraints can be answered quite efficiently in `cc(FD)`. We illustrate with an example. A constraint s solves the indexical constraint X *in* $..max(Y)$ iff $s \vdash (X \textit{ in } ..\, min(Y)_s$. Thus, all that needs to be done is to evaluate a constraint in the current store — if the check succeeds, then the Ask succeeds, no more work needs to be done.

Non-primitive Constraints. `cc(FD)` also provides a simple facility for defining Basic Agents. For each new non-primitive constraint, two actions need to be defined, **tell** and **ask**, via declarations of the form:

tell `D` :: $\texttt{B}_{\texttt{t}}$.
ask `D` :: $\texttt{B}_{\texttt{a}}$.

The first is used in situations where the user-defined constraint needs to imposed, and the second in situations where it is to be asked. If the user supplies only the **tell** action $\texttt{B}_{\texttt{t}}$ for a user-defined constraint `D`, the system will take the **ask** action to be $\texttt{B}_{\texttt{t}}$ as well. However, the **tell** action is usually an *incomplete approximation* to the intended interpretation of `D`, and it may be possible for the user to supply a better approximation for Ask. For this reason, we allow the user to specify the two actions separately. It is the user's responsibility to ensure that these two definitions are logically related to each other in the expected way.

Example 3.6 (`greatereqc(X,Y,C)`) This example illustrates the advantage of allowing the user to define the four basic actions differently. A typical definition for this constraint in `cc(FD)` would be:

tell `greatereqc(X, Y, C)` :: `X` *in* (*min*(`Y`) + `C` ..), `Y` *in* (.. *max*(`X`) − `C`).
ask `greatereqc(X, Y, C)` :: `X` *in* (*max*(`Y`) + `C` ..).

□

Denotational semantics. Above, we have discussed the denotational semantics of extended asks explicitly. The semantics of the other constructs (indexical constraints, cardinality and constructive disjunction) is straightforward: all of them can be seen as defining closure operators in the obvious way. Indeed the constructive disjunction operation is just the glb operation in the lattice of closure operators over the constraint system. (This is how they were first discovered; only later was their computational significance understood.)

Cardinality. In the following, σ' is a permutation of σ.

$$\#(l,\sigma,u) = \#(l,\sigma),\#(\sigma,u)$$

$\#(l,\sigma) = \#(l,\sigma')$		$\#(\sigma,u) = \#(\sigma',u)$	
$c,\#(l,[d \mid \sigma]) = \#(l,[e \mid \sigma])$		$c,\#([d \mid \sigma],u) = \#([e \mid \sigma],u)$	$(c,d) \iff (c,e)$
$\#(l,\sigma) = false$	$(\lvert\sigma\rvert < l)$	$\#(\sigma,u) = false$	$(u < 0)$
$\#(l,\sigma) = true$	$(l \leq 0)$	$\#(\sigma,u) = true$	$(\lvert\sigma\rvert \leq u)$
$\#(l,[false \mid \sigma]) = \#(l,\sigma)$		$\#([false \mid \sigma],u) = \#(\sigma,u)$	
$\#(l,[true \mid \sigma]) = \#(l-1,\sigma)$		$\#([true \mid \sigma],u) = \#(\sigma,u-1)$	
$\#(l,[c_1,\ldots,c_l]) = (c_1,\ldots,c_l)$		$\#([c_1,\ldots,c_n],0) = (\neg c_1),\ldots,(\neg c_n)$	

Constructive Disjunction.

$c_1 \nabla c_2 = c_1$	(if $c_2 \rightarrow c_1$)
$c,(d \nabla B) = c,(e \nabla B)$	(if $c \rightarrow (d \leftrightarrow e)$)
$c \nabla d = glb(c,d),(c \nabla d)$	
$B \nabla \mathtt{false} = B$	
$B_1 \nabla B_2 = B_2 \nabla B_1$	
$B_1 \nabla (B_2 \nabla B_3) = (B_1 \nabla B_2) \nabla B_3$	

Extended Asks.

Definition 3.1 $s \vdash B \stackrel{\triangle}{=} \uparrow s \subseteq B$ □

Proposition 3.1 $s \vdash (B_1, B_2)$ *iff* $s \vdash B_1$ *and* $s \vdash B_2$.

Definition 3.2 $B_1 \rightarrow B_2 \stackrel{\triangle}{=} \{s \mid s \not\vdash B_1\} \cup B_2$ □

Proposition 3.2 $s \vdash B_1 \rightarrow B_2$ *if* $B_1(s) \vdash B_2$.

The converse may not hold; there are simple counter-examples.
The following laws are satisfied by Extended Asks: (I is the identity closure operator)

$$A \rightarrow A = I$$
$$I \rightarrow A = A$$
$$B_1 \rightarrow B_2 = (B_1 \rightarrow (B_1, B_2)$$
$$B_1 \rightarrow (B_2 \rightarrow B_3) = (B_1, B_2) \rightarrow B_3$$
$$B_1 \rightarrow (B_2, B_3) = (B_1 \rightarrow B_2), (B_1 \rightarrow B_3)$$

Table 5: Simplification rules for basic combinators.

4 The language `cc(FD)`

`cc(FD)` is simply an instantiation of the above language framework, over the `FD` constraint system. In many ways it can be thought of as a "rational reconstruction" of CHIP. Below we discuss how the constraint-solver of CHIP can be programmed in `cc(FD)` in very straightforward ways.

Arithmetic Constraints. CHIP supports the use of numerical constraints over finite domains using reasoning over variation intervals. These constraints are constructed from natural numbers, domain variables, the operators $+$, $*$, $-$, and the relations $\leq$, $\geq$, $<$, $>$, $=$. It is easy to see that any such constraint can be expressed in terms of constraints having more than three variables. Hence it is sufficient to show how to simulate three-variable CHIP constraints in `cc(FD)`. For this we use indexical constraints. For instance, the constraint $\mathtt{X} \times \mathtt{Y} \geq \mathtt{Z}$ is modelled as:

```
gtemult(X, Y, Z) ::
    Z in (.. max(X) × max(Y)),
    X in (1 ..) → Y in (min(Z)/max(X)..),
    Y in (1 ..) → X in (min(Z)/max(Y)..).
```

Similarly for other constraints.

Disequations are only handled in `cc(FD)` exactly in the same way as in CHIP: suspend until exactly one variable is left in the constraint. *Minimum* and *Maximum* constraints can be defined directly as discussed earlier.

Symbolic Constraints. A number of symbolic constraints were available in CHIP and were instrumental in solving many practical problems. For example, CHIP provides the constraint `element(I,L,V)` which holds iff element `I` of list `L` is equal to `V`. The constraint is used with `I` and `V` being domain variables and `L` being a list of integers. In CHIP, the user may specify that the system reason with this constraint using either arc-consistency or reasoning about variation intervals. Both these reasoning techniques can be expressed as user-programs in `cc(FD)` as follows. To enforce arc-consistency on `element(I,L,V)`, generate the constraint $\mathtt{V} = \mathtt{e} \Leftrightarrow \mathtt{I\ in\ i_1} : \ldots : \mathtt{i_p}$ for all values `e` with occurrences $\mathtt{i_1}, \ldots, \mathtt{i_p}$ in `L`. To enforce reasoning with variation intervals, generate the constraint $V \geq e \Leftrightarrow I \notin \{i_1, \ldots, i_p\}$ for each value `e` in `L`, where $\mathtt{i_1}, \ldots, \mathtt{i_p}$ are all positions in `L` with values smaller than `e`. If desired, similar constraints can be generated for $\mathtt{V} \leq \mathtt{e}$ — and indeed that is the basic point of `cc(FD)`: the programmer can put in his own heuristics in a declarative way.

The most difficult constraint is `circuit` which enforces a circuit among a set of `n` variables wrt their indices. The easiest way to implement the constraint is to spawn a recursive predicate for each variable. Those constraints make sure that all variables at a distance $<$ `n-2` on the path starting from a variable are different from the index of that variable. The recursive predicate makes use of Ask to control the termination and find the successor (instead of guessing it). It is also possible to express the `circuit` constraint using only cardinality and universal quantification by creating variables for the successors of a variable.

Demons and Declarations. CHIP also has demon declarations and an `if_then_else` construct, which are easily expressed using Asks. It also has two declaration mechanisms to define non-primitive constraints using forward checking and lookahead. These two mechanisms are obsolete since arc-consistency can be enforced in an optimal manner using cardinality. In addition, Ask can delay a constraint until it is sufficiently instantiated thus obtaining the effect of forward checking.

Discussion. `cc(FD)` generalizes the finite domain part of CHIP in many ways. First, it provides, at the language level, all the principles that were previously buried inside the implementation. Typical examples are arithmetic reasoning and consistency techniques. Second, it proposes general ways of dealing with some issues such as disjunctions. Cardinality and constructive disjunction are particularly helpful in that respect. Third, it allows more efficient implementation techniques by taking advantage of the property of the constraints. This is one of the basic features of universal and existential quantification. Last, it allows non-primitive constraints to be handled in exactly the same way as primitive constraints.

5 Conclusion

This paper aimed at presenting an overview of the constraint processing facilities of `cc(FD)`. `cc(FD)` is best viewed as a systematic reconstruction and extension of the finite domain part of CHIP and supports both consistency techniques and nondeterminism in a declarative language. The main novelty in `cc(FD)` is to support consistency techniques at the language level through a number of combinators and constraint operations. This is to be contrasted with existing constraint logic programming languages where all constraint techniques are buried inside the constraint solver with the consequence that the user has no way to extend and modify the constraint solver and to define his own non-primitive constraints.

The main advantages which result from the design of `cc(FD)` are (1) the additional flexibility and extensibility which enables the user to define his own constraints (2) the clean semantic foundation in terms of an extremely simple constraint solver.

Another contribution of this research is the identification of a number of general-purpose constraint operations and combinators which will be valuable for other constraint systems as well.

Current research on this topic is devoted to a robust implementation of the language as well as the study of the combinators for other constraint systems, e.g. linear rational arithmetics.

Acknowledgements. We thank John Lamping, Johan deKleer, Danny Bobrow and Francesca Rossi for useful discussions.

References

[1] A. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transaction on Programming Languages and Systems*, 3(4):353–387, 1981.

[2] J. Carlier. Ordonnacement a contraintes disjunctives. *RAIRO* , 12:333–351, 1978.

[3] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the Car Sequencing Problem in Constraint Logic Programming. In *European Conference on Artificial Intelligence (ECAI-88)*, Munich, W. Germany, August 1988.

[4] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1-2):75–93, 1990.

[5] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988.

[6] R.E. Fikes. *A Heuristic Program for Solving Problems Stated as Non-deterministic Procedures.* PhD thesis, Comput. Sci. Dept., Carnegie-Mellon Univ. Pittsburgh, PA, 1968.

[7] C.L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[8] G.Gierz, K.H.Hoffman, K.Keimel, J.D.Lawson, M.Mislove, and D.S.Scott, editors. *A compendium of continuous lattices.* Springer-Verlag Berlin Heidelberg New York, 1980.

[9] T. Graf, P. Van Hentenryck, C. Pradelles, and L. Zimmer. Simulation of Hybrid Circuits in Constraint Logic Programming. *Computers and Mathematics with Applications*, 20(9/10):45–56, 1990.

[10] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *POPL-87*, Munich, FRG, January 1987.

[11] R. Jagadeesan, V. Shanbhogue, and V. Saraswat. Angelic non-determinism in concurrent constraint programming. Technical report, System Sciences Laboratory, Xerox PARC, January 1991.

[12] J-L. Lauriere. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10(1):29–127, 1978.

[13] Patrick Lincoln and Vijay Saraswat. Proofs as Concurrent Processes: A Logical Interpretation for Concurrent Constraint Programming. Technical report, Systems Sciences Laboratory, Xerox PARC, November 1991.

[14] Alan. K. Mackworth. Consistency in Networks of Relations. *AI Journal*, 8(1):99–118, 1977.

[15] Alan. K. Mackworth. Constraint Satisfaction. *Handbook of AI*, 205–211, 1988.

[16] R. Mohr and T.C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[17] V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, 1989.

[18] V.A. Saraswat and M. Rinard. Concurrent Constraint Programming. In *Proceedings of Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[19] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of Ninth ACM Symposium on Principles of Programming Languages*, Orlando, FL, January 1991.

[20] Dana S. Scott. Data types as lattices. *SIAM*, 5(3):522–587, 1976.

[21] Dana S. Scott. Domains for denotational semantics. In *Proceedings of ICALP*, 1982.

[22] H. Simonis. Test Generation using the Constraint Logic Programming Language CHIP. In *Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989.

[23] H. Simonis and M. Dincbas. Using an Extended Prolog for Digital Circuit Design. In *IEEE International Workshop on AI Applications to CAD Systems for Electronics*, pages 165–188, Munich, RFA, October 1987.

[24] H. Simonis and M. Dincbas. Using Logic Programming for Fault Diagnosis in Digital Circuits. In *German Workshop on Artificial Intelligence (GWAI-87)*, pages 139–148, Geseke, RFA, Septembre 1987.

[25] A. Tarski. *Logics, semantics and meta-mathematics*. Oxford University Press, 1956. Translated by J.H. Woodger.

[26] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[27] P. Van Hentenryck and Y. Deville. Operational Semantics of Constraint Logic Programming over Finite Domains. In *Third International Symposium on Programming Language Implementation and Logic Programming (PLILP-91)*, Passau (Germany), August 1991.

[28] P. Van Hentenryck and Y. Deville. The Cardinality Operator: A new Logical Connective and its Application to Constraint Logic Programming. In *Eighth International Conference on Logic Programming (ICLP-91)*, Paris (France), June 1991.

[29] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint Satisfaction Using Constraint Logic Programming. Technical Report (Forthcoming), Brown University, November 1991.

[30] D.H.D. Warren. The Andorra Principle. Presented at the 1987 GigaLips workshop, U. of Bristol.

A Closure operators

An operator over a partial order that is extensive, idempotent and monotone is called a *closure operator* (or, more classicaly, a *consequence* operator, [25]). Closure operators are extensively studied in [8]; continuous closure operators have been used in [20] to characterize data-types.

We list here some basic properties. Every closure operator can be uniquely represented by its range (which is the same as its set of fixed points); the function can be recovered by mapping each input to the least element in the range above it. (In the following, we shall often confuse a closure operator with its range, writing $x \in f$ to mean $f(x) = x$.) In fact, a subset of $|D|$ is the range of a closure operator iff it is closed under glbs of arbitrary subsets. (Thus, the range of every closure operator is non-empty, since it must contain $\top_{|D|} = glb\emptyset$.)

Also, it can be shown that a closure operator is continuous iff its range is closed under the lubs of directed subsets. The extensional partial order on closure operators ($f \leq g$ iff for all x, $f(x) \leq g(x)$) translates into reverse-inclusion on the set of fixed points: $f \leq g$ iff $f \supseteq g$. The identity operator I (with range $|D|$) is the bottom element; the operator which maps every element to $\top$ (and hence has range $\{\top\}$) is the top element. In fact it is easy to see that the set of closure operators over $|D|$ ordered by $\supseteq$ is a complete lattice, with meets and joins given by: $P \sqcap Q = \{c \sqcap d \mid c, d \in P \cup Q\}$ and $P \sqcup Q = P \cap Q$. As we shall see, joins correspond to parallel composition.

In the presence of angelic non-determinism (disjunction), it becomes necessary to take the denotation of a program to be a linear closure operator over $\mathcal{P}_I(|D|)$, the indeterminate power-domain of $|D|$ [11]. (Roughly, this power-domain has as elements upwards-closed subsets of $|D|$, ordered by reverse set inclusion. A function is linear if it satisfies the property that $f(S) = \cup_{s \in S} f(\{s\})$.) Such closure operators can also be characterized by the set of their fixed-points, which need not be closed under glbs.