An Open-Ended Finite Domain Constraint Solver

Mats Carlsson[1], Greger Ottosson[2], and Björn Carlson[3]

[1] SICS, PO Box 1263, S-164 29 KISTA, Sweden
[2] Computing Science Dept., Uppsala University
PO Box 311, S-751 05 UPPSALA, Sweden
[3] Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA

Abstract. We describe the design and implementation of a finite domain constraint solver embedded in a Prolog system using an extended unification mechanism via attributed variables as a generic constraint interface. The solver is essentially a scheduler for *indexicals*, i.e. reactive functional rules encoding local consistency methods performing incremental constraint solving or entailment checking, and *global constraints*, i.e. general propagators which may use specialized algorithms to achieve a higher degree of consistency or better time and space complexity.
The solver has an open-ended design: the user can introduce new constraints, either in terms of indexicals by writing rules in a functional notation, or as global constraints via a Prolog programming interface.
Constraints defined in terms of indexicals can be linked to 0/1-variables modeling entailment; thus indexicals are used for constraint solving as well as for entailment testing. Constraints can be arbitrarily combined using the propositional connectives by automatic expansion to systems of reified constraints.
Keywords: Implementation of Constraint Systems, Constraint Programming, Finite Domains, Indexicals, Global Constraints.

1 Introduction

We describe the design and implementation of the SICStus Prolog [11] finite domain constraint solver. The solver is essentially a scheduler for two entities: *indexicals* [24], i.e. reactive functional rules performing incremental constraint solving or entailment checking within the framework of the Waltz filtering algorithm [26], and *global constraints*, i.e. general propagators which may use specialized constraint solving algorithms.

An indexical for solving a constraint $C(X_1, \ldots, X_n)$ has the form X_i in r, where the expression r, called *range*, specifies the feasible values for X_i in terms of the feasible values for $X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n$. The basic idea is to express C as n indexicals, one for each X_i, encoding a local consistency method for solving C. Each indexical is a *projection* of C onto X_i; hence, indexicals are also known as *projection constraints* [21], and have been used in several implementations [9,21,6]. Ranges are defined using a "constraint programming assembly language", which gives the programmer precise control over the level of consistency, and can yield more efficient solutions than relying on the solver's built-in constraints.

An important feature of any constraint solver is *reification*, i.e. the ability to link a constraint to a 0/1-variable reflecting the truth value of the constraint. Thus constraints can be combined using propositional connectives to model cardinality and disjunction simply by flattening to systems of reified constraints and arithmetic constraints over their 0/1-variables. Reification subsumes several frequently used operations such as blocking implication [20] and the cardinality operator [23]. These 0/1-variables may be used in other constraints, thus complex constraints can often be decomposed into a system of many reified, simple constraints.

A crucial operation in reification is entailment detection. It has been shown that indexicals can be used not only for solving constraints, but also for incremental entailment detection [24,5]. The solver handles both kinds of indexicals, ensuring that certain preconditions are satisfied before admitting them for execution. We have built a general reification mechanism for constraints defined by indexicals on top of this idea.

A problem with local propagation methods is the small grain size: each invocation of an indexical does relatively little work (at most *one* domain is pruned), and the overhead of scheduling indexicals for execution becomes noticeable. Furthermore, consider a constraint C expressing some complex relation. Expressing this in terms of primitive constraints, usually through library constraints defined with indexicals, perhaps combined with reification, means spawning many small constraints each maintaining local consistency. However, the consistency for C as a whole can be poor and the large amount of suspensions incurs a scheduling overhead. The alternative is to treat C as a single *global* constraint, using a specialized algorithm that exploits the structure of the problem and retains a high degree of consistency. Also, specialized algorithms that outperform local propagation are available for important classes of constraints. We have addressed these problems by defining a clean interface by which global constraints can be defined in Prolog.

The solver has an open-ended design: the user can introduce new constraints, either in terms of indexicals by writing rules in a functional notation, or as global constraints via a Prolog programming interface.

Constraints defined in terms of indexicals can be linked to 0/1-variables modeling entailment; thus indexicals are used for constraint solving as well as for entailment testing. Constraints can be arbitrarily combined using the propositional connectives by automatic expansion to systems of reified constraints.

Our work has the following contributions.

- It is the first full implementation of an idea [24] to use indexicals to specify the four aspects of a reified constraint, viz. solving the constraint or its negation, and detecting entailment or disentailment of the constraint.
- It is a loosely coupled integration of indexicals into the Prolog abstract machine, with truly minimal extensions on the Prolog side.

- It provides an API for defining global constraints in Prolog.
- It provides mixed execution strategies with indexicals encoding local consistency methods within a Waltz-like algorithm combined with global constraints encoding specialized consistency algorithms. We maintain separate scheduling queues for the two; a global constraint is only resumed when no indexicals are scheduled for execution.
- It extends the indexical language with constructs that e.g. admit arbitrary binary relations to be encoded.
- It shows that a fully-fledged open-ended finite domain system with negative integers, non-linear arithmetic, reification, mixed execution strategies, loosely coupled to a Prolog abstract machine, is possible with competitive performance.

These contributions will be described in detail later. The rest of the paper is structured as follows. Section 2 defines the constraint system and our extended indexical language. Section 3 describes the architecture of the constraint solver, how the Prolog engine had to be extended to provide services specific for FD constraints, and briefly how constraints are compiled to calls to library constraints and/or to indexicals. Section 4 describes the global constraint API. In Sec. 5, we evaluate the basic performance of our reification mechanism, and compare the performance of our solver with four similar systems. Section 6 compares our results with other work. We end with some conclusions about our work.

2 The Constraint System

2.1 Domain Constraints, Indexicals, Entailment and Disentailment

The constraint system is based on domain constraints and indexicals. A *domain constraint* is an expression $X \in I$, where I is a nonempty set of integers. A set S of domain constraints is called a *store*. $[X]_S$, the *domain* of X in S, is defined as the intersection of all I such that $X \in I$ belongs to S (if no such $X \in I$ belongs to S, $[X]_S = \mathcal{Z}$). A store S' is an *extension* of a store S iff $\forall X : [X]_{S'} \subseteq [X]_S$.

The following definitions, adapted from [25], define important notions of consistency and entailment of constraints wrt. stores. Let $[[X]]_S$ denote the interval $\min([X]_S)..\max([X]_S)$:

A constraint C is *domain-consistent wrt.* S iff, for each variable X_i and value V_i in $[X_i]_S$, there exist values V_j in $[X_j]_S$, $1 \leq j \leq n, i \neq j$, such that $C(V_1, \ldots, V_n)$ is true. A constraint C is *domain-entailed by* S iff, for all values V_j in $[X_j]_S$, $1 \leq j \leq n$, $C(V_1, \ldots, V_n)$ is true. A constraint C is *interval-consistent wrt.* S iff, for each variable X_i there exist values V_j in $[[X_j]]_S$, $1 \leq j \leq n, i \neq j$, such that $C(V_1, \ldots, \min([X_i]_S), \ldots, V_n)$ and $C(V_1, \ldots, \max([X_i]_S), \ldots, V_n)$ are both true. A constraint C is *interval-entailed by* S iff, for all values V_j in $[[X_j]]_S$, $1 \leq j \leq n$, $C(V_1, \ldots, V_n)$ is true. Finally, a constraint is *domain-disentailed (interval-disentailed) by* S iff its negation is domain-entailed (interval-entailed) by S.

$$N ::= x \mid i, \text{ where } i \in \mathcal{Z} \mid \infty \mid -\infty$$

$$\begin{aligned} T ::= {} & N \mid T+T \mid T-T \mid T*T \mid \lceil T/T \rceil \mid \lfloor T/T \rfloor \mid T \bmod T \\ \mid {} & \min(x) \mid \max(x) \mid \mathrm{card}(x) \end{aligned}$$

$$\begin{aligned} R ::= {} & T..T \mid R \cap R \mid R \cup R \mid R \mathbin{?} R \mid \backslash R \\ \mid {} & R+T \mid R-T \mid R \bmod T \\ \mid {} & \mathrm{unionof}(x,R,R) \mid \mathrm{switch}(T,F) \mid \mathrm{dom}(x) \\ \mid {} & \text{a finite subset of } \mathcal{Z} \end{aligned}$$

$$F ::= \text{a finite mapping from } \mathcal{Z} \text{ to } R$$

Fig. 1. Syntax of range expressions

An indexical has the form x in r, where r is a *range* (generated by R in Fig. 1). When applied to a store S, x in r evaluates to a domain constraint $x \in r_S$, where r_S is the value of r in S (see below).

The value of a term t in S, t_S, is an integer computed by the scalar functions defined by T in Fig. 1. The expressions $\min(y), \max(y)$, and $\mathrm{card}(y)$ evaluate to the minimum, maximum, and size of $[y]_S$, respectively.

The value of a range r in S, r_S, is a set of integers computed by the functions defined by R in the figure. The expression $\mathrm{dom}(y)$ evaluates to $[y]_S$. The expression $t..t'$ denotes the interval between t_S and t'_S, and the operators $\cup$, $\cap$ and $\backslash$ denote union, intersection and complement respectively. The conditional range $r \mathbin{?} r'$ [21,4] evaluates to r'_S if $r_S \neq \emptyset$ and $\emptyset$ otherwise. The expressions $r+t$, $r-t$, and $r \bmod t$ denote the integer operators applied point-wise.

We have introduced two new range expressions that make it possible to encode arbitrary binary relations as indexicals:

- The value of the expression $\mathrm{switch}(t, f)$ in S is the set $f(t_S)_S$, if t_S is in the domain of f, or $\emptyset$ otherwise. This is implemented as a simple hash table.
- The value of the expression $\mathrm{unionof}(x, d, e)$ is $\bigcup_{x \in d_S} e_S$ i.e. x is quantified by the expression and is assumed to occur in e only. The implementation resembles a "for" loop over the elements of d_S.

For example, let $p(X, Y)$ denote the binary relation

$$\{(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)\}$$

Using the new range expressions, $p(X, Y)$ can be encoded by the two indexicals

```
X in unionof(B,dom(Y),switch(B,[1-{1,2,3},2-{2,3},3-{3}])),
Y in unionof(B,dom(X),switch(B,[1-{1},2-{1,2},3-{1,2,3}])).
```

$[x]_S$ **related to** r_S	r **monotone in** S	r **anti-monotone in** S
$[x]_S \cap r_S = \emptyset$	inconsistent	may become entailed
$[x]_S \subseteq r_S$	may become inconsistent	entailed
$[x]_S \neq ([x]_S \cap r_S) \neq \emptyset$	may become inconsistent	may become entailed

Table 1. Entailment/Inconsistency of x in r in a store S

2.2 Monotonicity of Indexicals

A range r is *monotone in* S iff for every extension S' of S, $r_{S'} \subseteq r_S$. A range r is *anti-monotone in* S iff for every extension S' of S, $r_S \subseteq r_{S'}$. By abuse of notation, we will say that x in r is (anti-)monotone iff r is (anti-)monotone.

The consistency and entailment of x in r in a store S is checked by considering the relationship between $[x]_S$ and r_S, together with the monotonicity of r in S (see Tab. 1). Suppose an indexical x in r is executed in store S where r is monotone in S. If $[x]_S \cap r_S = \emptyset$, an inconsistency is detected. Otherwise, if $[x]_S \subseteq r_S$, $[x]_S$ is already contained in the set of values that are compatible with r_S. Otherwise, $[x]_S$ contains some values that are incompatible with r_S. Hence, $x \in r_S$ is added to S, and we say that x is *pruned*.

3 The Constraint Solver

3.1 Design

The solver is essentially a scheduler for two entities: indexicals performing constraint solving within the framework of the Waltz filtering algorithm [26], and global constraints, i.e. general propagators which may use specialized consistency algorithms. At the heart of the solver is an evaluator for indexicals, i.e. an efficient implementation of the decision table shown in Tab. 1.

Indexicals and global constraints can be added by the programmer, giving precise control over aspects of the operational semantics of constraints. Trade-offs can be made between the computational cost of the constraints and their pruning power. They can yield more efficient solutions than relying on the solver's built-in constraints.

The indexical language provides many degrees of freedom for the user to select the level of consistency to be maintained depending on application-specific needs. For example, the constraint $X = Y + C$ may be defined as indexicals maintaining domain-consistency or interval-consistency as `eqcd/3` or `eqci/3` respectively of Fig. 2. The notation is explained in Sec. 3.2.

It is of course possible to write indexicals which don't make sense. Basically, an indexical only has declarative meaning if the set denoted by the range is monotonically decreasing. Consider the definition of a constraint C containing an indexical X in r. Let $\mathcal{T}(X, C, S)$ denote the set of values for X that can make C true in some ground extension of the store S. Then the indexical should obey the following coding rules:

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

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

Fig. 2. Indexicals expressing $X = Y + C$

1. if r is ground, $r_S = \mathcal{T}(X, C, S)$
2. if r is monotone, $r_S \supseteq \mathcal{T}(X, C, S)$
3. all arguments of C except X should occur in r

Fig. 3. Coding rules for a propagating indexicals X in r

Rule 1 says that it is safe to consider C entailed after pruning X if r is ground. This is a significant optimization [5], and is exploited as follows: Indexicals that are projections of the same constraint, as e.g. in `eqcd/3` in Fig. 2, are connected by references to a common flag. Whenever one of the indexicals is decided entailed, the flag is set. Before any indexical is executed, its associated entailment flag is checked and if set the indexical is ignored. The same optimization is used elsewhere [9,6].

Rule 2 is implied by rule 1 as follows: For all extensions S' of S that make r ground, if r is monotone, $r_{S'} \subseteq r_S$ and hence $\mathcal{T}(X, C, S') \subseteq r_S$. Rule 2 follows from this since, by definition, $\mathcal{T}(X, C, S)$ is the union of all $\mathcal{T}(X, C, S')$. The solver relies on this rule by requiring that X in r be monotone before admitting it for execution.

Finally, rule 3 is a natural consequence of rule 1 for any reasonable constraint.

It has been shown that anti-monotone indexicals can be used for expressing logical conditions for entailment detection [5]. As we will show in the following section, a reification mechanism can be built on top of indexicals. Coding rules analogous to those in Fig. 3 apply for indexicals detecting entailment.

The solver has been extended to handle both kinds of indexicals, and ensures that the (anti-)monotonicity precondition is satisfied before admitting any indexical for execution. This is achieved by suspending the indexicals until certain variables are ground. The set of variables to suspend on is easily computed at compile time [5]. For example, in `eqcd/3` in Fig. 2, the first indexical is not admitted for execution until C is known.

We maintain separate separate scheduling queues for indexicals and global constraints; a global constraint is only resumed when no indexicals are scheduled for execution. Thus, global constraints can be seen as having lesser priority than indexicals. This is reasonable, since indexicals are cheap to invoke (but may perform little useful work), while specialized algorithms for global constraints can be expensive (but yield many conclusions when invoked). Some other systems can assign different priorities to individual constraints [12,22].

The solver also provides the usual predefined search strategies (fixed order, fail-first principle, branch and bound for minimization or maximization) and utility predicates for accessing domains, execution statistics, controlling aspects of the answer constraint, etc.

3.2 FD Predicates and Reification

We have minimally extended the Prolog system by admitting the definition of constraints as the one shown in Fig. 2. The constraints become known to the Prolog system as *FD predicates*, and when called, the Prolog engine simply escapes to the constraint solver. In our design, contrary to e.g. clp($\mathcal{FD}$), indexicals can only appear in the context of FD predicate definitions; they are not constraints but projections of constraints.

The definitions in Fig. 2 provide methods for solving $X = Y + C$ constraints. If we want to reify such constraints, however, we need methods for detecting entailment and disentailment, and for solving the negated constraint. Thus, FD predicates may contain up to four "clauses" (for solving the constraint or its negation, and for checking entailment or disentailment). The role of each clause is reflected in the "neck" operator. Indexicals used for constraint solving are called *propagating*. Those used for entailment checking are called *checking*. Table 2 summarizes the different cases. Figure 4 shows the full definition of our example constraint with all four clauses for domain-consistency and -entailment. The reified constraint may be used as follows, expressing the constraint $U + V = 5 \Leftrightarrow B$:

```
?- eqcd(U,V,5) iff B.
```

The implementation spawns two coroutines corresponding to the clauses for detecting entailment and disentailment. Eventually, one of them will bind B to 0 or 1. A third coroutine is spawned, waiting for B to become bound, at which time the clause for posting the constraint (or its negation) is activated. In the mean time, the constraint may have been detected as (dis)entailed, in which case the third coroutine is dismissed.

role	**neck symbol**	**indexical type**	**precondition**
solve C	`+:`	propagating	monotone
solve $\neg C$	`-:`	propagating	monotone
check C	`+?`	checking	anti-monotone
check $\neg C$	`-?`	checking	anti-monotone

Table 2. Roles of FD predicate clauses

Alternative encodings of reification are described in Sec. 5.1.

```
eqcd(X,Y,C) +: % positive constraint solving
    X in dom(Y)+C,
    Y in dom(X)-C.
eqcd(X,Y,C) -: % negative constraint solving
    X in \{Y+C},
    Y in \{X-C}.
eqcd(X,Y,C) +? % entailment detection
    X in {Y+C}.
eqcd(X,Y,C) -? % disentailment detection
    X in \dom(Y)+C.
```

Fig. 4. Indexicals for reifying $X = Y + C$

3.3 Prolog Engine Extensions

The Prolog engine had to be extended to be able to cope with calls to FD predicates. This was done by introducing the FD predicate as a new predicate type known to the Prolog emulator. The emulator's call instruction dispatches on the type of the predicate being called, and if FD predicates are called as Prolog goals, the emulator will escape to the solver. No new abstract machine instructions were introduced.

FD predicate definitions are compiled by a source-to-source translation mechanism into directives that will store the object code and insert the new predicate into the Prolog symbol table. The Prolog compiler proper was not extended at all. Indexical ranges are compiled into postfix notation, which is then translated by the loader into byte code for a threaded-code stack machine. A small set of solver primitives provides the necessary back-end, managing memory, storing byte code in-core, etc.

The interface between the Prolog engine and the solver is provided in part by the attributed variables mechanism [13], which has been used previously to interface several constraint solvers into CLP systems [10,12,25,9,6,22]. This mechanism associates solver-specific information with variables, and provides hooks for extended unification and projection of answer constraints.

Thus, the only extension to the Prolog kernel was the introduction of a new predicate type, a truly minimal and modular extension.

3.4 Macro-expansion of Goals

The indexical language can be regarded as a low-level language for programming constraints. It is usually not necessary to resort to this level of programming—most commonly used constraints are available via library calls and/or via macro-expansion.

A very common class of constraints are equations, disequations and inequations, and propositional combinations of these. These are translated by a built-in macro-expansion mechanism into sequences of library constraint goals. The expanded code is linear in the size of the source code. Similar expansions are used in most other systems. Again, the Prolog compiler proper is not aware of this macro-expansion. For example, the Prolog clause:

```
p(X, Y, Z) :-
        X+2*Y+3*Z#>=4 #\/ 4*X+3*Y+2*Z #=< 1.
```

is expanded to:

```
p(X, Y, Z) :-
        scalar_product([-1,1,2,3], [D,X,Y,Z], #=, 0),
        4 #=< D iff E,
        scalar_product([-1,4,3,2], [F,X,Y,Z], #=, 0),
        F #=< 1 iff G,
        clpfd:'p\\/q'(E, G, 1).
```

4 The Global Constraint Interface

We have developed a programming interface by means of which new global constraints can be defined in Prolog. Constraints defined in this way can take arbitrary arguments and may use any constraint solving algorithm, provided it makes sense.

The interface maintains a private state for each invocation of a global constraint. The state may e.g. contain the domains of the most recent invocation, admitting consistency methods such as AC-4 [15]. The interface also provides means to access the domains of variables and operations on the internal domain representation.

To make the solver aware of a new global constraint, the user must assert a Prolog clause

```
dispatch_global(Constraint,
                State0,
                State, Actions) :- Body.
```

which the solver will call whenever a constraint of the new type is posted or resumed. A `dispatch_global` goal is true if *Constraint* is the constraint term itself, *State0* is the current state of the invocation, the conjunction *Body* succeeds, unifying *State* with the updated state and *Actions* with a list of requests to the solver. Such requests include notifications that the constraint has been detected entailed or disentailed, requests to prune variables, and requests to rewrite the constraint into some simpler constraints. *Body* is not allowed to change the state of the solver e.g. by doing recursive constraint propagation, as the scheduling queues are under the control of the solver and not globally accessible.

A global constraint invocation is posted to the solver by calling `fd_global(`*Constraint*`,`*State*`,`*Susp*`)` where *Constraint* is the constraint, *State* the initial state of the invocation, and *Susp* encodes how the constraints should be suspended on the variables occurring in it. A full example is shown in Fig. 5.

```
le_iff(X,Y,B) :-
    B in 0..1,      % suspend on bounds of X,Y and on value of B
    fd_global(le(X,Y,B), [], [minmax(X),minmax(Y),val(B)]).

:- multifile dispatch_global/4.
dispatch_global(le(X,Y,B), [], [], Actions) :-
    le_solver(B, X, Y, Actions).

le_solver(B, X, Y, Actions) :- var(B), !,
    (   fd_max(X, Xmax), fd_min(Y, Ymin), Xmax=<Ymin
    ->  Actions = [exit,B=1]          % entailed, B=1
    ;   fd_min(X, Xmin), fd_max(Y, Ymax), Xmin >Ymax
    ->  Actions = [exit,B=0]          % entailed, B=0
    ;   Actions = []                  % not entailed, no pruning
    ).
le_solver(0, X, Y, [exit,call('x>y'(X,Y))]).  % rewrite to X#>Y
le_solver(1, X, Y, [exit,call('x=<y'(X,Y))]). % rewrite to X#=<Y
```

Fig. 5. $x \leq y \Leftrightarrow b$ as a global constraint

Reification cannot be expressed in this interface; instead, reification of a global constraint may be achieved by explicitly passing it a 0/1-variable. Figure 5 illustrates this technique too.

Many of the solver's built-in constraints are implemented via this programming interface, for example:

- non-linear arithmetic constraints,
- constraints expressing sums and scalar products of a list of domain variables,
- all_different(L), constraining the elements of the list L to take distinct values. We have implemented a weak version simulating pairwise inequalities as well as a strong version based on Régin's algorithm [19].
- element(I, L, Y), constraining the I:th element of L to be equal to Y, uses a consistency algorithm based on AC-4.
- cumulative(S, D, R, L), modelling a set of tasks with start times S, durations D, and resource need R, sharing a resource with capacity L [1]. The implementation is based on several OR methods [7,2].

5 Performance Evaluation

The performance evaluation of our solver is structured as follows. First, we compare our low-level implementation of reification with alternative schemes. Then, we measure the general performance of our solver on a set of benchmark programs, and compare it with four similar CLP systems.

5.1 The Reification Mechanism

The mechanism as described in Sec. 3.2 is but one of several possible implementation options. There are well-known techniques in OR for reifying linear arithmetic constraints [27]. Sidebottom showed how reification can be encoded into indexicals using the conditional range operator [21]. Finally, reification can be expressed by a global constraint.

We measured the performance of these four schemes on the simple example $x, y \in 1..10,\ x \leq y \Leftrightarrow b$. Having posted this constraint, a failure driven loop was executed 10000 times, executing each of the relevant cases ($b = 1$, $b = 0$, entailment, disentailment). The four constraint formulations are listed in Figs. 5 and 6. The constant 10000 that occurs in the version using the OR method is an arbitrarily chosen sufficiently large constant. Our low-level reification mechanism was the fastest, with conditional ranges being some 17% slower, The OR method method some 76% slower, and the global constraint formulation some 82% slower.

```
le1(X,Y,B) :-           % low-level method
    X #=< Y iff B.

le2(X,Y,B) :-           % OR method
    B in 0..1,
    Xmax #=  X + 10000*B,
    Xmax #=< Y + 10000,
    Y    #<  Xmax.

le3(X,Y,B) +:           % conditional range method
    X in ((1..B) ? (inf..max(Y))) \/ ((B..0) ? (min(Y)+1..sup)),
    Y in ((1..B) ? (min(X)..sup)) \/ ((B..0) ? (inf..max(X)-1)),
    B in ((min(X)..max(Y)) ? (1..1)) \/ ((min(Y)+1..max(X)) ? (0..0)).
```

Fig. 6. Three encodings of $x \leq y \Leftrightarrow b$

5.2 Benchmark Results

The first part of Tab. 3 shows execution times for a set of small, well-known benchmark programs, with numbers in parentheses indicating relative times wrt. SICStus. The programs have been chosen to be clean and fairly representative of real-world problems, and coded straight-forwardly in a way a programmer without deep system specific knowledge would. Naive variable ordering has been used for all problems, as different first-fail implementations tend to break ties in slightly different ways. Where applicable, built-in constraints such as `all_different/1`, `element/3` and `atmost/3` have been used, whereas more complex global constraints are not used in these programs. For example, in the Squares 21 (packing 21 squares into a large square) benchmark, the constraint that no square can overlap with any other square is expressed with cardinality over four inequations and not with CHIP's `diffn/1` constraint. Thus, the collected figures represent a notion of the basic performance of a system.

The systems tested besides SICStus are CHIP version 5.0.0, ECL^{i}PS^e version 3.5.2, B-Prolog version 2.1 [29] and clp($\mathcal{FD}$) version 2.21. All benchmarks have been run on (or normalized to) a SUN SPARCstation 4 with a 85MHz microSPARC CPU and the times shown are in milliseconds. The performance comparison has been limited to the above systems since these are the Prolog based systems that were available to us, but could easily be extended. Furthermore, SICStus and Oz [22] have almost identical performance on the `alpha`, `eq10` and `eq20` benchmarks [28].

Of the eight programs, the last two, Magic 20 (magic series of length 20) and Squares 21 (packing 21 squares into a large square) need reification, which is not supported in clp($\mathcal{FD}$), B-Prolog and only partly supported in CHIP. The figure for Magic 20 with clp($\mathcal{FD}$) is with reification implemented using the OR method, in CHIP using conditional clauses and in B-Prolog reification has been done using delay-clauses. Squares 21 could not be run on B-Prolog due to lack of support for non-linear arithmetic, and on clp($\mathcal{FD}$) due to memory allocation problems that we were unable to debug.

Benchmarks

Problem	**clp($\mathcal{FD}$)**	**B-Prolog**	**CHIP**	**ECL^{i}PS^e**	**SICStus**
Alpha	3900 (0.23)	14800 (0.89)	42600 (2.55)	100000 (5.99)	16700
Cars$^{50\ \text{times}}$	970 (0.20)	2400 (0.50)	3700 (0.77)	4400 (0.92)	4800
Eq10$^{50\ \text{times}}$	2000 (0.35)	3900 (0.70)	6100 (1.09)	9100 (1.63)	5600
Eq20$^{50\ \text{times}}$	3400 (0.40)	11000 (1.29)	11500 (1.35)	18000 (2.12)	8500
Queens-8 all$^{10\ \text{times}}$	850 (0.20)	1000 (0.23)	1800 (0.42)	7500 (1.74)	4300
Queens-16 one$^{10\ \text{times}}$	5500 (0.21)	10000 (0.38)	12500 (0.48)	60000 (2.31)	26000
Magic 20	630 (1.05)	3800 (6.33)	6800 (11.3)	3300 (5.50)	600
Squares 21	n/a	n/a	59000 (1.98)	171000 (5.74)	29800
Arithmetic Mean	(0.38)	(1.5)	(3.2)	(3.2)	(1.0)
Harmonic Mean	(0.28)	(0.56)	(1.0)	(2.16)	(1.0)

Constraint Executions

1M prunings	806 (0.08)	5400 (0.55)	957 (0.09)	42710 (4.38)	9760

Table 3. Performance results for selected problems

The second part of Tab. 3 shows the performance on the unsatisfiable constraints $x, y \in 1..500000$, $x < y$, $y < x$. Posting these constraints will make the last two constraints trigger each other iteratively until failure, i.e. one million invocations and prunings. This gives an idea of the raw speed of the solvers on *intervals*.

Although the results show some peaks, e.g. for Alpha, Magic 20 and 1M, we conclude that SICStus shows performance comparable with that of all systems tested. SICStus performs significantly better than ECL^{i}PS^e, while it is lagging behind clp($\mathcal{FD}$). It is worth noting that SICStus performs quite well on the benchmarks that use reification, which shows that low-level support for reification is valuable.

6 Comparison with Other Work

A full comparison with all existing finite domain constraint solvers is clearly outside the scope of this paper. In this section, we will focus on particular features of our solver and how they relate to some other known systems.

Indexicals Indexicals were first conceived in the context of cc($\mathcal{FD}$) [24,25]. The vision was to provide a rational, *glass box* reconstruction of the FD part of CHIP[10], replacing a host of ad-hoc concepts by a small set of powerful concepts and combinators such as blocking implication and the cardinality operator. Indexicals were a key component of the design, but seem to have been abandoned later. Unfortunately, no implementation of cc($\mathcal{FD}$) is available for comparison. Other systems [9,21,6] have been solely based on indexicals. Notably, clp($\mathcal{FD}$) [9] demonstrated the feasibility of the indexical approach, achieving excellent performance by compiling to C. Our design is the first to be based on a mixture of indexicals and global constraints, compiling indexicals to byte code for a threaded-code stack machine. The comparison with clp($\mathcal{FD}$) indicates that indexicals require a tight integration, compiled to C or native code, to achieve truly competitive performance.

The indexical scheme can be readily extended, with for example conditional ranges [21,6], with "foreach" constructs as in our design, or with arbitrary functions written in C [9]. A generalization of indexicals to *m-constraints* encoding path-consistency methods was proposed in [8].

Reification CLP(BNR) [16] was the first system to allow propositional combinations of arithmetic constraints by means of reification. This is now allowed by many systems including ours [21,12,22,17]. Other systems provide blocking implication [10,25] or cardinality [25]. Only research prototypes have no reification support.

We have provided a full implementation of an idea [24] to use indexicals to specify the four aspects of a reified constraint, viz. solving the constraint or its negation, and detecting entailment or disentailment of the constraint.

Global constraints It is well known that local constraint propagation, even with reification, can be too weak. A constraint involving many variables, e.g. the constraint that the elements of a list all be distinct, may be modeled by $O(N^2)$ disequations. If the same constraint is expressed as a single, global constraint, we get much better ($O(N)$) space complexity, much smaller scheduler overhead, and the opportunity to employ a specialized, complete filtering algorithm [19] instead of merely mimicking the pairwise disequations. The need for specialized algorithms is most obvious on hard combinatorial problems [1,3,18], while the space complexity aspects can dominate on large instances of otherwise easy problems.

Consequently, solvers based solely on indexicals can hardly be competitive on these classes of problems. On the other hand, indexicals admit rapid prototyping of user-defined constraints: defining a global constraint usually requires much more programming effort. Also, in our implementation, an indexical formulation often outperforms a global one if the constraint involves few variables. The break-even point has not been determined.

Most solvers are based solely on what we have called global constraints, as e.g. [12,22,17,10,14]. Ours is based on a mixed approach, combining the best of both worlds.

Programming interfaces Any system that is not completely closed needs a programming interface for defining new constraints. In indexical-based systems, the indexical language provides such an interface. clp($\mathcal{FD}$) [9] allows the use of arbitrary C functions in indexicals. ECL^{i}PS^e [12] uses attributed variables as a generic constraint interface. By accessing these attributed variables and calling internal coroutining primitives, user-defined constraints can be programmed. Oz [22], Ilog Solver [14] and CHIP [10] provide programming interfaces in terms of C++ classes. CHIP also provides declarations that allow the user to use arbitrary Prolog code as constraints; we provide the same ability via a simple API.

Negative numbers In many finite domain constraint solvers, the constraints are over natural numbers [22,9,25,6,10]. The extension to the full integer domain strictly extends the expressive power of the language so that it can reason e.g. about differences, but complicates the non-linear arithmetic constraints somewhat. We share this extensions with some other systems [12,16,21,17].

Host language integration The design of clp($\mathcal{FD}$) [9] extends the underlying Prolog engine with several new abstract machine instructions supporting constraints, and compiles all source code to C.

AKL($\mathcal{FD}$) [6] integrated the indexical approach into a concurrent constraint language with deep guards and a generic constraint interface on the level of C. Constraint system specific methods for e.g. garbage collection must be provided in this interface.

As in ECL^{i}PS^e, we used attributed variables as a generic constraint interface, and minimally extended the Prolog engine by the FD predicate mechanism, handling all compilation issues by source-to-source translation.

7 Conclusions

We describe the design and implementation of the SICStus Prolog finite domain constraint solver. The solver has an open design, supports reification, and allows constraints to be added by the user by two complementary mechanism: (a) as indexicals that perform incremental constraint solving and entailment checking within a Waltz-like algorithm, and (b) as global constraints via a Prolog programming interface, admitting specialized consistency methods. We describe a loosely coupled integration of finite domain constraints into the Prolog abstract machine; thus the techniques can be generalized to other constraint systems. We extend the indexical language, thus enabling the encoding of arbitrary binary relations as indexicals. We compare the performance and functionality of the design with other work.

We have shown that a fully-fledged open-ended finite domain system with negative integers, non-linear arithmetic, reification, mixed execution strategies, loosely coupled to a Prolog abstract machine, is possible with competitive performance.

Acknowledgements

The research reported herein was supported in part by the Advanced Software Technology Center of Competence at Uppsala University (ASTEC), and in part by the Swedish Institute of Computer Science (SICS).

References

1. A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7):57–73, 1993.
2. P. Baptiste and C. Le Pape. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Canada, August 1995.
3. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.
4. B. Carlson and M. Carlsson. Compiling and Executing Disjunctions of Finite Domain Constraints. In *Proceedings of the Twelfth International Conference on Logic Programming*. MIT Press, 1995.
5. B. Carlson, M. Carlsson, and D. Diaz. Entailment of finite domain constraints. In *Proceedings of the Eleventh International Conference on Logic Programming*. MIT Press, 1994.
6. B. Carlson, M. Carlsson, and S. Janson. The implementation of AKL(FD). In *Logic Programming: Proceedings of the 1995 International Symposium*. MIT Press, 1995.
7. Y. Caseau and F. Laburthe. Improved clp scheduling with task intervals. In P. Van Hentenryck, editor, *Logic Programming, Proceedings of the Eleventh International Conference on Logic Programming*, pages 369–383, Santa Margherita Ligure, Italy, 1994. MIT Press.
8. Philippe Codognet and Giuseppe Nardiello. Enhancing the constraint-solving power of clp(FD) by means of path-consistency methods. In A. Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 37–61. Springer-Verlag, 1995.
9. D. Diaz and P. Codognet. A Minimal Extension of the WAM for CLP(FD). In *Proceedings of the International Conference on Logic Programming*, pages 774–790, Budapest, Hungary, 1993. MIT Press.

10. 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*, 1988.
11. Mats Carlsson et al. SICStus Prolog User's Manual. SICS Research Report, Swedish Institute of Computer Science, 1995.
URL: http://www.sics.se/isl/sicstus.html.
12. Micha Meier et al. ECLiPSe user manual. ECRC Research Report ECRC-93-6, European Computer Research Consortium, 1993.
13. C. Holzbaur. *Specification of Constraint Based Inference Mechanism through Extended Unification*. PhD thesis, Dept. of Medical Cybernetics and AI, University of Vienna, 1990.
14. ILOG. ILOG Solver C++. Reference manual, ILOG S.A., 1993.
15. R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
16. W. Older and A. Vellino. Constraint arithmetic on real intervals. In *Constraint Logic Programming: Selected Research (eds. Benhamou and Colmerauer)*. MIT Press, 1993.
17. PrologIA. Le manuel de Prolog IV. Reference manual, PrologIA S.A., 1997.
18. J.-F. Puget and M. Leconte. Beyond the glass box: Constraints as objects. In J. Lloyd, editor, *Proceedings of the International Logic Programming Symposium (ILPS-95)*, pages 513–527, Portland, 1995.
19. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, 1994.
20. Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1990.
21. Gregory Sidebottom. *A Language for Optimizing Constraint Propagation*. PhD thesis, Simon Fraser University, November 1993.
22. Gert Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, 1995.
23. 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.
24. Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint processing in cc(FD). Draft, Computer Science Department, Brown University, 1991.
25. Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation and evaluation of the constraint language cc(FD). In A. Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
26. D. Waltz. *The Psychology of Computer Vision (Ed. P. Winston)*, chapter Understanding line drawings of scenes with shadows. McGraw-Hill, New York, 1975.
27. H.P. Williams. *Model Building in Mathematical Programming*. J. Wiley and sons, New York, 1978.
28. Jörg Würtz, 1997. Personal Communication.
29. Neng-Fa Zhou. B-Prolog User's Manual Version 2.1. Technical report, Kyushu Institute of Technology, 1997.
URL: http://www.cad.mse.kyutech.ac.jp/people/zhou/bprolog.html.