The Anatomy of the SICStus Finite-Domain Constraint Solver

Mats Carlsson

RI.
SE

August 20, 2025

CLP(FD): Embedding in Prolog

search	Backtrack search!
variables	Logical variables with *attributes*
unification	Domain becomes singleton $\leftrightarrow$ logical variable gets bound
solver state	Backtrackable terms, using *mutables* and memory management hooks

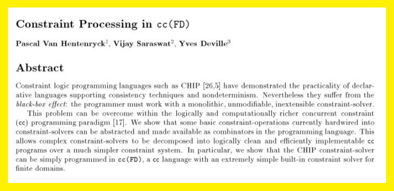
References

Christian Holzbaur: *Metastructures vs. Attributed Variables in the Context of Extensible Unification*. PLILP 1992: 260-268.

Abderrahmane Aggoun, Nicolas Beldiceanu: *Time Stamps Techniques for the Trailed Data in Constraint Logic Programming Systems*. SPLT 1990: 487-510.

The Glass-Box Approach to Constraint Programming

Pascal Van Hentenryck et al. suggested a clean approach to constraint propagation:



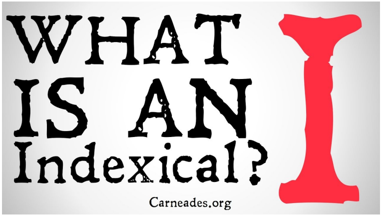
Constraint Processing in cc(FD)

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.



Reference

Pascal Van Hentenryck, Vijay A. Saraswat, Yves Deville: *Design, Implementation, and Evaluation of the Constraint Language cc(FD)*. Constraint Programming 1994: 293-316.

Indexicals: Key Idea

Consider $c(x_1, x_2, x_3)$. When the coroutine wakes up:

- $dom(x_1) \leftarrow dom(x_1) \cap f_1(dom(x_2), dom(x_3))$
- $dom(x_2) \leftarrow dom(x_2) \cap f_2(dom(x_1), dom(x_3))$
- $dom(x_3) \leftarrow dom(x_3) \cap f_3(dom(x_1), dom(x_2))$

Also known as *projection constraints*.

A First Implementation, ≈1998

- Attributes for domain variables, unification, and answer constraints
- Indexicals for propagation, with a programming API in Prolog
- Functional notation for arithmetic & Boolean constraints
- Domain representation: lists of unconnected intervals

Reference

Antonio J. Fernández, Patricia M. Hill: *A Comparative Study of Eight Constraint Programming Languages Over the Boolean and Finite Domains*. Constraints 5(3): 275-301 (2000)

Pros and Cons of Indexicals

- $(+)$They are lightweight and fast, e.g., for simple arithmetic constraints
- $(+)$Efficient, simple stack machine implementation
- $(-)$They take a *fixed number of arguments*
- $(-)$They are *stateless*—bad for incrementality
- $(-)$They are incapable of *deep reasoning* and propagate *global constraints* weakly
- …

An Evolving Implementation

SICStus evolved into a menagerie of propagator types:

- Indexicals (less and less)
- Global constraints (more and more)
- Boolean constraints (but no SAT solver)
- Daemons
- Reals (as of 4.10)

SICStus Global Constraints

`all_different/[1,2]`	`all_different_except_0/1`	`all_distinct/[1,2]`
`all_distinct_except_0/1`	`all_equal/1`	`all_equal_reif/2`
`assignment/[2,3]`	`automaton/[3,8,9]`	`bin_packing/2`
`bool_and/2`	`bool_channel/4`	`bool_or/2`
`bool_xor/2`	`case/[3,4]`	`circuit/[1,2]`
`count/4`	`cumulative/[1,2]`	`cumulatives/[2,3]`
`decreasing/[1,2]`	`diffn/[1,2]`	`disjoint1/[1,2]`
`disjoint2/[1,2]`	`element/[2,3]`	`geost/[2,3,4]`
`global_cardinality/[2,3]`	`increasing/[1,2]`	`keysorting/[2,3]`
`lex_chain/[1,2]`	`maximum/2`	`maximum_arg/2`
`minimum/2`	`minimum_arg/2`	`multi_cumulative/[2,3]`
`network_flow/[2,3]`	`nvalue/2`	`regular/2`
`relation/3`	`scalar_product/[4,5]`	`scalar_product_reif/[5,6]`
`seq_precede_chain/[1,2]`	`smt/1`	`sorting/3`
`subcircuit/[1,2]`	`sum/3`	`symmetric_all_different/1`
`symmetric_all_distinct/1`	`table/[2,3]`	`value_precede_chain/[2,3]`

Thanks

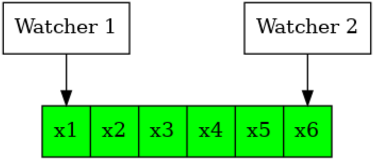
A lot of joint work with Nicolas Beldiceanu ≈ 2001–2017

RI.
SE

Boolean Variables and Constraints

Everything is faster for Boolean (0/1) variables:

- Pruning means fixing to 0 or 1
- Simpler propagation queue
- Internal shortcuts
- Do Booleans before globals
- Use *watchers* for disjunctions ($x_1 \vee x_2 \vee \cdots \vee x_n$)

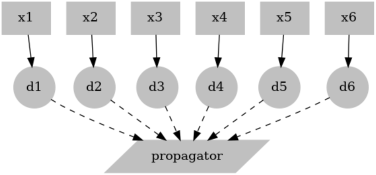


Reference

Moskewicz, Matthew W., et al. *Chaff: Engineering an efficient SAT solver*. Proc. 38th annual Design Automation Conference. 2001.

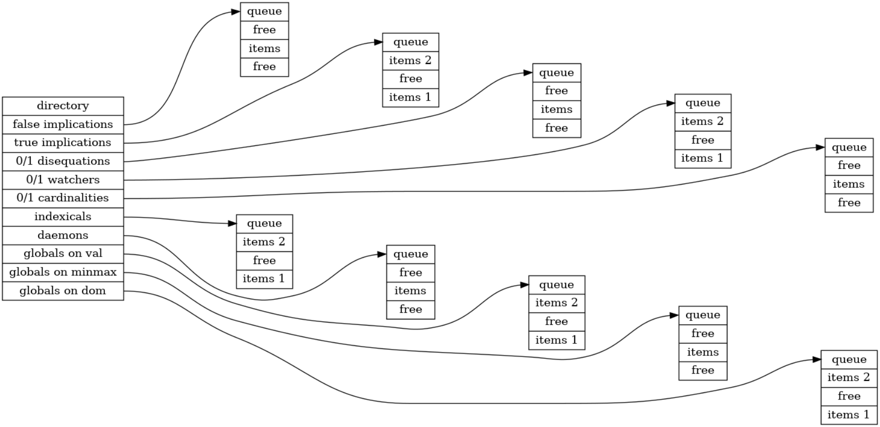
Daemons

- The problem: many propagators are somewhat heavy and often prune nothing
- A daemon is a quick check whether to run a propagator
- A daemon can help maintain propagator state
- A daemon knows *which* variable was pruned
- Enqueue the daemon, not the propagator



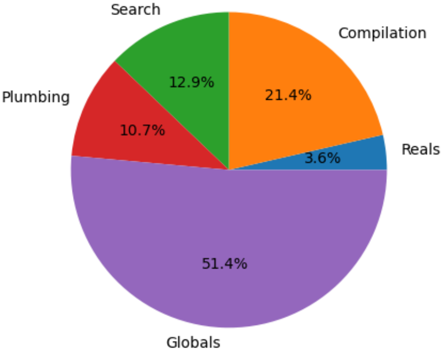
Propagation Queue Structure

By decreasing priority

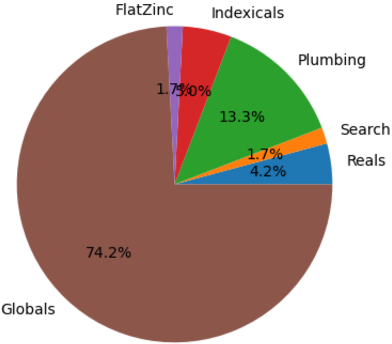


Code Base

Prolog: 14000 loc



C: 60000 loc



Variables And Constraints Over Reals?

- The physical world deals with $\mathbb{R}$-valued quantities
- In the physical world, laws of physics apply
- Modeling should be convenient
- First use case: product configuration
- Second use case: MiniZinc

Conventional Wisdom: Reals Can Be Modeled As Rationals

Then solver doesn't need changing at all

Problems with that:

- Modeling gets awkward or loses precision
- What about transcendental functions (sin, cos, sqrt, ...)?
- Exact rational arithmetic is prone to size explosion
- Consider, e.g., 3.14159265

Adding Reals: A Bitter Pill To Swallow

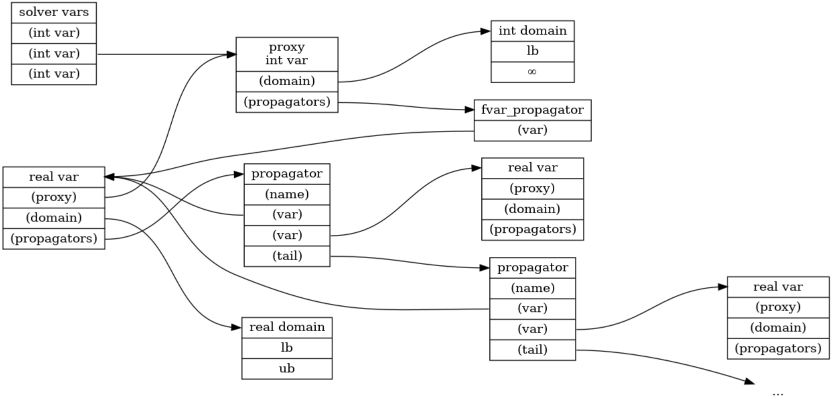
- Code structure: Polymorphism everywhere?
- Propagation: another fixpoint algorithm for reals?
- Domain representation?
- Rounding errors?
- Reals are not floats!
    - $3 \cdot x = 1.0$
- Constraints or expressions?
- Search

Adding Reals: Related Work

- Interval arithmetic (and constraint solving)
- F. Benhamou, W.J. Older, T.J. Hickey, A. Vellino, E. Hyvönen, P. Van Hentenryck, L. Michel, Y. Deville (to name a few)
- CLP(BNR), Numerica, Gecode (C++ based), OR-Tools (C++ based)

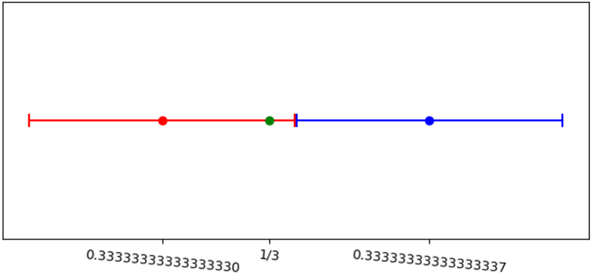
Adding Reals: Propagation and Domains

Piggy-back on existing fixpoint algorithm



- Every real var x has a proxy int var $x*$, known to solver
- Whenever x gets pruned, increase lb of $x*$
- Attached to $x*$ is a special propagator that runs propagators of x
- Domain of x is just an interval of floats

Adding Reals: Reals Are Not Floats



- Semantically, variables and constraints are over $\mathbb{R}$, not $\mathbb{F}$
- $x \in \mathbb{F}$ stand for a tiny real interval $[x - \epsilon, x + \epsilon]$
- Let C be a constraint, e.g., $3 \cdot x = 1.0$, and $\text{RANGE}(C)$ its variables, e.g., $\{x\}$
- An $\mathbb{R}$-solution of C is a tuple of numbers in $\mathbb{R}$
- An assignment to $\text{RANGE}(C)$ is a solution if its intervals contain at least one $\mathbb{R}$-solution of C
- Propagators endeavor to maintain bounds consistency
- Equality $(x = y)$ is *not* relaxed for rounding errors

Adding Reals: Constraints Or Expressions?

Expressions, mainly!

$$\textit{expr} \circ \textit{expr}$$

- $\mathbb{Z} : \circ \in \{$ `#< #=< #= #\= #> #>=` $\}$
- $\mathbb{R} : \circ \in \{$ `$=< $= $>=` $\}$ **N.B.** No $\{$ `$< $\= $>` $\}$
- Most existing functions become polymorphic: $x + y$, $x - y$, $x * y$, ... with compile-time type inference thanks to $\circ$
- Many new ($\mathbb{R} \rightarrow \mathbb{R}$) functions: SQRT($x$), EXP($x$), LOG($x$), SIN($x$), COS($x$), ...
- New rounding ($\mathbb{R} \rightarrow \mathbb{Z}$) functions: ROUND($x$), TRUNCATE($x$), FLOOR($x$), CEILING($x$).
- Channeling ($\mathbb{R} \leftrightarrow \mathbb{Z}$) functions: INTEGER($x$), FLOAT($y$).
- A few, selected constraints become polymorphic.

RI.
SE

Adding Reals: Search

- SICStus search predicate is reminiscent of MiniZinc `solve` annotation:

MiniZinc	**SICStus**
`seq_search`	`solve`
`bool_search`	`labeling`
`int_search`	`labeling`
`float_search`	`labeling`

- $\mathbb{R}$ and $\mathbb{Z}$ variables must be in separate `labeling` parts
- Ternary choice for $\mathbb{R}$-variables:
  1. try middle value
  2. try values less than middle
  3. try values greater than middle
- `labeling`([`precision`(p)], . . .) cuts the search if the domain sizes goes below p, similar to `float_search`.

Adding Reals: Examples

A small equation system over two variables that involves a trigonometric function:

```
| ?- domain([X,Y], 1.0, 2.0),
     tan(X) $= Y,
     X^2.0 + Y^2.0 $= 5.0,
     labeling([precision(1.0E-15)], [X,Y]).
X = 1.0966681287054714,
Y = 1.9486710896099533 ?
```

Exploring the set of solution to a high-degree equation:

```
| ?- X in 0.8..1.0,
     0.0 $= 35.0*X^256.0 -14.0*X^17.0 + X,
     labeling([precision(5.0E-16)], [X]).
X = 0.8479436608273154 ? ;
X = 0.995842494200498 ?
```

Lexicographic Optimization

SICStus

```
foo([X,Y,A,B]) :-
    table([[X,Y,A,B]], [[0,0,1,2],
                         [0,1,1,1],
                         [1,0,2,2],
                         [1,1,2,1]]),
    labeling([lex_minimize([A,B])], [X,Y]).

| ?- foo(L).
L = [0,1,1,1] ?
```

MiniZinc

```
solve :: lex_minimize([x,-y]) satisfy;
```

Pareto Optimization

SICStus

```
bar([X,A,B]) :-
    table([[X,A,B]], [[1,2,2],
                      [2,2,3],
                      [4,3,2],
                      [5,3,3],
                      [6,3,1],
                      [8,1,3]]),
    labeling([pareto_minimize([A,B])], [X]).

| ?- bar(L).
L = [8,1,3] ? ;
L = [6,3,1] ? ;
L = [1,2,2] ?
```

MiniZinc

```
solve :: pareto_minimize([x,-y]) satisfy;
```

How to Build A Solver That Is Robust And Fast

- It doesn't matter how fast your clever globals are if linear arithmetic and simple Booleans are too slow.
- It doesn't matter how fast your linear arithmetic and simple Booleans are if general propagation is too slow.
- Too much incrementality is bad for performance.
- Fuzz testing.
- The devil is in the details.

RI.
SE

Thank You