The Finite Domain Constraint Solver of SWI-Prolog

Markus Triska

Database and Artificial Intelligence Group
Vienna University of Technology
triska@dbai.tuwien.ac.at

Abstract. We present a new constraint solver over finite domains, freely available as `library(clpfd)`[1] in SWI-Prolog. Our solver has several unique features, which we describe in this paper: Reasoning over arbitrarily large integers, always terminating propagation, and a domain-specific language that concisely expresses the full semantics of constraint reification. The library is entirely written in Prolog and can be easily ported to other Prolog systems that support attributed variables. The constraint solver is fast enough for teaching and research purposes and is already being used in courses at several universities in France, Germany, Italy, Austria and other countries.

1 Introduction

CLP(FD), constraint logic programming over finite domains, is a declarative formalism for describing combinatorial problems such as scheduling, planning and allocation tasks ([1]). It is one of the most widely used instances of the general CLP($\cdot$) scheme that extends logic programming to reason over specialized domains. Since CLP(FD) is applied in many industrial settings like systems verification, it is natural to ask: How can we implement constraint solvers that are more reliable and more concise (and thus easier to read and verify)?

One frequent source of erroneous answers in common constraint systems are their – either implicit or explicit – restrictions to quite small values. A common limit is the magnitude of machine-size integers, and sometimes slightly below or above that. In some systems, such as GNU Prolog ([2]), a domain's limits additionally depend on properties of the domain itself, such as the presence of holes. Some constraint solvers, such as SICStus Prolog ([4]), use symbolic bounds to represent default domains and never explicitly mention (in top-level answers) the underlying finite limits that are the actual boundaries for all variables. The constraint solver of B-Prolog ([5]) avoids exposing implicit limits by showing symbolic residual constraints in answers when necessary. ECLiPSe ([3]) loses propagation strength and precision when domains exceed certain bounds.

Different constraint solvers do not behave consistently when their pre-defined finite limits are exceeded. For example, in GNU Prolog, the system's answer can

[1] Documentation: http://www.swi-prolog.org/man/clpfd.html

indicate when values *might* have been lost due to the system's limited domains, but – according to the manual – in some cases it cannot detect the loss of values and no message is emitted. Other systems, such as SICStus Prolog, raise a *representation error* when values outside the finite default domains arise. Neither behaviour is completely satisfying for users who are working with such values.

In this paper, we present a new finite domain constraint solver that overcomes this problem by allowing to reason over arbitrarily large integers, also called *bignums*. We also discuss how terminating propagation can be ensured in the presence of arbitrarily large domains, which is of theoretical interest in termination analysis of constraint logic programs, and also of practical importance for users. Finally, we present a domain-specific language (DSL) that concisely describes the full semantics of constraint reification in our solver.

2 Related work

Traditionally, finite domain constraint solvers have mostly been used to tackle problems that involve only quite small values. This is the case in many scheduling, allocation and combinatorial optimisation tasks for which constraint-based approaches are very well suited. A well-known benchmark library for constraints, CSPLib ([6]), consists almost exclusively of such examples. Therefore, the restricted default domains of existing constraint solvers may so far not have been perceived as very limiting.

On the other hand, the need for arbitrary precision integer arithmetic is widely recognised, and many common Prolog systems, such as SICStus ([4]), SWI ([7]) and YAP ([8]) provide transparent built-in support for arbitrarily large integers. This is of practical importance for several applications, such as encryption and authentication tasks. It is therefore surprising that no Prolog system has so far extended this support to its CLP(FD) solver as well.

The issue of inherent limits has so far not been given much attention. A notable exception is [9], where Apt and Zoeteweij remark for one of their examples: "the cost of using arbitrary length integers is roughly a factor four". However, they tested the impact of bignums only on a specialised hand-coded example.

Support for arbitrarily large values is taken for granted in solvers over rational numbers, such as Holzbaur's CLP(Q) implementation ([10]). CLP(Q) can be used to tackle some problems that can be solved with CLP(FD) solvers. In the presence of bignums, the converse also holds, since some constraint problems over rationals can be mapped to problems over integers by multiplying all values with a common multiple of all denominators.

In the context of CLP(FD), *indexicals* ([15]) are a well-known example of a DSL. The main idea of indexicals is to declaratively describe the domains of variables as functions of the domains of related variables. The indexical language consisting of the constraint "`in`" and expressions such as `min(X)..max(X)` also includes specialized constructs that make it applicable to describe a large variety of arithmetic and combinatorial constraints. GNU Prolog ([19]) and SICStus

Prolog ([18]) are well-known Prolog systems that use indexicals in the implementation of their finite domain constraint solvers.

The usefulness of deriving large portions of code automatically from shorter descriptions also motivates the use of *variable views*, a DSL to automatically derive *perfect* propagator variants, in the implementation of Gecode ([20]).

Action rules ([21]) and Constraint Handling Rules ([22]) are Turing-complete languages that are very well-suited for implementing constraint propagators and even entire constraint systems (for example, B-Prolog's finite domain solver).

3 Constraint solving over arbitrarily large integers

As an example where constraint solving over large integers is useful, consider the so-called "7-11 problem" ([11]), which is the following task: The total price of 4 items is \$ 7.11. The product of their prices is \$ 7.11 as well. What are the prices of the 4 items, and how many solutions are there? In [11], Pritchard and Gries employ elaborate problem-specific considerations to solve the problem efficiently. As a highly non-linear problem, it is beyond the current abilities of common CLP(Q) solvers. Even if it were possible to obtain a single solution with such solvers, they would typically not be able to enumerate *all* solutions. In contrast, the problem can be efficiently, completely and conveniently solved with CLP(FD) solvers that feature large enough integers. A CLP(FD) query for solving the problem is shown in Fig. 1. The product of all variables (line 2) equals a quite large constant, which exceeds the capabilities of most existing CLP(FD) solvers on still common 32-bit platforms. Given arbitrarily large integers, other interesting number-theoretic problems can also be expressed in CLP(FD).

```
1  ?- Vs = [A,B,C,D], Vs ins 0..711,
2     A * B * C * D #= 711*100^3,
3     A + B + C + D #= 711,
4     A #>= B, B #>= C, C #>= D,
5     labeling([ff], Vs).
```

Fig. 1. Solving the 7-11 problem with a single query

Perhaps most importantly, with arbitrarily large integers, CLP(FD) constraints can be used as a fully declarative alternative to conventional built-in arithmetic predicates over integers in all places. Consider for example the definition of **n_factorial**/2 and the sample interactions shown in Figure 2, which would lead to instantiation errors with built-in arithmetic (**is**/2 etc.).

4 Implementation

We implemented our constraint solver in Prolog, using attributed variables as described by Demoen ([12]). This interface is provided by (among others) SWI-Prolog and YAP, and our solver is freely available in the latest development

```
n_factorial(0, 1).
n_factorial(N, F) :- N #> 0,
        N1 #= N - 1, F #= N * F1,
        n_factorial(N1, F1).

?- n_factorial(47, F).
F = 258623241511168180642964355153611979969197632389120000000000 ;
false.

?- n_factorial(N, 1).
N = 0 ;
N = 1 ;
false.

?- n_factorial(N, 3).
false.
```

Fig. 2. CLP(FD) definition of **n_factorial**/2 and sample queries

versions of these systems as `library(clpfd)`. At the time of this writing, the library consists of about 6200 lines of Prolog code, including user documentation and comments.

Domains are represented as interval trees internally. The atoms **inf** and **sup** stand for infimum and supremum of the set of integers, and denote negative and positive infinity, respectively. This notation is borrowed from the constraint solver of SICStus Prolog, where these atoms are used to mask the underlying finite limits.

If all domains are finite, comparisons and computations can be delegated to the Prolog system's built-in support for big integers. For the infinite case, these operations must be generalised appropriately. We did this by implementing a Prolog predicate that behaves like **is**/2 and can also handle the symbolic domain boundaries. Similar predicates generalise the built-in comparisons.

We have implemented the common arithmetic constraints and equivalence relations (which are also reifiable), and some global constraints like **all_distinct**/1 (arc-consistent), **global_cardinality**/2 (arc-consistent), **circuit**/1 and **automaton**/3. The overhead of using generalised predicates for infinities is below 30% for many examples and could most likely be further reduced by implementing these predicates in C. This agrees with Apt and Zoeteweij ([9]), who estimate this overhead to be far less prominent in a full-fledged constraint solver than in isolated examples, and seems a small price for the new applications that this feature allows.

To transparently bring the performance of CLP(FD) constraints closer to that of conventional arithmetic predicates when the constraints are used in modes that can also be handled by built-in arithmetic, the library makes use of **goal_expansion**/2 to rewrite constraints at compilation time, automatically inserting code that dynamically checks whether built-in arithmetic predicates can be used directly.

5 Ensuring terminating propagation

Little attention has so far been given to termination properties of constraint solvers. This may again be due to the fact that most applications and benchmarks, and therefore most solvers, were so far focused on and limited to quite small domains. In a sense, all (sensible) propagations always terminate in the face of restricted domains, since either a fix-point, or an explicit or implicit domain-boundary is typically reached very quickly. But with the advent of 64-bit architectures and larger machine-size integers that become available in most solvers, "termination" in this sense can no longer be observed, leading to effectively non-terminating behaviour even for very simple queries that do not involve huge numbers. Fig. 3 shows three such examples. Line 2 shows that even enumeration predicates, which should ideally be complete, do not necessarily terminate in this sense. Line 3 shows a query that is clearly declaratively false, but this cannot be established if the first goal does not terminate.

```
1  ?- X #> abs(X).
2  ?- X #==> Y #> abs(Y), indomain(X).
3  ?- X #> abs(X), 0 #= 1.
```

Fig. 3. Effectively non-terminating queries with other solvers on 64-bit platforms

Universally terminating propagation is of theoretical interest in termination analysis of constraint logic programs (see for example [13]), and also of practical importance: Simple yet non-terminating queries are counter-intuitive to users, and – as shown above – may prevent a solver from detecting unsatisfiability, making it in effect weaker. These effects become especially prominent when arbitrarily large domains can occur.

We ensure terminating propagation by allowing the left and right boundaries, as well as the distance between the smallest and largest number occurring in a domain representation to be changed at most once after a constraint is posted, unless the domain is already finite. That this suffices to guarantee terminating propagation follows from how a non-terminating propagation chain can occur: Either the lower limit of some domain increases, or the upper limit of some domain decreases, or said distance of some domain increases without bound.

6 A domain-specific language for reification

Domain-specific languages (DSLs) are languages tailored to a specific application domain. DSLs are typically devised with the goal of increased expressiveness and ease of use compared to general-purpose programming languages in their domains of application ([16]). Examples of DSLs include *lex* and *yacc* ([17]) for lexical analysis and parsing, regular expressions for pattern matching, HTML for document mark-up, VHDL for electronic hardware descriptions and many other well-known instances. DSLs are also known as "*little languages*" ([14]), where "little" primarily refers to the typically limited intended or main practical application scope of the language. For example, PostScript is a "little language" for page descriptions.

In the context of CLP(FD) systems, DSLs were so far mainly used for the description and generation of constraint *propagation* code in practice. In this chapter, we contribute to these uses of DSLs by presenting a DSL that allows us to concisely express constraint *reification* with desirable declarative properties.

Reification means reflecting the truth values of (typically arithmetic) constraint relations into Boolean 0/1-variables. When implementing constraint reification, it is tempting to proceed as follows: For concreteness, consider reified equality (`#=/2`) of two CLP(FD) expressions E_1 and E_2. We could introduce two temporary variables, T_1 and T_2, and post the constraints T_1 `#=` E_1 and T_2 `#=` E_2, thus using the constraint solver itself to decompose the (possibly compound) expressions E_1 and E_2, and reducing reified equality of two *expressions* to equality of two finite domain *variables* (or integers), which is easier to implement. Unfortunately, this strategy yields wrong results in general. Consider for example the constraint (`#<==>/2` denotes Boolean equivalence):

```
(X/0 #= Y/0) #<==> B
```

It is clear that the relation `X/0 #= Y/0` cannot hold, since a divisor can never be 0. A valid (declaratively equivalent) answer to the above constraint is thus (note that X and Y must be constrained to integers for the relation to hold):

```
B = 0, X in inf..sup, Y in inf..sup
```

However, if we decompose the equality `X/0 #= Y/0` into two auxiliary constraints T_1 `#= X/0` and T_2 `#= Y/0` and post them, then (with strong enough propagation of division) both auxiliary constraints fail, and thus the whole query (incorrectly) fails. While devising a DSL for reification, we found one commercial Prolog system and one freely available system that indeed incorrectly failed in this case. After we reported the issue, the problem was immediately fixed.

To reflect the intended relational semantics, it is thus necessary to implement *definedness* correctly when reifying constraints. See also [23], where our constraint system (in contrast to others that were tested) correctly handles all reification test cases, which we attribute in part to the DSL presented in this chapter. Once any subexpression of a relation becomes undefined, the relation cannot hold and its associated truth value must be 0. Undefinedness can occur when $Y = 0$ in the expressions X/Y, $X \bmod Y$, and $X \text{ rem } Y$. Parsing an arithmetic expression that occurs as an argument of a constraint that is being reified is thus at least a ternary relation, involving the expression itself, its arithmetic result, and its Boolean definedness.

There is a fourth desirable component in addition to those just mentioned: It is useful to keep track of *auxiliary variables* that are introduced when decomposing subexpressions of a constraint that is being reified. The reason for this is that the truth value of a reified constraint may turn out to be irrelevant, for instance the implication `0 #==>` C holds for both possible truth values of the constraint C, thus auxiliary variables that were introduced to hold the results of subexpressions while parsing C can be eliminated. However, we need to be careful: A constraint propagator may *alias* user-specified variables with auxiliary variables. For example, in `abs(X) #= T, X #>= 0`, a constraint system may deduce `X = T`. Thus, if T was previously introduced as an auxiliary variable, and X was user-specified, X must still retain its status as a constrained variable.

These considerations motivate the following DSL for parsing arithmetic expressions in reified constraints, which we believe can be useful in other constraint systems as well: A parsing rule is of the form $H \to Bs$. A head H is either a term $g(G)$, meaning that the Prolog goal G is true, or a term $m(P)$, where P is a symbolic pattern and means that the expression E that is to be parsed can be decomposed as stated, recursively using the parsing rules themselves for subterms of E that are subsumed by variables in P. The body Bs of a parsing rule is a list of body elements, which are described in Table 1. The predicate `parse_reified/4`, shown in Figure 4, contains our full declarative specification for parsing arithmetic expressions in reified constraints, relating an arithmetic expression E to its result R, Boolean definedness D, and auxiliary variables according to the given parsing rules, which are applied in the order specified, committing to the first rule whose head matches. This specification is translated to Prolog code at compile time and used in other predicates.

`g(G)`	Call the Prolog goal G.
`d(D)`	D is 1 if and only if all subexpressions of E are defined.
`p(P)`	Add the constraint propagator P to the constraint store.
`a(A)`	A is an auxiliary variable that was introduced while parsing the given compound expression E.
`a(X,A)`	A is an auxiliary variable, unless `A == X`.
`a(X,Y,A)`	A is an auxiliary variable, unless `A == X` or `A == Y`.
`skeleton(Y,D,G)`	A "skeleton" propagator is posted. When Y cannot become 0 any more, it calls the Prolog goal G and binds $D = 1$. When Y is 0, it binds $D = 0$. When $D = 1$ (i.e., the constraint must hold), it posts `Y #\= 0`.

Table 1. Valid body elements for a parsing rule

Figure 4 declaratively expresses the intended semantics of reification in a very concise way that is also quite easy to read and reason about. The implementations of individual propagators like `pplus/3` and `pabs/2` are beyond the scope of the DSL introduced in this chapter, and for example indexicals can be used to describe their semantics.

The deletion of auxiliary variables and constraints when they are no longer necessary is useful when introducing constraint programming to beginners (since shorter answers of the system are easier to grasp), and often also for efficiency

```
parse_reified(E, R, D,
    [g(cyclic_term(E)) => [g(domain_error(clpfd_expression, E))],
     g(var(E))         => [g((constrain_to_integer(E), R=E, D=1))],
     g(integer(E))     => [g((R=E, D=1))],
     m(-X)             => [d(D), p(ptimes(-1,X,R)), a(R)],
     m(abs(X))         => [g(R#>=0), d(D), p(pabs(X, R)), a(X,R)],
     m(X+Y)            => [d(D), p(pplus(X,Y,R)), a(X,Y,R)],
     m(X-Y)            => [d(D), p(pplus(R,Y,X)), a(X,Y,R)],
     m(X*Y)            => [d(D), p(ptimes(X,Y,R)), a(X,Y,R)],
     m(X^Y)            => [d(D), p(pexp(X,Y,R)), a(X,Y,R)],
     m(min(X,Y))       => [d(D), p(pgeq(X, R)), p(pgeq(Y, R)),
                           p(pmin(X,Y,R)), a(X,Y,R)],
     m(max(X,Y))       => [d(D), p(pgeq(R, X)), p(pgeq(R, Y)),
                           p(pmax(X,Y,R)), a(X,Y,R)],
     m(X/Y)            => [skeleton(Y,D,X/Y #= R)],
     m(X mod Y)        => [skeleton(Y,D,X mod Y #= R)],
     m(X rem Y)        => [skeleton(Y,D,X rem Y #= R)],
     g(true)           => [g(domain_error(clpfd_expression, E))]]).
```

Fig. 4. Parsing arithmetic expressions in reified constraints with our DSL

reasons (since irrelevant constraints need no longer be considered). As an example, consider the query and its result:

```
?- X #= 3 #\/ Y #= 4 #<==> B, Y = 4.
Y = 4,
B = 1,
X in inf..sup.
```

Other constraint systems, such as SICStus, still retain and show an additional arithmetic constraint on the variable X in the case above although it is no longer semantically relevant. Removal of irrelevant constraints can also significantly improve performance on some benchmarks. As an example of a benchmark that uses reification extensively, we took a solution to the so-called "Nonogram"-puzzle that was generously posted to `comp.lang.prolog` by Bart Demoen on Jan. 22nd 2009. By dynamically removing constraints that are no longer semantically relevant, both run-time and inference count decrease by more than 30% in this case.

To the best of our knowledge, our constraint system is the first one to describe the full declarative semantics of reification in such brevity. While indexicals can be (and are) used to describe reification of individual atomic constraints, they cannot express when auxiliary constraints and variables that were introduced when decomposing nested expressions are no longer needed, in contrast to the DSL we propose in this chapter.

7 Extensions via custom propagators

Our constraint solvers does not yet provide a custom language to add user-defined constraints as other systems do. Instead, users can add new propagators directly in Prolog by following a few simple conventions that are explained in the library's documentation. Some users have already implemented custom propagators for specialized (for example: geometric) constraints in this way with satisfactory results.

8 Performance evaluation

Neng-Fa Zhou, the author of B-Prolog, has kindly integrated our constraint solver in his benchmarks, available from *http://www.probp.com/performance.htm*. The results show that our solver is on average two orders of magnitude slower on these benchmarks than the fastest system (B-Prolog itself), and about 30 times slower than the constraint solver of SICStus Prolog.

In part, this may certainly be attributed to the fact that SWI-Prolog itself (i.e., the system without the finite domain constraint solver) is already more than 4 times slower than B-Prolog (and more than 3 times than SICStus) on average on benchmarks that are deemed to be in some sense representative of a Prolog system's performance, and which are also available on the website. Our library is written in Prolog and is thus heavily influenced by the speed of the underlying Prolog system itself. We found that the same benchmarks are already faster with YAP by more than a factor of 2 on average.

On the other hand, if large integers are needed, our solver is the only option of all tested systems and can not be compared to any others at all in this case.

While the comparatively slow speed of our constraint solver will certainly rule out its usage in many industrial settings, it is already being used at several universities in France, Germany, Italy, Austria and other countries for teaching and research purposes and has so far shown more than acceptable performance for these use cases.

9 Conclusion and future work

We have presented a new finite domain constraint solver, freely available as `library(clpfd)` in SWI-Prolog and YAP. The library supports constraint solving over arbitrarily large integers, always terminating propagation, and uses a new DSL to concisely express constraint reification.

Future work includes correctness considerations and performance improvements for individual propagators. To this end, we may partially replace the current Prolog implementation of propagators by a more declarative description of constraints, such as indexicals ([2],[4]) or a language with similar desirable properties.

10 Acknowledgments

I thank Ulrich Neumerkel for discussions, and Nysret Musliu for feedback. I thank Tom Schrijvers for generously contributing several constraint libraries to SWI-Prolog; I learned a lot from Tom's coding style, and he also kindly answered several questions I had while working on early prototypes of `library(clpfd)`.

Above all, I thank Lisa Marie and Maggy Zitz for giving me the motivation and energy to finish this paper, and their great example of working, writing and living.

References

1. Jaffar, J., Lassez, J-L.: Constraint Logic Programming. POPL, 111–119 (1987)
2. D. Diaz and P. Codognet, Design and Implementation of the GNU Prolog System, Journal of Functional and Logic Programming 6 (2001)
3. Mark Wallace and Stefano Novello and Joachim Schimpf, ECLiPSe: A Platform for Constraint Logic Programming, Technical Report (1997)
4. M. Carlsson and G. Ottosson and B. Carlson, An Open-Ended Finite Domain Constraint Solver, LNCS 1292 (1997)
5. N.F. Zhou and I. Nagasawa, An Efficient Finite-domain Constraint Solver in Beta-Prolog , Journal of Japanese Society for Artificial Intelligence 9 (1994)
6. I.P. Gent and T. Walsh, CSPLib: A Benchmark Library for Constraints, Proceedings of the 5th Int. Conf. PPCP (1999)
7. Jan Wielemaker, An Overview of the SWI-Prolog Programming Environment, Proceedings of the 13th International Workshop on LP Environments (2003)
8. Anderson Faustino da Silva and Vítor Santos Costa, The Design and Implementation of the YAP Compiler, LNCS 4079 (2006)
9. Krzysztof R. Apt and Peter Zoeteweij, An Analysis of Arithmetic Constraints on Integer Intervals, Constraints 4 (2007)
10. Christian Holzbaur, OFAI CLP(Q,R) Manual, TR (1995)
11. Paul Pritchard and David Gries, The Seven-Eleven Problem, TR (1994)
12. Bart Demoen, Dynamic attributes, their hProlog implementation, and a first evaluation, Technical Report (2002)
13. cTI: Bottom-Up Termination Inference for Logic Programs, Serge Burckel and Sébastien Hoarau and Frédéric Mesnard and Ulrich Neumerkel, WLP 15 (2000)
14. Bentley, J.: Little languages. Communications of the ACM, 29(8), 711–21 (1986)
15. Codognet, P., Diaz, D.: Compiling Constraints in clp(FD). Journal of Logic Programming, 27(3) (1996)
16. Mernik, M., Heering, J., Sloane, A. M.: When and how to develop domain-specific languages. ACM Comput. Surv., 37(4), 316–344 (2005)
17. Johnson, S. C., Lesk, M. E.: Language development tools. Bell System Technical Journal, 56(6), 2155–2176 (1987)
18. Carlsson, M., Ottosson, G., Carlson, B.: An Open-Ended Finite Domain Constraint Solver. Proc. Prog. Lang.: Implementations, Logics, and Programs (1997)
19. Diaz, D., Codognet, P.: Design and Implementation of the GNU Prolog System. Journal of Functional and Logic Programming (JFLP), Vol. 2001, No. 6 (2001)
20. Schulte, Ch., Tack, G.: Perfect Derived Propagators. CoRR entry (2008)
21. Zhou, N-F.: Programming Finite-Domain Constraint Propagators in Action Rules. Theory and Practice of Logic Programming, Vol.6, No.5, pp. 483–508 (2006)
22. Frühwirth, T.: Theory and Practice of Constraint Handling Rules. Special Issue on Constraint Logic Programming, J. of Logic Programming, Vol 37(1–3) (1998)
23. Frisch, Alan M., Stuckey, Peter J.: The Proper Treatment of Undefinedness in Constraint Languages, CP 2009, LNCS 5732, 367–382 (2009)