Extending the Finite Domain Solver of GNU Prolog

Vincent Bloemen[1], Daniel Diaz[2], Machiel van der Bijl[3], and Salvador Abreu[4]

[1] University of Twente, Formal Methods and Tools group, The Netherlands
`v.bloemen@student.utwente.nl`
[2] University of Paris 1-Sorbonne - CRI, France
`daniel.diaz@univ-paris1.fr`
[3] Axini B.V., The Netherlands
`vdbijl@axini.com`
[4] Universidade de Évora and CENTRIA, Portugal
`spa@di.uevora.pt`

Abstract. This paper describes three significant extensions for the Finite Domain solver of GNU Prolog. First, the solver now supports negative integers. Second, the solver detects and prevents integer overflows from occurring. Third, the internal representation of sparse domains has been redesigned to overcome its current limitations. The preliminary performance evaluation shows a limited slowdown factor with respect to the initial solver. This factor is widely counterbalanced by the new possibilities and the robustness of the solver. Furthermore these results are preliminary and we propose some directions to limit this overhead.

1 Introduction

Constraint Programming [1,7,16] emerged, in the late 1980s, as a successful paradigm with which to tackle complex combinatorial problems in a declarative manner [11]. However, the internals of constraint solvers, particularly those over Finite Domains (FD) were wrapped in secrecy, only accessible to only a few highly specialized engineers. From the user point of view, a constraint solver was an opaque "black-box" providing a fixed set of (hopefully) highly optimized constraints for which it ensures the consistency.

One major advancement in the development of constraint solvers over FD is, without any doubt, the article from Van Hentenryck et al. [12]. This paper proposed a "glass-box" approach based on a single *primitive constraint* whose understanding is immediate. This was a real breakthrough with respect to the previous way of thinking about solvers. This primitive takes the form X `in` r, where X is an FD variable and r denotes a *range* (i.e. a set of values). An X `in` r constraint enforces X to belong to the range denoted by r *which can involve other FD variables*. An X `in` r constraint which depends on another variable Y becomes *store sensitive* and must be (re)activated each time Y is updated, to ensure the consistency. The X `in` r constraint can be seen as embedding the core propagation mechanism. Indeed, it is possible to define different propagation schemes for a given constraint, corresponding to different degrees of consistency.

It possible to define high-level constraints, such as equations or inequations, in terms of X `in` r primitive constraints. It is worth noticing that these constraints are therefore not built into the theory. From the theoretical point of view, it is only necessary to work at the primitive level as there is no need to give special treatment to high-level constraints. This approach yielded significant advances in solvers. From the implementation point of view, an important article is [5] which proposed a complete machinery (data structures, compiler, instruction set) to efficiently implement an FD solver based on X `in` r primitives. It also shows how some optimizations at the primitive level can, in turn, be of great benefit to all high-level constraints. The resulting system, called `clp(FD)`, proved the efficiency of the approach: the system was faster than CHIP, a highly optimized black-box solver which was a reference at the time. This work has clearly inspired most modern FD solvers (SICStus Prolog, bProlog, SWI Prolog's clpfd, Choco, Gecode, ...) but also solvers over other domains like booleans [4], intervals [10] or sets [9,2]. Returning to FD constraints, a key point is the ability to reason on the outcome of a constraint (success or failure). Again, the "RISC" approach restricted the theoretical work about *entailment* at the primitive level [3]. This allowed a new kind of constraints: *reified constraints*, in which a constraint becomes concretized. The "RISC" approach was also very convenient for *constraint retraction* [6].

When GNU Prolog was developed, it reused the FD solver from `clp(FD)`. The X `in` r constraint was generalized to allow the definition of new high-level constraints, e.g. other arithmetic, symbolic, reified and global constraints. Nevertheless, the internals of the solver were kept largely unchanged. The outcome is a fast FD solver, but also one with some limitations:

– First, the domain of FD variables is restricted to positive values (following the original paper [12]). This is not restrictive from a theoretical point of view: a problem can always be "translated" to natural numbers but, from a practical point of view, there are several drawbacks: the translation is error-prone, the resulting programs are difficult to read and can exhibit significant performance degradation.
– Second, the domain representation uses a fixed-size bit-vector to encode sparse domains. Even if this size can be controlled by the user, it is easily and frequently misused. In some cases, the user selects a very large value for simplicity, without being aware of the waste of memory nor the loss of efficiency this induces.
– Lastly, for efficiency reasons the GNU Prolog solver does not check for integer overflows. This is generally not a problem when the domains of FD variables are correctly specified, as all subsequent computation will not produce any overflow. However, if one forgets to declare all domains, or does not declare them at the beginning, an overflow can occur. This is the case of:

```
|?- X * Y #= Z.
No
```

Indeed, without any domain definition for X and Y the non-linear constraint `X * Y #= Z` will compute the upper bound of Z as $2^{28} \times 2^{28}$ which overflows 32 bits resulting in a negative value for the *max*, thus the failure (*max* < *min*). This behaviour is hard to understand and requires an explanation. We admit this is not the *Way of Prolog* and does not help to promote constraint programming to new users. We could raise an exception (e.g. `instantiation_error` or `representation_error`) but this would still be of little help to most users.

In this article we describe and report on initial results for the extension and modification of the GNU Prolog FD solver to overcome these three limitations. This is a preliminary work: special attention has been put on ensuring correctness and the implementation is not yet optimized. Nevertheless the results are encouraging, as we shall see, and there is ample room and directions to research on performance improvements.

The remainder of this article is organized as follows: Section 2 introduces some important aspects of the original FD solver required to understand the modifications. Section 3 is devoted to the inclusion of negative values in FD domains. Section 4 explains how integer overflow is handled in the new solver, while Section 5 explains the new representation for sparse domains. A performance evaluation may be found in Section 6. Section 7 provides some interesting directions to optimize the overall performance. A short conclusion ends the paper.

2 The GNU Prolog FD Solver

The GNU Prolog solver follows the "glass-box" approach introduced by Van Hentenryck et al. in [12], in which the authors propose the use of a single *primitive constraint* of the form X `in` r, where X is an FD variable and r denotes a *range* (ie. a set of values). An X `in` r constraint enforces X to belong to the range denoted by r which can be constant (e.g. the interval 1..10) but can also use the following *indexicals*:

– `dom(`Y`)` representing the whole current domain of Y.
– `min(`Y`)` representing the minimum value of the current domain of Y.
– `max(`Y`)` representing the maximum value of the current domain of Y.
– `val(`Y`)` representing the final value of the variable of Y (when its domain is reduced to a singleton). A constraint using this indexical is postponed until Y is instantiated.

An X `in` r constraint which uses an indexical on another variable Y becomes *store sensitive* and must be (re)activated each time Y is updated to ensure the consistency. Thanks to X `in` r primitive constraints it is possible to define high-level constraints such as equations or inequations. Obviously all solvers offer a wide variety of predefined (high-level) constraints to the programmer. Nevertheless, the experienced user can define his own constraints if needed.

The original FD solver of GNU Prolog is also based on indexicals. Its implementation is widely based on its predecessor, `clp(FD)` [5]. In the rest of this section we only describe some aspects of the original implementation which are important later on. The interested reader can refer to [8] for missing details.

2.1 The FD definition language

The original X `in` r is not expressive enough to define all needed constraints in practice. We thus defined the FD language: a specific language to define the constraints of the GNU Prolog solver. Figure 1 shows the definition of the constraint $A \times X = Y$ in the FD language:

```
ax_eq_y(int A, fdv X, fdv Y)                 /* here A != 0 */
{
  start X in min(Y) /> A .. max(Y) /< A       /* X = Y / A */
  start Y in min(X) *  A .. max(X) *  A       /* Y = X * A */
}
```

Fig. 1. Definition of the constraint $A \times X = Y$ in the FD language

The first line defines the constraint name (`ax_eq_y`) and its arguments together with their types (`A` is expected to be an integer, `X` and `Y` FD variables). The `start` instruction installs and activates an X `in` r primitive. The first primitive computes X from Y in the following way: each time a bound of Y is modified the primitive is triggered to reduce the domain of X accordingly. The operator `/>` (resp. `/<`) denote division rounded upwards (resp. downwards). Similarly, the second primitive updates (the bounds) of Y with repect to X. This is called *bound consistency* [1] : if a *hole* appears inside the domain of X (i.e. a value V different from both the min and the max of X has beed removed from the domain of X), the corresponding value $A \times V$ will not be removed from the domain of Y. If wanted, such a propagation (called *domain consistency*) could be specified using the `dom` indexical.

A compiler (called `fd2c`) translates an FD file to a C source file. The use of the C language as target is motivated by the fact that all the GNU Prolog system is written in C (so the integration is simple) but mainly by the fact that modern C compilers produce very optimized code (this is of prime importance if we consider that a primitive constraint can be awoken several thousand times in a resolution). When compiled such a definition gives rise to different C functions:

- the *main function*: a public function (`ax_eq_y`) which mainly creates an environment composed of the 3 arguments (A, X, Y) and invokes the installation functions for the involved X `in` r primitives.
- the *installation function*: a private function for each X `in` r primitive which is responsible for the installation of the primitive. This consists of installing

the dependencies (e.g. add a new dependency to Y, so that each time Y is modified the primitive is re executed to update X) and the execution function is invoked (this is the very first execution of the primitive).
- the *execution function*: a private function for each X `in` r primitive which computes the actual value of r and enforces $X \in r$. This function will be (re)executed each time an FD variable appearing in the definition of r is updated.

2.2 Internal domain representations

There are 2 main representations of a domain (range):

- *MinMax*: only the *min* and the *max* are stored. This representation is used for intervals (including `0..fd_max_integer`).
- *Sparse*: this representation is used as soon as a hole appears in the domain of the variable. In that case, in addition to the *min* and the *max*, a bit-vector is used to record each value of the range.

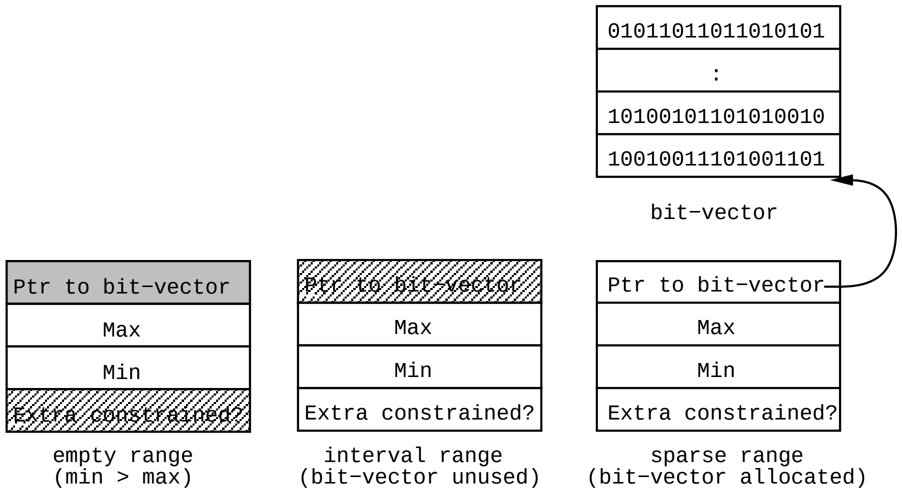

Fig. 2. Representations of a range

When an FD variable is created it uses a *MinMax* representation. As soon as a “hole” appears it is transparently switched to a *Sparse* representation which uses a bit-vector. For efficiency reasons all bit-vector have the same size inside `0..fd_vector_max`. By default `fd_vector_max` equals 127 and can be redefined via an environment variable or via a built-in predicate (this should be done before any constraint is told). When a range becomes *Sparse*, some values are possibly lost if `fd_vector_max` is less than the current *max* of the variable. To inform the user of this source of incompleteness, GNU Prolog maintains a flag to indicate that a range has been *extra constrained* by the solver (via an imaginary constraint $X\ in\ 0..$`fd_vector_max`). The flag *extra_cstr* associated to each range is updated by all operations, e.g. the intersection of two ranges is extra-constrained iff both ranges are extra constrained, thus the resulting flag is the logical *and* between the two flags. When a failure occurs on a variable whose domain is extra constrained a message is displayed to inform the user that some solutions can be lost since bit-vectors are too small. Finally an empty range is represented with $min\ >\ max$. This makes it possible to perform an intersection between R_1 and R_2 in *MinMax* mode simply with $Max(min(R_1), min(R_2))..Min(max(R_1), max(R_2))$ which returns $min > max$ if either R_1 or R_2 is empty. Figure 2 shows the different representations of a range.

3 Supporting Negative Values

In this section we describe how the inclusion of negative values in FD variables is realized. First we show why the current implementation does not support negative values. Then we show how to address the problems by mainly focusing on the implementation. This section only describes bound consistency; negative values are handled similarly in domain consistency due to the new sparse design, described in Section 5.

3.1 Current limitations

The current implementation does not support negative values, FD variables stay within the bounds `0..fd_max_integer`. Adding support for negative values seems obvious at a first glance, however some attention has to be paid. The modifications concern constraints whose current implementation implicitly utilize the fact that values are always positive, which is no longer valid. Other modifications concern constraints which are sign sensitive from the interval arithmetical point of view. This is the case for multiplication: if X is in $min..max$ then $-X$ is in $-max.. - min$. Let us consider again the case of the constraint $A \times X = Y$ whose current definition is presented in Figure 1. Presuming that A can be negative the current definition will not update the domains of X and Y correctly: in that case X will be constrained to $\lceil \frac{min(Y)}{A} \rceil..\lfloor \frac{max(Y)}{A} \rfloor$ which produces an empty interval since $min(X) > max(X)$. To support negative values in FD variables, this instance, as well as other arithmetical constraints require updating to handle negative values properly.

3.2 Method and approach

One possible approach to deal with negative numbers is to construct a mapping for negative values to natural numbers so that the arithmetic constraints can continue to operate strictly on the positive domain. Another approach is to update the constraints to be fully functional for both positive and negative domains. The former is undesirable since the translation quickly becomes cumbersome and would carry a considerable performance impact. Aside from that, several operations such as taking the power or root are affected by the variable sign. As the latter approach is less error-prone and more robust, we chose to implement it and thus need to reformulate several arithmetic constraints.

First, the initial domain bounds of FD variables are updated to range in `fd_min_integer..fd_max_integer`. To remain backwards compatible, an environment variable is created that, if set, will use the original bounds for FD variables.

On updating the arithmetic constraints, all possible cases for each FD variable need to be considered, that is < 0, $= 0$ and > 0 for both the *min* and *max* of the variable. For instance, the $A \times X = Y$ constraint from Figure 1 is updated as follows:

```
ax_eq_y(int A, fdv X, fdv Y)                /* A != 0 */
{
 start X in ite(A>0, min(Y), max(Y)) /> A    /* X = Y / A */
         .. ite(A>0, max(Y), min(Y)) /< A
 start Y in ite(A>0, min(X), max(X)) * A     /* Y = X * A */
         .. ite(A>0, max(X), min(X)) * A
}
```

where `ite` represents an if-then-else expression (corresponding to the C operator `?:`). This modification ensures that for all interpretations of A, X and Y the domains are updated correctly.

A more complex example is the constraint $X^A = Y$, where X and Y are FD variables and A is an integer > 2. In the current version, this constraint is given as follows:

```
x_power_a_eq_y(fdv X, int A, fdv Y)         /* A > 2 */
{
 start Y in Power(min(X), A)..Power(max(X), A)
 start X in Nth_Root_Up(min(Y), A)..Nth_Root_Dn(max(Y), A)
}
```

With the introduction of negative values, the constraint is specified as:

```
x_power_a_eq_y(fdv X, int A, fdv Y)         /* A > 2 */
{
 start X in ite(is_even(A),
                min_root(min(X), min(Y), max(Y), A),
                ite(min(Y) < 0,
                    -Nth_Root_Dn(-min(Y), A),
                    Nth_Root_Up(min(Y), A)))
         .. ite(is_even(A),
                max_root(max(X), min(Y), max(Y), A),
                ite(max(Y) < 0,
                    -Nth_Root_Up(-max(Y), A),
                    Nth_Root_Dn(max(Y), A)))

 start Y in ite(min(X) < 0 && is_odd(A),
                Power(min(X), A),
                Power(closest_to_zero(min(X), max(X)), A))
         .. ite(min(X) < 0 && is_even(A),
                Power(Max(abs(min(X)), max(X)), A),
                Power(max(X), A))
}
```

here, a couple of C functions and macros are introduced:

- `Min` and `Max` are used to compute the minimum resp. maximum of two values.
- `is_even` and `is_odd` return wether the variable is even or odd.
- `min_root` and `max_root` calculate the minimum and maximum value of $\pm\sqrt[A]{Y}$ that lie in the bounds of `min(X)..max(X)`.
- `Power` and `Nth_Root` refer to C functions that calculate the n^{th} power and n^{th} root of a variable.
- `closest_to_zero(A,B)` returns the closest value to 0 in the interval `A..B`.

In this specification, Y can only include negative values if X contains negative values and A is an odd number (e.g. $-2^3 = -8$). Similarly, if Y is strictly positive, X can only take negative values if A is an even number (e.g. $-2^4 = 16$). In short, the above constraint needs to distinguish between even and odd powers of X, which was originally unnecessary. With this definition, the following query correctly reduces the domains of X and Y:

```
|?- fd_domain([X,Y],-50,150), X ** 3 #= Y.
X = _#0(-3..5)
Y = _#17(-27..125)
```

The support for negative values in FD variables is achieved by carefully redesigning the arithmetic constraints. An obvious side-effect of the modifications is that some overhead is introduced, even when considering strictly positive FD variables. The benchmark tests, see Section 6, will show the impact of the modifications compared to the original solver.

4 Handling Integer Overflows

4.1 Current limitations

The current implementation of GNU Prolog does not check for overflows. This means that without preliminary domain definitions for X, Y and Z, the non-linear constraint $X \times Y = Z$ will fail due to an overflow when computing the upper bound of the domain of $Z : 2^{28} \times 2^{28}$. In 32-bit arithmetic, this overflow causes a negative result for the upper bound and the constraint then fails since $min(X) > max(X)$.

At present, the user needs to adapt the variable bounds beforehand to prevent this constraint from failing. To reduce the burden to the user and improve the robustness of the solver, we propose a better way of handling overflows.

4.2 Method and approach

There are two approaches to handle overflows. One is to report the problem via an ISO exception (e.g. `evaluation_error`), thereby informing the user that the domain definitions for the FD variables are too mild and should be made more restrictive. The other approach is to instrument the solver to detect overflows and cap the result. As placing less restrictions on the user and more robustness for the solver is desirable, the second approach is chosen.

The key idea behind preventing overflows is to detect when one would occur and provide means to restrict this from happening. For the solver this means that when a multiplication or power operation is applied in a constraint, an overflow prevention check should be considered. This can also be the case for other arithmetic operations.

Consider again the constraint $X \times Y = Z$. Because both $1 \times 2^{28} = 2^{28}$ and $2^{28} \times 1 = 2^{28}$, the maximum value that both X and Y can take is 2^{28}. Therefore the following (and current implementation) for finding the domain for Z causes an overflow:

```
start Z in min(X) * min(Y) .. max(X) * max(Y)
```

For this case and similar instances, the following function is designed to cap results of arithmetic, thereby preventing overflows:

```
static int inline mult(int a, int b)
{
  int64_t res = ((int64_t) a) * ((int64_t) b);
  if (res > max_integer)
    res = max_integer;
  else if (res < min_integer)
    res = min_integer;
  return (int) res;
}
```

Since integers only need 29-bits, the 64-bit result is enough to check if an overflow occurs and cap the result if needed. In the constraint definitions, the standard multiplication gets replaced with a `mult` call when it could cause an overflow. For the $X \times Y = Z$ constraint, this is as follows:[1]

```
start Z in mult(min(X), min(Y)) .. mult(max(X), max(Y))
```

[1] The constraint is further modified for negative values, along the same lines.

As a consequence, the $X \times Y = Z$ constraint now gives the following result:

```
| ?- X * Y #= Z.
X = _#3(-268435456..268435455)
Y = _#20(-268435456..268435455)
Z = _#37(-268435456..268435455)
```

where `-268435456 = fd_min_integer` and `268435455 = fd_max_integer`.

At first, we used `mult` for every applied multiplication in the constraint definitions. However, in some cases it is not necessary to check for overflows. For instance, consider the implementations for `ax_eq_y` and `x_power_a_eq_y` of Section 3.2. By first restricting the domain of X (in both cases), no overflow can occur when the domain of Y is calculated. Note that if the domain of Y is computed first, an overflow could happen. Note however, that such an optimization is not possible for some constraints, for instance $X \times Y = Z$, since the domains of X and Y do not necessarily get reduced.

In conclusion, even if several overflow problems could be resolved by rearranging the order of execution, in general it is necessary to take preventive measures.

5 New Domain Representation

5.1 Current limitations

In the current implementation, when a domain gets a *hole*, its representation is switched to the *Sparse* form, which stores domains using a static-sized bit-vector. The problem with this approach is that values which lie outside the range `0..fd_vector_max` are lost. An internal flag *extra_cstr* is set when this occurs to inform the user of lost values. Even though the user is able to globally set `fd_vector_max`, there are several problems with this representation:

- The user has to know the variable bounds in advance; an over-estimate of the domain size results in a waste of memory (and loss of efficiency).
- There is an upper-limit for `fd_vector_max` which is directly related to the available memory space in bits. Also note that doing operations on a large bit-vector can severely impact the performance.
- The current *Sparse* representation is unable to store negative values.

5.2 Method and approach

To deal with the limitations, a redesign is needed for the *Sparse* representation. Some research has been done in representing sparse domains [13,14]. Considering the requirements – remain efficient while taking away the limitations – there are several options for the redesign, while also considering alternatives and variations:

1. Use a list of *MinMax* chunks: Store only the minimum and maximum of consecutively set values. The values between two chunks are defined to be all unset. This is especially effective if the number of holes is small or large gaps exist in the domain.
2. Use a list of bit-vector chunks: Use a bit-vector together with an offset to store all (un)set actual values. The values between two chunks can either be defined as all set or all unset (possibly defined per chunk with a variable). This is in particular effective on small domains with many holes.
3. A combination of (1) and (2): Determine per chunk whether it should be a *MinMax* chunk or bit-vector chunk, so that the number of total chunks is minimal. This takes the advantages of both individual options but it does introduce extra overhead for determining which representation to choose and operations between two different chunk representations can become difficult.

Note that all suggested model takes away the limitations of the current design. Le Clément et al. [13] provide a more in-depth analysis on the different representations with respect to their time complexities. Note that differences arise for specific operations on domains: for instance, a value removal is done more efficiently in a bit-vector while iteration is more efficient on *MinMax* chunks.

We initially opted for the combination of the *MinMax* and bit-vector chunks because the extra overhead is presumed to not be a significant factor. For the moment, however, we implemented a list of *MinMax* chunks. Its performance compared to the original *Sparse* implementation shows a limited slowdown factor, as discussed in Section 6. Because of these results (a slowdown is expected anyway, due to the new possibilities), the addition of a bit-vector representation was postponed. We now discuss the new implementation using a list of *MinMax* chunks.

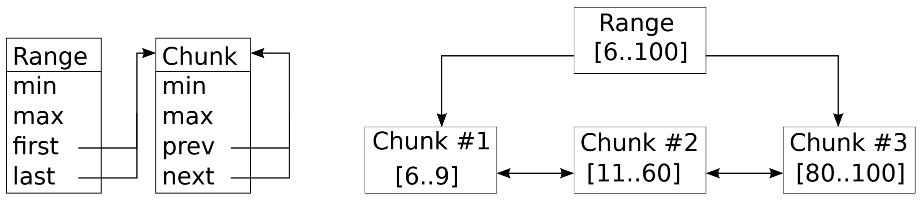

Fig. 3. Left: UML diagram of the new *Sparse* range, right: example for representing the set of values {`6..9,11..60,80..100`}.

The domain initially uses a *MinMax* representation (just a Range instance) which only stores min and max, with first and last being null pointers. When a hole appears, the domain switches to the *Sparse* representation by adding Chunk instances. The range keeps track of the first and last chunk of the list and continues to maintain the min and max of the whole domain (`range.min = range.first.min`). The list is a doubly-linked list for efficient insertion and removal of chunks, each chunk maintains its minimum and maximum values. This representation is depicted in Figure 3.

For every two consecutive chunks c_1 and c_2, we have $c_1.\mathsf{max} + 1 < c_2.\mathsf{min}$; chunks are sorted and always have at least one unset value between them. Furthermore, $c_i.\mathsf{min} \leq c_i.\mathsf{max}$.

Operations on *Sparse* ranges (e.g. intersection, union, ...) are efficiently done by iterating over the chunks and updating these in place whenever possible. An example of this is provided in Table 1 for intersecting two *Sparse* ranges. The implementation only considers one chunk of each range at a time and the cases are considered from top to bottom.

Case:	Action (in pseudo code):
`chunk_1.max` < `chunk_2.min`	- Remove `chunk_1` - `chunk_1` = `chunk_1.next` // advance `chunk_1`
`chunk_1.max` ≤ `chunk_2.max`	- Create new chunk and set before `chunk_1` with `min` = Max(`chunk_1.min, chunk_2.min`) and `max` = Min(`chunk_1.max, chunk_2.max`) - `chunk_1` = `chunk_1.next` // advance `chunk_1`
`chunk_1.min` > `chunk_2.max`	- `chunk_2` = `chunk_2.next` // advance `chunk_2`
`chunk_1.max` > `chunk_2.max`	- Create new chunk and set before `chunk_1` with `min` = Max(`chunk_1.min, chunk_2.min`) and `max` = Min(`chunk_1.max, chunk_2.max`) - `chunk_2` = `chunk_2.next` // advance `chunk_2`

Table 1. Implementation of the range intersection operation.

Because the solver may need to backtrack, domains need to be trailed. Modifications on domains can cause its chunks to disperse in memory, therefore all chunks of the domain are saved on the trail, upon modification. A classical timestamp technique is used to avoid trailing more than once per choice-point.

With this new implementation for the *Sparse* domain, it is now possible to store negative values and the domain bounds are no longer limited to a static arbitrary value, thereby rendering the *extra_cstr* flag useless.

6 Performance Analysis

In this section we compare the original FD constraint solver to a version that includes the new extensions. Table 2 presents the total execution times (in milliseconds) for runs of several benchmarks. *Neg + Ovfl* consists of the negative values extension and the overflow prevention (the *Ovfl* extension is implemented simultaneously with *Neg*). *Neg + Ovfl + Dom* includes all three extensions presented in this article. Times are measured on a 64-bit i7 Processor, 2.40GHz×8 with 8GB memory running Linux (Ubuntu 13.10).[2]

[2] The results can be reproduced with version 1.4.4 of GNU Prolog for the current version and the git branch `negative-domain` for the new version.

Program	Original	Neg + Ovfl		Neg + Ovfl + Dom	
	Time	Time	Speedup	Time	Speedup
`queens 29`	429	414	1.04	644	0.66
`digit8 ff` (×100)	787	1197	0.66	1082	0.73
`qg5 11` (×10)	610	593	1.03	813	0.75
`queens ff 100`	156	153	1.02	201	0.77
`partit 600`	200	266	0.75	254	0.79
`eq20` (×100)	189	249	0.76	228	0.83
`crypta` (×1000)	888	1016	0.87	1075	0.83
`langford 32`	551	549	1.00	646	0.85
`magsq 11`	810	802	1.01	923	0.88
`multipl` (×10)	567	577	0.98	604	0.94
`magic 200`	180	178	1.02	180	1.00
`donald` (×10)	167	158	1.06	166	1.00
`alpha` (×10)	409	407	1.00	396	1.03
`interval 256` (×10)	217	205	1.06	140	1.55
Geometric mean	364	389	**0.94**	413	**0.88**

Table 2. Performance Impact of Extensions (times in ms.)

The original implementation and the benchmark tests are solely designed for the positive domain. Therefore the domain bounds are restricted to positive values (using the environment variable discussed in Section 3.2), while making use of the updated constraint definitions. Multiple test runs show an estimated standard deviation of 3 milliseconds. The annotation (×10) indicates that the test time is formed from 10 consecutive executions (to reduce the effect of the deviation).

On average, the introduction of negative domains + overflow detection penalizes the benchmarks by 6%. This slowdown is an expected consequence of the increased complexity, and we are quite pleased that it turns out small. The worst case is for `digit8 ff` with a 34% performance loss (see [15] for a definition of "performance gain"). The reason for this is because the square root is often calculated, which is slower as both the positive and negative solutions are considered in the predicates. The best case scenario is for `donald`, which exhibits a 6% performance gain over the base version: the redesign for the predicates actually improved the solver's performance in several cases.

With the inclusion of the new *Sparse* domain alongside the other extensions, on average the benchmarks suffer a performance loss of 12%. The worst case test is `queens 29` with 34% and the best case, `interval 256`, has a 55% performance gain over the base version. The `queens 29` test creates a lot of holes on a small domain which is more efficient with a bit-vector than *MinMax* chunks. The `interval 256` test often iterates on domains: this is more efficient in the new *Sparse* domain because finding the n^{th} element is achieved in *O(nr. of holes)* time. The base version has to iterate over the bit-vector until the n^{th} element is found, making the time complexity *O(size of bit-vector)*.

Note that these benchmark tests do not utilize the enhanced capabilities of the new solver. For instance, test programs that use the negative domain cannot be tested in the original solver. It is therefore difficult to make a fair comparison.

7 Future Work

While the results show that the extensions only cause a limited slowdown factor, there is much room for improvements.

The measures taken to prevent overflows can be optimized further. In the new implementation, several unnecessary preventive checks are still being done: for instance, for the constraint $X + Y = Z$ no overflow detection is needed when computing Z, since adding two 29-bit values cannot cause overflow in 32-bit arithmetic, yet it's being checked for. Furthermore, when the run-time domain bounds imply that no overflows can occur; for instance if X and Y are in 0..10 there is no need to check for overflow in the constraint $X \times Y = Z$, since domains are reduced monotonically. As seen in section 3.2, supporting negative numbers for $X^A = Y$ implies testing the parity of A. At present this is done every time the constraint is reactivated, however, with a slightly more complex compilation scheme, there will be two versions of the execution function (see 2.1): one specialized for even As and another for odd. The installation function would be responsible to select the adequate execution function, depending on the actual value of A at run-time. This will entail enriching the FD language to be able to express user-specified installation procedures.

It will definitely be interesting to combine our new *Sparse* domain representation with bit-vectors, whenever applicable. We will experiment in this direction. Similarly, instead of using a (doubly-linked) list for maintaining chunks, a tree-structure is likely to be more efficient. Ohnishi et al. [14] describe how a balanced tree structure is realized on interval chunks. Incorporation of this structure should improve the time complexity on insertion and deletion from $O(n)$ to $O(\log n)$ (for n as the number of chunks) in worst case scenarios.

The added expressiveness allows us to tackle more complex problems, which were previously hard or impossible to model. These will also have to be benchmarked against other systems.

8 Conclusion

We presented a set of extensions to the GNU Prolog FD solver which allow it to more gracefully handle real-world problems. Central to these is a domain representation that, in order to gain generality, forgoes the compactness found in the existing solver: we moved from static vectors to dynamic data structures. The solver is now also capable of handling negative values and measures were taken to improve its robustness and correctness. The result is a system which can more easily model complex problems.

The performance evaluation of the initial, suboptimal, implementation shows encouraging results: the slowdown is quite acceptable, in the order of 12%. Furthermore, we have proposed ways to further reduce the impact of these design options, and thus hope to reclaim the lost performance.

References

1. Krzysztof R. Apt. *Principles of constraint programming.* Cambridge University Press, 2003.
2. Federico Bergenti, Alessandro Dal Palù, and Gianfranco Rossi. Integrating Finite Domain and Set Constraints into a Set-based Constraint Language. *Fundam. Inform.*, 96(3):227–252, 2009.
3. Björn Carlson, Mats Carlsson, and Daniel Diaz. Entailment of Finite Domain Constraints. In Pascal Van Hentenryck, editor, *ICLP*, pages 339–353. MIT Press, 1994.
4. Philippe Codognet and Daniel Diaz. clp(B): Combining Simplicity and Efficiency in Boolean Constraint Solving. In Manuel V. Hermenegildo and Jaan Penjam, editors, *PLILP*, volume 844 of *Lecture Notes in Computer Science*, pages 244–260. Springer, 1994.
5. Philippe Codognet and Daniel Diaz. Compiling Constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.
6. Philippe Codognet, Daniel Diaz, and Francesca Rossi. Constraint Retraction in FD. In Vijay Chandru and V. Vinay, editors, *FSTTCS*, volume 1180 of *Lecture Notes in Computer Science*, pages 168–179. Springer, 1996.
7. Rina Dechter. *Constraint processing.* Elsevier Morgan Kaufmann, 2003.
8. Daniel Diaz, Salvador Abreu, and Philippe Codognet. On the implementation of GNU Prolog. *TPLP*, 12(1-2):253–282, 2012.
9. Carmen Gervet. Conjunto: Constraint Logic Programming with Finite Set Domains. In Maurice Bruynooghe, editor, *ILPS*, pages 339–358. MIT Press, 1994.
10. Frédéric Goualard, Frédéric Benhamou, and Laurent Granvilliers. An Extension of the WAM for Hybrid Interval Solvers. *Journal of Functional and Logic Programming*, 1999(Special Issue 1), 1999.
11. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.
12. Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, Implementation, and Evaluation of the Constraint Language cc(FD). In Andreas Podelski, editor, *Constraint Programming*, volume 910 of *Lecture Notes in Computer Science*, pages 293–316. Springer, 1994.
13. Vianney Le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-Sets for Domain Implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, September 2013.
14. Shuji Ohnishi, Hiroaki Tasaka, and Naoyuki Tamura. Efficient Representation of Discrete Sets for Constraint Programming. In Francesca Rossi, editor, *CP*, volume 2833 of *Lecture Notes in Computer Science*, pages 920–924. Springer, 2003.
15. David A Patterson and John L Hennessy. *Computer Organization and Design: the Hardware/Software Interface.* Morgan Kaufmann, 2013.
16. F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming.* Elsevier, 2006.