Formally Verified Domains

Catherine Dubois

ENSIIE

EuroProofNet WG3 - 17 september 2025

When CP meets FM...

Context: formal verification of algorithms and results in Constraint Programming
$\longrightarrow$ CP(FD) solver formally verified in Coq [FM 2012, WFLP 2020]

CSP (constraint satisfaction problem) = $(\mathcal{X}, \mathcal{D}, \mathcal{C})$
- $\mathcal{X}$: set of variables,
- $\mathcal{D}(x)$: domain of variable x, finite set
- $\mathcal{C}$: set of constraints to be satisfied

Main techniques for solving a CSP: domain pruning + propagation + enumeration of values + backtracking

Representation of a domain: range sequences [Coq, Ledein & Dubois, JFLA 2019], bit vectors, sparse sets, ...

Sparse sets

Mutable data structure studied in 1993 by Briggs and Torczon, appearing as an exercise in "The Design and Analysis of Computer Algorithms" (1974, Aho, Hopcroft, and Ullman)

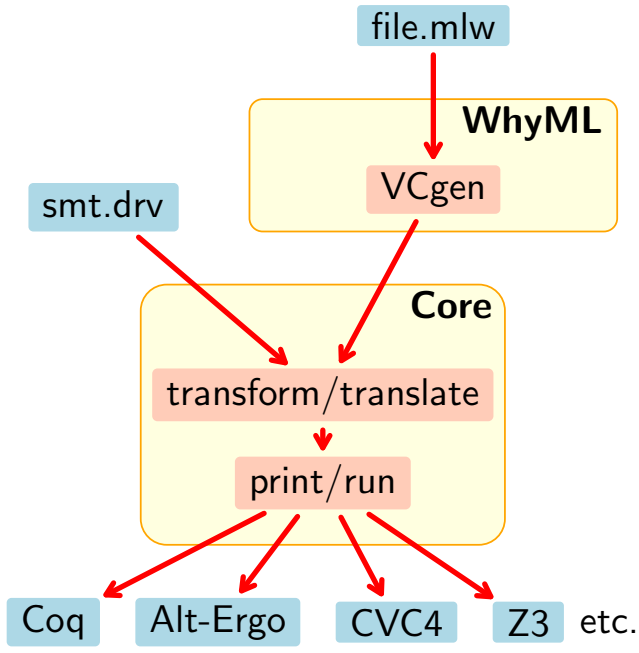
Different variants, for example: sparse arrays, Vacid-0 benchmark (2010), sparse sets as domains in CP (e.g. MiniCP, OsCaR) as in Le Clément de Saint-Marcq et al 2013, similarity to Knuth's dancing links

Numerous implementations on the web, in several languages
A formally verified implementation of *sparse arrays* in Why3 (mem/insert/remove) (Filliâtre and Paskevich)

$\longrightarrow$ Contribution: formally verified implementation of sparse sets as reversible domains for integer or set variables, developed with Why3 and extracted into OCaml code.

A very quick tour of Why3

- a specification and a programming language, WhyML
  - polymorphism, pattern-matching
  - exceptions, mutable data structures (controlled *aliasing*)
- a polymorphic first-order logic
  - algebraic datatypes, recursive definitions, inductive predicates
  - annotations and contracts (requires/ensures)
  - ghost code



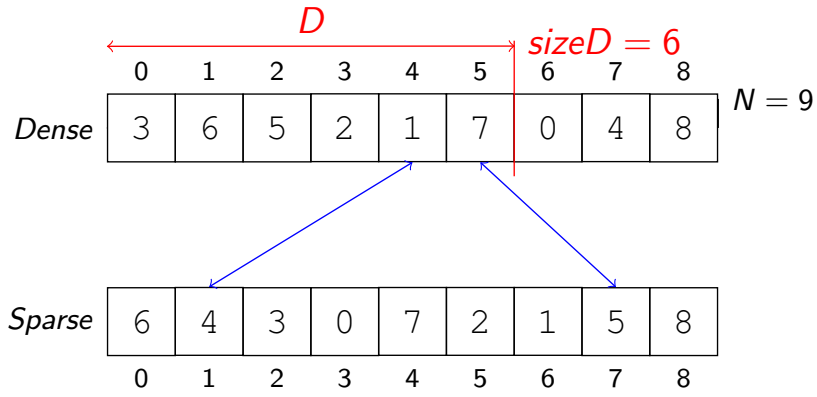
Credits: A. Paskevich

Auto-active verification (automation through additional guiding annotations, e.g. assertions, ghost code, lemma functions, etc.)

Extraction of OCaml (or C) code

Integer Domain represented by a sparse set

2 arrays and an integer to represent $D \subseteq 0..N-1$, $0 \leq N$



Invariants

$$D \subseteq [0, N-1] \wedge D = \{Dense[i] \mid 0 \leq i < sizeD\} \quad (P_1)$$

$$Sparse[v] = i \iff Dense[i] = v, \text{ pour tout } i \text{ et } v \quad (P_2)$$

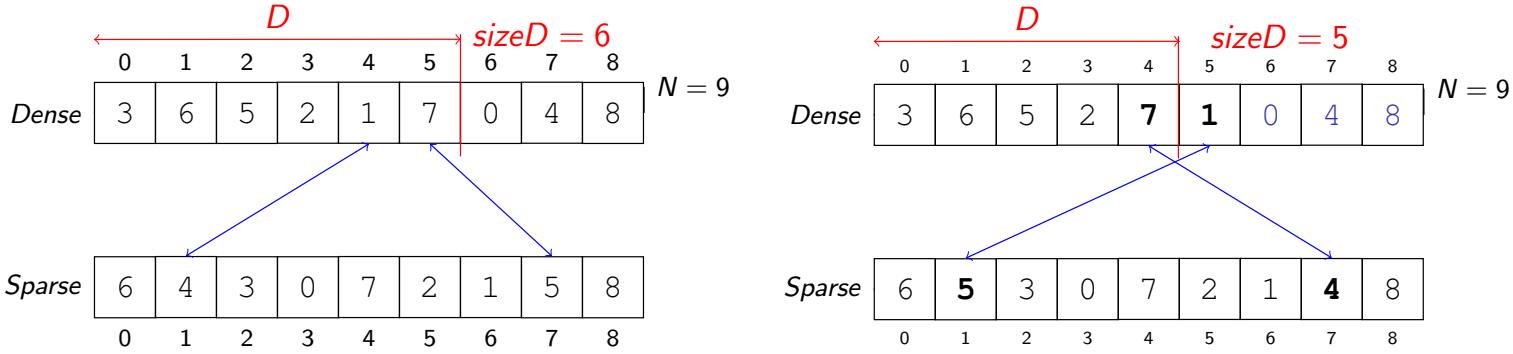
Operations

Efficient operations

- $v \in D$? $\longrightarrow$ $Sparse[v] < sizeD$?

- remove v from D $\longrightarrow$ swap v and $Dense[sizeD - 1]$, decrement $sizeD$ and update $Sparse$

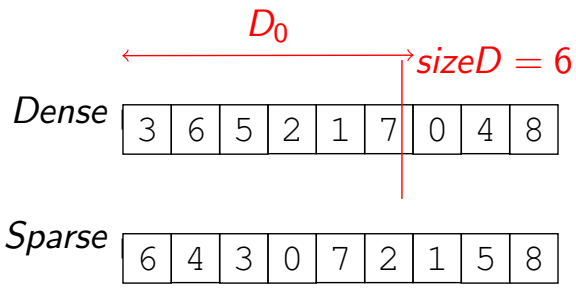
  Ex: remove 1



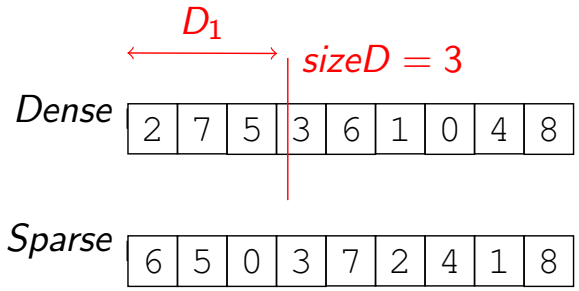
Operations

Easy to trail when backtracking

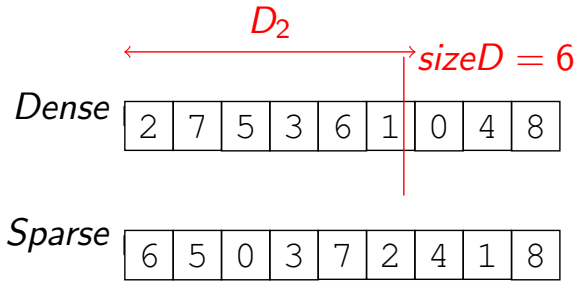
$\longrightarrow$ the only value to save and restore is *sizeD*



(a) Save



(b) After removing 1, 6 et 3



(c) Restore ($D_2 = D_0$)

WhyML Datatypes

```
type tsparse = { n : int;
        mutable dense: array int;
        mutable sparse: array int;
        mutable sizeD: int;
        mutable ghost setD : fset int; -- model = finite set

          }
invariant {
   dom_ran dense n  && dom_ran sparse n  &&
   0 <= sizeD <= n &&
   setD ⊆ (interval 0 n) && %P1
   (forall x:  int.  0<=x<n ->(sparse[x] < sizeD <-> x ∈ setD)) && %P1
   (forall i:int.  0 <= i < sizeD -> sparse[dense[i]]=i)  %P2
         }
```

WhyML Datatypes

```
type tsparse = { n : int;
        mutable dense: array int;
        mutable sparse: array int;
        mutable sizeD: int;
        mutable ghost setD : fset int; -- model = finite set

          }
invariant {
   dom_ran dense n  && dom_ran sparse n  &&
   0 <= sizeD <= n &&
   setD ⊆ (interval 0 n) && %P1
   (forall x:  int.  0<=x<n ->(sparse[x] < sizeD <-> x ∈ setD)) && %P1
   (forall i:int.  0 <= i < sizeD -> sparse[dense[i]]=i)  %P2
         }
```

⟶ **Verification conditions:**

- the type is inhabited
- each function producing/modifying a `tsparse` must preserve the invariant

WhyML Datatypes

```
type tsparse = { n : int;
        mutable dense: array int;
        mutable sparse: array int;
        mutable sizeD: int;
        mutable ghost setD : fset int; -- model = finite set
        mutable ghost states :  fmap(fset int);
          }
invariant {
   dom_ran dense n  && dom_ran sparse n  &&
   0 <= sizeD <= n &&
   setD ⊆ (interval 0 n) && %P1
   (forall x:  int.  0<=x<n ->(sparse[x] < sizeD <-> x ∈ setD)) && %P1
   (forall i:int.  0 <= i < sizeD -> sparse[dense[i]]=i)  %P2
   inv_states states setD n sparse          }
```

⟶ To specify `undo` :

- introduce the set of previous states ... as a ghost variable in the invariant of `tsparse`
- `states`: partial function from [0,`n`] to `fset(int)` with `states(p)`=value of `setD` when `sizeD` was `p`
- invariant property on `states`

WhyML Datatypes

```
type tsparse = { n : int;
        mutable dense: array int;
        mutable sparse: array int;
        mutable sizeD: int;
        mutable ghost setD : fset int; -- model = finite set

          }
invariant {
   dom_ran dense n  && dom_ran sparse n  &&
   0 <= sizeD <= n &&
   setD ⊆ (interval 0 n) && %P1
   (forall x:  int.  0<=x<n ->(sparse[x] < sizeD <-> x ∈ setD)) && %P1
   (forall i:int.  0 <= i < sizeD -> sparse[dense[i]]=i)  %P2
   inv_states states setD n sparse          }
```

⟶ Invariant on `states`:

- the domain of definition is included in `[|setD|,n]`
- the current state is stored in it
- each stored state is well-formed and contains `setD`
- link with sparse:
  if `states(i) = s` then
  `(forall x.  0<=x<n -> (sparse[x]<i <-> x ∈ s)`

Operation `remove` in WhyML

```
let remove (v : int) (a : tsparse)


=
swap_two_arrays a.dense a.sparse a.n a.sparse[v] (a.sizeD - 1);
a.sizeD <- a.sizeD - 1;
```

Operation `remove` in WhyML

```
let remove (v : int) (a : tsparse)
requires v ∈ a.setD
ensures a.setD == (old a).setD - {v}
=
swap_two_arrays a.dense a.sparse a.n a.sparse[v] (a.sizeD - 1);
a.sizeD <- a.sizeD - 1;
```

Operation `remove` in WhyML

```
let remove (v : int) (a : tsparse)
requires v ∈ a.setD
ensures a.setD == (old a).setD - {v}
=
swap_two_arrays a.dense a.sparse a.n a.sparse[v] (a.sizeD - 1);
a.sizeD <- a.sizeD - 1;
a.setD <- a.setD - {v};
a.states <- fmap_add a.sizeD a.setD a.states
```

Operation `remove` in WhyML

```
let remove (v : int) (a : tsparse)
requires v ∈ a.setD
ensures a.setD == (old a).setD - {v}
=
swap_two_arrays a.dense a.sparse a.n a.sparse[v] (a.sizeD - 1);
a.sizeD <- a.sizeD - 1;
a.setD <- a.setD - {v};
a.states <- fmap_add a.sizeD a.setD a.states
```

$\longrightarrow$ **Verification conditions:**

- after executing `remove v a`, the invariant is satisfied for the new value of `a` (it is assumed for the input `a`)
- the preconditions of `swap_two_arrays` are satisfied
- after executing `remove v a`, the postcondition is satisfied

Operation `undo` in WhyML

```
let undo (a : tsparse) (p : int) : unit

=

a.sizeD <- p ;
```

Operation `undo` in WhyML

```
let undo (a : tsparse) (p : int) : unit
requires p ∈ dom(a.states)
ensures exists s.  (old a).states(p) = s && a.setD == s
=


a.sizeD <- p ;
```

Operation `undo` in WhyML

```
let undo (a : tsparse) (p : int) : unit
requires p ∈ dom(a.states)
ensures exists s.  (old a).states(p) = s && a.setD == s
=
let ghost (v :  fset int) = a.states(p) in
a.setD <- v ; a.states <- [0, p-1] ⩤ a.states
a.sizeD <- p ;
```

Operation undo in WhyML

```
let undo (a : tsparse) (p : int) : unit
requires p ∈ dom(a.states)
ensures exists s.  (old a).states(p) = s && a.setD == s
=
let ghost (v :  fset int) = a.states(p) in
a.setD <- v ; a.states <- [0, p-1] ⩤ a.states
a.sizeD <- p ;
```

Operation `undo` in WhyML

```
let undo (a : tsparse) (p : int) : unit
requires p ∈ dom(a.states)
ensures exists s.  (old a).states(p) = s && a.setD == s
=
let ghost (v :  fset int) = a.states(p) in
a.setD <- v ; a.states <- [0, p-1] ⩤ a.states
a.sizeD <- p ;
```

Defensive version: we check in the code that `p` is a size already encountered

$\longrightarrow$ `sizeD` becomes a reversible integer (value + list of previous values)

```
type rint = { mutable value : int;
              mutable back : list int;
            }
invariant { sorted back && distinct back &&
            forall n. mem n back -> value < n }
```

Operation undo in WhyML

```
type tsparse = { n : int;
        mutable dense: array int; mutable sparse: array int;
        mutable sizeD: rint;
        mutable ghost setD : fset int; -- model = finite set
        mutable ghost states :  fmap(fset int);
          }
invariant {
   ...
  forall x.(x = sizeD.value ||mem x sizeD.back) <-> x ∈ dom states
        }

let undo (a : tsparse) (p : int) : unit
ensures exists s.  (old a).states(p) = s && a.setD == s
raises Inconsistent -> p ∈ dom(a.states)
=
try
begin
  undo_rint p a.sizeD ; (| -- replace a.sizeD <- p
  let ghost (v :  fset int) = a.states(p) in
  a.setD <- v ; a.states <- [0, p-1] ⩤ a.states
end
with  Not_found_rint -> raise Inconsistent
end
```

Why3 verification results

Specification, implementation and proof of 19 operations in total[1]
`https://gitlab.com/cdubois/why3_sparsesets`

830 lines of code, 4 lines of logical annotation for 1 line of *computation*

Automatic proof of verification conditions and a few lemmas with 3 automatic provers, CVC4, Alt-Ergo and Z3 using the strategy Auto Level 2.

Prover	*nb.proofs*	*min.time(s)*	*max.time(s)*	*av.time(s)*
Alt-Ergo 2.5.1	121	0.00	8.78	0.38
CVC4 1.6	397	0.01	9.55	0.29
Z3 4.8.9	6	0.11	4.33	0.05.

[1] C. Dubois. Deductive Verification of Sparse Sets in Why3. VSTTE 2024

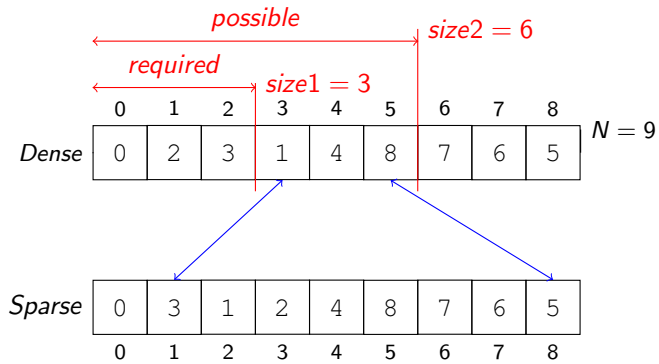
OCaml extraction

- To obtain OCaml code with native type `int`, update the previous WhyML code to use machine integers (`int63` instead `int`, `array63` instead `array`): syntactic modifications, coercions into logical assertions
- More VCs but here, no overflow verification conditions
- Reproof! The proof remains automatic
- Extracted code used in a simple sudoku solver (originally written in OCaml by Filliâtre): intense use of backtracking (undo operation)

Set domain represented by a sparse subset

Abstract model = set interval (Gervet, 1994) = [*req*, *poss*]
where *req* = set of required values and pos = set of possible values

represented by a sparse subset, i.e. 2 arrays and **two** integers to represent $D \subseteq \mathbb{P}([0, N-1])$, $0 \leq N$ (Le Clément de Saint-Marcq et al, 2013))



Invariants

$pos \subseteq [0, N-1] \wedge req \subseteq pos \wedge$
$D = \{s | req \subseteq s \subseteq pos\} 0 \leq size1 \leq size2 \leq N$
$0 \leq size1 \leq size2 \leq N$
$req = \{Dense[i] \mid 0 \leq i < size1\} \wedge pos = \{Dense[i] \mid 0 \leq i < size2\}$ (P_1')
$Sparse[v] = i \iff Dense[i] = v$, for all i and v (P_2)

Type of sparse subsets in WhyML

```
type tsparsesubset = {
  n : int;   mutable size12: rpair;
  mutable dense: array(int); mutable sparse: array(int);
  ghost mutable setD: fset (fset int);
  ghost mutable required: fset int;   ghost mutable possible: fset int
  ghost mutable states: fmap (int,int) (fset int, fset int) }
invariant {
possible ⊆ (interval 0 n) && required ⊆ possible &&
forall s.  s ∈ setD <->(required ⊆ s ⊆ possible) &&
dom_ran dense n && dom_ran sparse n &&
forall i.  0<=i<n -> sparse[dense[i]]=i &&
0<=get1(size12.value)<= n  &&
get1(size12.value)<=get2(size12.value) <= n &&
forall x.  0<=x<n ->  (sparse[x]<get1(size12.value) <-> x ∈ required)
forall x.  0<=x<n -> (sparse[x]<get2(size12.value) <-> x ∈ possible) &
inv_states_set states required possible size12 sparse &&
forall x. (x = sizeD.value || x ∈ sizeD.back) <-> x ∈ dom states }
```

Operations : create, full_interval, excludes_val, requires_val, excludesAll, requiresAll, etc. (inspired from OscaR)

$\longrightarrow$ All proved in Why3 and extracted to OCaml [JFPC 2025]

Conclusion

Summary

- Formally verified implementation of sparse sets (as domains) and sparse subsets using Why3
- Formally verified implementation of sparse sets as defined by Briggs and Torczon using Why3
- Code available online (`https://gitlab.com/cdubois/SparseSets`)

Conclusion

Summary

- Formally verified implementation of sparse sets (as domains) and sparse subsets using Why3
- Formally verified implementation of sparse sets as defined by Briggs and Torczon using Why3
- Code available online (`https://gitlab.com/cdubois/SparseSets`)

Future work

- C code extraction
- Formalization and proof of `sparse bitsets` (Demeulenaere et al, 2016)
- Integration of this family of data structures with *Snapshottable Stores* (Allain et al, 2024), which allow an arbitrary set of references to be captured and restored all at once