Formal Verification for Constraint Solving

Catherine Dubois

ENSIIE, Samovar, Évry, France

joined work with A. Butant, M. Carlier,
V. Clément, S. Elloumi, A. Gotlieb, A. Ledein and H. Mlodecki

Formal Verification of CP solvers

- Few work about CP solving & formal verification
- Verification of real code : too hard ! ... but maybe possible for some pieces ....
- Formal development (or proof-based development) :
  - Colibrics developed by F. Bobot (Why3),
  - {log} Set Constraints Resolution [Dubois and Weppe, 2018] (Coq/Rocq)
  - CoqBinFD and its extensions by C. Dubois et al (Coq/Rocq, Why3)
  - My objectives :
    - Develop a formally verified CP(FD) solver (at least as a reference solver)
    - Formalize and understand deeply some CP algorithms and results

Roadmap of the talk

1. Preliminary CP & FM background
2. Formally verified binary constraints solver
3. Extensions
   - From binary to non-binary constraints
   - Domain representations
   - Towards a AllDifferent constraint formally verified
4. Some ideas about proof logging

Preliminaries

A CSP (Constraint Satisfaction Problem or constraint network) is a triple (X, D, C) where

- X : a set of variables,
- D : a function that maps each variable of X to its domain (**here finite** set of possible values),
- C : a set of constraints (relations btw variables) over variables of X, arity of a constraint $=$ number of its variables.

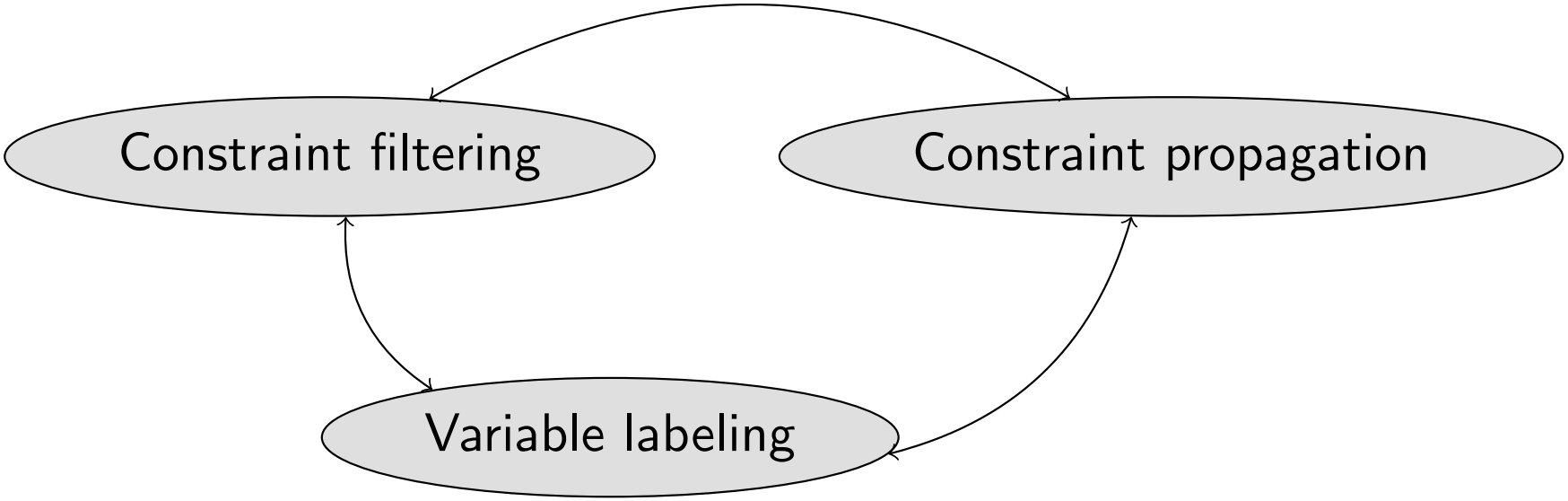
In a binary csp, all the constraints are binary - In a non-binary csp, at least one constraint has an arity ≥ 3

A solution of (X, C, D) is a valid (compatible with D) assignment defined for all the variables in X that satisfies all the constraints in C
A csp is unsatisfiable when it has no solution

CP solving

Main idea of CP(FD) solving algorithms = repeatedly removing inconsistent values from the domains (propagators).

3 interleaved processes



Maintain/enforce a local consistency property during search

Many local consistency properties : arc-consistency (AC), generalized arc-consistency (GAC), path consistency, bound-consistency, etc.

A very quick tour of Rocq

What is Rocq ?

- A functional programming language (recursion, algebraic datatypes, pattern-matching, (dependent) types)
- A specification language (higher order, inductive types and predicates)
- An interactive prover (but also decision procedures, user-defined tactics), a proof checker (Curry-Howard correspondence)
- A recognized tool used in some industries and education, a tool we can trust (thanks to the MetaRocq project)

To do what ?

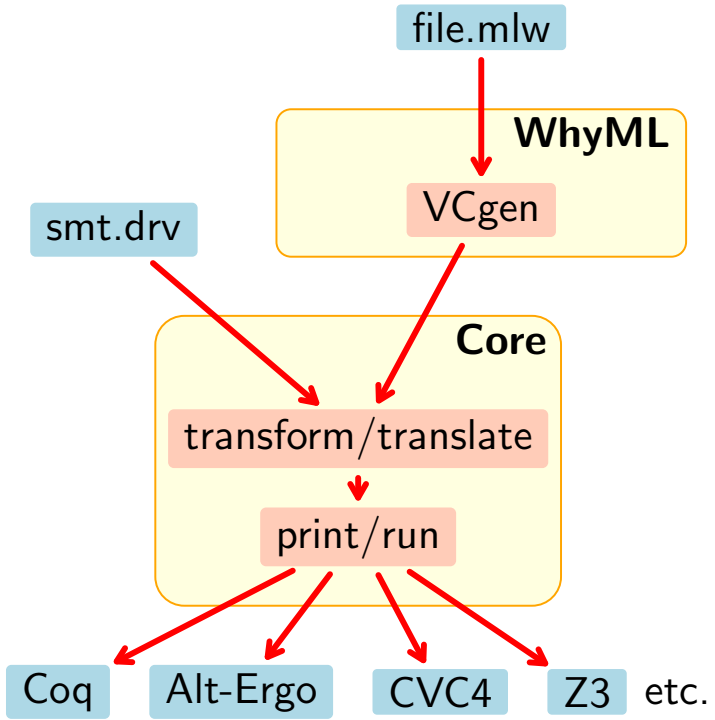
- Formalize and verify theorems (4 colors - Feit-Thompson)
- Build formally verified software (Compcert)

How ?

Write the code as in OCaml, write specs and proofs, and then extract standalone 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, récursives 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

CoqbinFD

- A formally verified CP(FD) solver relying on the Rocq interactive proof assistant
- proved sound and complete
- generic, parametrized by the language of constraints itself
- only dealing with binary CSPs
- implementing a classical algorithm AC3 (Mackworth 77) (at the heart of main existing solvers), focusing on arc-consistency
- written in OCaml, extracted from Rocq
- featuring a raisonnable efficiency (but not competitive with existing solvers)

Rocq formalization of a CSP

A key feature : genericity

- *variable* : any type equipped with a decidable equality and a strict order
- *value* : any type with a decidable equality
- *constraint* : also an abstract type, we ask for 2 functions :

  `Parameter` *interp* : *constraint* $\rightarrow$ *value* $\rightarrow$ *value* $\rightarrow$ *bool*.
  It gives the semantics of the constraints

  `Parameter` *get_vars* : *constraint* $\rightarrow$ *variable* $\times$ *variable*.
  It allows us to retrieve the variables of a constraint

  consistent c x u y v is defined as
  get_vars c $= (x,y) \wedge$ *interp c u v* $=$ *true*.

```
Record network : Type := Make_csp {
CVars : list variable ;
Doms : mapdomain ;
Csts : list constraint
}.
```

with *mapdomain* : type of maps indexed by variables with values as list (without replicates) of elements of type *value*, built from the Rocq map module.

Well-formedness of a constraint network

`Record` *network_inv csp* : `Prop` := *Make_csp_inv* {

Dwf : $\forall$ *x*, *In x* (*Doms csp*) $\leftrightarrow$ *In x* (*CVars csp*) ;

The map of domains is defined on the variables of the csp and only those ones.

Cwf1 : $\forall$ (*c* :*constraint*) (*x1 x2* : *variable*),
c $\in$ (*Csts csp*) $\rightarrow$ *get_vars c* = (*x1*, *x2*) $\rightarrow$
x1 $\in$ (*CVars csp*) $\wedge$ *x2* $\in$ (*CVars csp*) ;

The variables appearing in the constraints are variables of the csp.

Cwf2 : $\forall$ *x*, *x* $\in$ (*CVars csp*) $\rightarrow$ $\exists$ *c*,*c* $\in$ (*Csts csp*) $\wedge$
(*fst* (*get_vars c*) = *x* $\vee$ *snd* (*get_vars c*) = *x*) ;

Each variable is used at least in one constraint.

Norm : $\forall$ *c c'*, *c* $\in$ (*Csts csp*) $\rightarrow$ *c'* $\in$ (*Csts csp*) $\rightarrow$
get_vars c = *get_vars c'* $\rightarrow$ *c* = *c'*

Two different constraints share at most one variable.

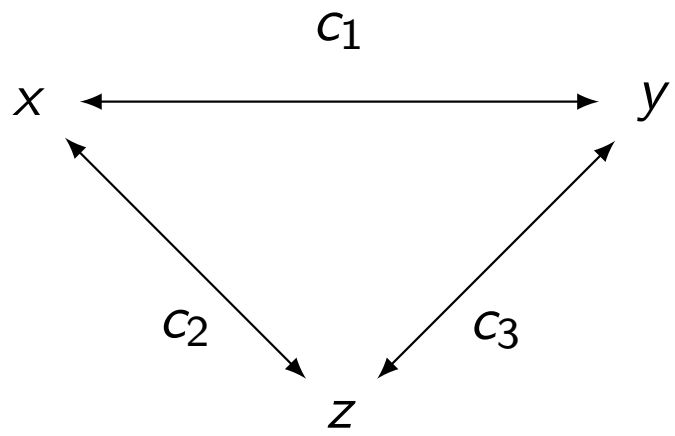
}.

Graph of Constraints

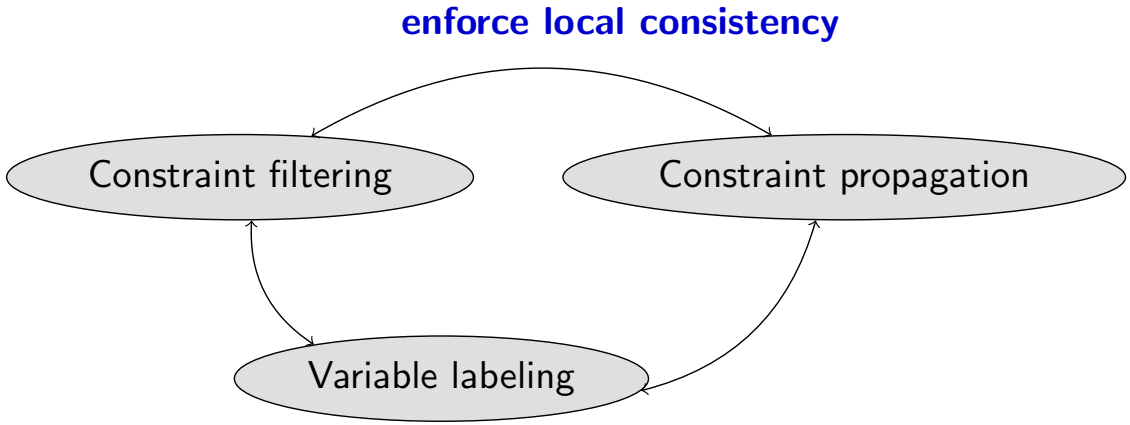
A CSP can be depicted by a constraint (symmetric directed) graph :

- nodes are variables,
- an arc relates 2 nodes x and y iff x and y are involved in a constraint c (label of the arc).

$get\_vars\ c_1 = (x, y)$
$\rightarrow$ 2 arcs
$c_1(x, y)$
$c_1(y, x)$



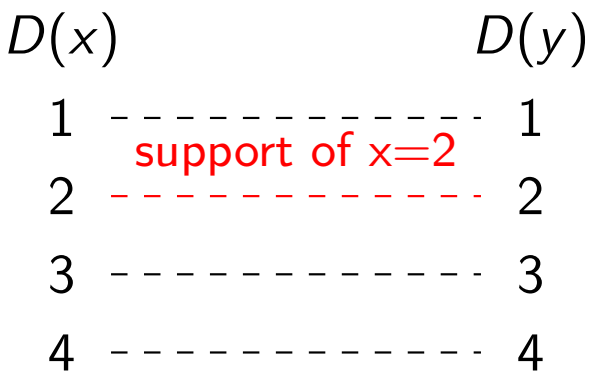
Arc-consistency



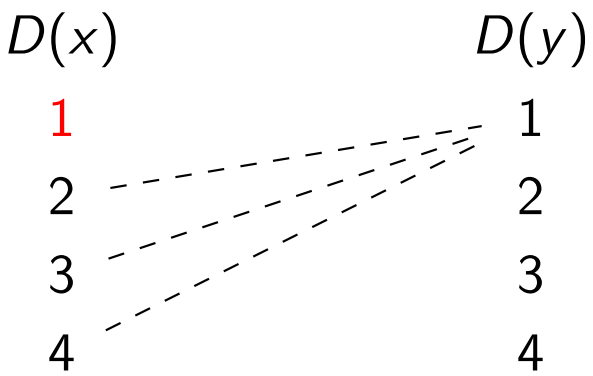
Definition

$c(x, y)$ is arc-consistent wrt (X, C, D) iff for all $u \in D(x)$, there exists at least a value (support) $v \in D(y)$ such that $c(x := u, y := v)$ is satisfied.

$c \equiv x \geq y$ arc-consistent

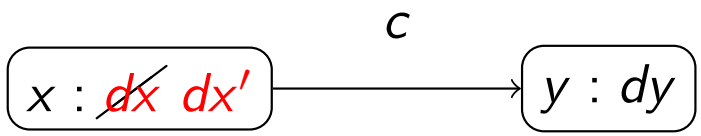


$c \equiv x > y$ not arc-consistent



The filtering algorithm *revise*= a function that prunes the domain of a variable according to a constraint.

- - x and y are the variables of c, $dx = \mathrm{D}(x)$, $dy = \mathrm{D}(y)$
revise c x y dx dy = (b, dx')
- - if b then dx' is the pruned domain of x, $dx' \subsetneq dx$
else dx'=dx.

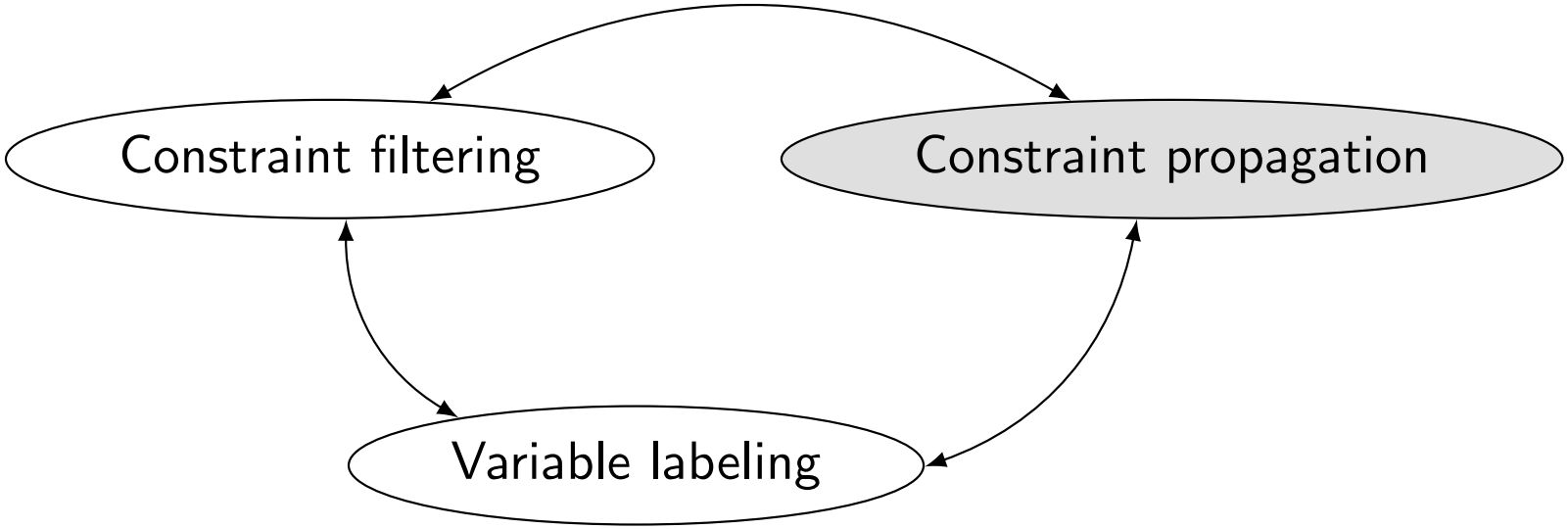



```
Fixpoint revise c x y dx dy :=
    match dx with
      nil ⇒ (false, dx)
    | v ::r ⇒ let (b, d) := revise c x y r dy in
              if List.existsb (fun t ⇒ consistent_value c x v y t) dy
              then (b, v ::d)
              else (true, d)
    end.
```

The Propagation Algorithm AC3

Propogation algorithm : a fixpoint computation algorithm that repeats filtering consistency (such as arc-consistency) over each constraint.

The most well-known one : AC3



Main idea of AC3 : just revise the arcs that may have been impacted

Maintain a worklist of the arcs to be revisited.

Definition of AC3

```
Function AC3 (doms : mapdomain, qu : list arc) {wf AC3_wf d_q} : option
mapdomain :=
match qu with
| nil ⇒ Some (doms)
| (x, c, y) : :r ⇒
  match find x doms, find y doms with
  | Some dx, Some dy ⇒
    let (bool_red, dx') := revise c x y dx dy in
      if bool_red then
        if is_empty dx'
        then None
        else AC3 (add x dx' doms, r ⊕ (incidentTo x y g))
      else AC3 (doms, r)
  | _, _ ⇒ None
  end
end.
```

lexicographic ordering *AC3_wf* based on the length of the worklist and sum of the lengths of the domains

The solver

It is implemented as a systematic search based on backtracking interleaved with propagation with a simple heuristics for selecting a variable and a value

$\longrightarrow$ a function `solve`
Either solve csp = Some a (`a` is provided as a solution) or
solve csp = None (no solution)

A sound and complete solver

- Prove soundness

  $\forall$ csp, $\forall$ a, wellformed csp $\rightarrow$
  solve csp = Some a $\rightarrow$ is_solution a csp.

  $\forall$ csp, wellformed csp$\rightarrow$
  solve csp = None $\rightarrow$ $\forall$ a, $\neg$(is_solution a csp)

- Prove completeness

  $\forall$ csp, $\forall$ a, wellformed csp $\rightarrow$
  is_solution a csp $\rightarrow$ $\exists a'$, solve csp = Some a'

  $\forall$ csp, wellformed csp $\rightarrow$
  ($\forall$ a, $\neg$(is_solution a csp)) $\rightarrow$ solve csp = None

Soundness and completeness of AC3

After application of AC3, arc-consistency is enforced.

```
Theorem AC3_sound : ∀ csp d',
  network_inv csp →
  AC3 (Doms csp, initq (Csts csp)) = Some d' →
  ∀ x y c, (x, c, y) ∈ (arcs (Csts csp)) →
      arc_consistent x y c d'.
```

The proof relies on the proof of an invariant : *if an arc is not consistent wrt d then it is in the queue.*

```
Definition PNC csts (d : mapdomain) (l : list arc) : Prop := ∀ x y c,
  (x, c, y) ∈ (arcs csts) → ¬(arc_consistent x y c d) →
    (x, c, y) ∈ l.
```

Soundness and Completeness of AC3

AC3 does not loose any solution

`Theorem` *AC3_complete* : ∀ *csp a d'*,
network_inv csp → *solution a csp* →
AC3 (*Doms csp*, (*initq* (*Csts csp*))) = *Some* (*d'*) →
solution a (*set_domains d' csp*).

Soundness and completeness

Both theorems rely on soundness and completeness of the filtering algorithm *revise*.

```
Theorem revise_arc_consistent : ∀ csp c x y ,
  c ∈ (Csts csp) → compat_var_const x y c →
  ∀ dx dy dx' b,
  find x (Doms csp) = Some dx → find y (Doms csp) = Some dy →
  revise c x y dx dy = (b, dx') →
    arc_consistent x y c (add x dx' (Doms csp)).
```

```
Theorem revise_complete : ∀ csp c x y dx dy (a : assign) ,
  network_inv csp →
  c ∈ (Csts csp) → compat_var_const x y c →
  find x (Doms csp) = Some dx → find y (Doms csp) = Some dy →
  solution a csp →
    ∀ newdx, revise c x y dx dy = (true, newdx) →
      solution a (set_domain x newdx csp).
```

Soundness and completeness

The following lemmas justify the filling of the AC3 worklist (propagation)

```
Lemma revise_x_y_consistent_y_x : ∀ csp c x y dx dy ,
  c ∈ (Csts csp) → compat_var_const x y c →
  find x (Doms csp) = Some dx → find y (Doms csp) = Some dy →
    ∀ newdx, revise c x y dx dy = (true, newdx) →
      arc_consistent y x c (Doms csp) →
        arc_consistent y x c (add x newdx (Doms csp)).
```

```
Lemma revise_x_y_consistent_x_z : ∀ d x y dx dy c newdx,
  compat_var_const x y c →
  find x d = Some dx → find y d = Some dy →
  revise c x y dx dy = (true, newdx) →
  ∀ z c0, compat_var_const x z c0 →
    arc_consistent x z c0 d →
      arc_consistent x z c0 (add x newdx d).
```

Conclusion

→ We have developed a correct constraint solver for finite domains (the first one)

- for any (binary) constraint language
- allowing to certify the absence of solutions for a CSP

→ The model counts 8500 lines of Rocq

→ The Ocaml code of the solver is extracted from Rocq, used on some different problems.

→ It is generic : propagation and labeling as functors.

- We have developed and proved 2 instances, one using AC3 and another well-known variant of it (AC2001), focusing on arc-consistency [Carlier et al, FM 2012]
- And later also another instance using bound-consistency.

From non-binary csps to binary csps

Resolution of a non-binary csp by translating/encoding it into an equivalent binary one and using well-established binary csp techniques or

Most popular encodings : dual graph encoding and hidden variable encoding (HVE) (1990)

Our work :

- Formalisation in Rocq of the hidden variable encoding, proof of its correctness
- Extension of CoqbinFD $\longrightarrow$ a formally verified CP(FD) solver for both binary and non-binary constraints.

[C. Dubois. Formally Verified Transformation of Non-binary Constraints into Binary Constraints. WFLP 2020]

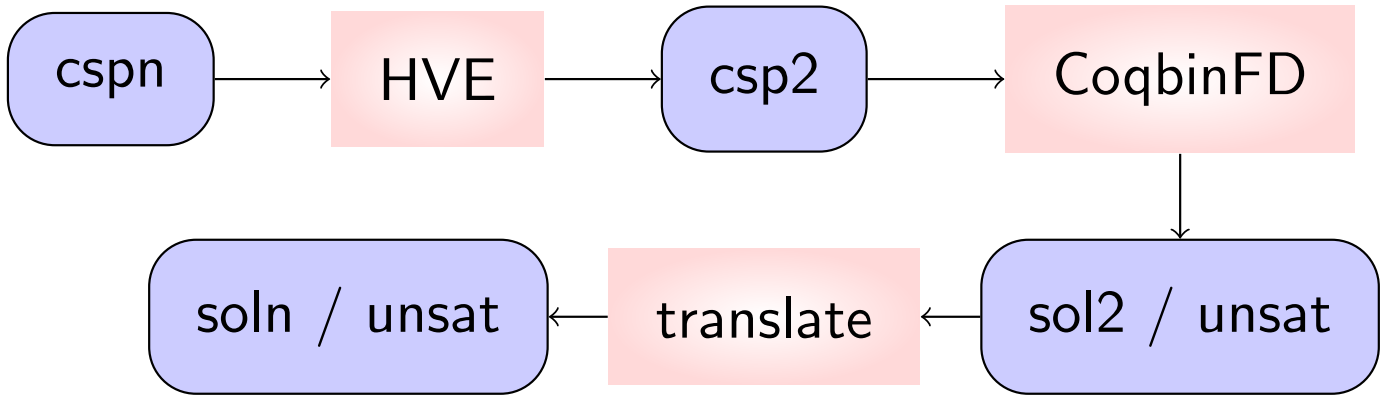
Hidden Variable Encoding

$csp(X, D, C)$	$\rightsquigarrow$	$csp'(X', D', C')$
$x \in X$	$\rightsquigarrow$	$x \in X'$ Ordinary variable
	$\rightsquigarrow$	$D'(x) = D(x)$
binary constraint c	$\rightsquigarrow$	c
n-ary constraint c' $(n > 2)$	$\rightsquigarrow$	$v_{c'} \in X'$ Hidden variable
$vars(v'_c) = \{x_1, \ldots, x_n\}$	$\rightsquigarrow$	$D'(v_{c'}) = D(x_1) \times D(x_2) \ldots D(x_n)$
	$\rightsquigarrow$	$c' \notin C'$
	$\rightsquigarrow$	$proj_i(v_{c'}, x_i) \in C', i \in [1..n]$

Each n-ary constraint $(n > 2)$ is replaced by n binary constraints
One hidden variable per non-binary constraint is added

Extension of CoqbinFD

Definition of *solve_csp_n*



solve_csp_n is proved sound and complete

- soundness and completeness of HVE (wrt satisfiability)
- soundness and completeness of *solve_csp*
- well-formedness of the HVE output
- correctness of the translation of the solution

Formally verified domains

Main domain representations in CP solvers :

Range sequences, Gap interval trees (lists or binary trees), Bit vectors, Sparse sets ...

Our contributions :

- Formalisation in Rocq of range sequences using lists (module Domain of FaCiLe)
- Formalisation in Why3 of sparse sets for integer and set variables

A. Ledein, C. Dubois. FaCiLe en Coq : vérification formelle des listes d'intervalles, JFLA 2020
C. Dubois. Deductive Verification of Sparse Sets in Why3. VSTTE 2024
C. Dubois. Domaines formellement vérifiés JFPC 2025

Range sequences - Rocq formalization

- Well-formedness of a range sequence

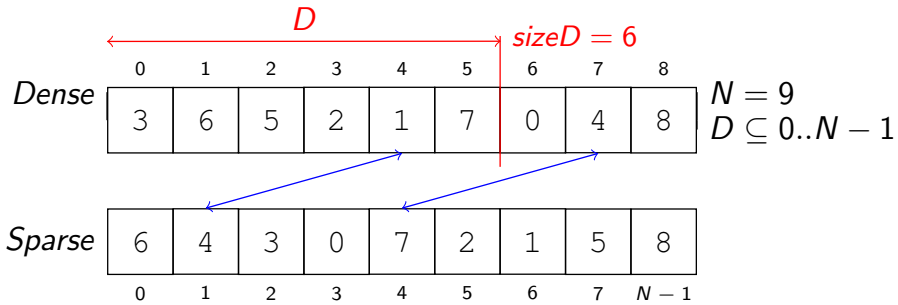
  Ranges must be sorted in the increasing order, disjoints and non empty. Two successive intervals must have a gap of at least one integer

  ```
  Inductive Inv_elt_list : Z → elt_list → Prop :=
  | invNil : ∀ b, Inv_elt_list b Nil
  | invCons : ∀ (a b j : Z) (q : elt_list),
        j ≤ a → a ≤ b → Inv_elt_list (b+2) q →
        Inv_elt_list j (Cons a b q).
  ```

- Definition of 17 set operations
- Add and remove : tricky - they require to merge or split intervals.
- For each operation, proof of the well-formedness preservation

Sparse sets as domains

2 arrays and an integer to represent $D \subseteq 0..N-1$, $0 \leq N$ (N called the width of the sparse set)



Invariants

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

$Sparse[v] = i \iff Dense[i] = v$, for all i and v (P_2)

Operations

Add, remove, membership, clear, cardinality : operations in constant time

exists, forall, tolist, copy : linear wrt the number of elements in D

... and easy to trail and backtrack : just store *sizeD*

Sparse sets as domains - Why3 formalization

Type of sparse sets

```
type tsparse = { n : int;
        mutable dense: array int;
        mutable sparse: array int;
        mutable sizeD: int;
        mutable ghost setD : fset int; -- modèle abstrait = ens fini
        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        }
```

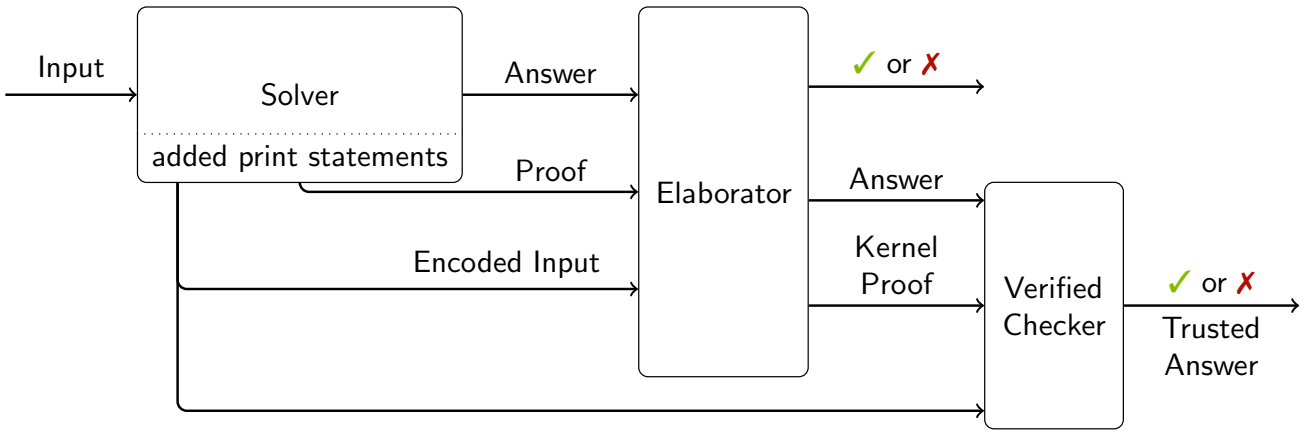
`states` records the previous states ... needed to express and prove an `undo` operation

For each operation, this invariant and also its postcondition are proved.

Extraction of OCaml code (with machine integers, proof of no overflows)

Details on Wednesday in WG3 workshop

Proof logging



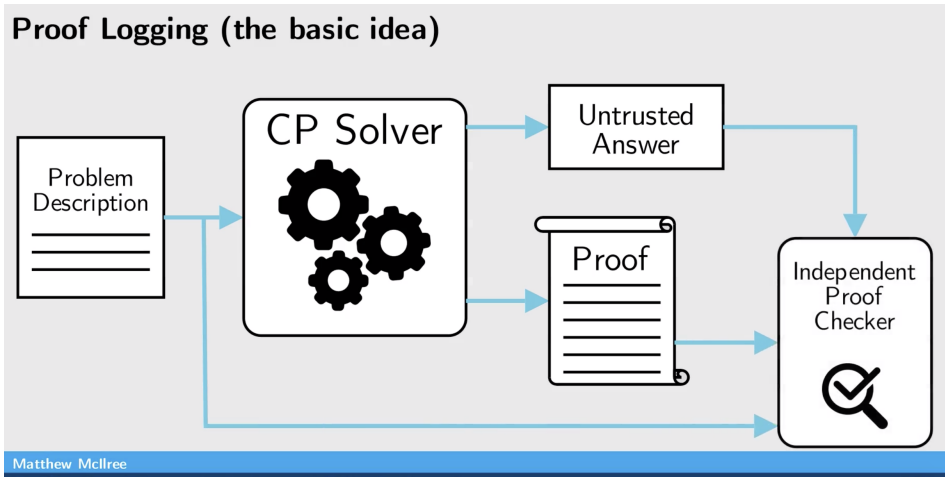
Credit : C. Mccreesh

Proposition : Dedukti/LambdaPi used as a proof checker

1. Reconstruct proof terms from the elaborated proof logs

   $\longrightarrow$ encode PB constraints and encode the proof rules (RUP, cutting places) in LambdaPi
   - Irat proofs can be checked by Dedukti (G. Burel)
   - support for arithmetic checks (e.g. for rule la_generic of SMTlib can be checked)
2. Check the proof terms by running Dedukti/LambdaPi

Proof logging of CP solvers



How can we benefit from a formally verified propagator in a proof logging approach ?