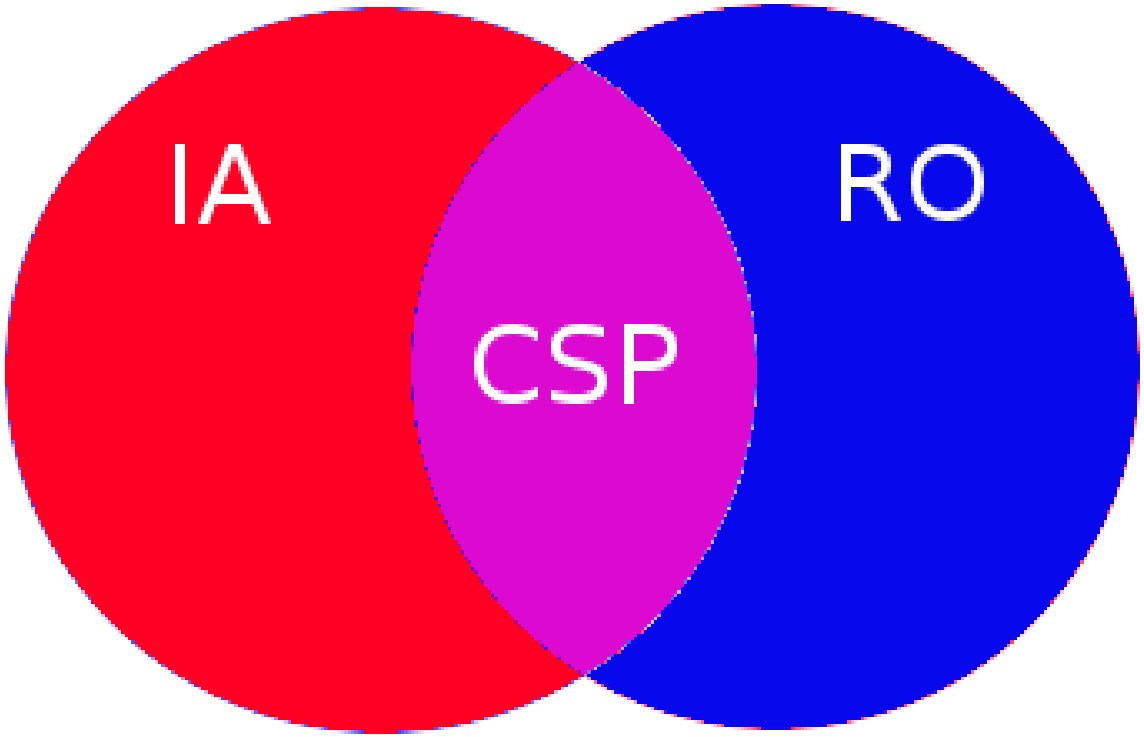
FaCiLe en Coq

Vérification formelle en Coq de l'implémentation d'un domaine d'une variable entière pour la résolution de contraintes

Amélie Ledein, en collaboration avec
Catherine Dubois

ENSIIE/SAMOVAR

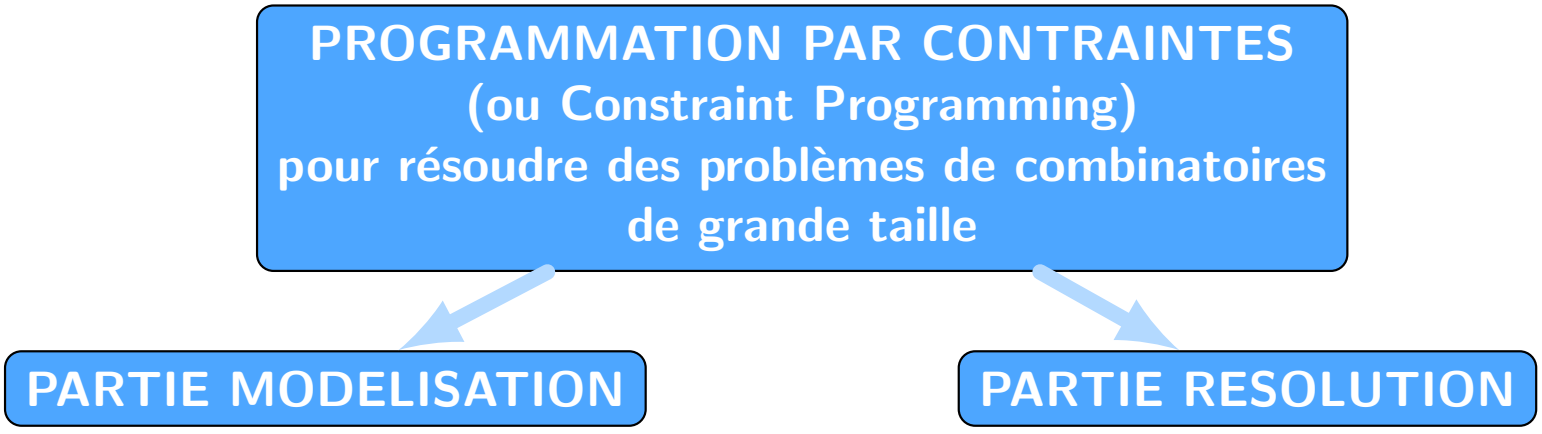
Contexte informatique



Un paradigme de programmation (années 70-80)

PROGRAMMATION PAR CONTRAINTES
(ou Constraint Programming)
pour résoudre des problèmes de combinatoires de grande taille

Un paradigme de programmation (années 70-80)



Un paradigme de programmation (années 70-80)

PROGRAMMATION PAR CONTRAINTES
(ou Constraint Programming)
pour résoudre des problèmes de combinatoires de grande taille

PARTIE MODELISATION

PARTIE RESOLUTION

Objectif : modéliser le problème à l'aide :

→ **d'inconnues** $\mathcal{X} = \{x_1, \ldots, x_n\}$

→ **de domaines** $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$

→ **de contraintes** $\mathcal{C} = \{C_1, \ldots, C_m\}$

Un CSP est un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$

Un paradigme de programmation (années 70-80)

PROGRAMMATION PAR CONTRAINTES
(ou Constraint Programming)
pour résoudre des problèmes de combinatoires de grande taille

PARTIE MODELISATION

Objectif : modéliser le problème à l'aide :
→ d'inconnues $\mathcal{X} = \{x_1, \ldots, x_n\}$
→ de domaines $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$
→ de contraintes $\mathcal{C} = \{C_1, \ldots, C_m\}$

Un CSP est un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$

PARTIE RESOLUTION

Objectif : réduire la taille de l'espace des solutions des inconnues grâce aux contraintes (propagation de contraintes / filtrage)
→ choix des algorithmes
→ choix de la représentation pour les domaines

Solveur de contraintes CoqbinFD

Solveur de contraintes CoqbinFD

- Développé par M. Carlier, C. Dubois et A. Gotlieb (2012)

Solveur de contraintes CoqbinFD

- Développé par M. Carlier, C. Dubois et A. Gotlieb (2012)
- Démontré correct et complet avec Coq

Solveur de contraintes CoqbinFD

- Développé par M. Carlier, C. Dubois et A. Gotlieb (2012)
- Démontré correct et complet avec Coq
- **Caractéristiques** :
  - Contraintes binaires
  - Domaines finis

Solveur de contraintes CoqbinFD

- Développé par M. Carlier, C. Dubois et A. Gotlieb (2012)
- Démontré correct et complet avec Coq
- **Caractéristiques** :
  - Contraintes binaires
  - Domaines finis
- **Représentation utilisée pour les domaines** : liste de valeurs

Solveur de contraintes CoqbinFD

- Développé par M. Carlier, C. Dubois et A. Gotlieb (2012)
- Démontré correct et complet avec Coq
- **Caractéristiques** :
  - Contraintes binaires
  - Domaines finis
- **Représentation utilisée pour les domaines** : liste de valeurs

$\rightarrow$ **Objectif du travail** : trouver une représentation plus efficace des domaines dans CoqbinFD

Etat de l'art des représentations d'un domaine

Etat de l'art des représentations d'un domaine

- Les représentations implémentées par des listes ou des arbres

Etat de l'art des représentations d'un domaine

- Les représentations implémentées par des listes ou des arbres
  - Ensemble d'intervalles (ECLiPSe Prolog et SICStus Prolog)

Etat de l'art des représentations d'un domaine

- Les représentations implémentées par des listes ou des arbres
  - Ensemble d'intervalles (ECLiPSe Prolog et SICStus Prolog)
  - Gap intervals tree (Naxos Solver)

Etat de l'art des représentations d'un domaine

- Les représentations implémentées par des listes ou des arbres
  - Ensemble d'intervalles (ECLiPSe Prolog et SICStus Prolog)
  - Gap intervals tree (Naxos Solver)
- Les représentations implémentées par des tableaux

Etat de l'art des représentations d'un domaine

- Les représentations implémentées par des listes ou des arbres
  - Ensemble d'intervalles (ECLiPSe Prolog et SICStus Prolog)
  - Gap intervals tree (Naxos Solver)
- Les représentations implémentées par des tableaux
  - Vecteur binaire (ILOG Solver)

Etat de l'art des représentations d'un domaine

- Les représentations implémentées par des listes ou des arbres
  - Ensemble d'intervalles (ECLiPSe Prolog et SICStus Prolog)
  - Gap intervals tree (Naxos Solver)
- Les représentations implémentées par des tableaux
  - Vecteur binaire (ILOG Solver)
  - Ensemble creux (Oscar et Castor)

Sommaire

Sommaire

FaCiLe : *Functional Constraint Library*

FaCiLe : *Functional Constraint Library*

- Bibliothèque pour la résolution de contraintes écrite en OCaml

FaCiLe : *Functional Constraint Library*

- Bibliothèque pour la résolution de contraintes écrite en OCaml
- Développée à l'ENAC à la fin des années 90

FaCiLe : *Functional Constraint Library*

- Bibliothèque pour la résolution de contraintes écrite en OCaml
- Développée à l'ENAC à la fin des années 90
- **Représentation utilisée pour les domaines** : liste d'intervalles

FaCiLe : *Functional Constraint Library*

- Bibliothèque pour la résolution de contraintes écrite en OCaml
- Développée à l'ENAC à la fin des années 90
- **Représentation utilisée pour les domaines** : liste d'intervalles
- **Composition du module** `domain` :
  - Structure de domaine
  - Domaines classiques (vide, booléen, un seul intervalle, etc.)
  - Opérations classiques (minimum, maximum, taille, etc.)
  - Opérations sur les domaines (appartenance, ajout, suppression, union, intersection, etc.)

FaCiLe : *Functional Constraint Library*

- Bibliothèque pour la résolution de contraintes écrite en OCaml
- Développée à l'ENAC à la fin des années 90
- **Représentation utilisée pour les domaines** : liste d'intervalles
- **Composition du module** `domain` :
  - Structure de domaine
  - Domaines classiques (vide, booléen, un seul intervalle, etc.)
  - Opérations classiques (minimum, maximum, taille, etc.)
  - Opérations sur les domaines (appartenance, ajout, suppression, union, intersection, etc.)
- **Caractéristique du module** `domain` :
  proche d'un code fonctionnel sans effet de bord

Sommaire

Sommaire

Structure de domaine basée sur les listes d'intervalles

- Type d'une liste d'intervalles :

```
1 Inductive elt_list :=
2 | Nil : elt_list
3 | Cons : Z -> Z -> elt_list -> elt_list.
```

- Type d'un domaine :

```
1 Record t := mk_t { domain : elt_list ;
2                    size : Z ;
3                    max : Z ;
4                    min : Z }.
```

Domaines de base

- `empty : t`
- `boolean : t`
- `interval : Z -> Z -> option t`
- `singleton : Z -> t`

Domaines de base

- `empty : t`
- `boolean : t`
- `interval : Z -> Z -> option t`
- `singleton : Z -> t`

Opérations élémentaires

- `process_size : elt_list -> Z`
- `process_max : elt_list -> Z`
- `get_min : elt_list -> Z -> Z`
- `is_empty : t -> bool`

Domaines de base

- `empty : t`
- `boolean : t`
- `interval : Z -> Z -> option t`
- `singleton : Z -> t`

Création d'un domaine

- `create : list Z -> t`

Opérations élémentaires

- `process_size : elt_list -> Z`
- `process_max : elt_list -> Z`
- `get_min : elt_list -> Z -> Z`
- `is_empty : t -> bool`

Domaines de base

- `empty : t`
- `boolean : t`
- `interval : Z -> Z -> option t`
- `singleton : Z -> t`

Opérations élémentaires

- `process_size : elt_list -> Z`
- `process_max : elt_list -> Z`
- `get_min : elt_list -> Z -> Z`
- `is_empty : t -> bool`

Création d'un domaine

- `create : list Z -> t`

Opérations sur les domaines

- `values : t -> list Z`
- `remove : Z -> t -> t`
- `member : Z -> t -> bool`
- `included : t -> t -> bool`

Domaines de base

- `empty : t`
- `boolean : t`
- `interval : Z -> Z -> option t`
- `singleton : Z -> t`

Opérations élémentaires

- `process_size : elt_list -> Z`
- `process_max : elt_list -> Z`
- `get_min : elt_list -> Z -> Z`
- `is_empty : t -> bool`

Création d'un domaine

- `create : list Z -> t`

Opérations sur les domaines

- `values : t -> list Z`
- `remove : Z -> t -> t`
- `member : Z -> t -> bool`
- `included : t -> t -> bool`

En plus de FaCiLe

- `fold_right`
- `exists_t : (Z->bool)->elt_list-> bool`
- `is_singleton : t -> Prop`
- `get_value : t -> option Z`

La fonction `create` : list Z → t

La fonction `create` : list Z → t

- Nous commençons avec une liste quelconque : `[5;3;1;5;2;1;0]`

La fonction `create` : list Z → t

- Nous commençons avec une liste quelconque : `[5;3;1;5;2;1;0]`
- Le tri `sortWithoutDup` trie et enlève les doublons : `[0;1;2;3;5]`

La fonction `create` : list Z → t

- Nous commençons avec une liste quelconque : `[5;3;1;5;2;1;0]`
- Le tri `sortWithoutDup` trie et enlève les doublons : `[0;1;2;3;5]`
- La fonction `make` peut alors former les intervalles : `[(0;3);(5;5)]`

```
/*
@requires :  min ≤ last
        Zl triée et sans doublons avec ∀x ∈ Zl, x > last
@ensures :  la liste d'intervalles correspondant à
[min,...,last]@Zl
*/
```

```
1  Fixpoint make (min last : Z) (Zl : list Z) :=
2  match Zl with
3  | nil => Cons min last Nil
4  | cons t q => if Zeq_bool t (last+1)
5                then make min (last+1) q
6                else Cons min last (make t t q)
7  end.
```

Nous appelons `make 0 0 [1;2;3;5]`.
Lorsque `last = 3`, nous appelons Cons 0 3 (make 5 5 []).
Ainsi `[0;1;2;3;5]` devient `[(0;3);(5;5)]`

- La fonction `last_and_length` peut ensuite récupérer le dernier élément ainsi que la taille de la liste, d'où :

```
{ domain = [(0;3);(5;5);(9;10)] ;
    size = 7 ;
     max = 10 ;
     min = 0 }.
```

Traduction de OCaml vers Gallina

Traduction de OCaml vers Gallina

- Traduction manuelle presque syntaxique :

Traduction de OCaml vers Gallina

- Traduction manuelle presque syntaxique :
  - preuves de terminaison (récursion structurelle pour la plupart (29/33))

Traduction de OCaml vers Gallina

- Traduction manuelle presque syntaxique :
  - preuves de terminaison (récursion structurelle pour la plupart (29/33))
  - élimination des exceptions de OCaml

Traduction de OCaml vers Gallina

- Traduction manuelle presque syntaxique :
  - preuves de terminaison (récursion structurelle pour la plupart (29/33))
  - élimination des exceptions de OCaml
    - → valeur par défaut (`get_min`)

Traduction de OCaml vers Gallina

- Traduction manuelle presque syntaxique :
  - preuves de terminaison (récursion structurelle pour la plupart (29/33))
  - élimination des exceptions de OCaml
    - → valeur par défaut (`get_min`)
    - → utilisation du type `option`

Traduction de OCaml vers Gallina

- Traduction manuelle presque syntaxique :
  - preuves de terminaison (récursion structurelle pour la plupart (29/33))
  - élimination des exceptions de OCaml
    - → valeur par défaut (`get_min`)
    - → utilisation du type `option`
  - désimbrication de certaines fonctions

Traduction de OCaml vers Gallina

- Traduction manuelle presque syntaxique :
  - preuves de terminaison (récursion structurelle pour la plupart (29/33))
  - élimination des exceptions de OCaml
    - → valeur par défaut (`get_min`)
    - → utilisation du type `option`
  - désimbrication de certaines fonctions
  - complétion des filtrages

Traduction de OCaml vers Gallina

- Traduction manuelle presque syntaxique :
  - preuves de terminaison (récursion structurelle pour la plupart (29/33))
  - élimination des exceptions de OCaml
    - → valeur par défaut (`get_min`)
    - → utilisation du type `option`
  - désimbrication de certaines fonctions
  - complétion des filtrages

∼ 100 lignes traduites et prouvées

Traduction de OCaml vers Gallina

- Traduction manuelle presque syntaxique :
  - preuves de terminaison (récursion structurelle pour la plupart (29/33))
  - élimination des exceptions de OCaml
    - → valeur par défaut (`get_min`)
    - → utilisation du type `option`
  - désimbrication de certaines fonctions
  - complétion des filtrages

∼ 100 lignes traduites et prouvées

∼ 100 lignes seulement traduites

Traduction de OCaml vers Gallina

- Traduction manuelle presque syntaxique :
  - preuves de terminaison (récursion structurelle pour la plupart (29/33))
  - élimination des exceptions de OCaml
    - → valeur par défaut (`get_min`)
    - → utilisation du type `option`
  - désimbrication de certaines fonctions
  - complétion des filtrages

∼ 100 lignes traduites et prouvées

∼ 100 lignes seulement traduites

∼ 100 lignes non traduites car moins utiles

Sommaire

Les intervalles doivent être rangés dans l'ordre croissant, tous disjoints et non vides. De plus, cette liste doit avoir un nombre d'intervalles minimum.

```
1  Inductive Inv_elt_list : Z -> elt_list -> Prop :=
2  | invNil : forall b, Inv_elt_list b Nil
3  | invCons : forall (a b  j: Z) (q : elt_list),
4        j <= a -> a <= b ->  Inv_elt_list (b+2) q ->
5        Inv_elt_list j (Cons a b q).
```

Invariant d'un domaine :

```
1  Definition Inv_t (d : t) :=
2      Inv_elt_list (min d) (domain d)
3      /\ (min d) = get_min (domain d) min_int
4      /\ (max d) = process_max (domain d)
5      /\ (size d) = process_size (domain d).
```

Bonne formation et préservation de l'invariant

Bonne formation et préservation de l'invariant

Domaines de base

- `empty : t`
- `boolean : t`
- `interval : Z->Z->option t`
- `singleton : Z -> t`

Création d'un domaine

- `create : list Z -> t`

Suppression

- `remove : Z -> t -> t`

Bonne formation et préservation de l'invariant

Domaines de base

- `empty : t`
- `boolean : t`
- `interval : Z->Z->option t`
- `singleton : Z -> t`

Theorem invBoolean :
`Inv_t (boolean).`

Création d'un domaine

- `create : list Z -> t`

Suppression

- `remove : Z -> t -> t`

Bonne formation et préservation de l'invariant

Domaines de base

- `empty : t`
- `boolean : t`
- `interval : Z->Z->option t`
- `singleton : Z -> t`

Theorem invBoolean :
`Inv_t (boolean).`

Theorem invInterval :
`∀ n p e, n ≤ p ->`
`(interval n p) = Some e ->`
`Inv_t e.`

Création d'un domaine

- `create : list Z -> t`

Suppression

- `remove : Z -> t -> t`

Bonne formation et préservation de l'invariant

Domaines de base

- `empty : t`
- `boolean : t`
- `interval : Z->Z->option t`
- `singleton : Z -> t`

Theorem invBoolean :
`Inv_t (boolean).`

Theorem invInterval :
`∀ n p e, n ≤ p -> (interval n p) = Some e -> Inv_t e.`

Création d'un domaine

- `create : list Z -> t`

Theorem create_Inv_t :
`∀ l, Inv_t (create l).`

Suppression

- `remove : Z -> t -> t`

Bonne formation et préservation de l'invariant

Domaines de base

- `empty : t`
- `boolean : t`
- `interval : Z->Z->option t`
- `singleton : Z -> t`

Theorem invBoolean :
`Inv_t (boolean).`

Theorem invInterval :
`∀ n p e, n ≤ p ->`
`(interval n p) = Some e ->`
`Inv_t e.`

Création d'un domaine

- `create : list Z -> t`

Theorem create_Inv_t :
`∀ l, Inv_t (create l).`

Suppression

- `remove : Z -> t -> t`

Theorem remove_Inv_t :
`∀ d, Inv_t d ->`
`∀ e, Inv_t (remove e d).`

Sommaire

En calculant la liste exhaustive des valeurs (`values`)

En calculant la liste exhaustive des valeurs (`values`)

Enumération d'un domaine

- `values : t -> list Z`

Appartenance

- `member : Z -> t -> bool`

En calculant la liste exhaustive des valeurs (values)

Enumération d'un domaine

- `values : t -> list Z`

Appartenance

- `member : Z -> t -> bool`

Theorem values_Sorted : ∀ d,
`Inv_t d -> Sorted (values d).`

Theorem values_NoDup : ∀ d,
`Inv_t d -> NoDup (values d).`

En calculant la liste exhaustive des valeurs (values)

Enumération d'un domaine

- `values : t -> list Z`

Theorem values_Sorted : `∀ d, Inv_t d -> Sorted (values d).`

Theorem values_NoDup : `∀ d, Inv_t d -> NoDup (values d).`

Appartenance

- `member : Z -> t -> bool`

Theorem member_correct :
`∀ v d, Inv_t d -> member v d = true <-> In v (values d).`

En calculant la liste exhaustive des valeurs (`values`)

En calculant la liste exhaustive des valeurs (`values`)

Opérations élémentaires

- `process_size : elt_list -> Z`
- `process_max : elt_list -> Z`
- `get_min : elt_list -> Z -> Z`
- `is_empty : t -> bool`

Itération sur un domaine

- `fold_inter` et `fold_right`

En calculant la liste exhaustive des valeurs (values)

Opérations élémentaires

- `process_size : elt_list -> Z`
- `process_max : elt_list -> Z`
- `get_min : elt_list -> Z -> Z`
- `is_empty : t -> bool`

Itération sur un domaine

- `fold_inter` et `fold_right`

Theorem min_elt_list_values :

```
∀ l y, Inv_elt_list y l ->
get_min l y =
min_list y (elt_list_values l).
```

En calculant la liste exhaustive des valeurs (values)

Opérations élémentaires

- `process_size : elt_list -> Z`
- `process_max : elt_list -> Z`
- `get_min : elt_list -> Z -> Z`
- `is_empty : t -> bool`

Itération sur un domaine

- `fold_inter` et `fold_right`

Theorem min_elt_list_values :

```
∀ l y, Inv_elt_list y l ->
get_min l y =
min_list y (elt_list_values l).
```

Theorem size_elt_list_values:

```
∀ l y, Inv_elt_list y l ->
process_size l =
Zlength (elt_list_values l).
```

En calculant la liste exhaustive des valeurs (values)

Opérations élémentaires

- `process_size : elt_list -> Z`
- `process_max : elt_list -> Z`
- `get_min : elt_list -> Z -> Z`
- `is_empty : t -> bool`

Itération sur un domaine

- `fold_inter` et `fold_right`

Theorem min_elt_list_values :

```
∀ l y, Inv_elt_list y l ->
get_min l y =
min_list y (elt_list_values l).
```

Theorem size_elt_list_values:

```
∀ l y, Inv_elt_list y l ->
process_size l =
Zlength (elt_list_values l).
```

Theorem fold_correct :

```
∀ (e:A) l, Inv_elt_list y l ->
fold_right e l = List.fold_right
f e (elt_list_values l).
```

En calculant la liste exhaustive des valeurs (`values`)

En calculant la liste exhaustive des valeurs (values)

Domaines de base

- `empty : t`
- `boolean : t`
- `interval : Z->Z->option t`
- `singleton : Z -> t`

Existence

- `exists_t : (Z->bool)->elt_list->bool`

Theorem boolean_correct :
`values (boolean) = [0;1].`

En calculant la liste exhaustive des valeurs (values)

Domaines de base

- `empty : t`
- `boolean : t`
- `interval : Z->Z->option t`
- `singleton : Z -> t`

Existence

- `exists_t : (Z->bool)->elt_list->bool`

Theorem boolean_correct :
`values (boolean) = [0;1].`

Theorem interval_correct :
`∀d n p,`
`interval n p = Some d ->`
`∀ x, In x (values d) ->`
`n ≤ x ≤ p.`

En calculant la liste exhaustive des valeurs (values)

Domaines de base

- `empty : t`
- `boolean : t`
- `interval : Z->Z->option t`
- `singleton : Z -> t`

Theorem boolean_correct :
`values (boolean) = [0;1].`

Theorem interval_correct :
`∀d n p,`
`interval n p = Some d ->`
`∀ x, In x (values d) ->`
`n ≤ x ≤ p.`

Existence

- `exists_t : (Z->bool)->elt_list->bool`

Theorem exists_t_correct :
`∀ l, exists_t l = true <->`
`∃a, In a (elt_list_values l)`
`∧ p a = true.`

En montrant des propriétés algébriques

Suppression

- `remove : Z -> t -> t`

Etre inclus dans

- `included : t->t -> bool`

En montrant des propriétés algébriques

Suppression

- `remove : Z -> t -> t`

Etre inclus dans

- `included : t->t -> bool`

Theorem remove_correct_1 :

```
∀ d v, Inv_t d ->
member v (remove v d)=false.
```

Theorem remove_correct_2:

```
∀ d v y, Inv_t d ->
y ≠ v ->
member y d = true ->
member y (remove v d)=true.
```

En montrant des propriétés algébriques

Suppression

- `remove : Z -> t -> t`

Theorem remove_correct_1 :

```
∀ d v, Inv_t d ->
member v (remove v d)=false.
```

Theorem remove_correct_2:

```
∀ d v y, Inv_t d ->
y ≠ v ->
member y d = true ->
member y (remove v d)=true.
```

Etre inclus dans

- `included : t->t -> bool`

Theorem included_correct :

```
∀ d1 d2, Inv_t d1 ->
Inv_t d2 ->
included d1 d2 = true <->
(∀ x, member x d1 = true ->
member x d2 = true).
```

En montrant des propriétés algébriques

Création d'un domaine

- `create : list Z -> t`

Etre un domaine singleton

- `is_singleton : t -> Prop`

Obtenir une valeur

- `get_value : t->option Z`

En montrant des propriétés algébriques

Création d'un domaine

- `create : list Z -> t`

Etre un domaine singleton

- `is_singleton : t -> Prop`

Obtenir une valeur

- `get_value : t->option Z`

Theorem create_correct: `∀l, (∀ y, member y (create l) = true <-> In y l).`

En montrant des propriétés algébriques

Création d'un domaine

- `create : list Z -> t`

Etre un domaine singleton

- `is_singleton : t -> Prop`

Obtenir une valeur

- `get_value : t->option Z`

Theorem create_correct: `∀l, (∀ y, member y (create l) = true <-> In y l).`

Theorem is_singleton_correct : `∀ d, Inv_t d -> is_singleton d <-> size d = 1.`

En montrant des propriétés algébriques

Création d'un domaine

- `create : list Z -> t`

Etre un domaine singleton

- `is_singleton : t -> Prop`

Obtenir une valeur

- `get_value : t->option Z`

Theorem create_correct: `∀l, (∀ y, member y (create l) = true <-> In y l).`

Theorem is_singleton_correct :
`∀ d, Inv_t d -> is_singleton d <-> size d = 1.`

Theorem get_value_correct :
`∀ d v, Inv_t d -> get_value d = Some v -> member v d = true.`

Conclusion

- Traduction

Conclusion

- Traduction
  - 97 lignes de code OCaml de la bibliothèque FaCiLe traduite
  - 105 lignes de code Coq, dont 7 lignes de preuve de terminaison

Conclusion

- Traduction
  - 97 lignes de code OCaml de la bibliothèque FaCiLe traduite
  - 105 lignes de code Coq, dont 7 lignes de preuve de terminaison
- Spécification

Conclusion

- Traduction
  - 97 lignes de code OCaml de la bibliothèque FaCiLe traduite
  - 105 lignes de code Coq, dont 7 lignes de preuve de terminaison
- Spécification
  - 8 lignes de code pour définir les invariants
  - 13 lignes de preuves

Conclusion

- Traduction
  - 97 lignes de code OCaml de la bibliothèque FaCiLe traduite
  - 105 lignes de code Coq, dont 7 lignes de preuve de terminaison
- Spécification
  - 8 lignes de code pour définir les invariants
  - 13 lignes de preuves
- Preuve de correction

Conclusion

- Traduction
  - 97 lignes de code OCaml de la bibliothèque FaCiLe traduite
  - 105 lignes de code Coq, dont 7 lignes de preuve de terminaison
- Spécification
  - 8 lignes de code pour définir les invariants
  - 13 lignes de preuves
- Preuve de correction
  - 1749 lignes au total (preuves de bonne formation et préservation de l'invariant + preuves de correction)

Bilan quantitatif

- Objectifs atteints :

Bilan quantitatif

- Objectifs atteints :
  - Réalisation d'un état de l'art des représentations d'un domaine

Bilan quantitatif

- Objectifs atteints :
  - Réalisation d'un état de l'art des représentations d'un domaine
  - Traduction d'une partie de la bibliothèque FaCiLe

Bilan quantitatif

- Objectifs atteints :
  - Réalisation d'un état de l'art des représentations d'un domaine
  - Traduction d'une partie de la bibliothèque FaCiLe
  - Spécification et preuve de correction de la structure de liste d'intervalles

Bilan quantitatif

- Objectifs atteints :
  - Réalisation d'un état de l'art des représentations d'un domaine
  - Traduction d'une partie de la bibliothèque FaCiLe
  - Spécification et preuve de correction de la structure de liste d'intervalles
- Caractéristiques :

Bilan quantitatif

- Objectifs atteints :
  - Réalisation d'un état de l'art des représentations d'un domaine
  - Traduction d'une partie de la bibliothèque FaCiLe
  - Spécification et preuve de correction de la structure de liste d'intervalles
- Caractéristiques :
  - Travail modulaire

Bilan quantitatif

- Objectifs atteints :
  - Réalisation d'un état de l'art des représentations d'un domaine
  - Traduction d'une partie de la bibliothèque FaCiLe
  - Spécification et preuve de correction de la structure de liste d'intervalles
- Caractéristiques :
  - Travail modulaire
  - Code extrait : perte d'efficacité due à des modifications de l'algorithme de tri

Perspectives

Perspectives

- Intégration au solveur CoqbinFD

Perspectives

- Intégration au solveur CoqbinFD
- Poursuite du travail pour fournir une nouvelle implémentation de l'interface FSet

Merci de votre attention.

	Ensemble d'intervalles (ou range sequences)	**Vecteur binaire** (ou bit vectors)
Implémentation	Liste chaînée ABR	Tableau d $v \in$ D *ssi* $d[v] = 1$ ($d[v] = 0$ *sinon*)
Solveur	ECLiPSe Prolog SICStus Prolog	ILOG Solver
Complexité spatiale	$O(k)$	$O(\|\Omega\|)$
Complexité de recherche	$O(k)$ au pire	$O(1)$
Complexité de suppression	$O(k)$ au pire	$O(1)$

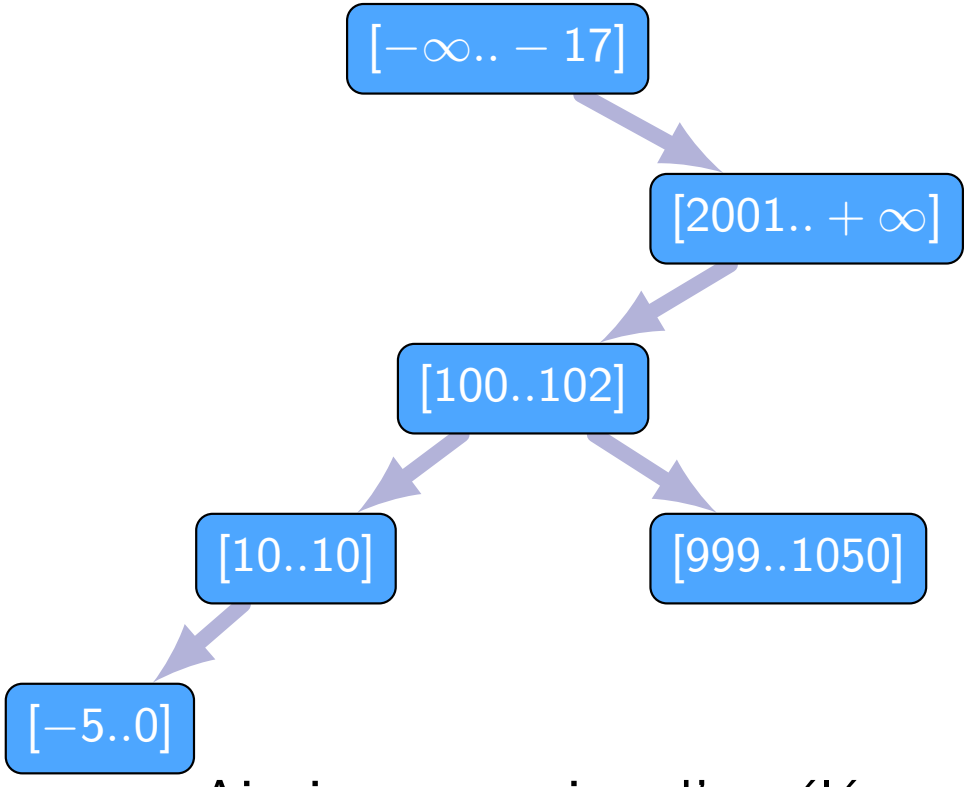
Notations : D : le domaine
Ω : univers tel que $\Omega = [min\ D\ ;max\ D]$
k : nombre d'intervalles dans D

	Gap intervals tree	**Ensemble creux**
Implémentation	ABR	2 tableaux dom_D et map_D
Solveur	Naxos Solver	Oscar et Castor
Complexité spatiale	O(k')	$O(\|\Omega\|)$ au pire
Complexité de recherche	O(k') au pire	$O(1)$
Complexité de suppression	O(k') au pire	presque $O(1)$

Notations : D : le domaine
Ω : univers tel que $\Omega = [min\ \mathrm{D}\ ;max\ \mathrm{D}]$
k' : nombre d'intervalles dans D

Représentation avec un "gap intervals tree"

= ABR contenant les écarts entre les intervalles initiaux.



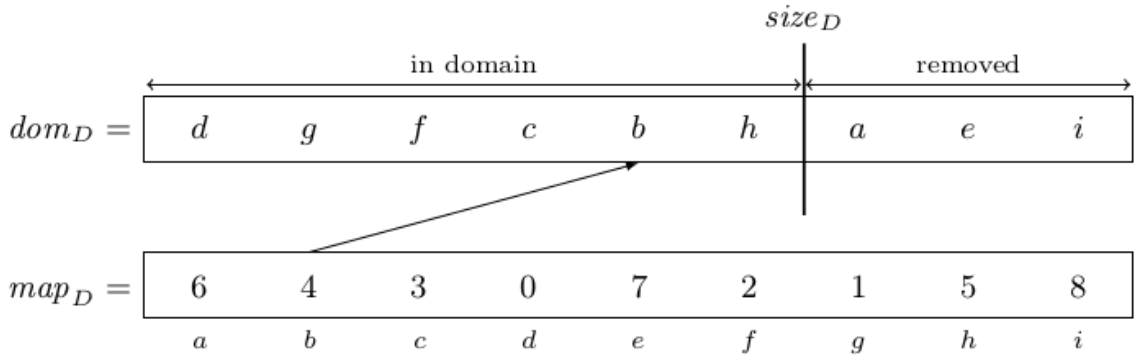
Représentation du domaine
$\{[-16..-6]; [1..9];$
$[11..99]; [103..998]; [1051..2000]\}$

$\rightarrow$ Ainsi, suppression d'un élément = insertion ou modification d'un seul nœud.

Représentation avec un ensemble creux ou clairsemé

Nous représentons un "ensemble clairsemé" avec 3 composantes :

- Un vecteur dom_D
- Un vecteur map_D
- Un nombre entier $size_D$ (6, ici)



Nous avons ainsi les propriétés suivantes :

- $D = \{dom_D[i] \mid 0 \leq i < size_D\}$
- Un vecteur nommé $map_D[v] = i \Leftrightarrow dom_D[i] = v$ donc $map_D[dom_D[i]] = i$
- Les valeurs appartenant à $dom_D[size_D..N]$ ne sont considérées par aucune opération.

Exemple avec le problème des n-reines

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ où x_i représente le numéro de la colonne où se place la dame de la ligne i;
- $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ où $\forall i \in \{1, .., n\}$, $\mathcal{D}_i = \{1, 2, ..., n\}$;
- $\mathcal{C} = C_1 \cup C_2 \cup C_3$ où
  $C_1 = \{x_i \neq x_j | i < j \text{ avec } (i,j) \in \{1, ..., n\}^2\}$,
  $C_2 = \{x_i - i \neq x_j - j | i < j \text{ avec } (i,j) \in \{1, ..., n\}^2\}$ et
  $C_3 = \{x_i + i \neq x_j + j | i < j \text{ avec } (i,j) \in \{1, ..., n\}^2\}$.

```
int n = 8;
Model model = new Model(n + "-queens problem");
IntVar[] vars = new IntVar[n];
for(int q = 0; q < n; q++){
    vars[q] = model.intVar("Q_"+q, 1, n);
}
for(int i  = 0; i < n-1; i++){
    for(int j = i + 1; j < n; j++){
        model.arithm(vars[i], "!=",vars[j]).post();
        model.arithm(vars[i], "!=", vars[j], "-", j - i).post();
        model.arithm(vars[i], "!=", vars[j], "+", j - i).post();
    }
}
Solution solution = model.getSolver().findSolution();
if(solution != null){
    System.out.println(solution.toString());
}
```

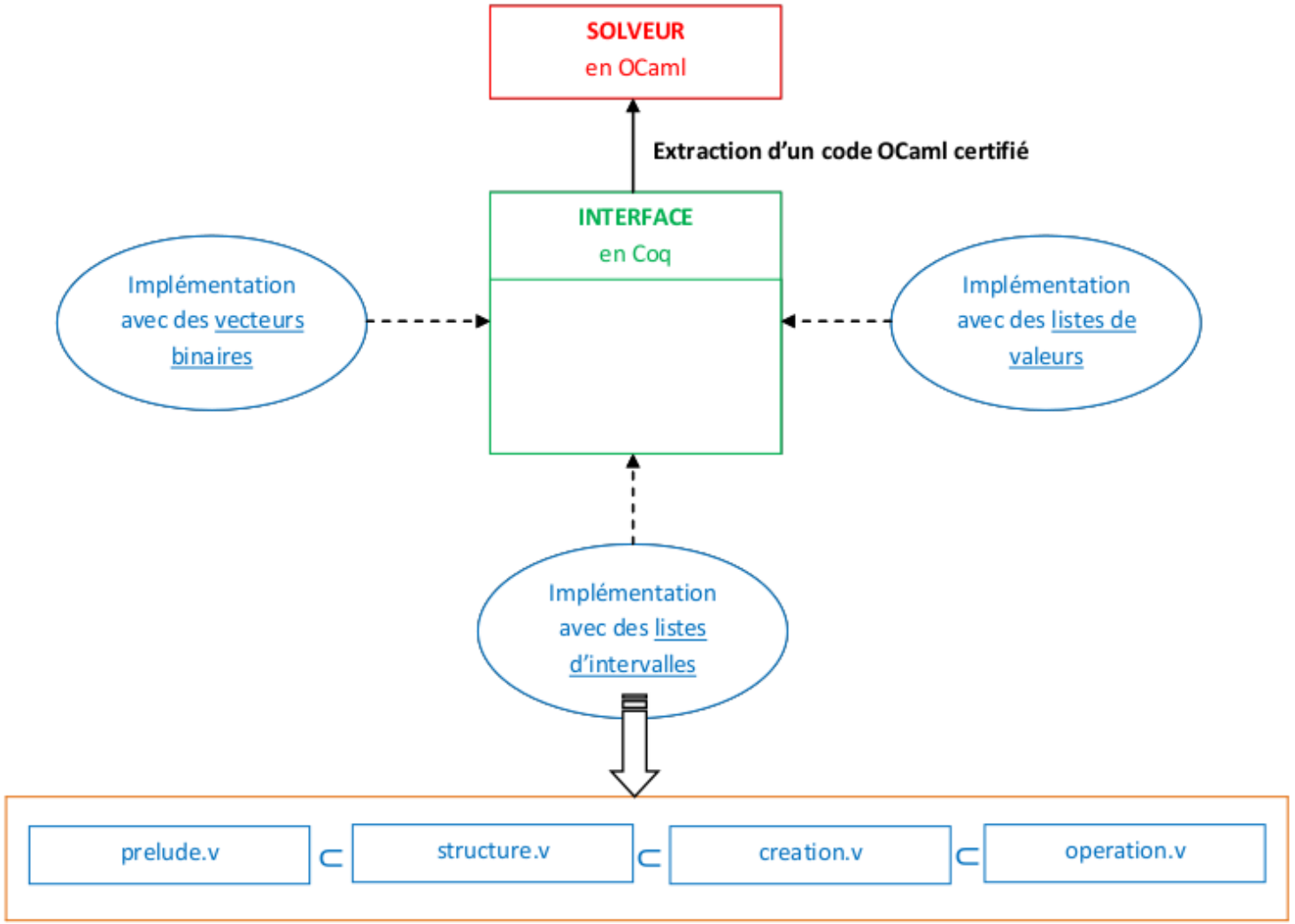
Fixation du paramètre du problème + nommage du problème

Création des inconnues du problème

Création des contraintes du problème

Résolution du problème

Figure: Résolution du problème des n-dames avec le solveur Choco



Inspiré de FaCiLe

= Bibliothèque de contraintes pour un langage fonctionnel typé (OCaml)
par Nicolas Barnier (CENA) et Pascal Brisset (ENAC)

- M. Carlier, C. Dubois, and A. Gotlieb. A certified constraint solver over finite domains. In Formal Methods (FM 2012), volume 7436 of LNCS, pages 116–131, Paris, France, August 2012.
- Nikolaos Pothitos and Panagiotis Stamatopoulos. Flexible management of large-scale integer domains in csps. In Stasinos Konstantopoulos, Stavros J. Perantonis, Vangelis Karkaletsis, Con- stantine D. Spyropoulos, and George A. Vouros, editors, Artificial Intelligence: Theories, Models and Applications, 6th Hellenic Conference on AI, SETN 2010, Athens, Greece, May 4-7, 2010. Proceedings, volume 6040 of Lecture Notes in Computer Science, pages 405–410. Springer, 2010.
- 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.