Vers les outils de test formellement vérifiés, de la résolution de contraintes à la génération de tests

Matthieu Carlier[1] Catherine Dubois[1,2] Arnaud Gotlieb[3]

1. CEDRIC-ENSIIE, Évry, France

2. INRIA, Paris, France

3. Certus V&V Center, SIMULA RESEARCH LAB., Lysaker, Norway

Génèse des travaux présentés : FocalTest

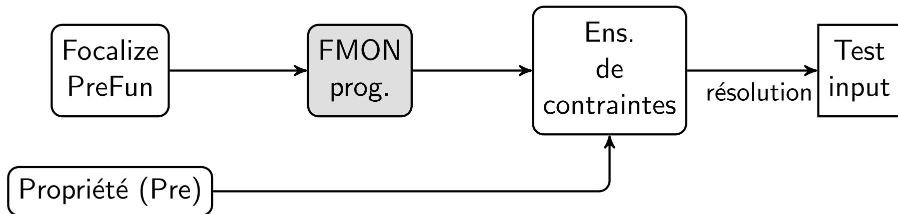
- Un outil automatique permettant la génération et l'exécution de tests, intégré à l'environnement de développement Focalize
  `http://focalize.inria.fr`
- Focus sur une ou plusieurs propriétés que le code (fonctionnel) doit satisfaire
- Propriété sous test est de la forme :

$$\forall X_1 \in T_1 \ldots X_m \in T_m, \underbrace{A_1 \Rightarrow \cdots \Rightarrow A_n}_{\texttt{Precondition}} \Rightarrow \underbrace{B_1 \vee \cdots \vee B_o}_{\texttt{Conclusion}}$$

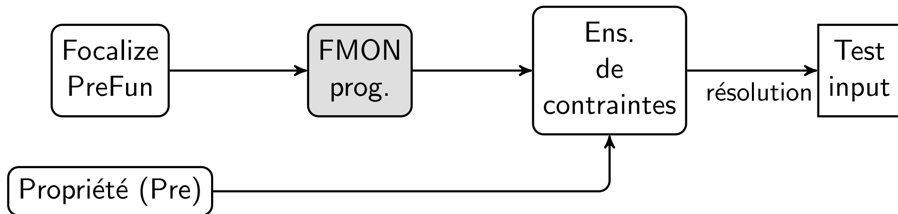
  A_i et B_i appels de fonctions (ou négations d'appels)
  - Un donnée de test : une valuation pour les X_is telle que Precondition est vraie (or fausse ou avec une couverture à la MC-DC des A_is)
    Comment ? transformation de Precondition en un ensemble de contraintes
    Une solution = une entrée de test
  - Verdict : évaluation de Conclusion $B_1 \vee \cdots \vee B_o$

.

Unité de test = propriété + un ensemble de fonctions - PreFun = fonctions concernées par le précondition.
FMON = langage intermédiaire



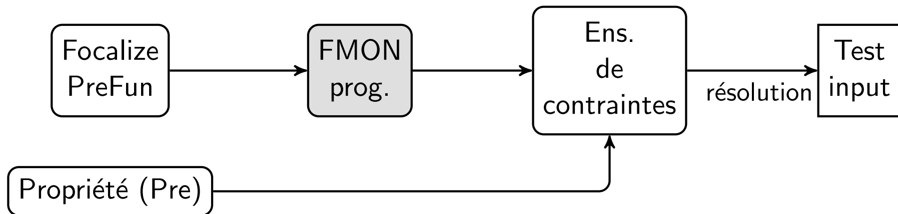
Unité de test = propriété + un ensemble de fonctions - PreFun = fonctions concernées par le précondition.
FMON = langage intermédiaire



L' ens. de contraintes capture la sémantique du programme initial.

C'est vrai ça ? Partie 1

Unité de test = propriété + un ensemble de fonctions - PreFun = fonctions concernées par le précondition.
FMON = langage intermédiaire



L' ens. de contraintes capture la sémantique du programme initial.

C'est vrai ça ? Partie 1

La résolution des contraintes utilise un solveur de contraintes.

Peut-on lui faire confiance ? Partie 2

Partie 1 : Prog. $\rightarrow$ Contraintes [TAP12]

Notre contribution

prouver la préservation sémantique de la traduction *prog. Focalize* $\rightarrow$ *contraintes*

- Preuve en deux temps : de Focalize à FMON, de FMON à contraintes ; Ici 2ème étape uniquement
- Preuve de correction vérifiée en Coq
- Extraction du traducteur

Les ingrédients :

- Sémantique opérationnelle de FMON
- Sémantique opérationnelle du langage de contraintes
- Définition de la fonction de traduction
- Théorème de correction
- Théorème de complétude

FMON

FMON : langage fonctionnel simplifié = variables attendues dans les arguments des appels de fonction, les match et la condition d'un if , filtrage de tête uniquement, pas d'ordre supérieur

Jugement d'évaluation

$$\mathcal{E}; \mathcal{E}_\mathrm{f} \vdash e \rhd v$$

- $\mathcal{E}_\mathrm{f}$ environnement des fonctions (associe une fermeture à tout identificateur de fonction *in* PreFun)

$$\mathcal{E}_\mathrm{f}(f) = <x_1, \ldots, x_n \leadsto e_f>$$

- $\mathcal{E}$ contexte d'évaluation (associe une valeur à chaque identificateur libre de e)
- v valeur (entier, booléen, valeur d'un type algébrique)

Le langage de contraintes

Syntaxe

- Valeurs : entières (FD) et valeurs algébriques (constructeurs)
  Variables : variables FD et variables algébriques algebraic variables
- Egalités et inégalités (bien typées) $X =_{fd} a, X =_{h} t$
- Contraintes utilisateurs `ite` et `match` (capturent les exp. conditionnelles er les filtrages)

  Informellement, la contrainte $\texttt{ite}(X, \sigma, \sigma')$ *où* X *est une variable algébrique, est satisfaite quand les contraintes de* σ *(resp.* σ'*) sont satisfaites si* X *est valuée à Ctrue (resp. Cfalse), Ctrue et Cfalse étant deux constantes spéciales.*

- Contraintes similaires à des appels de fonctions $\texttt{f}(X_1, \ldots, X_n)$
  $\equiv$ appel de prédicat Prolog.

- Cclosure $<X_1, \ldots, X_n, R \rightsquigarrow \sigma> =$ contrepartie de la fermeture fonctionnelle

  Eg Cclosure pour la fonction `app` (concaténation de deux listes) :

$$<L, G, R \rightsquigarrow \begin{array}{l} \texttt{match}(L, \\ \qquad [\texttt{pat}(L =_h nil, R =_h G) \\ \qquad \texttt{pat}(L =_h cons(H, T), R =_h cons(H, K), app(T, G, K)) \\ \qquad], \texttt{fail}) \end{array} >$$

```
type 'a list = nil | cons of 'a * 'a list;

let rec app(l, g) = match l with
            nil → g
          | cons(h, t) → let k = app(t, g) in cons(h, k);
```

Sémantique

Une solution d'un ensemble de contraintes : affectation totale telle que toutes les contraintes sont satisfaites.
$\Rightarrow$ sémantique définie comme une relation de satisfaction

Mais pb : appel de fonctions $\longrightarrow$ résoudre les contraintes relatives au corps de la fonction
$\longrightarrow$ nouvelles variables externes à l'affectation recherchée
$\Rightarrow$ prédicat de solution

$$\mathcal{A}; \mathcal{E}_{\texttt{cl}} \vdash \sigma \mapsto \mathcal{A}'$$

l'affectation $\mathcal{A}$ est étendue en l'affectation $\mathcal{A}'$, solution de l'ens. de contraintes σ en considérant l'environnement $\mathcal{E}_{\texttt{cl}}$

Traduction : FMON vers contraintes

- Dirigée par la syntaxe

$$\mathcal{T}_x, R \vdash_C e \longmapsto \sigma$$

  - $\mathcal{T}_x$: variable FMON *mapsto* variable de contrainte
  - R : variable associée au résultat de e

- Un exemple de règle de traduction :

$$\frac{\mathcal{F}reshC(X) \qquad \mathcal{T}_x; X \vdash_C e_1 \longmapsto \sigma_1 \qquad \mathcal{T}_x \oplus (x, X); R \vdash_C e_2 \longmapsto \sigma_2}{\mathcal{T}_x; R \vdash_C \texttt{let}\, x = e_1 \,\texttt{in}\, e_2 \longmapsto \sigma_1 \wedge \sigma_2}\ \text{LET}$$

Exemple

```
type 'a list = nil | cons of 'a * 'a list ;

let rec app(l, g) = match l with
        nil → g
        | cons(h, t) → let x1 = app(t, g) in cons(h, x1) ;

property rev_prop :
    all l l1 l2 : list(int), l = app(l1, l2) → ignoré
```

⟶ Précondition validée : $L =_h R, \texttt{app}(L_1, L_2, R)$

⟶ Précondition falsifiée : $L \neq_h R, \texttt{app}(L_1, L_2, R)$.

avec Cclosure pour app $<L, G, R \leadsto$

$$\texttt{match}(L, [\texttt{pat}(L =_h nil, R =_h G)\ \texttt{pat}(L =_h cons(H, T), R =_h cons(H, K), (\texttt{app}(T, G, K))\ ], \texttt{fail}) >.$$

Equivalence sémantique

Correction

Si $\mathcal{E};\mathcal{E}_\mathsf{f} \vdash e \rhd v$ et $\mathcal{T}_x; R \vdash_C e \longmapsto \sigma$ et $\mathcal{A} \models_{\mathcal{T}_x} \mathcal{E}$ alors il existe $\mathcal{A}'$ telle que $\mathcal{A};\mathcal{E}_{\texttt{cl}} \vdash R = v, \sigma \mapsto \mathcal{A}'$.

Vérifiée en Coq - 20 000 lignes de code (y compris toutes les définitions) - beaucoup de détails concernant les variables fraîches - a permis de trouver une erreur dans la spécification du prédicat de solution et d'injecter les informations de type minimales dans FMON.

Complétude

Si $\mathcal{A};\mathcal{E}_{\texttt{cl}} \vdash R = v, \sigma \mapsto \mathcal{A}'$ and $\mathcal{T}_x; R \vdash_C e \longmapsto \sigma$ et $\mathcal{A} \models_{\mathcal{T}_x} \mathcal{E}$ alors $\mathcal{E};\mathcal{E}_\mathsf{f} \vdash e \rhd v$.

Partie 2 : Solveur formellement vérifié [FM12]

Solveurs CP(FD) : efficaces, outils complexes

Confiance ? Crucial quand utilisés pour du logiciel critique.

Validation *a posteriori* :
- OK lorsque le solveur donne une solution, il suffit d'évaluer les contraintes ...
- mais ? ? ? ? quand le solveur répond `UNSAT`

Notre réponse : utiliser un solveur correct

Notre contribution

- Un solveur CP(FD) formellement vérifié
- Prouvé correct et complet
- Générique, paramétré par le langage de contraintes
- Mettant en œuvre un algorithme classique AC3 (Mackworth 77) au coeur de nombreux solveurs existants, reposant sur la consistance d'arc
- Purement fonctionnel, écrit en OCaml, extrait d'un développement en Coq.

Extraction de programmes - un petit intermède

Assistants à la preuve (Coq, HOL4 etc.)
peuvent être utilisés pour vérifier du code ≈ ML/Haskell

```
Definition f (n: nat) :=
n+2.

Theorem th : forall n,
(f n) > n.
```

Langages de prog. fonct. (OCaml, Haskell)
exécution efficace, prog. autonomes

```
let f n = n + 2;;
```

Extraction de programme

- convertir des fonctions Coq en syntaxe OCaml (avec effacement des aspects logiques si besoin)
- pratique de plus en plus courante (Compcert)
- hyp : sémantiques correspondent !

Définition d'un CSP

CSP : Constraint Satisfaction Problem

Un CSP (ou réseau de contraintes) est un triplet (X, C, D) où

- X : un ensemble de variables,
- C : un ensemble de contraintes (relations) sur les variables de X,
- D : une fonction qui associe à chaque variable de X son domaine (ensemble fini des valeurs possibles).

Une solution de (X, C, D) est une affectation des variables de X compatible avec D et qui satisfait les contraintes de C.

Un système de contraintes est insatisfiable s'il n'a pas de solution.

Ici : contraintes binaires et normalisées (2 contraintes différentes ont au plus une variable en commun)

Solveur générique

paramétré par :

- type *variable* + ordre et égalité décidable
- type *value* + égalité décidable
- type *constraint* + 2 fonctions
  - *interp* : $constraint \rightarrow value \rightarrow value \rightarrow bool$.
  (* donne la sémantique des contraintes *)

  - *get_vars* : $constraint \rightarrow variable \times variable$.
  telle que $\forall\ c\ x1\ x2,\ get_vars\ c = (x1, x2) \rightarrow x1 < x2$.

Ces types et fonctions sont à définir directement par l'utilisateur en OCaml.

Formalisation Coq d'un CSP

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

avec *mapdomain* : type des tables indexées par des variables avec des valeurs de type *list value*, sans doublons - instance du module Coq map.

Propriétés de bonne formation d'un réseau de contraintes :

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

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

Le domaine de la map contient les variables du csp, et uniquement celles-ci

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

Les variables qui apparaissent dans les contraintes sont des variables du csp

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

Toute variable du csp apparaît dans une contrainte au moins.

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

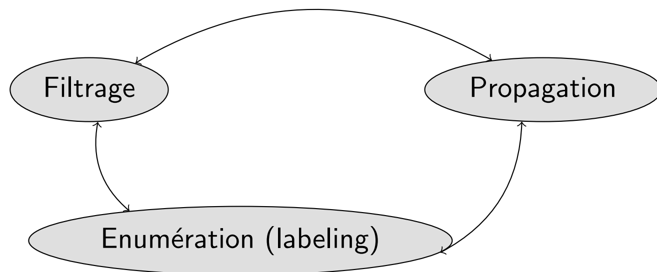
Le csp est normalisé

}.

Résolution

Idée principale des algos de résolution CP(FD) = supprimer les valeurs inconsistentes des domaines, de manière itérative

3 processus entrelacés :



Filtrage par arc-consistance

Definition

$c(x, y)$ est arc-consistante pour (X, C, D) ssi pour toute valeur $u \in D(x)$, il existe au moins une valeur (support) $v \in D(y)$ telle que c est satisfaite. Et vice-versa pour y et x.

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

$D(x)$		$D(y)$
1		1
2	- - - - - - - support de x=2 - - - - - - -	2
3		3
4		4

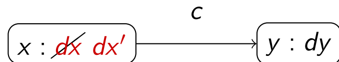
$c \equiv x > y$ non arc-consistante

$D(x)$	$D(y)$
1	1
2	2
3	3
4	4

Pas de support pour x=1

revise = fonction qui réduit le domaine d'une variable relativement à une contrainte.

- - x et y sont les variables de c, $dx = \mathrm{D}(x)$, $dy = \mathrm{D}(y)$
revise c x y dx dy = (dx', b)
- - si b alors dx' est le nouveau domaine de x, $dx' \subsetneq dx$
sinon $dx'{=}dx$.



revise c x y dx dy = (b, dx')

Correction et Complétude de *revise*

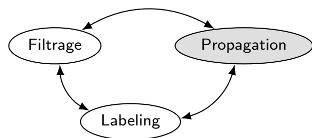
Après la révision de dx=D(x) pour la contrainte $c(x, y)$, toutes les valeurs de dx' ont un support dans dy pour c

```
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)).
```

revise ne perd pas de solution

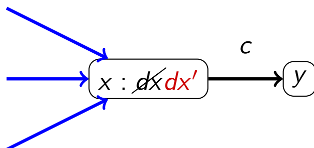
```
Theorem revise_complete : ∀ csp c x y dx dy (a : assign) ,
  network_inv csp → c ∈ (Csts csp) → compat_var_const x y c →
  find x (Doms csp) = Some dx → find y (Doms csp) = Some dy →
  solution a csp →
    ∀ newdx, revise c x y dx dy = (true, newdx) →
      solution a (set_domain x newdx csp).
```

Algo. de propagation AC3



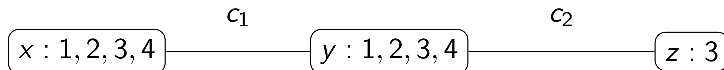
Application du filtrage sur toutes les contraintes du csp jusqu'à obtention d'un point fixe.

Idée principale de AC3 = maintenir l'ensemble des arcs à revoir (un arc = une contrainte et une variable dont il faut *réviser* le domaine)
= ceux dont la consistance a pu être affectée.



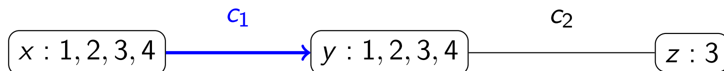
AC3 en action

Soient x, y, $z \in X$ des variables entières telles que
$D(x) = D(y) = \{1, 2, 3, 4\}$ et $D(z) = \{3\}$.
Considérons les contraintes $c_1 \equiv x < y$, $c_2 \equiv y \neq z$,



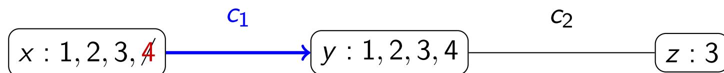
AC3 en action

Soient x, y, $z \in X$ des variables entières telles que $D(x) = D(y) = \{1, 2, 3, 4\}$ et $D(z) = \{3\}$.
Considérons les contraintes $c_1 \equiv x < y$, $c_2 \equiv y \neq z$,



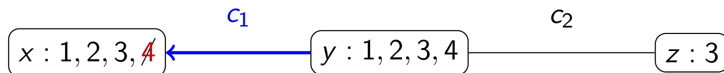
AC3 en action

Soient x, y, $z \in X$ des variables entières telles que $D(x) = D(y) = \{1, 2, 3, 4\}$ et $D(z) = \{3\}$.
Considérons les contraintes $c_1 \equiv x < y$, $c_2 \equiv y \neq z$,



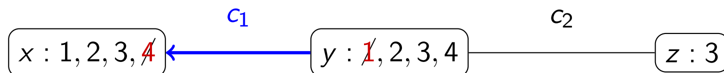
AC3 en action

Soient x, y, $z \in X$ des variables entières telles que $D(x) = D(y) = \{1, 2, 3, 4\}$ et $D(z) = \{3\}$.
Considérons les contraintes $c_1 \equiv x < y$, $c_2 \equiv y \neq z$,



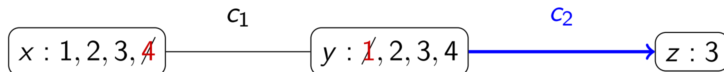
AC3 en action

Soient x, y, $z \in X$ des variables entières telles que $D(x) = D(y) = \{1, 2, 3, 4\}$ et $D(z) = \{3\}$.
Considérons les contraintes $c_1 \equiv x < y$, $c_2 \equiv y \neq z$,



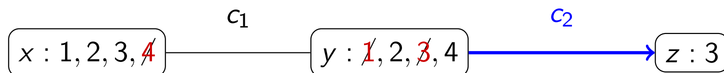
AC3 en action

Soient x, y, $z \in X$ des variables entières telles que
$D(x) = D(y) = \{1, 2, 3, 4\}$ et $D(z) = \{3\}$.
Considérons les contraintes $c_1 \equiv x < y$, $c_2 \equiv y \neq z$,



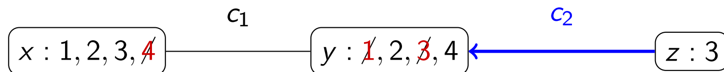
AC3 en action

Soient x, y, $z \in X$ des variables entières telles que $D(x) = D(y) = \{1, 2, 3, 4\}$ et $D(z) = \{3\}$.
Considérons les contraintes $c_1 \equiv x < y$, $c_2 \equiv y \neq z$,



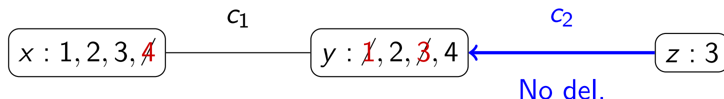
AC3 en action

Soient x, y, $z \in X$ des variables entières telles que
$D(x) = D(y) = \{1, 2, 3, 4\}$ et $D(z) = \{3\}$.
Considérons les contraintes $c_1 \equiv x < y$, $c_2 \equiv y \neq z$,



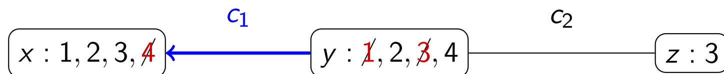
AC3 en action

Soient x, y, $z \in X$ des variables entières telles que $D(x) = D(y) = \{1, 2, 3, 4\}$ et $D(z) = \{3\}$.
Considérons les contraintes $c_1 \equiv x < y$, $c_2 \equiv y \neq z$,



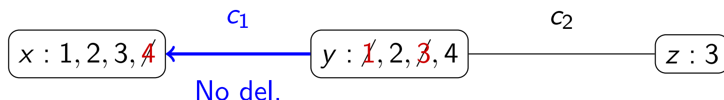
AC3 en action

Soient x, y, $z \in X$ des variables entières telles que $D(x) = D(y) = \{1, 2, 3, 4\}$ et $D(z) = \{3\}$.
Considérons les contraintes $c_1 \equiv x < y$, $c_2 \equiv y \neq z$,



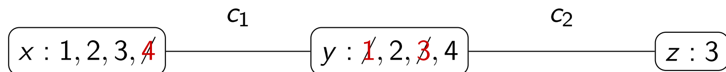
AC3 en action

Soient x, y, $z \in X$ des variables entières telles que $D(x) = D(y) = \{1, 2, 3, 4\}$ et $D(z) = \{3\}$.
Considérons les contraintes $c_1 \equiv x < y$, $c_2 \equiv y \neq z$,



AC3 en action

Soient x, y, $z \in X$ des variables entières telles que $D(x) = D(y) = \{1, 2, 3, 4\}$ et $D(z) = \{3\}$.
Considérons les contraintes $c_1 \equiv x < y$, $c_2 \equiv y \neq z$,



Toutes les contraintes sont arc-consistantes.

⚠ Arc-consistance n'est pas suffisante pour déterminer la réponse (une solution ou UNSAT). $x = 2$ $y = 2$ $z = 3$ n'est pas une solution !

Définition de AC3

```
Function AC3 (g : graph constraint) (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 g (add x dx' doms) (r ⊕ (incidentTo x y g))
      else AC3 g doms r
  | _, _ ⇒ None
  end
end.
```

g : représentation de la liste des contraintes sous forme d'un graphe (noeuds = variables - un arc étiqueté par $c(x, y)$ entre x et y).

Terminaison de AC3

2 mesures :

- sur l'ensemble des arcs à revoir qu = nombre d'éléments,
- sur la table des domaines $doms$ = somme des longueurs des domaines.

Les arguments des appels récursifs décroissent strictement selon ces mesures.

Correction et Complétude de AC3

Après application de AC3, toutes les contraintes sont arc-consistantes.

Theorem *AC3_sound* : $\forall$ *csp d'*,
network_inv csp $\rightarrow$
AC3 (*Csts csp*) (*Doms csp*, *complete_graph* (*Csts csp*)) = *Some d'* $\rightarrow$
$\forall$ *x y c*, (*x*, *c*, *y*) $\in$ (*complete_graph* (*Csts csp*)) $\rightarrow$
arc_consistent x y c d'.

Repose sur la preuve de l'invariant : si un arc n'est pas consistent pour *d* alors il est dans l'ensemble des arcs à revoir

Definition *PNC csts* (*d* : *mapdomain*) (*l* : *list arc*) : Prop := $\forall$ *x y c*,
(*x*, *c*, *y*) $\in$ (*complete_graph csts*) $\rightarrow$ $\neg$(arc_consistent *x y c d*) $\rightarrow$
(*x*, *c*, *y*) $\in$ *l*.

AC3 ne perd aucune solution

`Theorem` *AC3_complete* : $\forall$ *csp* (*a* : *assign*) *d'*,
network_inv csp $\rightarrow$ *solution a csp* $\rightarrow$
AC3 (*Csts csp*) (*Doms csp*, (*complete_graph* (*Csts csp*))) = *Some* (*d'*) $\rightarrow$
solution a (*set_domains d' csp*).

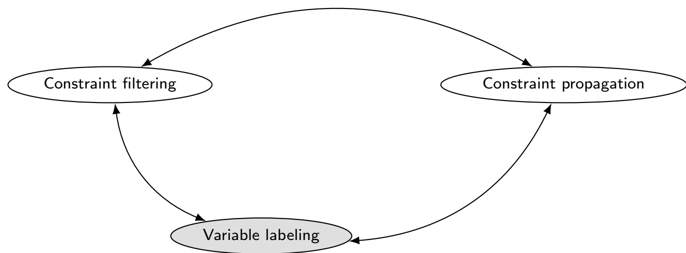
Labeling

AC3 permet d'obtenir un CSP arc-consistant.

Si l'un des domaines devient vide, il n'y a pas de solution. Fin
Si tous les domaines sont des singletons, on a une solution. Fin

Mais ce n'est pas le cas général.

$\longrightarrow$ recherche systématique basée sur une énumération des variables entrelacée avec filtrage et propagation.



1. Appliquer AC3
2. Choisir une variable x non instanciée ($D(x)$ n'est pas un singleton)
3. Choisir une valeur $v \in D(x)$
4. Ré-établir l'arc-consistance en appliquant AC3 sur les contraintes concernant x
5. Si NOK (pas de solution avec $x = v$) choisir une autre valeur pour x et recommencer en 3 si possible, NOK sinon.
6. Si OK, reprendre en 2 si possible, OK sinon.

Ici : heuristiques de choix élémentaires.

$\longrightarrow$ procédure de recherche correcte et complète (prouvé en Coq)

Conclusion et Perspectives

→ Développement d'un solveur de contraintes pour les domaines finis correct et complet (le premier)

- pour tout langage de contraintes binaires
- permettant de certifier l'absence de solutions.

→ 8500 lignes de Coq
Code Ocaml extrait opérationnel, utilisé sur différents types de problèmes.

→ Développement générique : propagation (paramétré par l'algo de filtrage) et recherche (paramétré par l'algo de propagation) = foncteurs

- consistance d'arc : AC3 et une variante AC2001
- consistance de bornes : algo. similaire à AC3

Perspectives

- Plus d'efficacité
  - structure de données plus efficaces, plus *impératives*
  - adaptation aux contraintes n-aires
  - intégration d'algorithmes spécifiques aux contraintes globales (ex. alldiff)
  - exploitation d'information sémantique
- Autres types de variables : flottants, types algébriques, pointeurs ...