

FaCiLe : librairie de contraintes pour un langage fonctionnel typé

Pascal Brisset

Laboratoire d'Optimisation Globale

ENAC / CENA

21 mars 2002

Plan

1	Choix	2
2	Fonctionnalités de FaCiLe	8
3	Mise en œuvre	29
4	Invariants	37
5	Benchmarks	42
6	TODO	43

1 Choix

- Quoi : une librairie documentée en Objective Caml
- Qui : Nicolas Barnier, Pascal Brisset
- Pour qui : ceux qui ne supportent plus C++ ou Prolog
- Pourquoi : parce que Choco n'existait pas encore : -)

`www.choco-constraints.net` :

- FaCiLe est petite : moins de 5000 lignes
- FaCiLe est extensible : briques de bases, fonctorisation
- FaCiLe est efficace : presque
- FaCiLe est simple : comme son nom l'indique
- FaCiLe est gratuite : licence LGPL

1.1 Langage

Éviter la « yet another language » approche

Objective Caml (caml.org) :

- langage fonctionnel fortement typé (pas de surcharge) ;
- compilateur natif ;
- porté sur de nombreuses plateformes ;
- licence à la LGPL ;
- efficace : www.bagley.org/~doug/shootout

1.2 Inspirations

- MALI, Machine Adaptée aux Langages Indéterministes (Irisa, Rennes)
- ECLiPSe (ECRC, IC-Parc)
- ILOG Solver

1.3 Licence LGPL

- Utilisable pour la recherche, l'enseignement et dans les applications industrielles.
- Documentée extensivement
- Distribuée sous forme source ou paquetage
- Platesformes : toutes celles supportées par OCaml (Linux, UNIXes, MacOS, Windows... / x86, Alpha, PPC, HPPA, IA64...)

2 Fonctionnalités de FaCiLe

2.1 Un aperçu

L'inévitable exemple : séquence magique

```
let xs = Fd.array n 0 (n-1)
let is_equal_to i x = fd2e x ==~~ i2e i
Array.iteri
  (fun i xi ->
    let cardi = Arith.sum (Array.map (is_equal_to i) xs) in
    Cstr.post (fd2e xi ==~ cardi))
  xs
Goals.solve (Goals.Array.forall Goals.indomain xs)
```


Légende

```
let xs = Fd.array n 0 (n-1)
```

- mot-clés et librairie standard d'OCaml en `courrier`
- valeurs prédéfinies de FaCiLe en **gras**
- valeurs préfixées par le nom du module de définition (e.g. **Fd**)
- pas de parenthèses autour des arguments en ML

Ici, construction (et non déclaration) d'un tableau de variables de domaines intervalles.

Typage fort

fd2e x =~~ i2e i

Variable vers expression **fd2e** : $Fd.t \rightarrow Arith.t$

Constante vers expression **i2e** : $int \rightarrow Arith.t$

Contrainte d'égalité =~ : $Arith.t \rightarrow Arith.t \rightarrow Cstr.t$

Contrainte réifiée =~~ : $Arith.t \rightarrow Arith.t \rightarrow Arith.t$

- Pas d'erreur à l'exécution ;
- pas d'ambiguïtés à la différence d'une solution avec surcharge.

Itérateurs

- Librairie standard d'OCaml :

$$\text{Array.map} : (\alpha \rightarrow \beta) \rightarrow \alpha \text{ array} \rightarrow \beta \text{ array}$$

- FaCiLe :

$$\text{Goals.Array.forall} : (\alpha \rightarrow \text{Goals.t}) \rightarrow \alpha \text{ array} \rightarrow \text{Goals.t}$$

Itération conjonctive d'un but sur un tableau.

Le langage nous offre la **quantification** et les **agrégats** :

- **Clôture**
- **Fonctionnelles**

—→ pas de syntaxe supplémentaire

—→ \approx langage de modélisation

Profil d'un programme utilisant FaCiLe

1. Création de domaine : **Domain.create**
2. Création de variable : **Fd.create**, **Fd.interval**
3. Création d'expressions : $+~$, **Arith.sum**
4. Création de contraintes : $<~$, **Alldiff.cstr**
5. Pose des contraintes : **Cstr.post**
6. Création de buts : **Goals.indomain**, **Goals.Array.forall**
7. Recherche : **Goals.solve**

Exécution d'un programme utilisant FaCiLe

1. *Toplevel* :

- apprentissage (enseignement)
- mise au point
- génération de la documentation

2. Compilation *bytecode* :

- rapidité de compilation
- mise au point : exécution pas à pas avec reprise
- portabilité

3. Compilation *native-code* :

- efficacité

2.2 Un tour complet

- Variables :
 - domaine entier,
 - domaine ensembliste (à la Conjunto),
 - intervalle flottant ;
- Contraintes :
 - arithmétiques, ensemblistes,
 - réifiées, globales,
 - utilisateurs ;
- Buts (recherche en profondeur d'abord) :
 - et, ou, itérateurs,
 - branch & bound,
 - utilisateurs.

2.2.1 Domaines

Domaines :

- entiers : liste d'intervalles ;
- ensemble d'entiers : intervalle de domaines d'entiers.

Représentation fonctionnelle :

- pas de modification en place ;
- partage possible.

2.2.2 Variables

Une variable est soit :

- instanciée : une valeur (entier, ensemble, intervalle, ...)
- inconnue : un domaine et des événements attachés (ensembles de contraintes)

Pour la gestion mémoire, il est essentiel que ces deux cas soient exclusifs.

C'est le pattern-matching qui permet de faire le test :

```
match Fd.value variable with  
  Val x -> ...  
| Unk attr -> ...
```

Quatre événements associés : instantiation, modification, min, max

2.2.3 Contraintes

- **Fonction** appelée lorsqu'un événement est *schédulé*
- Une contrainte peut-être :
 - postée
 - réveillée (propagation)
 - testée
 - niée
 - ...
- Priorités
- *Constraint store* (non racine d'accès, pointeurs faibles)

Contraintes arithmétiques

Égalité et inégalité d'expressions :

- variable
- constante
- somme, produit
- valeur absolue
- exponentiation
- division entière, modulo

Les expressions sont systématiquement normalisées : factorisation, ordonnancement des sous-expressions, ...

Les expressions booléennes (valeurs 0 ou 1) sont détectées et traitées spécialement.

Expressions et contraintes ensemblistes (Conjunto)

- cardinal
- union, intersection
- sous-ensemble, appartient
- inférieur
- tous-disjoints (globale)
- au-plus-un-en-commun (globale)
- ...

Contraintes réifiées

La plupart des contraintes peuvent être réifiées en variables booléennes.

```
let c = fd2e x =~ i2e 1729 in  
let b = Reify.boolean c in  
Cstr.post (fd2e y =~ i2e 42 *~ fd2e b)
```

Indirectement, opérateurs sur les contraintes :

```
(||~~) : Cstr.t -> Cstr.t -> Cstr.t
```

Contraintes globales

- **FdArray.get, FdArray.min, FdArray.max**
- **Alldiff** : « tous différents » (algo de bin-matching)
- **Gcc** : « cardinalité » [Régin, 96]
- **Sort** : « tri » [Colmerauer, 97]

Contrainte utilisateur

```
let different x y =
  let update _ =
    match (Fd.value x, Fd.value y) with
      | (Val a, Val b)      -> a <> b || Stak.fail "different"
      | (Val a, Unk attr_y) -> remove_value y attr_y a; true
      | (Unk attr_x, Val b) -> remove_value x attr_x b; true
      | _                    -> false

  and delay ct =
    Var.delay [Var.Attr.on_subst] x ct;
    Var.delay [Var.Attr.on_subst] y ct
  in
  Cstr.create update delay
```

2.2.4 Langage de buts

Un *but* est une clôture évaluée paresseusement par un *interpréteur* **Goals.solve**. Le type correspondant est abstrait dans le module **Goals**.

Conjonction et disjonction

La boucle d'énumération disjonctive des valeurs d'une variable se définit :

```
let enum x = ( Goals.indomain x &&~ Goals.fail ) ||~ Goals.success
```

Itérateurs À chaque structure de données est associée un *itérateur*.

FaCiLe fournit les itérateurs compatibles avec le contrôle des buts :

Goals.Array.forall : $(\alpha \rightarrow \mathbf{Goals.t}) \rightarrow \alpha \text{ array} \rightarrow \mathbf{Goals.t}$

D'où grâce à l'application partielle :

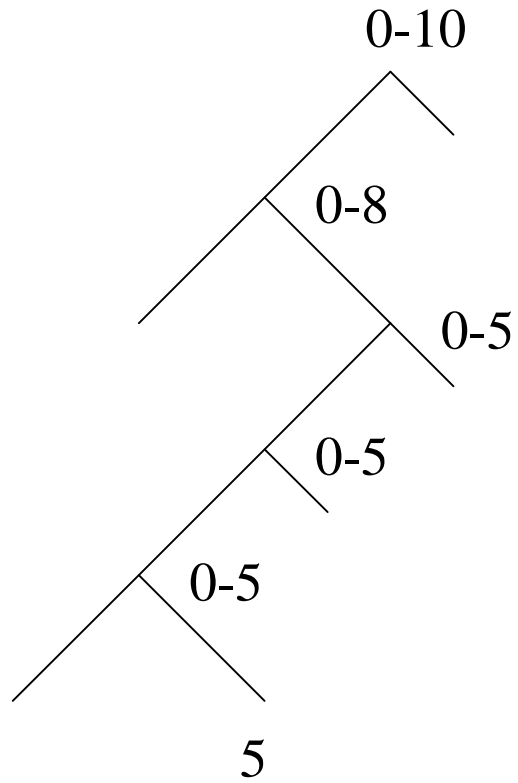
```
let labeling_array = Goals.Array.forall Goals.indomain
```

```
let labeling_matrix = Goals.Array.forall labeling_array
```

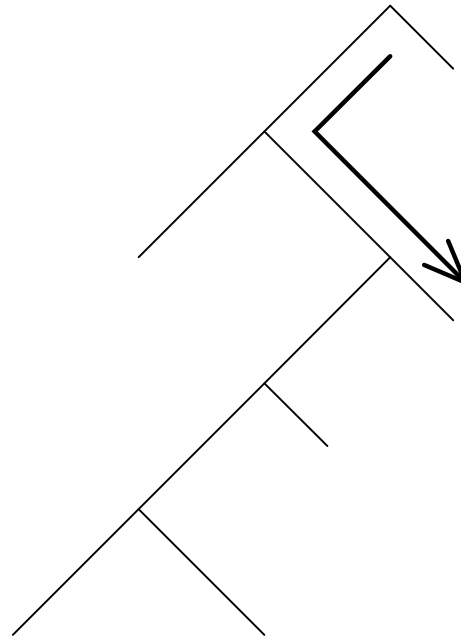

Optimisation par *branch & bound*

Recherche avec contrainte dynamique sur une variable de coût. Deux modes :

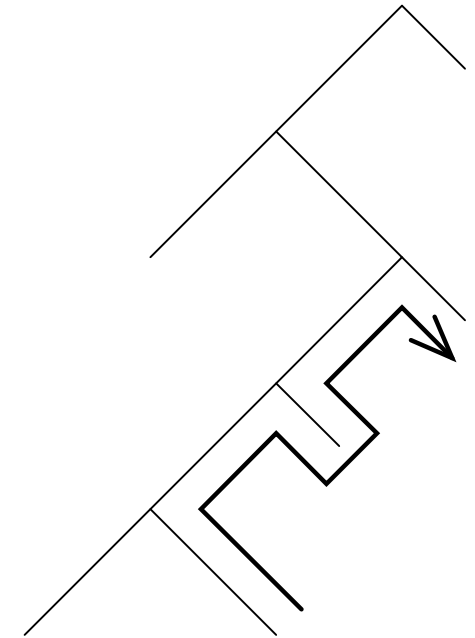
- redémarrage à la racine à chaque solution ;
- continuation à chaque solution.



Premiere solution



Restart



Continue

But utilisateur

1. Butification d'une fonction basique :

```
Goals.atomic (fun () -> print_string "Hello World")
```

2. But composé, éventuellement récursif :

```
let rec iter_disj fgoal list =
  Goals.create
    (function
      [] -> Goals.fail
      | x::xs -> fgoal x ||~ iter_disj fgoal xs)
  list
```

3. Cas particulier des but récursifs à argument constant

```
let repeat = Goals.create_rec (fun self -> Goals.success ||~ self)
```

À la Prolog

Suppression non-déterministe de l'élément d'une liste

```
delete([X | Xs], X, Xs).
```

```
delete([X | Xs], Y, [X | Ys]) :- delete(Xs, Y, Ys).
```

avec FaCiLe :

```
let rec delete l cont =
```

```
  Goals.create
```

```
    (function
```

```
      [] -> Goals.fail
```

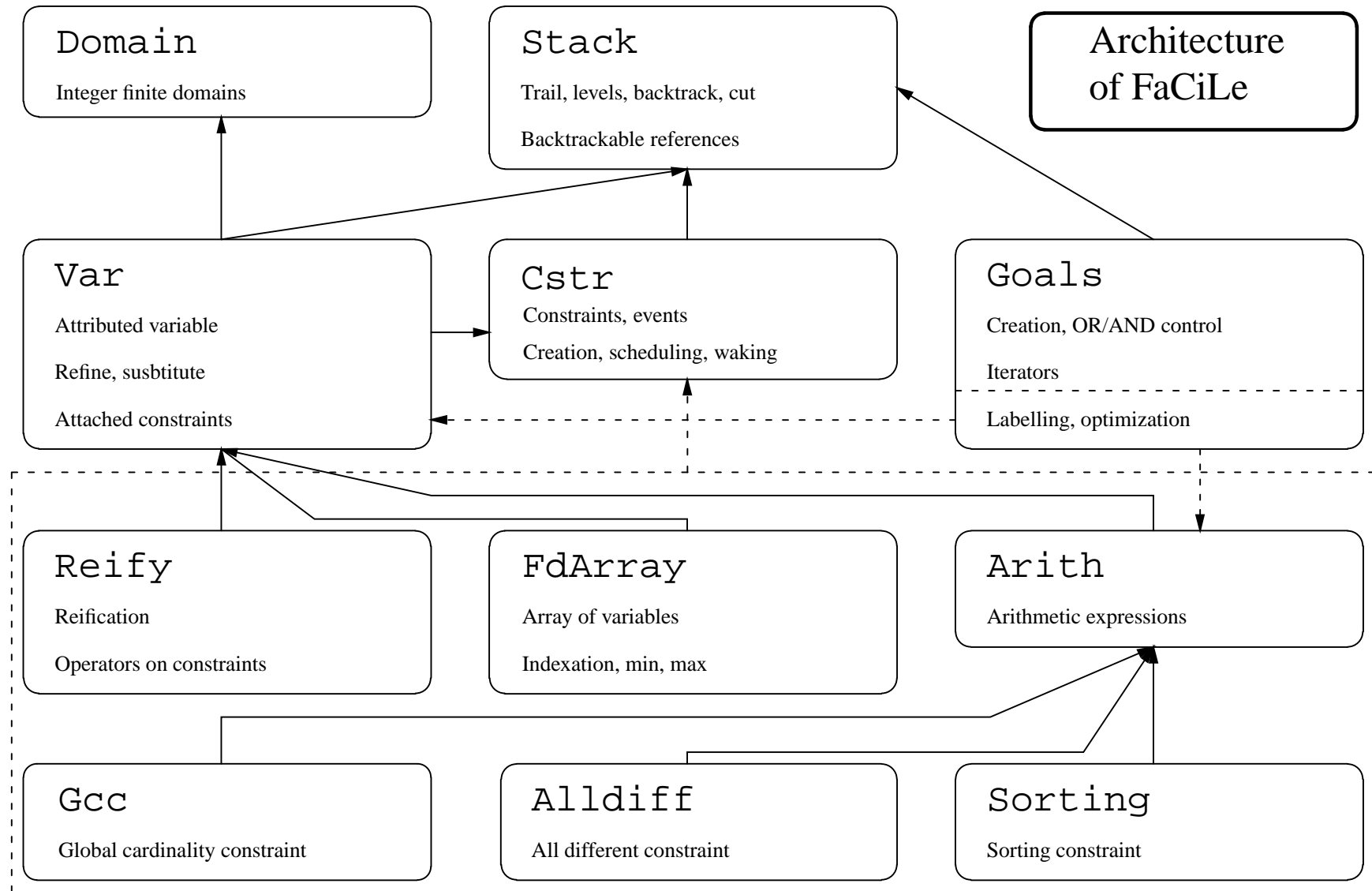
```
      | x :: xs ->
```

```
        cont x xs ||~ delete xs (fun y ys -> cont y (x :: ys))
```

```
    )
```

3 Mise en œuvre

- Moins de 5000 lignes d'OCaml
- Hiérarchie d'une douzaine de modules
- 1600 lignes de signatures (interfaces) documentées
- Encapsulation permettant de séparer signature interne (développeur) et externe (utilisateur)



3.1 Pile

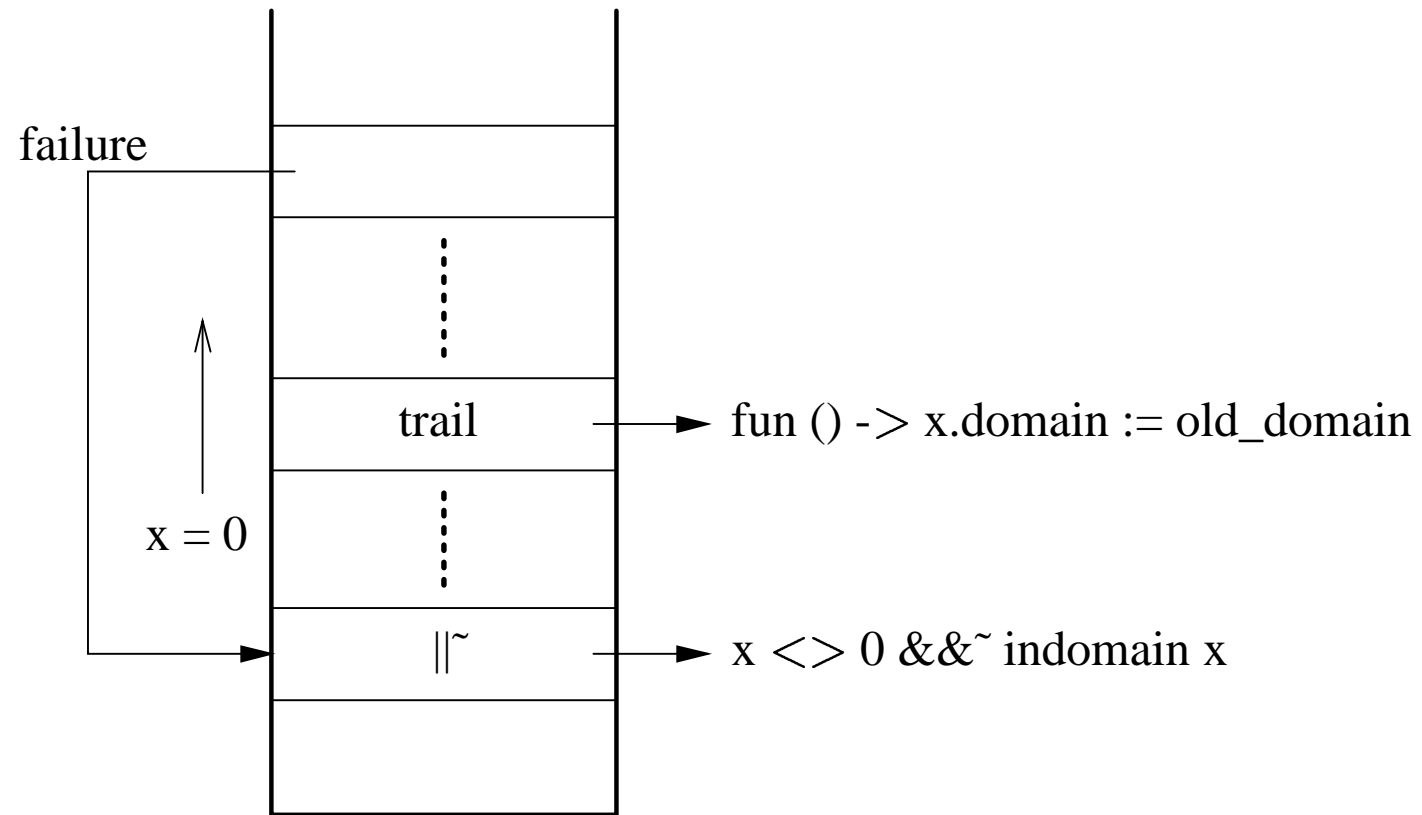
Continuation d'échec pour le retour-arrière :

- structurée en niveaux ;
- un niveau contient
 - une continuation de succès,
 - une *traînée* ;
- les éléments de la traînée sont des clôtures.

Opérations :

- sauvegarde ;
- retour-arrière ;
- coupure.

Le module fournit aussi un type de base : la référence *backtrackable*



3.2 Contrainte

Une collection de fonctions :

- **delay** : attachement à des événements après post
- **update** : propagations après réveil
- **check** (optionnel) : vérification (Oui, Non, NeSaitPas)
- **not** (optionnel) : négation
- **fprint** (optionnel) : affichage

Paramétrage du réveil :

- priorité associée à la contrainte
- identification associée à l'événement (e.g. indice de la variable concernée)

Exemple : contrainte sur un tableau de variables

```
let c = fun tab ->
  let n = Array.length tab in
  let update i = if i < n then ... else ...
  and delay ct =
    List.iteri
      (fun i tabi ->
        Var.delay [ Var.Attr.on_subst ] tabi ~waking_id:i ct;
        Var.delay [ Var.Attr.on_refine ] tabi ~waking_id:(i+n) ct)
      tab
  in
  Cstr.create ~priority:later ~nb_wakings:(2*n) update delay
```

Gestion des réveils :

- trois piles (trois priorités)
- à tout instant, un seul réveil possible par contrainte et par identificateur
- marquage des contraintes résolues

Les contraintes actives sont stockées (utile pour détecter le floundering).

La pile et le *constraint store* constituent un **état global**.

3.3 Recherche

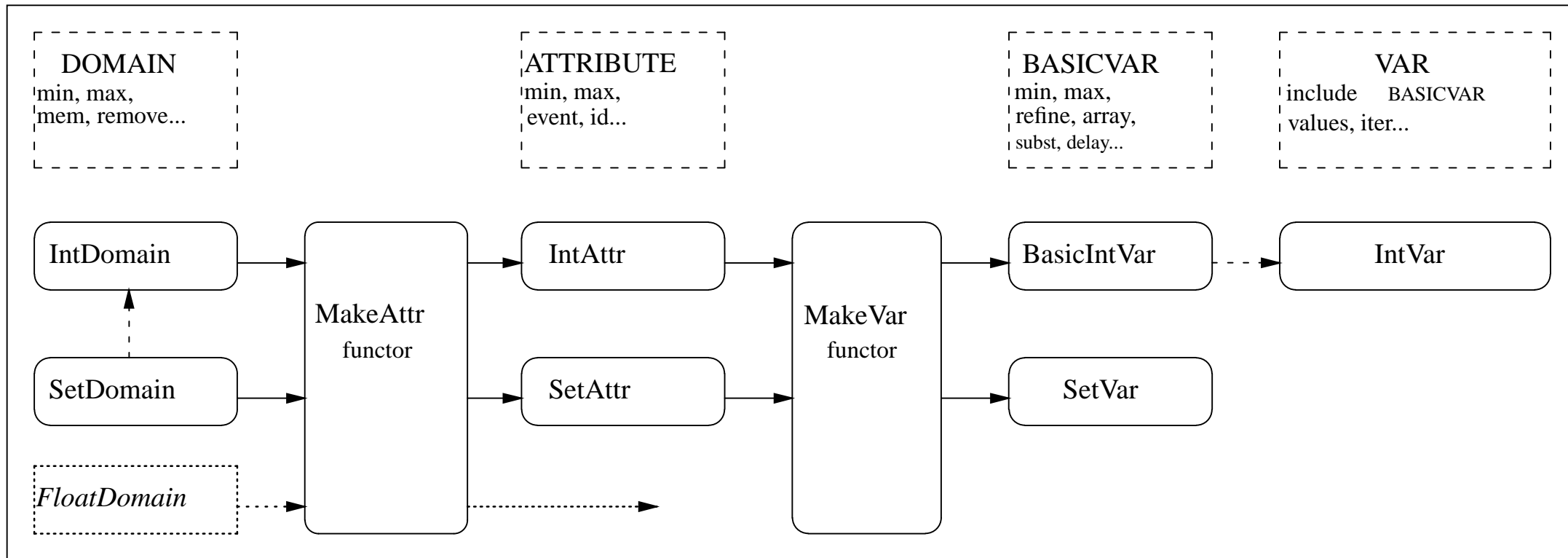
- Continuation de succès : une liste de clôtures
- Continuation d'échec : la pile

L'échec est contrôlé avec le mécanisme natif d'exception.

Sont associés au retour-arrière :

- un événement (utile pour le *branch&bound*) ;
- une fonction utilisateur (pouvant lever une exception).

3.4 Variables : functorisation



4 Invariants

À la Localizer [Van Hentenryck 97] : maintenance d'équations fonctionnelles pour la recherche locale

$$v = exp$$

où v est une variable et exp un expression faisant intervenir d'autres variables.

$$x = y + 3z$$

Une modification de y ou z doit entraîner une mise à jour de x

Les équations doivent être maintenues efficacement :

$$s = \sum_{i=1}^n x_i$$
$$p = \prod_{i=1}^n x_i$$
$$m = \max_{1 \leq i \leq n} x_i$$

Pour la modification d'un x_i , recalcul :

- s en $O(1)$
- p en $O(1)$ (sauf ancienne valeur à 0, $O(n)$)
- m en $O(\log n)$

4.1 Références invariantes dans FaCiLe

- Opérations de base : création, modification, accès
- Opérateurs prédéfinis :
 - somme, produit
 - indexation, argmin dans un tableau
- Encapsulation de fonction quelconque

$$\mathbf{Inv.unary} : (\alpha \rightarrow \beta) \rightarrow (\alpha \mathbf{Inv.t} \rightarrow \beta \mathbf{Inv.t})$$

- Liaison avec les variables à domaine : **Inv.Fd.min**, **Inv.SetFd.max**, ...

4.2 Utilisation des références invariantes

- Recherche locale pure
- Maintien de critère de choix pour la recherche
- Hybridation : à creuser...

4.3 Implémentation des références invariantes

- Une référence invariante :
 - une référence backtrackable
 - un événement activé lors d'une modification

NB : une référence invariante ne stocke pas d'« ancienne » valeur

- Une équation invariante :
 - une contrainte ... jamais résolue

Mécanisme précis de réveil → précision de la mise à jour

- Liaison avec les variables à domaines :
 - foncteur applicable à un module de variables

5 Benchmarks

Temps en secondes sur Ultra Sparc pour FaCiLe, Ilog Solver 4.3 et ECLiPSe 5.2.

Magic	100			200			400			800		
	0.25	0.05	1.5	1.0	0.15	6.1	4.9	0.48	26	21	2	#
Queens	16			32			64			128		
	0.03	0.03	10	0.07	0.04	∞	0.26	0.06	∞	1.2	0.2	∞
Golomb	6			7			8			9		
	0.04	0.04	0.18	0.33	0.11	1.2	3.3	0.75	11	30	7.0	99
Golf	8-4-9			5-3-4			5-3-6			5-3-7		
gcc	2.0	0.7	5.3	370	100	373	1660	400	2146	243	62	521
sort	2.1	–	9.5	380	–	14800	1740	–	11250	250	–	575
sets	1.6	1.1	22	390	96	2215	3000	380	26093	380	240	4380

6 TODO

- Release (ensembles, invariants, ...)
- Contraintes globales : *edge-finding*, ...
- Recherche : LDS, ...
- Compilation d'OPL vers FaCiLe
- Hybridation avec recherche locale (invariants)
- Programmation par intervalles pour les flottants
- Publicité

<http://www.recherche.enac.fr/opti/facile>