Sémantiques de Focal

C. Dubois, D. Delahaye

Foc, Focal, Focalize

`http://focalize.inira.fr`

langage de spécification, de preuve et d'implantation (Focal) + des outils = environnement Focal

ici focus mis sur le langage et sa sémantique formelle, plus particulièrement espèces, héritage, redéfinition

Plan

- Présentation informelle des composants du langage Focal
- Sémantique dénotationnelle à base d'enregistrements dépendants (thèse S. Boulmé - 2000)
- Sémantique par traduction : générateurs de méthodes (thèse de V. Prevosto - 2003)

Il existe également une sémantique à base de catégories (S. Boulmé) et une sémantique opérationnelle pour les aspects calculatoires uniquement (thèse de S. Fechter - 2005)

Les ingrédients de Focal

un langage fonctionnel pur à la ML + une pincée d'OO + propriétés FOL

- Espèces

Une espèce est un ensemble de fonctions et de propriétés : méthodes.

Une fonction peut être définie (`let`) ou déclarée (`signature`)

Une propriété peut être prouvée (`theorem`) ou déclarée (`property`)

Ces fonctions et propriétés sont définies sur un type support (`representation`) référencé par le mot-clé `Self`
Le type support est déclaré ou omis (type est alors abstrait) ou défini (à l'aide d'un type à la ML)

Expèce complète = espèce où tout est défini/prouvé

```
species setoid =
```

déclaration une relation binaire d'*égalité*

```
  signature equal : Self -> Self -> bool;
```

Spécification : `equal` est une relation d'équivalence

```
  property equal_reflexive : all x in Self, equal(x,x);
  property equal_symmetric : all x y in Self,
    equal(x,y) -> equal(y,x);
  property equal_transitive : all x y z in Self,
    equal(x,y) -> equal(y,z) -> equal(x,z);
```

Déclaration d'un élément (un sétoïde est non vide)

```
  signature elt :  Self;
```

Définition d'une relation d'*inégalité*

```
  let different (x,y) = basics#not_b(equal(x,y));
```

Définition d'un théorème : Zenon

```
  theorem same_is_not_different : all x y in Self,
     equal(x,y) -> not(different(x,y))
  proof: by  def different;
  end
```

- Héritage

Les espèces sont regroupées en une hiérarchie : une structure est souvent définie à partir de structures préexistantes en y ajoutant des opérations et/ou des propriétés.

Mécanisme d'héritage (`inherit`) : ajout de méthodes déclarées et/ou définies, définition de méthodes déclarées, redéfinition de méthodes déjà définies.

L'héritage peut être multiple : vérification de la cohérence des types pour les méthodes de mêmes noms, choix d'une définition (la dernière) pour les méthodes définies de mêmes noms.

Redéfinitions : pas "sans danger" pour les preuves qui dépendent des définitions !

setoid_with_one hérite de setoid

```
species setoid_with_one =
inherit setoid ;
```

Déclaration d'un élément one

```
 signature one : Self;
```

Définition de la méthode is_one

```
 let is_one(x) = equal(x,one);
```

Définition de l'élément elt

```
 let elt = one;
end
```

- Dépendances

Une méthode m_1 dépend d'une méthode m_2 si m_1 contient un appel à m_2.

La présence de propriétés conduit à distinguer deux types de dépendances :

- m_1 decl-dépend de m_2 si la définition de m_1 ne dépend que du type de m_2.
- m_1 def-dépend de m_2 si la définition de m_1 dépend de la définition de m_2.

Exemple : `same_is_not_different` def-dépend de `different`.
Si on redéfinit `different` dans une espèce qui hérite de `setoid`, la preuve de `same_is_not_different` doit être refaite.

- Interfaces

Interface $\approx$ espèce dont toutes les méthodes ne sont que déclarées.

Toute espèce doit pouvoir être associée à une interface ... ce qui peut poser des problèmes si on accepte des "def-dépendances" dans les "types" des méthodes.

Exemple Espèce *s* contenant les méthodes :

- type support : `representation=int`
- fonction `inc` de type `Self -> Self` définie par `function x -> x+1`
- théorème `inc_spec` prouvant *all x in* `Self`, $\texttt{inc}(x) \geq x + 1$

L'énoncé de `inc_spec` n'a de sens que si l'on sait que `representation=int` ... impossible de construire une interface.
$\rightsquigarrow$ Pas def-dépendances dans la déclaration d'une méthode.

Le compilateur vérifie qu'une interface peut être associée à chaque espèce.

- Collection

créé par abstraction de la représentation d'une espèce complètement définie.
c'est la structure qui permet de faire des calculs effectifs

- Espèces paramétrées

Les espèces peuvent être paramétrées par des collections (ou des entités, éléments d'une collection)

Espèce s paramétrée par une collection d'interface I : si l'interface de c est incluse dans I, $s(c)$ produit une espèce.

le produit cartésien de deux sétoïdes s_1 et s_2 est un sétoïde ...

```
species cartesian (s1 is setoid, s2 is setoid) =
inherit setoid ;
```

... dont le type support est le produit des types supports de s_1 et s_2 ...

```
  representation = s1*s2 ;
```

... muni d'un "constructeur" `term` ...

```
  let term(a,b) in Self = #crp(a,b) ;
```

... sur lequel l'égalité est définie à partir des égalités sur s_1 et s_2 ...

```
  let equal(a,b) = let a1 = #first(a) in let a2 = #scnd(a) in
                   let b1 = #first(b) in let b2 = #scnd(b) in
                   #and_b(s1!equal(a1,b1),s2!equal(a2,b2));
```

... et vérifie les propriétés souhaitées ...

```
proof of equal_reflexive = ...;
proof of equal_symmetric = ...;
proof of equal_transitive = ...;
```

... le produit cartésien est non-vide ...

```
let elt = #crp(s1!elt,s2!elt) ;
```

Sémantique dénotationnelle

On associe à chaque espèce un mixDrec = enregistrement dépendant pouvant contenir des champs définis et des champs déclarés.

Beaucoup de travaux sur les enregistrements dépendants depuis 2000 (Pollack, Luo etc.)

Codage en Coq réalisé par S. Boulmé disponible à l'adresse `http://www-lsr.imag.fr/Les.Personnes/Sylvain.Boulme/focal.html`

- enregistrement : nuplet étiqueté
  type enregistrement : type des enregistrements : associe à chaque étiquette son type
  Une collection peut être représentée par un enregistrement, son interface par un type d'enregistrement
- type enregistrement dépendant (signature) : type des enregistrements dont les champs peuvent avoir des types qui dépendent des valeurs des champs définis plus tôt
  exemple : $\{n = 2; v = [4; 5]\} : \{n : nat; v : vect(n)\}$

$$Rec : List(Lab) \rightarrow Type \qquad \frac{}{\{\} : Rec_{\emptyset}} \qquad \frac{T : Type,\ a \notin l,\ f : T \rightarrow Rec_l}{\{a : T; f\} : Rec_{a,l}}$$

- Relation de sous-signature $s :> s'$ si s contient au moins tous les labels de s avec le même type mais dans un ordre éventuellement différent

- Héritage simple : extension d'une signature s avec de nouveaux champs (à la fin car ils peuvent dépendre des types des champs de s)
- Héritage multiple : fusion de 2 signatures dont les champs communs ont le même type

MixDrec

- Enregistrement dépendant partiellement défini : MixDrec : champs déclarés et champs définis
- Arbre formé de 3 sortes de noeuds : noeuds vides (E), noeuds unaires abstraits (A), noeuds binaires manifestes (M)
  - Noeuds vides : feuilles de l'arbre
  - Noeuds abstraits : correspondent aux champs déclarés : un nom, un type, un fils (la suite du record)
  - Noeuds manifestes : correspondent aux champs définis : un nom, un type, une définition et deux fils.
    Dans le fils de gauche, la suite du record avec le champ abstrait
    Dans le fils de droite, la suite du record avec le champ défini
    Les deux fils contiennent les mêmes champs dans le même ordre.
    Mais certains noeuds abstraits à gauche peuvent être définis à droite.

```
                    A_equal
                       |
                    A_eqreflex
                       |
                    A_eqsym
                       |
                    A_eqtrans
                       |
                    A_elt
                       |
                    M_different
                /           \
A_same....                    M_same...
|                           /         \
E                         E            E
```

Ajoutons un théorème dans `setoid` ou dans une espèce qiu hérite de `setoid`

```
theorem diff_nrefl : all x in Self, not(different(x,x))
proof : by same_is_not_different, equal_reflexive.
```

```
MixDRec associé :            A_equal
                                |
                             A_eqreflex
                                |
                             A_eqsym
                                |
                             A_eqtrans
                                |
                             A_elt
                                |
                             M_different
                         /          \
            A_same....                M_same...
            |                       /        \
            M_diff_nrefl     M_diff_nrefl   M_diff_nrefl
           / |                      /\          /\
         E   E                     E  E        E  E
```

Opérations sur MixDrecs

- plongement d'un mixDrec dans une signature plus riche : on ajoute les champs non présents comme des noeuds abstraits
  si permutation, on est amené à faire des effacements
- fusion de 2 mixDrecs M1 et M2 de même signature
  - précédence de M1 sur M2
  - si x est défini dans M1 et M2, on retient la définition de M1 et on suit le fils gauche de x dans M2 (car on a modifié la définition de x) En effet sur cette branche les noeuds qui ont une def-depe vis-à-vis de x sont abstraits.
  - si x est abstrait dans M1, on ne doit pas pouvoir utiliser une définition donnée par M2 (il peut y avoir des def-dependances).