FOCAL

Réunion de l'ACI Alidecs
21 octobre 2004

Thérèse Hardin et Charles Morisset -LIP6

version 0.2 beta distribuée sur

`http://focal.inria.fr`

Les membres du projet FOCAL

LIP6-SPI : M. Jaume, R. Rioboo, T. Hardin (responsable scientifique)

CNAM-CEDRIC : C. Dubois, V. Viguié, D. Delahaye, O. Pons

INRIA Rocq. : D. Doligez, P.Weis (INRIA Rocq.)

Doctorants: S. Fechter, R. Bonichon, Y. Noyer, C. Morisset, J-F Etienne

Ex-doctorants : V. Prévosto, S. Boulmé

ACI Sécurité MODULOGIC (T. Hardin) et EDEMOI (Y. Ledru)

Objectifs

Spécifier, concevoir, développer un langage

"inspirant confiance"

Pour

Spécifier, concevoir, développer une application par une approche formelle .... restant pragmatique

Déterminer des besoins

Par l'étude de la construction d'une librairie manipulant des polynômes

Un exemple servant de test : calcul du résultant (pgcd) de :

$$P = x^{30} + ax^{20} + 2ax^{10} + 3a$$
$$Q = x^{25} + 4bx^{15} + 5bx^5$$

a entier variant entre 10^{100} et 10^{700} et $b = a + 1$.

Spécifier le langage

- **besoin de déclarations/définitions de fonctions, énoncés/preuves d'assertions regroupés dans une unité appelée *espèce*,**
- **⇒ Spécification en Coq de la notion d'espèce - S. Boulmé**
- **besoin de raffinement, héritage multiple, liaison tardive, paramétrisation par des espèces et des entités ("éléments") des espèces, pour construire de nouvelles espèces, d'abstraction pour utiliser correctement les espèces**
- **⇒ Spécification en Coq de ces traits - S. Boulmé par des enregistrements à champs dépendants.**
- **Conclusion : certaines dépendances peuvent conduire à des incohérences logiques.**

Concevoir le langage

- **Réalisation de prototypes de la librairie, en Ocaml, en suivant différentes approches**
- **⇒ Elaboration d'une discipline de programmation**
- **Validation de la discipline et tests d'efficacité**
- **Définition de restrictions garantissant la cohérence logique**

Définition d'une syntaxe concrète

- oblige à respecter la discipline de programmation
- oblige à éviter certaines formes de dépendances
- expression des propriétés par un langage de style premier ordre

.... Syntaxe peut encore être améliorée ...

Compilation de FOCAL

(Thèse de V. Prevosto)

- **Première passe :typage et analyse statique, détectant les dépendances prohibées.**
- **Seconde passe à quatre sorties**
  - **Source Ocaml**
  - **Source pour un langage intermédiaire (FocDoc) permettant de créer des fichiers de documentation automatique sous différents formats.**
  - **Chapitre Coq permettant de réaliser une preuve en Coq en disposant de toutes les propriétés connues**
  - **Source pour Zenon, outil de démonstration automatique, qui réalise automatiquement (autant que possible) la preuve et produit un terme-preuve pour Coq (D. Doligez)**

Espèces

Un ensemble d'outils, appelés méthodes, travaillant sur une représentation des données manipulées (parfum types algébriques)

1. **Première méthode, nommée `rep`, décrit la représentation choisie (une variable de type, un type (à la ML) plus défini).**

   Définition de `rep` : directement ou par héritage

2. **Déclarations `(sig)` donnant le nom et le type, hypothèses `(property)` (qui devront être démontrées). `self` dans un type désigne `self!rep`.**

3. **Définitions `(let)` donnant le nom et le corps des fonctions, `(letprop)` nommant une fonction qui retourne une proposition.**

4. **Preuves `(theorem)` constituées d'un énoncé et d'un script de preuve Coq ou Zenon.**

Paramétrisation

`species foo (r is ring, elt in r)`

Utilisation de `r`: `r!m` désigne une méthode de `r`.

`r` doit être instanciable par toute implantation respectant ce qui est dénoté par `ring`

Dans le corps de `foo`, est-il sain d'utiliser la définition de `r!rep`? celle des autres méthodes?

Non ⇒ `ring` ne doit fournir que le nom des méthodes ainsi que leur type dans lequel `r!rep` est abstrait.

Une `interface` est constituée des noms des méthodes et de leur type (ou énoncé) dans lequel le type de `rep` est abstrait.

Peut-on construire une interface pour toute espèce?

Paramétrisation

Soit une espèce s contenant les méthodes :

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

L'énoncé de `inc_spec` **n'a de sens que si l'on sait que** `rep=int` **...**

Impossible de construire une interface pour s

Le compilateur vérifie qu'une interface peut être associée à chacune des espèces.

Paramétrisation

`species foo (r is ring, elt in r) ....`

`foo (mon-r, mon-elt)`

`mon-r` doit posséder les méthodes présentes dans `ring`, leur type étant plus précis que celui demandé par `ring`.

Aucune méthode de `mon-r` ne doit être seulement déclarée: garantie de bonne exécution, construction d'un arbre de preuve.

Une collection est créée par abstraction de la représentation d'une espèce complètement définie (ce qui est vérifié par le compilateur).

`collection bar implements espece-ring`

`mon-r` désigne une collection.

Héritage multiple

- **vérification de la cohérence des types pour les méthodes de même nom**
- **choix d'une définition (la dernière) pour les méthodes définies de même nom.**

Redéfinitions

Pas "sans danger" pour les preuves qui dépendent des définitions

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

- **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.**

$\rightsquigarrow$ Pas de def-dépendances dans la déclaration d'une méthode

Demo

Confiance ?

- **Source Ocaml vérifié par le typeur Ocaml.**
- **Dans une collection, les propriétés sont obligatoirement démontrées.**
- **Une preuve peut être "assumed" mais cela est (sera) mentionné dans la documentation.**

  Dans ce cas, seule la construction de l'espèce est vérifiée par Coq: cohérence de l'héritage, des utilisations des paramètres, ...
- **Documentation produite automatiquement, garantissant une certaine cohérence avec l'implantation**

Conclusion

- **FOCAL : un langage permettant de spécifier, programmer et certifier des propriétés**
- **Passage de la spécification au code facilité par la généricité, le raffinement, l'héritage multiple, la redéfinition, la paramétrisation**
- **Gestion conjointe des déclarations, assertions, définitions et preuves**
- **Les espèces n'ont pas d'état interne, pas d'affectations.**
- **Compilation vers Ocaml: langage de programmation efficace**
- **Compilation vers CoQ permettant la vérification des preuves.**

FOCAL et Alidecs

Absence en FOCAL de prise en compte du temps

- **Quels sont les choix possibles pour pouvoir traiter des spécifications "concurrentes"? Que faut-il ajouter?**

Pistes :

- **traitement de la concurrence par des systèmes comme B ou ELAN ?**
- **intégration de "calculs de flot" et de "calculs dans l'instant" à la Lucid Synchrone ?**
- **intégration d'aspects réactifs ?**
- **Utilisation de logiques temporelles?**