L'atelier FOCAL

Catherine Dubois*, Mathieu Jaume**, Olivier Pons*, Virgile Prevosto***

* Institut d'Informatique d'Entreprise, laboratoire Cedric
18, allée Jean Rostand, 91025 Evry, France
{dubois, pons}@iie.cnam.fr
** Laboratoire d'Informatique de Paris VI
8, rue du Capitaine Scott, 75015 Paris, France
jaume@lip6.fr
*** Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Allemagne
prevosto@mpi-sb.mpg.de

Résumé

L'atelier FOCAL est un atelier intégré de construction modulaire de logiciels certifiés. Il permet l'écriture de modules constitués de déclarations, définitions, énoncés et preuves. Les déclarations peuvent être raffinées en définitions et les énoncés en preuves, par passage progressif de la spécification à l'implantation, à l'aide de mécanismes d'héritage, de redéfinition et d'instanciation de paramètres génériques. La compilation des modules développés avec l'atelier FOCAL produit un code exécutable OCaml et un développement formel Coq.

1 L'atelier FOCAL et ses outils

L'atelier FOCAL[1] (voir [1, 2] et la bibliographie proposée sur [4]), développé au sein des équipes SPI (LIP6), CPR (Cedric) et Cristal (INRIA), est un atelier intégré de construction modulaire de logiciels certifiés. Il permet l'écriture de modules constitués de déclarations, définitions, énoncés et preuves. Les déclarations peuvent être raffinées en définitions et les énoncés en preuves, par passage progressif de la spécification à l'implantation, à l'aide de mécanismes d'héritage, de redéfinition et d'instanciation de paramètres génériques.

L'atelier FOCAL constitue une interface privilégiée pour interagir avec un compilateur vers un langage de programmation, ici OCaml, et un assistant à la preuve, ici Coq. L'originalité de notre approche réside dans la réalisation d'une intégration formelle de ces outils.

L'atelier FOCAL et la méthodologie sous-jacente ont principalement été appliqués jusqu'à présent au calcul formel. C'est en fait ce domaine qui a servi de modèle et donné les principales lignes directrices du développement de l'atelier. Grâce à l'atelier FOCAL, R. Rioboo a pu construire une bibliothèque de calcul formel assez conséquente [3], comprenant des algorithmes complexes et dont l'efficacité est comparable à celle des meilleurs systèmes de calcul formel.

L'atelier FOCAL, disponible à l'adresse `http://www-spi.lip6.fr/foc`, comprend essentiellement un langage, le langage FOCAL, un compilateur qui produit conjointement du code exécutable OCaml et du code Coq à des fins de certification. L'atelier fournit également l'outil FOCDOC capable d'engendrer automatiquement la documentation au format XML.

Le développement de l'atelier FOCAL se poursuit dans le cadre du projet Modulogic[2] où l'accent est mis tout particulièrement sur la recherche de preuve en intégrant, entre autres, réécriture et déduction modulo.

2 Les concepts principaux du langage FOCAL

Dans cette partie, nous énumérons les concepts fondamentaux du langage FOCAL. Nous illustrons cette partie à l'aide d'extraits de la hiérarchie des listes triées dont le code complet se trouve à l'adresse `http://www-spi.lip6.fr/foc`.

Espèce. Les espèces sont les nœuds de la hiérarchie formée par les modules du développement. Une espèce définit un ensemble de valeurs appelées *entités* qui ont une représentation unique ainsi qu'un ensemble d'opérations applicables sur ces entités. Chaque opération est identifiée par son nom et peut être soit déclarée (et donc abstraite) ou définie (c'est-à-dire implantée). Une espèce définit également le type support introduit par `rep` qui est le type de la représentation des entités et enfin, côté propriétés, les énoncés et théorèmes (accompagnés d'une preuve). Opérations, type support, énoncés et preuves sont appelés méthodes.

[1] anciennement FOC

[2] ACI Sécurité des systèmes informatiques, `http://modulogic.inria.fr`

Héritage. Les espèces sont construites directement ou à partir d'espèces existantes par héritage. Ainsi lorsqu'une espèce `B` hérite d'une espèce `A`, elle hérite des méthodes de l'espèce `A`. Les méthodes déclarées dans `A` peuvent être définies dans `B`. Toute méthode définie, à l'exception du type support, peut être redéfinie dans une des espèces filles.

Par exemple, nous écrivons ci-dessous l'espèce `setoid` qui introduit la notion de setoïde, soit un ensemble muni d'une relation d'équivalence `equal`.

```
species setoid inherits basic_object =
 sig equal in self->self->bool;
 property equal_refl : all x in self, !equal(x,x)
 property equal_sym : all x, y in self,
  !equal(x,y) -> !equal(y,x);
 property equal_trans : all x, y, z in self,
  !equal(x,y) -> !equal(y,z) -> !equal(x,z);
end
```

L'espèce `setoid` hérite de la racine de la hiérarchie `basic_object` qui contient trois méthodes : le type support, **rep**, seulement déclaré à ce stade, et donc instancié au fil de l'héritage entre espèces et les deux utilitaires `parse` et `print`. L'espèce `setoid` introduit la méthode `equal` qui lui est propre. D'autre part, on énonce les propriétés qui traduisent que `equal` est une relation d'équivalence. Ici, `self` désigne le type support, plus exactement la version concrète du type support.

Ensuite, nous écrivons les espèces `setoid_ordre` et `ordre_total` qui enrichissent la structure de setoïde : la première munit le setoïde d'une relation d'ordre, la seconde contraint cette relation d'ordre à être un ordre total.

```
species setoid_ordre inherits setoid
 sig leq in self -> self -> bool
 property leq_refl : all x, y in self,
  !equal(x,y) -> !leq(x,y);
 property leq_sym : all x, y in self,
  !leq(x,y) -> !leq(y,x) -> !equal(x,y);
 property leq_trans : all x, y, z in self,
  !leq(x,y) -> !leq(y,z) -> !leq(x,z);
end

species ordre_total inherits setoid_ordre
 property leq_total : all x, y in self, !leq(x,y) or !leq(y,x);
end
```

Collection. Une *collection* est une espèce complètement définie : toute méthode doit être définie (code et preuve), tout paramètre instancié. Les collections sont les feuilles de la hiérarchie. De plus, une collection est vue au travers de son *interface* : le type support est caché et les entités de la collection ne sont manipulables que par le biais des opérations.

Paramètres. Une espèce peut être paramétrée par une entité ou une collection. Lorsque le paramètre est une collection, cette dernière doit satisfaire une certaine interface : elle doit alors offrir les méthodes précisées dans l'interface.

Définissons l'espèce des listes, paramétrée par une collection `a` d'interface `setoid`. Ici, nous avons choisi comme type support le type prédéfini

des listes, `list(a)` (`a` désigne dans cette définition le type support de la collection `a`), c'est-à-dire un type somme défini par les deux constructeurs `[]` (liste vide) et `cons` (ajout en tête). De plus, l'espèce des listes est décrite comme un setoïde (d'où la clause d'héritage) : l'égalité de deux listes est définie par double inclusion. A ce stade, nous sommes en mesure de prouver que l'égalité de listes est bien une relation d'équivalence. Les preuves des assertions sont réutilisées[3] dans les unités qui dérivent de `liste`.

Les preuves peuvent se faire directement avec le langage de tactiques de Coq ou avec un prouveur dédié en cours de développement.

```
species liste (a : setoid) inherits setoid;
rep = list(a);
let lg (l in self) = ... longueur
let app (l in self) = ... appartenance
let incl (l1 in self, l2 in self) = ... inclusion
let equal(l1 in self, l2 in self) =
 #and_b(!incl(l1, l2), !incl(l2, l1));
proof of equal_refl = ...
proof of equal_sym = ...
proof of equal_trans = ...
end
```

Définissons maintenant l'espèce des listes ordonnées à partir de l'espèce précédente : elle est paramétrée par un setoïde totalement ordonné. Outre les nouvelles fonctions d'insertion et de tri, nous spécifions le prédicat `est_ordonnée` qui décrit ce qu'est une liste ordonnée. Puis nous prouvons les propriétés de l'insertion et du tri.

```
species liste_ordonnée (a : ordre_total) inherits liste(a);
let rec insérer (l in self, e in a) in self = ...
  insertion dans une liste ordonnée
let rec tri (l in liste(a)) in self = ...
  tri par insertion
let est_ordonnée ( l in self) = ...
property insérer_préserve_ordre : all l in self, all e in a,
  !est_ordonnée(l) -> !est_ordonnée(!insérer(l,e))
  proof ... ;
property tri_correct : all l in liste(a),
  !equal(!tri(l), l) and !est_ordonnée(!tri(l))
  proof ... ;
end
```

Références

[1] V. Prevosto. *Conception et implantation du langage* FOC *pour le développement de logiciels certifiés*. PhD thesis, Université Paris VI, 2003.

[2] V. Prevosto and D. Doligez. Inheritance of algorithms and proofs in the computer algebra library foc. *Journal of Automated Reasoning*, 29(3-4) :337–363, 2002.

[3] R. Rioboo. *Programmer le calcul formel, des algorithmes à la sémantique*. Habilitation à diriger des recherches, Université Paris 6, 2002.

[4] The FOCAL Project. `http://www-spi.lip6.fr/foc`.

[3] De manière générale, une analyse des dépendances permet de vérifier que cette réutilisation est possible, c'est-à-dire que ni l'assertion, ni la preuve ne dépendent de méthodes redéfinies lors de l'héritage. Dans le cas contraire, FOCAL exigera que l'on refasse les preuves.