Traits orientés objet en λΠ-calcul modulo

Compilation de FoCaLize vers Dedukti

Raphaël Cauderlier, Catherine Dubois, Olivier Hermant, Deducteam/INRIA

20 août 2012

Table des matières

1 Fiche de synthèse

Le contexte général

Le sujet de ce travail porte sur la sémantique opérationnelle des traits orientés objet du système FoCaLize[13] dans le formalisme $\lambda\Pi$-calcul modulo[8] tel qu'il est implémenté par Dedukti[4].

Il se trouve donc à l'intersection de deux domaines : la sémantique opérationnelle des langages objets d'une part, et l'encodage de systèmes complexes en $\lambda\Pi$-calcul modulod'autre part.

FoCaLize et Dedukti sont des systèmes qui ont en partie été développés et maintenus par des membres de l'équipe Dedukteam de l'INRIA dans laquelle mon stage a eu lieu.

La sémantique de FoCaLize et de ses traits objets a été étudiée d'abord par Boulmé[6] puis par Prevosto[16] et Fechter[10].

Des sémantiques opérationnelles ont été proposées pour d'autres langages objets, par exemple pour C++[19] et Eiffel[2]. Mais les dépendances entre les méthodes de ces langages sont moins complexes que dans FoCaLize qui permet aux types des méthodes de contenir des appels aux méthodes précédentes. Cette complexité de dépendances s'exprimme plus facilement dans un système de types plus riche tel que celui de Coq ou de Dedukti.

Par ailleurs, des travaux ont été effectués pour encoder des systèmes logiques dans Dedukti tels que Coq[3] et HOL[1] mais aucun de ces systèmes ne présente de caractéristique objet.

Le problème étudié

L'objectif de ce stage était de traduire un sous-ensemble, le plus large possible, de FoCaLize vers Dedukti, définissant ainsi une sémantique opérationnelle à ce sous-ensemble.

Cette question n'est pas véritablement nouvelle dans la mesure où la sémantique de FoCaLize est déjà exprimée par sa compilation vers Coq et que Coq est traduisible vers Dedukti par l'outil Coqine[3]. Cependant cette traduction indirecte se révèle être très lourde c'est pourquoi une traduction directe est intéressante.

La contribution proposée

Nous avons étendu le compilateur de FoCaLize focalizec pour qu'il produise un fichier Dedukti, c'est-à-dire un système de réécriture, décrivant la sémantique opérationnelle à petit pas des mécanismes objets du fichier source FoCaLize.

Le trait qui s'est révélé être le plus difficile à traduire est la redéfinition de méthodes. Nous avons choisi pour siplifier le problème de ne traduire dans un premier temps que les autres traits, puis nous avons rajouté ce trait plus complexe qui a nécessité une grosse adaptation de notre traduction.

Pour simplifier le problème, nous nous sommes dans un premier temps interessés à la traduction des traits objets sans redéfinition de méthodes.

De plus nous avons travaillé sur des exemples simples illustrant quelques traits que nous avons traduits à la main avant de passer à l'implémentation.

Les arguments en faveur de sa validité

Au cours de ce travail, je me suis efforcé de suivre les directives strictes des développeurs de focalizec[].

De plus, la majeure partie du compilateur focalizec étant indépendante du langage vers lequel le code source est compilé, je réutilise ce code commun aux traductions vers OCaml et Coq pour ma traduction vers Dedukti, m'appuyant ainsi sur le travail ce qui facilite grandement la compatibilité sémantique avec les autres compilations de FoCaLize.

Par ailleurs, ma traduction des traits objets de FoCaLize en Dedukti est modulaire et suit presque linéairement le fichier source ce qui facilite la compréhension du code Dedukti produit.

Le bilan et les perspectives

Il y a plusieurs manières d'étendre ce travail de traduction de FoCaLize vers Dedukti. Tout d'abord, il est possible d'étendre la partie de FoCaLize qui est compilée jusqu'à arriver à traduire tout le langage FoCaLize. La difficulté de cette étape réside dans la traduction des preuves puisque FoCaLize ne manipule pas de preuve mais délègue la vérification des preuves soit au prouveur automatique Zenon[5] auquel il faudra ajouter une sortie pour Dedukti, soit directement à Coq.

De manière plus théorique, nous n'avons rien prouvé sur cette traduction et il serait important de montrer que cette nouvelle sémantique des traits objets de FoCaLize est équivalente aux précédentes.

Enfin, ce travail peut aussi être vu comme un modèle de traduction des langages objets en Dedukti et il serait intéressant de traduire d'autres langages objets vers Dedukti pour les étudier et prouver des propriétés sur ces langages.

2 Introduction

Devant la complexité croissante des logiciels et le caractère non exhaustif des tests, on a souvent recours à des outils pour prouver plus ou moins automatiquement la correction des programmes, en particulier quand des vies humaines reposent sur l'utilisation d'un logiciel. Le problème de la correction d'un programme étant, d'après le théorème de Rice, indécidable, c'est généralement à l'utilisateur que revient la tâche de démontrer que le code respecte effectivement sa spécification.

FoCaLize[13] est un environnement de développement de programmes certifiés, c'est-à-dire corrects vis-à-vis de leurs spécifications. Il permet de spécifier, implanter et prouver des algorithmes. Il est facile à utiliser grâce à sa philosophie objet qui se révèle très pratique pour la formalisation des structures algébriques.

Dedukti[4] est un vérificateur de preuves écrites dans un formalisme qui peut être considéré comme un standard pour les systèmes de preuves. En effet il utilise un langage très simple et suffisamment puissant et extensible pour encoder les autres systèmes.

L'objectif de ce stage est de compiler les traits orientés objet de FoCaLize en Dedukti. L'intérêt de ce travail est multiple :

- Il permet de donner à FoCaLize une sémantique opérationnelle ; la manière dont est calculé un terme ou une preuve de FoCaLize est exactement ce que définit le système de réécriture produit par la traduction du terme.
- Il permet d'étudier la sémantique de la spécification et du développement certifié en présence d'objets en se penchant en particulier sur des problèmes de dépendances complexes entre les méthodes.
- Il donne au $\lambda\Pi$-calcul modulo$^{\sim}$ (la logique de Dedukti) un exemple d'encodage d'un langage orienté objet qui pourra être adapté pour la traduction d'autres systèmes logiques incorporant des constructions similaires.

Nous proposons une traduction des traits objets de FoCaLize en Dedukti ainsi qu'une extension du compilateur de FoCaLize qui implémente cette traduction.

Nous allons présenter dans un premier temps l'état de l'art de la sémantique opérationnelle des mécanismes objets. Nous nous intéresserons ensuite plus précisément au système FoCaLize et en particulier à ces traits orientés objet. Les particularités de Dedukti seront ensuite exposées. Enfin, nous détaillerons notre traduction des traits objets de FoCaLize en Dedukti.

3 Cadre scientifique

Ce travail se place dans le cadre de l'étude de la sémantique opérationnelle des langages objets, en particulier quand les méthodes sont spécifiées et certifiées.

3.1 Sémantiques des langages orientés objets

Une classe est un regroupement de fonctions (éventuellement constantes) appelées méthodes. Les classes peuvent être instanciées par des objets sur lesquels il est possible d'appeler ces méthodes. Il est donc assez naturel de représenter les classes et les objets comme des enregistrements de fonctions, chaque fonction correspondant à une méthode.

Nous nous intéressons à des méthodes typées. Le type d'une méthode peut contenir des appels à d'autres méthodes de la même classe mais ne peut pas être redéfini.

Considérons par exemple, une classe S représentant la structure mathématique de setoïde, c'est-à-dire un ensemble muni d'une relation d'équivalence : les méthodes de S sont rep de type Type représentant le type des éléments du setoïde, eq de type rep $\rightarrow$ rep $\rightarrow$ Bool, la relation binaire sur rep et sa spécification eq_equiv de type relation_equivalence(eq).

Pour représenter ces méthodes, il est possible d'utiliser des enregistrements dépendants comme ceux de Coq par exemple. Un champ peut alors être déclaré en fonction des champs précédents. On aurait donc quelque chose comme $S = \{\text{rep} : \text{Type}, \text{eq} : \text{rep} \rightarrow \text{rep} \rightarrow \text{Bool}, \text{eq\_equiv} : \text{relation\_equivalence(eq)}\}$.

Une étape supplémentaire de généralisation peut être effectuée pour prendre en compte sous une forme arborescente les méthodes déclarées[17].

3.2 Travaux connexes

Le système Maude[14] est un outil puissant utilisé en logique pour écrire des prouveurs automatiques [7] et un model-checker[9]. Il utilise beaucoup la réécriture à la fois pour définir des fonctions et pour étudier l'évolution de systèmes de transitions. En particulier, il possède une extension, définie de manière interne, qui permet l'utilisation du paradigme objet. Maude permet donc de combiner réécriture, objets et logique ce qui le rend proche de nos travaux. Cependant la couche objet de Maude ne s'intéresse pas aux mécanismes objets ; les notions telles que l'héritage, la liaison retardée ou la redéfinition de méthode sont absentes de Maude. Par ailleurs, Maude ne repose pas comme FoCaLize et Dedukti sur un langage fonctionnel dans lequel les propriétés logiques peuvent s'interpréter comme le type de leurs preuves.

La sémantique opérationnelle des mécanismes objets qui nous intéressent a été étudiée dans le cadre de différents langages tels que C++[19], un sous-ensemble de Java[15] et Eiffel[2].

4 Le langage source : FoCaLize

4.1 Spécifications et preuves via la Correspondance Curry-Howard

Nous ne nous intéressons pas simplement à donner un sens aux traits objets mais aussi à étudier la spécification et la certification de méthodes qui sont un élément très important de FoCaLize.

Un parallèle entre le monde de la logique et celui de la programmation est fait par la correspondance de Curry-De Bruijn-Howard[11]. Elle permet en effet de considérer une preuve comme un λ-terme dont le type est la propriété prouvée. Vérifier une preuve consiste alors à typer le λ-terme représentant la preuve. Les différents constructeurs et règles de la logique reçoivent alors une interprétation dans le monde de la programmation.

Pour pouvoir énoncer des propriétés intéressantes, nous avons besoin d'un système de types assez riche pour qu'il permette de les exprimer.

Par ailleurs, certains assistants de preuve comme Coq[18] utilisent nativement la correspondance Curry-De Bruijn-Howard. Le prouveur automatique Zenon[5] est capable de produire en sortie un λ-terme de preuve vérifiable par Coq.

4.2 FoCaLize

FoCaLize[13] est un environnement pour le développement de programmes certifiés. Il a entre autres été utilisé pour certifier une bibliothèque de calcul formel[6] et pour formaliser les règles de

sécurité de l'aviation civile[**?**].

Le langage de programmation sous-jacent à FoCaLize est un langage fonctionnel très inspiré par OCaml[12]. Il présente les constructions les plus habituelles de ce langage telles que le polymorphisme, les types de données algébriques et les types enregistrement.

Il y ajoute certains traits objets parmi les plus utilisés : héritage, passage de paramètres (objets ou valeurs) aux classes, méthodes déclarées mais non définies (que nous appellerons simplement « méthodes déclarées », dans d'autres langages objets, elles sont aussi fréquemment appelées « abstraites » ou « virtuelles »), liaison retardée et redéfinition qui permettent de passer progressivement de la spécification formelle d'algorithmes à leurs implémentations certifiées.

Les classes de FoCaLize s'appellent des espèces et se distinguent principalement des classes des autres langages objets par une méthode particulière appelée la représentation qui renvoie un type représentant le support de l'espèce sur lequel travaillent les autres méthodes. Le support est le type des éléments manipulés par les méthodes de l'espèce. Quand cette méthode représentation est simplement déclarée, elle permet de donner à l'espèce la généricité d'une classe dépendant d'un paramètre polymorphe.

Dans FoCaLize, les spécifications et certifications [1] des méthodes sont elles-mêmes des méthodes dont le type est l'énoncé logique vu à travers la correspondance Curry-Howard et la définition est la preuve de cet énoncé ; une méthode logique déclarée correspond donc à une propriété qui n'a pas encore reçu de preuve, c'est-à-dire une spécification. Cette vision unifiée permet d'appliquer aux méthodes logiques les mêmes traitements qu'aux autres méthodes, en particulier en ce qui concerne l'héritage, la liaison retardée et la redéfinition.

Il y a donc trois sortes de méthodes pour FoCaLize ; les méthodes calculatoires qui retournent une valeur, les méthodes logiques qui retournent une preuve de leur type et la méthode représentation qui retourne un type. L'exemple de l'espèce des setoïdes contient ces trois sortes de méthodes ; rep est la représentation, eq est une méthode calculatoire et eq_equiv est une méthode logique.

Une autre particularité du système FoCaLize est son processus de double compilation : un fichier FoCaLize est en effet compilé à la fois vers un fichier OCaml pour être exécuté et vers un fichier Coq pour être vérifié (en particulier les preuves).

L'utilisateur a deux manières de fournir les preuves ; il peut soit donner directement la preuve Coq dans le fichier FoCaLize soit utiliser le prouveur automatique Zenon qui recherchera une preuve et la renverra à Coq.

La compilation de FoCaLize vers Coq utilise des enregistrements dépendants qui reflètent les dépendances entre les déclarations des méthodes.

Les constructions de FoCaLize seront présentées de manière plus détaillées dans la section concernant la traduction de FoCaLize vers Dedukti.

4.3 Dépendances entre les méthodes

Dans [16], Virgile Prevosto distingue deux formes de dépendances pour les méthodes de FoCaLize :

- decl-dépendance
  Une méthode m_1 decl-dépend d'une méthode m_2 si m_1 a besoin de connaître de m_2 son nom et son type.

1. Par certification, nous entendons la preuve qu'un algorithme est correct vis-à-vis de sa spécification.

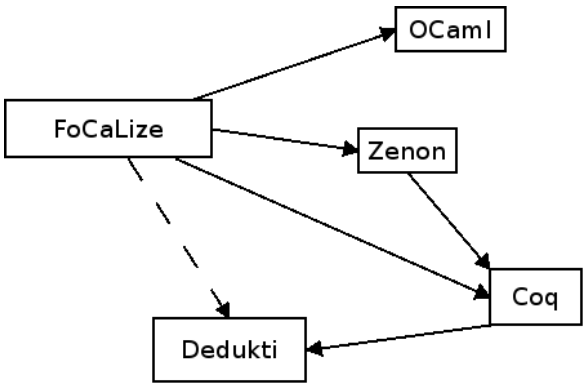

FIGURE 1 – Graphe de compilation de FoCaLize

Dans un compilateur, cette forme de dépendance se résout en général en ajoutant à m_1 un argument qui est la fonction m_2 elle-même. Dans le cas où la decl-dépendance a lieu dans le type de m_1, cet argument supplémentaire intervient dans le typage dépendant de m_1.

– def-dépendance
Une méthode m_1 def-dépend d'une méthode m_2 si m_1 a besoin de connaître la définition de la méthode m_2.
C'est le cas en particulier quand m_1 est la méthode qui prouve la correction de m_2 en déroulant sa définition. Si m_1 def-dépend de m_2 et que m_2 est redéfinie, la définition de m_1 doit être invalidée, la méthode m_1 redevient alors une méthode déclarée contrairement à ce qu'il se passerait dans le cas d'une decl-dépendance qui n'invalide jamais une définition. Remarquons qu'une def-dépendance ne peut pas avoir lieu au niveau du type de m_1.

5 Le langage destination : Dedukti

5.1 Types dépendants

Les types dépendants sont un moyen de définir des familles de types indexées par des valeurs. Ils interviennent naturellement dans la formalisation des mathématiques, en particulier avec des valeurs entières (polynômes de degré $\leq n$, matrices carrées de taille n, $\mathbb{K}$-espaces vectoriel, etc. . .).

Les types dépendants sont construits par l'utilisation du produit dépendant $\Pi x : A.B(x)$ qui est le type des fonctions qui à un argument x de type A renvoient un résultat de type $B(x)$. Les types de fonctions $A \to B$ que l'on rencontre dans le λ-calcul simplement typé sont un cas particulier de type dépendant pour lequel le type du résultat de la fonction ne dépend pas de l'argument.

$$A \to B = \Pi x : A.B$$

5.1.1 Exemple de type dépendant

L'exemple habituel pour présenter les types dépendants est celui des listes d'entiers de longueur n. Si le type Nat représente les entiers naturels de Peano (via les constructeurs $0 : \mathrm{Nat}$ et $\mathrm{S} : \mathrm{Nat} \to \mathrm{Nat}$), alors le type des listes d'éléments de type Nat et de longueur $n : \mathrm{Nat}$ peut être vu comme $\mathrm{list} : \mathrm{Nat} \to \mathrm{Type}$ avec les constructeurs $\mathrm{Nil} : \mathrm{list}\,0$ et $\mathrm{Cons} : \Pi n : \mathrm{Nat}\,.\,\mathrm{Nat} \to \mathrm{list}\,n \to \mathrm{list}(\mathrm{S}\,n)$.

5.1.2 Types dépendants et quantification universelle

L'interprétation logique du produit dépendant Π est celle du quantificateur universel $\forall$. En effet, dire par exemple que la fonction tail est de type $\Pi n : \mathrm{Nat}\,.\Pi l : \mathrm{list}(\mathrm{S}\,n).\,\mathrm{list}\,n$ est la même chose qu'énoncer le théorème « Pour tout $n \in \mathbb{N}$ et toute liste de longueur $n+1$ l, la queue de l est une liste de longueur n. » Prouver une propriété de la forme $\forall x : T, P(x)$ est la même chose que renvoyer une preuve de $P(x)$ pour un argument x arbitraire, autrement dit que de définir une fonction qui à x de type T renvoie une preuve de $P(x)$, c'est-à-dire une fonction de type $\Pi x : T.P(x)$.

5.1.3 Types dépendants et decl-dépendance

Supposons maintenant que nous ayons une classe pourvue des méthodes suivantes :
- A : Type
- $\mathrm{eq} : A \to A \to \mathrm{Type}$ (une relation binaire sur A)

Déclarer une méthode refl spécifiant que la relation eq est réflexive revient à énoncer le théorème $\forall x : A, \mathrm{eq}\ x\ x.$ qui se traduit par $\Pi x : A.\,\mathrm{eq}\ x\ x$. Cette méthode refl decl-dépend de A puisqu'elle a besoin de savoir que c'est un type et de eq puisqu'elle a besoin de savoir que c'est une relation binaire sur le type A. Convertir cette méthode en une déclaration autonome peut se faire en passant ses dépendances en paramètre :

$$\mathrm{refl} : \Pi A : \mathrm{Type}\,.\Pi\,\mathrm{eq} : (A \to A \to \mathrm{Type}).\Pi x : A.\,\mathrm{eq}\ x\ x$$

5.2 Réécriture

5.2.1 Système de réécriture

Plutôt que de donner une définition formelle de ce qu'est un système de réécriture, commençons par regarder un exemple très classique, la définition de l'addition sur les entiers de Peano.

$$\begin{aligned} \mathrm{Nat} &: \mathrm{Type} \\ 0 &: \mathrm{Nat} \\ \mathrm{S} &: \mathrm{Nat} \to \mathrm{Nat} \\ \mathrm{plus} &: \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat} \end{aligned}$$

Ce système se compose tout d'abord de déclarations : un type Nat et les constructeurs 0 et S qui définissent à eux seuls les entiers naturels et une fonction plus. Ces déclarations forment la signature du système.

$$\begin{array}{ll} [m : \mathrm{Nat}\qquad\quad\ ] & \mathrm{plus}\ \ 0\ m \hookrightarrow m \\ [n : \mathrm{Nat}, m : \mathrm{Nat}] & \mathrm{plus}\ (\mathrm{S}\,n)\ m \hookrightarrow \mathrm{S}(\mathrm{plus}\ \ n\ m) \end{array}$$

La fonction plus est ensuite définie par les deux règles de réécriture ci-dessus qui permettent de simplifier toute expression dont le symbole de tête est plus. Chaque règle de réécriture est composée d'un contexte (éventuellement vide, le contexte vide est noté []) qui liste les variables libres et leur associe leur type, d'un membre gauche similaire à un motif d'un langage de programmation fonctionnel et un membre droit plus simple [2] qui doit avoir le même type que le membre gauche.

2. On exige généralement que la relation définie par un système de réécriture soit confluente et fortement normalisante.

5.2.2 Avantages de la réécriture

L'utilisation d'un système de réécriture permet une programmation très libre. Si l'on souhaite par exemple définir deux fonctions mutuellement récursives even : Nat → Bool et odd : Nat → Bool qui renvoient un booléen indiquant si leur argument est pair ou impair respectivement, c'est faisable simplement par réécriture sans nécessité de déclarer leurs définitions comme mutuellement récursives :

$$\begin{aligned} \text{even} &: \text{Nat} \to \text{Bool} \\ \text{odd} &: \text{Nat} \to \text{Bool} \end{aligned}$$

$$\begin{aligned} [\quad\quad] \quad & \text{even } 0 \hookrightarrow \text{true} \\ [n : \text{Nat}] \quad & \text{even } (\text{S } n) \hookrightarrow \text{odd } n \\ [\quad\quad] \quad & \text{odd } 0 \hookrightarrow \text{false} \\ [n : \text{Nat}] \quad & \text{odd } (\text{S } n) \hookrightarrow \text{even } n \end{aligned}$$

Par ailleurs, rien n'oblige l'utilisateur à donner aux fonctions des définitions totales ; par exemple il est possible de définir partiellement une fonction head renvoyant la tête d'une liste, là encore sans déclaration particulière :

$$\begin{aligned} A &: \text{Type} \\ \text{list} &: \text{Type} \\ \text{Nil} &: \text{list} \\ \text{Cons} &: A \to \text{list} \to \text{list} \end{aligned}$$

$$[a : A, l : \text{list}] \ \text{head}(\text{Cons } A\ l) \hookrightarrow a$$

Remarquons que le terme head Nil est correctement typé de type A et ne réduit pas (ni par β-réduction ni par réécriture) ni ne lève d'exception.

Cependant cette facilité a un prix ; dans le contexte de la réécriture, il est souvent plus difficile de montrer que les fonctions terminent et quand on a besoin de savoir que les fonctions sont totales (autrement dit, que les preuves sont exhaustives), on a perdu cette assurance que l'on pouvait avoir dans des langages qui rejettent les définitions partielles.

De plus, dans Dedukti, il y a peu d'inférence de type ; l'utilisateur doit fournir le type des fonctions de la signature et de l'environnement de chaque règle de réécriture.

5.3 Dedukti

5.3.1 Le λΠ-calcul modulo, réécriture avec types dépendants

Dedukti[4] est un système qui combine un λ-calcul à types dépendants (le $\lambda\Pi$-calcul) à un système de réécriture pour former le système logique $\lambda\Pi$-calcul modulo.

Dans ce système, la réécriture et le typage sont intimement liés ; les règles de réécriture doivent toutes être bien typées, c'est-à-dire que le membre gauche et le membre droit doivent avoir des types (qui peuvent dépendre de λ-termes arbitraires) convertibles pour la relation induite par la β-réduction et les règles de réécriture précédentes. Voir Les règles de typage du $\lambda\Pi$-calcul modulo.

Reprenons par exemple le cas des listes de longueur n. Pour commencer, nous avons besoin de définir les entiers pour pouvoir compter les éléments des listes.

$$\begin{aligned} \text{Nat} &: \text{Type} \\ 0 &: \text{Nat} \\ \text{S} &: \text{Nat} \to \text{Nat} \\ \text{plus} &: \text{Nat} \to \text{Nat} \to \text{Nat} \end{aligned}$$

$$\begin{aligned} [m : \text{Nat}\quad\quad\quad\quad\quad\quad\quad\quad] \quad & \text{plus } 0\ m \hookrightarrow m \\ [n : \text{Nat}, m : \text{Nat}] \quad & \text{plus } (\text{S } n)\ m \hookrightarrow \text{S}(\text{plus } n\ m) \end{aligned}$$

Puis les listes de longueur n (d'éléments de A, un type fixé [3]) peuvent être définies par :

$$\begin{aligned} A &: \text{Type} \\ \text{list} &: \text{Nat} \to \text{Type} \\ \text{Nil} &: \text{list } 0 \\ \text{Cons} &: \Pi n : \text{Nat}\,.A \to \text{list } n \to \text{list}(\text{S } n) \end{aligned}$$

Nous pouvons alors définir par exemple l'opération append de concaténation sur les listes :

$$\text{append} : \Pi n : \text{Nat}\,.\Pi m : \text{Nat}\,.\,\text{list } n \to \text{list } m \to \text{list}(\text{plus } n\ m)$$

$$\begin{aligned} [m : \text{Nat}, r : \text{list } m\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad] \quad & \text{append } 0\ m\ \text{Nil } r \hookrightarrow r \\ [n : \text{Nat}, a : A, l : \text{list } n, m : \text{Nat}, r : \text{list } m] \quad & \text{append } (\text{S } n)\ m\ (\text{Cons } n\ a\ l)\ r \hookrightarrow \text{Cons } (\text{S } n)\ a\ (\text{append } n\ m\ l\ r) \end{aligned}$$

Ces dernières règles de réécriture sont bien typées grâce à la définition préalable de plus.

5.3.2 Un système logique fait pour encoder les autres

Malgré sa simplicité, le $\lambda\Pi$-calcul modulo˜se révèle être un langage logique très puissant vers lequel on peut traduire un certain nombre d'autres systèmes logiques : actuellement, le Calcul des Constructions Inductives tel qu'implémenté par Coq est traduit vers Dedukti par l'outil Coqine[3], et les systèmes reposant sur HOL via OpenTheory peuvent aussi être traduits par Holide[1]. De manière plus théorique, tout système de types pur (PTS) fonctionnel peut être traduit dans le $\lambda\Pi$-calcul modulo[8]. De plus, toutes ces traductions sont ce que l'on appelle des encodages superficiels, c'est-à-dire que les preuves des différents systèmes sont des λ-termes sur lesquels il est possible de calculer à l'intérieur de Dedukti par réécriture et β-réduction pour, par exemple, appliquer un algorithme d'élimination de coupures.

6 La traduction

6.1 Présentation générale

La traduction de FoCaLize vers Dedukti que nous allons présenter a été implémentée comme une extension du compilateur de FoCaLize. Nous avons voulu rester fidèles à l'esprit du compilateur en écrivant cette extension de la manière la plus indépendante possible et en réutilisant la partie commune aux processus de compilation vers OCaml et vers Coq. En particulier, nous disposons du

3. Dedukti n'est pas polymorphe mais il est possible d'encoder le polymorphisme dans Dedukti comme nous le verrons (Encodage du polymorphisme).

graphe de dépendances entre les méthodes et l'héritage est résolu c'est-à-dire que nous savons dire, pour chaque méthode, dans quelle espèce elle a été définie ou déclarée pour la dernière fois.

Par ailleurs notre traduction est aussi modulaire que les traductions vers OCaml et Coq ; le code produit pour chaque espèce est regroupé au même endroit et une définition de méthode se traduit toujours en une seule fois.

La première chose qui différencie la compilation vers Dedukti des compilations vers OCaml et Coq est l'absence de bibliothèque standard ; il a été nécessaire d'encoder en Dedukti quelques éléments de programmation haut-niveau tels que les enregistrements dépendants et le polymorphisme ainsi qu'un peu de calcul sur les booléens et les propriétés logiques.

Le compilateur de FoCaLize nous fournit les informations suivantes :

- Pour chaque espèce, nous connaissons son nom, la liste de ses méthodes et la liste de ses paramètres ainsi que le graphe de dépendances (def- et decl-dépendances) entre les méthodes.
- Pour chaque méthode de chaque espèce, nous connaissons son nom, son caractère définie ou déclarée, locale ou héritée (et de quelle espèce parente) et enfin si la méthode est une fonction, une propriété logique ou la méthode représentation.

Nous ne nous intéressons pas aux preuves de terminaison des fonctions[2]. Au sein d'une même espèce, nous traitons les méthodes comme toutes potentiellement mutuellement récursives (c'est-à-dire que nous les déclarons toutes avant de définir la première).

Le point le plus délicat dans la traduction des traits objets de FoCaLize est la redéfinition de méthode.

La difficulté du traitement de la redéfinition réside dans le fait que l'espèce doit avoir un moyen de choisir la bonne définition des méthodes et que ce choix dépend du contexte.

Pour simplifier le problème, nous nous sommes intéressés, dans un premier temps, à la traduction de FoCaLize sans redéfinition et avons, dans un deuxième temps, ajouté ce trait à notre traduction ce qui a nécessité de grandement modifier notre encodage.

6.2 Traduction sans redéfinition

6.2.1 Encodage des enregistrements dépendants

Il est possible d'encoder les structures d'enregistrements dépendants en Dedukti. Pour définir un enregistrement dépendant

$$R = \{f_1 : T_1, f_2 : T_2(f_1), \ldots, f_n : T_n(f_1, \ldots, f_{n-1})\}$$

il suffit de définir un type R, un constructeur Make_R de type

$$\mathrm{Make}_R : \Pi f_1 : T_1.\Pi f_2 : T_2(f_1).\ldots\Pi f_n : T_n(f_1, \ldots, f_{n-1}).R$$

et les projecteurs Proj_{R,f_i} de type

$$\mathrm{Proj}_{R,f_i} : R \to T_i(f_1, \ldots, f_{i-1})$$

avec les règles de projection

$$[f_1, \ldots, f_i, \ldots, f_n]\, \mathrm{Proj}_{R,f_i}(\mathrm{Make}_R\, f_1 \ldots f_i \ldots f_n) \hookrightarrow f_i.$$

Pour plus de lisibilité, nous notons $\{f_1 = d_1; \ldots; f_n = d_n\}$ le terme $\mathrm{Make}_R\, d_1 \ldots d_n$ quand il n'y a pas d'ambiguïté sur le type R.

6.2.2 Encodage du polymorphisme

Le $\lambda\Pi$-calcul modulo˜n'est pas polymorphe, les arguments des types ne peuvent pas eux-mêmes être des types. Il est cependant possible d'encoder le polymorphisme en ne déclarant pas directement des types A : Type. Pour cela on déclare un type U_{Type} appelé l'univers des types et une fonction qui permet de voir tout élément de U_{Type} comme un type : $\varepsilon_{\text{Type}} : U_{\text{Type}} \to \text{Type}$.

Par exemple, définissons dans Dedukti la fonction MapCons, composée des fonctions sur les listes Map et Cons, c'est-à-dire la fonction qui à a et à la liste de listes $[l_1, \ldots, l_n]$ associe la liste de listes $[a :: l_1, a :: l_2, \ldots, a :: l_n]$. Pour cela nous commençons par définir le type des listes polymorphes par les constructeurs Nil et Cons :

$$\begin{aligned} U_{\text{Type}} &: \text{Type} \\ \varepsilon_{\text{Type}} &: U_{\text{Type}} \to \text{Type} \\ \text{list} &: U_{\text{Type}} \to U_{\text{Type}} \\ \text{Nil} &: \Pi A : U_{\text{Type}}.\varepsilon_{\text{Type}}(\text{list}\ \ A) \\ \text{Cons} &: \Pi A : U_{\text{Type}}.\Pi a : \varepsilon_{\text{Type}}\ A.\varepsilon_{\text{Type}}(\text{list}\ \ A) \to \varepsilon_{\text{Type}}(\text{list}\ \ A) \end{aligned}$$

La fonction MapCons peut alors être définie par :

$$\text{MapCons} : \Pi A : U_{\text{Type}}.\varepsilon_{\text{Type}}\ A \to \varepsilon_{\text{Type}}(\text{list}(\text{list}\ \ A)) \to \varepsilon_{\text{Type}}(\text{list}(\text{list}\ \ A))$$

$$\begin{aligned} &[A : U_{\text{Type}}, a : \varepsilon_{\text{Type}}\ A] \\ &\quad \text{MapCons}\ \ A\ a\ (\text{Nil}(\text{list}\ \ A)) \ \hookrightarrow\ \text{Nil}(\text{list}\ \ A) \\ &[A : U_{\text{Type}}, a : \varepsilon_{\text{Type}}\ A, l : \varepsilon_{\text{Type}}(\text{list}\ \ A), L : \varepsilon_{\text{Type}}(\text{list}(\text{list}\ \ A))] \\ &\quad \text{MapCons}\ \ A\ a\ (\text{Cons}\ (\text{list}\ \ A)\ l\ L) \hookrightarrow\ \text{Cons}\ (\text{list}\ \ A)\ (\text{Cons}\ \ A\ a\ l)\ (\text{MapCons}\ \ A\ a\ L) \end{aligned}$$

6.2.3 Propriétés logiques et booléens

La différence entre un booléen et une propriété logique n'est pas directement visible par l'utilisateur de FoCaLize mais est néanmoins fondamentale. En effet les méthodes renvoyant un booléen sont souvent utilisées telles quelles à l'intérieur des propriétés logiques. Par exemple, une méthode spécifiant la réflexivité d'une méthode eq sera simplement déclarée de type $\forall x :, \text{eq}(x, x)$. Dans cette expression, $\text{eq}(x, x)$ est considéré non plus comme un booléen mais comme une propriété logique. Cette distinction est introduite dans Dedukti pour lequel les propriétés logiques sont des types sur lesquels on ne veut pas faire de calcul. Les méthodes logiques sont typées par le type Prop tandis que les méthodes calulatoires booléennes sont typées par le type Bool.

Nous avons donc choisi, comme travail préliminaire, d'écrire une bibliothèque sur les booléens et leurs propriétés et une fonction $\varepsilon_{bool} : \text{Bool} \to \text{Prop}$ nous permet d'interpréter tout booléen comme la propriété correspondante.

6.2.4 Les espèces de FoCaLize

Une espèce S de FoCaLize est compilée en une liste de fonctions $\text{meth}_{S,m_1} \ldots \text{meth}_{S,m_n}$. Ces fonctions prennent en argument un enregistrement s contenant les informations manquantes, c'est-à-dire en l'absence de redéfinition, les définitions des méthodes déclarées. $\text{meth}_{S,m_i}\ s$ est alors définie par :

– La projection correspondant au champ m_i si la méthode est déclarée

– La définition de m_i si la méthode est définie.

Reprenons l'exemple du setoïde de la section Sémantiques des langages orientés objets. On considère une espèce Setoid disposant des méthodes suivantes :
– rep, méthode représentation, déclarée
– $\text{eq} : \text{rep} \to \text{rep} \to \text{Bool}$, définie par $\text{eq}(x, y) = \text{true}$
– $\text{refl} : \forall x : \text{rep}, \text{eq}(x, x)$, déclarée,
Les informations qu'il nous manque pour définir $\text{meth}_{\text{Setoid,rep}}$, $\text{meth}_{\text{Setoid,eq}}$ et $\text{meth}_{\text{Setoid,refl}}$ sont les définitions des méthodes rep et refl. Cependant nous ne pouvons pas passer à nos fonctions l'enregistrement dépendant

$$s = \{\text{rep} : \text{Type}, \text{refl} : (\Pi x : \text{rep} \,.\, \text{eq}\, x x)\}$$

car celui-ci est mal typé (la variable eq est libre). Une solution est de rajouter le paramètre manquant eq en arguments comme ceci :

$$s = \{\text{rep} : \text{Type}, \text{refl} : (\Pi\, \text{eq} : (\text{rep} \to \text{rep} \to \text{Bool}).\Pi x : \text{rep} \,.\, \text{eq}\, x x)\}$$

et de définir les fonctions par les règles de réécriture suivantes :

$$[\text{rep}_{\text{def}}, \text{refl}_{\text{def}}]\ \text{meth}_{\text{Setoid,rep}}\{\text{rep} = \text{rep}_{\text{def}}; \text{refl} = \text{refl}_{\text{def}}\} \hookrightarrow \text{rep}_{\text{def}}$$

$$[s]\, \text{meth}_{\text{Setoid,eq}}\ s \hookrightarrow \lambda x y.\, \text{true}$$

$$[\text{rep}_{\text{def}}, \text{refl}_{\text{def}}]\, \text{meth}_{\text{Setoid,refl}}\{\text{rep} = \text{rep}_{\text{def}}; \text{refl} = \text{refl}_{\text{def}}\} \hookrightarrow \text{refl}_{\text{def}}(\text{meth}_{\text{Setoid,eq}}\{\text{rep} = \text{rep}_{\text{def}}; \text{refl} = \text{refl}_{\text{def}}\})$$

6.2.5 Héritage

Dans FoCaLize, une espèce peut hériter d'une ou plusieurs autres espèces. Dans ce cas, l'espèce courante aura accès à toutes les méthodes des espèces parentes en plus de celles introduites par l'espèce elle-même. L'héritage mis en oeuvre dans FoCaLize suit la politique de la liaison retardée et impose que le type d'une méthode ne change pas dans les espèces qui en héritent.
– Pour les méthodes déclarées

  Si m est une méthode déclarée, aucune définition ne peut être trouvée dans l'espèce courante ni dans ses parents. L'héritage ne nous donne pas d'information sur la valeur de la méthode et la fonction $\text{meth}_{S,m}$ est définie comme nous venons de le voir.

– Pour les méthodes définies

  Si P est l'espèce parente dans laquelle la méthode a été définie et S l'espèce courante, on réécrit l'appel de la méthode m vers l'appel de la même méthode dans P.

  $$[d_1 : t_1, \ldots, d_n : t_n]\ \text{meth}_{S,m}\{m_1 = d_1; \ldots; m_n = d_n\} \hookrightarrow \text{meth}_{P,m}\{\ldots\}$$

  où les définitions passées à l'enregistrement de type P sont les définitions des méthodes déclarées de P vues de S c'est-à-dire les $\text{meth}_{S,m_i}\{m_1 = d_1; \ldots; m_n = d_n\}$ pour chaque méthode déclarée m_i de l'espèce P.

6.2.6 Liaison retardée

La terminaison de cette implémentation de l'héritage est assurée par le fait que la réécriture ne fait que remonter la hiérarchie sans jamais la redescendre. Il ne peut donc y avoir de cycle de réécriture à cause de l'héritage puisqu'une espèce ne peut hériter que d'une espèce définie avant elle.

Considérons par exemple une espèce S_1 déclarant une méthode a et définissant une méthode b par $b = a$ et une espèce S_2 héritant de S_1 et définissant a par $a = 2$. La valeur de la méthode b de S_2 est 2.

6.2.7 Représentation (type support)

La représentation est traduite comme les autres méthodes, lorsqu'elle est définie, le type des méthodes qui en dépendent (presque toutes en général) est automatiquement converti pour le prendre en compte par la règle de réécriture qui définit la méthode. Notons que nous n'avons pas de problème de redéfinition ici car le système FoCaLize ne permet pas de redéfinir la représentation.

6.2.8 Les collections

Les collections de FoCaLize correspondent aux objets des langages objets [4]. Les collections instancient les espèces complète (c'est-à-dire les espèces dont toutes les méthodes sont définies donc les espèces dont l'enregistrement est vide). Sur une collections, il est possible d'appeler les méthodes directement sans passer de paramètres en argument et même sans savoir à quelle collection on a affaire.

En effet, les collections sont passées en paramètre des espèces non pas directement mais en tant qu'implémentations d'espèces dont elles héritent. Par exemple, si l'on dispose d'une espèce $\mathcal{K}$ représentant la structure algébrique de corps, on peut définir en FoCaLize l'espèce des espaces vectoriels sur un corps K comme une espèce dépendant d'un paramètre K qui est une collection instanciant une espèce héritant de l'espèce des corps. L'appel de la méthode zero de K ne peut se traduire par une application de $\mathrm{meth}_{\mathcal{K},\mathrm{zero}}$ puisque, pour la collection K, l'espèce $\mathcal{K}$ n'est qu'une espèce du processus d'héritage parmis d'autres ; d'ailleurs la définition de la méthode zero de K peut même ne rien avoir à voir avec l'espèce $\mathcal{K}$.

Pour traiter indifféremment n'importe quel descendant de $\mathcal{K}$, nous introduisons un type des collections que nous appellons Collection. Et pour chaque méthode m (de type $t(S)$) de chaque espèce S nous introduisons une fonction $\mathrm{coll\_meth}_m : \Pi C : \mathrm{Collection}.t(C)$. [5]

Ces fonctions sont définies au moment où les espèces complètes sont instanciées en collections. Par exemple, si une collection C instancie une espèce complète S, nous définissons partiellement pour chaque méthode m de S la fonction $\mathrm{coll\_meth}_m$ par la règle de réécriture

$$[]\,\mathrm{coll\_meth}_m\,C \hookrightarrow \mathrm{meth}_{S,m}$$

où C est le constructeur de type Collection qui représente la collection C.

4. La différence entre une collection de FoCaLize et un objet réside dans le fait qu'une collection est totalement statique. En particulier on ne peut pas créer de collection dans une définition de méthode.

5. Si plusieurs méthodes d'espèces différentes portent le même nom mais ont des types différents, nous devons les renommer par exemple en les indexant par leur type.

6.3 La redéfinition de méthodes

Implémenter la redéfinition de méthode nécessite de pouvoir changer le comportement d'une espèce *a posteriori* puisque changer la valeur d'une méthode change la valeur de toutes les méthodes qui en decl-dépendent [6].

Pour ce faire, nous avons besoin de rajouter les méthodes définies aux champs de notre enregistrement mais nous avons aussi besoin de pouvoir préciser qu'une méthode n'est pas redéfinie et que c'est la définition telle que vue depuis l'espèce qui doit être utilisée.

Reprenons notre exemple illustrant la redéfinition ; nous avons

- Une espèce S_1 définissant deux méthodes a et b par $a = 1$ et $b = a$.
- Une espèce S_2 héritant de S_1 redéfinissant a par $a = 2$
- Une collection C_1 instanciant S_1
- Une collection C_2 instanciant S_2

Les espèces S_1 et S_2 n'ont pas de méthodes déclarées donc leur enregistrement pour l'encodage sans redéfinition est vide. S_1 a besoin d'un moyen de savoir si sa méthode a a été redéfinie par S_2 (auquel cas c'est la nouvelle définition qui doit être utilisée) ou si c'est la définition locale $a = 1$ qui compte.

Pour cela, nous ajoutons à l'enregistrement un champ optionnel pour chaque méthode définie ; si ce champ est présent, la définition qu'il contient est utilisée prioritairement, sinon c'est la définition locale ou héritée qui est appelée comme précédemment.

Les enregistrements à champs optionnels sont implémentés par un type polymorphe option défini par les constructeurs None et Some :

$$\text{option} : U_{\text{Type}} \to U_{\text{Type}}.$$

$$\text{None} : \Pi A : U_{\text{Type}}.\varepsilon_{\text{Type}}(\text{option}\, A).$$

$$\text{Some} : \Pi A : U_{\text{Type}}.\varepsilon_{\text{Type}} A \to \varepsilon_{\text{Type}}(\text{option}\, A).$$

Pour alléger les notations, nous écrirons $\mathcal{N}_A$ pour $\text{None}\, A$ et $\mathcal{S}_A$ pour $\text{Some}\, A$.

Le type du constructeur d'enregistrement Make_{S_1} devient alors :

$$\text{Make}_{S_1} : \Pi a_{\text{opt}} : \varepsilon_{\text{Type}}(\text{option int}).\Pi b_{\text{opt}} : \varepsilon_{\text{Type}}(\text{option int}).S_1.$$

Et la fonction $\text{meth}_{S_1,a}$ est définie par deux règles de réécriture :

$$[b_{\text{opt}} : \varepsilon_{\text{Type}}(\text{option int})]\, \text{meth}_{S_1,a}\{a = \mathcal{N}_{\text{int}}; b = b_{\text{opt}}\} \hookrightarrow 1.$$

qui signifie que si a n'est pas redéfinie par la suite, c'est la définition locale $a = 1$ qui est utilisée. Et

$$[a_{\text{def}} : \varepsilon_{\text{Type}}\, \text{int}, b_{\text{opt}} : \varepsilon_{\text{Type}}(\text{option int})]\, \text{meth}_{S_1,a}\{a = \mathcal{S}_{\text{int}} a_{\text{def}}; b = b_{\text{opt}}\} \hookrightarrow a_{\text{def}}.$$

qui signifie que si a est redéfinie, c'est la nouvelle définition a_{def} qui est utilisée à la place.

6. Les définitions des méthodes qui def-dépendent d'une méthode redéfinie sont invalidées par le compilateur de FoCaLize.

6.3.1 Encodage plus profond du polymorphisme

Cet encodage de la redéfinition des méthodes calculatoires provoque cependant des problèmes de typage pour les deux autres sortes de méthode à savoir la représentation et les méthodes logiques, voir la section FoCaLize. En effet, les champs optionnels nécessitent d'être appliqués à des méthodes de type $\varepsilon_{\text{Type}} A$ pour un certain $A : U_{\text{Type}}$. Or la méthode représentation est de type U_{Type} et les méthodes logiques sont de type $\varepsilon_{\text{Prop}} P$ avec P de type Prop.

Pour uniformiser la gestion des méthodes définies, nous avons choisi d'encoder plus profondément le polymorphisme en rajoutant un niveau d'indirection. De la même manière que nous avions remplacé les types par des termes de U_{Type} grâce à la fonction $\varepsilon_{\text{Type}} : U_{\text{Type}} \to \text{Type}$, nous pouvons remplacer les termes de type Prop (resp. le terme U_{Type}) par des termes de U_{Type} (resp. le terme $U_{U_{\text{Type}}} : U_{\text{Type}}$) grâce à la fonction $\varepsilon\varepsilon_{\text{Prop}} : \text{Prop} \to U_{\text{Type}}$ (resp. $\varepsilon\varepsilon_{\text{Type}} : U_{U_{\text{Type}}} \to U_{\text{Type}}$. Ainsi toutes nos méthodes ont désormais un type de la forme $\varepsilon_{\text{Type}} A$ où A est $U_{U_{\text{Type}}}$ si la méthode est la représentation, $\varepsilon\varepsilon_{\text{Prop}} P$ avec $P : \text{Prop}$ si la méthode est logique ou n'importe quoi d'autre pour les méthodes calculatoires.

6.3.2 Collections

Les méthodes d'une collection ne pouvant être redéfinies, nous n'avons, comme dans le cas sans redéfinition, pas besoin d'enregistrement pour traduire une collection mais juste d'un terme C de type Collection. Et ses méthodes peuvent être définies par

$$[]\,\text{coll\_meth}_m\, C \hookrightarrow \text{meth}_{S,m}\{m_1 = \mathcal{N}_{t_1}; \ldots; m_n = \mathcal{N}_{t_n}\}.$$

Dans le cas de notre exemple, l'appel de la méthode b de la collection C_2 se réécrira de la façon suivante :

$$\begin{aligned}
\text{coll_meth}_b\, C_2 &\hookrightarrow \text{meth}_{S_2,b}\{a = \mathcal{N}_{\text{int}}; b = \mathcal{N}_{\text{int}}\} \\
&\hookrightarrow \text{meth}_{S_1,b}\{a = \mathcal{S}_{\text{int}}(\text{meth}_{S_2,a}\{a = \mathcal{N}_{\text{int}}; b = \mathcal{N}_{\text{int}}\}); b = \mathcal{N}_{\text{int}}\} \\
&\hookrightarrow \text{meth}_{S_1,b}\{a = \mathcal{S}_{\text{int}} 2; b = \mathcal{N}_{\text{int}}\} \\
&\hookrightarrow \text{meth}_{S_1,a}\{a = \mathcal{S}_{\text{int}} 2; b = \mathcal{N}_{\text{int}}\} \\
&\hookrightarrow 2
\end{aligned}$$

7 Conclusion et perspectives

La traduction des traits orientés objets de FoCaLize en Dedukti est un pas important vers la compilation du langage FoCaLize complet. La traduction des autres éléments du langage FoCaLize (tels que la modularité ou les types de données algébriques) n'apportent rien de nouveau puisqu'ils sont présents en Coq et traduits vers Dedukti par Coqine.

Cependant une difficulté se posera au moment de la traduction des preuves car elles sont inaccessibles au compilateur de FoCaLize. En effet, FoCaLize n'est ni un assistant de preuve, ni un prouveur automatique ; il n'est pas chargé de vérifier la correction des preuves mais délègue cette tâche généralement à Zenon et parfois à Coq directement. Il sera donc nécessaire d'adapter les preuves pour Coq ainsi que la sortie de Zenon pour Dedukti.

Cette compilation de tout le système FoCaLize vers Dedukti permettrait d'une part de bénéficier de théorèmes prouvés dans d'autres systèmes comme HOL, d'autre part de bénéficier dans ces autres systèmes de développements écrits pour FoCaLize comme sa bibliothèque de calcul formel.

8 Bibliographie

Références

[1] Ali Assaf and Guillaume Burel. Holide. `https://www.rocq.inria.fr/deducteam/Holide/index.html`.

[2] Isabelle Attali, Denis Caromel, and Michael Oudshoorn. A formal definition of the dynamic semantics of the eiffel language. In *Australian Computer Science Conference (ACSC)*, 1993.

[3] M. Boespflug and G. Burel. CoqInE : Translating the calculus of inductive constructions into the λΠ-calculus modulo. 2012.

[4] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. Dedukti. `https://www.rocq.inria.fr/deducteam/Dedukti/index.html`.

[5] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In *LPAR*, pages 151–165, 2007.

[6] Sylvain Boulmé. *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel.* Thèse de doctorat, Université Paris 6, 2000.

[7] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic, 1998.

[8] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In *In TLCA*, pages 102–117. Springer, 2007.

[9] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The maude ltl model checker. *Electronic Notes in Theoretical Computer Science*, 71(0) :162 – 187, 2004.

[10] Stéphane Fechter. *Sémantique des traits orientés objet de* FOCAL. Thèse de doctorat, Université Paris 6, Juillet 2005.

[11] W. A. Howard. The formulæ-as-types notion of construction. In *The Curry-Howard Isomorphism.* 1969.

[12] caml@inria.fr. Ocaml. `http://caml.inria.fr`.

[13] focalize-devel@inria.fr. Focalize. `http://focalize.inria.fr`.

[14] P. Lincoln M. Clavel, S. Eker and J. Meseguer. Principles of maude. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[15] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, 1998.

[16] Virgile Prevosto. *Conception et Implantation du langage FoC pour le développement de logiciels certifiés.* Thèse de doctorat, Université Paris 6, September 2003.

[17] Virgile Prevosto and Boulmé Sylvain. Proof contexts with late binding. In Pawel Urzyczyn, editor, *TLCA*, volume 3461 of *LNCS*, pages 324 – 338, Nara, Japan, April 2005. Springer.

[18] The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr/`.

[19] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety proof for multiple inheritance in c++. In *OOPSLA '06 : Object oriented programming, systems, languages, and applications.* ACM Press, 2006.

9 Annexe : Les règles de typage du $\lambda\Pi$-calcul modulo

$$\Gamma, \Delta ::= \cdot \mid \Gamma, x{:}A$$

$\boxed{\Gamma \text{ WF}}$ Le contexte Γ est bien formé

$$(empty)\ \frac{}{\cdot \text{ WF}} \qquad (decl)\ \frac{\Gamma \text{ WF} \qquad \Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x{:}A \text{ WF}}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$\boxed{\Gamma \vdash M : A}$ Dans le contexte Γ, le terme M est bien typé de type A.

$$(sort)\ \frac{\Gamma \text{ WF}}{\Gamma \vdash \mathsf{Type} : \mathsf{Kind}} \qquad (var)\ \frac{\Gamma \text{ WF} \qquad x{:}A \in \Gamma}{\Gamma \vdash x : A}$$

$$(prod)\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x{:}A \vdash B : s}{\Gamma \vdash \Pi x{:}A.\ B : s}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$$(abs)\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x{:}A \vdash B : s \qquad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\ M : \Pi x{:}A.\ B}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$$(app)\ \frac{\Gamma \vdash M : \Pi x{:}A.\ B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : \{N/x\}B}$$

$$(conv)\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}\ A \equiv B$$