BBFoC

Stéphane Fechter
Laboratoire d'informatique de Paris 6
8 rue du Capitaine Scott
75000 Paris, France
Stephane.Fechter@lip6.fr

Olivier Boite
Laboratoire CEDRIC/IIE du CNAM
18, allée Jean Rostand
91025 Evry, France
boite@cnam.fr

Résumé

Cet article décrit un modèle d'un langage de programmation et de spécifications de structures algébriques pour le calcul formel. Ce modèle comporte un aspect orienté objet, et une notion nouvelle de type support pour les entités manipulées. L'article décrit la preuve mécanisée de la sûreté du typage du modèle avec le système Coq.

1 Introduction

Le projet FoC [FoC] (acronyme pour Formel OCaml Coq) a pour objectif de développer un environnement certifié pour le calcul formel. Cet objectif se décompose en deux sous-buts. Le premier est de réaliser une librairie certifiée pour le calcul formel. Le second est d'offrir un langage utilisateur, appelé langage FoC, permettant de spécifier des structures algébriques, d'implanter les algorithmes nécessaires au calcul formel et de prouver formellement les propriétés des structures et des algorithmes. La librairie est écrite à l'aide du langage FoC.

Par souci de réutilisation du code et des preuves, et pour être proche des méthodes de construction des mathématiques, le langage FoC est un langage fonctionnel qui propose des traits orientés objet comme l'héritage, la liaison retardée et le raffinement. Le projet FoC nous offre un ensemble d'outils permettant de construire des structures mathématiques par raffinement, de les visualiser et d'analyser leurs dépendances. Les éléments de spécifications et de preuves d'un programme FoC sont compilés vers le système d'aide à la preuve Coq [BBC+02], tandis que les éléments calculatoires sont compilés vers le langage

OCAML [LDG+02]. Ce dernier a été choisi pour ses caractéristiques orientées objet et modulaires [BHR99].

Ici seul le langage FoC, et plus particulièrement les traits orientés objet, nous intéressent. Une sémantique formelle pour FoC a été définie à l'aide du modèle Objective FoC [Fec01], [Fec02a], [Fec02b] qui présente les éléments du langage FoC, vu comme un langage de programmation. Les travaux présentés dans cet article suivent trois objectifs. Le premier est de présenter la sémantique et le système de types d'un sous-ensemble d'Objective FoC : BBFoC. Cela permet d'assurer au projet FoC, que le langage repose sur des bases solides, sûres et non contradictoires. Le deuxième consiste à présenter la formalisation et la preuve de la sûreté de ce système de types en COQ. Enfin, le troisième objectif s'insère dans une étude de la réutilisation de preuves formelles en sémantique [BD01, Boi02]. Les travaux présentés ici constituent une étude de cas supplémentaire permettant de dégager les élements nécessaires à l'étude de langages orientés objet. Notre développement en COQ s'insère dans la famille des travaux sémantiques mécanisés autour des langages orientés objet comme [NvOP00, NvO98], [Sym99], [Lau97], [MIRO].

La structure de l'article est la suivante : à la section 2 nous présentons BBFoC. Nous y exposons les éléments du langage FoC, puis la syntaxe, la sémantique et le typage. La section 3 présente la formalisation de la sûreté du typage en COQ.

2 Présentation de BBFoC

Le langage FoC introduit la notion d'**espèce**. Celle-ci permet de représenter des structures mathématiques comme par exemple les groupes, les anneaux, les polynômes. Dans une espèce, les opérations manipulant les éléments de la structure sont appelées **méthodes**, et **déclarations** quand elles sont abstraites. Les éléments d'une structure mathématique donnée ont une représentation commune. Dans le langage FoC, elle est caractérisée par un type nommé **type support**. Dans une espèce, ce type est plus ou moins abstrait selon le niveau d'abstraction de la structure que nous voulons spécifier. Une espèce est analogue à la notion de classe des langages orientés objet. Ainsi on utilise un mécanisme d'héritage pour définir de nouvelles espèces à partir d'espèces existantes. Ce mécanisme d'héritage présente, par rapport à un modèle plus classique, quelques restrictions dû à la présence du type support souvent associé aux méthodes et aux déclarations. En outre, la redéfinition du type support est interdite car en calcul formel, même si l'on construit des structures de plus en plus riches, l'ensemble sous-jacent reste le même et par conséquent la représentation des éléments de cet ensemble est fixe. L'utilisateur final (par exemple un ingénieur) manipule uniquement des **collections**. Il s'agit d'une instanciation d'une espèce où toutes les déclarations ont été matérialisées en méthodes, et où le type support est totalement défini. Une collection est totalement figée. De ce fait, il est impossible pour une collection de redéfinir et/ou d'hériter une méthode. L'utilisateur envoie (ou invoque) une méthode de la même manière que l'on envoie un message sur un objet. Ainsi il interagit indirectement sur les éléments de la collection grâce à l'abstraction du type support. En pratique, on peut voir la collection comme une sorte de calculatrice dont le fonctionnement est totalement caché à l'utilisateur.

La sémantique du langage FoC vu en tant que langage de programmation, est étudiée au travers du modèle Objective FoC [Fec01, Fec02b]. Ce modèle propose une sémantique opérationnelle et spécifie son système de types. Il est inspiré d'Objective ML [RV98] modèle que l'on peut rapprocher de OCAML. Dans le cadre de cet article nous nous intéressons à un sous-ensemble d'Objective FoC : BBFoC. Les différences entre les deux modèles sont données dans le tableau suivant :

Traits	BBFoC	Objective FoC
collections	oui	oui
espèces	avec simple héritage	avec multi-héritage
espèces paramétrées	non	oui
déclarations	oui	oui
méthodes	oui	oui
redéfinition de méthodes	oui	oui
héritage du type support	oui	oui
liaison retardée	oui	oui
self référence	oui	oui
typage	monomorphe	polymorphe

Dans la suite, nous présentons la syntaxe, le langage de types, la sémantique opérationnelle et les règles de typage de BBFoC, ce qui nous permet de caractériser formellement les notions d'espèce, de collection et de type support.

2.1 Syntaxe

Les expressions du langage donné sont décrites formellement dans la figure 1.

Dans ce qui suit, nous supposons un ensemble de constantes $c \in \mathcal{C}$, un ensemble de noms de variables $x \in \mathcal{X}$ et un ensemble de noms de méthodes $m \in \mathcal{M}$. On suppose aussi que pour l'élément de construction syntaxique $(m_1 = a_1 , \ldots , m_n = a_n)$, tous les m_i sont distincts deux à deux. Nous avons la même hypothèse pour la construction $(m_1 : \tau_1 , \ldots , m_n : \tau_n)$.

La syntaxe met en évidence trois catégories d'expressions :

(1) les expressions fonctionnelles de base :
La seule différence avec ML, porte sur la construction $fun(x\ in\ a_1)a_2$. Elle comporte une restriction par rapport à l'abstraction classique. Ici, comme on le verra par la suite, par contrainte de typage, a_1 devra être

FIG. 1 – *syntaxe de BBFoC*

$$
\begin{array}{lll}
a & ::= & c \mid x \mid fun(x\ in\ a)\ a \mid a\ a \qquad (1)\\
 & \mid & self \mid (\tau_b;\ m_1 = a_1, \ldots, m_n = a_n) \mid a\sharp m \mid new\ a \qquad (2)\\
 & \mid & species\ z = a\ in\ a \qquad (3)\\
 & \mid & \begin{array}{l} struct \\ \quad \delta \\ \quad (m'_1 : \tau_1\ , \ldots, m'_k : \tau_k) \\ \quad (m_1 = a_1\ , \ldots, m_n = a_n) \\ end \end{array} \quad \mid \quad \begin{array}{l} struct\_h \\ \quad inherit\ a \\ \quad \delta \\ \quad (m'_1 : \tau_1\ , \ldots, m'_k : \tau_k) \\ \quad (m_1 = a_1\ , \ldots, m_n = a_n) \\ end \end{array}
\end{array}
$$

une collection. Le paramètre x a pour type le type support de a_1.

Dans la suite, on continue d'appeler *abstraction* cette construction particulière.

(2) les expressions de collections :
Une collection prend la forme syntaxique $(\tau_b; m_1 = a_1, \ldots, m_n = a_n)$, les m_i étant éléments de $\mathcal{M}$. Le type support de la collection est dénoté par τ_b dont la description est détaillée plus loin dans le texte.
La variable $self$ permet de de référencer la collection elle-même, et de mettre en œuvre la liaison retardée.
L'invocation d'une méthode m d'une collection a est dénotée par $a\sharp m$ (la construction rappelle celle des langages orientés objet).
Enfin la construction *new a* permet de créer une collection à partir d'une espèce (comme dans les langages orientés objet).

(3) les expressions d'espèces :
La construction des espèces peut se faire à l'aide de deux formes syntaxiques. La forme :

$$
\begin{array}{l} struct \\ \quad \delta \\ \quad (m'_1 : \tau_1, \ldots, m'_k : \tau_k) \\ \quad (m_1 = a_1\ , \ldots, m_n = a_n) \\ end \end{array}
$$

qu'on appelle **espèce simple**, permet de créer directement une espèce. Dans cette construction $(m_1 = a_1\ , \ldots, m_n = a_n)$, représente la liste des méthodes, et $(m'_1 : \tau_1, \ldots, m'_k : \tau_k)$ la liste des déclarations. Le corps d'une méthode peut utiliser une déclaration. Le champ δ représente la définition (donnée plus loin) du type support.

La forme :
struct_h
inherit a
δ
$(m'_1 : \tau_1, \ldots, m'_k : \tau_k)$
$(m_1 = a_1 \ , \ldots, \ m_n = a_n)$
end
représente une espèce avec héritage d'une autre espèce. Pour l'héritage, on utilise la phrase *inherit* a; où a est une espèce. On hérite alors des méthodes de l'espèce a ainsi que des déclarations dont la définition n'est pas donnée dans l'espèce courante. Les autres champs ont la même signification que ceux de la construction précédente. En revanche, le champ δ peut être absent. Dans ce cas, le type support de l'espèce est hérité de l'espèce a.
Une espèce peut se définir localement par la construction *species* $z = a$ *in* a. La signification de cette construction est la même que celle de la construction classique *let* $x = a$ *in* a de ML.

Enfin, le type support est soit syntaxiquement absent ($\varnothing$), soit défini, c'est alors une expression de type à la ML (τ_b) construite ici à partir des types de base (*const_t*) et de la flèche.

$$\begin{array}{lcl} \delta & ::= & \varnothing \mid \tau_b \\ \tau_b & ::= & const\_t \mid \tau_b \rightarrow \tau_b \end{array}$$

2.2 Sémantique opérationnelle de BBFoC

La sémantique que nous présentons ici est une sémantique opérationnelle à réduction. Une telle sémantique permet de visualiser chaque étape de l'évaluation d'une expression vers son résultat final (sa valeur) s'il existe. Une étape élémentaire est donnée par la relation $\rightarrow_\epsilon$ permettant de réduire en tête du terme. Une expression qui ne peut pas se réduire avec cette relation, est réduite en "profondeur" grâce à la règle du contexte. Elle permet de dégager les sous-expressions qui se réduisent avec la relation $\rightarrow_\epsilon$.
Nous utilisons les notations suivantes pour le reste de l'article :
- w est une abréviation d'une liste de méthodes.
- D est une abréviation d'une liste de déclarations.
- $dom(A)$ renvoie le domaine d'une relation A.

Les valeurs sont données ci-dessous :

$$\begin{array}{lcll} v & ::= & c & \text{constante} \\ & \mid & fun(x \ in \ a_1) \ a_2 & \text{abstraction} \\ & \mid & (\tau_b; w) & \text{collection} \\ & \mid & struct \ \delta \ D \ w \ end & \text{espèce simple} \end{array}$$

Remarquons que l'expression a_1 qui contraint le paramètre de l'abstraction n'a pas besoin d'être une valeur car elle représente une contrainte sur le paramètre.

Réductions élémentaires

La définition de $\rightarrow_\epsilon$ est donnée par les règles (1) à (5).

$$(fun(x\ in\ a_1)\ a_2)\ v \quad \rightarrow_\epsilon \quad a_2[v/x] \tag{1}$$

$$species\ z = v\ in\ a \quad \rightarrow_\epsilon \quad a[v/z] \tag{2}$$

$$new(\ struct\ \tau_b\ w\ end\ ) \quad \rightarrow_\epsilon \quad (\tau_b; w) \tag{3}$$

$$(\tau_b; w\ )\sharp m \quad \rightarrow_\epsilon \quad w(m)[(\tau_b; w\ )/self] \tag{4}$$

$$\begin{array}{l} struct\_h \\ \quad inherit \\ \qquad (\ struct\ \ \delta_1\ \ D_1\ \ w_1\ \ end\ ) \\ \quad \delta_2 \\ \quad D_2 \\ \quad w_2 \\ end \end{array} \quad \rightarrow_\epsilon \quad \begin{array}{l} struct \\ \quad \delta \\ \quad D_1 \oplus D_2 \setminus dom(w_1 \oplus w_2) \\ \quad w_1 \oplus w_2 \\ end \end{array} \tag{5}$$

Avec $\delta = \begin{cases} \delta_1 \text{ si } \delta_2 = \varnothing \\ \delta_2 \text{ sinon} \end{cases}$

Les règles (1) et (2) sont les règles classiques de β-réduction. La règle (3) met en évidence que la création d'une collection est faite à partir d'une espèce sans héritage dont le type support est défini.

L'invocation d'une méthode m d'une collection consiste à retourner le corps de cette méthode où toutes les occurences de la variable $self$ sont remplacées par la collection elle-même (règle (4)).

La règle (5) exprime la réduction de l'héritage d'une espèce. Pour cela on utilise l'opérateur binaire $\oplus$. Il permet d'ajouter un élément e dans un ensemble A. Si e est déjà présent dans A, alors $\oplus$ le remplace par la nouvelle définition de e. Par abus d'écriture, $A \oplus l_e$ désignera l'ajout des éléments de la liste l_e dans l'ensemble A. Cet opérateur est décrit plus formellement dans la section consacrée aux tables. Ainsi on utilise $\oplus$ afin de fusionner la liste des méthodes de l'espèce héritée et la liste des méthodes de l'espèce qui est réduite. De même on utlise $\oplus$ pour la déclaration $D_1 \oplus D_2$. On enlève ensuite de liste obtenue, les déclarations qui ont reçus une définition $dom(w_1 \oplus w_2)$. Enfin on résoud l'héritage du type support : si l'espèce ne définit pas de type support, on hérite du type support de l'espèce héritée. Sinon le type support résultant est celui défini dans l'espèce.

Réduction par contexte

Un contexte E est une expression avec un "trou" $\bullet$. Ce trou est destiné à être rempli par une autre expression a. Cette opération se note $E[a]$. Les contextes de réduction sont les suivants :

$$
\begin{array}{lcll}
E & ::= & \bullet \mid E\ a \mid v\ E \mid E\sharp m \mid & struct\_h \\
& & & \quad inherit\ E \\
& & & \quad \delta \\
& & & \quad D \\
& & & \quad w \\
& & & end
\end{array}
$$

où $\bullet$ représente le contexte vide. La relation de réduction entre expressions, noté $\rightarrow$, est définie par :

$$
\frac{a \rightarrow_{\epsilon} a'}{E[a] \rightarrow E[a']} \qquad \frac{a \rightarrow a'}{E[a] \rightarrow E[a']}
$$

2.3 Langage de types

Le langage des types est défini par :

$$
\begin{array}{lcl}
\tau & ::= & \alpha \mid \tau_b \mid \tau \rightarrow \tau \mid \langle \tau_b; m_1 : \tau_1 \ , \ldots, m_n : \tau_n \rangle \\
& \mid & sup(\delta)sig(\tau)(m_1 : \tau_1 \ , \ldots, m_n : \tau_n)(m'_1 : \tau'_1 \ , \ldots, m_k : \tau'_k)
\end{array}
$$

où :

- α est une variable de type.
- dans la construction $m_1 : \tau_1 \ , \ldots, m_n : \tau_n$, les m_i sont distincts deux à deux.
- dans la construction $m'_1 : \tau'_1 \ , \ldots, m_k : \tau'_k$, les m'_i sont distincts deux à deux.

Le type d'une collection est de la forme $\langle \tau_b; m_1 : \tau_1 \ , \ldots, m_n : \tau_n \rangle$ où τ_b est le type support de la collection et $m_1 : \tau_1 \ , \ldots, m_n : \tau_n$ est la liste des types des méthodes.

Le type $sup(\delta)sig(\tau)(m_1 : \tau_1 \ , \ldots, m_n : \tau_n)(m'_1 : \tau'_1 \ , \ldots, m_k : \tau'_k)$ est celui d'une espèce, où δ est le type support défini par l'espèce (directement ou par héritage). Le champ $m_1 : \tau_1 \ , \ldots, m_n : \tau_n$ représente la liste des types des méthodes (définies directement dans l'espèce et/ou par héritage). Puis le champ $m'_1 : \tau'_1 \ , \ldots, m_k : \tau'_k$ représente la liste des déclarations (définies directement dans l'espèce et/ou par héritage). Enfin le champ $sig(\tau)$ est ce qu'on appelle la **signature** de l'espèce : elle représente le type de la collection construite à partir de l'espèce qu'on est en train de typer. Entre autres, elle permet de typer la variable $self$ dans les méthodes de l'espèce.

2.4 Typage

Les séquents de typage ont la forme $\Gamma \vdash a : \tau$ où Γ est un environnement de typage défini formellement par :

$$\Gamma ::= \varnothing \mid \Gamma \oplus x : \tau \mid \Gamma \oplus self : \tau$$

Nous utilisons également les notations suivantes :

- ω est une abréviation d'une liste de types de méthodes.
- Δ est une abréviation d'une liste de déclarations.

Les règles de typage pour les expressions de base sont les suivants :

$$\textsc{Var}\quad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad\qquad \textsc{Self}\quad \frac{}{\Gamma \vdash self : \Gamma(self)}$$

$$\textsc{Abstraction}\quad \frac{\Gamma \vdash a_1 : \langle \tau_b ; m_1 : \tau_1 \ , \ldots , m_n : \tau_n \rangle \qquad \Gamma \oplus (x : \tau_b) \vdash a_2 : \tau'}{\Gamma \vdash fun(x \ in \ a_1) a_2 : \tau_b \to \tau'}$$

$$\textsc{App}\quad \frac{\Gamma \vdash a : \tau' \to \tau \qquad \Gamma \vdash a' : \tau'}{\Gamma \vdash a \ a' : \tau}$$

Seule l'abstraction diffère du schéma classique. Ici le paramètre est contraint à être un élément d'une collection a_1, son type est donc le support τ_b de cette collection. C'est donc dans l'environnement Γ augmenté de l'hypothèse $x : \tau_b$, que le corps de l'abstraction est typé.

$$\textsc{Collection}\quad \frac{(\forall m \in dom(w), \ \Gamma \oplus (self : \langle \tau_b ; \omega \rangle) \vdash w(m) : \omega(m)) \qquad dom(\omega) = dom(w)}{\Gamma \vdash (\tau_b ; w) : \langle \tau_b ; \omega \rangle}$$

Pour typer une collection, on type l'expression associée à chaque méthode dans l'environnement Γ augmenté de la variable $self$. Le type associé à $self$ est le type de la collection elle-même. D'autre part, on autorise les appels de méthodes sur une collection, seulement si elles sont définies. Pour respecter cette contrainte, les noms des méthodes dans le type de la collection doient être dans la collection. Formellement, ceci est exprimé par $dom(\omega) = dom(w)$.

$$\textsc{Send}\quad \frac{\Gamma \vdash a : \langle \tau_b ; \omega \rangle \qquad \omega(m) = \tau}{\Gamma \vdash a \sharp m : \tau}$$

L'envoi d'un message m sur une expression a est correct si le type de a est celui d'une collection. La condition $\omega(m) = \tau$ dans la règle ci-dessus impose que m soit présent dans le type de la collection. Le type qui lui est associé est le type du message. Par contre on ne vérifie pas que m appartient à l'expression a, car la méthode m peut être une déclaration (virtuelle dans le contexte orienté objet).

$$\textsc{New}\ \frac{\Gamma \vdash a : sup(\tau_b)sig(\ \langle \tau_b ; (m_1 : \tau_1\ , \ldots, m_n : \tau_n) \rangle\ )(m_1 : \tau_1\ , \ldots, m_n : \tau_n)(\varnothing)}{\Gamma \vdash new\ a : \langle\ \tau_b ; (m_1 : \tau_1\ , \ldots, m_n : \tau_n)\ \rangle}$$

La création d'une collection à partir d'une espèce est correcte à condition que l'espèce ait défini un type support et que toutes ses méthodes soient définies. Cela se traduit par :

- le type support de la signature et celui ramené par le type de l'espèce doivent être identiques.
- la liste des déclarations est vide
- la liste des types des méthodes de la signature et celle ramenée par le type de l'espèce sont identiques.

Le typage d'une espèce définie par héritage se fait avec la règle Species Body dont nous allons analyser la construction :

- On type l'espèce héritée afin de ramener son type support (δ_h), les types de ses méthodes (Φ_h) et de ses déclarations (Δ_h) (ligne (a)).
- On type les méthodes de l'espèce courante (w) dans l'environnement Γ augmenté de la variable $self$. Le type de $self$ doit être celui de la signature du type retourné pour l'espèce. Ceci permet de typer les occurrences de la variable $self$ dans le corps de chaque méthode de w. Enfin il faut vérifier que les noms de méthodes de w sont inclus dans la liste des méthodes définissant le type de $self$ (ligne (c)).
- On résout l'héritage du type support, entre le type support hérité et celui de l'espèce courante, en utilisant l'opérateur V (ligne (b)). Si le type support de l'espèce courante est dans l'état non défini, on retourne naturellement celui de l'espèce héritée. Dans le cas contraire, il faut qu'il soit identique à celui de l'espèce héritée.
  Enfin, il faut assurer la cohérence du type support δ_f et celui de la variable $self$ (δ_s) : δ_f est soit non défini, soit égal à δ_s (prémisse (1), ligne(b)).

Ces étapes permettent de retourner le type de l'espèce avec :

- le type support calculé par l'opérateur V,
- la signature correspondant au type de $self$,
- la concaténation de la liste des types des méthodes héritée avec la liste des types des méthodes dans w. Mais il se peut qu'une méthode héritée soit redéfinie dans l'espèce courante. Dans ce cas, il faut que les types soient égaux. Cette vérification est faite avec la prémisse (2) (ligne (d)).
- la concaténation des déclarations héritées (Δ_h) avec les déclarations de l'espèce (D). Mais il faut retirer de cette nouvelle liste, les déclarations qui ont été définies et sont devenues des méthodes. De plus, les déclarations

de l'espèce doivent se retrouver dans la liste des types des méthodes de la variable $self$. Cette contrainte s'exprime avec la prémisse (3) (ligne (f)). Enfin il se peut qu'une déclaration se retrouve dans la liste des déclarations de l'espèce. Dans ce cas, il faut que leurs types soient égaux. La prémisse (4) permet cette dernière vérification (ligne (e)).

Enfin, les méthodes héritées ne peuvent pas redevenir des déclaraions. Ceci est une contrainte du langage FoC qui impose que les constructions se font d'un niveau abstrait vers un niveau concret, et non l'inverse. La prémisse (5) (ligne (f)) permet d'imposer cela.

SPECIES BODY

$$\frac{\begin{array}{lr}
\Gamma \vdash a_h : sup(\delta_h)sig(\ \langle \delta_s ; \omega_s \rangle\ )(\Phi_h)(\Delta_h) & (a)\\
V(\delta_h,\delta) = \delta_f \quad\ \ \delta_f = \delta_s \vee \varnothing\ _{(1)} & (b)\\
(\ \forall m \in dom(w),\ \Gamma \oplus (\ self : \langle \delta_s ; \omega_s \rangle\ ) \vdash w(m) : \omega_s(m)\ ) \quad\ dom(w) \subseteq dom(\omega_s) & (c)\\
m \in dom(\Phi_h) \wedge m \in dom(w) \Rightarrow \Phi_h(m) = \omega_s(m)\ _{(2)} & (d)\\
(m' : \tau) \in \Delta_h \wedge (m' : \tau) \in D \Rightarrow \tau = \tau'_{(4)} & (e)\\
D \subseteq \omega_s\ _{(3)} \quad\ D \not\subseteq \Phi_h\ _{(5)} & (f)
\end{array}}{\Gamma \vdash \begin{array}{l} struct_h \\ \quad inherit\ a_h \\ \quad \delta \\ \quad D \\ \quad w \\ end \end{array} \quad : sup(\delta_f)sig(\ \langle \delta_s ; \omega_s \rangle\ )(\Phi)(\Delta)}$$

avec

$$\begin{array}{lcl}
\Phi & = & \Phi_h \oplus \{w(m) : \omega_s(m)\ /\ m \in dom(w)\} \\
\Delta & = & \Delta_h \oplus D \setminus dom(\Phi) \\
V(\delta_h,\delta) & = & \left\{ \begin{array}{l} \delta_h \text{ si } \delta = \varnothing \\ \delta \text{ avec } \delta = \delta_h \end{array} \right.
\end{array}$$

SIMPLE SPECIES BODY

$$\frac{\begin{array}{l}
\Gamma \oplus (self : \langle \tau_b ;\ \omega_s \rangle) \vdash w_0 : \Phi \\
\delta = \tau_b \vee \varnothing \qquad D \subseteq \omega_s \qquad dom(w_0) \cap D = \varnothing\ _{(*)}
\end{array}}{\Gamma \vdash \begin{array}{l} struct \\ \quad \delta \\ \quad D \\ \quad w_0 \\ end \end{array} : sup(\delta)sig(\langle \tau_b ;\ \omega_s \rangle)(\Phi)(D)}$$

La règle de typage d'une espèce simple est similaire à celle pour l'espèce avec héritage. Notons que la liste des déclarations doit appartenir à la liste des types des méthodes de la variable $self$. Ceci permet d'assurer la cohérence des déclarations avec leur future instance. Enfin l'intersection de la liste des méthodes et des déclarations doit être vide. Ceci est nécessaire pour rejeter l'exemple suivant qui appelle la construction *new* sur une espèce contenant encore des déclarations :

$$
\begin{array}{ll}
(new\ (& struct \\
& \quad int \\
& \quad m : int \rightarrow int \\
& \quad m = fun(x\ in\ c)\ x + 1 \\
& end\ ) \\
) &
\end{array}
$$

où c est une collection dont le type support est int. Cette expression est bien typée si on ne tient pas compte de la prémisse $(*)$. Mais elle ne peut être réduite par $\rightarrow_\epsilon$ (règle (3)), étant donné que la liste des déclarations n'est pas vide.

Local species

$$
\frac{\Gamma \vdash a_1 : sup(\delta)sig(\tau)(\omega)(\delta) \qquad \Gamma \oplus (\ z : sup(\delta)sig(\tau)(\omega)(\delta)\ ) \vdash a_2 : \tau}{\Gamma \vdash species\ z = a_1\ in\ a_2 : \tau}
$$

Le typage de la définition locale d'une espèce est très similaire à celui du $let \ldots in$ dans le cas du typage monomorphe. La règle contraint a_1 à être une espèce.

3 Sûreté du typage en Coq

Dans cette section, nous nous intéressons à la formalisation en Coq de BB-FoC, et à la preuve de la sûreté du typage pour une sémantique à réduction. Cette preuve est formelle et mécanisée dans le système d'aide à la preuve Coq [BBC+02], dont nous supposons que le lecteur a une connaissance minimale. Ce travail bénéficie d'une première expérience pour une preuve similaire d'un langage à la ML [BD01].

3.1 Spécifications du langage BBFoC en Coq

Outre la définition des expressions, types, règles de typage, et de la sémantique, nous détaillons la description d'une structure `Table`, déjà présentée dans [BD01]. Cette table servira à spécifier les environnements de typage, les listes de méthodes ou déclarations, ainsi que leur type.

3.1.1 Table

D'un point de vue abstrait, une table est une table d'association entre des clés et des valeurs. Concrètement, souvent, le choix est fait de l'implémenter à l'aide d'une liste de couples. Un gros défaut de cette représentation est l'opération d'accès à partir d'une clé, encore appelé application, ainsi que l'opération d'ajout avec les redondances éventuelles à gérer. Ces opérations nécessitent une recherche séquentielle dans la liste.

Notre table est une structure abstraite dans le sens où on ne la manipule qu'à travers des opérateurs d'ajout, d'appartenance, et d'application. Le type des tables (et de ces opérations) est générique, paramétré par les types des clés et des valeurs. Ces paramètres sont représentés localement à l'intérieur d'une section par des variables A et B, et deviennent des paramètres à instancier en dehors de la section. Il s'agit d'un type inductif en Coq, ne possédant qu'un seul constructeur ayant pour arguments une liste de clés qui est le domaine de la table, et une fonction des clés vers les valeurs.

```
Inductive table : Set :=
   intro_table : list → (A → B) → table.
```

Nous utilisons la fonctionnalité de Coq pour trouver une valeur associée à une clé, ce qui nous évite une recherche séquentielle.

Une table d'association possède deux propriétés importantes, concernant l'opération d'ajout notée $\oplus$:

$$(t \oplus (c,v_1)) \oplus (c,v_2) = (t \oplus (c,v_2))$$

$$(t \oplus (c_1,v_1)) \oplus (c_2,v_2) = (t \oplus (c_2,v_2)) \oplus (c_1,v_1) \quad si\ c_1 \neq c_2$$

Elle ne doit pas contenir deux occurrences d'une même clé, donc l'ajout dans une table doit écraser l'ancienne valeur. La deuxième propriété est la commutativité. Autrement dit, l'ordre des ajouts n'a pas d'importance pour des clés différentes. Ces propriétés sont vérifiées par notre implantation fonctionnelle d'une table, et par notre opération d'ajout strict (pas de redondance dans le domaine d'une table).

Une fonction toute seule ne suffit pas à représenter une table. En effet, en Coq les fonctions sont totales et une table d'association est partielle. Donc se pose le problème de l'application d'une clé hors du domaine. Or notre structure possède une liste de clés qui représente justement le domaine de la table. Notre opération d'application est en fait un prédicat qui juge si l'application d'une clé dans une table retourne la valeur v, en s'assurant que la clé appartient à la liste, et que v est bien la valeur retournée par la fonction encapsulée.

```
Inductive apply_table : table → A → B → Prop :=
 intro_apply_table : (a:A)(dom:list)(f:A→B)
        (in_prop a dom)→
        (apply_table (intro_table dom f) a (f a)).
```

Une table vide est une table dont la liste des clés est vide :

```
Definition empty_table [f:A→B] : table := (intro_table Nil f).
```

Nous avons ensuite prouvé une quinzaine de lemmes pour manipuler une table. En instanciant A et B par le type des variables et des types de notre langage, nous modélisons un environnement de typage. Cette structure permet aussi de représenter des enregistrements. Ainsi, une collection peut être représentée par une table instanciée sur le type des noms de méthode et le type des expressions. De même, une collection est représentée par une table entre noms de méthode et types.

3.1.2 Expressions

La grammaire des expressions est représentée par le type inductif `expr`. Le type `message` est un type abstrait pour les noms de méthodes.
Le type `basic` est un type inductif codant les types de base. Le type `carrier` est un type inductif pour le type support qui peut être soit `non_def`, soit défini à l'aide de types de base.

Pour rester concis, nous ne donnerons pas tous les constructeurs de `expr`. Pour simplifier la gestion des ajouts et suppressions de méthodes dans la liste des déclarations et des définitions, lors de l'héritage, nous avons choisi d'avoir une construction pour l'héritage qui ne comporte qu'une déclaration et qu'une définition de méthode, au lieu de listes comme nous l'avons présenté.

```
Definition declarations := (table message type).

Inductive expr : Set :=
...
 | Collection : basic → (table message expr) →  expr
 | Send : expr → message → expr
 | New  : expr → expr
 | LetSpecies : identifier → expr → expr → expr
 | Struct   : carrier  → declarations → (table message expr ) → expr
 | Struct_h : expr → carrier → message → type → message → expr → expr.
```

Nous utilisons un prédicat pour déterminer si une variable x est libre dans une expression.

```
Inductive free_ident [x:identifier] : expr → Prop
```

Comme pour l'application sur une table, notre opération de substitution n'est pas une fonction, mais un prédicat qui s'assure en passant qu'il n' y a pas de capture de variable en utilisant le prédicat précédent. Ainsi, nous n'avons pas de renommage à faire. `(subst_expr e x e' er)` signifie que er est le résultat de la substitution de x par e' dans e.

```
Inductive subst_expr : expr → identifier → expr → expr → Prop
```

3.1.3 Typage

L'algèbre de types suit la définition donnée dans la présentation.

```
Inductive type  : Set :=
   Const_t: basic → type
 | Var_t: stamp → type
 | Arrow: type  → type → type
 | Collection_t : basic → (table message type) → type
 | Species_t : carrier → type → (table message type) → type.
```

Le type `stamp` est un type abstrait pour les variables de type.

Les règles de typage sont représentées par un prédicat inductif `type_of`. Nous écrirons
`env |- e: t` pour `(type_of env e t)`. Des règles de *pretty-printing* dans le système Coq permettent d'utiliser cette notation par la suite.

```
Inductive type_of  : env_typ → expr → type → Prop :=
...
 | type_of_collection : (env:env_typ)(w:(table message expr))(lt:(list message))
                        (ft:message→type)(ts:basic)
     ((m:message)(wm:expr)(apply_table w m wm) →
      env(+)(self,(Collection_t ts (intro_table lt ft))) |- wm : (ft m)) →
      (dom_of w)=lt →
      env |- (Collection ts w) : (Collection_t ts (intro_table lt ft))
...
```

Le constructeur détaillé ci-dessus est la traduction de la règle d'inférence (Collection). L'opérateur `V` suit la définition donnée dans la section 2.

On définit également une relation d'ordre notée $\sqsubseteq$ sur les expressions qui servira dans nos preuves de sûreté de typage.

```
Definition less_typable [e1,e2:expr]: Prop :=
 (env:env_typ)(t:type) env |- e1 : t  →  env |- e2 : t.
```

La notation $e1 \sqsubseteq e2$, pour e_1 est moins typable que e_2, (en Coq `(less_typable e1 e2)`), signifie que tout typage de e_1 permet de typer e_2.

3.1.4 Sémantique

Comme annoncé, nous utilisons une sémantique à réduction, ou dite "à petits-pas". Nous devons décrire quelles sont les valeurs dans notre langage, et comment réduire nos expressions.

```
Inductive is_value : expr → Prop :=
  Cst_val : (c:constant)(is_value (Const c))
| Fun_val : (i:identifier)(e,a:expr)(is_value (Fun i a e))
| Collection_val : (t:basic)(w:lmeth)(is_value (Collection  t w))
| Struct_val : (t:carrier)(d:declarations)(w:lmeth)(is_value (Struct t d w)).
```

Les types `lmeth` et `declarations` sont définis respectivement par `(table message exp)`, `(table message type)`. Les réductions sont des règles de réécriture spécifiées par `red_epsilon`, que l'on n'utilisera que dans certains contextes.

```
Inductive red_epsilon : expr → expr → Prop :=
  beta1 : (x:identifier)(e,a,er,v:expr)(is_value v)→(subst_expr e x v er)→
          (red_epsilon (Apply (Fun x a e) v) er)
| beta2 : (z:identifier)(a,er,v:expr)(is_value v)→(subst_expr a z v er)→
          (red_epsilon (LetSpecies z v a) er)
| new_red : (w:lmeth)(tb:basic)
          (red_epsilon (New (Struct (Def (Def_type_basic tb)) empty_decl w)) (Collection tb
| send_red : (w:lmeth)(m:message)(er,wm:expr)(t:basic)
    (apply_table w m wm)→
    (subst_expr wm self (Collection t w) er)→
```

```
    (red_epsilon (Send (Collection  t w) m) er)
| struct_h_red : (t1,t2,t3:carrier)(w:lmeth)(m:message)(a:expr)(n:message)(t:type)
                 (d,decl:declarations)   t3=(coherence t1 t2) ->
         (remove_from_table d(+)(n,t) m decl) ->
                 (red_epsilon (Struct_h (Struct t1 d w) t2 n t m a)
              (Struct t3 decl w(+)(m,a))).
```

La fonction `is_coherent` fait le calcul du type support, comme décrit dans la section 2.3. `empty_decl` représente une table de déclarations vide. Le prédicat `(remove_from_table d m d')` établit que `d'` est la table `d` privée de la clé `m`.

Nous représentons les contextes de manière fonctionnelle, afin de pouvoir les appliquer facilement :

```
Definition context := expr → expr.

Inductive is_context : context → Prop :=
  hole      : (is_context ([x:expr] x))
...
| new_ctx   : (c:context) (is_context c) → (is_context ([x:expr] (New (c x))))
| species_ctx : (c:context)(z:identifier)(a:expr) (is_context c) →
                (is_context ([x:expr] (LetSpecies z (c x) a)))
| struct_h_ctx: (c:context)(e:expr)(m:message)(ts:carrier)(n:message)(t:type)
                (is_context c) → (is_context ([x:expr] (Struct_h (c x) ts n t m e))).
```

Ainsi la relation de réduction se résume à deux possibilités : une réduction de base, ou une réduction dans un contexte.

```
Inductive red : expr → expr → Prop :=
  tete  : (e1,e2:expr)
          (red_epsilon e1 e2)→(red e1 e2)
| cont  : (e1,e2:expr)(ctx:context) (red e1 e2) → (is_context ctx) →
          (red (ctx e1) (ctx e2)).
```

On définit le prédicat `is_reducible e` qui affirme l'existence d'une expression `e'` dans laquelle `e` puisse se réduire grâce à `red`. On définit également `red_star`, la fermeture réflexive et transitive de `red`.

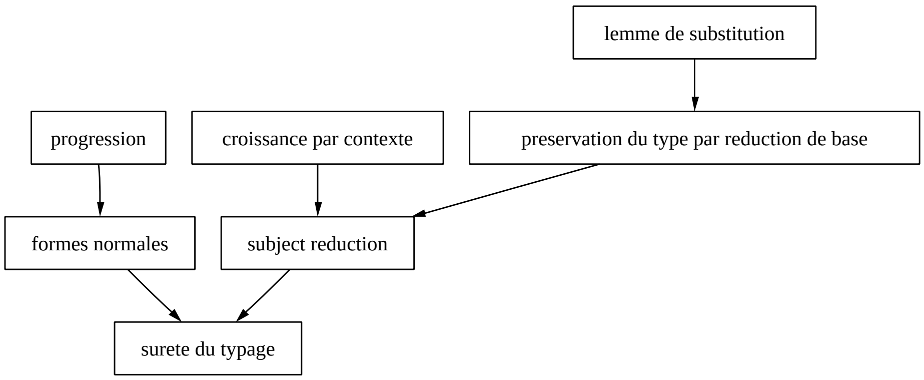
3.2 Preuve de la sûreté de typage

La sûreté du typage est une propriété fondamentale, particulièrement recherchée lors de la création d'un nouveau langage. En effet, elle assure que pour toute expression bien typée, si son évaluation termine, celle-ci termine sur une valeur. La version forte de ce théorème assure de plus que cette valeur est du même type que l'expression de départ.

On l'énonce en Coq comme suit.

```
Theorem type_safe :(a,a':expr)(t:type)
 (red_star a a') →
  empty_table |- a : t →
   ~(is_reducible a') →
    (is_value a')  ∧  ∃ env:env_typ, env |- a' : t.
```

Pour les langages typés, à noyau fonctionnel, le schéma de preuve est pratiquement toujours le même, et représenté ci-dessous.



La sûreté du typage est la conséquence du théorème des formes normales, et du théorème de "Subject-Reduction", énoncés ci-dessous :

```
Theorem well_typed_nf_are_value :(a:expr)(t:type)
 empty_table |- a : t -> ~(is_reducible a) -> (is_value a).

Theorem Subject_reduction : (a1,a2:expr)
 (red a1 a2) -> (less_typable a1 a2).
```

Le théorème `well_typed_nf_are_value` affirme qu'une expression qu'on ne peut plus réduire (en forme normale), bien typée dans un environnement vide, est une valeur. C'est un corollaire du lemme de progression suivant :

```
Lemma progression : (a:expr)(t:type)
 empty_table |- a : t -> ~(is_value a) -> (is_reducible a).
```

Le théorème `Subject_reduction` énonce la préservation du typage par réduction. Il se montre par induction sur la relation `red` : soit nous avons une réduction de base, soit nous avons une réduction dans un contexte. Le théorème est donc conséquence des deux lemmes suivants : la croissance de `less_typable` par passage au contexte (`less_typable_grows`), et la préservation du typage par $\longrightarrow_\epsilon$ (`type_preservation_by_red_epsilon`).

```
Lemma less_typable_grows :(c:context)(a1,a2:expr)
  (is_context c) -> (less_typable a1 a2) ->  (less_typable (c a1) (c a2)).

Lemma type_preservation_by_red_epsilon :(a1,a2:expr)
    (red_epsilon a1 a2) -> (less_typable a1 a2).
```

Le lemme `less_typable_grows` se montre par induction sur le contexte c, tandis que le lemme `type_preservation_by_red_epsilon` se montre par induction sur e. Dans le cas de β-réductions, nous utilisons un lemme de substitution, énoncé en Coq ci-après.

```
Lemma substitution_lemma:
```

```
(e1,e2,e:expr)(x:identifier)(env:env_typ)(t1,t:type)
 (subst_expr e1 x e e2) →
   env(+)(x,t) |- e1 : t1 →
     env |- e : t →
       env |- e2 : t1.
```

Ce lemme se montre par induction sur $e1$. Il assure la préservation du typage lors d'une substitution d'une variable par une expression du même type que cette variable.

Nous passons sous silence des lemmes de manipulation de l'environnement, et des lemmes auxiliaires qui ne présentent pas d'intérêt, sauf à mesurer la difficulté de la preuve.

La taille du script Coq pour la partie spécification du langage est de 400 lignes. Mais tout ce qui concerne les tables (définitions, opérations, et lemmes de manipulation), compte plus de 1300 lignes. La partie preuve de la sûreté du typage, ainsi que les lemmes nécessaires atteignent les 1700 lignes, mais qui ne mettent en œuvre que peu d'automatisation. Les fichiers Coq sont accessibles sur http://www.iie.cnam.fr/~boite/BBFoc.tgz

4 Conclusion

Nous avons formalisé et prouvé mécaniquement la sûreté du typage du modèle BBFoC. Ce modèle étant le noyau de celui de Objective FoC, nous avons une première validation de ce dernier. C'est le premier intérêt de ce travail. Cette formalisation est certes incomplète puisqu'elle ne prend pas en compte le polymorphisme, l'héritage multiple, les types supports abstraits, et le paramétrage des espèces. Cependant, ces aspects ne modifient profondément pas l'essence du langage. L'incorporation de ces aspects dans la formalisation en Coq ne remettrait pas en question le travail déjà fait. Le polymorphisme a déjà été traité pour un noyau fonctionnel de ML [Dub00], dans le système Coq. Cette formalisation est conséquente mais pourrait être reprise pour notre cas. L'aspect extension de langage et preuves en Coq fait l'objet d'études de cas et de formalisation [BD01, Boi02]. C'est le deuxième intérêt de ce travail : il constitue un élément de plus pour la formalisation de ce cadre d'extension. L'ajout des aspects de Objective Foc manquant à BBFoc pourraient à terme être réalisés dans ce cadre.

Remerciements

Nous remercions Catherine Dubois pour sa relecture et ses corrections qui ont permis de lever le voile sur des ambiguités et difficultés, ainsi que Luigi Liquori pour ses conseils.

Références

[BBC+02] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Munoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof-assistant reference manual.* INRIA, 7.3 edition, 05 2002. http://pauillac.inria.fr/coq/doc/main.html.

[BD01] O. Boite and C. Dubois. Proving Type Soundness of a Simply Typed ML-like Language with References. In R. Boulton and P. Jackson, editors, *Supplemental Proceedings of TPHOL 2001, Informatics Research Report EDI-INF-RR-0046 of University of Edinburgh*, pages 69–84, 2001.

[BHR99] S. Boulmé, T. Hardin, and R. Rioboo. Modules, objets et calcul formel. In *In Actes des Journées Francophones des Langages Applicatifs*, 1999.

[Boi02] O. Boite. Language Framework. draft, 2002.

[Dub00] C. Dubois. Proving ML Type Soundness Within Coq. *Theorem Proving in Higher Order Logics : 13th International Conference, TPHOLs 2000, Lecture Notes in Computer Science*, 1869:Springer–Verlag, 126–144, 2000.

[Fec01] S. Fechter. Une sémantique pour FoC. Rapport de D.E.A., Université Paris 6, Septembre 2001. available at http://www-spi.lip6.fr/~fechter.

[Fec02a] S. Fechter. An object-oriented model for the certified computer algebra library. Paper presented at FMOODS 2002 PhD workshop, March 2002. http://www-spi.lip6.fr/~fechter.

[Fec02b] S. Fechter. Objective FoC. draft, http://www-spi.lip6.fr/~fechter, 2002.

[Lau97] O. Laurent. Sémantique Naturelle et Coq : vers la spécification et les preuves sur les languages à objets. Technical Report 3307, INRIA, novembre 1997.

[LDG+02] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system release 3.04 Documentation and user's manual.* INRIA, 2002. http://pauillac.inria.fr/ocaml/htmlman/.

[MIRO] Action MIRÓ Avant-projet MIRO, 2002. http://www.loria.fr/equipes/miro/.

[NvO98] T. Nipkow and D. von Oheimb. Java-light is Type-Safe — Definitely. In L. Cardelli, editor, *Conference Record of the 25th Symposium on Principles of Programming Languages (POPL'98)*, pages 161–170, San Diego, California, 1998. ACM Press.

[NvOP00] T. Nipkow, D. von Oheimb, and C. Pusch. Java: Embedding a Programming Language in a Theorem Prover. *In F.L. Bauer and R. Steinbruggen, editors, Foundations of Secure Computation. IOS Press*, 2000.

[RV98] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):p. 27–50, 1998.

[Sym99] D. Syme. Proving Java Type Sound. *In Jim Alves-Foss, editor, Formal Syntax and Semantics of Java, volume 1523 of LNCS. Springer*, 1999.

[FoC] The FoC Project. http://www-spi.lip6.fr/˜foc.