Rapport Final

Terminaison de fonctions récursives pour l'atelier Focal

Présenté par :
William Bartlett

Encadrants :
M^me^ Dubois
M. Forest

7 septembre 2007

Table des matières

Chapitre 1

Introduction

1.1 Cadre du sujet

Les *méthodes formelles* (*par exemple* Méthode B, Coq) permettent de produire du code accompagné d'une certification, c'est-à-dire une preuve formelle assurant que celui-ci est valide vis-à-vis de sa spécification. Ces méthodes sont de plus en plus prisées dans les domaines d'application de l'informatique où la sécurité de personnes rentre en jeu (*par exemple* aéronautique, transport ferroviaire). Elles imposent des règles qui doivent être vérifiées par tout programme, et peuvent aussi assurer la correspondance entre spécification et implantation. Souvent le respect de ces règles se traduit par des preuves que le développeur doit décharger, les obligations de preuve.

Focal [Foc] est un atelier qui permet le développement de programmes certifiés. Cet atelier possède un langage de programmation incorporant des notions orientées objets et fonctionnelles. De plus, le développeur peut imposer des propriétés et en donner les preuves, le tout étant vérifié ultimement par un assistant à la preuve. Grâce à l'héritage il est possible de commencer par une spécification générale puis de préciser une ou plusieurs implantations au travers de raffinements successifs pour aboutir à du code exécutable.

Une des propriétés importantes imposée par la plupart des méthodes formelles est qu'*une fonction, pour être correctement définie, doit terminer.* Pour des fonctions simples, comme $x \mapsto x^2$, cette propriété est triviale. Mais dès que le corps d'une fonction contient des boucles ou que la fonction est récursive sa terminaison est moins évidente. Actuellement, Focal ne sait donner les preuves de terminaison que de certaines fonctions récursives. Le but de ce mémoire est d'augmenter l'espace des fonctions traitables par Focal.

Dans le cadre de ce stage je suis accueilli par l'équipe CPR (Conception et Programmation Raisonnées) du laboratoire CEDRIC. Cette équipe conduit ses recherches selon quatre axes majeurs :

- Développement d'un langage de spécification orienté objets (Focal)
- Réutilisation des preuves et des spécifications
- Compilation certifiée
- Certification de preuves de terminaison

1.2 Description du sujet

Certification de Focal par Coq. La compilation d'un fichier source Focal contenant une ou plusieurs *espèces* (structure proche d'une classe) produit deux fichiers. Premièrement, du code exécutable en OCaml, un langage de programmation fonctionelle orientée objets [oca]. Deuxièmement, un script de preuve vérifiable par Coq [coq], un assistant à la preuve en logique intuitionniste (Voir figure 1.1).

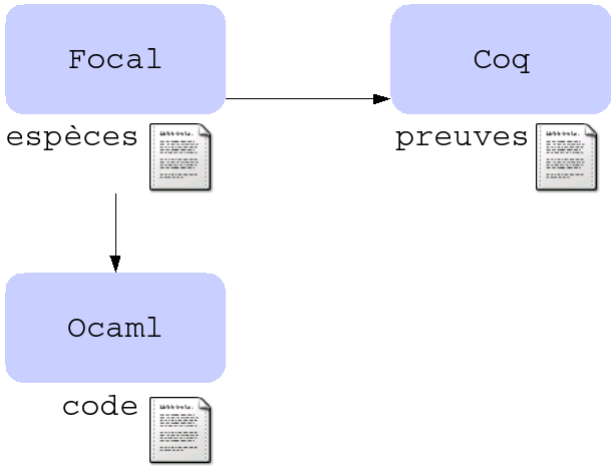

FIG. 1.1 – Compilation d'un fichier source Focal.

Terminaison de fonctions. Dans la version actuelle de Focal, le script de preuve Coq correspondant à une fonction récursive n'est correct que dans le cas d'une *induction structurelle* (voir table 1.1).

Cela est très restrictif, car un bon nombre de fonctions récursives ne se met pas sous la forme d'une induction structurelle en Focal (*par exemple* ex-

ponentiation rapide, tri rapide). La raison de cette restriction est que, jusqu'à tout récemment, Coq offrait seulement la possibilité de déclarer facilement des fonctions récursives par induction structurelle en utilisant le mot clé **Fixpoint**. Dans tous les autres cas il était très difficile de définir la fonction récursive en Coq.[1] Ceci a conduit au développement d'une nouvelle construction en Coq, **Function** [BFPR06], disponible expérimentalement depuis la version 8.1. Cette construction permet de déclarer directement une fonction récursive générale (sans appels imbriqués) en fournissant un ordre bien fondé pour lequel l'argument de chaque appel récursif décroît strictement (pour un exemple voir la table 1.2).

Sujet. Le travail demandé consiste à étendre la compilation d'un fichier source Focal vers Coq, en profitant de l'apparition de la construction **Function**.

[1] La difficulté vient de la nécessité, à la fois, de (1) définir la fonction récursive ; (2) prouver sa terminaison en utilisant un ordre bien fondé. Plusieurs solutions existent pour accomplir ces deux tâches en définissant des types et en prouvant des lemmes intermédiaires. Certaines de ces solutions sont détaillées dans [BFPR06].

Tab. 1.1 – Exemple d'une fonction récursive par induction structurelle.
Le type des entiers naturels se définit inductivement de la façon suivante (en OCaml) :

```
type nat =
  Zero
| Succ of nat
```

Ceci veut dire que le type `nat` est muni d'un *élément de base* `Zero` et d'un *constructeur* `Succ` qui engendre un élément de type `nat` à partir d'un autre élément de type `nat`.
La fonction factorielle est définie alors comme une fonction récursive par induction structurelle (on suppose que la multiplication, `mult_nat`: `nat -> nat -> nat` est déjà définie) :

```
let rec fact = function
| Zero -> Succ(Zero)
| Succ n -> mult_nat (Succ n) (fact n)
```

Chaque appel récursif se fait avec un argument (ici `n`) qui est un sous terme strict de l'argument initial (ici `Succ n`).

TAB. 1.2 – Exemple d'utilisation de **Function**
La fonction `pow2` prend en argument un entier n et calcule 2^n. Il s'agit de la définition fondée sur l'algorithme dichotomique :

$$\begin{cases} 2^0 & = & 1 & \\ 2^{2n} & = & \texttt{square}(2^n) & \text{où } \texttt{square} : x \mapsto x^2 \\ 2^{2n+1} & = & \texttt{double}(2^n) & \text{où } \texttt{double} : x \mapsto 2x \end{cases}$$

En Coq cela donne :

```
Function pow2 (n: nat) {wf lt n}: nat:=
match n with
| 0 ⇒ 1
| S q ⇒ match (even_odd_dec (S q)) with
   | left _ ⇒ square (pow2(div2(S q)))
   | right _ ⇒ double (pow2 q)
  end
end.
```

où `even_odd_dec` permet de *décider* de la parité de son argument : **left** `_` signale que l'argument est pair et **right** `_` signale le contraire.
Cette définition génère les obligations de preuves données ci-dessous. Les deux premières concernent la décroissance des arguments des appels récursifs et la dernière correspond à la bonne fondation de l'ordre `lt`.

– ∀ `n q`: `nat`, `n` = `S q` →
  ∀ `e`: `even`(`S q`), `even_odd_dec`(`S q`) = **left** (`odd` (`S q`)) `e` →
  `div2` (`S q`) < `S q`
– ∀ `q`: `nat`, `n`=`S q` →
  ∀ `o`: `odd`(`S q`), `even_odd_dec`(`S q`) = **right** (`even`(`S q`)) `o` →
  `q` < `S q`
– `well_founded lt`

où `x` < `y` n'est rien d'autre qu'une *notation* pour `lt x y`.
L'utilisateur doit démontrer ces trois formules pour que la définition soit considérée comme complète.

1.3 Plan du rapport

L'objectif de ce rapport est de fournir une nouvelle façon de compiler les fonctions récursives écrites sous Focal qui prend en compte la preuve de terminaison. Dans un premier temps, je décrirai Coq et l'atelier Focal de manière plus détaillée : je présente des principes généraux et des aspects les plus pertinents.

Dans un second temps, je m'appuierai sur un exemple de fonction récursif non structurel pour déterminer par étapes successives la forme que peut prendre la preuve Coq fournie par le compilateur de Focal. Je propose aussi une syntaxe pour déclarer une fonction récursive sous Focal et en fournir la preuve de terminaison. Le chapitre 4 présente la situation actuelle et souligne les points à retravailler. Ensuite, en prenant du recul, le chapitre 5 développe la théorie du modèle mathématique de Focal avec les preuves de terminaison. Enfin, le chapitre 6 abouti à la solution adoptée sous Coq et sous Focal.

Chapitre 2

Coq

2.1 Un assistant à la preuve interactif

Coq est l'assistant à la preuve qui est utilisé pour certifier tout code Focal. Cet outil, en se servant de la correspondance entre théorie des types et logique intuitioniste, permet de définir des structures inductives, type ou prédicat, et de montrer des théorèmes sur ces structures. Le langage est purement fonctionnel et sa bibliothèque standard définit une grande partie des objets mathématiques les plus utiles (calcul sur les entiers naturels ou relatifs, les listes, propriétés de corps ou d'anneaux).

Dès que l'utilisateur veut définir un lemme, propriété ou fonction qui nécessite une preuve, Coq génère des sous-buts à prouver. Lorsque tous les sous-buts ont été prouvés, le lemme ou la fonction est alors définie. Pour ce faire, il existe un langage de *tactiques* où chaque tactique correspond à une ou plusieurs règle(s) de la logique intuitionniste (par exemple, le *Modus ponens*). L'application d'une tactique peut modifier un sous-but, le prouver entièrement auquel cas il est enlevé de la liste, ou faire apparaître d'autres sous-buts.

2.2 Ordres bien fondés et preuves de terminaison

Coq définit dans sa bibliothèque standard la notion de bonne fondation d'un ordre. Pour ce faire, on se sert du prédicat d'*accessibilté* : Pour tout ensemble A et toute relation R sur A un élément x de A est R-accessible si et seulement si tout élément qui lui est comparable à gauche est R-accessible. Ainsi, R est bien fondé si et seulement si tout élément A est R-accesible.

En Coq cela donne :

```
Variable A : Type.
Variable R : A → A → Prop.

Inductive Acc (x: A) : Prop :=
  Acc_intro : (∀ y:A, R y x → Acc y) → Acc x.

Definition well_founded := ∀ a:A, Acc a.
```

Lorsqu'on souhaite définir une fonction récursive non structurelle en Coq il faut fournir, entre autres, un ordre et montrer qu'il vérifie la propriété `well_founded`.

Chapitre 3

Focal

Un fichier source Focal est écrit en un langage fonctionnel orienté objets qui permet au développeur de spécifier des types de données accompagnés de propriétés qui doivent être vérifiés par toute implantation. Ensuite, il peut raffiner progressivement l'implantation tout en vérifiant les propriétés de départ. Ainsi, l'implantation finale sera certifiée. En effet, tout fichier source Focal est compilé à la fois vers OCaml, pour premettre la génération d'un programme exécutable, et vers Coq, pour vérifier que toutes les preuves nécessaires ont été faites et que ces preuves sont valides. Les compilations de Focal vers Coq et de Focal vers OCaml sont certifiées ce qui assure la correspondance entre le code exécutable et la preuve associée.[Pre03]

3.1 Espèces, collections et méthodes

Espèces et méthodes. Les aspects orientés objets rentrent en jeu dans Focal dès que le développeur organise son code sous forme d'*espèces* dont certaines héritent d'autres (l'héritage multiple est autorisé) mais toutes contiennent un type appelé *type support* et des *méthodes* qui sont des fonctions manipulant des éléments de ce type ou des propriétés. Le type support peut être déclaré ou défini comme étant un des types de base (**int**, **string**, **bool** ou **unit**), un type inductif (*par exemple* `list(a)`) ou encore un type construit à partir de deux autres et le constructeur des types fonctions (`a→b`) ou produits cartésiens (`a * b`). Il peut aussi être abstrait ; le type sera à définir par une ou plusieurs espèce(s) qui hérite(nt) de celle-ci.

Il en va de même pour les méthodes de l'espèce : on peut seulement les déclarer en donnant leur type, ou les définir avec leur code. En ce qui concerne les méthodes qui expriment des *propriétés*, leur implantation est une preuve de la propriété en question.

Exemple (Setoïde). Un setoïde est un ensemble non vide muni d'une relation d'équivalence appelée égalité. Dans l'espèce correspondante on ne fait que déclarer le type de repésentation (avec la mention **rep**) et les méthodes nécessiares (dont les propiétés assurant que l'égalité est bien une relation d'équivalence).

```
species setoide =
  rep;
  sig egal in self → self → bool;
  sig element in self;

  property refl: all x in self, !egal(x,x);
  property symm: all x y in self,
    !egal(x,y) → !egal(y,x);
  property trans: all x y z in self,
    !egal(x,y) → !egal(y,z) → !egal(x,z);
end
```

où l'appel à la méthode m est notée $!m$.

Héritage. Comme il a été mentionné, une espèce peut hériter d'un certain nombre d'autres espèces. Dans ce cas, l'espèce en question aura accès à toutes les méthodes des espèces parentes en plus de celles introduites par l'espèce elle-même. On peut aussi redéfinir l'implantation d'une méthode héritée. L'héritage de Focal suit la politique de la *liaison retardée* et impose que le type d'une méthode ne change pas dans une sous-espèce.

Exemple (Monoïde hérite de Setoïde). Un *monoïde* est un setoïde muni d'une loi de composition interne souvent appelée *multiplication* et d'un élément neutre pour cette loi.

```
species monoide inherits setoide =
  sig multiplie in self → self → self;
  sig un in self;

  let element = !multiplie(!un,!un);
end
```

À ce stade il est possible de définir `element` et ainsi assurer que tout monoïde contient au moins un élément.

Dépendances. Une méthode dépend d'une autre lorsque la première se sert de l'existence de l'autre dans sa définition. Ces dépendances sont importantes tout d'abord parce que Focal interdit à une méthode, qui n'est pas une

fonction récursive, de dépendre d'elle-même. Si une preuve se sert seulement de la déclaration d'une certaine méthode de l'espèce on dit que la propriété *decl-dépend* de la méthode. Si, par contre, elle nécessite la connaissance de l'implantation exacte de la méthode on dit qu'il y a une *def-dépendance*. Il faut alors constater que toute espèce doit redémontrer les propriétés qui sont héritées et qui def-dépendent d'une méthode dont l'implantation est redéfinie.

Dans l'espèce repésentant un monoïde, la définition de la méthode `element` introduit une dépendance avec `multiplie` et `un`. Vu qu'aucune des propriétés contenues dans cette espèce n'est prouvée il n'y a pas de def-dépendances.

Collections. Lorsque toutes les méthodes d'une espèce sont implantées et toutes les propriétés prouvées, le développeur peut créer à partir de cette espèce une *collection* qui est en quelque sorte une instanciation de l'espèce. Cette collection représente une structure mathématique bien définie où on peut mener des calculs effectifs. L'utilisateur d'une collection ne voit que son *interface*, l'ensemble des méthodes et leurs types, mais n'a pas accès ni au type de représentation ni aux implantations des méthodes. Ceci assure la sûreté du programme et évite une rupture des propriétés prouvées.

Exemple (La collection des entiers). L'ensemble des entiers (**int**) et des opérations dessus étant prédéfinies dans la bibliothèque standard de Focal, on peut créer une collection qui les représente comme un monoïde.

```
collection entiers implements monoide =
  rep = int;
  un = 1;
  multiplie = #int_mult;
  egal = #int_eq;
  proof of refl: assumed;
  proof of symm: assumed;
  proof of trans: assumed;
end
```

où l'appel à une fonction globale est précédée d'un dièse (#).

Paramétrisation. Focal autorise le développeur à définir des espèces par rapport à un ou plusieurs *paramètres* qui peuvent être soit de type (basique, inductif ou construit) soit d'espèce. Le corps de l'espèce peut alors utiliser ce paramètre dans la définition de ses méthodes et, si c'est un paramètre d'espèce, se servir des méthodes de l'espèce. En revanche, on ne peut accéder à l'implantation d'une de ces méthodes. En effet, lorsqu'on crée une collection qui implémente cette espèce, il faut fournir pour chaque paramètre

d'espèce une collection qui l'implémente. Si on voulait faire des preuves qui def-dépendent de l'implantation du paramètre, il faudrait imposer que cette implantation ne change pas au fil de l'héritage.

Exemple (Produit cartésien de deux setoïdes). On peut montrer en Focal que le produit cartésien de deux setoïdes est aussi un setoïde en définissant une espèce représentant ce produit qui hérite de setoïde et définit les méthodes nécessiares. Cette espèce est paramétrée par chacun des setoïdes qui forment le produit et le corps de l'espèce pourra accéder à la méthode m du paramètre p grâce à la notation $p!m$ et on peut abréger $p!\mathbf{rep}$ en p.

```
species setoide_produit(a is setoide, b is setoide) inherits seto
  rep = a * b;

  let egal(x,y) = #and(
    a!egal(#first(x),#first(y)),
    b!egal(#scnd(x),#scnd(y)));

  let element = #crp(a!element, b!element);

  [...]
end
```

où `crp` est le constructeur d'un couple, et `first` et `scnd` sont les deux projections d'un couple.

3.2 Zenon

Focal incorpore un outil de preuve non-interactif qui automatise des preuves simples, Zenon. En fournissant les lemmes nécessaires, Zenon est capable de trouver la bonne combinaison de ces lemmes pour prouver le théorème initial.

La totalité d'une preuve étant difficile à automatiser, Zenon attend des morceaux de preuves à faire ; c'est à l'utilisateur de faire le découpage. Ainsi une preuve doit être est mise sous la forme d'une succession de lemmes intermédiaires terminée par la preuve du théorème à partir de ces lemmes. Même si cela nécessite un travail préliminaire, une preuve mise sous cette forme est plus proche de la manière de raisonner en mathématiques. En effet, on présente souvent une preuve comme une succession d'étapes dont chacune est justifiée par un ou plusieurs lemmes.

Pour rendre cet outil plus accessible au développeur Focal il suffit de déterminer une syntaxe pour formuler les étapes de la preuve.

Exemple (Exemple de preuve Zenon dans Focal). Le théorème que je souhaite montrer est celui de l'unicité de l'élément neutre de l'addition dans un monoïde additif.

Chaque étape est notée par un nombre entre chevrons (¡,¿) suivi d'un identificateur (ici, je ne me sers que de chiffres). Le nombre indique la *profondeur* de l'étape et l'identificateur le différentie des autres étapes de même profondeur. Une étape comporte d'éventuelles hypothèses (par la clause **assume**) et un énoncé qui peut être prouver soit directement (la clause **by**) soit par les étapes à la profondeur supérieure. Lorsque cette dernière solution est utilisée, une des étapes doit contenir l'énoncé **qed** qui indique qu'on peut conclure grâce à l'étape précédente.

```
theorem zero_is_unique: all o in self,
  (all x in self,!equal(x,!plus(x,o))) → !equal(o,!zero)
  proof:
    <1>1 assume o in self
      H1: all x in self, !equal(x,!plus(x,o))
      prove !equal(o,!zero)
      <2>1 prove !equal(!zero,!plus(!zero,o))
        by <1>:H1
      <2>2 prove !equal(o,!zero)
        by <2>1, !zero_is_neutral,
          !equal_transitive,
          !equal_symmetric
      <2>3 qed
    <1>2 qed
;
```

Ainsi, l'étape 1.1 propose de montrer le résultat de l'implication dans l'énoncé général en faisant l'hypothèse des prémises de cette implication ; l'étape 1.2 conclus grâce à la règle d'introduction de l'implication de la logique classique. L'étape 1.1 prouve son but en deux étapes : en 2.1, on montre que $0 = 0 + o$ grâce à l'hypothèse $H1$ introduite à la profondeur 1, puis en 2.2 on montre que $o = 0$ grâce à la neutralité de 0 pour l'addition, et la transitivité et la symmétrie de l'égalité.

Chapitre 4

Existant

Ce chapitre illustre la compilation vers Coq telle qu'elle est faite actuellement et met en évidence les modifications à effectuer dans cette preuve pour la rendre convenable.

4.1 Présentation de l'exemple utilisé

Je me suis appuyé sur l'exemple du tri rapide des listes d'entiers par ordre croissant. L'algorithme s'écrit en OCaml de la façon suivante :

partition. Partitionne une liste en deux en comparant chaque élément à un pivot donné.

paramètres

pivot un entier

l une liste à partitionner

valeur de retour un couple de listes $(less, more)$ où $less$ contient les éléments plus petits que le pivot et $more$ les éléments strictement plus grands

```
let rec partition pivot l =
match l with
| [] → ([],[])
| h::t → let (less,more) = partition pivot t in
    if h <= pivot
    then (h::less,more)
    else (less,h::more)
```

Puisque l'appel récursif de partition se fait sur un sous terme strict de l'argument `l` (en l'occurence `t`), la fonction `partition` est récursive structurelle.

quicksort. Trie une liste d'entiers par ordre croissant.

paramètre l la liste à trier

valeur de retour une permutation de l dont les éléments sont rangés par ordre croissant

```
let rec quicksort l =
match l with
| [] → []
| h::t → let (less,more) = partition h t in
    List.append (quicksort less) (h::(quicksort more))
```

Contrairement à `partition`, `quicksort` n'est pas appliqué récursivement sur un sous-terme strict de l'argument. Par contre on peut montrer que la fonction termine grâce à l'ordre défini par :

$$l \prec l' \Leftrightarrow \texttt{length}\ l < \texttt{length}\ l'$$

En effet, pour tous h : `int`, t : `int list`, on a

$$(less, more) = \texttt{partition}\ h\ t \quad \Rightarrow \quad less \prec h :: t \ \wedge \ more \prec h :: t$$

4.2 Implantation en Focal

En Focal, j'ai écrit une espèce `sorting` contenant ces deux fonctions. Cette espèce se sert de la définition des listes suivante.

```
type list(a) =
  Nil in list(a);
  Cons in a → list(a) → list(a);
;;
```

Espèce sorting. Je signale en tête du fichier que je me sers des définitions contenues dans :

- **basics.foc** où apparaît la définition des listes, et l'espèce de base de la hiérarchie, `basic_object` ;
- **lists_foc.foc** qui contient les fonctions de base sur les listes.

De plus, pour des raisons pratiques, je précise aussi que j'emploie les définitions des deux fichiers comme si elles étaient locales. Cela évite d'avoir à précéder chaque appel à une fonction par le nom du fichier d'où elle vient.

```
uses basics;;   open basics;;
uses lists_foc;; open lists_foc;;
```

L'implantation de chacune des deux fonctions est incluse dans une espèce qui hérite de `basic_object`. Notons que l'appel à une fonction ou constructeur global est précédé d'un dièse (#) et que l'appel à un membre de l'espèce courante est précédé d'un point d'exclamation (!).

```
species comparable inherits basic_object =
  sig leq in self → self → bool;
end

species sorting(a is comparable) inherits basic_object =
  rep = list(a);

  let rec partition(pivot in a, l in self) in (self*self) =
    match l with
      #Nil → (#Nil,#Nil)
    | #Cons(h,t) → let (less,more) = !partition(pivot, t) in
        if a!leq(h, pivot)
        then (#Cons(h,less),more)
        else (less,#Cons(h,more))
    end;

  let rec quicksort(l in self) in self =
    match l with
      #Nil → #Nil
    | #Cons(h,t) → let (less,more) = !partition(h, t) in
        #append(!quicksort(less),#Cons(h,!quicksort(more)))
    end;
end
```

Actuellement, le compilateur Focal n'oblige pas le développeur à fournir une preuve de terminaison pour définir des fonctions récursives ; la syntaxe actuelle de Focal ne donne pas la possibilité d'inclure cette preuve.

4.3 Compilation vers Coq

Lorsqu'on compile cette espèce, on obtient, entre autres, une preuve Coq. Cette preuve doit tenir compte de deux aspects du langage Focal importants : *les abstractions* correspondant aux méthodes seulement déclarées, *l'héritage* avec *liaison retardée*.

Pour le premier aspect, il suffit d'inclure toutes les définitions correspondant à une espèce dans une **Section** ou **Chapter**. Ces deux constructions

permettent à la fois de séparer les espèces s'il y en a plusieurs dans un même fichier et de déclarer avec **Variable** ou **Hypothesis** des valeurs abstraites. Coq ajoute, en fin de **Section**, chaque **Variable** et chaque **Hypothesis** en argument aux définitions de la **Section**.

En ce qui concerne l'héritage et la liaison retardée, ces mécanismes doivent être produits par des *générateurs de méthodes*. Un générateur de méthode, en plus des arguments de la méthode qu'il génère, attend les implantations des fonctions dont la méthode decl- ou def-dépend. Une espèce qui ne redéfinit pas une méthode héritée, utilise le même générateur de méthode. Ainsi, on peut préciser quelle implantation est utilisée par une méthode qui en appelle une autre.

Le fichier produit commence par une liste de modules à importer. Ces modules contiennent les définitions de base de Focal et de Zenon, et les modules Coq correspondant à la compilation de `basics` et de `lists_foc`.

```
Require Import zenon.
Require Import zenon_coqbool.
Require Export generic_proof_cases.
Require Export Foc_init.
Open Scope Z_scope.
Require Import basics.
Require Export lists_foc.
```

Tout ce qui concerne l'espèce `sorting` est contenu dans un **Chapter** qui porte le même nom.

```
Chapter Sorting.
```

Le **Chapter** se découpe en trois parties :
- définition du type enregistrement de l'espèce ;
- déclaration des méthodes abstraites, définition des générateurs de méthode et liaisons locales ;
- définition d'un constructeur de l'espèce et des fonctions de changement de type vers les espèces parents.

Regardons maintenant ce que le compilateur Focal met dans chacune de ces trois parties, en commençant par le type enregistrement.

4.3.1 Type enregistrement

Ce type, qui contient un champ pour chaque membre de l'espèce, correspond à l'interface de l'espèce.

```
Record sorting ( a_T: Set ): Type:=
  mk_sorting {
```

```
  sorting_T:> Set;

  sorting_print: (sorting_T) → string__t;

  sorting_parse: (string__t) → sorting_T;

  sorting_partition:
    (a_T) → (self_T) → (prod sorting_T sorting_T);

  sorting_quicksort:
    (sorting_T) → sorting_T
}.
```

On remarque que ce type contient non seulement les membres déclarés dans `sorting` mais aussi ceux qui sont hérités de `basic_object`. En effet, ce type enregistrement est indépendant de celui de `basic_object`, et ce n'est que par une fonction de changement de type déclarée à la fin que le lien sera fait.

4.3.2 Déclaration des méthodes

Ensuite, le type support est déclaré, suivi de la définition locale des deux méthodes héritées de `basic_object`, définition qui fait la liaison entre un appel sur **self** à une des deux méthodes et leurs implantations respectives dans `basic_object`.

```
Variable a_T: Set.
Variable a_leq: a_T → a_T → bool__t.

Let self_T: Set := list__t(a_T).
Let self_print: (self_T) → string__t:=
(basic_object__print self_T ).
Let self_parse: (string__t) → self_T:=
(basic_object__parse self_T ).
```

Il est alors possible de définir la fonction `partition`. Pour faire la différence avec d'éventuelles méthodes du même nom dans d'autres espèces (en particulier dans les sous-espèces de `sorting`), la définition dans le script Coq est précédé du nom de l'espèce suivi de deux caractères de tiret bas (`_`).

```
Fixpoint sorting__partition (pivot: a_T)
  (l: self_T) {struct l}:
(prod self_T self_T):=
```

```
let self_partition:= (sorting__partition) in
match l with
| Nil ⇒ (Nil, Nil)
| (Cons h t) ⇒
    let less:= (__g_first (self_partition pivot t)) in
    let more:= (__g_scnd (self_partition pivot t)) in
    if (a_leq h pivot)
    then (Cons h less, more)
    else (less, Cons h more)
  end.
```

On remarque que trois appels à des fonctions dont les noms commencent par `__g_` apparaissent dans le corps de cette fonction. Ce préfixe est utilisé pour les fonctions déclarées au *toplevel* (en dehors d'une espèce). `__g_first` (resp. `__g_scnd`) renvoie le premier (resp. le deuxième) élément d'un couple et `__g_int_leq` renvoie le booléen résultant de la comparaison $\leq$ sur les entiers relatifs.

Pour en finir avec la fonction `partition`, le compilateur fait une définition locale qui lie un appel sur `self` de cette fonction à l'implantation qu'on vient d'en donner.

```
Let self_partition:
  (a_T) →
  (self_T) →
  (prod self_T self_T):=
  (sorting__partition).
```

Définition de `quicksort` Le compilateur Focal détecte déjà que la définition de `quicksort` est plus délicate et la place alors dans une **Section** à part.

```
Section Quicksort.
```

Le compilateur commence par déclarer une fonction correspondant à l'abstraction de la définition de `sorting__partition` ce qui assure qu'il n'y a qu'une def-dépendance entre ces deux fonctions.

```
Variable param__a_T: Set.
Let abst_T: Set := list(param__a_T).

Variable abst_partition:
  (param__a_T) →
  (abst_T) →
  (prod abst_T abst_T).
```

Puis il déclare un ordre sur l'argument de `quicksort`, montre qu'il est bien fondé et prouve la décroissance des arguments des appels récursifs.

```
Let my_order:
  (abst_T) → (abst_T) → Prop:=
  (fun(x: abst_T) ⇒
  (fun(y: abst_T) ⇒
  (magic_order x y))).
Let well_founded_my_order:(well_founded my_order).
  eapply dontwanttoproveit.
Qed.
Let lemma_6:
  ∀ _aux1:(abst_T),
  let l:=_aux1 in
  ∀ h:param__a_T, ∀ t:(abst_T),
    (l=h::t) →
  let less:=(__g_first (abst_partition h t)) in
  let more:=(__g_scnd (abst_partition h t)) in
    unit → (my_order more _aux1).
  eapply dontwanttoproveit.
  Qed.
```

`lemma_6` est suivi de cinq autres lemmes qui sont similaires et qui sont prouvés avec la même tactique.

Grâce à ces lemmes, et à la construction **Fix**, on définit `sorting__quicksort` puis, après avoir fermé la **Section**, on relie l'appel à `quicksort` sur `self` à son implantation, en précisant l'implantation de la fonction qui avait été déclarée comme abstraite.

```
Let quicksort_aux: ∀ _aux1:(abst_T),
(∀ _y:(abst_T),(my_order _y _aux1) →
(abst_T)) → (abst_T):=
fun _aux1:(abst_T) ⇒
fun quicksort_aux_rec:
  (∀ _y:(abst_T),
  (my_order _y _aux1) →
  (abst_T)) ⇒
let l:= _aux1 in
match l with
| Nil ⇒ Nil
```

[1] `magic_order` est un ordre sur tout ensemble qui est déclaré comme axiome.

[2] Ici on applique un lemme qui est déclaré comme axiome et qui a pour type ∀ `P`, `P`.

```
    | (Cons h t) ⇒
      let less:= (__g_first (abst_partition h t)) in
      let more:= (__g_scnd (abst_partition h t)) in
      (__g_append
        (quicksort_aux_rec less
          (lemma_4 _aux1 h t (dontwanttoproveit
            (eq l (Cons h t))) tt))
        (Cons h (quicksort_aux_rec more
          (lemma_6 _aux1 h t (dontwanttoproveit
            (eq l (Cons h t))) tt))))
    end.
  Let sorting_quicksort_aux:=
    (Fix well_founded_my_order
      (fun(x:(abst_T)) ⇒
      (abst_T)) quicksort_aux).
  Definition sorting__quicksort:=
    fun l ⇒ (sorting_quicksort_aux l).
End Quicksort.
Let self_quicksort:
  (abst_T) → (abst_T):=
  (sorting__quicksort a_T self_partition).
```

4.3.3 Constructeur et coercions

Enfin, il ne manque plus qu'à relier toutes les déclarations locales à l'espèce en définissant un constructeur `new_sorting` et définir une fonction de changement de type qui permet de voir toute instance de `sorting` comme une instance de `basic_object`. Le premier est défini comme la création d'un élément du type enregistrement en fournissant l'implantation qui convient de chaque méthode. Le deuxième est défini de façon similaire, mais cette fois-ci on utilise le constructeur du type enregistrement de `basic_object` appliqué aux implantations des méthodes héritées.

```
Definition new_sorting:=
  (mk_sorting
   a_T self_T self_print self_parse self_partition self_quicksort
End Sorting.
Definition sorting_basic_object:=
  fun a_T: Set ⇒ fun(S: (sorting a_T)) ⇒
  (mk_basic_object
   (sorting_T _ S) (sorting_print _ S) (sorting_parse _ S)).
```

On remarque que `sorting_basic_object` respecte la liaison tardive. En effet, même si la fonction crée une instance de `basic_object`, cette instance n'utilisera pas les implantations de `basic_object` mais celles de `sorting`.

4.4 Bilan

La définition de `sorting__quicksort` a nécessité l'utilisation des deux éléments qui rendent la preuve incorrecte :

- `magic_order` est simplement déclaré comme un ordre sur tout type dont il n'existe pas d'implantation.
- `dontwanttoproveit` est un axiome de type ∀ `P`:**Prop**, `P`. Se servir de cet axiome est certes pratique mais cela invalide la preuve. En fait, il est surtout utilisé pour signaler une partie de preuve difficile sur laquelle il faudra revenir.

De plus, la définition de `sorting__quicksort` faite actuellement par le compilateur Focal contient du code (majoritairement parmi les lemmes) inutile.

Chapitre 5

Analyse statique

Le but de ce chapitre est de reprendre l'analyse faite dans [Pre03] et de l'adapter pour accomoder les preuves de terminaison. Dans [Pre03], on présente des contraintes sur un programme Focal, un modèle mathématique représentant un programme qui respecte ces contraintes et un algorithme de transformation d'un programme vers le modèle. Ici, je décris le résultat des modifications que j'apporte à ce travail préallable, lesquelles concernent entièrement la gestion des fonctions récursives.

5.1 Espèce

Dans ce chapitre, je m'intéresse aux différents champs introduits dans une espèce, laquelle pouvant hériter d'autres espèce. À partir de ces informations on cherche à transformer, suivant un certain algorithme, une telle espèce sous une forme équivalente[1] où l'héritage n'apparaît plus. Cette nouvelle forme sera appelée *forme normale* et elle permettra de produire du code correspondant à l'espèce dans un autre langage de programmation ou de preuve.

Définition 5.1.1 (Champ d'une espèce)**.**
Un *champ* d'une espèce peut être construit d'une des manières suivantes :

[1] *équivalente* : deux espèces sont équivalentes si elles ont la même interface (l'ensemble des noms de méthodes et leurs types) et si elles réagissent de la même manière aux appels de méthode

	déclaration	définition
type support	**rep**	**rep** $:= \tau$
calcul	**sig** $x\colon \tau$	**let** $x\colon \tau := e$ **let rec** $x\colon \tau := e$ [**proof**: p]
logique	**property** $x\colon s$	**theorem** $x\colon s$ **proof**: p

où x est un nom de champ, τ un type, e une expression, s une proposition, p une preuve, et les crochets ([ ]) signalent une partie optionnelle. Dans le cas d'une fonction récursive, introduite par la construction **let rec** $x\colon \tau := e$, j'ajoute la possibilité de fournir une preuve de terminaison.

Définition 5.1.2 (Espèce)**.**
On appelle *espèce* le couple d'une liste d'espèces qui constitue les espèces parents, et d'une liste de champs.

Exemple. Soient Φ une liste de champs, et ψ un champ.
- $s_1 = ([\,], \Phi)$ est l'espèce qui contient les champs de Φ
- $s_2 = ([\,], [\psi])$ est l'espèce qui contient le seul champ ψ
- $s_3 = ([s_1; s_2], [\,])$ est l'espèce qui hérite de s_1 et de s_2, qui n'introduit aucun nouveau champ, et ne redéfinit pas les champs hérités.

Définition 5.1.3 (Noms introduits par un champ)**.**
À tout champ ϕ, on associe l'ensemble des noms déclarés par ϕ (noté $\mathcal{N}(\phi)$) et l'ensemble des noms définis par ϕ (noté $\mathcal{D}(\phi)$). On a :

$$\begin{aligned} \mathcal{N}(\textbf{let } x\colon \tau := e) = \mathcal{N}(\textbf{sig } x\colon \tau) &= \{x\} \\ \mathcal{N}(\textbf{let rec } x\colon \tau := e \ [\textbf{proof}\colon p]) &= \{x\} \\ \mathcal{N}(\textbf{rep } [:= \tau]) &= \{\text{rep}\} \end{aligned}$$

$$\begin{aligned} \mathcal{D}(\phi) &= \emptyset \quad \text{si } \phi \in \{\textbf{sig } x\colon \tau; \textbf{rep}\} \\ \mathcal{D}(\phi) &= \mathcal{N}(\phi) \quad \text{sinon} \end{aligned}$$

Définition 5.1.4 (Espèce bien spécifiée)**.**
Soit $s = ([s_i]_{1\leq i\leq m}, [\phi_j]_{1\leq j\leq n})$ une espèce.

s est dite *bien spécifiée* si et seulement si, pour tout $i \in [\![1, m]\!]$, s_i est bien spécifiée et :

$$\forall i, j \in \mathbb{N}, \quad 0 \leq i < j \leq n \quad \Rightarrow \quad \mathcal{N}(\phi_i) \cap \mathcal{N}(\phi_j) = \emptyset$$

On peut étendre les définitions de $\mathcal{N}$ et $\mathcal{D}$ aux espèces bien spécifiées : Il s'agit de l'ensemble des noms déclarés ou définis dans une espèce (y compris les noms hérités d'espèces parents).

Définition 5.1.5 (Noms des méthodes d'une espèce)**.**
Soit $s = ([s_i]_{1\leq i\leq m}, [\phi_j]_{1\leq j\leq n})$ une espèce bien spécifiée. On définit,

$$\begin{aligned}\mathcal{N}(s) &= \left(\bigcup_{1\leq i\leq m} \mathcal{N}(s_i)\right) \cup \left(\bigcup_{1\leq j\leq n} \mathcal{N}(\phi_j)\right)\\ \mathcal{D}(s) &= \left(\bigcup_{1\leq i\leq m} \mathcal{D}(s_i)\right) \cup \left(\bigcup_{1\leq j\leq n} \mathcal{D}(\phi_j)\right)\end{aligned}$$

Définition 5.1.6 (Corps d'une méthode dans une espèce)**.**
Soient $s = ([s_i]_{1\leq i\leq m}, [\phi_j]_{1\leq j\leq n})$ une espèce bien spécifiée et $x \in \mathcal{N}(s)$. $\mathcal{B}_s(x)$ est le corps de x au vu de s, *i.e.* si x est défini dans un des champs de s c'est l'expression (ou le type s'il s'agit de **rep**) contenue dans cette définition, sinon on s'intéresse à la définition donnée dans la dernière espèce parente dans l'ordre de la liste (c.f. règle Inh). $\mathcal{B}_s(x)$ est défini inductivement par les règle suivantes :

$$\textsc{Decl}\ \frac{x \notin \mathcal{D}(s)}{\mathcal{B}_s(x) = \bot} \qquad \textsc{LetDef}\ \frac{\exists j \in [\![1,n]\!]\,,\ \phi_j \in \{\textbf{let}\ x\colon \tau := e; \textbf{let}\ \textbf{rec}\ x\colon \tau := e\ [\textbf{proof}\colon p]\}}{\mathcal{B}_s(x) = e}$$

$$\textsc{RepDef}\ \frac{\exists j \in [\![1,n]\!]\,,\ \phi_j = \textbf{rep} := \tau}{\mathcal{B}_s(rep) = \tau} \qquad \textsc{ThmProof}\ \frac{\exists j \in [\![1,n]\!]\,,\ \phi_j = \textbf{theorem}\ x\colon s\ \textbf{proof}\colon p}{\mathcal{B}_s(x) = \bot}$$

$$\textsc{Inh}\ \frac{x \notin \bigcup_{1\leq j\leq n} \mathcal{D}(\phi_j) \qquad i_0 = \max\{i \in [\![1,m]\!] \mid x \in D(s_i)\} \qquad \mathcal{B}_{s_{i_0}}(x) = e}{\mathcal{B}_s(x) = e}$$

Définition 5.1.7 (Preuve d'une méthode dans une espèce)**.**
Soient $s = ([s_i]_{1\leq i\leq m}, [\phi_j]_{1\leq j\leq n})$ une espèce bien spécifiée et $x \in \mathcal{N}(s)$. À l'instar de $\mathcal{B}_s$, on définit inductivement $\mathcal{P}_s(x)$, la preuve associée à x au vu de s, par les règles suivantes :

$$\textsc{Decl}\ \frac{x \notin \mathcal{D}(s)}{\mathcal{P}_s(x) = \bot} \qquad \textsc{Def}\ \frac{\exists j \in [\![1,n]\!]\,,\ \phi_j \in \{\textbf{rep}\ [:= \tau]; \textbf{let}\ x\colon \tau := e; \textbf{let}\ \textbf{rec}\ x\colon \tau := e\}}{\mathcal{P}_s(rep) = \bot}$$

$$\textsc{Proof}\ \frac{\exists j \in [\![1,n]\!]\,,\ \phi_j \in \{\textbf{let}\ \textbf{rec}\ x\colon \tau := e\ \textbf{proof}\colon p; \textbf{theorem}\ x\colon s\ \textbf{proof}\colon p\}}{\mathcal{P}_s(x) = p}$$

$$\textsc{Inh}\quad \frac{x \notin \bigcup_{1\leq j\leq n} \mathcal{D}(\phi_j) \qquad i_0 = \max\{i \in [\![1,m]\!] \mid x \in D(s_i)\} \qquad \mathcal{P}_{s_{i_0}}(x) = p}{\mathcal{P}_s(x) = p}$$

5.2 Dépendances

Définition 5.2.1 (Decl-dépendances)**.**
On définit la fonction de *decl-dépendance* (notée $\wr \cdot \int^{\text{decl}}$) comme étant la fonction qui à une expression associe l'ensemble des noms de l'espèce courante qui apparaissent dans cette expression.[2]

De la même façon, on définit cette fonction sur les preuves de théorèmes ou de terminaison.

On étend ensuite cette définition à un champ.

$$\begin{aligned}
\wr \textbf{let } x\colon \tau := e \int^{\text{decl}} &= \wr e \int^{\text{decl}} \\
\wr \textbf{let rec } x\colon \tau := e\ [\textbf{proof}\colon p] \int^{\text{decl}} &= (\wr e \int^{\text{decl}} \setminus \{x\}) \cup \wr p \int^{\text{decl}} \\
\wr \textbf{sig } x\colon \tau \int^{\text{decl}} = \wr \textbf{rep } [:= \tau] \int^{\text{decl}} &= \emptyset \\
\wr \textbf{theorem } x\colon s\ \textbf{proof}\colon p \int^{\text{decl}} &= \wr s \int^{\text{decl}} \cup \wr\, p \int^{\text{decl}} \\
\wr \textbf{property } x\colon s \int^{\text{decl}} &= \wr s \int^{\text{decl}}
\end{aligned}$$

Enfin, soit s une espèce bien spécifiée. Il est possible d'étendre encore cette définition aux noms de s. Comme pour $\mathcal{B}_s(x)$ où $x \in \mathcal{N}(s)$, dans le cas d'un héritage multiple, $\wr x \int_s^{\text{decl}}$ s'intéresse au dernier champ où x apparaît.

Définition 5.2.2 (Def-dépendances)**.**
On définit la fonction de *def-dépendance* (notée $\wr \cdot \int^{\text{def}}$) comme étant la fonction qui à une preuve associe les noms de l'espèce courante dont la définition est utilisée.[3]

Cette définition s'étend aux champs et aux noms d'une espèce bien spécifiée de la même manière que $\wr \cdot \int^{\text{decl}}$.

Remarque. Même dans le cas d'une espèce paramétrée, il est impossible d'introduire une def-dépendance par rapport à d'autres noms car leurs définitions ne sont pas visibles.

[2] Dans [Pre03] on présente les règles qui définissent $\wr \cdot \int^{\text{decl}}$ sur l'ensemble des expressions mais, vu que je ne m'intéresse pas particulièrement aux expressions, je préfère donner une définition textuelle.

[3] Ces noms sont signalés dans la preuve par le mot clé **def**.

Définition 5.2.3 (Relations de dépendance)**.**
Soient s une espèce bien spécifiée et $x_1, x_2 \in \mathcal{N}(s)$. On définit la relation de decl-dépendance (notée $x_1 \blacktriangleleft_s^{\text{decl}} x_2$) de la manière suivante :

On a $x_1 \blacktriangleleft_s^{\text{decl}} x_2$ si et seulement si il existe une suite de noms $(y_i)_{1 \leq i \leq n}$ telle que $y_1 = x_1$, $y_n = x_2$ et

$$\forall i \in [\![1, n-1]\!] \,,\, y_{i+1} \in \wr y_i \int_s^{\text{decl}}$$

Il en est de même pour la relation de def-dépendance (notée $x_1 \blacktriangleleft_s^{\text{def}} x_2$).

Remarque. Pour toute espèce s bien spécifée, on a $\blacktriangleleft_s^{\text{def}} \subset \blacktriangleleft_s^{\text{decl}}$.

Définition 5.2.4 (Bonne formation d'une espèce)**.**
Une espèce s bien spécifiée est dite *bien formée* si et seulement si :

$$\forall x \in \mathcal{N}(s),\, \neg(x \blacktriangleleft_s^{\text{decl}} x)$$

5.3 Espèces paramétrées

Comme il a été annoncé, une espèce peut déclarer des paramètres et les utiliser dans les champs de l'espèce. Dans le cas où ces paramètres sont d'espèce, cela rajoute des noms dont on peut créer des dépendances. Or les fonctions de def- et de decl-dépendance ne prennent compte que des noms de l'espèce courante et il n'est pas nécessaire d'y ajouter les paramètres. En effet, considérer les appels à des méthodes d'un paramètre d'espèce comme une decl-dépendance n'introduira pas de cycle de dépendance et on doit interdire les def-dépendances avec ces méthodes pour éviter la rupture d'une propriété par des redéfinitions dans des sous-espèces de l'espèce paramètre.

De plus, l'instanciation d'un paramètre est défini comme une substitution, donc on ne perd pas de généralité à omettre cet aspect de l'étude.

5.4 Typage des espèces

Le typage dans les expressions suit les mêmes contraintes que pour les langages purement fonctinnelles. L'héritage d'une espèce ajoute deux contraintes :

– on ne peut pas redéfinir le type de représentation ;
– les membres redéfinis ne doivent pas changer de type.

Enfin, pour éviter des problèmes comme celui illustré dans la table 5.1, on interdit le polymorphisme dans le corps d'une espèce.

Toutes ces règles sont formellement présentée dans [Pre03] mais il suffit ici de donner la définition suivante.

TAB. 5.1 – Problème de typage avec le polymorphisme
Considérons les espèces suivantes (en rappelant que la notation `s!m` est une référence à la méthode `m` de l'espèce `s`) :

```
species poly = rep; let id = fun x → x; end

species id_bool (x is poly) =
  rep = unit;
  let elt = x!id(true);
end

species id_int inherits ploy =
  rep = unit;
  let id = fun x → x + 1;
end

collection coll_int implements id_int

collection error implements id_bool(coll_int)
```

Notons que `id_bool` et `id_int` font intervenir deux instances parfaitement correctes du type de poly !id. Néanmoins, en les réunissant dans la collection `error`, le corps de `error!elt` s'évalue à `true` + 1 qui devrait être rejeté.

Définition 5.4.1 (Espèce bien typée)**.**
Soit s une espèce bien spécifiée. s est bien typée si et seulement si :
- l'ensemble des expressions calculatoires ou logiques sont bien typées ;
- le type de représentation est définie dans une espèce parente, celui-ci n'est pas changé par l'espèce courante ;
- une méthode est héritée d'une espèce parente, son type n'est pas changé par l'espèce courante ;
- aucune méthode n'a un type polymorphe.[4]

5.5 Mise sous forme normale

Je reprend d'abord la définition exacte d'une *forme normale* puis je définit deux opérations sur des champs : l'*effacement* et la *fusion*. La première servira à effacer les preuves héritées qui def-dépendent d'au moins une méthode redéfinie. La deuxième premettra de résoudre l'héritage en fusionnant le corps d'un champ au vu de l'espèce courante avec un champ au vu d'une espèce parente le résultat devant respecter la liaison tardive. Enfin, en me servant de ces opérations, je donne un algorithme qui transforme une espèce bien formée vers une espèce équivalente et en forme normale.

Définition 5.5.1 (Forme normale d'une espèce)**.**
Soient s une espèce, H la liste des espèces parentes de s et $[\phi_i]_{1\leq i\leq n}$ la liste de ces champs. L'espèce s est dite sous *forme normale* si et seulement si toutes les conditions suivantes sont vérifiées :
- H est la liste vide
- s est bien spécifiée
- s est bien typée
- Toute définition ne dépend que des noms introduits dans les champs précédents, *i.e.*

$$\forall i \in [\![1, n]\!], \quad \wr\phi_i\int^{\text{decl}} \subset \bigcup_{1\leq j<i} \mathcal{N}(\phi_j)$$

Propriété 5.5.2.
Soit s une espèce. Si s est en forme normale alors s est bien formée.

Définition 5.5.3 (Effacement d'une définition)**.**
L'effacement d'un champ est défini par cas :

[4] Notons qu'on peut écrire des méthodes en Focal dont le corps a un type polymorphe mais on doit restreindre le type de la méthode à une instance précise de ce type.

$$\begin{aligned}
\mathcal{E}(\textbf{let rec } x\colon \tau := e \textbf{ proof}\colon p) &= \textbf{let rec } x\colon \tau := e \\
\mathcal{E}(\textbf{theorem } x\colon s \textbf{ proof}\colon p) &= \textbf{property } x\colon s \\
\text{sinon, pour tout autre champ } \phi & \\
\mathcal{E}(\phi) &= \phi
\end{aligned}$$

Définition 5.5.4 (Effacement dans un contexte)**.**
Soit N une liste de noms. On définit l'effacement dans le contexte N (noté $\mathcal{E}_N$) de la manière suivante :

$$\begin{aligned}
\mathcal{E}_N([\,]) &= [\,] \\
\mathcal{E}_N(\phi.l) &= \mathcal{E}(\phi).\mathcal{E}_{N\cup\mathcal{N}(\phi)}(l) \quad \text{si } \wr\, \phi \,\int^{\text{def}} \cap N \neq \emptyset \\
\mathcal{E}_N(\phi.l) &= \phi.\mathcal{E}_N(l) \quad \text{sinon}
\end{aligned}$$

Théorème 5.5.5 (Préservation des noms)**.**
Soient s une espèce en forme normale, Φ la liste de ses champs et N un ensemble de noms. On a

$$\begin{aligned}
\mathcal{N}(\mathcal{E}_N(\Phi)) &= \mathcal{N}(\Phi) \\
\mathcal{D}(\mathcal{E}_N(\Phi)) &\subset \mathcal{D}(\Phi)
\end{aligned}$$

Démonstration. Immédiat par induction sur la longueur de Φ. □

Théorème 5.5.6 (Effacement et def-dépendance)**.**
Soient $\Phi = [\phi_i]_{1\leq i\leq n}$ une liste de champs et N un ensemble de noms vérifiant :

1. $\forall i \in [\![1, n]\!]\,, \forall j \in [\![i+1, n]\!]\,, \quad \mathcal{N}(\phi_i) \cap \mathcal{N}(\phi_j) = \emptyset$
2. $\forall i \in [\![1, n]\!]\,, \quad \wr\phi_i\int^{\text{def}} \subset N \cup \bigcup_{1\leq j<i} \mathcal{N}(\phi_j)$
3. $N \cap \bigcup_{1\leq i\leq n} \mathcal{N}(\phi_i) = \emptyset$

On a alors :

$$\forall x \in \bigcup_{1\leq i\leq n} \mathcal{D}(\phi_i), \quad N \cap \wr x\int^{\text{decl}} \neq \emptyset \Rightarrow x \notin \mathcal{D}(\mathcal{E}_N(\Phi))$$

Démonstration. Voir [Pre03]. □

Définition 5.5.7 (Fusion de deux champs)**.**
Soit $\mathcal{U} : (\tau_1, \tau_2) \mapsto \tau_3$ qui calcule τ_3, le type le plus général qui unifie τ_1 et τ_2. Soient ϕ_1 et ϕ_2 deux champs tels que $\mathcal{N}(\phi_1) \cap \mathcal{N}(\phi_2) \neq \emptyset$.

La fusion du champ ϕ_1 avec ϕ_2 (noté $\phi_1 \circledcirc \phi_2$) est défini par cas.

ϕ_1	ϕ_2	$\phi_1 \circledcirc \phi_2$
sig x: τ	**sig** x: τ	**sig** x: τ
sig x: τ_1	**let** x: $\tau_2 := e$	**let** x: $\tau_3 := e$
sig x: τ_1	**let rec** x: $\tau_2 := e$ [**proof**: p]	**let rec** x: $\tau_3 := e$ [**proof**: p]
property x: s	**property** x: s	**property** x: s
property x: s	**theorem** x: s **proof**: p	**theorem** x: s **proof**: p
let x: $\tau_1 := e$	**sig** x: τ_2	**let** x: $\tau_3 := e$
let rec x: $\tau_1 := e$ [**proof**: p]	**sig** x: τ_2	**let rec** x: $\tau_3 := e$ [**proof**: p]
theorem x: s **proof**: p	**property** x: s	**theorem** x: s **proof**: p
let x: $\tau_1 := e_1$	**let** x: $\tau_2 := e_2$	**let** x: $\tau_3 := e_2$
let rec x: $\tau_1 := e_1$ **proof**: p_1	**let rec** x: $\tau_2 := e_2$ **proof**: p_2	**let rec** x: $\tau_3 := e_2$ **proof**: p_2
theorem x: s **proof**: p	**theorem** x: s **proof**: p	**theorem** x: s **proof**:

Propriété 5.5.8 (Préservation des noms et définitions)**.**
Soient ϕ_1 et ϕ_2 deux champs tels que $\mathcal{N}(\phi_1) \cap \mathcal{N}(\phi_2) \neq \emptyset$. On a

$$\mathcal{N}(\phi_1 \circledcirc \phi_2) = \mathcal{N}(\phi_1) \cup \mathcal{N}(\phi_2)$$

et

$$\mathcal{D}(\phi_1 \circledcirc \phi_2) = \mathcal{D}(\phi_1) \cup \mathcal{D}(\phi_2)$$

Algorithme de mise en forme normale Grâce aux opérations de fusion et d'effacement, il est possible de constuire un algorithme simple qui construit une espèce en forme normale à partir d'une espèce bien formée. Pour cela on construit en une liste contenant les champs des espèces parents et ceux de l'espèce courante. Ensuite, il faut fusionner les champs définissants les mêmes noms pour obtenir enfin une liste où chaque nom est introduit une seule fois. A chaque fusion il faut aussi effectuer les effacements nécessaires. La liste résultante constitue alors la liste des champs d'une espèce en forme normale qui est équivalente à l'espèce de départ.

L'algorithme proposé prend en entrée une espèce $s = ([s_i]_{1 \leq i \leq m}, [\phi_j]_{1 \leq j \leq n})$ bien formée et procède par trois étapes qui sont décrites dans les paragraphes suivantes.

Première étape Il s'agit de constuire la liste des champs nécessaires et ce dans un ordre approprié au bon fonctionnement de l'algorithme de l'étape suivante.

Sachant qu'il n'y a pas de cycle possible dans le graphe d'héritage, on suppose que, pour tout $i \in [\![1, m]\!]$, $\text{norm}(s_i)$ est la liste des champs calculé par l'application de cet algorithme sur s_i. Soit σ une permutation de $[\![1, n]\!]$ telle que

$$\forall i \in [\![1, n]\!], \quad \wr \phi_{\sigma(i)} \int_s^{\text{decl}} \subset \bigcup_{1 \leq j \leq i} \mathcal{N}(\phi_{\sigma(j)})$$

Cette permutation existe puisque s est bien formée donc il suffit de prendre la permutation correspondant à un tri des champs suivant l'ordre $\blacktriangleleft_s^{\text{decl}}$.

On définit alors

$$\Phi = \left(\bigoplus_{1 \leq i \leq m} \text{norm}(s_i) \right) \oplus [\phi_{\sigma(j)}]_{1 \leq j \leq n}$$

où $\oplus$ est l'opération de concaténation de deux listes.

Deuxième étape A partir de la liste Φ, on construit Ψ, une liste de champs introduisant des noms deux à deux distincts. L'algorithme 1 donne les détails de cette construction. Cette liste Ψ permet de constuire une espèce s' que l'on définit par :

$$s' = ([\,], \Psi)$$

Algorithme 1 Consolidation des champs

Entrée: une liste de champs Φ
Sortie: une liste de champs Ψ introduisant des noms deux à deux distincts

initialiser Ψ à $[\,]$
tant que $\Phi \neq [\,]$ **faire**
  Soient ϕ un champ et l une liste tels que $\phi.l = \Phi$
  si $\mathcal{N}(\phi) \cap \mathcal{N}(\Psi) = \emptyset$ **faire**
    $\Phi \leftarrow l$
    $\Psi \leftarrow \Psi.\phi$
  sinon
    Soient h, t deux listes et ψ un champ tels que : $\begin{cases} \Psi = h \oplus \psi.t \\ \mathcal{N}(\phi) \cap \mathcal{N}(h) = \emptyset \\ \mathcal{N}(\phi) \cap \mathcal{N}(\psi) \neq \emptyset \end{cases}$
    $\Phi \leftarrow l$
    $\Psi \leftarrow h.(\mathcal{E}_{\mathcal{N}(\psi)}(t)) \oplus [\psi \ominus \phi]$

Théorème 5.5.9 (Unicité des noms)**.**
Pour tous ψ, ψ' éléments distincts de Ψ, on a

$$\mathcal{N}(\psi) \cap \mathcal{N}(\psi') = \emptyset$$

Démonstration. Il s'agit d'un invariant de la boucle dans l'algorithme 1. □

Théorème 5.5.10 (Def-dépendances)**.**
Pour tout $i \in [\![1, n]\!]$, on a

$$\wr \psi_i \int^{\text{def}} \subset \bigcup_{1 \leq j < i} \mathcal{N}(\psi_j)$$

Démonstration. Il s'agit d'un invariant de la boucle dans l'algorithme 1. □

Théorème 5.5.11 (Bonne formation)**.**
L'espèce s' est bien formée.

Démonstration. [Pre03] démontre ce résultat grâce aux propriétés de l'effacement et à partir de la bonne formation de s. □

Troisième étape La deuxième étape assure que les champs de Ψ sont ordonnés selon $\blacktriangleleft_{s'}^{\text{def}}$ mais pas selon $\blacktriangleleft_{s'}^{\text{decl}}$. Or, l'espèce est bien formée, donc $\blacktriangleleft_{s'}^{\text{decl}}$ est un ordre strict et il suffit de trier Ψ selon cet ordre pour obtenir une liste de champs définissant une espèce sous forme normale.

5.6 Bilan

Grâce aux définitions et algorithmes de ce chapitre, il est possible de constuire le début d'un compilateur Focal vers n'importe quel language même si celui-ci n'est pas orienté objets. En effet, l'algorithme de mise en forme normale élimine l'héritage. Ce qui manque est une manière de traduire une espèce en forme normale vers le langage voulu. Notons que si ce langage supporte la modularité, il est souhaitable de s'en servir pour le cloisonnement et la réutilisation de code.

Chapitre 6

Traduction vers Coq

Pour définir la traduction vers Coq, je me baserai sur la traduction existante (chapitre 4) et sur l'analyse statique (chapitre 5). D'abord je définirai la forme d'un générateur de méthode pour une fonction récursive. Ensuite je me tournerai vers la formulation de la preuve de terminaison sous Focal et la compilation de cette preuve vers Coq. Enfin, quelques considérations sur l'héritage et les effacements éventuelles d'une preuve de terminaison seront en ordre.

6.1 Définition d'une fonction récursive

La forme du générateur de méthode sera dictée par l'emploi de **Function** et par quelques points importants concernant l'analyse statique et les champs introduisant une fonction récursive.

La preuve d'une fonction récursive fait partie du champ ainsi que le corps de la fonction. De plus, cette preuve peut être retardée, effacée et refaite. Or ceci n'est pas possible dans Coq et il faut alors séparer la fonction et sa preuve.

La possibilité de retarder la preuve implique aussi qu'on doit considérer que la fonction est récursive générale et se ramener à cette situation dans les cas d'une récursion structurelle.[1]

Néanmoins, cette preuve ne peut en aucun cas figurer parmi les champs du type enregistrement. En effet, le fait qu'une fonction termine va de pair avec le fait qu'on puisse la définir sur un domaine non vide. Ceci implique qu'il est impossible (pour ne pas dire inutile) de formuler ce fait sous la forme d'une proposition en Coq. Aussi faut-il identifier l'énoncé en Coq de la

[1] Il est tout à fait possible de faire l'inverse, *i.e* supposer qu'on peut se ramener à une récursion structurelle, mais vu le travail fait pour **Function**, ce serait bien plus délicat.

terminaison à la conjonction des obligations de preuve générées par **Function** et changeantes selon le corps de la fonction récursive.

Enfin, vu ces contraintes et vu ce qu'attend **Function**, il est évident que cette preuve de terminaison et l'ordre utilisée dans la preuve doivent figurer parmi les arguments du générateur de méthode.

Dans le cas de `quicksort` cela donne le code Coq suivant :

```
Section Quicksort.
  Variable param__a_T: Set.
  Let abst_T: Set := list__t(param__a_T).
  Variable abst_partition:
    (param__a_T) →
    (abst_T) →
    prod abst_T abst_T.
  Variable abst_wforder:
    (abst_T) → (abst_T) → Prop.
  Variable abst_termination:
    (∀ (h: param__a_T) (t: abst_T),
    abst_wforder (__g_first(abst_partition h t)) (Cons h t))
    ∧ (∀ (h: param__a_T) (t: abst_T),
    abst_wforder (__g_scnd(abst_partition h t)) (Cons h t))
    ∧ (well_founded abst_wforder).

  Function sorting__quicksort (l: abst_T)
    {wf my_order l}: (abst_T) :=
    match l with
    | Nil ⇒ Nil
    | Cons h t ⇒
      let (less, more) := abst_partition h t in
      __g_append (sorting__quicksort less)
      (Cons h (sorting__quicksort more))
    end.
  Proof.
    decompose [and] abst_termination; assumption.
    decompose [and] abst_termination; assumption.
    decompose [and] abst_termination; assumption.
  Qed.
End Quicksort.
```

6.2 Preuve de terminaison

6.2.1 Preuve sous Focal

En conçevant une manière de montrer la terminaison sous Focal, il faut considérer deux points importants.

Premièrement, il n'est pas souhaitable de considérer que la terminaison d'une fonction prend des formulations différentes selon le corps de la fonction. En effet, un des principaux traits de Focal est sa simplicité par rapport à d'autres ateliers de preuve. En plus d'être simple, considérer qu'il existe un prédicat de terminaison sur l'ensemble des fonctions offre plus de souplesse pour introduire des principes prouvant la terminaison autres que la bonne fondation d'un ordre et des appels décroissants (un exemple ne serait rien d'autre que la structuralité).

Le deuxième souci est que la terminaison d'une fonction, dans le cas général, fait intervenir la notion de bonne fondation. Or le type du constructeur du prédicat d'accessibilité dépasse les capacités du typage en Focal.

indications de preuve. La réponse à cette première, dont une ébauche est déjà dessiné, est de considérer que l'utilisateur doit prouver la Terminaison (de chaque fonction récursive et de lui fournir quelque moyens de le faire qui passent par des mots clés annonçant quel moyen est utilisé dans un cas précis suivi des éléments nécessaires pour compléter la construction. J'en donne deux : les preuves par bonne fondation et par structuralité. Les deux syntaxes sont résumées ci-dessous.

– Dans le cas où `f` est une fonction récursive structurelle :

```
let rec f(x1, x2, ..., xi, ...) =
  (* corps de f *)
proof: structural xi;
```

où `f`, `x1`, `x2`, ... sont des identificateurs et le type de `xi` est inductif.

– Dans le cas où `f` est une fonction récursive *non* structurelle :

```
let rec f(x1, x2, ..., xi, ...) =
  (* corps de f *)
proof:
  <1>1 wforder o xi
    <2>1
      (* preuve de bonne fondation de o *)
    <2>2
      (* preuve de décroissance de chaque
       * appel récursif                  *)
```

```
    <1>2 qed;
```

où `f`, `x1`, `x2`, ... sont des identificateurs et `o` un ordre sur le type de `xi`.

inclusion de la bonne fondation. Pour manipuler le prédicat de bonne fondation dans les preuves Focal, et ainsi prouver que certains ordres sont bien fondés, il convient d'employer le méchanisme d'inclusion de code Coq pour importer ce prédicat directement dans Focal. En effet, le type du prédicat en question est tout à fait correct (il attend en argument un type et un ordre sur ce type) et le lien entre la notion sous Focal et sous Coq est fait trivialement. Ensuite, pour montrer la bonne fondation d'un ordre, vu qu'on n'a pas accès aux constructeurs de Coq à partir du code source Focal, mettre à disposition quelques ordres bien fondés sur les types de base (dont la preuve est faite nécessairement en Coq) et quelques théorèmes qui montrent la bonne fondation d'un ordre à partir d'un ordre qui lui est lié. Les détails de cet ajout à la bibliothèque standard, ainsi que la preuve de bonne fondation nécessaire à l'exemple étudié au long de ce mémoire, sont contenus à l'annexe A.

le cas de quicksort. Dans le cas de `quicksort`, la terminaison dépend des lemmes suivants dont on suppose la preuve :

```
theorem partition_length_l: all p in a, all l in self,
  #int_lt(
    #length(#first(!partition(p,l))),
    #length(p::l))
proof: [...];

theorem partition_length_r: all p in a, all l in self,
  #int_lt(
    #length(#scnd(!partition(p,l))),
    #length(p::l))
proof: [...];
```

On peut à partir de ces lemmes et de la bonne fondation de `length_lt` démontrer la terminaison de `quicksort`.

```
proof:
  <1>1 wforder #length_lt l
    by !partition_length_l, !partition_length_r, #wf_length_lt
    def #length_lt
  <1>f qed
```

6.2.2 Preuve sous Coq

Compte tenu des remarques faites à la section précédente, la preuve de terminaison et l'ordre bien fondé utilisé dans cette preuve figureront dans le code Coq sous la forme de générateurs de méthodes, même s'il ne s'agit pas de méthodes. En effet, dans le cas contraire, les dépendances introduites par la preuve ne seraient pas prises en compte.

La compilation vers Coq de cette preuve donne alors :

```
Definition sorting__wforder_quicksort(l l': self_T): Prop:=
  __g_length_lt.
Let self_wforder_quicksort := sorting__wforder_quicksort.

Section Termination_quicksort.
  Variable param__a_T: Set.
  Let abst_T: Set := list__t(param__a_T).
  Let abst_wforder:
    abst_T → abst_T → Prop :=
    __g_length_lt.
  Variable abst_partition:
    (param__a_t)→ (abst_T)→
    (prod abst_T abst_T).
  Variable abst_partition_length_r:
    ∀ (p: param__a_T), ∀ (l: abst_T),
    __g_int_lt(
      __g_length(__g_first(abst_partition(p,l))),
      __g_length(Cons(p,l))).
  Variable abst_partition_length_l:
    ∀ (p: param__a_T), ∀ (l: abst_T),
    __g_int_lt(
      __g_length(__g_scnd(abst_partition(p,l))),
      __g_length(Cons(p,l))).

  Theorem sorting__termination_quicksort:
    (∀ (h: param__a_T) (t: abst_T),
    __g_length_lt (__g_first(abst_partition h t)) (Cons h t))
    ∧ (∀ (h: param__a__t) (t: abst_T),
    __g_length_lt (__g_scnd(abst_partition h t)) (Cons h t))
    ∧ (well_founded abst_wforder).
  (* preuve générée par Zenon *)
  Qed.
End termination_quicksort.
```

```
Let self_termination_quicksort :=
  (sorting__termination_quicksort
   self_wforder_quicksort self_partition).
```

Chapitre 7

Conclusion

Grâce à la nouvelle construction **Function**, nous avons vu comment il est simple de définir proprement des fonctions récursives en Coq. Il est possible aussi de s'appuyer sur cette construction pour définir de manière analogue ces fonctions récursives en Focal et une nouvelle syntaxe accompagnée d'une sémantique peut être introduite qui est cohérente avec l'existent.

Pourtant il reste quelques points à voir avant d'implanter ces changements. On remarque aussi que dans la compilation de la preuve de terminaison, j'ai nommé le lemme correspondant en ajoutant le préfixe `termination_` au nom de la fonction. Or, cela peut créer des problèmes si on choisit des noms de membres qui commencent comme une fonction récursive de l'espèce et qui terminent par ce suffixe. En effet, la preuve de terminaison et le membre auraient le même nom dans le script Coq ce qui n'est pas permis. Mais des techniques de renommage comme celles employées déjà par le compilateur Focal, peuvent éviter de telles collisions.

Ensuite, la preuve de terminaison nécessite de connaître l'expression exacte des obligations de preuves. Dans le cas de `quicksort` les obligations sont faciles à déterminer à la main, mais dans d'autres cas on ne peut pas laisser cette tâche au développeur : il serait plus simple d'inclure un outil dans l'atelier Focal qui pourrait les générer.

Enfin, trouver un ordre bien fondé pour lequel les arguments des appels récursifs décroissent est plutôt trivial dans le cas de `quicksort`, mais on rencontre très vite des fonctions où cela est bien plus difficile. Il existe des outils qui traîtent de la terminaison de fonctions ou de systèmes de réécriture[1] et qui peuvent alors être utiles à l'atelier Focal.

[1] À toute fonction, même récursive, on peut faire correspondre un système de réécriture dont la preuve de terminaison est équivalente à celle de la fonction.

Annexe A

Bonne fondation de `Zwf`

Dans cette annexe, je donne les détails de la preuve de bonne fondation de `length_lt` et celle du lemme `well_founded_Zwf_compat` qui permet, à partir de l'existence d'une fonction de compatibilité entre un ordre et `Zwf`, de démontrer que cet ordre est bien fondé.

Sachant que `__g_length` correspond à la compilation de la fonction de Focal qui renvoie la longueur d'une liste et que `__g_length_ge_0` assure que la longueur d'une liste est positive ou nulle, le raisonnement développé à la section **??** correspond à la preuve Coq suivante :

Listing A.1 – quicksort.v (suite)

```
Section ZwfLtCompat.
Open Scope Z_scope.
Variable A:Type.
Variable f:A → Z.
Variable R: A → A → Prop.
Variable c:Z.

Hypothesis H_compat: ∀ (x y:A), R x y → Zwf c (f x) (f y).

Let g (z:Z):= Zabs_nat (z-c).

Theorem well_founded_Zwf_compat: well_founded R.
Proof.
  red in |- *; intros.
  assert (∀ (n:nat) (a:A),
      (g (f a) < n)%nat ∨ f a < c → Acc R a).
    clear a; simple induction n; intros.
(* n=0 *)
```

```
    case H; intros.
    case (lt_n_O (g (f a))); auto.
    apply Acc_intro; intros.
    assert (H2:=H_compat _ _ H1);
    unfold Zwf in H2;
    assert False;
    omega || contradiction.
(* inductive case *)
    case H0; clear H0; intro; auto.
    apply Acc_intro; intros.
    apply H.
    assert (H2:=H_compat _ _ H1);
    unfold Zwf in H2.
    case (Zle_or_lt c (f y)); intro; auto with zarith.
    left.
    red in H0.
    apply lt_le_trans with (g (f a)); auto with arith.
    unfold g in |- *.
    apply Zabs.Zabs_nat_lt; omega.
    apply (H (S (g(f a)))); auto.
Qed.

End ZwfLtCompat.

Lemma well_founded_Zwf: ∀ c:Z, well_founded (Zwf c).
Proof.
  intro.
  apply well_founded_Zwf_compat with (c:=c) (f:=fun x:Z ⇒
x).
  intros; assumption.
Qed.

Definition length_lt (l l':list__t int__t): Prop:=
  (__g_length l < __g_length l')%Z.

Lemma well_founded_length_lt: well_founded length_lt.
Proof.
  apply well_founded_Zwf_compat with (c:=0) (f:=__g_length).
  intros. unfold Zwf. split.
    apply __g_length_ge_0.
    auto.
```

Qed.

Bibliographie

[BFPR06] G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive functions : a practical tool for the Coq proof assistant. In M. Hagiya and P. Wadler, editors, *Proceedings of FLOPS'06*, volume 3945 of *Lecture Notes in Computer Science*, pages 114–129. Springer-Verlag, 2006.

[coq] The Coq proof assistant. http ://coq.inria.fr.

[DHD04] C. Dubois, T. Hardin, and V. Viguié Donzeau Gouge. Building certified components within FOCAL. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming*, volume 5, pages 33–48, Bristol, UK, 2004. Intellect.

[Foc] Equipe Focal. http ://focal.inria.fr.

[oca] Le langage Caml. http ://caml.inria.fr.

[Pre03] V. Prevosto. *Conception et implantation du langage FOC pour le développement de logiciels certifiés*. PhD thesis, Université Paris 6, 2003.