Typage des modules récursifs en Caml

Stage au LIP, ENS Lyon

Romain Bardou

Juin–Juillet 2005

Table des matières

1 Introduction

Programmer modulairement présente de nombreux avantages : réutilisabilité du code lui-même plus ciblé et indépendant, compilation séparée... Cette modularité est facilitée si le langage propose des constructions dédiées : les objets en sont un exemple, le langage des modules en Caml [4] en est un autre. Ce dernier présente l'avantage de posséder des foncteurs, fonctions sur les modules. Ainsi on peut créer des modules paramétrés, implémentés indépendamment du contexte et donc plus facilement réutilisables.

Nous allons nous intéresser au cas des modules en Caml, et plus particulièrement aux modules récursifs ou mutuellement récursifs. En effet, lorsque l'on découpe un projet en modules, il n'est pas rare qu'apparaissent naturellement des dépendances circulaires. Néanmoins, dans le cas d'un langage qui se veut fortement typé tel que Caml, il est nécessaire de vérifier à la compilation le bien-fondé de ces constructions récursives.

Nous verrons donc dans un premier temps quelles sont les limites de l'implémentation actuelle des modules récursifs en Caml. Ensuite nous décrirons la proposition de Tom Hirschowitz et Sergueï Lenglet [2]. Le travail effectué pendant le stage a consisté à intégrer ce système au sein de Caml.

2 Les modules récursifs en Caml

2.1 Définition

2.1.1 Module et signature

Un *module* est un ensemble de déclarations de types, de valeurs, ou d'autres modules. Une *signature* est le type d'un module, il s'agit de l'ensemble des types, valeurs et modules qu'il nous permet d'utiliser de l'extérieur. On peut par exemple décrire ce qu'est un type ordonné par la signature suivante :

```
module type OrderedType =
sig
  type t
  val compare : t -> t -> int
end
```

Cette signature possède un type nommé `t`, et une opération de comparaison sur les éléments de ce type. Notez qu'on ne donne pas la définition de `t` : ce type est abstrait. Ainsi, si on n'a pas plus d'information, tout ce qu'on pourra faire sur une valeur de type `t` sera appliquer la fonction `compare`.

Le module suivant peut se voir comme un `OrderedType` :

Exemple 1

```
module OrderedInt =
struct
  type t = int
  let compare = compare
end
```

Ce module décrit le type ordonné des entiers. Le type `t` est donc `int`, et `compare` utilise `Pervasives.compare`, qui est une fonction de comparaison générique en Caml. On peut voir ce module comme un `OrderedType` en oubliant que `t` est implémenté par `int`. Il s'agit de l'opération d'abstraction de type.

2.1.2 Coercion

Le module `OrderedInt` a pour signature :

```
sig type t = int
    val compare: int -> int -> int end
```

On a vu qu'on pouvait voir ce module comme un `OrderedType`. On peut créer un module de signature `OrderedType` mais implémenté comme `OrderedInt` par une *coercion*, en notant :

```
module C = (OrderedInt : OrderedType)
```

On peut éventuellement « oublier » des champs afin d'interdire l'utilisateur du module de les utiliser. De plus, l'ordre des champs dans la signature peut changer lors d'une coercion. Dans l'exemple suivant, le champ `pi` n'apparaît pas dans la signature, et on a inversé l'ordre des champs `circ` et `aire` :

Exemple 2

```
module C : sig val circ: float -> float val aire: float -> float end =
struct
  let pi = 3.141592
  let aire r = pi *. r *. r
  let circ r = 2. *. pi *. r
end
```

2.1.3 Foncteur

Un *foncteur* est une fonction qui prend un module en argument et qui renvoie un module calculé à partir de cet argument. La librairie standard de Caml [4] fournit par exemple le foncteur `Set.Make`, qui prend en argument un module de signature `OrderedType` et qui renvoie un module de signature `Set.S`. Ce nouveau module fournit des opérations courantes sur les ensembles, comme l'union ou l'intersection ; et la particularité de cet ensemble est que ses éléments ont pour type le type `t` passé en argument.

Par exemple, le module :

```
module IntSet = Set.Make(OrderedInt)
```

permet de gérer des ensembles d'entiers. Le module `IntSet` a été calculé par `Set.Make`, qui n'a eu besoin pour cela que du type des éléments (`t`) et d'une opération de comparaison sur ceux-ci (pour pouvoir les trier et ainsi obtenir une implémentation efficace).

On peut ensuite utiliser `IntSet` comme n'importe quel module, voici par exemple comment on pourrait créer l'ensemble d'entiers $\{1, 2\}$, puis afficher le contenu de cet ensemble :

```
let set12 = IntSet.add 1 (IntSet.singleton 2)
IntSet.iter
  (fun x -> print_string (string_of_int x);
            print_newline () )
  set12
```

2.1.4 Récursion mutuelle

Voyons d'abord un exemple de module récursif, autrement dit un module qui s'utilise lui-même :

```
module rec A : sig val f : int -> int end =
struct
  let f x = if x = 0 then 1 else x*(A.f (x-1))
end
```

La fonction `A.f` est la fonction factorielle. Notez comme on l'a définie à partir de `A.f`.

Voici maintenant un exemple classique [3, 2] de modules mutuellement récursifs, l'exemple des arbres de degrés arbitraires :

Exemple 3

```
module rec A :
sig
  type t = Leaf of string | Node of ASet.t
  val compare: t -> t -> int
end = struct
  type t = Leaf of string | Node of ASet.t
  let compare t1 t2 =
    match (t1, t2) with
      (Leaf s1, Leaf s2) -> Pervasives.compare s1 s2
    | (Leaf _, Node _) -> 1
    | (Node _, Leaf _) -> -1
    | (Node n1, Node n2) -> ASet.compare n1 n2
end
and ASet : Set.S with type elt = A.t = Set.Make(A)
```

Cet exemple définit deux modules : `A` et `ASet`. `A` décrit un `OrderedType` particulier puisqu'il s'agit d'un arbre. Ses éléments sont donc soit des feuilles (`Leaf`), soit des nœuds (`Node`) qui peuvent avoir un nombre arbitraire de fils. On a donc décidé de représenter ces nœuds par un ensemble d'arbres. Cet ensemble est décrit par `ASet`, qui n'est rien d'autre que `Set.Make` appliqué à `A`.

Ainsi, `A` dépend de `ASet` et `ASet` dépend de `A`.

2.1.5 Dépendance forte ou faible

Remarquons rapidement une chose à propos du foncteur `Set.Make` : il n'a pas besoin de connaître, pour produire le module `IntSet`, la fonction `compare` ; il a simplement besoin de savoir qu'elle existe. En effet, le module renvoyé par `Set.Make` propose des fonctions qui utilisent `compare`, mais on n'a pas besoin d'appliquer `compare` pour générer ces fonctions elles-mêmes. On dit que `Set.Make` dépend *faiblement* de son argument.

Au contraire, le foncteur suivant a besoin de connaître une au moins des valeurs de son argument (en l'occurrence `i`) et en dépend donc *fortement* :

Exemple 4
```
module F = functor (Arg: sig val i : int end) ->
struct
  let j = Arg.i + 1
end;
```

2.2 Représentation

2.2.1 Représentation des modules

Dans nos exemples de programmes compilés on utilisera une syntaxe proche de Caml à une différence près : on se permettra de définir directement des enregistrements. De plus, tous leurs champs seront mutables. Enfin, on suppose qu'il existe une valeur notée `nil` qui représente un contenu non défini.

On représente la valeur d'un module par un enregistrement, et un foncteur par une fonction. Voici comment compiler les exemples 1 et 4 :

```
let OrderedInt = { compare = Pervasives.compare }
let F = fun Arg -> { j = Arg.i + 1 }
```

Les coercions sont-elles même compilées. En effet, il peut être nécessaire de :
- réordonner les champs
- ne pas conserver tous les champs

L'exemple 2 pourra se compiler ainsi :

```
let C' = { pi = 3.141592;
           aire = fun r -> pi *. r *. r;
           circ = fun r -> 2. *. pi *. r }
let C = { circ = C'.circ;
          aire = C'.aire }
```

2.2.2 Représentation naturelle des modules récursifs

Une représentation naturelle des modules récursifs serait la suivante : pour chaque module, on crée une boîte vide (un enregistrement avec des champs valant `nil`). Cette boîte vide tiendra lieu de valeur temporaire pour le module correspondant. Ensuite on calcule le contenu de chacun des champs et on copie la valeur obtenue dans le champ qui valait auparavant `nil`.

Par exemple, la définition suivante :

Exemple 5
```
module rec A : sig val f : int -> int end =
struct
  let f x = B.i + x
end
and B : sig val i : int end =
struct
  let i = 0
```

```
  let j = 1
end
```

Peut se compiler ainsi :

```
(* initialisation *)
let A = { f = nil } in
let B = { i = nil } in
(* calcul *)
let A_f = fun x -> B.i + x in
let B_i = 0 in
let B_j = 1 in
(* recopie *)
  A.f <- A_f;
  B.i <- B_i
```

Notez que les champs qui n'apparaissent pas dans la signature (ici `B.j`) sont calculés, pour assurer que les effets de bord aient bien lieu, mais qu'on n'a pas besoin de les copier ensuite.

Ce qu'il faut retenir de cette méthode, c'est l'idée qu'on se fait naturellement des modules récursifs (et même des définitions récursives en général) : on n'a besoin que du nom des modules et des champs pour calculer l'ensemble des modules. Si, pendant la création des modules (entre l'initialisation et la recopie), on accède à un champ d'un des modules, une erreur se produit puisqu'on accède à un module initialisé à `nil`.

Il faut donc s'assurer que l'on n'accédera pas à ces champs pendant la création du module. On pourrait pour cela n'autoriser, comme pour les `let rec`, que des abstractions (`fun x -> ...`) ou des calculs paresseux (`lazy`[1]) comme membres droits des `let` ; mais ça serait trop restrictif dans le cas des modules (notamment, le programme de l'exemple 3 serait refusé).

Le but est donc de trouver une méthode plus fine pour assurer le bien-fondé des définitions récursives.

2.2.3 Représentation actuelle des modules récursifs

Xavier Leroy a proposé la représentation suivante [3], qui est celle utilisée dans la version 3.08.3 de Caml.

Supposons qu'on ait n modules mutuellement récursifs. On commence par déterminer les modules dits *sûrs*. Les modules sûrs sont ceux ceux dont tous les éléments sont de type fonctionnel ou paresseux. Par exemple, le module `A` dans l'exemple 3 est sûr, puisque sa signature possède un type et une valeur de type fonctionnel. Le module `ASet`, lui, n'est pas sûr, puisque sa signature contient la valeur `empty` qui n'est pas sûre (en l'occurrence il s'agit de l'objet représentant l'ensemble vide dont le type n'est ni fonctionnel ni paresseux).

Ensuite, on regarde les dépendances entre les modules. On vérifie qu'il n'y a pas de cycle de dépendance qui passe uniquement pas des modules non sûrs. Si c'est le cas on peut trouver un ordre dans lequel compiler les modules, un ordre

[1] On peut demander explicitement de retarder le calcul d'une expression `e` à l'aide de `lazy e`. On obtient une expression de type `type_e Lazy.t` où `type_e` est le type de `e`. `e` n'est pas calculé, il le sera lorsqu'on fera la première fois `Lazy.force` pour lire sa valeur. [4]

qui assure que l'on n'accède pas au contenu d'un module non sûr qui ne serait pas encore défini.

Ensuite, on initialise les blocs qui vont contenir les modules sûrs. On se souvient que ceux-ci ne contiennent que des valeurs sûres, et on initialise les fonctions par :

```
fun _ -> raise Undefined_recursive_module
```

Et les `lazy` par :

```
lazy (raise Undefined_recursive_module)
```

Ainsi, si on appelle la fonction ou qu'on déclenche le calcul du `lazy`, une exception est déclenchée.

Une fois cette initialisation effectuée, on calcule la valeur de chaque module dans l'ordre que l'on a déterminé plus tôt. Ce calcul peut utiliser les autres modules en train d'être définis. Pour les modules sûrs, on utilise le bloc initialisé à l'aide des exceptions. Pour les autres, l'ordre que l'on a trouvé assure que cela a du sens.

Enfin, on remplace le contenu de tous les blocs d'initialisation des modules sûrs, par la valeur du module calculée. On obtient donc, au final, les dépendances récursives que l'on souhaitait.

Cette méthode ressemble à la méthode naturelle, sauf que :
- `nil` est remplacé par une levée d'exception puisqu'on s'autorise quelques définitions mal fondées ;
- les modules sont séparés en modules sûrs et non sûrs ;
- les modules sont réordonnés.

Les définitions mal fondées qui ne sont pas refusées ne génèrent néanmoins pas d'erreurs à l'exécution[2] : à la place elles déclenchent une exception. En effet, si on accède à un champ d'un module, alors soit le module est sûr et dans ce cas une exception est lancée, soit le module n'est pas sûr et on sait qu'on l'a déjà calculé. L'exemple 5 se compile alors ainsi :

```
(* initialisation des modules surs *)
let A = { f = fun _ -> raise Undefined_recursive_module } in
(* calcul des modules dans un ordre respectant les dependances *)
let B_i = 0 in
let B_j = 1 in
(* B n'etant pas sur, on le cree directement *)
let B = { i = B_i } in
let A_f = fun x -> B.i + x in
(* recopie des modules surs *)
  A.f <- A_f
```

2.2.4 Limites

Exemple 6

```
module rec A : sig val f : int -> int end =
struct
```

[2] Erreurs de segmentation, entre autres. Les exceptions ne sont pas considérées comme des erreurs à l'exécution parce qu'elles font partie du déroulement normal du programme.

```
  let f x = x
end
and B : sig val i : int end =
struct
  let i = A.f 1
end;;
```

Cet exemple lance l'exception `Undefined_recursive_module`. En effet, `A` est sûr, `B` ne l'est pas et dépend de `A`. Donc on compile l'exemple ainsi :

```
let A = { mutable f = fun _ -> raise Undefined_recursive_module } in
let B = { i = A.f 1 } in
  A.f <- fun x -> x
```

On voit donc bien que `B` applique `A.f` alors qu'il n'est pas encore défini.

Si l'on avait écrit cet exemple sans utiliser de récursion, on aurait obtenu exactement ce qu'on voulait. En effet, dans cet exemple on n'a fondamentalement pas besoin de récursion, puisque `A` ne dépend ni de `A` ni de `B`, et que `B` ne dépend que de `A`. C'est un premier problème.

Un second problème est qu'au lieu de refuser de compiler l'exemple, au lieu de le rejeter au moment de typer les modules, on obtient une exception à l'exécution. Ceci affaiblit le typage des modules. Néanmoins il reste sûr, puisqu'il assure toujours l'absence d'erreurs à l'exécution.

3 Approche proposée

L'approche proposée consiste à compiler autrement les modules récursifs et à déterminer plus finement le bien-fondé des récursions.

3.1 Représentation par modification en place immédiate

Remarquons que la méthode de compilation précédente calcule tous les modules d'un coup, et ensuite seulement met à jour les modules sûrs par leur valeur. Au lieu d'attendre d'avoir tout calculé, on va mettre à jour les modules sûrs immédiatement après leur calcul, sans les réordonner. Ainsi l'exemple précédent devient correct, puisqu'il se compile de la façon suivante :

```
let A = { mutable f = fun _ -> raise Undefined_recursive_module } in
  A.f <- fun x -> x;
  let B = { i = A.f 1 }
```

Remarquons que l'exemple 3 passe dans les deux méthodes de compilation.

On pourrait penser qu'on obtient ainsi une expressivité plus forte. Néanmoins, s'il est vrai que certains programmes qui ne compilaient pas correctement auparavant passent désormais sans erreur, l'inverse est aussi vrai. En effet, l'ordre dans lequel on calcule les modules a une importance plus forte. Ce n'est pas forcément souhaitable. Et on ne recalcule plus cet ordre : le programmeur doit y faire attention lui même.

Ceci nous permet cependant de compiler plus de définitions récursives de modules, par exemple une définition faisant intervenir deux modules non sûrs dépendant l'un de l'autre. En effet, on n'a plus besoin d'initialiser les modules avec des levées d'exceptions. Il faudra par contre faire plus attention au typage.

3.2 Typage

L'implémentation actuelle de Caml se base uniquement sur le type des signatures pour savoir si une définition récursive de modules est bien fondée. L'idée est de faire, au typage toujours, une analyse plus fine du code pour vérifier le bien-fondé de la récursion.

Pour cela, on construit un graphe représentant les dépendances de chaque module envers les autres. Par exemple, si `B` apparaît dans `A` comme variable libre, alors `A` dépend de `B`. On ajoute alors une arête $\texttt{A} \longrightarrow \texttt{B}$.

Ensuite, il faut vérifier qu'un module n'accède pas à un module non défini. Pour cela, il suffit de vérifier que le graphe de dépendances est inclus dans l'ordre de définition des modules. Autrement dit, si `A` est défini avant `B`, alors il ne doit pas y avoir d'arête $\texttt{A} \longrightarrow \texttt{B}$.

Dépendance forte ou faible

Cependant, cela n'est pas encore assez fin. Aucun module ne pourrait utiliser les modules définis après lui. Pour être plus précis on différencie deux types de dépendances : les dépendances faibles et les dépendances fortes. L'idée est que si le calcul de `A` requiert l'accès à un champ de `B`, alors `A` dépend fortement de `B`.

On utilise en réalité une approximation de cette définition. Par exemple, si `B` apparaît dans `A` sous une abstraction (dans le corps d'une fonction) non appliquée ou sous un `lazy` non forcé, alors la dépendance est faible.

Pour simplifier, si `A` a plusieurs champs qui dépendent de `B`, on dit que `A` dépend lui-même de `B`. Si l'une des dépendances était forte, la dépendance de `A` en `B` est forte et on note $\texttt{A} \xrightarrow{\bullet} \texttt{B}$. Sinon elle est faible et on note $\texttt{A} \xrightarrow{\circ} \texttt{B}$.

Transitivité

Cependant, que se passe-t-il si $\texttt{A} \xrightarrow{\bullet} \texttt{B} \xrightarrow{\circ} \texttt{C}$? `A` risque d'accéder au contenu de `B`, qui dépend de `C`. Alors par transitivité `A` dépend de `C`, et il est nécessaire de définir qu'il en dépend fortement. En effet, considérons l'exemple suivant :

```
module rec A : sig val a : int end =
  struct let a = B.f 0 end
and B : sig val f : int -> int end =
  struct let f x = x + (C.f x) end
and C : sig val f : int -> int end =
  struct let f x = x + A.a end
```

On a $\texttt{A} \xrightarrow{\bullet} \texttt{B} \xrightarrow{\circ} \texttt{C} \xrightarrow{\circ} \texttt{A}$. Mais pour calculer `A` on applique `B.f`, donc on applique `C.f`, donc on a besoin de connaître `A.a`. Donc `A` dépend fortement de `A`.

La règle doit donc être la suivante : lorsqu'on a $\texttt{A} \xrightarrow{\bullet} \texttt{B} \longrightarrow \texttt{C}$, alors on a $\texttt{A} \xrightarrow{\bullet} \texttt{C}$.

De même, on a la règle suivante : si $\texttt{A} \xrightarrow{\circ} \texttt{B} \longrightarrow \texttt{C}$ alors $\texttt{A} \xrightarrow{\circ} \texttt{C}$. En effet, si `A` dépend faiblement de `B`, il n'accède de toute façon pas à ses champs et cela ne change rien que `B` dépende fortement ou non de `C` : dans les deux cas `A` dépend faiblement de `C`.

Au final, pour savoir si une définition récursive de modules est bien fondée, on vérifie que la restrictions au arêtes $\xrightarrow{\bullet}$ de la clôture transitive ainsi définie

du graphe de dépendances est incluse dans l'ordre (strict) de définition des modules.

Application de foncteurs

Il reste cependant un cas dont on n'a pas parlé : que se passe-t-il quand on applique un foncteur, par exemple dans l'exemple 3 ? Dans :

```
and ASet : ... = Set.Make(A)
```

Que doit-on mettre comme dépendances ? Tout d'abord, il est clair que `ASet` dépend de `Set` et de `A`. Et il est nécessaire, de toute façon, de mettre une dépendance forte en `Set`, puisqu'on accède à son champ `Make` (donc il faut qu'il soit initialisé au moment où on calcule `ASet`).

Supposons que l'on mette une dépendance forte en `A`. Alors on aurait les dépendances `ASet` $\xrightarrow{\bullet}$ `A` $\xrightarrow{\circ}$ `ASet` et, par transitivité, `ASet` $\xrightarrow{\bullet}$ `ASet`. Donc l'exemple serait refusé. Mais est-il légitime de mettre une dépendance faible ?

On a remarqué en 2.1.5 que `Set.Make` était un foncteur qui dépendait faiblement de son argument. On dit qu'il s'agit d'un foncteur faible. Cela signifie qu'il n'a besoin que du nom de son argument pour calculer son résultat. Donc en réalité, `ASet` peut dépendre faiblement de `A` parce que d'une part `A` n'est qu'une variable, et d'autre part `Set.Make` est un foncteur faible.

Remarque 1 *En fait, la notion de foncteur faible proposée initialement [2] est un peu plus restrictive : elle impose que le corps du foncteur ne dépende fortement d'aucune variable. Cette condition a d'ailleurs été affaiblie par l'extension proposée en 4.1.2, qui l'autorise à dépendre fortement de variables extérieures à toute définition récursive.*

Résumons

La modification proposée pour le système de type se base sur deux idées principales. D'une part, on différencie les foncteurs faibles et les foncteurs forts. On crée donc un nouveau type, que l'on notera à l'aide d'une flèche double et de la liste des arguments du foncteur entre crochets, par exemple :

```
module F : functor [Arg: sig ... end] => sig ... end
  = functor (Arg: sig end) -> struct ... end
```

D'autre part, on analyse de façon statique le graphe de dépendances des modules définis récursivement afin de savoir si on peut assurer, dans le cadre de la modification en place immédiate, qu'aucun module ne tentera d'accéder au contenu d'un module qui n'est pas encore défini.

Remarque 2 *Pour déterminer si un foncteur sera typé faiblement ou non, on se base sur le graphe de dépendances de son corps. Et pour calculer le graphe on a besoin du type des foncteurs. Les deux idées sont donc reliées.*

4 Travail réalisé

4.1 Améliorations théoriques

4.1.1 Simplification des règles de typage

En programmant les règles de typage [2], on s'est rendu compte que le test d'inclusion du graphe des dépendances dans l'ordre de définition des modules était parfois effectué deux fois. En effet, si une définition de modules mutuellement récursifs apparaît dans une structure[3] `S`, on effectue le test d'inclusion au moment de typer cette structure (règle LetRec), mais aussi au moment de construire le graphe (règle G-Struct). Comme de toute façon on type toutes les définitions récursives, on n'a pas besoin de le faire au moment de construire le graphe.

4.1.2 Nœuds internes et externes

Considérons l'exemple suivant :

```
module F = functor (Arg : sig end) ->
struct
  let i = 1 + 2
end
```

De quoi dépend `F` ? Apparemment d'aucun autre module. Mais en réalité, il utilise la fonction `(+)` du module `Pervasives`, qui est le module automatiquement ouvert au démarrage de tout programme Caml.

Cette dépendance est néfaste. En effet, `F` dépend fortement d'un autre module. Comment savoir si cette dépendance est dangereuse ou non ? Si cet autre module était en train d'être défini récursivement en même temps que `F`, le foncteur devrait être considéré comme un foncteur fort, puisqu'au moment de l'appliquer on risque d'accéder à ce module.

Pour résoudre ce problème, on différencie les nœuds du graphe de dépendances. On distingue deux cas.

Nœud interne Il représente un module auquel on a accès mais qui n'est pas forcément déjà calculé.

Nœud externe Il représente un module auquel on a accès et qui a déjà été calculé, par exemple un élément de la librairie standard.

On peut alors typer faiblement un foncteur s'il n'y a pas de dépendance forte de son corps en un nœud interne.

4.1.3 Pré-allocation et modification en place profonde

Lorsqu'on a une application de foncteur, on ne met une dépendance faible en l'argument que si :

- le foncteur est faible et
- l'argument est une variable.

[3] Donc, par exemple, dans un module lui-même défini récursivement.

Cette dernière condition s'est révélée trop forte, en particulier pour typer l'exemple de Jacques Garrigue [1] (voir partie 4.3). En fait nous avons remarqué qu'on pouvait utiliser non seulement des variables, mais aussi des projections, du type `A.B.C.D`.

Pour cela il faut cependant étendre la modification en place immédiate. En effet, il ne faut plus initialiser les modules par des blocs vides mais par un arbre de blocs partiellement initialisés. Par exemple, un module `A` ayant la signature suivante :

```
sig
  module B : sig
    module C : sig val i: int end
    val j: int -> int
  end
  module D : sig end
end
```

Serait initialisé ainsi :

```
let A = {
  B = {
    C = { i = nil };
    j = nil };
  D = {} }
```

Ainsi, on peut parler de `A.B.C` même si `A` n'est pas encore calculé. Une fois `A` calculé, il ne faut plus se contenter de le recopier dans son bloc pré-alloué. Il faut recopier toute l'arborescence :

```
A.B.C.i <- ... ;
A.B.j <- ...
```

Non seulement cette extension nous permet de typer plus de définitions, mais elle est aussi plus proche de l'implémentation des modules dans la dernière version de Caml.

4.1.4 Coercions et dépendances fortes

On a vu que les coercions ont un contenu calculatoire et qu'une copie des champs pouvait avoir lieu. C'est pourquoi, dans un premier temps, on a considéré que toute coercion créait des dépendances fortes en les variables libres du module coercé. Mais cela implique de refuser certains programmes corrects et intéressants, comme celui de Jacques Garrigue [1].

Nous avons remarqué que les coercions qui sont effectuées au niveau de la déclaration des modules récursives ne sont pas néfastes. En effet, au moment où la coercion a lieu, le membre droit est de toute façon sensé être copié et il a donc déjà été initialisé. La coercion ne fait que remplacer une copie complète par une copie partielle. On peut donc en toute sûreté ignorer ces coercions lors du calcul des dépendances.

4.2 Implémentation

Le code source d'OCaml se présente en plusieurs parties. Celles qui nous intéressent sont les suivantes :

- Parseur : On doit y rajouter notre syntaxe pour le type des foncteurs faibles. Ceci demande de modifier le parseur ainsi que l'arbre de syntaxe abstraite.
- Typeur : Celui-ci prend un arbre de syntaxe abstraite et renvoie un arbre y ressemblant fort, mais avec des informations de type en plus. On ajoute à cet arbre le type des foncteurs faibles. Ceci permet d'ailleurs de savoir où on devra modifier le code source existant, puisqu'au moment de compiler, on a un message d'avertissement pour chaque endroit où on doit rajouter le traitement des foncteurs faibles.
  Ensuite on modifie le typage des déclarations de modules récursives pour vérifier la condition sur le graphe de dépendances, et on modifie le typage des foncteurs pour essayer de les typer faiblement. Enfin, on rajoute les règles de sous-typage.
  On a écrit un module de graphes orientés étiquetés pour gérer les graphes de dépendances, et un module qui gère toutes les règles de typage spécifiques au typage des modules récursifs (en particulier le calcul du graphe de dépendances).
- Compilateur : On remplace la compilation des modules récursifs actuelle par la modification en place immédiate.
- Divers : On peaufine ensuite les détails, en ajoutant par exemple une option sur la ligne de commande permettant d'afficher les graphes de dépendances.

Le plus intéressant est de comprendre le code source d'origine pour s'y intégrer. On rencontre cependant de nombreuses difficultés pour comprendre les nombreux détails de l'implémentation actuelle. Bien que l'on se soit intéressé uniquement aux modules récursifs, il est nécessaire de parcourir toute les constructions possibles pour construire le graphe de dépendances ; et une opération banale telle que le calcul des variables libres d'une expression peut étrangement paraître d'un coup moins évidente.

Entre autres, il est intéressant de voir comment est compilé un programme Caml ; celui-ci est en fait d'abord compilé en un langage intermédiaire appelé « lambda », lui-même compilé en bytecode ou en code natif. C'est lors de cette compilation que disparaît la notion de module. Pour implémenter la modification en place immédiate il faut donc découvrir et comprendre le lambda.

4.3 Résultats

Au final, le tout marche très bien, du moins sur les exemples que l'on a testés. L'exemple 3, entre autres, type et compile correctement. L'exemple suivant [3], au lieu de lancer une exception `Undefined_recursive_module`, est refusé au typage :

```
module rec Bad : sig val f : int -> int end =
struct
  let f = let y = Bad.f 5 in
          fun x -> x+y
end
```

L'exemple 6 page 7 type, compile et s'exécute correctement au lieu de lancer l'exception `Undefined_recursive_module`.

Nous avons aussi testé l'exemple `mixmod2` de Jacques Garrigue [1] qui fonctionne lui-aussi sans problèmes. C'est d'ailleurs pour faire fonctionner cet exemple que nous avons dû rajouter les extensions présentées dans les parties 4.1.3 et 4.1.4. Il s'agit d'implémenter de façon extensible un langage de programmation, et cet exemple utilise intensivement les modules récursifs.

5 Travaux futurs : application au langage de base

En Standard ML (SML), on ne peut définir que des `let rec` dont les membres droits sont des fonctions. Imaginons qu'on veuille par exemple définir un menu `m` composé de deux sous-menus `mi1` et `mi2` dont les étiquettes sont respectivement « A » et « B » et dont l'action consiste à appliquer une fonction donnée (ici `f`) sur son voisin. Voici une façon de représenter un tel menu[4] :

```
let rec m = (mi1, mi2)
and mi1 = ("A", fun () -> f mi2)
and mi2 = ("B", fun () -> f mi1)
```

Cette définition est refusée en SML. En effet un couple n'est pas une fonction.

En Caml, cette définition est acceptée (pour peu que `f` ait le bon type, par exemple `'a -> unit`). En effet la condition à respecter par les membres droits des définitions récursives est plus compliquée. On ne va pas la détailler, mais on va simplement signaler le fait qu'elle reste insuffisante. En effet, si pour construire nos sous-menus on avait une fonction `makemenuitem`, on aurait utilisé le code suivant :

```
let rec m = (mi1, mi2)
and mi1 = makemenuitem "A" (fun () -> f mi2)
and mi2 = makemenuitem "B" (fun () -> f mi1)
```

Si on compile, on obtient le message d'erreur :

```
This kind of expression is not allowed as right-hand side of 'let rec'
```

Et ce, quelque soit l'implémentation de `makemenuitem`, en particulier celle qui renvoie le couple (nom, action) :

```
let makemenuitem name f = (name, f)
```

Le programmeur, lorsqu'il rencontre ce type d'erreur, doit trouver un moyen de contourner le problème. Souvent il doit expliciter lui-même la façon de compiler la récursion, par exemple à l'aide de références d'options[5] :

```
let content r = match !r with
    None -> failwith "Empty content"
  | Some x -> x
```

[4] On conserve la syntaxe Caml pour nos exemples SML.

[5] Le type `option` est défini par : `type 'a option = None | Some of 'a` et fait partie de la librairie standard [4].

```
let mi1' = ref None
let mi2' = ref None
let _ = mi1' := Some(makemenuitem "A" (fun () -> f (content mi2')))
let _ = mi2' := Some(makemenuitem "B" (fun () -> f (content mi1')))
let mi1 = content mi1'
let mi2 = content mi2'
let m = (mi1, mi2)
```

Les inconvénients sont multiples :
- le programmeur doit faire lui-même la conversion[6],
- il y a un risque d'erreurs à l'exécution,
- des structures dont on aimerait s'abstraire apparaissent (ici les références d'options), et
- l'idée que toutes les définitions récursives sont définies « en même temps » est perdue puisque le programmeur doit trouver et spécifier l'ordre dans lequel créer ses différents objets.

Une idée pour améliorer le système actuel serait de reprendre la notion de dépendance faible vue pour les modules.

6 Conclusion

L'implémentation actuelle des modules récursifs en Caml présente l'inconvénient de ne pas refuser certaines définitions mal fondées, qui lanceront une exception à l'exécution. Tom Hirschowitz et Sergueï Lenglet [2] on proposé une autre façon de compiler et de typer les définitions récursives de modules, et l'implémenter dans Caml a permis de tester ce système sur des exemples pratiques. Ces tests ont permis de rencontrer les limites du système et de les repousser en ajoutant plusieurs extensions.

Pendant mon stage j'ai donc eu l'occasion de comprendre une théorie jusqu'à être capable de l'implémenter dans un « vrai » système, plus riche et en perpétuelle évolution. L'avantage est qu'on est sûr de comprendre jusqu'au moindre détail, et qu'on a une vision du problème plus proche de la réalité. Cela m'a permis de mieux comprendre un certain nombre d'idées qui se cachent derrière la programmation d'aujourd'hui, et pas seulement à propos des modules récursifs. Et en même temps cela m'a donné une idée plus concrète de l'implémentation actuelle de Caml.

Je tiens à remercier tout particulièrement Tom Hirschowitz et les autres membres du LIP pour leur accueil et leur disponibilité tout au long de mon stage dans leur équipe.

Références

[1] Jacques Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, 2000. Exemples disponibles

[6] Non seulement cela lui donne du travail en plus, mais cela crée un risque d'erreur supplémentaire : le compilateur ne peut pas assurer le programmeur qu'il obtiendra un point fixe pour sa définition récursive.

sur le Web, `http://wwwfun.kurims.kyoto-u.ac.jp/~garrigue/papers/fose2000.html`.

[2] Tom Hirschowitz and Sergueï Lenglet. A practical type system for generalized recursion. Technical Report RR2005-22, UMR CNRS - ENS Lyon - UCB Lyon - INRIA 5668, 2005.

[3] Xavier Leroy. A proposal for recursive modules in Objective Caml. Disponible sur le Web, `http://pauillac.inria.fr/~xleroy/publi/recursive-modules-note.pdf`, 2003.

[4] Xavier Leroy, Damien Doligez, Jacques Garrigue, and Jérôme Vouillon. The Objective Caml system. Logiciel et documentation disponibles sur le Web, `http://caml.inria.fr/`, 1996–2003.