

Typage des modules récursifs en Caml

Stage au LIP, ENS Lyon

Juin-Juillet 2005

Typage des modules récursifs en Caml

- Modules
- Représentation
- Typage
- Problèmes
- Améliorations proposées
- Résultats

Programmation modulaire

Découper un programme en modules permet :

- de traiter chaque module indépendamment
- d'avoir un code plus facilement réutilisable

Modules

Programmer modulairement est plus facile si le langage propose des constructions adaptées :

- Objets
- Modules

Module

Un module est une liste de définitions.

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

Signature

Une signature est le type d'un module.

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

Foncteur

Un foncteur est un module paramétré.

```
module F(Arg: OrderedType) =  
  struct  
    let compare2 x1 x2 y1 y2 =  
      Arg.compare x1 x2, Arg.compare y1 y2  
  end
```

```
module IntComp2 = F(OrderedInt)
```

Représentation d'un module

Une structure est représentée par un enregistrement.

```
let OrderedInt = {  
  compare = Pervasives.compare  
}
```


Représentation d'un foncteur

Un foncteur est représenté par une fonction.

```
let F Arg = {  
  compare2 = fun x1 x2 y1 y2 ->  
    Arg.compare x1 x2, Arg.compare y1 y2  
}
```

```
let IntComp2 = F OrderedInt
```

Module récursif

Un module peut être récursif.

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

Modules mutuellement récursifs

Des modules peuvent être mutuellement récursifs.

```
module rec A : sig val pair: int -> bool end =  
  struct  
    let pair x = (x = 0) || (B.impair (x-1))  
  end  
and B : sig val impair: int -> bool end =  
  struct  
    let impair x = not (x = 0) &&  
      ( (x = 1) || (A.pair (x-1)) )  
  end
```

Calcul des modules récursifs

Comment calculer A, si A dépend de A ?

Il faut que A n'accède pas à son propre contenu au moment où on veut le calculer.

Récursion mal fondée :

```
module rec A :  
sig val i: int end =  
struct  
  let i = A.i  
end
```

```
type t = { i: int }  
let rec A = {  
  i = A.i  
}
```

Calcul des modules récursifs

On pourrait réserver un bloc non initialisé pour chaque module.

```
let A = { fact = nil } in
let A_fact = fun x ->
  if x = 0 then 1 else x*(A_fact (x-1)) in
A_fact <- A_fact
```

Calcul des modules récursifs

```
let A = { pair = nil } in
let B = { impair = nil } in
let A_pair = fun x ->
  (x = 0) || (B.impair (x-1)) in
let B_pair = fun x -> not (x = 0) &&
  ( (x = 1) || (A.pair (x-1)) ) in
A.pair <- A_pair;
B.pair <- B_pair
```

A et B sont dans le corps de fonctions qui ne sont pas appliquées avant d'avoir tout calculé.

Calcul des modules récursifs

```
module rec A : sig val i: int end =  
struct  
  let i = A.i  
end
```

Devient :

```
let A = { i = nil } in  
let A_i = A.i in  
  A.i <- A_i
```

Ici on accède à A.i qui n'est pas initialisé (**nil**).

Représentation actuelle

En Caml les modules récursifs sont compilés ainsi :

1) Séparation en modules sûrs et non sûrs

Module sûr : module dont la signature ne contient que :

- des fonctions,
- des types paresseux.

```
module type Safe =  
sig  
  type t  
  val f : int -> int  
  val v : int Lazy.t  
end
```

```
module type Unsafe =  
sig  
  type t  
  val f : int  
  val v : int Lazy.t  
end
```


Représentation actuelle

En Caml les modules récurifs sont compilés ainsi :

2) Calcul d'un graphe de dépendances :

Les sommets du graphe sont les modules.

Si le module A dépend du module B, alors on met une arête allant de A à B.

3) Présence d'un cycle ne faisant intervenir que des modules non sûrs ?

Si c'est le cas, on ne pourra pas trouver un ordre dans lequel calculer les modules correctement.

Représentation actuelle

En Caml les modules rékursifs sont compilés ainsi :

4) On ordonne les modules :

On place en premier les modules non sûrs, ordonnés de telle façon que si A est avant B, alors A ne dépend pas de B.

Ensuite viennent les modules sûrs.

Représentation actuelle

En Caml les modules récursifs sont compilés ainsi :

5) On peut enfin générer nos enregistrements :

On commence par les modules sûrs en créant un enregistrement non initialisé, avec :


```
nil = raise Undefined_recursive_module
```

Ensuite on calcule chaque module dans l'ordre déterminé précédemment.

Puis on recopie les modules sûrs dans le bloc qui leur a été réservé.

Représentation actuelle

```
module rec A : sig ... end =  
struct  
  let i = C.f 1  
end  
and B : sig ... end =  
struct  
  let i = A.i + 1  
  let f x = x + 1  
end  
and C : sig ... end =  
struct  
  let f x = x + B.f 1  
end
```

- Seul C est sûr.
- Graphe : 
- Ordre : A, B, C

```
let C = { f = fun () -> raise  
Undefined_recursive_module }  
let A = { i = C.f 1 }  
let B = { i = A.i + 1;  
  f = fun x -> x + 1 }  
let C_f = fun x -> x + B.f 1 in C.f  
<- C_f
```

Typage

Le typage des modules consiste simplement à typer chaque élément pour obtenir une signature.

Il permet de refuser certaines définitions récursives mal fondées.

Problèmes

- Certaines définitions mal fondées sont acceptées au typage (et, à l'exécution, lancent l'exception `Undefined_recursive_module`).
- Certaines définitions bien fondées sont refusées à la compilation alors qu'on a des exemples utiles qui les utilisent.
- Certaines définitions bien fondées lancent l'exception `Undefined_recursive_module`.

Amélioration proposée


- Conserver l'idée du graphe de dépendances pour détecter les définitions mal fondées.
- Utiliser un graphe plus détaillé pour accepter plus de définitions bien fondées.
- Modifier la compilation des modules récursifs.


Dépendances fortes et faibles


- Un module A peut dépendre d'un module B faiblement ou fortement.
- La dépendance est forte si le module A accède au contenu du module B.
- Au contraire, la dépendance est faible si le module A n'a besoin que du nom de B (en l'occurrence, « B »).


Dépendances fortes et faibles

```
module rec A : sig ... end =  
struct  
  let f x = x + B.i  
end  
and B : sig ... end =  
  let i = A.f 1  
end
```

A dépend faiblement de B :  A $\xrightarrow{\circ}$ B

B dépend fortement de A :  B $\xrightarrow{\bullet}$ A

A dépend faiblement de A :  A $\xrightarrow{\circ}$ A

B dépend fortement de B :  B $\xrightarrow{\bullet}$ B

Dépendances fortes et faibles

Si $A \overset{\bullet}{\rightarrow} B \longrightarrow C$, alors $A \overset{\bullet}{\rightarrow} C$.

Si $A \overset{\circ}{\rightarrow} B \longrightarrow C$, alors $A \overset{\circ}{\rightarrow} C$.

Règle de typage des structures

Si A dépend fortement de B, il faut avoir calculé B pour calculer A.

On impose que B soit défini avant A.

Typage des foncteurs

Un foncteur peut dépendre faiblement de son argument (ou pas).

```
module FWeak(Arg: sig val i: int end) =  
  struct  
    let f x = Arg.i + x  
  end
```

```
module FStrong(Arg: sig val i: int end) =  
  struct  
    let j = Arg.i + 1  
  end
```

Typage des foncteurs

```
module rec A : sig val i: int end = F(A)
```

Si F est un foncteur fort : A dépend fortement de A.
Sinon : A dépend faiblement de A.

Dans tous les cas A dépend fortement de F.

Sous-typage des foncteurs

Un foncteur faible peut être vu comme un foncteur fort. L'inverse n'est pas vrai.

Résultats

Tous nos objectifs sont atteints.

Deux détails néanmoins :

- Le programmeur doit faire attention au type des foncteurs.
- On a pas une expressivité rigoureusement plus grande, dans le sens où certains programmes auparavant acceptés et bien fondés, sont maintenant refusés au typage.

Ch'tite démo

... emacs est notre ami ...

... ou pas ...

Conclusion

Implémenter une telle modification fait apprendre pas mal de choses sur :

- l'organisation d'un compilateur « moderne »,
- le pourquoi du parce que des idées qui sont derrière Caml (qui a dit Xavier Leroy ?), en particulier sur le typage.

J'ai eu l'occasion d'implémenter une théorie très... théorique, dans un milieu « réel », et même de l'étendre.

FIN

:)