Calcul statique des applications de modules paramétrés

Julien Signoles

1: Laboratoire de Recherche en Informatique UMR 8623
Bât 490., 91405 Orsay Cedex, France
Julien.Signoles@lri.fr

Résumé

Le système de module d'*Objective Caml* fournit au programmeur la possibilité de réutiliser facilement du code grâce aux modules paramétrés encore appelés foncteurs. Cependant, l'utilisation des foncteurs entraîne un surcoût à l'exécution et pose problème aux outils d'analyse statique.

Bêta-réduire statiquement les applications de foncteurs pallie ces défauts mais pose des problèmes de captures de variables et de préservation des effets de bord. Dans cet article, nous présentons une méthode effectuant cette opération en résolvant ces problèmes. D'un point de vue théorique, nous montrons qu'elle préserve la sémantique, y compris en présence d'effets de bord. Nous présentons également son implantation.

1. Introduction

La programmation de composants logiciels réutilisables est essentielle dans la conception et la construction de logiciels. Les *systèmes de modules* à la SML [Que84] fournissent au programmeur la possibilité d'écrire de tels composants grâce aux *modules paramétrés*, appelés *foncteurs*.

Cependant, ils possèdent deux défauts majeurs restreignant le cadre de leur utilisation :

- D'une part, les outils d'analyse statique d'un programme ne fonctionnent en général pas correctement en présence de modules paramétrés.
- D'autre part, un programme utilisant des modules paramétrés est traditionnellement compilé en un code plus lent qu'un programme équivalent n'en comportant pas.

Ce sont les deux raisons pour lesquelles calculer statiquement les applications de foncteurs est une opération utile pour élargir leur champ d'utilisation.

C'est cette opération, nommée *défonctorisation*, que nous présentons ici. Elle consiste à récrire un programme en un autre programme qui lui est équivalent mais qui ne comporte plus d'applications de foncteurs à calculer au moment de l'exécution.

Cet article est organisé de la manière suivante. Section 2, nous présentons le système de modules d'*Objective Caml* (OCaml) [LDG+] et notamment les modules paramétrés. Section 3, nous exposons le principe de la défonctorisation, les problèmes qu'elle pose et comment il est possible de les résoudre. Nous montrons en section 4 que la transformation principale conduisant à la défonctorisation préserve la sémantique d'un programme, même impératif. Section 5, nous comparons notre travail avec ceux déjà existants.

2. Le système de modules d'OCaml

Nous présentons ici le système de modules d'OCaml. Il peut être vu comme un langage fonctionnel typé relativement indépendant du langage de base. Les termes de ce langage sont les *modules* qui peuvent être ou non paramétrés.

Seules les constructions importantes pour notre étude sont présentées. Le lecteur intéressé peut se référer au manuel d'utilisation d'OCaml [LDG+] pour plus de détails sur les autres constructions.

2.1. Structures

Les modules non paramétrés sont appelés *structures* et regroupent des définitions apparentées (comme la définition d'un type de données et des opérations sur ce type).

Par exemple, la structure `OrdInt` (ci-dessous) définit le type des entiers muni d'une relation d'ordre :

```
module OrdInt = struct
  type t = int
  let eq x y = (x = y)
  let lt x y = (x < y)
end
```

On accède à la définition `eq` de `OrdInt` en utilisant la notation pointée `OrdInt.eq`.

Le type d'une structure est une *signature* représentant son interface. En poursuivant le même exemple, la signature du module `OrdInt` est :

```
module type ORD = sig
  type t
  val eq : t → t → bool
  val lt : t → t → bool
end
```

Un autre exemple de signature, donné ci-après, permet de définir des dictionnaires polymorphes :

```
module type DICT = sig
  type key                                 (* type des clés du dictionnaire *)
  type 'a t                                         (* type du dictionnaire *)
  val empty : 'a t                                     (* dictionnaire vide *)
  val add : key → 'a → 'a t → 'a t                     (* ajout d'une donnée *)
  val find : key → 'a t → 'a                      (* recherche d'une donnée *)
end
```

2.2. Foncteurs

Les modules paramétrés sont appelés *foncteurs* et sont des fonctions des modules dans eux-mêmes.

Par exemple, la signature `DICT` donnée précédemment peut être implantée sous la forme d'un arbre équilibré si l'on dispose d'une fonction de comparaison sur ses clés. Pour cela, on peut écrire un foncteur prenant en paramètre une structure modélisant les clés :

```
module MakeDict = functor(Key : ORD) → struct
    type key = Key.t
    type 'a t = Empty | Node of key * 'a * 'a t * 'a t
```

```
    let empty = Empty
    let add = ...
    let find = ... Key.lt ...
  end
```

Il est alors possible de réaliser des dictionnaires basés sur des entiers en *appliquant* ce foncteur à la structure «entiers ordonnés» :

```
module DictInt = MakeDict(OrdInt)
```

2.3. `open` et `include`

Il existe deux constructions dans le langage ayant pour but de faciliter l'écriture de programmes modulaires : les instructions `open` et `include`. La première permet d'ajouter, dans l'espace de noms courant, les noms définis dans un module spécifié. La seconde permet d'effectuer la même chose et ajoute, en outre, ces noms dans l'interface en cours de définition.

Par exemple,

```
module X = struct
  open OrdInt
  (* on peut utiliser directement le type t de OrdInt *)
  type u = t
end
```

définit un module `X` ayant un type `u` équivalent à `OrdInt.t` et

```
module Y = struct
  include OrdInt
  (* les noms définis dans OrdInt sont exportés dans Y *)
  let gt x y = (x > y)
end
```

définit un module `Y` ayant la même signature que `OrdInt` augmentée de la fonction `gt : t → t → bool`.

3. Présentation de la défonctorisation

3.1. Motivation

En OCaml, une fonction est appelée *via* sa clôture. Cependant, si tous les arguments sont connus au moment de la compilation, le compilateur natif appelle directement le code de cette fonction sans passer par sa clôture, ce qui supprime une indirection. En outre, l'application d'une fonction utilisant un paramètre d'un foncteur est toujours partielle. Par suite, l'appel d'une telle fonction se fera toujours *via* sa clôture. De plus, une telle application partielle peut masquer des optimisations au compilateur.

En outre, les foncteurs posent également des problèmes aux outils d'analyse statique qui ont du mal à gérer leur présence.

Le but de la défonctorisation est de résoudre ces problèmes en transformant un programme en un autre qui lui soit équivalent mais qui ne comporte plus d'applications de foncteurs.

3.2. Principe

Le principe de la défonctorisation est très simple : il s'agit de remplacer chaque application par le corps du foncteur dans lequel l'argument réel a été substitué au paramètre formel. Cette opération est assimilable à une β-réduction en lambda-calcul.

En reprenant l'exemple des dictionnaires de la section 2.2, le module `DictInt` est transformé en :

```
module DictInt = struct
  type key = OrdInt.t
  type 'a t = Empty | Node of key * 'a * 'a t * 'a t
  let empty = Empty
  let add = ...
  let find = ... OrdInt.lt ...
end
```

Toutes les occurences de `Key` ont été substituées par `OrdInt`.

Malheureusement, se contenter de n'effectuer qu'une simble substitution terme à terme ne fonctionne pas toujours, car un certain nombre de problèmes peuvent survenir.

3.3. Problème 1 : arguments quelconques et effets de bord

Un premier problème résulte de la possibilité de passer n'importe quelle structure - et pas seulement un identificateur - en argument d'une application de foncteur. Ainsi, le programme suivant est-il correct :

```
module F = functor(X : ...) → struct
  let y = X.x
  let z = X.x
end
module M = F(struct let x = <<corps de x>> end)
```

Cependant, ce programme ne peut pas être défonctorisé tel quel. En effet, dans ce cas, deux problèmes se posent, un syntaxique et un sémantique.

D'une part, on ne peut pas, en OCaml, substituer une variable par une structure quelconque dans un chemin d'accès car, syntaxiquement, ces derniers ne peuvent être constitués que d'identificateurs. Ainsi, dans l'exemple précédent, on ne peut pas substituer `X` par `struct let x = «corps de x» end` dans le corps du foncteur.

D'autre part, si l'on opère la défonctorisation du programme ci-dessus, la sémantique des effets de bord n'est pas préservée. En effet, alors que dans le programme initial les effets de bord présents dans `<<corps de x>>` ne sont calculés qu'une fois avant l'application du foncteur, ceux-ci sont calculés dans chaque chemin d'accès après défonctorisation et donc exécutés deux fois.

Il est possible de résoudre cette difficulté assez simplement en «sortant» l'argument de l'application de foncteur, en lui assignant un nouveau nom de module et en passant ce nom en argument. Il devient alors possible de défonctoriser normalement. Sur l'exemple précédent, on obtient :

```
module F = functor(X : ...) → struct
  let y = X.x
  let z = X.x
end
module New1 = struct let x = <<corps de x>> end
module M = F(New1)
```

3.4. Problème 2 : capture de variables

En substituant toutes les occurrences du paramètre d'un foncteur par le nom de son argument, il est possible de *capturer* certaines variables : au moment où une variable est utilisée, elle ne fait plus référence à sa déclaration originale, comme dans l'exemple suivant :

```
module X = struct
  let x = ...
  module Y = struct
    module F = functor(...) → struct
      let y = x
    end
    let x = ...
    module M = F(...)
  end
end
```

L'occurrence de `x` qui apparaît dans `M` après calcul de l'application ne fait plus référence à la première déclaration de `x`, mais à la seconde : *elle a été capturée* !

Ce problème est bien connu en lambda-calcul et, pour y remédier, on effectue un simple renommage de variables. Cependant, cette solution ne peut s'appliquer dans notre contexte de programmation modulaire. En effet, si on renomme une des deux variables `x`, on change la signature du module qu'on est en train de définir et, par là même, on change la sémantique du programme.

En fait, on remarque qu'un identificateur a deux rôles distincts : celui de *lieur* d'une part, et celui de *nom* d'autre part. Les *lieurs* sont renommables alors que les *noms* ne le sont pas. En lambda-calcul, seule existe la notion de lieur, celle de nom n'existant pas. Ainsi, la solution au problème de captures de variables, proposée dans [HL94], consiste à distinguer les deux rôles.

Le module `X` ci-dessus se récrit alors de la façon suivante :

```
module X as X1 = struct
  let x as x1 = ...
  module Y as Y1 = struct
    module F as F1 = functor(...) → struct
      let y as y1 = x1
    end
    let x as x2 = ...
    module M as M1 = F1(...)
  end
end
```

On associe à chaque définition de nom `x` un lieur *via* le mot-clé `as`. A l'intérieur de `X`, on accède à `x` par son lieur et, à l'extérieur, on y accède par son nom `X.x`. Comme on peut s'assurer de l'unicité des lieurs, il n'y a plus de risques de capture de variables dans `M` au moment du calcul de l'application de foncteur.

Cependant, de telles déclarations avec lieurs n'existent pas dans OCaml. C'est la raison pour laquelle on va se placer dans un autre système de modules possédant de tels lieurs : le système de modules $\mathcal{MC}$ introduit dans [Cou97a, Cou97b].

Sur un plan pratique, il faudra convertir les modules d'OCaml en modules de $\mathcal{MC}$ (ce qui est trivial) et réciproquement (ce qui l'est moins, cf. 3.5).

3.5. Problème 3 : suppression des lieurs

Nous venons de voir qu'il nous fallait pouvoir convertir un programme $\mathcal{MC}$ en un programme OCaml pour résoudre le problème de capture de variables et, donc, qu'il nous fallait supprimer les lieurs introduits.

Il est bien évident qu'il ne faut pas se contenter de remplacer les lieurs par les noms des variables correspondantes sous peine de retomber sur le problème de capture de variables.

Pour éviter celui-ci, on peut *encapsuler* chaque déclaration de variables dans un nouveau module et, lors de l'utilisation d'une variable, on y fait référence par son nom long. Malheureusement, ce faisant, on ne préserve pas la sémantique de la structure. En effet, les variables encapsulées n'apparaîssent plus dans la signature. Pour résoudre cette difficulté, il faut exporter chacune de ces variables dans la structure.

Ainsi, la structure

```
struct
  let x as x1 = ...
  let y as y1 = x1
end
```

est-elle transformée en

```
struct
  module New1 = struct let x = ... end
  let x = New1.x
  module New2 = struct let y = New1.x end
  let y = New2.y
end
```

3.6. Problème 4 : `open` et `include`

Les instructions `open` et `include` posent également problème. En effet, elles peuvent entraîner des captures de variables, comme le montre l'exemple suivant :

```
let x = ... (* 1 *)
module F = functor(X : sig end) -> struct
  open X
  let y = x
end
module A = struct let x = ... (* 2 *) end
module M = F(A)
```

Sur cet exemple, l'occurrence de `x` dans `F` fait référence à la déclaration 1 de la première ligne. Cependant, lorsque le module `M` est défonctorisé, la déclaration 2 de `x` dans `A` capture l'occurrence de `x` à cause du `open` qui masque la déclaration 1. Le même problème survient en remplaçant le `open` par un `include`.

Pour résoudre ce problème, on peut supprimer les `open` et les `include` dans une phase préliminaire à la défonctorisation.

3.7. Algorithme

Les sections précédentes ont mis en lumière les différentes phases à respecter pour défonctoriser un programme OCaml. En résumé, la structure de l'algorithme effectuant la défonctorisation est donc la suivante :

Entrée :

Un programme OCaml $\mathcal{P}$

Phases :

1. Suppression des `open` et des `include`
2. Passage de OCaml à $\mathcal{MC}$
3. Réduction des applications de foncteurs (dans $\mathcal{MC}$)
4. Passage de $\mathcal{MC}$ à OCaml

Sortie :

Un programme sans application de foncteur équivalent à $\mathcal{P}$.

4. Formalisation

Le but de cette section est de montrer que l'algorithme établi à la section 3.7 préserve la sémantique, *i.e.* que le programme résultant de l'algorithme est équivalent au programme initial.

Dans cet article, nous nous concentrons sur la phase centrale, à savoir la phase 3 de réduction des applications de foncteurs dans le langage $\mathcal{MC}$. Nous souhaitons montrer que cette phase préserve la sémantique, y compris la sémantique des effets de bord. Or, le langage $\mathcal{MC}$ tel qu'il est défini et étudié dans [Cou98] est un langage fonctionnel pur. Nous allons donc considérer une extension de $\mathcal{MC}$ intégrant des effets de bord : le langage $\mathcal{MC}_{SE}$.

En outre, nous intéressant à la sémantique, nous n'étudierons pas le typage de ce langage afin de rendre l'étude praticable.

4.1. Grammaire de $\mathcal{MC}_{SE}$

La grammaire du langage $\mathcal{MC}_{SE}$ se subdivise en une grammaire sur le langage de modules et en une autre sur le langage de base.

Le langage de modules est semblable à celui donné dans [Cou97b] tandis que le langage de base est un lambda-calcul typé avec constantes entières auquel on ajoute des références afin d'y intégrer des effets de bord.

Chemins d'accès :

$p ::=$	x	Identificateur
	$\mid\ m.x$	Accès à une valeur

Expressions de modules :

$m ::=$	p	Chemin d'accès
	$\mid$ `struct` s `end`	Structure
	$\mid$ `functor`$(x : M)m$	Foncteur
	$\mid\ m(m)$	Application de foncteur

Corps de structure :

$s ::=$	ϵ	Structure vide
	$\mid\ b\,;s$	Liste de composants

Composants de structure :

b ::=	t	Définition du langage de base
	$\mid$ `module` $x \triangleright^1$ x' = m	Définition de module

[1] Signatures :

M ::=	`sig` S `end`	Corps de signature
	$\mid$ `functor`$(x : M)M$	Signature de foncteur

Corps de signature :

S ::=	ϵ	Corps vide
	$\mid$ B ; S	Liste de composants

Composants de signature :

B ::=	t	Déclaration du langage de base
	$\mid$ `module` $x \triangleright x'$: M	Déclaration de module

Déclarations du langage de base :

t ::=	`let` $x \triangleright x'$ = e	Définition de valeur
	$\mid$ `val` $x \triangleright x'$: τ	Déclaration de valeur
	$\mid$ `type` $x \triangleright x'$ = τ	Type manifeste
	$\mid$ `type` $x \triangleright x'$	Type abstrait

Expressions du langage de base :

e ::=	p	Chemin d'accès
	$\mid$ $\underline{\mathrm{n}}$	Constante entière
	$\mid$ $\lambda x.e$	λ-abstraction
	$\mid$ $(e\ e)$	λ-application
	$\mid$ `ref`(e)	Allocation
	$\mid$ $!e$	Lecture
	$\mid$ $p \leftsquigarrow e$	Ecriture

Types du langage de base :

τ ::=	p	Chemin d'accès
	$\mid$ `int`	Entiers
	$\mid$ $\tau \to \tau$	Type flèche
	$\mid$ `ref`(τ)	Référence

4.2. Sémantique de $\mathcal{MC}_{SE}$

La sémantique de $\mathcal{MC}_{SE}$ fait usage d'une mémoire (ensemble dénombrable d'adresses-mémoires énuméré par une fonction $nième_{\mathcal{L}}$). On étend le langage $\mathcal{MC}_{SE}$ en $\mathcal{MC}_{SE}^{\mathcal{L}}$ contenant en plus des expressions usuelles des constantes adresses-mémoires.

Définition 1 (Valeur)
Les *valeurs* de $\mathcal{MC}_{SE}$ appartiennent à $\mathcal{MC}_{SE}^{\mathcal{L}}$ et sont définies inductivement par la syntaxe abstraite ci-après. L'ensemble de ces valeurs est noté $\mathcal{V}$.

Valeurs des expressions de modules :

v_m ::=	`struct` v_s `end`	Valeur de structure
	$\mid$ `functor`$(x : M)m$	Foncteur

[1] la notation $\triangleright$ correspond au mot clé `as` introduit au paragraphe 3.4.

Valeurs des corps de structure :

v_s ::=	ϵ	Structure vide
	$\mid\ v_b\ ;\ v_s$	Liste de valeurs de composant

Valeurs des composants de structure :

v_b ::=	v_t	Valeur du langage de base
	$\mid$ `module` $x \triangleright x' = v_m$	Valeur de module

Valeurs du langage de base :

v_t ::=	`let` $x \triangleright x' = v_e$	Valeur d'expression
	$\mid$ `type` $x \triangleright x' = v_\tau$	Valeur de type

Valeurs des expressions :

v_e ::=	l_τ	Constante adresse-mémoire de type τ
	$\mid\ \underline{\mathrm{n}}$	Constante entière
	$\mid\ \lambda x.e$	λ-abstraction

Valeur des types :

v_τ ::=	`int`	Entiers
	$\mid\ v_\tau \to v_\tau$	Valeur de type flèche
	$\mid$ `ref`(v_τ)	Valeur de type référence

La relation d'ϵ-réduction, notée $\underset{\epsilon}{\longrightarrow}$ et présentée figure 1, est une sémantique en appel par valeur. Nous avons choisi un ordre d'évaluation de droite à gauche (règles B, C, D et K, L, M). En OCaml, l'ordre d'évaluation des arguments est non spécifié. Notre choix est donc correct vis-à-vis de la sémantique d'OCaml. En outre, en pratique, l'évaluation s'effectue de la droite vers la gauche. De plus, la transformation effectuée pour défonctoriser, présentée en 4.4, oblige à «déterminiser» le calcul. D'où le choix effectué. Cependant, si ce choix est pertinent pour les applications de foncteurs, il l'est moins pour les λ-applications. Notre transformation ne repose pas sur ces dernières et il aurait donc été plus rigoureux de leur donner une sémantique non déterministe. Cela aurait néanmoins alourdit notre étude.

Les règles F et G forcent la substitution, par sa valeur, du lieur du composant qui vient d'être évalué, dans la suite de la liste, avant l'évaluation de cette dernière. On assure ainsi qu'il n'y a plus d'identificateur en tête d'un chemin d'accès lorsqu'un terme est évalué.

Les règles N à T concernent les références. Un *état-mémoire* associe des adresses-mémoires à des valeurs d'expressions. Cependant, on doit être en mesure d'*allouer* une nouvelle adresse-mémoire afin d'y stocker une valeur. C'est la raison pour laquelle une *mémoire* adjoint à un état-mémoire un entier naturel indiquant quelles adresses-mémoires ont déjà été allouées. La règle O assigne une nouvelle adresse-mémoire à chaque nouvelle référence créée. De plus, lors d'une lecture (resp. écriture) d'une (resp. dans une) adresse-mémoire, il faut récupérer l'index de la mémoire correspondant à l'adresse lue (resp. écrite). Ceci est réalisé par les règles Q et T.

Les règles U, V et W concernent les chemins d'accès. Elles ne présentent pas de difficultés particulières, la règle F ayant préalablement assurée que la partie gauche d'un chemin d'accès est une valeur.

Proposition 1 (Ensemble des formes normales de $\underset{\epsilon}{\longrightarrow}$)
L'ensemble des formes normales de $\underset{\epsilon}{\longrightarrow}$ est exactement $\mathcal{V}$. Plus formellement :
soient $v \in \mathcal{MC}^{\mathcal{L}}_{SE}$ bien typé et $\mathcal{E}$ une mémoire.

$$v \in \mathcal{V} \iff \nexists < v' \mid \mathcal{E}' >.\ < v \mid \mathcal{E} > \underset{\epsilon}{\longrightarrow} < v' \mid \mathcal{E}' >$$

$$\frac{<s|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <s'|\mathcal{E}'>}{<\texttt{struct}\ s\ \texttt{end}|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <\texttt{struct}\ s'\ \texttt{end}|\mathcal{E}'>} \tag{A}$$

$$\frac{<m_2|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <m_2'|\mathcal{E}'>}{<m_1(m_2)|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <m_1(m_2')|\mathcal{E}'>} \tag{B}$$

$$\frac{<m_1|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <m_1'|\mathcal{E}'> \qquad v_2 \in \mathcal{V}}{<m_1(v_2)|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <m_1'(v_2)|\mathcal{E}'>} \tag{C}$$

$$\frac{v_2 \in \mathcal{V}}{<(\texttt{functor}(x : M)m_1)(v_2)|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <m_1\{x \leftarrow v_2\}|\mathcal{E}>} \tag{D}$$

$$\frac{<b|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <b'|\mathcal{E}'>}{<(b;s)|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <(b';s)|\mathcal{E}'>} \tag{E}$$

$$\frac{v \in \mathcal{V} \qquad \mathit{lieur}(v) \in s}{<(v;s)|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <(v; s\{\mathit{lieur}(v) \leftarrow \mathit{expr}(v)\})|\mathcal{E}>} \tag{F}$$

$$\frac{\begin{array}{c} v \in \mathcal{V} \qquad \mathit{lieur}(v) \notin s \\ <s|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <s'|\mathcal{E}'> \end{array}}{<(v;s)|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <(v;s')|\mathcal{E}'>} \tag{G}$$

$$\frac{<m|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <m'|\mathcal{E}'>}{<\texttt{module}\ x \rhd\ x' = m|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <\texttt{module}\ x \rhd\ x' = m'|\mathcal{E}'>} \tag{H}$$

$$\frac{<e|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <e'|\mathcal{E}'>}{<\texttt{let}\ x \rhd\ x' = e|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <\texttt{let}\ x \rhd\ x' = e'|\mathcal{E}'>} \tag{I}$$

$$\frac{<\tau|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <\tau'|\mathcal{E}'>}{<\texttt{type}\ x \rhd\ x' = \tau|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <\texttt{type}\ x \rhd\ x' = \tau'|\mathcal{E}'>} \tag{J}$$

$$\frac{<e_2|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <e_2'|\mathcal{E}'>}{<(e_1 e_2)|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <(e_1 e_2')|\mathcal{E}'>} \tag{K}$$

$$\frac{<e_1|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <e_1'|\mathcal{E}'> \ v_2 \in \mathcal{V}}{<(e_1 v_2)|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <(e_1' v_2)|\mathcal{E}'>} \tag{L}$$

$$\frac{v \in \mathcal{V}}{<(\lambda x.e v)|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <e\{x \leftarrow v\}|\mathcal{E}>} \tag{M}$$

$$\frac{<e|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <e'|\mathcal{E}'>}{<\texttt{ref}(e)|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <\texttt{ref}(e')|\mathcal{E}'>} \tag{N}$$

$$\frac{v \in \mathcal{V}}{<\texttt{ref}(v)|(\mathcal{E};i)> \underset{\epsilon}{\longrightarrow} <\mathit{nime}_{\mathcal{L}}(i+1)|(\mathcal{E} \cup \{\mathit{nime}_{\mathcal{L}}(i+1) \mapsto v\}; i+1)>} \tag{O}$$

$$\frac{<p|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <p'|\mathcal{E}'>}{<!p|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <!p'|\mathcal{E}'>} \tag{P}$$

$$\frac{l \in \mathcal{L}}{<!l|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <\mathcal{E}(\mathit{nime}_{\mathcal{L}}^{-1}(l))|\mathcal{E}>} \tag{Q}$$

$$\frac{<e|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <e'|\mathcal{E}'>}{<p \leftsquigarrow e|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <p \leftsquigarrow e'|\mathcal{E}'>} \tag{R}$$

$$\frac{v \in \mathcal{V} \qquad <p|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <p'|\mathcal{E}'>}{<p \leftsquigarrow v|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <p' \leftsquigarrow v|\mathcal{E}'>} \tag{S}$$

$$\frac{v \in \mathcal{V} \qquad l \in \mathcal{E}}{<l \leftsquigarrow v|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <\epsilon|\mathcal{E} \cup \{\mathit{nime}_{\mathcal{L}}^{-1}(l) \mapsto v\}>} \tag{T}$$

$$\frac{v = \texttt{struct}\ v_1; \texttt{module}\ x \rhd\ x' = v_x; v_2 \texttt{end} \qquad v \in \mathcal{V}}{<v.x|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <v_x|\mathcal{E}>} \tag{U}$$

$$\frac{v = \texttt{struct}\ v_1; \texttt{let}\ x \rhd\ x' = v_x; v_2 \texttt{end} \qquad v \in \mathcal{V}}{<v.x|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <v_x|\mathcal{E}>} \tag{V}$$

$$\frac{v = \texttt{struct}\ v_1; \texttt{type}\ x \rhd\ x' = v_x; v_2 \texttt{end} \qquad v \in \mathcal{V}}{<v.x|\mathcal{E}> \underset{\epsilon}{\longrightarrow} <v_x|\mathcal{E}>} \tag{W}$$

Fig. 1 – ϵ-réduction

PREUVE
Chaque sens de l'implication se traite par induction sur la taille de v. Le sens «$\Longleftarrow$» se traitant de plus par contraposée.

4.3. Equivalence entre termes de $\mathcal{MC}_{SE}$

Pour établir que le programme résultant de notre transformation (énoncée en 4.4) est équivalent au programme initial, nous allons utiliser la notion de *bisimulation* pour pouvoir raisonner par coinduction sur les structures du langage de modules.

Pour cela, nous définissons un opérateur monotone [ ] sur un ensemble de relations :

DÉFINITION 2 (OPÉRATEUR [ ])
Soit $\mathcal{R} = (\mathcal{R}_\sigma)_\sigma$. La relation $[\mathcal{R}]$ est définie à partir de $\mathcal{R}$ par cas sur σ par :

- $t\ [\mathcal{R}]_{\texttt{int}}\ t' \iff$

$$(\forall \mathcal{E}, \mathcal{E}', \underline{n} \in \mathbb{Z}.$$

$$< t|\mathcal{E} > \xrightarrow[\epsilon]{fn} < \underline{n}|\mathcal{E}' > \iff < t'|\mathcal{E} > \xrightarrow[\epsilon]{fn} < \underline{n}|\mathcal{E}' >)$$

- $t\ [\mathcal{R}]_{\tau \to \tau'}\ t' \iff \forall\ e : \tau.\ (t\ e)\ \mathcal{R}_{\tau'}\ (t'\ e)$
- $t\ [\mathcal{R}]_{\texttt{ref}(\tau)}\ t' \iff\ !t\ \mathcal{R}_\tau\ !t'$
- $\forall\ S.\ t\ [\mathcal{R}]_{\texttt{sig}\ \ S\ \texttt{end}}\ t' \iff$

  1.

$$(\forall \mathcal{E} \in \mathcal{M}.$$

$$(\exists < v_t \mid \mathcal{E}' >.\ < t \mid \mathcal{E} > \xrightarrow[\epsilon]{fn} < v_t \mid \mathcal{E}' >)$$

$$\Longrightarrow$$

$$(\exists < v'_t \mid \mathcal{E}'' >.\ < t' \mid \mathcal{E} > \xrightarrow[\epsilon]{fn} < v'_t \mid \mathcal{E}'' >))$$

$$\wedge$$

  2.

$$\forall x \in N(S), (\mathcal{E}, \mathcal{E}') \in \mathcal{M}^2.$$

$$\forall v.\ < t.x \mid \mathcal{E} > \xrightarrow[\epsilon]{fn} < v \mid \mathcal{E}' > \implies \exists v'.\ < t'.x \mid \mathcal{E} > \xrightarrow[\epsilon]{fn} < v' \mid \mathcal{E}' > \wedge\ v\ \mathcal{R}_{\sigma_x}\ v'$$

$$\wedge$$

$$\forall v'.\ < t'.x \mid \mathcal{E} > \xrightarrow[\epsilon]{fn} < v' \mid \mathcal{E}' > \implies \exists v.\ < t.x \mid \mathcal{E} > \xrightarrow[\epsilon]{fn} < v \mid \mathcal{E}' > \wedge\ v'\ \mathcal{R}_{\sigma_x}\ v$$

- $t\ [\mathcal{R}]_{\texttt{functor}(x:M_x)M}\ t' \iff \forall\ m : M_x.\ t(m)\ \mathcal{R}_M\ t'(m)$

Cette définition indique que deux termes sont en relation par $[\mathcal{R}]$ si :
- ce sont deux entiers se réduisant vers une même valeur ;
- ou bien ce sont deux fonctions ou deux foncteurs qui, appliqués au même argument, calculent deux valeurs en relation ;
- ou bien ce sont deux références référençant deux valeurs en relation ;
- ou bien ce sont deux structures dont l'évaluation de l'une se termine si l'évaluation de l'autre se termine et qui calculent deux valeurs en relation pour chaque terme qu'elles définissent.

DÉFINITION 3 (BISIMULATION)
Une bisimulation est un point postfixe de [ ], *i.e.* une relation $\mathcal{B}$ telle que $\mathcal{B} \subseteq [\mathcal{B}]$.

DÉFINITION 4 (BISIMILARITÉ)
Par monotonicité de $[\,]$, et d'après le théorème de Tarski-Knaster (voir, par exemple, [Pit97]), $[\,]$ possède une plus grande bisimulation, notée $=$ et appelée bisimilarité, telle que pour toute bisimulation $\mathcal{B}$, on a $\mathcal{B} \subseteq =$. De plus, $=$ est un point fixe de $[\,]$.

Une relation $\mathcal{R}$ peut être étendue en une relation $\mathcal{R}^o$ comportant des variables libres présentes dans un environnement de typage. On peut ainsi faire passer la notion de bisimulation au contexte. On est alors en mesure d'énoncer notre principe de coinduction sur les structures :

PROPOSITION 2 (PRINCIPE DE COINDUCTION SUR LES STRUCTURES)
Soient Γ un environnement de typage, S tel que $\Gamma \vdash S$ `modtype`, t et t' tels que $\Gamma \vdash t : \texttt{sig}\ S\ \texttt{end}$ et $\Gamma \vdash t' : \texttt{sig}\ S\ \texttt{end}$. Alors :

$$\Gamma \vdash t =_{\texttt{sig}\ S\ \texttt{end}} t' \iff \exists \mathcal{R}^o\ S\text{-bisimulation dans } \Gamma.\ \Gamma \vdash t\ \mathcal{R}^o\ t'$$

La notion de S-bisimulation utilisée par ce principe de coinduction est un cas particulier de bisimulation sur les structures.

Grâce à ce principe de coinduction, montrer que deux structures sont (contextuellement) équivalentes revient à exhiber une S-bisimulation les mettant en relation.

4.4. Suppression des applications de foncteurs

La transformation principale de l'algorithme énoncé en 3.7, qui correspond à la phase de réduction des applications de foncteurs, est définie formellement par la relation $\xrightarrow[T]{}$ ci-dessous :

DÉFINITION 5 (TRANSFORMATION)
Seul le cas intéressant concernant les applications de foncteurs est donné :
Si m correspond à un foncteur de la forme $\texttt{functor}(z_1 : M_1)\texttt{functor}(z_2 : M_2) \ldots \texttt{functor}(z_n : M_n)m'$, alors :

$$\begin{array}{c}
\texttt{struct}\ s_1\ ;\ \texttt{module}\ x \triangleright x' = m(m_1)(m_2)\ldots(m_n)\ ;\ s_2\ \texttt{end} \\
\xrightarrow[T]{} \\
\texttt{struct} \\
s_1\ ; \\
\texttt{module}\ x_n \triangleright x'_n = m_n\ ; \\
\ldots \\
\texttt{module}\ x_i \triangleright x'_i = m_i\ ; \\
\ldots \\
\texttt{module}\ x_1 \triangleright x'_1 = m_1\ ; \\
\texttt{module}\ x \triangleright x' = m'\{z_1 \leftarrow x'_1\}\ \{z_2 \leftarrow x'_2\}\ldots\{z_n \leftarrow x'_n\}\ ; \\
s_2 \\
\texttt{end}
\end{array}$$

Cette relation indique qu'il faut d'abord «sortir» chaque argument de l'application, en tenant compte de l'ordre d'évaluation de droite à gauche, et leur assigner à chacun un nouvel identificateur. Ensuite, on substitue dans le corps du foncteur chacun des paramètres formels par leurs arguments réels respectifs.

On a les résultats suivants :

THÉORÈME 3 (TERMINAISON DE $\xrightarrow[T]{}$)
$\xrightarrow[T]{}$ est fortement normalisante.

PREUVE
La preuve s'appuie sur le fait que l'évaluation des applications de foncteurs est fortement normalisante dans le langage $\mathcal{MC}$ (cf. [Cou98]).

PROPOSITION 4 (FORME DES T-FORMES NORMALES)
Toute structure de $\mathcal{MC}_{SE}$ en T-forme normale est une expression de module sans application de foncteur (sauf éventuellement à l'intérieur d'un foncteur).

PREUVE
La preuve est effectuée par contraposée et par cas suivant la forme de l'expression de module.

Le résultat qui suit est le résultat fondamental :

THÉORÈME 5 (PRÉSERVATION DE LA SÉMANTIQUE PAR $\xrightarrow[T]{}$)
Soient t et t' deux termes de $\mathcal{MC}_{SE}$ et σ tels que $t : \sigma$.

$$t \xrightarrow[T]{} t' \implies t =_\sigma t'$$

PREUVE
La preuve est effectuée en utilisant le principe de coinduction établi à la section 4.3.

5. Implantation et comparaison avec l'existant

Un défonctoriseur, nommé *ocamldefun*, implantant les transformations décrites dans cet article est en fin d'implantation et sera bientôt disponible.

Outils existants A l'heure actuelle, il existe déjà des outils - autres que ce défonctoriseur - défonctorisant un système de modules, qu'il s'agisse de celui d'OCaml ou de celui de SML.

Concernant OCaml, le seul outil permettant de défonctoriser est un «patch» [Mal] à appliquer au compilateur OCaml. Concernant SML, il existe trois défonctoriseurs différents. Un d'entre eux a été créé dans le cadre du projet BANE [BAN] qui produit des outils d'analyse statique de programmes. Les deux autres sont intégrés à deux compilateurs SML différents : ML-Kit [TBE$^+$] et MLton [CFJW].

	SML			Ocaml	
	BANE	MLton	ML-Kit	patch	ocamldefun
Justification formelle	$\oslash$	$\oslash$	$\surd$	$\oslash$	$\surd$
Ordre supérieur	$\oslash$	$\surd$	$\surd$	$\oslash$	$\surd$
Captures de variables	$\oslash$	$\surd$	$\surd$	$\surd$	$\surd$
Gestion des `open`	-[2]	-[2]	-[2]	$\surd$	$\surd$
Gestion des `include`	$\surd$	$\surd$	$\surd$	$\oslash$	$\surd$
Compilation incrémentale	$\oslash$	$\surd$	$\surd$	$\surd$	$\surd$
Production de codes sources	$\surd$	$\oslash$	$\oslash$	$\oslash$	$\surd$
Indépendant du langage de base	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\surd$

FIG. 2 – Tableau comparatif des différents défonctoriseurs

Caractéristiques des différents outils La figure 2 présente les caractéristiques des différents outils. Elle met en évidence que seul *ocamldefun* possède toutes les caractéristiques présentées.

Aucune des implantations réalisées n'est formellement justifiée, sauf ML-Kit. On ne peut donc être certain que la (les) transformation(s) réalisée(s) préserve(nt) la sémantique du programme. D'ailleurs, certaines d'entre elles ne traitent pas tous les problèmes susceptibles de se poser.

En outre, ML-Kit n'est pas source à source (il compile directement le programme défonctorisé) et ne permet donc pas l'utilisation d'outils d'analyse statique sur le programme généré.

Indépendance vis-à-vis du langage de base La défonctorisation est une opération concernant uniquement le langage de modules. Implanter le défonctoriseur indépendamment du langage de base permet de réutiliser facilement une grande partie des codes sources pour effectuer la défonctorisation d'un autre langage.

Cela va d'ailleurs dans le sens de la thèse soutenue dans [Ler00] par X. Leroy qui dit : «The module calculus makes few assumptions about the base language and should accommodate a variety of base languages».

6. Conclusion et perspectives

Nous avons établi une méthode effectuant la défonctorisation sur les programmes OCaml. Nous avons également montré que les transformations effectuées préservent la sémantique d'un programme, y compris d'un programme impératif : le programme initial est équivalent au programme obtenu après défonctorisation.

En outre, une implantation de ces transformations a été réalisée. Nous avons vu que cette implantation posséde un certain nombres d'avantages par rapport à celles déjà existantes.

En perspective, si l'on peut être confiant que le typage est bien préservé lors des phase 2 (passage de OCaml à $\mathcal{MC}$) et 3 (réduction des applications de foncteurs) de l'algorithme établi en 3.7, il reste encore à montrer que le typage est bien préservé lorsqu'on convertit un programme $\mathcal{MC}$ en un programme OCaml.

En outre, il est possible d'améliorer le passage de $\mathcal{MC}$ à OCaml en ne renommant que les variables en conflit pour améliorer à la fois la lisibilité et les performances du code produit.

Nous comptons réaliser des benchmarks pour mesurer le gain de performances. Un de nos objectifs est notamment de défonctoriser le logiciel C*i*ME [CMMU].

Remerciements

J'aimerais remercier Giuseppe Castagna et Judicaël Courant pour leurs commentaires avisés et leurs aides précieuses qu'ils m'ont fournis pour réaliser ce travail.

Références

[BAN] The Berkeley ANalysis Engine Group - BANE. *The BANE de-functorizer*. `http://www.cs.berkeley.edu/Research/Aiken/bane.html`.

[2] Il n'y a pas de `open` en SML.
Plus précisément, il existe une construction `open` mais sa sémantique correspond à celle du `include` d'OCaml.

[CFJW] H. Cejtin, M. Fluet, S. Jagannathan, and S. Weeks. *MLton User Guide.* `http://www.mlton.org`.

[CMMU] E. Contejean, C. Marché, B. Monate, and X. Urbain. *The CiME System.* L.R.I. `http://cime.lri.fr`.

[Cou97a] J. Courant. An applicative module calculus. In *Theory and Practice of Software Development 97*, volume 1214 of *Lecture Notes in Computer Science*, pages 622–636, Lille, France, avril 1997. Springer-Verlag.

[Cou97b] J. Courant. A module calculus for pure system type. In *Typed Lambda Calculi and Applications 97*, volume 1210 of *Lecture Notes in Computer Science*, pages 112–128, Nancy, France, avril 1997. Springer-Verlag.

[Cou98] J. Courant. *MC : un calcul de modules pour les systèmes de types purs.* Thèse de doctorat, Ecole Normale Supérieure de Lyon, 1998.

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st symposium Principles of Programming Languages*, pages 123–137. ACM Press, 1994.

[LDG+] X. Leroy, (D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon). *The Objective Caml System.* I.N.R.I.A. - Projet Cristal. http ://pauillac.inria.fr/ocaml.

[Ler00] X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3) :269–303, 2000.

[Mal] Malcy. Specialized functors. Message sur la Caml-list d'avril 2002. http ://caml.inria.fr/archives/200204/msg00405.html.

[Pit97] A.M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 241–298. Cambridge University Press, 1997.

[Que84] D. Mac Queen. Modules for standard ml. In *ACM Conference on LISP and Functional Programming*, pages 198–207, 1984.

[TBE+] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, and P. Sestoft. *Programming with Regions in the ML-Kit.* `http://www.it-c.dk/research/mlkit`.