Janvier 1997 – Journées Francophones des Langages Applicatifs – JFLA97

Ajout d'une conditionnelle dans un système de modules à la Caml

François Pessaux

Institut National de Recherche en Informatique et Automatique
(INRIA Rocquencourt) - Projet Cristal
78153 Le Chesnay CEDEX, France
francois.pessaux@inria.fr
http://pauillac.inria.fr/~pessaux/

1. Introduction

Les systèmes de modules actuels de SML [13, 16] et Objective Caml [10] se présentent comme un langage à part entière, plus ou moins disjoint du langage de base, et relativement restreint puisque ne comportant que les structures (modules de base), les foncteurs (fonctions de modules dans les modules) et l'application.

Tous les foncteurs sont polymorphes paramétriques en leurs arguments, au sens de la théorie de la paramétricité [17]. Autrement dit, l'unique corps du foncteur paramétrique se doit de fonctionner quelque soit la signature de ses paramètres effectifs.

Cette forme de polymorphisme peut se révéler insuffisante si l'on cherche à créer des foncteurs implantant des structures de données génériques pouvant être optimisées en fonction de certains types d'entrée (une structure d'ensemble générale sous forme d'arbre peut être améliorée en tableau de bits en cas d'ensemble d'entiers), tout en conservant une interface homogène.

De plus, ce travail semble être un premier pas nécessaire vers l'étude des systèmes de modules comme AXIOM [7] en propose, permettant en autre la discrimination sur le type des arguments d'un foncteur ou sur d'autres informations sémantiques comme l'associativité d'une composante de ce foncteur, en vue de les introduire dans le cadre des langages avec inférence de type comme Caml.

Dans le présent article, nous nous proposons d'étendre les systèmes à la Caml avec un mécanisme permettant de discriminer sur les types des arguments d'un foncteur, de la même manière qu'une fonction ML discrimine par filtrage sur les valeurs de ses arguments. Cette construction conditionnelle sur les types au niveau du langage de modules permet de définir des foncteurs polymorphes ad-hoc.

Au contraire de [2] qui s'appuie sur une surcharge avec résolution implicite, nous discriminons explicitement sur les types (l'ordre des cas est explicitememt donné par le programme) ce qui nous évite toute ambiguité.

De même que [2] peut se voir comme une application aux modules des travaux sur la surcharge dans les langages fonctionnels $F_{\leq}$ [12], Haskell [5], Alcool [15], notre système peut être considéré comme une application aux modules du polymorphisme extensionnel [18], des dynamiques [11] ou du `typecase` de Modula-3 [14].

La suite de cet article se présente de la façon suivante. Dans la section 2, nous faisons un bref rappel sur les notions de base des modules de ML. Dans la section 3, nous exposons en détail les motivations du présent travail. La syntaxe de notre construction conditionnelle est abordée en $4^{ème}$ section. La section 5 traitera de la sémantique dynamique de cette construction. Le typage sera abordé en section 6 et la $7^{ème}$ partie de ce document parlera de l'implantation ayant résulté de cette recherche. Un parallèle avec les travaux déjà existant sur le sujet sera ensuite réalisé en section 8.

2. Rappels sur les modules

Les systèmes de modules à la SML [13] et Objective Caml [10] sont très puissants. Ils autorisent la segmentation de programmes en vue d'en accroître la maintenance et la réutilisabilité.

Actuellement le système comporte deux formes d'expressions de modules : les *structures* (entités composées d'un regroupement de déclarations de types, de valeurs, d'exceptions et de sous-modules) :

```
module MyModule = struct
  type t = int
  val v = "Eat at Joe's"
  module MySubModule = struct
    type u = string
    end
  end
```

et les *foncteurs* que l'on pourrait assimiler à des fonctions des modules dans les modules :

```
module MyFunctor = functor (X:sig type t = int end) ->
  struct
```

```
  type u = X.t
  val x = true
  end
```

Le corps des foncteurs peut utiliser les composantes présentes dans ses modules paramètres, permettant ainsi la définition paramétrée d'expressions de modules, ainsi que l'extension de structures. La référence de composantes d'une structure est réalisée via la notation dite *pointée* (`X.t`), comme illustré dans le corps du foncteur ci-dessus.

Tout comme les expressions de Caml (le langage de base) sont typées, les expressions du langage de modules se doivent également de l'être. Le type d'une expression de module est appelé sa *"signature"*.

La signature d'une structure se présente comme une collection de spécifications de composantes de cette structure. La signature d'une structure ne se doit pas de contenir, comme nous le verrons plus tard, autant d'informations que la structure à laquelle elle se rapporte.

```
sig
  type t = int
  val v : string
  module Bar : sig
    type u = string
    end
  end
```

Comme nous le voyons sur cet exemple de signature (correspondant à la structure `MyModule` ci-dessus), la spécification d'une composante de type est elle-même, celle d'une composante de valeur est le type de cette valeur, et celle d'un module est la signature de ce module.

La signature du foncteur précédent se présente donc sous la forme suivante :

```
functor(X : sig type t = int end)
  sig
    type u = X.t
    val x : bool
  end
```

Nous avons dit précedemment qu'une signature pouvait contenir moins d'informations que la structure à laquelle elle se rapporte. C'est ce que nous appelerons la notion d'*abstraction*. Celle-ci peut être réalisée de deux façons : par oubli de composantes (la structure définit des composantes non spécifiées dans la signature) et par abstraction de type (la signature spécifie le type abstraitement alors que la structure l'implémente concrètement).

Par exemple, la signature suivante est encore valide pour la structure `MyModule` prise en exemple précédemment :

```
sig
  type t
  module Bar : sig
    type u = string
    end
  end
```

Nous dirons alors que la stucture *satisfait* la signature.

De cette possibilité pour une expression de module de posséder plusieurs signatures possibles, nous pouvons déduire une relation de sous-typage entre signatures et nous dirons qu'une signature S_1 est sous-type d'une signature S_2, noté $S_1 <: S_2$, *si tout module satisfaisant S_1 satisfait aussi S_2*. On pourra consulter [9] pour une définition formelle du sous-typage.

Néanmoins, une limitation de ce système actuel est que le code dans les modules retournés par un foncteur ne peut pas dépendre des types présents dans les paramètres de ce foncteur.

3. Intérêt de la conditionnelle

Par l'ajout de notre construction conditionnelle, nous cherchons la possibilité de fournir une implantation de modules en fonction de *types*. En particulier, nous aimerions que l'implantation du résultat d'un foncteur puisse dépendre de la structure des composantes de type présentes dans ses expressions de modules arguments.

Par extension triviale, nous pouvons envisager de discriminer sur n'importe quelle composante de type, pas simplement sur les paramètres d'un foncteur, ce qui inclut les déclarations ayant été faites en dehors de modules, mais présente alors peu d'intérêt.

Examinons l'utilité d'une telle construction. Imaginons que nous veuillons implanter des modules gérant des ensembles. Plutôt que programmer un module par type des éléments voulus pour nos ensembles, nous programmerions directement un foncteur prenant en argument un module implantant le type des éléments que nous voulons assembler et les opérations nécessaires sur ce type, et rendant directement un module d'ensemble d'éléments de ce type.

Au cours de l'utilisation de ce foncteur, plusieurs types d'éléments lui seront soumis, par exemples : entiers, flottants, booléens, etc.

Une représentation générale possible d'un ensemble est *l'arbre binaire*. Néanmoins, lorsque l'on a affaire à un ensemble d'entiers, il est une façon plus efficace de le représenter (en terme de compacité des données), bien connue dans le domaine des systèmes d'exploitation (table d'allocation de secteurs disque, table d'allocation de granules mémoire etc...) : le *tableau de bits*.

Nous voudrions donc utiliser un arbre pour représenter des ensembles en général, sauf quand il s'agit d'ensemble d'entiers. Ainsi, notre foncteur doit être capable de savoir, grâce à la conditionnelle, si le type des éléments demandé pour l'ensemble est `int` et le cas échéant implanter le type des ensembles comme un tableau de bits.

```
functor (Elt : sig type t end) ->
  typecase Elt.t oftype
     int => struct
              type set = bool array
              let empty = [| |]
              let member = ...
            end
    || _ => struct
              type set = Elt.t tree
              ...
            end
```

Sans la conditionnelle, pour implanter un tel mécanisme, le programmeur aurait dû écrire explicitement un foncteur pour le cas général et une structure pour le cas particulier des entiers. Plus simplement encore, le programmeur aurait pu seulement écrire le foncteur général, sacrifiant l'efficacité à la simplicité. Nous nous trouvons donc dans une optique d'optimisation qui prend plus d'attrait si l'on considère que nos ensembles doivent faire partie d'une bibliothèque. En effet, intégrer en une seule expression de module (notre foncteur conditionnel) la fabrication des ensembles permet d'obtenir une interface plus cohérente pour notre bibliothèque, étant donné que le cas particulier des entiers ne se manifeste pas par une structure isolée. Seul sera alors visible le foncteur de création d'ensembles et c'est lui qui se chargera de gérer l'optimisation possible des ensembles d'entiers.

Le champ d'application visé pour la conditionnelle est l'ensemble des programmes requérant des algorithmes pouvant être optimisés en fonction des structures de données leur étant soumises. Cette nécessité semble réelle dans des problèmes de calcul formel, Axiom [7] fournissant un mécanisme similaire. Par exemple, l'évaluation de polynômes peut être réalisée de façon plus efficace que par la méthode de Horner lorsqu'il s'agit de les appliquer à des nombres complexes [8]. Ce dernier exemple a d'ailleurs été implanté pour tester le prototype notre conditionnelle [3].

Notons que la structure de notre conditionnelle ne se limitera pas à un `if ... then ... else`, nous considèrerons directement une conditionnelle généralisée (du style `swith ... case ...` en C).

4. Syntaxe

Notre construction conditionnelle vient se rajouter aux expressions de modules déjà connues du langage [9] :

Expression de valeur :	
$e \ ::= v_i$	Identificateur de valeur
$\mid p.v$	Champ de valeur d'une structure
$\mid \dots$	Dépendant du langage de base
Expression de type :	
$\tau \ ::= t_i$	Identificateur de type
$\mid p.t$	Champ de type d'une structure
$\mid \texttt{int} \mid \tau \to \tau \mid \dots$	Dépendant du langage de base
Expression de module :	
$m \ ::= x_i$	Identificateur de module
$\mid p.x$	Champ de module d'une structure
$\mid \texttt{struct}\ s\ \texttt{end}$	Construction d'une structure
$\mid \texttt{functor}\ (x_i : M)\, m$	Foncteur
$\mid m_1(m_2)$	Application de foncteur
Type de module (signature) :	
$M ::= \texttt{sig}\ S\ \texttt{end}$	Type de structure
$\mid \texttt{functor}\ (x_i : M)\, M'$	Type de foncteur
Corps de structure :	
$s \ ::= \varepsilon \mid d; s$	
Composante de structure :	
$d \ ::= \texttt{val}\ v_i = e$	Définition de valeur
$\mid \texttt{type}\ t = \tau$	Définition de type
$\mid \texttt{module}\ x_i = m$	Définition de module
Corps de signature :	
$S \ ::= \emptyset \mid S_c; S$	
Composante de signature :	
$S_c ::= \texttt{val}\ v_i : \tau$	Déclaration de valeur
$\mid \texttt{type}\ t_i$	Déclaration de type abstrait
$\mid \texttt{type}\ t_i = \tau$	Déclaration de type manifeste
$\mid \texttt{module}\ x_i : M$	Déclaration de module

Chemins d'accès :

$$p \quad ::= x_i \mid p.x$$

sous la forme syntaxique suivante :

$$m ::= \dots \\ \quad \mid \texttt{typecase}\ \tau\ \texttt{oftype}\ \tau_1 \Rightarrow m_1 \mid\mid \dots \mid\mid \tau_i \Rightarrow m_i \mid\mid \_ \Rightarrow m$$

Il convient alors de s'intéresser à la forme des expressions t et t_i. Celles-ci représentent des expressions de type bien formées au sens Caml. Par exemple : `int list`, `'a set`, etc ... Néanmoins, certaines restrictions doivent être imposées, comme nous allons le voir.

4.1. Bonne formation des expressions de motif

Il est nécessaire d'imposer que les termes dénotant des variables de type dans les motifs existent déjà dans l'expression de type sur laquelle on discrimine. En effet, ne pas imposer une telle restriction obligerait à effectuer des renommages de variables pour réaliser le lien entre les termes dénotant les variables de type de l'expression discriminée et ceux de l'expression motif, posant le problème de renommer le cas suivant :

```
typecase ('a,'b) t oftype
  'c * 'd => ...
```

Deux substitutions sont alors possibles ['a$\leadsto$'c;'b$\leadsto$'d] ou ['a$\leadsto$'d;'b$\leadsto$'c], donnant lieu à deux définitions de type possibles :
`type ('a, 'b) t = 'a * 'b` ou `type ('a, 'b) t = 'b * 'a`.

Nous pouvons donc utiliser, comme motifs, des expressions de type bien formées au sens Caml, du moment que les variables de type présentes dans celle-ci sont aussi présentes dans l'expression sur laquelle on discrimine.

4.2. Bonne formation de l'expression discriminée

Considérons le cas suivant :

```
typecase int t oftype
  int list => ...
```

Du point de vue de la sémantique à l'exécution, le "test conditionnel" réussira si `int t` s'évalue en le même terme que `int list`. Néanmoins, la réussite à runtime de cette égalité peut se faire pour plusieurs raisons : soit `t` est défini comme `type 'a t = 'a list`, soit `t` est défini comme `type 'a t = int list`.

En anticipant un peu sur la suite de ce travail, nous devons établir un lien *("manifestisation")* entre l'expression de type sur laquelle on discrimine et chacune des expressions de motif, afin de refléter le fait que l'expression de module de chaque branche est succeptible d'être résultat de la conditionnelle s'il y a "égalité" entre les deux expressions de type. On devra donc considérer, au sein d'une branche, que le type sur lequel on discrimine est défini égal au motif de cette branche.

Deux façons de manifestiser *(rendre manifeste)* `t`, c'est à dire de modifier sa définition sont alors possibles : `type 'a t = int list` ou `type 'a t = 'a list`.

Techniquement, la manifestisation en présence de types paramétrés et sans restriction sur le type filtré se réduit à un problème d'unification du second ordre, problème connu pour ne pas avoir de solution principale.

Afin d'éviter ceci, les expressions de type sur lesquelles on discrimine sont donc les expressions de type bien formées au sens Caml, pour lesquelles les constructeurs ne sont appliqués qu'à des termes dénotant des variables de type ('a, 'b, ...) toutes distinctes. Autrement dit, on discrimine sur des constructeurs apportant leur arité.

5. Sémantique dynamique

Nous désirons que le résultat d'un foncteur puisse dépendre d'une discrimination sur des expressions de type. Les applications de foncteur étant réalisées dynamiquement en Caml, à l'initialisation du programme, nous allons devoir discriminer *à l'exécution* sur ces expressions de type. Il convient donc maintenant de conserver trace des types à l'exécution.

5.1. Représentation des structures

Les définitions de type se trouvant à l'intérieur des structures (composantes de type), il est nécessaire de s'intéresser à la nouvelle forme des structures.

Avant l'ajout de notre construction conditionnelle, les structures pouvaient être vues comme des *enregistrements* dans lesquels on avait supprimé les champs correspondant aux définitions de types, ces définitions ne servant alors que durant la phase de typage. Devant maintenant conserver ces types, nous verrons dorénavant les structures comme des *enregistrements* comportant des champs correspondant aux composantes de définition de type. La sémantique donnée ci-dessous s'inscrit dans la continuité de celle que l'on peut trouver dans [9].

Étant donné que les ensembles de valeurs du langage et de valeurs du langage de type sont disjoints, nous pouvons les fusionner en un seul et même environnement que nous noterons ρ dans les sections qui vont suivre.

$$
\begin{array}{rcl}
[\![\mathrm{X}]\!]_\rho & = & \rho(X) \\
[\![\mathrm{p.X}]\!]_\rho & = & [\![\mathrm{p}]\!]_\rho.\mathrm{X} \\
[\![\texttt{struct}\ d_1;\ d_2;\ \ldots;\ d_n\ \texttt{end}]\!] & = & \{b_1; b_2; \ldots; b_n\} \text{ avec} \\
 & & b_1 = [\![\ d_1\ ]\!]_\rho \\
 & & b_2 = [\![\ d_2]\!]_{\ \rho\oplus b_1} \\
 & & \ldots \\
 & & b_n = [\![\ d_n]\!]_{\ \rho\oplus b_1\oplus\ldots\oplus b_{n-1}} \\
[\![\texttt{functor}\ (\mathrm{X} : \mathrm{M}) \to \mathrm{m}]\!]_\rho & = & \lambda d.\ [\![\mathrm{m}]\!]_{\ \rho\oplus\{X\mapsto d\}} \\
[\![\mathrm{m}(\mathrm{m'})]\!]_\rho & = & [\![\mathrm{m}]\!]_\rho\ ([\![\mathrm{m'}]\!]_\rho) \\
 & & \\
[\![\texttt{let}\ \mathrm{v} = \mathrm{a}]\!]_\rho & = & (\mathrm{v} = [\![\mathrm{a}]\!]_\rho) \\
[\![\texttt{module}\ \mathrm{X} = \mathrm{m}]\!]_\rho & = & (\mathrm{X} = [\![\mathrm{m}]\!]_\rho) \\
[\![\texttt{type}\ \alpha_i\ \mathrm{t} = \tau\ ]\!]_\rho & = & (\mathrm{t} = \lambda\vec{x_i}.\ [\![\tau]\!]_\rho\ [\vec{\alpha_i} \leftarrow \vec{x_i}])
\end{array}
$$

5.2. Représentation des types

Puisque nous ajoutons des champs pour les définitions de type dans nos enregistrements, il convient de savoir quelle sorte de valeur nous devons y mettre. Autrement dit, il faut savoir comment seront interprétées les expressions de type.

La transformation d'expressions de type en "valeurs de type" est réalisée par une interprétation de ces expressions dans le domaine constitué : des termes finis du 1^{er} ordre pour les expressions de type et des fonctions de ces termes dans eux-mêmes, pour les définitions de type. Dans ce qui suit, nous noterons les termes sous la forme : *constructeur* $\langle$*arguments*$\rangle$. De plus, nous noterons $\overline{t}$ la représentation d'un type primitif t.

$$
\begin{array}{lcc}
[\![\alpha]\!]_\rho & = & \overline{\alpha} \\
[\![path\ \tau_1 \ldots \tau_n]\!]_\rho & = & \overline{path}\ \langle\ [\![\tau_1]\!]_\rho\ \ldots [\![\tau_n]\!]_\rho \rangle \\
 & & \text{si } path \text{ est un type de base ou abstrait} \\
[\![path\ \tau_1 \ldots \tau_n]\!]_\rho & = & \rho(path)\ ([\![\tau_1]\!]_\rho) \ldots ([\![\tau_n]\!]_\rho) \\
 & & \text{si } path \text{ est un type manifeste}
\end{array}
$$

La “valeur” d’une expression de type sera donc un terme représentant la structure de cette expression une fois normalisée modulo les équivalences de type présentes dans l’environnement (induites par les définitions de type).

La “valeur” d’une définition de type sera une fonction prenant autant d’arguments que l’arité du type défini, et rendant *le terme représentant l’application* du constructeur ainsi défini aux arguments effectifs qui seront fournis à cette fonction.

Exemple : Considérons les deux définitions de type suivantes (en considérant les constructeurs $\star$ et *list* comme primitifs) : `type 'a t = 'a * 'a` et `type ('a, 'b) u = ('a -> 'b) list`.

Ces définitions s’évalueront respectivement en les fonctions : $\lambda x_1 \,.\, \overline{\star}\,\langle x_1 ; x_1\rangle$ et $\lambda x_1 x_2 \,.\, \overline{list}\langle \overline{\to}\,\langle x_1 ; x_2\rangle\rangle$

Considérons maintenant l’expression : `(int t, bool) u`. Celle ci va être représentée par le terme construit par l’application de la fonction représentant `t` à $\overline{int}$ (représentation du type *int* alors considéré comme un type primitif), ce résultat lui-même appliqué, avec $\overline{bool}$, à la fonction représentant `u`, nous donnant : $\overline{list}\;\langle \overline{\to}\;\langle \overline{\star}\;\langle \overline{int};\overline{int}\rangle;\overline{bool}\rangle\rangle$

5.3. Sémantique de la conditionnelle

Notre construction conditionnelle va fonctionner par égalité sur les représentations de type. C’est à dire que l’expression de module résultat d’une conditionnelle sera la première pour laquelle il y a égalité structurelle, que nous noterons $\equiv$, entre la représentation de l’expression sur laquelle on discrimine et le motif.

$$[\![\texttt{typecase}\ t\ \texttt{oftype}\ t_i \Rightarrow m_i\ \ ||\ \ \_ \Rightarrow m\ ]\!]_\rho = \begin{array}{l} \text{if } [\![t]\!]_\rho \equiv [\![t_1]\!]_\rho \text{ then } [\![m_1]\!]_\rho \\ \text{else if } \ldots \\ \text{else if } [\![t]\!]_\rho \equiv [\![t_n]\!]_\rho \text{ then } [\![m_n]\!]_\rho \\ \text{else } [\![m]\!]_\rho \end{array}$$

Notons que nous ne serons confrontés qu’à *l’égalité entre termes finis du premier ordre*. En effet, nous n’avons jamais à affronter l’égalité de deux fonctions. Ceci pourrait *a priori* se produire si l’on cherche à déterminer si un constructeur de type `t` est égal à `list`, par exemple. Par définition de la bonne formation des expressions de type (en particulier qu’elles doivent être bien formées au sens Caml), cette opération est interdite. Pour effectuer cette opération, qui sémantiquement revient à comparer deux constructeurs paramétrés, nous devrons comparer leur application à des mêmes variables formelles, c'est à dire vérifier si leur application à des termes dénotant des mêmes variables de type (comme le terme représentant une variable `'a`), sont structurellement égales.

Exemple : soient les définitions de type

`type 'a t = 'a * (bool list)` et `type 'b u = int * ('b list)`. On a bien : `int t` $\equiv$ `bool u` et `'a t` $\equiv$ `'a t` mais `'a t` $\not\equiv$ `'b t`.

Par choix, notre construction conditionnelle ne va pas respecter l'abstraction de type à l'exécution. Autrement dit, si une composante de type a été abstraite, alors nous pourrons quand même discriminer sur sa structure à l'exécution, bien que cette structure ait réellement été perdue lors de la phase de typage.

On peut également envisager des sémantiques dynamiques respectant l'abstraction avec génération à l'exécution de nouveaux identificateurs de type lorsqu'on abstrait un type par une contrainte de signature.

6. Le typage de la conditionnelle

Notre construction conditionnelle se rajoutant aux expressions de modules déjà existantes, il convient de la typer. Nous avons vu qu'elle était composée de plusieurs branches représentant chacune une expression de modules. Nous allons donc chercher une signature unique pour cet ensemble de branches : une signature commune.

$$\frac{\Gamma \vdash t \text{ bien formé} \quad \Gamma \vdash t_i \text{ bien formés} \quad \Gamma \oplus t = t_i \vdash m_i : S \quad \Gamma \vdash m : S}{\Gamma \vdash \texttt{typecase}\ t\ \texttt{oftype}\ t_i \Rightarrow m_i \ \|\ \_ \Rightarrow m : S}$$

Règle de typage de la conditionnelle

Il est ainsi toujours possible de trouver, via la relation de sous-typage, une signature commune à plusieurs signatures de structure (pour ces dernières, `sig end` en est une preuve), nous discuterons des foncteurs ultérieurement. Néanmoins, il conviendrait d'obtenir une signature donnant le plus d'informations possible sur les signatures qu'elle récapitule. Particulièrement, l'obtention de `sig end` comme signature commune ne nous serait d'aucune utilité étant donné que toutes ses composantes étant cachées, le module résultat de la conditionnelle nous serait inutilisable. Nous cherchons donc à obtenir la signature commune minimale. Nous reviendrons sur ce point ultérieurement.

La première démarche à réaliser est, comme nous venons de le citer, le typage des différentes expressions de module se trouvant à droite des branches de notre expression.

6.1. Le typage des différentes branches

Celui-ci va être réalisé par l'intermédiaire de l'algorithme d'inférence déjà existant dans Caml. Par exemple, afin de typer la construction suivante :

```
module FOO =
  typecase t oftype
       int => struct type u = int end
    || string => struct type u = string end
```

nous devons d'abord déterminer les signatures de chacunes des branches, ce qui nous donnerait intuitivement :

```
sig type u = int end
sig type u = string end
```

Si nous essayons maintenant, de synthétiser intuitivement, une signature commune à ces deux branches, nous obtenons `sig type u end`. En effet, les deux définitions du type `u` sont différentes et donc incompatibles et les règles de sous-typage entre signatures nous demandent alors de rendre ce type `u` abstrait.

Néanmoins un examen approfondi de ces deux branches révèle la connaissance d'une équivalence de type supplémentaire.

En effet, tout comme la construction `match x with 3 -> ...` de Caml nous garantit que la variable `x` sera égale à la valeur 3 dans l'expression à droite du filtre, notre conditionnelle se doit de lier l'expression de type `t` à `int` dans la première branche, et à `string` dans la seconde branche. Ainsi, dans l'environnement de typage de la première branche devra se trouver la définition du type de `t` modifiée en : `type t = int` et dans celui de la seconde branche en : `type t = string`.

Grâce à cela, nous pourrons déduire que dans chaque cas, le type `u` n'est pas seulement égal à `int`, respectivement `string`, mais aussi à `t`. Une signature plus précise pourra alors être déduite pour notre conditionnelle, à savoir :

```
sig type u = t end
```

Le typage de chaque branche de la conditionnelle doit donc être effectué sous l'hypothèse supplémentaire que le type sur lequel on discrimine est défini égal au motif de la branche.

Cette modification (que nous nommerons *manifestisation*) est une obligation si l'on espère tirer quelque puissance de notre système de conditionnelle. En effet, s'il était impossible de bénéficier de ce lien entre les deux expressions de type, il serait impossible de faire dépendre l'implémentation du module de l'implémentation du type sur lequel on discrimine. De ce fait, le code de chaque implémentation du module ne pourrait jamais être spécifique à la structure de type découverte grâce à la conditionnelle. Dans l'exemple ci-dessus, ceci était la seule façon de conserver notre composante de valeur `x`.

Nous savons maintenant comment typer les différentes branches de notre construction conditionnelle. Il convient alors de s'occuper de la synthèse proprement dite d'une signature commune à plusieurs signatures de structures.

Nous ne détaillerons ici que ce qui concerne les structures. Les foncteurs seront rapidement mentionnés ultérieurement.

6.2. Synthèse de signature commune

L'idée de cette phase est d'appliquer une forme d'anti-unification [6] sur les composantes communes des signatures. Nous pouvons en effet oublier directement les composantes non présentes dans toutes les signatures puisque la relation de sous-typage nous les fera oublier dans la super-signature que nous recherchons.

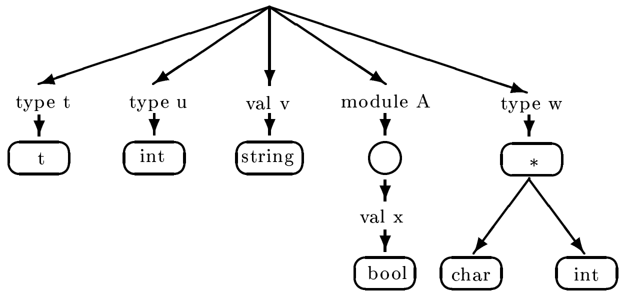
Pour effectuer cette anti-unification, il est nécessaire, au moins pour le principe théorique, de transformer les signatures obtenues après typage (appelées *signatures syntaxiques*) en une forme plus aisée à anti-unifier. Cette forme, appelée *signature sémantique* [13] sera matérialisée par un arbre, dont les branches représentent les composantes de la signature et les sous-arbres, les composantes de module de cette même signature.

Chaque feuille de l'arbre sera alors étiquetée par le représentant canonique de la classe d'équivalence des types à laquelle elle appartient.

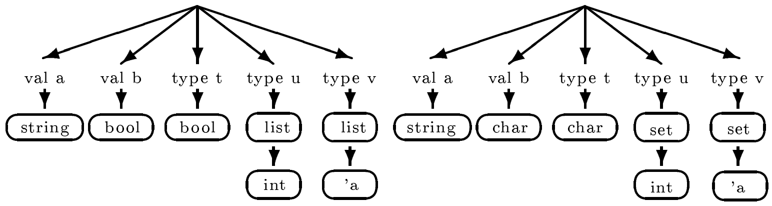
Ainsi, la signature suivante :

```
sig
  type t
  type u = int
  val v : string
  module A : sig val x : bool end
  type w = char * u
end
```

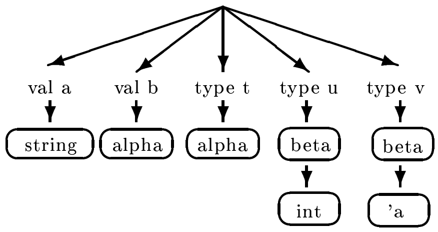
serait représentée par l'arbre (signature sémantique) :



Une fois les signatures syntaxiques transformées en signatures sémantiques, nous pouvons aborder l'anti-unification. Considérons-le sur l'exemple suivant dans lequel nous partons directement des signatures sémantiques :



Par anti-unification conventionnelle, nous obtenons la signature sémantique résultat suivante (où les lettres grecques désignent les *variables d'anti-unification* et les 'a, 'b, …, *les termes dénotant des variables de type ML*) :



Nous devons maintenant transformer inversement cette signature sémantique en signature acceptable par Caml, c'est à dire en signature syntaxique. A des fins explicatives, effectuons cette transformation de manière informelle.

La composante de valeur `a` est étiquetée du type `string`. De ce fait, cette composante peut être re-transcrite directement dans la signature syntaxique sous la forme `val a : string`.

Vient ensuite la composante de valeur `b` dont le type est indéterminé car nommé par une variable d'anti-unification. Il semblerait que l'on ne puisse pas garder cette composante puisque son type est inconnu. Mais en regardant un peu plus loin, on trouve une composante de type `t` de même étiquette. Donc nous pouvons considérer que celle-ci définit cette variable d'anti-unification. Nous posons alors la composante de type `t` abstraite et considérons que la valeur `a` est de ce type. Nous obtenons alors les deux composantes : `type t`, et `val b : t`.

Une démarche similaire se produit avec les deux dernières composantes. En effet, la composante de type `u` est étiquetée par `int` β, sans qu'à ce niveau, β ne soit encore définie. Néanmoins, l'examen de la composante `v` montre que `u` est une instance de `t` avec `'a` remplacé par `int`. Nous pouvons alors définir en premier la composante de type `v` abstraite et `u` comme étant `int v`. Nous obtenons alors bien la signature escomptée :

```
sig
  val a : string
  type t
  val b : t
  type 'a v
  type u = int v
end
```

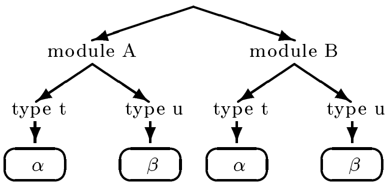
La méthode décrite ici semble bien convenir à notre problème initial. En effet, si l'on travaille sans types paramétrés, nous restons dans les termes finis du premier ordre, pour lesquels nous savons qu'il existe toujours un anti-unificateur principal. Qui plus est, nous bénéficions des propriétés d'associativité de l'opération d'anti-unification ce qui est appréciable pour la généralisation de la synthèse à plus de 2 signatures.

Néanmoins, dans le cas qui nous intéresse, il convient d'avoir un jugement plus critique. En effet, avec ML comme langage de base, nous travaillons avec des types simplement paramétrés (donc des termes du second ordre). Or l'existence d'un anti-unificateur principal sur les termes du second ordre n'est plus assurée [6]. De ce fait, il n'existe pas de super-signature commune *la plus* précise, d'où pas de solution unique au problème d'inférer une signature pour l'expression `typecase`.

Par ailleurs, la traduction de signature sémantique en signature syntaxique que nous avons effectuée informellement se révèle être une opération complexe. En effet, nous avons vu qu'il faut parfois inverser l'ordre des composantes pour déterminer les dépendances qui peuvent nommer certaines variables d'anti-unification. En l'absence de caractéristiques qui pourraient réduire ces choix, il faut envisager un parcours combinatoire avec "essai-erreur" ce qui nous garantit d'ores et déjà une forte complexité de l'opération.

De plus, certaines façons de parcourir les signatures sémantiques peuvent aboutir à des signatures mal formées pour Caml Special Light (qui interdit la récursivité dans ses signatures syntaxiques).

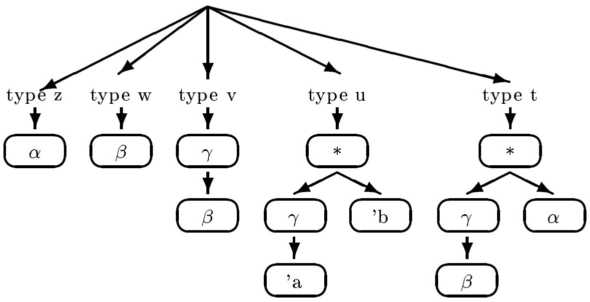
Considérons la signature sémantique suivante :



Si nous décidons de nommer le conflit dénoté par la variable α par `A.t` et celui dénoté par la variable β par `B.u`, nous obtenons une signature récursive, donc inacceptable.

Si par contre, nous avions nommé β par `A.u`, la signature aurait été correcte.

Pour clore les inconvénients de cette anti-unification, sachons qu'il existe des cas pour lesquels plusieurs parcours différents donneront plusieurs signatures incomparables entre elles, d'où encore une fois l'impossibilité de déterminer une seule et unique signature minimale. Cherchons à relire la signature sémantique suivante :



Il apparaît alors que nous pouvons extraire deux signatures sémantiques :

```
sig
  type z
```

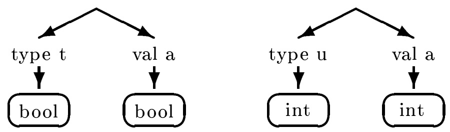
```
  type w
  type v
  type ('a,'b) u
  type t = v * z    OU    type t = (w,z) u
end
```

7. Implantation, approximation

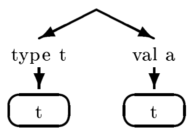
Malgré les problèmes majeurs que l'anti-unification peut poser, ce travail a été matérialisé par un prototype de langage bénéficiant de cette construction conditionnelle [3]. Pour y parvenir, nous proposons de remplacer l'anti-unification conventionnelle par une approximation évitant l'apparition de variables d'anti-unification (donc une relecture trop complexe) et l'anti-unification du second ordre.

Bien entendu, la contrepartie de cette méthode est qu'il existera des cas où, si plusieurs signatures minimales possibles existent, une seule sera fournie, et où même s'il en existe une seule, alors on ne sera pas forcément à même de la trouver.

Dans une telle optique, l'idée est de ne plus nommer les conflits d'anti-unification par des variables, mais directement par le nom de la composante de type qui pose ce conflit. Considérons en exemple les deux signatures suivantes à anti-unifier :



En trouvant le conflit (*bool* $\leftrightarrow$ *int*), lors du traitement des composantes de type `t`, nous le considèrerons comme résolu par `t` dans le traitement des composantes à venir. Ainsi, lorsque nous rencontrerons les deux définitions de la composante de valeur `a`, le conflit (*bool* $\leftrightarrow$ *int*) se posera à nouveau, mais avec `t` comme solution et nous obtiendrons la signature sémantique commune suivante :



qui peut facilement être réécrite en la signature syntaxique :

```
sig
  type t
  val a : t
end
```

Ceci nous évite ainsi l'apparition de variables d'anti-unification.

Pour éviter les problèmes dus au second ordre, nous évitons de descendre dans la structure des expressions de type. Aussi, considérons-nous des conflits globaux. Si nous trouvons deux définitions d'une composante sous la forme `int list` et `char set`, nous ne mémoriserons pas les conflits ($int \leftrightarrow char$) et ($'a\ list \leftrightarrow' a\ set$), mais directement ($int\ list \leftrightarrow char\ set$).

Ces restrictions impliquent bien évidemment une perte de puissance par rapport au modèle original. Particulièrement, *les résultats obtenus dépendent fortement de l'ordre de traitement des composantes.* En effet, nous nommons les conflits d'anti-unification par des identificateurs de types. De ce fait si le conflit est rencontré pour la première fois sur une composante de valeur, il n'y a aucun moyen de le nommer pour la suite. Si nous avions considéré l'exemple précédent en inversant l'ordre des composantes (c'est à dire en mettant la composante de valeur en premier), alors la première rencontre du conflit ($int \leftrightarrow bool$) aurait été sur la valeur `a` et donc aucun nommage n'aurait pu être possible. De ce fait, nous aurions seulement obtenu la signature `sig type t end`.

Il semble donc nécessaire de réorganiser les composantes avant anti-unification. Malheureusement, le "bon ordre" de traitement n'existe pas forcément de façon unique. Le dernier exemple de la section 6.2 en est la preuve.

Une approximation simple consistant à regrouper les composantes de type et de sous-module avant les composantes de valeur au sein d'un module semble donner des résultats utilisables dans des cas d'utilisation courante. Néanmoins, ceci ne suffit pas toujours, car il peut y avoir dépendance entre des définitions de type ou avec des définitions se trouvant à l'intérieur de sous-modules.

Notons que pour procéder ainsi, nous n'avons plus besoin pratiquement de transformer les signatures syntaxiques en signatures sémantiques. Ceci est d'un grand intérêt car, l'implémentation d'Objective Caml effectuant son typage des expressions de modules en s'appuyant directement sur les signatures syntaxiques, cela nous évite de devoir dupliquer toute une portion de l'algorithme d'inférence pour le faire fonctionner sur des signatures sémantiques.

8. Travaux connexes

La notion de conditionnelles dans les modules, ou de spécialisation de foncteurs a déjà été étudiée sous différents aspects.

Le premier d'entre eux concerne les langages comportant les modules comme valeurs de première classe, comme Quest [1] ou le calcul de module de R. Harper et M. Lillibridge [4]. Tout comme notre système, ces langages autorisent des `if ... then ... else ...` dont les deux branches sont des expressions de modules. Notre travail se différencie de Quest en ce sens que ce dernier propose en effet une conditionnelle binaire, mais ne recherche pas de signature commune aux deux branches. Il choisit arbitrairement une des branches et vérifie que l'autre est bien sous-type de la première.

D'un autre côté, le calcul de modules de Harper et Lillibridge n'a pas non plus à considérer cette synthèse étant donné qu'il ne comporte pas d'inférence de type, la vérification se trouvant elle-même indécidable. De plus, la forme de conditionnelle de ce langage diffère un peu de la nôtre en ce sens qu'il s'agit de faire dépendre une expression de modules de valeurs et non de types.

Le second lien à faire concerne le travail de G. Castagna et M.V. Aponte [2], dans lequel il est question de surcharge et de liaison tardive pour les foncteurs.

Dans ce travail, la formation des branches des foncteurs n'est pas libre. Un foncteur surchargé peut être vu comme un foncteur à plusieurs branches, dont la signature est l'union des signatures de chaque branche : $\texttt{functor}\ (x : S_1)S'_1 \cup \ldots \cup \texttt{functor}\ (x : S_n)S'_n$. Afin de garantir la cohérence de la résolution automatique de la surcharge, la première contrainte de bonne formation impose que si $S_i <: S_j$ alors $S'_i <: S'_j$. De plus, dynamiquement, lors d'une application, s'il n'existe aucune branche possédant une signature filtrant exactement celle de l'argument, la branche choisie est la plus petite (dans l'ordre du sous-typage) entre celles filtrant la signature dudit argument. Ceci impose alors l'existence d'une plus petite signature.

La modification de la forme des signatures de foncteurs de façon à refléter la surcharge, peut s'avérer une gêne dans le cas où l'on cherche à fabriquer des bibliothèques (interfaces complexes et variables selon le nombre de branches) ou lorque l'on cherche à rajouter un cas particulier (une branche) à un foncteur. Dans ce dernier cas, il faut alors changer l'interface de la bibliothèque et re-compiler tous les programmes utilisant ce module.

Le fait de changer la forme des signatures évite aussi la nécessité de recherche d'une signature commune à toutes signatures résultats des branches.

9. Conclusion

L'extension du système de modules de Caml Special Light ainsi proposée se classe donc dans les apports qualitatifs et non quantitatifs. Elle ne permettra pas de créer de programmes que l'on ne pouvait écrire avant, mais elle permettra d'en écrire certains plus clairement, en particulier dans l'optique de bibliothèques efficaces.

On peut toutefois se demander s'il est bien nécessaire de conserver les types à l'exécution du programme, comme c'est le cas actuellement, ce qui implique de profondes modifications du noyau interne du langage quant à la compilation et à l'interprétation. Il semble en effet possible d'effectuer les calculs de discrimination, avant l'exécution du programme puisque les expressions de type sur lesquelles on discrimine peuvent *a priori* être déterminées statiquement (elles sont présentes explicitement dans le source du programme).

Le seul obstacle à cette élimination des types est la notion d'abstraction de type. En effet, une fois qu'une composante de type a été abstraite, le typeur oublie sa structure. Il serait donc nécessaire d'effectuer un typage à deux niveaux, l'un respectant réellement l'abstraction des types pour les anciennes expressions du langage de modules, l'autre conservant la structure des types malgré l'abstraction pour notre expression conditionnelle. Le résultat d'une telle manipulation serait en effet une possibilité de déterminer statiquement à link-time (pas plus tôt du fait de la compilation séparée) quelle branche serait prise dans une expression de module conditionnelle, mais au prix d'un dédoublement de portions du système de typage.

Notons qu'au cours de ce travail, la synthèse de signature commune à des signatures de foncteurs a été pris en compte de façon simplifiée, c'est à dire en contraignant toutes les signatures de branches représentant des foncteurs à posséder des signatures d'arguments identiques. Du fait de la contravariance dans le sous-typage des foncteurs, il aurait fallu synthétiser une signature répondant au problème inverse de la synthèse d'une signature commune minimale.

Remerciements

Je tiens à remercier Xavier Leroy et François Rouaix pour leur support durant mon stage de DEA, suite auquel j'ai pu réaliser le travail donnant lieu au présent papier, ainsi que tout au cours de la réalisation de ce document.

Je n'oublierai pas non plus Thérèse Hardin qui a beaucoup contribué indirectement à ce travail.

Références

[1] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal description of programming concepts*, pages 431–507. Springer-Verlag, 1989.

[2] Maria-Virginia Aponte et Giuseppe Castagna. Programmation modulaire avec surcharge et liaison tardive. In *Actes des Journées Francophones des Langages Applicatifs*. INRIA, 1996.

[3] François Pessaux. *Ajout de conditionnelles dans un système de modules à la Caml Special Light*, 1996. Rapport de D.E.A Université Paris VI - INRIA Rocquencourt disponible sur le Web, `http://pauillac.inria.fr/~pessaux/dea96.dvi`.

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

[5] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5), 1992.

[6] Gérard Huet. *Résolution d'équations dans les langages d'ordre 1,2,...,ω*. Thèse de doctorat, Université Paris 7, 1976.

[7] Sutor R.S. Jenks R.D. *AXIOM - The scientific Computation System*. The Numerical Algorithms Group Limited - Oxford, UK, 1992.

[8] Donald E. Knuth. *Seminumerical algorithms - The art of computer programming - Vol 2, seconde édition pp. 467-469.* Addison Wesley, 1966.

[9] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st symposium Principles of Programming Languages*, pages 109–122. ACM Press, 1994.

[10] Xavier Leroy. The Objective Caml system. Logiciel et documentation disponibles sur le Web, `http://pauillac.inria.fr/ocaml/`, 1996.

[11] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[12] Giuseppe Longo, Giuseppe Castagna, and Giorgio Ghelli. A calculus for overloaded functions with subtyping. In *Information and Computation, vol. 117, n°1, pp. 115-135.* ACM Press, 1995.

[13] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML.* The MIT Press, 1990.

[14] Greg Nelson, Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, and Bill Kalsow. Modula-3 report (revised). Rapport de recherche 52, DEC Systems Research Center, 1989.

[15] François Rouaix. *ALCOOL-90: Typage de la surcharge dans un langage fonctionnel.* Thèse de doctorat, Université Paris 7, 1990.

[16] Mads Tofte. Four lectures on standard ML. Rapport de recherche ECS-LFCS-89-73, University of Edinburgh, 1989.

[17] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture 1989.* ACM Press, 1989.

[18] Pierre Weis and Catherine Dubois et François Rouaix. Extentional polymorphism. In *22nd symposium Principles of Programming Languages* ACM Press, 1995.