Ocaml-templates, méta-programmation à partir des types

François Maurel

Preuves, Programmes et Systèmes
Université Denis Diderot & CNRS
Case 7014, 2 place Jussieu
75251 Paris Cedex 05
`Francois.Maurel@pps.jussieu.fr`

Résumé

Le système de type du langage OBJECTIVE CAML interdit le paramétrage du type d'une fonction quelconque par un type explicite ou par la valeur d'un argument. Ainsi, les fonctions polymorphes définies par l'utilisateur ne peuvent observer précisément leurs arguments sans sortir du cadre typé. Par exemple, pour typer la fonction `Printf.printf`, il a fallu intégrer un cas spécifique au système d'inférence du compilateur. Cela empêche typiquement l'écriture d'une fonction `print` générique ou une fonction de sérialisation bien typée (le module `Marshal` qui s'occupe de la sérialisation ne permet pas de vérifier les structures lues).

Un autre besoin en OBJECTIVE CAML est celui d'itérateurs sur un type quelconque. Quelques modules fournissent des itérateurs polymorphes comme `List.map` ou `Hashtbl.fold` mais une extension est nécessaire pour générer une fonction de copie qui duplique une structure arbitraire ou des fonctions de parcourt compliquées.

Nous proposons une extension de OBJECTIVE CAML permettant de générer du code arbitraire à la compilation à partir d'informations de type comme la déclaration de type ou l'instance d'un type. Cette extension, appelée OCAML-TEMPLATES, est écrite à l'aide du préprocesseur CAMLP4.

Un prototype est téléchargeable sur `http://www.pps.jussieu.fr/~maurel/programmation/`.

1. Introduction

Un trait caractéristique des langages de la famille ML [2, 3] est un système de type puissant autorisant un polymorphisme important mais relativement contraint. Dans le langage OBJECTIVE CAML, les types des arguments possibles d'une fonction définie par l'utilisateur doivent tous vérifier un même schéma de type. Cela est légèrement généralisé en STANDARD ML où la surcharge est possible quand le type est connu. Le langage G'CAML [1] permet de rajouter de nouveaux schémas de type permettant de typer une fonction `int -> string | float -> string` sans que celle-ci ne soit constante ou déjà définie dans le compilateur. Dans le langage HASKELL [4], la notion de classe de type permet de résoudre ces problèmes de surcharge et même de générer automatiquement du code dans certains cas particuliers. Ces dernières méthodologies alourdissent l'exécution et pour des cas de taille réelle, la génération automatique de code nécessite un outil plus puissant comme TEMPLATE HASKELL ou CAMLP4.

Les systèmes de templates existants Les templates, appelés aussi modèles en français, apportent un confort de programmation en permettant de compléter du code à la compilation. En C++ [8], le programmeur peut paramétrer des définitions de types ou de fonctions avec des types ou des expressions. L'inspection de ses paramètres de type est possible dans certains cas mais difficile.

Les templates C++, bien que très générales, sont en pratique difficilement utilisables : il semble extrêmement compliqué de construire des exemples comparables aux nôtres en C++. Avec TEMPLATE HASKELL [7], le programmeur peut générer du code arbitraire à la compilation comme il le ferait avec un pré-processeur comme CAMLP4 avec la même facilité mais aussi la même généralité qui limite la réutilisation de code et la facilité de programmation. Une comparaison de différents systèmes de macros ou de templates est menée dans [7].

La méta-programmation La génération de code source, appelée aussi *meta-programmation*, est une technique consistant à programmer sur au moins deux niveaux : un programme engendre un programme qui est ensuite exécuté. De façon générale, un choix possible est de générer du code puis de l'exécuter ou de faire un va-et-vient entre la production de code et l'exécution. Dans un environnement typé, une question cruciale est de savoir si toute la phase de typage a lieu à la compilation ou si elle s'effectue en plusieurs étapes. Quelques systèmes de génération de code dans les langages fonctionnels :

Camlp4 Le pré-processeur CAMLP4 est l'outil standard de manipulation de code en OBJECTIVE CAML. Il permet des ajouts de syntaxe, de la manipulation arbitraire de code et a des fonctions d'affichage en standard. Un intérêt fondamental réside dans la possibilité pour une extension de générer du code source à la compilation ou dans une première phase et le programmeur ne diffuse qu'un code OBJECTIVE CAML compilable par tous y compris sans l'extension.

Metaml Le langage METAML [9] est une extension du langage ML permettant la programmation à plusieurs niveaux. Le code est produit à l'exécution.

Lisp, Scheme Ces langages possèdent un système de macros très puissant mais, naturellement, l'absence de types empêche une structuration similaire aux templates.

Haskell Le langage HASKELL prévoit quelques classes de type dérivables automatiquement comme `Show` ou `Read`. Cela signifie que HASKELL est capable de générer du code pour afficher ou pour lire un type plus ou moins arbitraire. Le programmeur doit utiliser une extension comme TEMPLATE HASKELL pour généraliser ces constructions.

La méthodologie Une fois le besoin d'un mécanisme de templates en OBJECTIVE CAML évalué, la mise en œuvre par l'outil CAMLP4 est la solution naturelle. La complexité de CAMLP4, due à sa généralité, impose de bâtir une extension particulière pour l'objectif visé. La programmation autour de OCAML-TEMPLATES s'effectue sur trois niveaux : (1) le noyau du système appelé OCAML-TEMPLATES programmé une fois pour toutes, (2) les templates de base distribuées avec le système et les templates que l'utilisateur peut écrire et enfin (3) l'application des templates pour créer des programmes d'intérêt général. Le niveau (1) est programmée en utilisant OBJECTIVE CAML et CAMLP4, le niveau (2) en utilisant de plus l'extension de syntaxe spécifique — définie niveau (1) — à la définition des templates ainsi que des fonctions prédéfinies qui complètent CAMLP4, enfin, le niveau (3) est programmé en utilisant *a priori* uniquement OBJECTIVE CAML ainsi que l'extension de syntaxe — définie elle aussi niveau (1) — permettant d'utiliser des templates — définis niveau (2) — qui s'appliquent au cas considéré.

L'organisation de l'article La section 2 introduit quelques exemples de templates et des notions sur leur utilisation. La section 3 apporte une classification informelle des templates suivant leurs *sortes*. Cette section décrit aussi brièvement les templates déjà écrits. La section suivante est consacrée à la description du système de templates : les modules utilisés, la grammaire concrète, le système d'options et de *patch*. Enfin, la dernière section décrit le mécanisme pour créer de nouveaux templates.

Remerciements Je remercie Jean-Baptiste Tristan qui a participé activement à l'implémentation de plusieurs templates ainsi que les rapporteurs anonymes pour leurs commentaires et leurs remarques.

2. Quelques exemples d'utilisation de Ocaml-templates

Comme aide au lecteur, quelques templates de base sont présentés ici et permettent de se faire une idée de l'utilisation et de la puissance du système des templates.

2.1. Le template `Random`

Un template assez pratique dans une phase de développement est le template `Random`. Il fournit du code pour créer, au hasard, une valeur d'un type donné.

Un exemple d'arbre binaire On veut définir le type des arbres binaires dont les feuilles sont annotées par des entiers et on veut créer une fonction `random_tree` générant un arbre au hasard.

La définition est simplement

```
type tree : Random =
    Leaf of int
  | Node of tree * tree
```

qui définit un type

```
type tree =
    Leaf of int
  | Node of tree * tree
```

et une fonction récursive

```
let rec random_tree () =
  match Random.float 1. with
    x when x >= 0.5 -> Leaf (Random.int 100)
  | _ -> Node (random_tree (), random_tree ())
```

Le système Ocaml-templates a appelé le template `Random` sur la définition de type correspondante et ce template a créé la fonction `random_tree`.

Le template `Random`, comme tous les templates, possède des options diverses dont une option pour les constructeurs permettant de définir une probabilité de partir dans une branche. Cette probabilité est uniforme par défaut.

Ainsi, le code

```
type tree : Random =
    Leaf / Random(proba={0.8}) of int
  | Node / Random(proba={0.2}) of tree * tree
```

produira la fonction `random_tree` suivante

```
let rec random_tree () =
  match Random.float 1. with
    x when x >= 0.2 -> Leaf (Random.int 100)
  | _ -> Node (random_tree (), random_tree ())
```

L'option du template `Random` est ici `proba` et prend un flottant en argument.

Un exemple d'arbre d'arité non bornée On veut maintenant changer notre notion d'arbre pour utiliser des arbres d'arité non bornée. Pour cela, on utilise des listes à chaque nœud. La fonction `random_tree` du paragraphe précédent devrait être totalement changée pour ce nouveau type. Ici, elle sera générée naturellement.

```
type tree : Random =
    Leaf of int
  | Node of tree list
```

définit le type sous-jacent et une nouvelle fonction récursive

```
let rec random_tree =
  let rec cree_liste n f () =
    if n = 0 then [] else f () :: cree_liste (n - 1) f ()
  in
  fun () ->
    match Random.float 1. with
      x when x >= 0.5 -> Leaf (Random.int 100)
    | _ -> Node (cree_liste (Random.int 5) random_tree ())
```

Un template appelé sans déclaration On peut aussi appliquer un template à une instance de type comme par exemple `{|Random of int list * float |}` qui crée du code produisant une paire constituée d'une liste aléatoire d'entiers et d'un flottant. Cette possibilité, très facile à mettre en œuvre, doit être prévue dans la définition du template. Ainsi, la définition `let f x = x :: {| Random of float list |}` crée une fonction qui prend en argument un flottant puis renvoie une liste constituée de cet argument et d'une liste tirée au hasard. La liste renvoyée est naturellement potentiellement différente à chaque appel de la fonction.

Les types de classes Un template peut aussi s'appliquer à un type de classe et crée typiquement dans ce cas une classe. Cela permet par exemple de faire un `new` sur un type de classe et non uniquement sur une classe. Cette possibilité doit de même être prévue dans la définition du template.

2.2. Un itérateur : Map

Le template `Map` permet de généraliser aux types paramétrés les itérateurs
`List.map : ('a->'b) -> 'a list -> 'b list`, ou
`Array.map : ('a->'b) -> 'a array -> 'b array`.

Un exemple typique est

```
type 'a tree : Map =
    Leaf of 'a
  |Node of 'a tree * 'a tree
```

qui est traduit par Ocaml-templates en

```
type 'a tree =
    Leaf of 'a
  | Node of 'a tree * 'a tree
let rec map_tree __map_a =
    function
```

```
    Leaf a_1 -> Leaf (__map_a a_1)
  | Node (a_1, a_2) -> Node (map_tree __map_a a_1, map_tree __map_a a_2)
```

qui a le type qu'on attend de lui `('a -> 'b) -> 'a tree -> 'b tree`. Sans les templates, le code de `map_tree` serait très simple mais fastidieux à écrire et tout changement de définition du type `tree` induirait une modification manuelle du code de `map_tree`.

2.3. La puissance du système d'options : définition de printf

Le système d'options des templates est suffisamment puissant pour qu'on puisse plonger un pré-processeur quelconque à l'intérieur. Une application simple est la fonction `Printf.printf`. Un template très simplifié et non optimisé a été écrit.

L'expression `{| Print {"i = %d j = %d  "} |}`, qui signifie "appelle le template `Print` avec l'option `"i = %d j = %d  "` sans l'appliquer à aucun type", est réécrite en l'expression :

```
fun a_1 a_2 ->
    print_string
      ("i = " ^ string_of_int a_1 ^ " j = " ^ string_of_int a_2 ^ "  ")
```

Ce qui donne bien le comportement attendu : l'expression
`{| Print {"i = %d j = %d  "} |} 5 12` affiche la chaîne `"i = 5 j = 12 "`.

2.4. Les templates Copy et Patch

Le template `Copy` fournit du code pour dupliquer un type quelconque. Il est assez utile en lui-même et devient très pratique en conjonction avec un système de patch. Ainsi,

```
type foo : Copy =
    A of int * string
  | B of float * int
```

fournit le code

```
let rec (copy_foo : foo -> foo) =
  function
    A (a_1, a_2) -> A (a_1, a_2)
  | B (a_1, a_2) -> B (a_1, a_2)
```

Si maintenant on désire changer le comportement en doublant les entiers et en annulant les flottants, on peut évaluer les deux expressions

```
let _ = {|Patch ({Copy},{fun x->2*x}) of int|}
let _ = {|Patch ({Copy},{fun _->0.}) of float|}
```

Les expressions signifient respectivement "dans le template `Copy`, appliquer la fonction `fun x->2*x` aux entiers" et "dans le template `Copy`, appliquer la fonction `fun _->0.` aux flottants".

À partir de ce point, et tant que ces modifications ne sont pas annulées, la définition

`type foo : Copy = A of int * string | B of float * int` génère le code

```
let rec (copy_foo : foo -> foo) =
  function
    A (a_1, a_2) -> A ((fun x -> 2 * x) a_1, a_2)
  | B (a_1, a_2) -> B (0., (fun x -> 2 * x) a_2)
```

On peut remarquer au passage que le template `Copy` réécrit `(fun _ -> 0.) a_1` en `0.` qui lui est équivalent dans ce cadre. Cela montre un exemple très simple d'évaluation partielle permise par les templates. Un autre exemple typique est la simplification de l'application de l'identité à une expression `exp` qui est réécrite en l'expression `exp`.

2.5. Interaction entre templates

Les templates sont appelés indépendamment par OCAML-TEMPLATES et le code généré n'interfère pas. Ainsi la déclaration suivante

```
type 'a arbre : Random : Copy (no_types) =
  Feuille of int
  | Noeud of 'a * 'a arbre * 'a arbre
```

Produit les fonctions suivantes en plus de la déclaration de type sous-jacente

```
let rec random_arbre random_a () =
  match Random.float 1. with
    x when x >= 0.5 -> Feuille (Random.int 100)
  | _ -> Noeud (random_a (), random_arbre random_a (), random_arbre random_a ())
let rec copy_arbre __copy_a =
  function
    Feuille a_1 -> Feuille a_1
  | Noeud (a_1, a_2, a_3) ->
      Noeud (__copy_a a_1, copy_arbre __copy_a a_2, copy_arbre __copy_a a_3)
```

L'option `no_types` du template `Copy` est simplement une option qui indique au template de ne pas déclarer dans le code le type de la fonction `copy_arbre`. Elle est utilisée dans cet exemple pour augmenter la lisibilité du code produit.

La convention typique de nommage (définie template par template) est que le template `Foo` appliqué au type `t` crée une fonction `foo_t`. Dans le cas de définitions récursives de types, un template est appelé sur la totalité de la définition récursive mais connaît les types sur lesquels il doit s'appliquer. Ainsi, il ne créera que les `foo_t` des types pour lesquels l'utilisateur a déclaré utiliser le template `Foo`.

3. Une classification informelle des sortes de templates

Ce paragraphe n'a pas de vocation à être formel mais plus à expliquer intuitivement quelques différences et caractéristiques des templates. Le terme *sorte* est employé en référence à la théorie des types où les fonctions entre types ont des sortes et non des types. Cette notion n'a pas de sens formel en OBJECTIVE CAML mais fournit une intuition utile.

Plusieurs styles de templates peuvent être mis en lumière : les *transformateurs* comme `Map`, `Copy`, ou `Proj` (qui sert à projeter un type sur un autre), les *consommateurs* comme `Print`, `Compare` ou `Iter` et les *créateurs* comme `Read`, `Default` ou `Random`.

Les transformateurs seront typiquement de sorte `type -> type`, les consommateurs de sorte `type -> cst_type` ou `type * type -> cst_type` et les créateurs de sorte `cst_type -> type` où `type` est le type sur lequel le template s'applique et `cst_type` un type constant comme `string` ou `unit`.

Une piste future est de donner un sens formel à cette notion de sorte et de typer les templates dès la compilation suffisamment précisément pour assurer que l'application des templates à n'importe quel type sera bien du bon type. Le typage d'une extension CAMLP4 arbitraire parait bien plus hors de portée que celui d'un template car le lien entre les templates et le typage est clair.

Définition 1 (Variance d'une formule) *Une formule en une variable est (1) covariante si toutes les occurrences de la variables sont à des positions covariantes, (2) contravariantes si toutes les occurrences de la variables sont à des positions contravariantes et (3) invariante si elle n'est ni covariante ni contravariante.*

La position racine de la formule est considérée comme covariante et chaque connecteur, suivant sa variance, change (cas de la contravariance) ou non (cas de la covariance) la variance des positions de ses sous-formules.

Définition 2 (Variance d'une sorte) *La variance d'une sorte S est la variance de la formule S par rapport à la variable `type` sachant que la flèche est covariante à droite et contravariante à gauche et que les autres constructions classiques (n-uplets, listes, tableaux, somme...) sont covariantes en leurs variables.*

Définition 3 (Consommateur) *Un template est un* consommateur *si sa sorte est contravariante.*

Définition 4 (Créateur) *Un template est un* créateur *si sa sorte est covariante.*

Définition 5 (Transformateur) *Un template est un* transformateur *si sa sorte est invariante.*

3.1. Les transformateurs : Map, Copy, Proj

Le template `Map` appliqué à un type `'a foo` fournit une fonction `map_foo : ('a -> 'b) -> 'a foo -> 'b foo`. Quelques optimisations permettent de ne pas parcourir toute la structure d'une expression possédant un type qui ne contiendrait pas de variable `'a`.

Le template `Copy` appliqué à un type `bar` fournit une fonction `copy_bar : bar -> bar` qui copie toute la structure. Le type de `copy_bar` est le même que celui de `map_bar` mais on a l'assurance de l'indépendance des structures. De plus, comme déjà indiqué, vu l'absence d'optimisations pour ne pas parcourir les structures à copier, il est très facile d'appliquer un *patch* au template `Copy`.

Le template `Proj` est plus compliqué. Lors de la définition d'un type, il permet d'en définir un autre par renommage des champs, ajout ou retrait de types, et de nouveaux cas. De plus, il fournit une fonction de projection du premier type sur le deuxième.

Ainsi, le code

```
type foo : Proj (rename = {"foo"/"bar"}, remove = {float}) =
    A_foo of int * float /add={string,"une chaine"}
  | B_foo of string * int list /remove
```

est réécrit en les définitions suivantes :

```
type foo =
    A_foo of int * float
  | B_foo of string * int list
type bar =
    A_bar of int * string
  | B_bar of string
let rec (bar_of_foo : foo -> bar) =
  function
    A_foo (a_1, a_2) -> A_bar (a_1, "une chaine")
  | B_foo (a_1, a_2) -> B_bar (a_1)
```

Ce template est utilisé dans le module Pa_templates pour fournir au programmeur de templates une grammaire abstraite ne contenant plus les positions des caractères lus dans les fichiers alors qu'ils sont gardés dans une première phase pour permettre une gestion d'erreur via CAMLP4.

Ces trois transformateurs, peut-être les plus utiles dans une phase de développement, ont des codes assez proches mais fournissent des fonctions bien distinctes. Ils sont, pour l'instant, définis sur un sous-ensemble de la grammaire des types très suffisant en pratique.

3.2. Les consommateurs : Print, String_of, Compare, Tdpe, Iter

Le template `Print` a pour but de définir à la fois un mécanisme similaire à `printf` et mais aussi de fournir des fonctions d'affichage pour un type quelconque autre que fonctionnel et objet. Il doit commuter raisonnablement avec le template `Read` ce qui devrait en faire une alternative au module `Marshal` de la bibliothèque standard ou plus précisément à la bibliothèque IoXML [6].

Le template `Compare` a pour but de fournir des fonctions de comparaison paramétrables sur un type en négligeant par exemple des champs inutiles. Il pourra être optimisé pour être plus rapide et plus sûr que la comparaison de OBJECTIVE CAML.

Le template `Iter` fournit une extension de `List.iter` à la manière de `Map`. On pourrait de même écrire un template `Fold`.

Le template `Tdpe`, qui n'a pas vocation à aller dans une bibliothèque standard mais plus de servir de test pour développer les templates, est un template qui à partir d'un type totalement polymorphe crée une paire de fonctions *down/up* permettant d'appliquer l'algorithme de "Type Directed Partial Evaluation" qui "décompile" une expression d'un type donné.

Ainsi, le code suivant

```
type lambda =
    Var of string
  |Lambda of string * lambda
  |Application of lambda * lambda;;
type 'a t : Tdpe = ('a -> 'a) -> ('a -> 'a);;
let deux f x = f (f x);;
let trois f x = f (f (f x));;
let d,u=tdpe_t ();;
d (deux trois);;
```

renvoie

```
Lambda ("x_1", Lambda ("x_3",
  Application (Var "x_1",  Application (Var "x_1", Application (Var "x_1",
    Application (Var "x_1",  Application (Var "x_1", Application (Var "x_1",
      Application (Var "x_1",  Application (Var "x_1", Application (Var "x_1",
        Var "x_3"))))))))))
```

qui a bien "décompilé" l'application du lambda terme de Church 2 au lambda terme 3.

3.3. Les créateurs : Read, Default, Random

Le template `Read` a pour but de généraliser à la fois `int_of_string` et `Marshal.of_string`.

Le template `Default` a pour but de créer une valeur par défaut sur un type quelconque. Celle-ci correspond aux valeurs nulles par défaut de certains langages à objet par exemple.

Le template `Random` ressemble au template `Default` mais est bien plus complexe doit permettre de tirer des valeurs au hasard.

3.4. Typage et récursion polymorphe

L'algorithme de typage de OBJECTIVE CAML ne généralise pas les types des fonctions récursives et cela empêche parfois la définition de fonctions parfaitement valables. Ainsi, comme soulevé par un rapporteur anonyme du présent article, la définition suivante

```
type 'a foo : Map =
    A of 'a
  | N of 'a list foo
```

crée le code suivant non typable

```
let rec map_foo __map_a =
  function
    A a_1 -> A (__map_a a_1)
  | N a_1 -> N (map_foo (List.map __map_a) a_1)
```

Cette expression n'est pas typable car `map_foo` devrait avoir un type quantifié universellement et il n'est quantifié que faiblement universellement. Une solution actuellement utilisée est de relâcher les contraintes de typage à l'aide de l'instruction `Obj.magic`. Cette solution est mise en place à l'aide de l'option `magic` pour ce template (`type 'a foo : Map(magic)`) et fournit la définition suivante

```
let rec map_foo __map_a =
  function
    A a_1 -> A (__map_a a_1)
  | N a_1 ->
      N ((Obj.magic map_foo : ('a1 -> 'b2) -> 'a1 foo -> 'b2 foo)
           (List.map __map_a) a_1)
```

Ainsi que la ligne suivante

```
let _ = let module Unused : sig val x : ('a1 -> 'b2) -> 'a1 foo -> 'b2 foo end =
    struct let x = map_foo end in ()
```

qui permet de *vérifier* la validité du typage au prix d'un coût minime à l'exécution... Nous aurions pu créer très facilement une expression équivalente typable sans `Obj.magic` avec les enregistrements ou les modules récursifs mais le code engendré aurait été moins rapide et moins lisible. L'idée est plus d'espérer un changement du système de typage de OBJECTIVE CAML qui pourrait accepter ces expressions et de tricher légèrement avec le typage pour l'instant plutôt que de créer du code superflu. Un argument fort pour cette approche est la sûreté du code produit comme le prouve la transformation avec des enregistrements (et la vérification avec le module `Unused`). Une solution simple à utiliser — mais peut-être difficile à implémenter — serait d'autoriser l'utilisateur à indiquer le type explicitement polymorphe des fonctions récursives et que le système de typage vérifie ce type comme cela se fait pour les enregistrements.

Cette utilisation de `Obj.magic` est uniquement mise en place pour le template `Map` mais doit être encore étudiée et similairement adaptée aux autres templates.

4. L'organisation du système

Les partis-pris pour la construction de OCAML-TEMPLATES sont multiples :
– Le compilateur reste inchangé. Seule la construction de l'arbre de syntaxe est touchée.

- Il doit être possible de créer, transformer, ajouter et utiliser facilement des templates.
- La gestion des templates doit être centralisée au maximum ce qui impose une syntaxe générique pour l'appel des templates.
- Les templates doivent être facilement paramétrables par un système d'options spécifiques à chaque template. Ces options doivent être gérées uniformément par le système avant de rentrer dans les templates.
- On veut avoir la puissance suffisante pour écrire un `printf` et un `marshal` ou définir des itérateurs sur des types quelconques comme par exemple `'a list list * 'a array`.

4.1. Le noyau

Le noyau du système se compose de quatre modules : extension de syntaxe pour utiliser les templates, extension de syntaxe pour définir les templates, manipulation des templates et modification des templates à la volée. Une description plus précise suit :
- Le module `Pa_templates` définit l'extension de syntaxe pour l'utilisation des templates dans un programme Objective Caml quelconque.
- Le module `Pa_template_definition` définit une extension de syntaxe pour aider à définir les templates. Ce module n'est pas indispensable à l'utilisation du système mais apporte du *sucre syntaxique* utile. Ce module peut être encore complété.
- Le module `Templates` définit les grammaires abstraites des types et fournit des fonctions utiles à la définition des templates. Il s'occupe aussi de gérer l'ensemble des templates chargés au moment de l'analyse syntaxique d'un programme Objective Caml.
- Le petit module `Te_patch` permet la modification à la volée de parties du code des templates. Accessoirement, c'est lui-même un template.

4.1.1. Le schéma général

Chaque template est défini dans un fichier `.ml`. Une grammaire spécifique définie dans le module `Pa_template_definition` permet d'aider à la définition du template. Dans le code du template, il y a un appel à la fonction `Templates.add` qui permet d'ajouter ce template à l'ensemble des templates gérés par le module `Templates`. Le code du template consiste fondamentalement à créer du code à partir d'une information de type qui lui est passée en paramètre au moment de l'analyse syntaxique d'un fichier `.ml` par le module `Pa_templates`. Ce module définit l'extension de syntaxe et lors de l'analyse d'une déclaration de type ou d'un appel avec la syntaxe `{| ... |}` s'occupe d'appeler le template correspondant avec les options qui le concerne.

4.1.2. Grammaire générique

La grammaire concrète utilisée est un sur-ensemble de la grammaire standard d'Objective Caml pour les déclarations de types et les expressions. La notation pour définir la grammaire est la suivante : $< \ldots >$ appelle un symbole non terminal de la grammaire, $\Big|$ sépare les cas, $\Big\{ \ldots \Big\}$ décrit une liste éventuellement vide et $\Big[\ldots \Big]$ décrit un élément optionnel. Les symboles terminaux sont "{|", "|}", "=", "*", "/", ":", ";", "," ainsi que "of", "default", "mutable" et les identificateurs.

La figure 1 décrit la grammaire concrète utilisée. Ne sont notés que les cas supplémentaires par rapport à Objective Caml dans la grammaire ou les cas éclairants.

On peut remarquer sur cette grammaire que les templates autorisent le placement d'options à divers endroits et qu'il y a un système de template par défaut ce qui évite d'avoir à spécifier le template considéré lors du passage d'une option.

La syntaxe est étendue de façon similaire pour les types de classes.

$$
\begin{array}{lll}
\mathit{expr} & ::= & \{|\ <\mathit{Template}> \left[<\mathit{options}>\right] \left[\text{of} <\mathit{type\text{-}expr}> \left\{* <\mathit{type\text{-}expr}>\right\}\right]\ |\} \\
 & & \\
\mathit{name} & ::= & \textit{Identificateur commençant par une minuscule} \\
\mathit{Template} & ::= & \textit{Identificateur commençant par une majuscule} \\
\mathit{option} & ::= & <\mathit{name}> \left[= \{<\mathit{expr}>\}\right] \\
\mathit{options} & ::= & <\mathit{option}> \\
 & | & (<\mathit{option}> \left\{, <\mathit{option}>\right\}) \\
 & & \\
\mathit{type\text{-}expr} & ::= & <\mathit{type\text{-}expr}> \left\{/ \left[<\mathit{Template}>\right] <\mathit{options}>\right\} \\
 & & \\
\mathit{type\text{-}def} & ::= & \left[<\mathit{parametres}>\right] \left\{: \left[\text{default}\right] <\mathit{Template}> \left[<\mathit{options}>\right]\right\} \\
 & & <\mathit{name}><\mathit{type\text{-}info}> \\
\mathit{type\text{-}info} & ::= & \left[<\mathit{type\text{-}equation}>\right] \left[<\mathit{type\text{-}representation}>\right] \left\{<\mathit{type\text{-}constraint}>\right\} \\
\mathit{type\text{-}equation} & ::= & =<\mathit{type\text{-}expr}> \\
\mathit{type\text{-}representation} & ::= & =<\mathit{constr\text{-}decl}> \left\{| <\mathit{constr\text{-}decl}>\right\} \\
 & | & = \{<\mathit{field\text{-}decl}> \left\{; <\mathit{field\text{-}decl}>\right\}\} \\
\mathit{constr\text{-}decl} & ::= & <\mathit{constr\text{-}name}> \left\{/ \left[<\mathit{Template}>\right] <\mathit{options}>\right\} \\
 & | & <\mathit{constr\text{-}name}> \left\{/ \left[<\mathit{Template}>\right] <\mathit{options}>\right\} \text{ of} \\
 & & \qquad <\mathit{type\text{-}expr}> \left\{* <\mathit{type\text{-}expr}>\right\} \\
\mathit{field\text{-}decl} & ::= & \left[\text{mutable}\right] <\mathit{field\text{-}name}> \left\{/ \left[<\mathit{Template}>\right] <\mathit{options}>\right\} \\
 & & \qquad :<\mathit{poly\text{-}typexpr}>
\end{array}
$$

FIG. 1 – Grammaire concrète

4.1.3. Analyse syntaxique

L'analyse syntaxique se fait en étendant la grammaire d'OBJECTIVE CAML telle que décrite dans le fichier `pa_o.ml` de CAMLP4. La difficulté de l'extension réside dans plusieurs facteurs : le système d'options pour les templates impose d'utiliser une structure différente pour les types de celle par défaut dans CAMLP4 et de traduire ensuite si besoin cette structure. De plus, la structure que l'on désire donner au programmeur qui crée un template est plus structurée que celle de CAMLP4 qui écrase trop de constructions. L'idée a donc été de réécrire une grammaire abstraite pour les types à la fois plus proche de celle de OBJECTIVE CAML et contenant les informations suffisantes pour recréer des arbres de syntaxe abstraite CAMLP4. Cela a nécessité de plus de réécrire pratiquement entièrement la partie consacrée à l'analyse syntaxique des types.

4.1.4. Le système d'options

Le type des options est `string * MLast.expr option` qui représente le nom de l'option et la valeur éventuellement transportée (une expression OBJECTIVE CAML quelconque). Les expressions transportées doivent être décodées au niveau de chaque template qui seul sait ce qui est attendu dans chaque option. Ce décodage se fait typiquement à l'aide d'un `match ... with` et n'utilise que des fonctions bien typées.

Pour acheminer les options aux templates correspondants, le module Pa_templates utilise des itérateurs appelés map_e_type_def ou map_e_ctyp définis dans le module Templates. Naturellement, ces itérateurs ont eux-mêmes été créés à l'aide du template Map.

4.2. Les templates prédéfinis

Le principe sous-jacent à OCAML-TEMPLATES est que l'utilisateur final utilise en priorité les templates déjà existants et n'écrive ses propres templates qu'en cas de nécessité. Une distribution raisonnable doit donc inclure des templates de base d'utilisation générale un peu à la manière de la bibliothèque standard. Un programmeur typique utilisant les templates utilisera un template déjà existant en priorité, éventuellement à l'aide d'un *patch*.

4.3. Le choix du template

Le programmeur désirant utiliser un template a plusieurs options : (1) il possède un template qui fait exactement ce qu'il désire et il peut l'utiliser facilement ; (2) il possède un template sur lequel il peut appliquer un *patch* qui fera ce qu'il désire ou (3) il n'a pas de template adapté. Dans le premier cas, il lui suffit d'appeler son template. Dans le deuxième cas, il peut appliquer le *patch* avant d'appeler son template et peut éventuellement enlever son *patch* après. Ces opérations se font directement dans le même fichier simplement avant et après l'utilisation de son template. Dans le dernier cas, il lui faut définir un template dans un autre fichier et le compiler au préalable comme expliqué à la section 5. Il peut aussi générer du code et le retoucher après même si cette approche n'est pas très réutilisable et donc contraire à l'esprit des templates.

4.3.1. Le système de patch

Un système de *patch* existe par défaut. Les commandes accessibles sont
- L'ajout d'un même patch pour une liste de noms de types :
  {| Patch ({<Template>},{<code>}) of <nom_type_1> * ... * <nom_type_n> |}. Suivant la définition du template Template, le code code remplacera le code habituel du template pour ces types.
- La mise à zéro des patches : {| Patch reset |}.
- La mise à zéro des patches d'un template : {| Patch reset={Template} |}.
- Retrait de dernier patch d'un template : {| Patch remove={Template} |}.

Avec la syntaxe définie dans le module Pa_template_definition, la partie d'un template qui s'occupe des patches est simplement le code cryptique suivant

```
match get_patch_template tp with
  Some e->e
|None->
```

placé lors du désassemblage du type qui serait un chemin de type (*type path*). L'extension de syntaxe se contente de traduire get_patch_template en get_patch "Template" où "Template" est une chaîne de caractère représentant le nom du template. Ainsi, tout le travail de gestion est fait de façon centralisée dans le module Pa_templates.

Pour l'instant, on ne peut appliquer des patches que sur des types nommés et non sur des constructions plus complexes. Le système de patch permet naturellement de rajouter des comportements au template sur des types définis dans des modules externes sans que ces définitions aient été prévues à l'écriture du template. Le système d'options permet de faire des modifications plus importantes si besoin mais les options doivent être prévues à l'écriture du template.

5. Comment créer un template

La définition d'un template est fondamentalement une définition par cas sur la structure du type et quelques lignes pour inscrire le template dans `Pa_templates`.

La création d'un template à partir de zéro est compliquée par plusieurs facteurs : (1) une connaissance minimale du système de quotations de CAMLP4 est nécessaire pour créer du code. Ce système n'étant pas complet, il est parfois nécessaire de s'impliquer un petit peu plus..., (2) la variété des types en OBJECTIVE CAML est assez importante et impose une taille minimale aux templates.

Un exemple typique est le template `Default` dans lequel on trouve les fonctions : `generator` qui teste si son argument est un type somme, enregistrement ou une expression de type et fournit le code correspondant, `generator_type_path` qui crée du code pour les noms de types (éventuellement dans des modules externes), `generator_pol` qui s'occupe des types polymorphes (pour les types de classe et les enregistrements), ainsi que la fonction `generator_core` qui crée du code pour les expressions de types (n-uplet, flèche, application, alias, variable, objets...) et leurs options.

5.1. Les choix de conception

Le programmeur de templates doit fixer la sorte exacte de son template dès le début : partir en aveugle est dangereux puisque le typage du template n'impose pas le typage de l'application du template à un type donné. La sorte influe aussi de manière subtile sur la stratégie de nommage dans les fonctions créées.

Un point important est la sorte des appels récursifs. Cela peut être expliqué en comparant les lignes correspondant au cas du type `bool` dans les templates `Random` et `Print`.

```
Bt_Bool ->
   <:expr< Random.bool () >>

Bt_Bool ->
   <:expr< fun  [ True -> print_string "true"
                | False -> print_string "false"]  >>
```

Dans le template `Random`, on manipule récursivement des expressions de sorte `type` alors que dans `Print`, on manipule des fonctions de sorte `type -> unit` alors que les sortes de ces templates sont `unit -> type` et `type -> unit`. Cette distinction doit être faite dès le début de l'écriture du template. Une légère difficulté mise en lumière par cet exemple est aussi l'obligation de CAMLP4 d'utiliser la syntaxe révisée [5] dans les quotations (de la forme `<:expr< ... >`).

5.2. Des fonctions d'aide à la définition de templates

Quelques fonctions d'aide à la définition de templates sont définies dans le module `Pa_templates`. On peut citer (1) un système de gestion de définitions locales : pendant la création de code, on indique quelles définitions locales sont nécessaires et le template préfixe le code généré par ces définitions locales, (2) des fonctions qui améliorent les quotations de base de CAMLP4, (3) des fonctions pour créer des listes d'expressions ou de motifs `a_1,...,a_n` et (4) des fonctions un peu ésotériques comme `eta_expand_if_necessary` qui permet de faire une eta-expansion avant une définition récursive pour qu'elle soit acceptée plus facilement par les tests de OBJECTIVE CAML.

5.3. Une syntaxe adaptée

Le module pa_template_definition ajoute du sucre syntaxique pour aider à définir un template. Ce module est susceptible d'évoluer fortement dans le futur. Un fichier typique aura cet aspect :

```
template Template with
      te_ti  = ti  (* pour les expressions {| Template of ... |} *)
  and te_td  = td  (* pour les declarations de types *)
  and te_ctd = ctd (* pour les declarations de types de classe *)
struct
  let rec generator = ...
  and
  ...
  ... (* code du template *)
  ...
end
```

Un point à remarquer est que `Template` n'est pas sous forme de chaîne de caractères mais simplement un identificateur commençant par une majuscule.

Une autre technique encore très expérimentale en vue de simplifier l'écriture est la possibilité de copier/coller du code. Ainsi un template, comme `Default`, pourra se présenter comme ceci :

```
get_code_file "modeles/mo_vide.ml.code"
template Template with ...
struct
  let rec generator = inherit_code generator
  and ...
end
```

La commande `get_code_file "modeles/mo_vide.ml.code"` charge le code des définitions du fichier `modeles/mo_vide.ml.code` et l'expression `inherit_code generator` colle le code de la fonction `generator` à cet endroit. Des possibilités de modifications de code, et non simplement de copie, existent mais ne sont pas encore utilisées. Ces outils se trouvent dans le répertoire `code_ref` de la distribution.

Conclusions et développements futurs

Le système de templates est pour l'instant à l'état de prototype. Le noyau est assez raisonnablement complet mais la bibliothèque de templates est pour l'instant incomplète en quantité — il faudrait rajouter des templates — mais aussi, et c'est sûrement plus important, en qualité — les templates doivent être complétés et parfois corrigés. Des templates naturels à écrire sont `Share` qui permettrait de gérer la partage dans les structures et de les minimiser par exemple ou `Compress` qui pourrait fournir un algorithme de compression dédié...

Le code de OCAML-TEMPLATES utilise environ 300 lignes générées par des templates pour créer des itérateurs. Ces lignes ont pour la plupart été écrites une première fois à la main mais certaines (qui concernent les types de classes) ont pu être générées à partir de zéro. La possibilité de générer ces lignes simplifie grandement le travail de développement puisque le changement de la définition d'un type ne s'accompagne plus d'un travail fastidieux de recopie du type comme pour le cas du `Map` sur un type somme. Cette possibilité a été utilisée dans OCAML-TEMPLATES pour créer des itérateurs sur les structures internes de CAMLP4 permettant un portage aisé entre différentes versions (3.06 et 3.07).

Les templates sont typés à la compilation mais le type est très peu expressif : tous les templates ont le même type. Cela provient de CAMLP4 mais n'est pas un défaut dans le cadre général car la manipulation de code passe par des phases *instables* — typiquement, les variables ne sont pas forcément toujours liées pendant les phases de création de code. Après application, il faut encore typer l'expression obtenue. Une direction de recherche, pour l'instant très prospective, est de réussir à typer les templates directement au moment de la compilation du template de manière suffisamment expressive pour pouvoir assurer le typage *après* application. Il faudrait pour cela bien définir une notion de sorte comme expliqué section 3. Des idées existent pour le faire directement à l'intérieur du système de typage de OBJECTIVE CAML. Il faut remarquer que typer la génération de code en général semble plus difficile que simplement typer les templates qui manipulent des types. Les types que l'on peut envisager seraient `'a type_expr -> ('a -> unit) code` où `'a type_expr` représente une expression de type décrivant un type `'a` et `('a -> unit) code` représente le code d'une expression de type `'a -> unit`.

Références

[1] Jun Furuse. Generic polymorphism in ML. *Journées Francophones des Langages Applicatifs.* 2001.

[2] Xavier Leroy *et al.* The Objective Caml system – Documentation and user's manual. Disponible sur le site `http://caml.inria.fr/`, 2003.

[3] Robin Milner, Mads Tofte and Robert Harper. The Definition of Standard ML : version 3. *Technical Report EFS-LFCS-89-81, University of Edinburgh*, 1989.

[4] Simon Peyton Jones. Haskell 98 Language and Libraries. *Cambridge University Press.* 2003. Hardback, 272 pages, ISBN : 0521826144.

[5] Daniel de Rauglaudre. Camlp4 - Reference Manual. Disponible sur le site `http://caml.inria.fr/camlp4`, 2003.

[6] Daniel de Rauglaudre. IoXML. Disponible sur le site `http://cristal.inria.fr/~ddr/IoXML`, 2002.

[7] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *ACM SIGPLAN Haskell Workshop 02*, 2002.

[8] Bjarne Stroustrup. The C++ Programming Language (Special Edition). *Addison Wesley.* Reading Mass. USA. 2000. ISBN 0-201-70073-5. 1029 pages. Hardcover.

[9] Walid Taha, Zine-el-abidine Benaissa and Tim Sheard. The essence of staged programming. Technical Report, Oregon University. 1997. Disponible sur le site `http://www.cse.ogi.edu/PacSoft/projects/metaml/`