Cast objet pour la construction de hiérarchies de classes en Objective Caml

Emmanuel Chailloux[1]

1: Equipe Preuves, Programmes et Systèmes (EP 2025),
Université Paris 7
175, rue du Chevaleret
75013 Paris, France
`Emmanuel.Chailloux@pps.jussieu.fr`

Résumé

L'extension objet du langage Objective Caml s'intègre au typage statique et à l'inférence de types du langage. Cette extension comporte des traits de haut niveau comme l'héritage multiple et les classes paramétrées en distinguant bien les relations d'héritage et de sous-typage. Elle offre par la liaison retardée et la relation de sous-typage le polymorphisme d'inclusion des langages à objets. Néanmoins il lui manque une véritable hiérarchie de classes. Cela provient principalement de l'absence de «cast» dynamique entre un objet d'une classe vers une sous-classe. On propose ici une extension de langage, réalisée par le préprocesseur `camlp4`, qui autorise ce transtypage entre types «objets» appartenant à une même hiérarchie de classes et vérifiant la relation de sous-typage. À partir de cette nouvelle extension on construit des modèles de conception («design patterns») jusque là impossibles à écrire pour la définition de hiérarchies de classes réutilisables.

Introduction

Objective Caml [5], [2] comme son nom l'indique possède une extension objet qui apporte un langage de classes au noyau fonctionnel/impératif de Caml. Une déclaration de classe construit un nouveau type «objet» et une fonction de construction d'instances de ce type. Les classes peuvent être en relation d'héritage et en relation de sous-typage. Ces deux relations sont bien distinctes. Les déclarations de classes acceptent l'héritage multiple et les classes paramétrées. Par contre la surcharge de méthode n'est pas acceptée. Du point de vue de l'exécution la liaison est toujours retardée.

Il faut considérer le langage Objective Caml comme un langage muni d'une extension objet et non pas comme un langage à objets. Néanmoins grâce à la liaison retardée il permet de structurer des composants logiciels plus réutilisables [1]. Malheureusement cette extension ne permet pas de construire facilement des hiérarchies de classes générales. En particulier certains modèles de conceptions classiques [4] comme les fabriques ou les modèles à délégation à la AWT de Java ne peuvent pas être définis. Dans ces cas il n'est pas toujours possible de connaître le type exact retourné par l'appel d'une méthode : seul le type d'une classe ancêtre est connu comme type du résultat. Pour résoudre ce problème il est nécessaire d'ouvrir le dogme du typage statique en autorisant une vérification dynamique de type en cas de contraintes de types («downcast») entre objets.

C'est ce que l'on propose dans cet article. Cette nouvelle extension, appelée `coca-ml`[1] réalisée grâce à l'outil `camlp4` [6] autorise le «cast» dynamique d'objets d'une classe `c` vers une autre classe `d` si d'une part la classe `d` est bien sous-type de la classe `c` et d'autre part si `d` est bien une sous-classe,

[1]Pour "Cast objet en caml", URL = `http://www.pps.jussieu.fr/~emmanuel/Public/Dev/coca-ml`

au sens de l'héritage, de la classe `c` et de toutes les sous-classes entre `c` et `d` de l'arbre d'héritage. La première propriété est vérifiée statiquement au typage, la deuxième propriété nécessite une vérification dynamique à l'exécution. Il devient donc possible après avoir contraint le type d'un objet de classe `d` vers le sur-type `c`, de pouvoir de nouveau considérer cet objet comme de classe `d` ou de toute classe intermédiaire entre `c` et `d` de l'arbre d'héritage. Pour être sûr qu'une classe `d` est bien sous-type de toutes les classes dérivées de `c` vers `d` il est nécessaire de le vérifier soit au moment de l'affaiblissement de types («upcast»), soit au moment de la définition de la classe `d` en testant que cette classe est bien sous-type des classes dont elle hérite. Ces deux techniques ont été implantées mais cette dernière a notre préférence car elle ne modifie pas l'opérateur d'affaiblissement (`:>`) du langage tout en étant plus facile à utiliser et en étant plus efficace.

Cet article s'organise en quatre parties. La première effectue une présentation des relations d'héritage et de sous-typage du langage Objective Caml. La deuxième présente l'extension `coca-ml` et montre sa compatibilité avec le système de types du langage. Elle est illustrée par plusieurs exemples. Son implantation y est décrite. La troisième partie s'intéresse à l'implantation du modèle de conception «Fabrique Abstraite». La quatrième partie compare les modèles objets de différents langages en présence de typage statique. Enfin une courte conclusion clôt cet article. Pour le confort de lecture on distingue typographiquement le nom d'une classe `c` de son abréviation de type c.

1. Héritage et sous-typage en Objective Caml

La définition d'une classe Objective Caml définit une abréviation d'un type objet et une fonction de construction d'instances de ce type. Un type objet correspond à une suite de noms de méthodes associés à leur type :

$$< m_1 : \tau_1 ; m_2 ; \tau_2 ; ... ; m_n : \tau_n >$$

Les types τ_i peuvent être fonctionnels ou objets, et les types objets peuvent être récursifs et paramétrés. Les variables d'instance n'apparaissent pas dans le type engendré par la définition d'une classe.

Les deux relations principales entre les classes sont :

- la relation structurelle d'héritage;
- la relation de sous-typage entre les types des classes.

Bien que certains langages objets confondent ces deux relations en une seule, Objective Caml les distingue.

1.1. Relation d'héritage

Lorsqu'une classe `d` est définie en héritant de la classe `c`, toutes les méthodes apparaissant dans l'abréviation de type `c` sont présentes dans l'abréviation de type d. Si `d` définit de nouveaux noms de méthodes, celles-ci apparaîtront dans le type inferé. Si `d` redéfinit une méthode héritée, celle-ci devra avoir le même type que la méthode héritée : il n'y a pas de surcharge en Objective Caml. Néanmoins une méthode redéfinie peut contenir le type en cours de définition dit «type de `self`».

Supposons que le type suivant a été inferé par la déclaration de la classe `c` :

$$c = < m1 : \alpha \rightarrow \tau_1 ; m2 : \tau_2 \rightarrow \alpha > \;\; as\ \alpha$$

On note une définition récursive du type. Ce n'est pas un type paramétré car la variable α n'apparaît pas dans le terme gauche et est liée par le `as`.

Si maintenant on déclare une classe `d` qui hérite de `c` et qui définit une nouvelle méthode `m3` de type int, on obtient le type suivant :

$$d =< m1 : \alpha \rightarrow \tau_1; m2 : \tau_2 \rightarrow \alpha; m3 : int > \ as\ \alpha$$

La classe `d` a hérité des méthodes définies dans `c`, mais comme ces méthodes contenaient le type de l'instance (type de `self`) dans leurs types celui-ci correspond au type `d` dans la définition de `d`. Le fait d'hériter des méthodes contenant le type de `self` change en fait le type de ces méthodes dans la mesure où le type de `self` change.

1.2. Sous-typage

La relation de sous-typage est une relation réflexive et transitive entre types.

Un type objet d $= < m1 : \tau_1'; ...; m_p : \tau_p'; m_{p+1} : \tau_{p+1}'; ...; m_n : \tau_n' >$ est un sous-type d'un type objet c $= < m1 : \tau_1; ...; m_p : \tau_p >$ si tous les noms de méthodes apparaissant dans c existent aussi dans d et tous les types des méthodes m_i (pour i allant de 1 à p) de d sont sous-type des types de ces mêmes méthodes dans c. Cette relation est notée $\leq$.

Comme le type d'une méthode est quelconque cette relation s'applique à tous les types du langage de types. Les types de base ne sont sous-type que d'eux-mêmes. La relation vérifie pour les constructeurs `*` (paire) et `->` (fonctionnel) les propriétés suivantes :

- si $\tau_i \leq \tau_i'$ alors $(\tau_1 * \tau_2) \leq (\tau_1' * \tau_2')$
- si $(\tau_1' \leq \tau_1)$ et $(\tau_2 \leq \tau_2')$ alors $(\tau_1 \rightarrow \tau_2) \leq (\tau_1' \rightarrow \tau_2')$

Le constructeur flèche est co-variant à droite et contra-variant à gauche.

Soit un objet o de type τ_1, si $\tau_1 \leq \tau_2$ alors il est possible d'utiliser l'objet o en lieu et place de valeurs de type τ_2. Cette contrainte de type est explicite dans le langage Objective Caml et s'exprime de la manière suivante : `(expr : tau1 :> tau2)` indiquant que si `expr` est du type τ_1 et si $\tau_1 \leq \tau_2$ alors `expr` est vu comme du type τ_2. Une fois l'objet `o` considéré de type τ_2, il est impossible de le reconsidérer comme un objet de type τ_1 si les types sont différents.

Cette contrainte de typage se retrouve dans la littérature francophone sous différentes appellations selon le point de vue adopté. Au niveau des types on parlera d'«affaiblissement» du type, au niveau des classes de «sur-classage». Si on intègre les notions d'héritage et de types on peut employer «transtypage ascendant» ou «upcast».

1.3. Héritage n'est pas sous-typage

Ces deux relations sont de natures différentes. En Objective Caml elles ne sont pas équivalentes, en particulier l'héritage ne garantit pas le sous-typage. Cela provient des méthodes où apparaît le type de `self` dans un type fonctionnel et de la contra-variance à gauche du constructeur fonctionnel flèche.

Si on construit une classe `c` dont le type engendré est $c =< eq : \ 'a \rightarrow bool > \ as\ 'a$ alors une classe `d` qui hérite de `c` et qui définit une nouvelle méthode `m` n'est pas sous-type de `c`. En effet pour que $d \leq c$ il faut que le type de la méthode `eq` de `d` soit sous-type du type de la méthode `eq` de `c`. C'est-à-dire $d \rightarrow bool \leq c \rightarrow bool$. De part la contra-variance à gauche cela implique que $c \leq d$. Ce qui n'est pas vrai car la méthode `m` de `d` n'existe pas dans `c`.

Le seul cas où l'héritage conserve la relation de sous-typage en présence de méthodes comportant le type de `self` en position paramètres est quand les types des classes sont identiques. Par contre quand les classes ne possèdent pas de telles méthodes, alors l'héritage donne bien aussi la relation de sous-typage.

C'est justement cette propriété qui va permettre de proposer une extension au langage Objective Caml pour faire du «downcast» sur les objets. On trouve là aussi plusieurs traductions comme les termes de «transtypage descendant», «sous-classage», «spécialisation» ou «rétrécissement» ou encore «renforcement».

2. Extension `coca-ml`

Le but principal de l'extension `coca-ml` est d'autoriser le «downcast» objet sans casser le système de types du langage. Cela implique de devoir effectuer une vérification de type à l'exécution. L'information de types d'un programme Objective Caml est perdue une fois le programme correctement typé. Cela nécessite de conserver un minimum d'informations pour connaître la classe de construction d'un objet, l'appartenance d'une classe à un arbre d'héritage de classes et la vérification de la relation de sous-typage entre classes.

On introduit alors les opérateurs suivants :

- `cc(o)` : donne la classe de construction de l'objet `o`;
- `d` *hérite de* `c` : qui indique que `d` vaut `c` ou bien que `d` a `c` comme ancêtre dans l'arbre d'héritage;
- `d` *sous-hérite de* `c` : équivalent à `d` *hérite de* `c` et $d \leq c$.

La relation de sous-héritage est elle aussi réflexive et transitive.

Pour se faire chaque classe possède une clé unique de type *id* sous forme d'une déclaration globale et chaque déclaration de classe ajoute les cinq méthodes de la figure 1 dans le corps de la classe.

méthode	type	action
`get_id`	*id*	retourne la clé de la classe de construction
`check_id`	$id \rightarrow bool$	teste si la classe de construction de l'objet est identique à l'argument
`check_instanceof_id`	$id \rightarrow bool$	teste si la classe de construction de l'objet est une sous-classe de l'argument;
`check_subinstanceof_id`	$id \rightarrow bool$	teste si la classe de construction est en relation de sous-héritage avec la classe passé en argument
`check_subclass`	$id \rightarrow id \rightarrow bool$	teste si la classe de construction d'un objet est en relation de sous-héritage avec le premier argument et que celui-ci sous-hérite du second.

Figure 1: Méthodes ajoutées

L'idée principale de `coca-ml` est d'autoriser une contrainte de type («cast») d'un objet `o` considéré comme du type c d'une classe `c` vers le type d d'une classe `d` si `cc(o)` *sous-hérite de* `d`. On distingue alors une contrainte de type ascendante («upcast») qui vérifie `cc(o)` *sous-hérite de* `c` et `c` *sous-hérite de* `d` d'une contrainte de type descendante («downcast») qui vérifie `cc(o)` *sous-hérite de* `d` et `d` *sous-hérite de* `c`. L'affaiblissement de type d'une contrainte ascendante se doit juste de vérifier la relation de sous-typage, ce qui est le cas par la relation de sous-héritage, y compris par affaiblissements successifs de par la transitivité de la relation de sous-héritage. Le renforcement de type d'une contrainte descendante considère l'objet du type de sa classe d'origine puis l'affaiblit vers la classe cible. Il est à noter que du point de vue des types il n'est jamais dangereux de contraindre le type d'un objet vers le type de sa classe de construction. Cela garantit la correction de ces contraintes de types en particulier pour la contra-variance. La sémantique n'est pas forcément évidente pour le programmeur en présence d'héritage multiple et si l'on désire conserver l'emploi de l'opérateur d'affaiblissement actuel (`:>`). De plus ce mécanisme est totalement dynamique et ne satisfait pas forcément le programmeur Objective Caml qui apprécie la cadre du typage statique. Pour cela nous allons détailler la syntaxe et la sémantique de l'extension.

2.1. Syntaxe et sémantique

L'extension `coca-ml` ajoute au langage d'origine trois types de constructions :

- la relation de sous-héritage qui devient un nouveau champ des structures de classe;
- des prédicats vérifiant des propriétés de la classe de construction d'un objet par rapport à une classe particulière;
- des opérateurs de contraintes de type qui changent le type statique d'un objet sans modifier sa représentation.

2.1.1. Relation de sous-héritage

La relation de sous-héritage (mot clé `subinherit`) s'emploie de la même manière que la relation d'héritage (`inherit`) dans le corps de la définition d'une classe : `subinherit c` $arg_1 \ldots arg_n$.
La classe en cours de définition hérite de la classe `c` et son type est sous-type de c.

2.1.2. Prédicats sur les relations

On définit syntaxiquement trois prédicats qui vérifient que la classe de construction d'un objet est par rapport à la classe passée en paramètre respectivement «identique», «sous-classe» et en relation de «sous-héritage» comme présenté à la figure 2.

	`expr of c1`	`expr instanceof c1`	`expr subinstanceof c1`
Syntaxe	1/ `expr` est une expression 2/ `c1` est un identificateur de classe		
Typage	1/ type de `expr` : $<$ *nom_methode* : *id* $\rightarrow$ *bool* $>$ avec `nom_methode`		
	`check_id`	`check_instanceof_id`	`check_subinstanceof_id`
	2/ le type de l'instruction est *bool*		
Evaluation	1/ `expr` vaut `o`		
	2/ appel de la méthode		
	`check_id`	`check_instance_id`	`check_subinstanceof_id`
	sur `o` avec comme argument la clé de la classe `c1`		

Figure 2: Prédicats sur les relations

Ces prédicats ne changent pas le type d'un objet mais permettent de vérifier qu'une contrainte de type pourra s'appliquer sur un objet.

2.1.3. Instructions de contrainte de type

Les instructions de contrainte de type sur des objets ne modifient pas la valeur de l'objet sur lequel elles s'appliquent mais le forcent à être considéré d'un autre type. Elles sont présentées à la figure 3. En cas d'échec des vérifications des relations de classes à l'évaluation, une exception est déclenchée.

	`cast expr to d`	`upcast expr from c to d`	`downcast expr from c to d`
Syntaxe	1/ `expr` est une expression		
	2/ `d` est un identificateur de classe		
	3/	`c` est un identificateur de classe	
Typage	1/	*expr* : c	
	2/	$c \leq d$	$d \leq c$
	3/ `expr` est alors considéré du type d		
Evaluation	1/ `expr` vaut `o`		
	2/ `cc(o)` sous-hérite `d`		
	3/	`cc(o)` sous-hérite de `c`	
	4/	`c` sous-hérite de `d`	`d` sous-hérite de `c`

Figure 3: Instructions de contraintes de type

La première construction (`cast expr to d`) est la plus générale. Elle considère l'expression `expr` comme du type de la classe `d` et teste à l'exécution que la classe de construction de l'objet `o` résultat de l'évaluation de `expr` est bien en relation de sous-héritage avec la classe `d`. On part du postulat qu'un objet peut toujours être contraint sans risque vers le type de sa classe de construction. En effet comme l'objet a déjà été considéré de ce type il peut l'être de nouveau. Au typage de vérifier que le contexte d'emploi l'autorise. Une fois l'objet considéré du type de sa classe d'origine, il est possible de contraindre son type vers toute classe `x` dont la classe de construction sous-hérite. La seule contrainte sur le type de `expr` est d'être un type objet possédant la méthode `check_subinstanceof` (voir figure 1).

On obtient sur l'exemple suivant un objet `o` de classe de construction `z` est contraint une première fois vers la classe `a` puis ensuite vers la classe `m` :

```
cast (cast (new z) to a) to m;;
```

La figure 4 montre les trois cas simples possibles si `z` est sous-classe de `a` et `m`. On présente la construction des classes sous la forme de schémas UML simplifiés dans la mesure où les variables d'instance n'apparaissent pas. On différencie la relation d'héritage habituelle (flèche simple) de la relation de sous-héritage (flèche barrée du symbole <=) dans les schémas de relation. Le premier cas

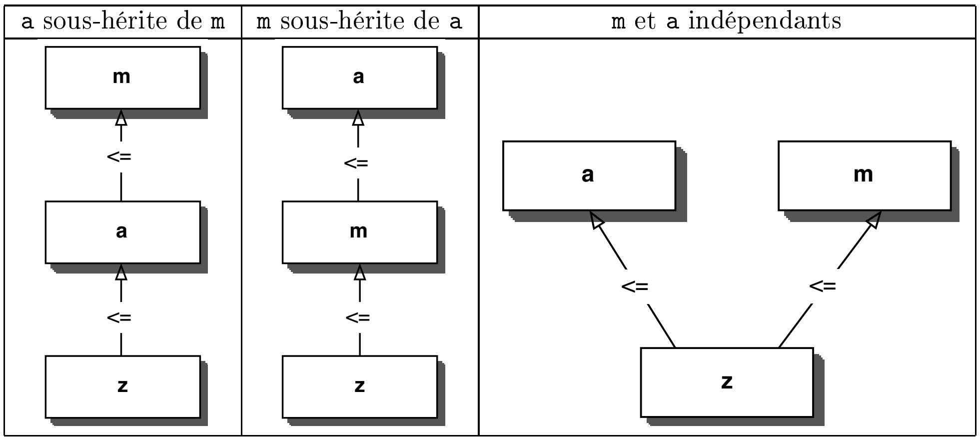


Figure 4: Trois graphes d'héritage possibles

correspond à deux «upcast» successifs; le deuxième cas est le plus classique, c'est un «upcast» suivi d'un «downcast» vers une classe intermédiaire. Le troisième cas est introduit par l'héritage multiple. Dans ces trois cas comme `z` sous-hérite de `a` et de `m` il n'y a pas de danger dans la contrainte.

Les deux autres constructions sont des restrictions de la première permettant plus de contrôle statique de types. Elles orientent le sens de la contrainte de type par rapport à l'arbre de sous-héritage. De plus le type de l'expression doit être compatible avec le type de la classe d'origine (`from`) de la contrainte. Enfin les classes d'origine et de destination (`to`) sont en relation de sous-typage cohérente par rapport au sens de la contrainte. ces restrictions permettent de détecter au typage des contraintes qui ne pourront jamais aboutir.

Si l'on avait écrit : `downcast (upcast (new z) from z to a) from a to m;;`, seul le deuxième cas de la figure 4 aurait été autorisé.

2.1.4. Compatibilité les opérateurs de contrainte et d'affaiblissement de types

Le langage Objective Caml possède deux opérateurs de contrainte de types :

- l'opérateur «`:`» : `(expr : t)` qui unifie le type de `expr` avec le type `t`
- l'opérateur d'affaiblissement «`:>`» : `( expr :> d )`[2] qui considère `expr` de type `d` si $type(expr) \leq d$.

Ces deux opérateurs ne tiennent absolument pas compte des relations d'héritage entre classes. Ainsi un objet pourra être contraint ou affaiblit vers le type d'une classe n'appartenant pas à sa hiérarchie de classes au sens de l'héritage.

Il est souhaitable de conserver la sûreté des nouvelles contraintes de types en présence d'objets contraints ou affaiblis par les opérateurs d'Objective Caml. Le cas général est celui d'un objet `o` de classe de construction `z` qui a été contraint vers le type de la classe `a` et qui va être contraint vers la classe `m` comme dans l'exemple suivant :

```
let o_z = new z;;
let o_a = ( o_z :> a);;
let o_m = cast o_a to m;;
```

La classe de construction de `o_z`, `o_a` et `o_m` est toujours `z` dans la mesure où un objet n'est pas modifié par une contrainte de type. On obtient les deux contraintes suivantes :

- $z \leq a$
- `z` *sous-hérite de* `m`

Les classes `a` et `m` ne sont pas forcément en relation de sous-typage dans la mesure où dans chacune d'elle peut apparaître une méthode n'existant pas dans l'autre. Néanmoins la contrainte de type reste sans danger si `z` *sous-hérite de* `m`. Certes la sémantique peut paraître un peu étrange dans le cas où la classe `a` n'est pas en relation d'héritage et de sous-typage avec `m`, mais ce qui compte est la relation de la classe de construction avec la classe de destination. On retrouve le cas 3 d'héritage multiple de la figure 4 pour la relation de sous-typage.

[2] On ne tient pas compte des différences sémantiques de la forme complète `(expr : c :> d)` dans la mesure où seule la contrainte de type vers `d` nous intéresse ici.

2.2. Exemples

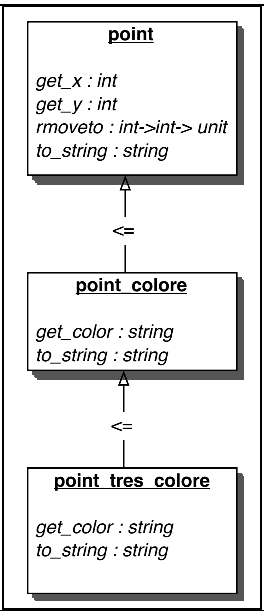
On présente trois petits exemples qui montre les nouvelles possibilités de cette extension au langage. Le premier construit une hiérarchie linéaire à trois niveaux. Le deuxième montre comment traduire des méthodes binaires. Le dernier détaille un héritage multiple.

2.2.1. Plusieurs niveaux d'héritage

La figure 5 reprend l'exemple inévitable sur l'héritage de points colorés. Il y a trois classes `point`, `point_colore` et `point_tres_colore` :

- `point_tres_colore` sous-hérite de `point_colore` qui sous-hérite elle-même de `point`;

Hiérarchie	Programme




```
# let p1 = new point 2 3;;
val p1 : point = <obj>
# let p2 = new point 3 4;;
val p2 : point = <obj>
# let pc1 = new point_colore 1 2 "bleu";;
val pc1 : point_colore = <obj>
# let ptc1 = new point_tres_colore 4 5 "vert";;
val ptc1 : point_tres_colore = <obj>

# let p3 = upcast pc1 from point_colore to point;;
val p3 : point = <obj>
# let p5 = upcast ptc1 from point_tres_colore to point;;
val p5 : point = <obj>

# let pc3 = downcast p3 from point to point_colore;;
val pc3 : point_colore = <obj>
# let ptc3  = downcast p5 from point to point_tres_colore;;
val ptc3 : point_tres_colore = <obj>
# let pc6 = downcast p5  from point to point_colore;;
val pc6 : point_colore = <obj>
```

Figure 5: Exemple avec plusieurs niveaux d'héritage

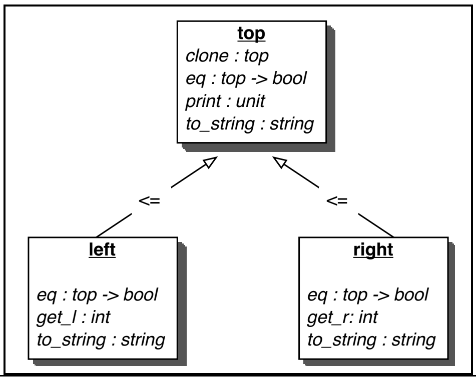
On construit les instances `p1` et `p2` de la classe `point`, `pc1` de `point_colore` puis `ptc1` de `point_tres_colore`. `p3` (respectivement `p5`) est une instance de `point_colore` (respectivement `point_tres_colore`) considérée comme du type `point`.

On obtient donc trois «downcast» possibles : `p3` est contraint en `point_colore`, `p5` est contraint en `point_tres_colore` et `p5` est contraint en `point_colore`.

2.2.2. Réécriture de méthodes binaires

Pour conserver dans l'héritage la relation de sous-typage, il est nécessaire de masquer les méthodes possédant un type fonctionnel où le type de `self` apparaît en position paramètre. La figure 6 construit une classe `top` ayant une méthode `eq` qui teste l'égalité entre deux instances de `top` et deux sous-classes `left` et `right` qui hérite de `top` et redéfinissent la méthode `eq`.
La définition de la classe `right` diffère de la classe `left` par une variable d'instance `r` (à la place de `l`), une méthode `get_r` (à la place de `get_l`) qui retourne la valeur de `r` et surtout par la redéfinition de

<table>
<tr><th>Hiérarchie</th><th>Programme</th></tr>
<tr><td>



</td><td>
<pre>
class virtual top =
  object(self)
    method virtual to_string : string
    method print = print_string (self#to_string)
    method clone = Oo.copy(self)
    method eq (a : top) = true
 end;;
class left (x : int) =
  object (self)
  subinherit top
  val l = x
  method get_l = l
  method to_string = string_of_int l
  method eq (x : top) =
    let xl = downcast x from top to left in
        l = (xl#get_l)
end;;
</pre>
</td></tr>
</table>

Figure 6: Exemple de premiers niveaux d'une hiérarchie

`eq` qui suppose recevoir une instance de `right` comme paramètre et effectue alors le «cast» vers `right` de l'argument. Le petit programme suivant crée deux instances de `left` (`l` et `l2`) et une de `right` (`r`) puis effectue deux appels de la méthode `eq` sur `l` avec respectivement `l2` et `r` comme paramètre contraint au type `top`.

```
let l = new left 10;;
let l2 = new left 10;;
let r =  new right 10;;
if l#eq(cast l2  to  top) then print_string "OK"
else print_string "PB";;
try
 ignore(l#eq (cast r to  top));
  print_string "PB"
with Cast.Cast_failure _ -> print_string "OK";;
```

Ce style de programmation s'apparente plus au style Java avec ses avantages et ses inconvénients. Il est à noter que la méthode `clone` est du type α (type de `self`) et ne casse pas la relation de sous-typage de la hiérarchie de classe dans la mesure où la méthode n'a pas un type fonctionnel.

2.2.3. Héritage multiple

Le troisième exemple (voir figure 7) montre un cas d'héritage multiple. La classe `abc` hérite des classes `a`, `b` et `c`. Ces trois dernières héritent de la classe `t`.

Le dernier calcul (`nr`) montre comment passer directement d'une instance de la classe `abc` vue comme du type `b` vers le type `a`. Cet exemple aurait pu s'écrire en deux étapes : `let nr = upcast (downcast b1 from b to abc) from abc to  a;;`.

2.3. Implantation

Cette implantation ne touche pas le compilateur de la distribution d'Objective Caml. `coca-ml` transforme un programme Objective Caml en un autre programme Objective Caml. Pour cela elle

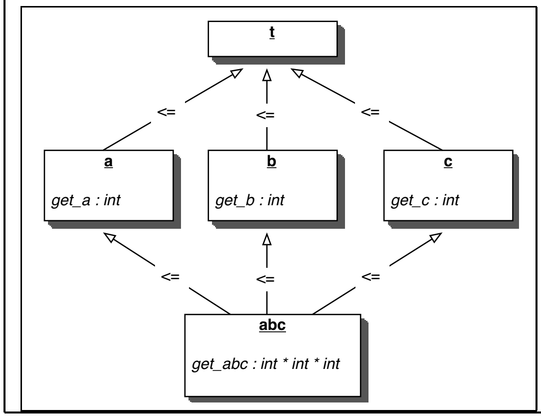
Hiérarchie	Programme
	# let abc1 = new abc 1;; val abc1 : abc = <obj> # let a1 = (upcast abc1 from abc to a);; val a1 : a = <obj> # let b1 = (upcast abc1 from abc to b);; val b1 : b = <obj> # downcast b1 from b to abc;; - : abc = <obj> # let t1 = upcast a1 from a to t;; val t1 : t = <obj> # downcast t1 from t to a;; - : a = <obj> # let r = downcast t1 from t to abc;; val r : abc = <obj> # let nr = cast b1 to a;; val nr : a = <obj>

Figure 7: Exemple d'héritage multiple

utilise l'excellent outil camlp4 qui permet de préprocesser un fichier source ou une phrase au toplevel par une nouvelle grammaire. Dans le cas présent la syntaxe du langage Objective Caml fournie avec camlp4 est modifiée.

coca-ml modifie principalement la déclaration de classes et ajoute six nouvelles instructions qui manipulent les noms de classe (of, instanceof, subinstanceof, cast, upcast et downcast) aux expressions du langage ainsi que la déclaration de sous-héritage (subinherit) dans une déclaration de classe.

L'implantation a d'autre part un fichier cast.ml qui contient la définition du type id et de l'exception Cast_failure ainsi que les fonctions d'appel des méthodes de vérification sur un objet.

2.3.1. Déclaration de classe

Une déclaration de classe c va maintenant créer une clé unique d'identification appelée key_c, de type abstrait *id*, et ajouter les méthodes publiques de la figure 1 à la classe.

Le choix d'implantation est de coder dans les objets les méthodes de vérification de propriétés d'héritage. Voici le code des méthodes principales ajoutées d'une classe z qui hérite d'une classe a et sous-hérite d'une classe m.

```
let ___z = (( Cast.make_id ()) : Cast.id );;
class z =
  object
    inherit a as ___super_0
    inherit m as ___super_1
    method ___check_instanceof_id (k : Cast.id) =
      (___z == k || ___super_1#___check_instanceof_id k)
                 || ___super_0#___check_instanceof_id k
    method ___check_subinstanceof_id (k : Cast.id) =
      (___z == k || ___super_1#___check_subinstanceof_id k)
    method ___check_subclass (k1 : Cast.id) (k2 : Cast.id) =
      if (___z == k1) then
        if (k1 == k2) then true else ___super_1#___check_subclass k2 k2
      else ___super_1#_check_subclass k1 k2
```

```
...
end;
fun ___x -> (___x : z :> m);;
```

Le préfixe ___ est ajouté à tout nouvel identificateur créé. Les noms des classes héritées sont conservés si ils existent ou engendrés dans le cas contraire.

Chaque classe possède une clé unique qui sera vérifiée physiquement dans les tests. En cas de sous-héritage une expression à l'extérieur de la classe vérifie la relation de sous-typage. Les classes simplement héritées apparaissent dans la méthode `check_instanceof_id` mais pas dans `check_subinstanceof_id`. Dans le cas où la classe n'hérite de personne, le corps de ces deux méthodes teste la clé d'identification de la classe qui a construit l'objet avec la clé passée en argument. Dans le cas où la classe hérite ou sous-hérite d'autres classes le corps de ces méthodes comprend les appels des méthodes de même nom des classes ancêtres sur l'objet lui-même avec la même clé comme argument. Comme la liaison des méthodes des classes ancêtres est statique, les tests s'effectueront bien avec la clé des classes ancêtres. La méthode `check_subclass` effectue une double remontée dans l'arbre de sous-héritage pour vérifier l'ordre de la classe de construction avec les deux classes passées en paramètres.

2.3.2. Nouvelles instructions

On distingue les instructions correspondant aux prédicats des relations des instructions effectuant effectivement une contrainte de types sur les objets.

- Les instructions `o of c1`, `o instanceof c1` et `o subinstanceof c1` sont traduites respectivement vers l'appel des méthodes `check_id`, `check_instanceof_id` et `check_subinstanceof_id` sur l'objet `o` avec la clé de `c1` passée en argument. En syntaxe simplifiée pour `o instanceof c1` cela donne : `o#check_instanceof_id(key_c1)`
- L'instruction `cast expr to c1` vérifie dynamiquement que la classe de construction de l'objet résultat de l'évaluation de `expr` est bien en relation de sous-héritage avec la classe `c1` :

```
let o = expr in
  if o#check_subinstance_of(key_c1)
  then ((Obj.magic o ) : c1)
  else raise (Cast_failure loc ...)
```

- Les instructions `upcast expr from c2 to c1` et `downcast expr from c2 to c1` vérifie statiquement que `expr` a le type de `c2`, que la relation de sous-typage est bien vérifiée entre les deux classes `c1` et `c2`, et enfin que la relation de sous-héritage est bien vérifiée (dynamiquement) entre ces deux classes. Voici le code engendré pour `downcast expr from c2 to c1` :

```
let o = ( expr : c1) in
  let x = ((Obj.magic o) : c1 ) in
    let _ = ( x :> c2) in
      if o#subclass key_c1 key_c2 then ( (Obj.magic o) : c2)
      else raise (Cast_failure ...)
```

Seules les instructions qui utilisent `magic` comportent un danger de transgresser le système de types. Le principal `magic` est celui qui considère l'objet du type de la classe de destination. Comme dans les trois instructions il y a la vérification que la classe de construction de l'objet est en relation de sous-héritage avec la classe destination, on garantit que l'objet peut être contraint vers ce type. Les autres instruction `magic` sont utilisées pour effectuer des restrictions statique en vérifiant la relation de sous-typage entre classe supposée d'origine et la classe destination.

2.3.3. Modules et classes

Les méthodes supplémentaires sont ajoutées aux signatures des modules. Elles ne doivent pas apparaître dans celles-ci. Seules les classes possédant une structure (`object .. end`) peuvent être traitées. La déclaration d'une classe en fonction d'une autre : `class b = a` ne possède pas assez d'informations pour être traitées et différencier les clés de ces classes. Les modules paramétrés peuvent être appliqués sur des structures possédant des déclarations de classes y compris dans le cas des modules locaux.

2.3.4. Limitations de l'extension

La principale limitation de cette extension est de ne pas accepter des classes paramétrées dans les «cast». Mais ce problème ne peut pas être résolu dans le cas général. Prenons par exemple une classe `'a d` qui hérite d'une classe `c`. Si on crée une instance `o` de type *int d*, il est bien sur possible de contraindre son type vers `c`. Si le type de cette instance est ensuite contraint vers la classe `d` quel est le type acceptable pour le paramètre de type du type `d`? Vu la technique utilisée pour l'implantation actuelle rien n'interdit d'écrire : `downcast o from c to float d` ce qui casse la sûreté du typage.

2.3.5. Coût de l'extension

Le choix de cette implantation a été de coder dans les définitions de classe les méthodes de vérification des héritages. La première conséquence est de faire grossir la taille du code par le code de ces méthodes supplémentaires. Il est à noter que la table des méthodes d'une classe est partagée par toutes les instances de cette classe. Les objets n'en seront pas plus gros pour autant. Seul le GC qui conserve la table des méthodes est légèrement ralenti dans la mesure où cette table occupe un peu plus d'espace. Il n'y a pas de déclarations statiques de ces tables pour traiter de manière uniforme les classes créées par des modules locaux. Donc tant que l'on ne s'en sert pas le coût de l'extension est extrêmement faible.

Si l'on s'en sert, le coût est alors proportionnel à la profondeur de la hiérarchie de classes. Il est à noter que l'appel d'une méthode ne profite pas des optimisations de l'application totale de la partie fonctionnelle du langage ce qui est dommage dans la mesure où les méthodes supplémentaires sont toujours appliquées avec tous leurs arguments. Cela est déjà le cas pour les paramètres fonctionnels des fonctions et reste d'un coût accepté.

3. Fabrique Abstraite

La fabrique abstraite fait partie des modèles créateur décrits dans [4]. Il permet de modéliser une interface pour créer des objets en autorisant à l'exécution le choix de la sous-classe à instancier. Il y a délégation de responsabilité. Le système ne connaît alors que les interfaces (ou classes abstraites) des objets qu'il utilise ne sachant pas quelle classe concrète les a créés. En particulier cela autorise de définir à l'exécution de nouvelles sous-classes implantant les interfaces pour la création de nouveaux objets.

Un exemple classique tiré de [4] est celui de la création de labyrinthes. La figure 8 montre la hiérarchie de classes adaptée d'une version Java que l'on peut trouver à l'URL suivante :
`http://dept-info.labri.u-bordeaux.fr/~griffaul/Enseignement/POO/Cours/`

Les classes `lieu`, `salle`, `mur`, `porte` et `fabriqueDedale` sont abstraites. Comme le programme est trop important pour être listé, on isole les quelques points délicats. La classe `lieu` possède une méthode abstraite `entrer` de type *unit* $\rightarrow$ *bool*. Une salle possède des cotés vus comme des instances de `lieu`.

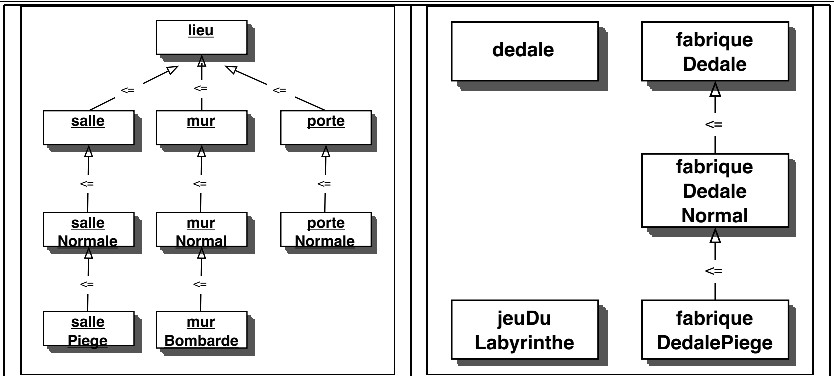


Figure 8: Hiérarchie pour fabrique de labyrinthes

Premier point, il y aura besoin de «downcast». Une salle pourra contenir des murs normaux ou des murs piégés et selon le cas la méthode `entrer` effectuera un travail particulier sur ceux-ci.

```
class sallePiege num =
object(self)
  subinherit salleNormale num as super
  val mutable alea = false
  initializer Random.init 10; if Random.float 1.0 < 0.25 then alea <- true
  method entrer () =
    if alea then (self#propagerExplosion(); alea <- false; alea)
    else true
method propagerExplosion () =
  for i=0 to 3 do
    try
      let mb = downcast cotes.(i) from lieu to murBombarde in          (* downcast *)
      mb#endommager()
    with Cast.Cast_failure (s,_,_) -> ()
  done
end;;
```

Deuxième point, il y aura besoin d'«upcast». La fabrique `fabriqueDedalePiege` possède une méthode de création de murs, qui seront en fait des murs bombardés.

```
class fabriqueDedalePiege () =
object
  method creeDedale () = new dedale ()
  method creeMur d s =
    upcast (new murBombarde d s) from murBombarde to mur              (* upcast *)
...
end;;
```

Troisième point, il y aura délégation. La classe `jeuDuLabyrinthe` possède une méthode `creeDedale` dont l'argument respecte l'interface `fabriqueDedale`. Ce que construit réellement la méthode `creeMur` de l'argument de cette méthode n'est pas connu.

```
class jeuDuLabyrinthe () =
object
```

```
  method creeDedale ( fabrique : fabriqueDedale)  =
    let unDedale = fabrique#creeDedale()  in
    for i=1 to 3 do
      let s = fabrique#creeSalle  i in
        unDedale#ajouteSalle(s);
        ignore(fabrique#creeMur est s);
        ignore(fabrique#creeMur ouest s);
...
end;;
```

Quatrième point, on peut ainsi construire des labyrinthes dont les éléments ont des comportements différents.

```
let main () =
  let fabriqueN = new fabriqueDedaleNormal () and fabriqueP = new fabriqueDedalePiege () in
  let jeuN = new jeuDuLabyrinthe () and jeuP = new jeuDuLabyrinthe () in
  ignore(jeuN#creeDedale(fabriqueN));
  ignore(jeuP#creeDedale(fabriqueP));
```

Sans le «downcast» apporté par `coca-ml` on peut implanter ce modèle en Objective Caml soit en utilisant le modèle des «visiteurs» au prix d'une certaine lourdeur et en devant modifier des classes ancêtres en cas d'ajout de nouvelles sous-classes, soit en masquant complètement les différences de type dans chaque hiérarchie de classes, c'est-à-dire sans ajouter de méthodes dans les sous-classes.

4. Comparaison avec d'autres langages objet

Objective Caml est l'un des rares langages statiquement typé polymorphe paramétrique avec inférence de types possédant une extension objet qui s'intègre bien au système de types. L'apport principal est en fait la liaison retardée pour l'appel de méthodes. En effet hors liaison retardée l'héritage pourrait être simulé par des modules paramétrés. De part la relation de sous-typage en présence de liaison retardée on obtient le polymorphisme d'inclusion des langages objets. Enfin le prix à payer pour l'inférence de types est l'absence de surcharge.

La notion de classe n'est pas présente en Objective Caml. Il en résulte une absence de hiérarchie de classes dans les bibliothèques du langage.

`coca-ml` est un premier pas pour intégrer la notion de classe dans le langage en ajoutant un certain niveau de «downcast» au langage. Ce «transtypage» existe dans la plupart des langages à objets ou possédant une extension objet, y compris dans des langages statiquement typés. On peut citer dans l'ordre d'apparition C++, Eiffel et Ada95. Ces langages présentés comme statiquement typés et acceptés tels quels autorisent respectivement l'affectation inverse et les contraintes de types vers des sous-types. C++ et Ada95 possèdent un modèle de surcharge et surtout ne laissent pas échapper le type de `self` dans le type des méthodes. La difficulté réside alors dans l'algorithme de résolution de la surcharge. Bien que Java soit présenté comme un langage statiquement typé, l'absence de classes paramétrées oblige les programmeurs à effectuer très souvent des «downcast» pour pouvoir écrire des bibliothèques réutilisables. Objective C, l'autre extension objet de C, s'apparente plus aux langages dynamiquement typé à la SmallTalk. Les langages dynamiquement typés ont l'obligation d'autoriser la vérification dynamique de types.

Le «downcast» au sens de l'héritage est un trait que l'on retrouve quasiment toujours dans les langages à objets et à classes. Certes il est souvent possible de s'en passer grâce aux classes paramétrées mais ce n'est pas souvent très naturel ni très pédagogique. Et surtout il existe certains cas impossibles à traiter. L'ajout du «downcast» en Objective Caml ouvre des possibilités de programmation objet plus large en particulier pour l'implantation de modèles de conception comme la fabrique de la section précédente et pour l'interfaçage avec des bibliothèques existantes.

Conclusion

L'utilisation de `camlp4` a été une agréable surprise de par la puissance de l'outil. Il permet de s'abstraire des évolutions fréquentes du langage tout en manipulant l'arbre de syntaxe. Il reste des choix d'implantation à effectuer pour `coca-ml` pour factoriser l'information sur les classes et pouvoir diminuer le nombre de méthodes supplémentaires.

Ce travail est un début d'intégration de la notion de classes dans Objective Caml. Cette intégration nous semble importante si l'on désire construire une véritable hiérarchie de classes dans le langage. Bien qu'il faille toujours considérer Objective Caml comme une extension objet et non pas comme un langage à objets, il est surprenant que, maintenant que la couche objet est stabilisée, il n'y ait que peu de bibliothèques de classes. Certes les modules paramétrés du langage sont en soit déjà puissants, la liaison retardée dans l'appel de méthodes permet une nouvelle structuration. Il semble intéressant de poursuivre ce travail vers une intégration plus poussée (à la mixin/unit [3]) entre classes et modules paramétrés en présence de «downcast». Pour cela une définition d'une classe ancêtre à toutes les classes peut s'avérer utile. Une autre voie de poursuite de ces travaux se porte sur les persistants objets. Si l'on désire pouvoir écrire et lire des objets stockés à l'extérieur d'un programme, il est indispensable d'effectuer un test dynamique pour vérifier la correction de la contrainte de types à la relecture. Un persistant objet ne contiendra pas le code de ces méthodes mais saura les retrouver par une identification unique de sa classe au moment de sa relecture. Ce type de communication de valeurs sans fermeture facilite la communication de valeurs structurées pour les applications réparties.

Remerciements

Ce travail est la conséquence de plusieurs discussions sur l'extension objet d'Objective Caml. Je tiens à remercier Bruno Pagano, Christian Queinnec et Didier Vaudène pour celles-ci ainsi que les rapporteurs dont les remarques ont influencé l'évolution de l'extension, et aussi Sylvain Baro et Jérôme Vouillon pour leur relecture.

Bibliographie

[1] Sylvain Boulmé, "Modules, Objets et Calcul Formel", In proceedings of "Journées Francophones des Langages Applicatifs", 1999.

[2] Emmanuel Chailloux, Pascal Manoury and Bruno Pagano, "Développement d'applications avec Objective Caml", O'Reilly, Apr, 2000, Paris.

[3] Robert Bruce Findler and Matthew Flatt, "Modular Object-Oriented Programming with Units and Mixins", In proceedings of "International Conference on Functional Programming", 1998.

[4] Erich Gamma, Richard Heml, Ralph Johnson and John Vlissides, «Design Patterns : catalogue de modèles de conception réutilisables», Vuibert Informatique, Jul, 1999.

[5] Xavier Leroy and all, «Objective Caml system 3.02 : Documentation and user's manual», Documentation INRIA disponible sur le site `http://caml.inria.fr`, 2001.

[6] Daniel de Rauglaudre, «camlp4 : Reference Manual» Documentation INRIA disponible sur le site `http://caml.inria.fr`, 2001.

[7] Didier Rémy and Jérôme Vouillon, «Objective ML: An effective object-oriented extension to ML», Journal «Theory And Practice of Object Systems», volume 4, pages 27–50, 1998.