Chapitre 3

MlObj

Ce chapitre donne des exemples réels d'interaction avec l'interpréteur MlObj. A quelques différences près, la syntaxe utilisée se rapproche beaucoup de celle d'Objective Caml. Ce chapitre se présente comme un tutoriel, introduisant les principales possibilités offertes par l'interpréteur sur des exemples. Le lecteur est renvoyé au mini-manuel de référence en Annexe A pour une présentation complète de MlObj, sa syntaxe, ses fonctions prédéfinies, etc...

Dans la suite, les zones encadrées indiquent des interactions avec l'interprète. Les commandes entrées à la suite du prompt `>` sont en caractères droits. Les réponses de l'interprète sont en italiques[10]. Lors de l'évaluation d'une expression, cette réponse prend la forme "*`- : type = valeur`*" où *`valeur`* est le résultat de l'évaluation de l'expression et *`type`* son (schéma de) type inféré par le système (ce type peut éventuellement se prolonger sur les lignes suivantes s'il est récursif ou s'il contient des contraintes sur les degrés, comme on va le voir sur des exemples plus loin). Lors de l'évaluation d'une déclaration globale `let (rec) ...`, la réponse prend la forme "*`val nom : type = valeur`*" où *`nom`* est le nom de la variable stockée dans l'environnement global.

3.1 Exemples fonctionnels simples

Commençons par une définition simple : la fonction identité. On notera au passage la syntaxe utilisée pour les types, ici le type de retour correspond au $\forall\alpha.\ \alpha^0 \to \alpha$ inféré par le système de types (les variables de type polymorphes sont introduites par une apostrophe `'` suivie d'une lettre minuscule, les degrés sont introduits par un accent circonflexe `^`).

```
> let id = fun x -> x;;
val id : 'a ^0-> 'a = <fun>
```

Deuxième exemple simple : la fonction factorielle (on notera que le test d'égalité s'écrit `==` et non `=` comme en Caml).

```
> let rec fact n = if n == 0 then 1 else n * fact (n - 1);;
val fact : int ^0-> int = <fun>
> fact 5;;
- : int = 120
```

[10] Merci à Jens Klöcker pour son script `caml-tex` et son package `caml.sty`, que j'ai pu adapter à mes besoins, afin d'automatiser l'évaluation et la mise en page.

Un peu de manipulation de références : cette partie est strictement identique au code utilisé en Objective Caml.

```
> let x = ref 0;;
val x : int ref = ref 0
> x := 1;;
- : unit = ()
> !x;;
- : int = 1
```

Définissons récursivement l'itérateur `map` sur les listes, à partir des fonctions prédéfinies `null`, `hd` et `tl` :

```
> let map f =
    let rec loop l = if null l then [] else f (hd l) :: loop (tl l)
    in loop;;
val map : ('a ^b-> 'c) ^d-> 'a list ^0-> 'c list = <fun>

> map (fun x -> x * x) [1, 2, 3];;
- : int list = [1, 4, 9]
```

A seule fin de vérifier que l'interpréteur génère bien des types récursifs quelconques, voici le typage pour $\Delta = \lambda x\ xx$. Les types de la forme `Xn` sont des abréviations (pas forcément récursives), dont le contenu est donné après le mot-clé `where`.

```
> let delta x = x x;;
val delta : X1 ^0-> 'a
where X1 = X1 ^b-> 'a
 = <fun>
```

3.2 Degrés et contraintes

Afin de mettre en évidence les contraintes sur les degrés, voici les fonctions application et composition de fonctions :

```
> fun f x -> f x;;
- : ('a ^b-> 'c) ^d-> 'a ^e-> 'c
where ^e<=^b  = <fun>
> fun f g x -> f (g x);;
- : ('a ^b-> 'c) ^d-> ('e ^f-> 'a) ^g-> 'e ^h-> 'c
where ^h<=^b ^h<=^f  = <fun>
```

On notera que l'inférence de types produit le type $(a^b \to c)^d \to a^e \to c$ dans le premier cas avec la contrainte supplémentaire $e \leqslant b$, contrainte indiquée par l'interpréteur par le mot-clé `where`.

Les instructions suivantes illustrent la mise en oeuvre des techniques exposées à l'Exemple 2.1. La fonction polymorphe `ignore` nous servira d'exemple (le caractère `_` désigne un identifieur muet dont on ne se sert pas dans la suite). Dans le premier cas, `fun x -> ignore x`, le degré `^b` est bien rendu polymorphe, et la contrainte `^b` $\leqslant$ `^c` où `^c` est produite par l'instanciation de `ignore` a bien été supprimée.

```
> let ignore _ = ();;
val ignore : 'a ^b-> unit = <fun>
> fun x -> ignore x;;
- : 'a ^b-> unit = <fun>
```

Dans le deuxième cas, on crée une référence `r` sur `ignore`, dont toutes les variables sont évidemment monomorphes (comme en OCaml, une variable monomorphe est précédée par un underscore `_`). Puis, on évalue `fun x -> !r x`. Comme prévu, le degré `^_b` n'est cette fois-ci pas généralisé, et on conserve la contrainte `^_b` $\leqslant$ `^_c` associée. Une consultation de `r` nous permet de constater que la même contrainte a été rajoutée (dans les deux réponses de l'interpréteur, le `^_b` de l'une correspond au `^_c` de l'autre).

```
> let r = ref ignore;;
val r : ('_a ^_b-> unit) ref = ref <fun>
> fun x -> !r x;;
- : '_a ^_b-> unit
where ^_b<=^_c  = <fun>
> r;;
- : ('_a ^_b-> unit) ref
where ^_c<=^_b  = ref <fun>
```

3.3 Enregistrements

Les deux exemples suivants montrent respectivement la syntaxe utilisée pour un enregistrement clos composé de trois champs (deux entiers et une chaîne de caractères), et pour une fonction qui prend un enregistrement et l'étend avec un nouveau champ `l`.

```
> let e = { posx = 0, posy = 1, couleur = "bleu" };;
val e : { couleur : string, posx : int, posy : int } = { couleur =
"bleu", posx = 0, posy = 1 }
> let etend = fun enr -> { enr, l = 0 };;
val etend : 'A ^0-> { 'A, l : int }
where 'A :: {l}  = <fun>
```

Dans le deuxième cas, on notera qu'une variable de rangée (ici polymorphe) est introduite par une apostrophe `'` suivie d'une lettre majuscule, et que la liste de ses champs non autorisés est donnée à la suite du mot-clé `where`.

Les deux exemples suivants montrent respectivement la sélection et la restriction de champs :

```
> e.couleur;;
- : string = "bleu"
> (etend e) \ posy;;
- : { couleur : string, l : int, posx : int } = { couleur = "bleu", l =
0, posx = 0 }
```

Passons maintenant à des exemples plus concrets. La fonction `point` suivante définit une “classe” pour les points, contenant un champ position modifiable et une méthode pour bouger le point. Elle est paramétrée par une position initiale `x`.

```
> let point x self = { pos = ref x,
                       move y = self.pos := !self.pos + y };;
val point : 'a ^b-> { 'A, pos : int ref } ^c-> { move : int ^0-> unit,
pos : 'a ref }
where 'A :: {pos}  = <fun>
```

Afin de créer des instances de cette classe, nous avons besoin de la fonction `fix`, qui calcule le point fixe d'une fonction :

```
> let fix = fun f -> let rec x = f x in x;;
val fix : ('a ^1-> 'a) ^0-> 'a = <fun>
```

On notera, conformément à nos attentes, le degré `^1` dans le type. Ainsi, par unification, ceci nous assure que `fix` ne pourra être appliquée qu'à des fonctions qui protègent leur argument.

Il est maintenant possible de créer un nouveau point `p` et de le bouger :

```
> let p = fix (point 4);;
val p : { move : int ^0-> unit, pos : int ref } = { move = <fun>, pos =
ref 4 }
> p.move 2;;
- : unit = ()
> p;;
- : { move : int ^0-> unit, pos : int ref } = { move = <fun>, pos = ref
6 }
```

Les commandes suivantes créent une nouvelle classe qui hérite de `point` et qui lui ajoute une méthode pour remettre la position à 0, et l'instancient :

```
> let clearable_point x self = { point x self, clear _ = self.pos := 0 };;
val clearable_point : 'a ^b-> { 'A, pos : int ref } ^c-> { clear : 'd
^e-> unit, move : int ^0-> unit, pos : 'a ref }
where 'A :: {pos}  = <fun>
> let q = fix (clearable_point 5);;
val q : { clear : '_a ^_b-> unit, move : int ^0-> unit, pos : int ref }
= { clear = <fun>, move = <fun>, pos = ref 5 }
> q.clear ();;
- : unit = ()
> q;;
- : { clear : unit ^_a-> unit, move : int ^0-> unit, pos : int ref } = {
clear = <fun>, move = <fun>, pos = ref 0 }
```

Etant donné que dans notre langage, les “classes” n'existent pas en tant que telles, mais sont en fait des valeurs fonctionnelles standards, il est possible de définir directement une fonction qui prend une super-classe et qui renvoie la classe héritière avec un champ `color` en plus. Par exemple, on peut l'appliquer aussi bien à la classe `point` qu'à sa sous-classe `clearable_point` :

```
> let color c super = fun x self -> { super x self, color = ref c };;
val color : 'a ^b-> ('c ^d-> 'e ^f-> 'A) ^g-> 'c ^h-> 'e ^i-> { 'A,
color : 'a ref }
where 'A :: {color} ^i<=^f  = <fun>
> fix ((color 256 point) 0);;
- : { color : int ref, move : int ^0-> unit, pos : int ref } = { color =
ref 256, move = <fun>, pos = ref 0 }
> fix ((color 256 clearable_point) 0);;
- : { clear : '_a ^_b-> unit, color : int ref, move : int ^0-> unit, pos
: int ref } = { clear = <fun>, color = ref 256, move = <fun>, pos = ref
0 }
```

En revanche, on peut vérifier qu'il n'est pas possible d'ajouter deux fois le champ `color`, le typage nous le signale par un échec :

```
> color 123 (color 256 point);;
Unable to unify 'A
where 'A :: {color}
and { color : int ref, move : int ^0-> unit, pos : 'a ref }
```

Revenons un moment sur la classe `point`. Le champ position `pos` est explicitement déclaré comme une référence. Est-il possible de s'en passer ? La nouvelle classe `point` ci-dessous suit cette idée. En contrepartie, puisqu'il n'est plus possible de modifier directement la *valeur* du champ, la méthode `move` renvoie `self` dans lequel on remplace le champ `pos` par sa nouvelle valeur.

```
> let point x self = { pos = x,
                       move d = { self, pos <- self.pos + d }};;
val point : 'a ^b-> { 'A, pos : int } ^c-> { move : int ^0-> { 'A, pos :
int }, pos : 'a }
where 'A :: {pos}  = <fun>
> (((fix (point 0)).move 3).move 2).pos;;
- : int = 2
```

Nous invitons le lecteur à comprendre pourquoi le résultat de l'évaluation est 2 et non pas 5 comme on le voudrait. Au moment du point fixe par `fix`, la valeur `self` dans `self.pos` est liée à l'enregistrement créé. Ceci a pour conséquence que lors du deuxième appel `move`, la valeur `self.pos` fait toujours référence à l'enregistrement initial, et non pas à celui retourné par le premier appel de `move`. Autrement dit, il faut prendre garde qu'avec ce style de programmation, le paramètre `self` ne désigne pas l'objet courant sur lequel on appelle la méthode comme c'est le cas dans les langages à objets usuels, mais l'objet au moment de sa création (et il serait donc préférable d'utiliser un autre nom que `self`...).

Nous reviendrons sur ce point au chapitre suivant, en apportant une solution dans le cadre plus général des mixins.

3.4 Simulation de paires

On pourra noter que notre langage ne propose ni les paires ni les tuples de façon générale. Cependant, il est facile de les encoder à l'aide des enregistrements. Les définitions suivantes donnent les fonctions nécessaires, à savoir la création de paires et la consultation de ses composantes.

```
> let pair x y = { fst = x, snd = y };;
val pair : 'a ^b-> 'c ^0-> { fst : 'a, snd : 'c } = <fun>
> let fst p = p.fst;;
val fst : { 'A, fst : 'a } ^0-> 'a
where 'A :: {fst}  = <fun>
> let snd p = p.snd;;
val snd : { 'A, snd : 'a } ^0-> 'a
where 'A :: {snd}  = <fun>
> fst (pair 1 2);;
- : int = 1
```

On pourra objecter à la vue de leur type que ces fonctions `fst` et `snd` peuvent prendre en argument bien plus que des véritables paires : en fait n'importe quel enregistrement qui contient le champ `fst` ou `snd`. Par exemple :

```
> snd { x = 0, snd = "boo", f x = x + 1 };;
- : string = "boo"
```

Si on désire limiter l'usage de `fst` et `snd` aux paires créées par `pair`, il est possible de limiter le type de leur argument en l'indiquant explicitement :

```
> let fst (p : { fst:'a, snd:'b }) = p.fst;;
val fst : { fst : 'a, snd : 'b } ^0-> 'a = <fun>
> let snd (p : { fst:'a, snd:'b }) = p.snd;;
val snd : { fst : 'a, snd : 'b } ^0-> 'b = <fun>
```

Les deux fonctions ont alors le type souhaité, et l'exemple ci-dessus n'est plus possible [11]. De façon générale, on peut apporter des contraintes sur le type d'un identifieur ou d'une expression entière par `(x : type)` ou `(expr : type)`. Au moment de l'inférence, le type le plus générique inféré est alors unifié avec `type`.

[11] Une autre façon de simuler les paires serait de construire une classe avec comme méthodes les projections.

Chapitre 4

Mixins et exemples avancés

Dans ce Chapitre, nous introduisons une couche syntaxique supplémentaire à notre langage, formellement puis sur des exemples, afin de démontrer sa puissance expressive, aussi bien opérationnellement que du point de vue du typage. Cet ajout implémente un langage pour la programmation orientée objet à base de *mixins*. Grossièrement, les mixins peuvent être vus comme des définitions de classes paramétrées par leur super-classe ; ils furent introduits pour la première fois dans des langages orientés objets basés sur LISP dans les années 80 (on pourra également trouver quelques similarités avec les classes paramétrées d'EIFFEL [Mey86] ou les "classes virtuelles" de BETA [MP89]). Dans [BC90], les mixins sont présentés comme la "brique de base" pour l'héritage, point de vue que nous ne pouvons que partager ; des travaux plus récents s'y sont également à nouveau intéressé [AZ98, FKF98, BPSM99a].

4.1 Présentation générale

Nous commençons par donner quelques éléments de terminologie.

Un *générateur* est une fonction $Gen = \lambda\ self.\{\ldots\}$ qui prend en paramètre $self$ et retourne un enregistrement (ce que nous appelions "classes" dans le chapitre précédent étaient donc des générateurs paramétrés par des valeurs initiales). Informellement, un *objet* est alors le point fixe d'un générateur (voir [CP89]) : (`fix` *Gen*), autrement dit un enregistrement récursif (comme c'est le cas dans [Sny86, Car88, CHC94, Wan94]). Pour qu'un générateur puisse générer un objet par point fixe, il faut que le type de $self$ soit unifiable avec le type de retour du générateur, et en outre que ce générateur soit une fonction protectrice pour $self$.

Un *mixin* est une fonction qui transforme un générateur en un autre générateur. Elle est donc de la forme $\lambda g \lambda s\ \{\ldots\}$. En règle générale, l'action d'un mixin sera de modifier le générateur donné en paramètre en lui ajoutant, retranchant ou modifiant des champs.

Un *champ* de mixin est un champ au sens habituel des enregistrements, contenant une valeur soit non modifiable, soit modifiable (c'est-à-dire une référence sur cette valeur).

Une *méthode* est également un champ au sens des enregistrements. Afin de respecter le système de types, il faudra que ce soit toujours une fonction. Pour cette raison, une méthode prendra toujours un paramètre unit `()` comme premier argument (une telle fonction est appelée un "thunk"), afin de geler l'évaluation de la méthode lors du calcul de point fixe. L'appel de méthode devra en conséquence passer `()` en premier argument pour dégeler la fonction. De plus, deux paramètres particuliers sont accessibles dans le corps des méthodes : `super` et `self` (ce ne sont pas des mots-clés, mais bien des identifieurs soumis à l'α-conversion). Lors de l'évaluation de la méthode, `self` sera lié à l'objet courant en tant qu'instance de la classe courante, et `super` à l'objet courant en tant qu'instance de la super-classe (ceci est nécessaire dans le cas où l'on redéfinit une méthode qui a besoin d'appeler son ancienne version, voir les exemples plus loin).

Une *classe* est une fonction paramétrée par des paramètres d'instance (typiquement des valeurs initiales), et retournant un mixin. Elle a donc la forme générale

$$\lambda x_1 \ldots x_n \; \lambda g \; \lambda s \; \{\ldots \text{champs} \ldots \text{méthodes} \ldots\}$$

Pour construire une *instance* d'une classe C, il faut lui fournir, outre les paramètres d'instance, un générateur de départ g. Dans le cas standard, il s'agira du générateur vide $\lambda s \; \{\}$ (mais pour d'autres usages, on peut imaginer d'autres générateurs de base ; ceci permet par exemple de définir des champs ou des méthodes qui apparaissent systématiquement dans les objets créés). La forme générale d'une instance de C est donc :

$$\texttt{fix} \; (C \; N_1 \cdots N_n \; (\lambda s \; \{\}))$$

où les N_i sont les paramètres d'instance.

Pour cette raison, nous définirons la fonction `new` suivante :

$$\texttt{new} = \lambda m \;\; \texttt{fix} \; (m \; (\lambda s \; \{\}))$$

et on écrira

$$\texttt{new} \; (C \; N_1 \cdots N_n)$$

pour la création d'une instance.

4.2 Sucre syntaxique pour les mixins

Nous allons mettre en oeuvre le modèle des mixins par l'intermédiaire de constructions syntaxiques spécifiques, qui sont automatiquement transformées dans le langage de base des enregistrements.

Un mixin est délimité par les mots-clés `mixin` et `end`. Son contenu est une suite d'instructions introduites chacune par un mot-clé spécifique. Il n'y a pas de délimiteur (les mots-clés suffisent). Les instructions sont parcourues dans l'ordre où elles sont données, et modifient le mixin au fur et à mesure. Intuitivement, si g est le générateur donné en paramètre au mixin (au mot-clé `mixin`), ces instructions font des modifications sur g et retournent un générateur final g' (au niveau du `end`), qui est la valeur de retour du mixin.

Nous allons maintenant donner la fonction de traduction $[\![M]\!]$ pour un mixin M. On prendra garde au fait que, dans les définitions qui suivent, M représente une séquence éventuellement vide d'expressions.

$$[\![\textbf{mixin} \; M \; \textbf{end}]\!] = [\![M]\!]$$

Commençons par la traduction des champs. Les mots-clés `var` et `cst` introduisent des champs modifiables et constants respectivement :

$$[\![M \; \textbf{var} \; l = N]\!] = \lambda g \lambda s \; \{[\![M]\!] \; g \; s, \; l = \; \textbf{ref} \; [\![N]\!]\}$$
$$[\![M \; \textbf{cst} \; l = N]\!] = \lambda g \lambda s \; \{[\![M]\!] \; g \; s, \; l = [\![N]\!]\}$$

Ici, puisque M désigne le mixin construit jusqu'ici, $[\![M]\!] \; g \; s$ est l'enregistrement qui lui correspond pour un générateur g et un paramètre self s donnés. Il suffit donc d'étendre cet enregistrement avec le champ correspondant (par le biais d'une référence pour les champs modifiables) et de rendre le mixin correspondant.

Les méthodes sont introduites par le mot-clé `meth` et existent sous deux formes : définition d'une nouvelle méthode avec = et redéfinition d'une méthode existante avec ←. Le nom de la méthode l est obligatoirement suivi de deux identifieurs x et y (dont le choix est libre), correspondant à `super` et `self` respectivement.

$$\begin{array}{ll} [\![M \textbf{ meth } l(y,x) & = N]\!] = \lambda g \lambda s\ (\textbf{let } z = [\![M]\!]\ g\ s \textbf{ in } \{z, l = \lambda(u : \mathit{unit})\ (\lambda y \lambda x\ [\![N]\!])\ z\ s\}) \\ [\![M \textbf{ meth } l(y,x) & \leftarrow N]\!] = \lambda g \lambda s\ (\textbf{let } z = [\![M]\!]\ g\ s \textbf{ in } \{z \setminus l, l = \lambda(u : \mathit{unit})\ (\lambda y \lambda x\ [\![N]\!])\ z\ s\}) \end{array}$$

Comme précédemment, $z = [\![M]\!]\ g\ s$ désigne l'enregistrement créé jusqu'ici, que l'on étend par un nouveau champ l. Ce champ est un thunk (une fonction dont le paramètre est forcé à *unit*) dont le corps est $(\lambda y \lambda x\ [\![N]\!])\ z\ s$, autrement dit le corps de la méthode $[\![N]\!]$, dans lequel y et x sont liés à z et s respectivement. Au moment de la création de l'objet, s sera bien l'objet lui-même, et z l'enregistrement construit jusqu'à ce point. Pour une redéfinition de méthode, la traduction est la même, sauf que l'ancien champ l est d'abord enlevé de z avant de lui être rajouté.

La présence du thunk nous oblige à prendre pour l'appel de méthode une syntaxe différente de celle de la sélection de champs, puisqu'il faut lui passer une valeur de type *unit* :

$$[\![M\#l]\!] = ([\![M]\!].l)\ ()$$

Le mot-clé `inherit` introduit l'*héritage* : toutes les transformations induites par le mixin N sont reprises à cet endroit.

$$[\![M \textbf{ inherit } N]\!] = \lambda g \lambda s\ ([\![N]\!]\ ([\![M]\!]\ g)\ s)$$

Plus précisément, le générateur g passé en paramètre du mixin-résultat est d'abord donné à $[\![M]\!]$, qui nous renvoie un générateur g', que l'on passe ensuite à $[\![N]\!]$. On notera que cette traduction se transforme en $[\![N]\!] \circ [\![M]\!]$ par η-expansion. L'héritage se résume donc à de la composition de mixins ! Nous gardons cependant la première traduction car cette deuxième version produit un type trop générique.

Les mots-clés `without` et `rename ... as` permettent respectivement de supprimer et de renommer un champ ou une méthode d'un mixin. Au vu des cas précédents, leur traduction ne pose pas de problème :

$$\begin{array}{l} [\![M \textbf{ without } l]\!] = \lambda g \lambda s\ ([\![M]\!]\ g\ s) \setminus l \\ [\![M \textbf{ rename } l \textbf{ as } l']\!] = \lambda g \lambda s\ (\textbf{let } x = [\![M]\!]\ g\ s \textbf{ in } \{x \setminus l, l' = x.l\}) \end{array}$$

Enfin, pour être complet, il reste à définir le mixin “identité” qui sert de point de départ :

$$[\![\ ]\!] = \lambda g \lambda s\ (g\ s)$$

Un mot sur le typage On pourra vérifier sur les règles de traduction ci-dessus qu'un mixin $[\![M]\!]$ protège toujours son deuxième argument s (à condition que ce soit le cas de N dans M **inherit** N) ; ceci pourrait être faux pour les méthodes s'il n'y avait pas le thunk. Il est donc toujours possible de calculer un objet par point fixe, au moins en ce qui concerne les degrés (l'encodage des mixins a été conçu en ce sens).

En revanche, tout mixin n'est pas forcément instanciable : il faut (et il suffit) pour cela que le type du deuxième argument s soit unifiable avec le type de retour du mixin lorsque le générateur donné en premier argument a pour type de retour l'enregistrement vide $\{\}$. La condition nécessaire et suffisante pour pouvoir appliquer `new` sera donc “lisible” sur le type du mixin (en pratique, la lecture des types générés par les mixins commence à être difficile ; mais ils restent indispensables en cas d'erreur). Nous verrons dans les exemples quels mixins sont instanciables et lesquels ne le sont pas (à juste titre). Ces derniers n'ont alors qu'un rôle de "modifieurs" pour d'autres mixins (un peu à la manière des classes virtuelles dans les langages à objets).

4.3 Exemples dans MLOBJ

La sur-couche syntaxique et la traduction présentée ci-dessus sont intégrées telles qu'elles dans MLOBJ. Nous allons maintenant présenter quelques exemples d'utilisation et tenter de donner la mesure de la puissance expressive apportée par ce formalisme.

Avant toute chose, il nous faut définir la fonction `new` qui comme `fix` ne fait pas partie intégrante du langage[12] :

```
> let new m = fix (m (fun s -> {}));;
val new : (('a ^b-> { }) ^c-> 'd ^1-> 'd) ^0-> 'd = <fun>
```

Commençons par notre exemple préféré, exprimé comme un mixin :

```
> let point x =
   mixin
    var pos = x
    meth move(super,self) d = self.pos := !self.pos + d
   end;;
val point : 'a ^b-> (X1 ^c-> 'A) ^d-> X1 ^e-> { 'A, move : unit ^f-> int
^0-> unit, pos : 'a ref }
where X1 = { 'B, pos : int ref }
'B :: {pos} 'A :: {pos,move} ^e<=^c  = <fun>
```

Et un exemple d'utilisation :

```
> let p = new (point 5);;
val p : { move : unit ^_a-> int ^0-> unit, pos : int ref } = { move =
<fun>, pos = ref 5 }
> p#move 2;;
- : unit = ()
```

Le mixin suivant définit une classe contenant une couleur et une méthode pour la modifier :

```
> let coloring c =
   mixin
    var color = c
    meth paint(super,self) c' = self.color := c'
   end;;
val coloring : 'a ^b-> (X1 ^c-> 'A) ^d-> X1 ^e-> { 'A, color : 'a ref,
paint : unit ^f-> 'g ^h-> unit }
where X1 = { 'B, color : 'g ref }
'B :: {color} 'A :: {color,paint} ^e<=^c  = <fun>
```

[12] La raison est que d'autres variantes sont possibles (voir par exemple `new_init` plus loin), et que MLOBJ étant un langage expérimental, nous préférons laisser un maximum de libertés.

A partir de ces deux mixins, il est possible de construire une classe “point colorée” par héritage multiple :

```
> let colorPoint x c =
   mixin
    inherit point x
    inherit coloring c
   end;;
val colorPoint : 'a ^b-> 'c ^d-> (X1 ^e-> 'A) ^f-> X1 ^g-> { 'A, color :
'c ref, move : unit ^h-> int ^0-> unit, paint : unit ^i-> 'j ^k-> unit,
pos : 'a ref }
where X1 = { 'B, color : 'j ref, pos : int ref }
'B :: {color,pos} 'A :: {color,paint,pos,move} ^g<=^e  = <fun>
```

A noter que les deux instructions `inherit` pourraient ici être interverties sans changer le résultat.

```
> let cp = new (colorPoint 42 255);;
val cp : { color : int ref, move : unit ^_a-> int ^0-> unit, paint :
unit ^_b-> int ^_c-> unit, pos : int ref } = { color = ref 255, move =
<fun>, paint = <fun>, pos = ref 42 }
> cp#move 3;;
- : unit = ()
> cp#paint 128;;
- : unit = ()
```

Le mixin `reset` suivant définit uniquement une méthode qui réinitialise le champ `pos`.

```
> let reset =
   mixin
    meth reset(super,self) = self.pos := 0
   end;;
val reset : (X1 ^a-> 'A) ^b-> X1 ^c-> { 'A, reset : unit ^d-> unit }
where X1 = { 'B, pos : int ref }
'B :: {pos} 'A :: {reset} ^c<=^a  = <fun>
```

A noter que ce mixin est bien typable, alors qu'il ne contient pas de champ `pos` défini... En revanche, comme nous l'avons expliqué plus haut et comme on peut le constater sur son type, il n'est pas instanciable. La seule utilisation que nous pourrons en faire afin d'arriver à une classe instanciable sera donc d'en hériter dans un autre mixin qui aura un champ `pos` défini par ailleurs. Néanmoins, l'intérêt est que nous pourrons le réutiliser dans des situations différentes sans avoir à récrire le code de la méthode `reset`. Par exemple, les deux classes suivantes respectent les conditions nécessaires et suffisantes décrites plus haut, et sont donc instanciables :

```
> let resetPoint x =
   mixin
    inherit point x
    inherit reset
   end;;
val resetPoint : 'a ^b-> (X1 ^c-> 'A) ^d-> X1 ^e-> { 'A, move : unit
^f-> int ^0-> unit, pos : 'a ref, reset : unit ^g-> unit }
where X1 = { 'B, pos : int ref }
'B :: {pos} 'A :: {reset,pos,move} ^e<=^c  = <fun>

> let resetColorPoint x c =
   mixin
    inherit colorPoint x c
    inherit reset
   end;;
val resetColorPoint : 'a ^b-> 'c ^d-> (X1 ^e-> 'A) ^f-> X1 ^g-> { 'A,
color : 'c ref, move : unit ^h-> int ^0-> unit, paint : unit ^i-> 'j
^k-> unit, pos : 'a ref, reset : unit ^l-> unit }
where X1 = { 'B, color : 'j ref, pos : int ref }
'B :: {color,pos} 'A :: {reset,color,paint,pos,move} ^g<=^e  = <fun>
```

On peut aller encore plus loin et définir un mixin qui fait appel à des méthodes définies dans une hypothétique super-classe :

```
> let wrapper =
   mixin
    meth reset(super,self) d <- super#reset; super#paint d
   end;;
val wrapper : ('a ^b-> { 'A, paint : X1, reset : unit ^c-> unit }) ^d->
'a ^e-> { 'A, paint : X1, reset : unit ^f-> 'g ^0-> 'h }
where X1 = unit ^i-> 'g ^j-> 'h
'A :: {paint,reset} ^e<=^b  = <fun>
```

Grâce au typage, ce mixin ne pourra être repris dans une classe instanciable que s'il est dans l'environnement adéquat, à savoir avec une super-classe qui contient des méthodes `reset` et `paint`. Par exemple, puisque `resetColorPoint` vérifie cette condition, on pourra instancier la classe suivante :

```
> let resetColorPoint2 x c =
   mixin
    inherit resetColorPoint x c
    inherit wrapper
   end;;
val resetColorPoint2 : 'a ^b-> 'c ^d-> (X1 ^e-> 'A) ^f-> X1 ^g-> { 'A,
color : 'c ref, move : unit ^h-> int ^0-> unit, paint : unit ^i-> 'j
^k-> unit, pos : 'a ref, reset : unit ^l-> 'j ^0-> unit }
where X1 = { 'B, color : 'j ref, pos : int ref }
'B :: {color,pos} 'A :: {reset,color,paint,pos,move} ^g<=^e  = <fun>
```

Nous allons maintenant montrer comment il est possible d'exécuter automatiquement une procédure au moment de l'instanciation. La technique est simple : il suffit de rajouter une méthode `init` aux classes pour lesquelles on souhaite ce comportement et de les instancier par la fonction `new_init` ci-dessous. Cette fonction crée l'objet de façon standard par la méthode `new`, appelle sa méthode `init`, puis supprime cette méthode avant de renvoyer l'objet (ce qui évite de la réexécuter par erreur).

```
> let new_init m =
   let o = new m in
   o#init; o\init;;
val new_init : (('a ^b-> { }) ^c-> X1 ^1-> X1) ^0-> 'A
where X1 = { 'A, init : unit ^d-> unit }
'A :: {init}  = <fun>
```

En guise d'exemple, nous donnons une variante de la classe `point` qui imprime un message à l'écran lors de la création d'un nouveau point :

```
> let printable_point x =
   mixin
    inherit point x
    meth print(super,self) = print_int !self.pos
    meth init(super,self) = print_string "new point at "; self#print;
                            print_newline()
   end;;
val printable_point : 'a ^b-> (X1 ^c-> 'A) ^d-> X1 ^e-> { 'A, init :
unit ^f-> unit, move : unit ^g-> int ^0-> unit, pos : 'a ref, print :
unit ^h-> unit }
where X1 = { 'B, pos : int ref, print : unit ^i-> unit }
'B :: {print,pos} 'A :: {pos,move,print,init} ^e<=^c  = <fun>
> let p = new_init (printable_point 17);;
new point at 17
val p : { move : unit ^_a-> int ^0-> unit, pos : int ref, print : unit
^_b-> unit } = { move = <fun>, pos = ref 17, print = <fun> }
```

On vérifie que l'objet ne contient plus de méthode `init` au final. Si on souhaite utiliser des initialiseurs pour des classes qui héritent l'une de l'autre, il faut prendre garde à ce que lors de la redéfinition des méthodes `init`, on commence par appeler la méthode `init` de la super-classe :

```
meth init(super,self) <- super#init ; ...
```

On peut même imaginer de généraliser cette technique à toutes les classes et faire en sorte que tous les mixins possèdent une méthode `init`. Il suffit pour cela d'utiliser la fonction suivante à la place de `new`. Cette fonction fournit comme générateur de base à tous les mixins un générateur qui renvoie un enregistrement avec une fonction `init` qui ne fait rien :

```
> let new_init m =
   let o = fix (m (fun s -> { init _ = () })) in
   o#init; o\init;;
val new_init : (('a ^b-> { init : 'c ^d-> unit }) ^e-> X1 ^1-> X1) ^0->
'A
where X1 = { 'A, init : unit ^f-> unit }
'A :: {init}  = <fun>
```

Les mixins associés au types récursifs permettent d'imaginer encore de nombreuses combinaisons, par exemple des mixins dont une méthode retourneraient `self` ou même un autre mixin. Tout cela est rendu possible par le fait que les mixins ont le statut de simple valeurs.

```
> let m =
   mixin
    meth another(super,self) = mixin
                                meth myself(super,self) = self
                               end
   end;;
val m : ('a ^b-> 'A) ^c-> 'a ^d-> { 'A, another : unit ^e-> ('f ^g-> 'B)
^h-> 'f ^i-> { 'B, myself : unit ^j-> 'f } }
where 'B :: {myself} 'A :: {another} ^d<=^b ^i<=^g  = <fun>
> (new ((new m)#another))#myself;;
- : X1
where X1 = { myself : unit ^_a-> X1 }
 = { myself = <fun> }
```

A l'aide d'une classe récursive, il est également possible de définir une méthode pour cloner un objet. Dans l'exemple suivant, la méthode `clone` retourne une instance fraîche en gardant la même position pour le point. Cette classe contient aussi une méthode `colorless` qui supprime toutes les informations relatives à la couleur (afin de les cacher par exemple).

```
> let rec clonablePoint x c =
   mixin
    inherit colorPoint x c
    meth clone(z,s) = new (clonablePoint !s.pos !s.color)
    meth colorless(z,s) = s \ color \ paint
   end;;
val clonablePoint : int ^a-> 'b ^c-> (X1 ^1-> { }) ^d-> X1 ^1-> X1
where X4 = unit ^e-> int ^0-> unit
X3 = unit ^f-> { clone : X2, colorless : X3, move : X4, pos : int ref }
X2 = unit ^g-> X1
X1 = { clone : X2, color : 'b ref, colorless : X3, move : X4, paint :
unit ^h-> 'b ^i-> unit, pos : int ref }
 = <fun>
```

Comme prévu, une modification sur le point de départ n'affecte pas son clone (et inversement) :

```
> let p = new (clonablePoint 10 256) in
   let q = p#clone in
    p#move 5; q#colorless;;
- : X1
where X4 = unit ^_a-> int ^0-> unit
X3 = unit ^_b-> X1
X2 = unit ^_c-> { clone : X2, color : int ref, colorless : X3, move :
X4, paint : unit ^_d-> int ^_e-> unit, pos : int ref }
X1 = { clone : X2, colorless : X3, move : X4, pos : int ref }
 = { clone = <fun>, colorless = <fun>, move = <fun>, pos = ref 10 }
```

Il est également possible de programmer des interactions du type “sujet/observateur”. Dans l’exemple suivant, une fenêtre de la classe `window` est déclarée en tant que sujet d’un observateur `o`. Lorsqu’elle est déplacée, elle notifie l’observateur en appelant sa méthode `moved` et en se passant elle-même en paramètre. Ici, le comportement de l’observateur (en l’occurrence un gestionnaire de fenêtres) sera d’appeler une méthode de la fenêtre pour qu’elle se redessine.

```
> let subject o =
   mixin
    meth notify(z,s) y = y o s
   end;;
val subject : 'a ^b-> ('c ^d-> 'A) ^e-> 'c ^f-> { 'A, notify : unit ^g->
('a ^h-> 'c ^i-> 'j) ^0-> 'j }
where 'A :: {notify} ^f<=^d  = <fun>

> let window x o =
   mixin
    inherit point x
    inherit subject o
    meth move(z,s) d <- z#move d; s#notify (fun o -> o#moved)
    meth draw(z,s) = print_string "Je me redessine en x = ";
                     print_int !s.pos; print_newline ()
   end;;
val window : 'a ^b-> 'c ^d-> (X1 ^e-> 'A) ^f-> X1 ^g-> { 'A, draw : unit
^h-> unit, move : unit ^i-> int ^0-> 'j, notify : unit ^k-> ('c ^l-> {
'B, notify : unit ^m-> ({ 'C, moved : unit ^n-> 'o } ^0-> 'o) ^p-> 'j,
pos : int ref } ^q-> 'r) ^0-> 'r, pos : 'a ref }
where X1 = { 'B, notify : unit ^m-> ({ 'C, moved : unit ^n-> 'o } ^0->
'o) ^p-> 'j, pos : int ref }
'C :: {moved} 'B :: {notify,pos} 'A :: {notify,pos,move,draw} ^g<=^e  =
<fun>
```

```
> let manager =
   mixin
    meth moved(z,s) w = w#draw
   end;;
val manager : ('a ^b-> 'A) ^c-> 'a ^d-> { 'A, moved : unit ^e-> { 'B,
draw : unit ^f-> 'g } ^0-> 'g }
where 'B :: {draw} 'A :: {moved} ^d<=^b  = <fun>

> let m = new manager in
   let w = new (window 42 m) in
    w#move 10;;
Je me redessine en x = 52
- : unit = ()
```

En guise de dernier exemple, nous allons traiter la question des méthodes binaires. La classe suivante définit des points qui possèdent une méthode permettant de les comparer entre eux.

```
> let pointEq x =
   mixin
    inherit point x
    meth eq(super,self) p = !self.pos == !p.pos
   end;;
val pointEq : 'a ^b-> (X1 ^c-> 'A) ^d-> X1 ^e-> { 'A, eq : unit ^f-> {
'B, pos : int ref } ^0-> bool, move : unit ^g-> int ^0-> unit, pos : 'a
ref }
where X1 = { 'C, pos : int ref }
'C :: {pos} 'B :: {pos} 'A :: {pos,move,eq} ^e<=^c  = <fun>
> let p = new (pointEq 0);;
val p : { eq : unit ^_a-> { '_A, pos : int ref } ^0-> bool, move : unit
^_b-> int ^0-> unit, pos : int ref }
where '_A :: {pos}  = { eq = <fun>, move = <fun>, pos = ref 0 }
> let q = new (pointEq 1);;
val q : { eq : unit ^_a-> { '_A, pos : int ref } ^0-> bool, move : unit
^_b-> int ^0-> unit, pos : int ref }
where '_A :: {pos}  = { eq = <fun>, move = <fun>, pos = ref 1 }
> p#eq q;;
- : bool = false
```

Jusqu'ici tout se passe bien. On peut cependant constater que le type de `p` (et `q`) a été précisé suite à l'appel de la méthode `eq`. Ceci est dû au fait que `p` n'étant pas polymorphe, toutes ses variables restent monomorphes. Et l'appel à `eq` en instancie certaines.

```
> p;;
- : { eq : unit ^_a-> { eq : unit ^_b-> { '_A, pos : int ref } ^0->
bool, move : unit ^_c-> int ^0-> unit, pos : int ref } ^0-> bool, move :
unit ^_d-> int ^0-> unit, pos : int ref }
where '_A :: {pos}  = { eq = <fun>, move = <fun>, pos = ref 0 }
```

Si on appelle p sur lui-même, on va encore préciser certaines des variables dans son type.

```
> p#eq p;;
- : bool = true
> p;;
- : { eq : unit ^_a-> { eq : unit ^_a-> { eq : X1, move : X2, pos : int
ref } ^0-> bool, move : unit ^_b-> int ^0-> unit, pos : int ref } ^0->
bool, move : unit ^_b-> int ^0-> unit, pos : int ref }
where X2 = unit ^_b-> int ^0-> unit
X1 = unit ^_a-> { eq : X1, move : X2, pos : int ref } ^0-> bool
 = { eq = <fun>, move = <fun>, pos = ref 0 }
```

En revanche, alors que la classe `colorPoint` hérite de la classe `point` et contient aussi un champ `pos`, il n'est pas possible de l'utiliser comme argument de la méthode `eq` :

```
> p#eq (new (colorPoint 1 256));;
Unable to unify { eq : unit ^a-> { eq : X1, move : X2, pos : int ref }
^0-> bool, move : unit ^b-> int ^0-> unit, pos : int ref }
where X2 = unit ^b-> int ^0-> unit
X1 = unit ^a-> { eq : X1, move : X2, pos : int ref } ^0-> bool

and { color : int ref, move : unit ^c-> int ^0-> unit, paint : unit ^d->
int ^e-> unit, pos : int ref }
```

Par contre, si on recrée un nouveau point, l'instruction précédente devient possible. Mais par contraste, il n'est plus possible de comparer le point à lui-même...

```
> let p = new (pointEq 0);;
val p : { eq : unit ^_a-> { '_A, pos : int ref } ^0-> bool, move : unit
^_b-> int ^0-> unit, pos : int ref }
where '_A :: {pos}  = { eq = <fun>, move = <fun>, pos = ref 0 }
> p#eq (new (colorPoint 1 256));;
- : bool = false
> p#eq p;;
Unable to unify { color : int ref, move : unit ^a-> int ^0-> unit, paint
: unit ^b-> int ^c-> unit, pos : int ref }
and { eq : unit ^d-> { color : int ref, move : unit ^a-> int ^0-> unit,
paint : unit ^b-> int ^c-> unit, pos : int ref } ^0-> bool, move : unit
^e-> int ^0-> unit, pos : int ref }
```

En conclusion, les méthodes binaires sont utilisables, mais elles interagissent mal avec l'héritage [13]. Plus précisément, en MLOBJ, pour une méthode binaire d'un objet `o` donné, il est nécessaire de n'utiliser en argument que des instances issues d'une même classe (qui n'est pas forcément celle de `o`). Il est probable que l'utilisation de méthodes polymorphes dans les objets permettraient de résoudre cette question (entre autres), mais leur mise en oeuvre dans le système de types avec degrés n'est pas évidente... Une solution intermédiaire serait d'utiliser des types avec intersection...

[13]Pour une étude complète des problèmes posés par les méthodes binaires dans les langages de programmation orientés objets et quelques techniques permettant d'y remédier, on pourra se reporter avec intérêt à [BCC+95].

Annexe A

Mini-manuel de référence de MLOBJ

L'interpréteur est téléchargeable à l'adresse suivante :

`http://www-sop.inria.fr/mimosa/Pascal.Zimmer/mlobj.html`

A.1 Ligne de commande

La ligne de commande de MLOBJ prend la forme suivante :

```
mlobj <options> [fichier.mlobj] ...
```

Les options possibles sont les suivantes :
- `-v` : donne davantage d'informations en cas d'erreur de typage
- `-no_typing` : désactive la phase de typage ; l'interpréteur devient un simple évaluateur (aucune garantie de sûreté n'est plus donnée bien entendu)
- `-help` ou `--help` : affiche les informations d'usage pour la ligne de commande

Les fichiers donnés en ligne de commande sont chargés au lancement de l'interpréteur, dans l'ordre où ils sont donnés, comme s'ils avaient été tapés interactivement. En cas d'erreur de syntaxe ou de typage, la lecture du fichier est stoppée et l'on passe au suivant. Lorsque tous ont été lus, l'interpréteur continue en mode interactif normal.

A.2 Analyse lexicale

Identifieurs Un identifieur est une suite non vide de caractères, commençant par une lettre ou un underscore `_`, suivi de lettres, chiffres, underscores ou apostrophes `'`.

Par exception, les mots-clés suivants sont réservés et ne peuvent pas être utilisés comme identifieurs : `let`, `rec`, `and`, `type`, `unsafe`, `use`, `in`, `fun`, `if`, `then`, `else`, `mixin`, `end`, `var`, `cst`, `meth`, `inherit`, `as`, `without`, `rename`.

Labels Les labels ont la même syntaxe que les identifieurs.

Entiers Un entier est une suite non vide de chiffres, éventuellement précédée du caractère `-`.

Chaîne de caractères Une chaîne de caractères (*string*) est une suite de caractères commençant et terminant par des guillemets `"`. Les caractères intermédiaires peuvent être quelconques, excepté les guillemets.

Variables de type et de rangée Une variable de type commence par une apostrophe `'`, suivie d'une lettre minuscule, puis d'une suite quelconque de lettres, chiffres, underscores et apostrophes. Une variable de rangée commence par une apostrophe `'`, suivie d'une lettre majuscule, puis d'une suite quelconque de lettres, chiffres, underscores et apostrophes.

Espaces, etc... Les caractères espaces, retour à la ligne et tabulations sont ignorés.

Commentaires On utilise une syntaxe à la C. Les caractères suivant `//` sont ignorés jusqu'au prochain retour à la ligne. Les caractères entre `/*` et `*/` sont également ignorés (attention à ne pas utiliser ces séquences dans une chaîne de caractères).

A.3 Analyse syntaxique

Cette section décrit la grammaire utilisée par MLOBJ.

Commandes et instructions Une commande à l'interpréteur se termine toujours par `;;`. C'est seulement lorsque ce marqueur de fin est lu que l'évaluation de l'instruction est lancée. Si une expression seule est donnée, elle est typée, puis évaluée et son résultat est imprimé. Les instructions introduites par `let` et `let rec` sont traitées de la même manière, mais leur(s) résultat(s) est(sont) stocké(s) dans l'environnement global et peuvent être accédé(s) par la suite. Les instructions `#type` et `#unsafe` ne réalisent respectivement que le typage et l'évaluation de l'expression donnée. Ces commandes sont fournies dans un but de mise au point et peuvent bien évidemment conduire au plantage de l'interpréteur en ce qui concerne `#unsafe`. On notera qu'afin de garantir au maximum l'intégrité des évaluations suivantes, seules des expressions peuvent être évaluées par `#type` et `#unsafe` (pas de stockage possible, sauf à contourner la restriction en effectuant une affectation à une référence bien entendu). Enfin, l'instruction `#use` suivie d'un nom de fichier permet de charger un fichier source dans l'interpréteur.

$$
\begin{array}{rcl}
\textit{command} & ::= & \textit{instr}\ \texttt{;;} \\
\textit{instr} & ::= & \texttt{let}\ \textit{let\_binding} \\
& | & \texttt{let rec}\ \textit{let\_binding} \\
& | & \textit{expr} \\
& | & \texttt{\#type}\ \textit{expr} \\
& | & \texttt{\#unsafe}\ \textit{expr} \\
& | & \texttt{\#use}\ \textit{string}
\end{array}
$$

Liaisons A la suite d'un `let` ou `let rec`, on trouve des liaisons (au moins une) de la forme `f x1 x2 ... = e` séparées par le mot-clé `and`. `f` désigne le nom de la variable pour la liaison, `x1`, `x2`... sont les éventuels arguments de la fonction définie (voir plus loin les remarques sur le sucre syntaxique). Ces arguments sont soit des variables simples, soit des variables avec une information de type (qui sera unifiée avec le type inféré lors du typage).

$$
\begin{array}{rcl}
\textit{let\_binding} & ::= & \textit{ident params}\ \texttt{=}\ \textit{expr} \\
& | & \textit{ident params}\ \texttt{=}\ \textit{expr}\ \texttt{and}\ \textit{let\_binding} \\
\textit{params} & ::= & \textit{typable\_id params} \\
& | & \\
\textit{typable\_id} & ::= & \textit{ident} \\
& | & \texttt{(}\ \textit{ident}\ \texttt{:}\ \textit{typexpr}\ \texttt{)}
\end{array}
$$

Expressions La grammaire des expressions ci-dessous ne demande pas de commentaires particuliers ; elle est identique ou similaire à celle de la plupart des langages fonctionnels. De même que pour les liaisons, les arguments d'une fonction `fun` peuvent contenir des informations de typage. Les mixins (voir plus bas) sont introduits par les mots-clés `mixin` et `end`.

$$
\begin{array}{rcl}
expr & ::= & \texttt{let}\ let\_binding\ \texttt{in}\ expr \\
 & | & \texttt{let rec}\ let\_binding\ \texttt{in}\ expr \\
 & | & \texttt{fun}\ typable\_id\ params\ \texttt{->}\ expr \\
 & | & \texttt{if}\ expr\ \texttt{then}\ expr\ \texttt{else}\ expr \\
 & | & \texttt{mixin}\ mixin\ \texttt{end} \\
 & | & expr\ infix\_op\ expr \\
 & | & appl \\
appl & ::= & appl\ term \\
 & | & term
\end{array}
$$

Termes De même, la grammaire des termes n'appelle que peu de remarques. Les enregistrements sont introduits par des accolades. Il est possible de donner une information de type sous la forme `( e : tau )`. Le type inféré sera alors unifié avec le type donné ; ceci permet dans certains cas de limiter le type inféré automatiquement lorsqu'il est trop générique par rapport à l'usage que l'on souhaite faire de l'expression. Les opérateurs infixes, ainsi que `!`, peuvent être accédés en tant que fonction préfixe classique (par exemple, `(+)` pour la fonction addition). Les listes sont introduites par des crochets et leurs éléments sont séparés par des virgules. Enfin, on retrouve les opérateurs de sélection et de restriction d'un champ pour les enregistrements et les mixins.

$$
\begin{array}{rcl}
term & ::= & ident \\
 & | & entier \\
 & | & string \\
 & | & \texttt{\{}\ record\ \texttt{\}} \\
 & | & \texttt{(}\ expr\ \texttt{)} \\
 & | & \texttt{(}\ expr\ \texttt{:}\ typexpr\ \texttt{)} \\
 & | & \texttt{(}\ op\ \texttt{)} \\
 & | & \texttt{[}\ list\ \texttt{]} \\
 & | & term\ \texttt{.}\ label \\
 & | & term\ \texttt{\#}\ label \\
 & | & term\ \backslash\ label \\
 & | & \texttt{!}\ term \\
list & ::= & expr\ \texttt{,}\ list \\
 & | & expr \\
 & | &
\end{array}
$$

Opérateurs et relations infixes La grammaire comprend une batterie d'opérateurs classiques pour un langage de programmation. Ces opérateurs sont détaillés à la Section A.6.

$$
\begin{array}{rcl}
infix\_op & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{:=} \mid \texttt{==} \mid \texttt{<>} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{::} \mid \texttt{;} \mid \texttt{@} \mid \texttt{\^{}} \\
op & ::= & infix\_op \mid \texttt{!} \mid
\end{array}
$$

Enregistrements Les enregistrements sont une suite de définitions de champs de la forme `label x1 x2 ... = e` ou de modifications de champs de la forme `label x1 x2 ... <- e`. Optionnellement, en tête, peut se trouver une expression qui désigne l'enregistrement dont on part pour réaliser les extensions et modifications (sinon on part implicitement d'un enregistrement vide).

record	::=	*label params* = *expr*
	|	*expr* , *label params* = *expr*
	|	*record* , *label params* = *expr*
	|	*expr* , *label params* <- *expr*
	|	*record* , *label params* <- *expr*
	|	

Mixins Les mixins sont une suite d'instructions introduites par un jeu de mots-clés différents. Le mot-clé `inherit` introduit une expression s'évaluant en un mixin dont on va hériter tous les champs. Le mot-clé `var` introduit une nouvelle variable modifiable. Le mot-clé `cst` introduit un champ non modifiable. La séquence `meth label(super,self) ...` désigne une méthode de nom `label` ; les arguments `super` et `self` désigne respectivement le mixin avant modification et le mixin courant et sont destinés à être utilisés dans le corps de la méthode. Cette méthode est soit déclarée nouvelle (par =), soit comme remplaçant d'une méthode existante de même nom (par <-). Le mot-clé `without` supprime un champ. Enfin, le mot-clé `rename` permet le renommage d'un champ existant.

mixin	::=	*mixin* `inherit` *appl*
	|	*mixin* `var` *label params* = *expr*
	|	*mixin* `cst` *label params* = *expr*
	|	*mixin* `meth` *label* (*typable_id* , *typable_id*) *params* = *expr*
	|	*mixin* `meth` *label* (*typable_id* , *typable_id*) *params* <- *expr*
	|	*mixin* `without` *label*
	|	*mixin* `rename` *label* `as` *label*
	|	

Expressions de type Une expression de type est soit une constante, soit une variable de type, soit un type fonction, soit le type d'un enregistrement, soit un type liste, soit un type référence. Le type d'un enregistrement commence optionnellement par une variable de rangée et est suivi des noms de champs avec leurs types respectifs. On notera que cette grammaire ne fait pas mention des degrés pour les fonctions et qu'il n'est donc pas possible en l'état de l'interpréteur de "forcer" ce paramètre (pour chaque type fonctionnel entré, l'analyseur syntaxique introduit implicitement une variable de degré fraîche ; nous n'avons pas trouvé d'utilisation pratique qui justifierait de proposer cette fonctionnalité).

typexpr	::=	`int` | `bool` | `unit` | `string`
	|	*typ_ident*
	|	*typexpr* -> *typexpr*
	|	(*typexpr*)
	|	{ *typrecord* }
	|	{ *row_ident* , *typrecord* }
	|	*typexpr* `ref`
	|	*typexpr* `list`
typrecord	::=	*label* : *typexpr*
	|	*label* : *typexpr* , *typrecord*
	|	

A.4 Sucre syntaxique

Comme c'est le cas en Objective Caml, il est possible de définir des fonctions de manière "compacte". Ainsi, les instructions

```
fun x1 ... xn -> expr
```

```
let (rec) f x1 ... xn = expr in ...
```

```
{ ... , f x1 ... xn = expr , ... }
```

sont respectivement équivalentes à

```
fun x1 -> ... -> fun xn -> expr
```

```
let (rec) f = fun x1 -> ... -> fun xn -> expr in ...
```

```
{ ... , f = fun x1 -> ... fun xn -> expr , ... }
```

Ce "sucre syntaxique" est admis devant tous les opérateurs `=`, `<-` et `->`, autrement dit dans les définitions globales et locales introduites par `let (rec)`, dans les fonctions introduites par `fun`, dans les champs d'enregistrements extensibles, dans les champs modifiables et non modifiables de mixins, et dans leurs définitions de méthodes.

Par ailleurs, la redéfinition de champs pour un enregistrement est directement convertie en son équivalent restriction, puis extension. Par exemple :

```
{ expr, foo x <- x * x }
```

est transformé en :

```
{ expr \ foo, foo x = x * x }
```

Enfin, un autre ensemble de transformations syntaxiques a lieu en ce qui concerne les mixins, afin de les traduire dans le sous-langage composé seulement du cœur fonctionnel, des références et des enregistrements. Cette transformation est décrite à la Section 4.2.

A.5 Priorité des opérateurs

Cette table montre la priorité relative et l'associativité pour les opérateurs du langage, en commençant par la priorité la plus élevée.

Construction ou opérateur	Associativité
`\`	gauche
`. #`	gauche
`!`	droite
`* / %`	gauche
`+ -`	gauche
`::`	droite
`@ ^`	droite
`== <> < <= > >=`	aucune
`&&`	gauche
`||`	gauche
`:=`	droite
`else`	aucune
`;`	droite
`->`	droite
`in`	aucune

A.6 Librairie prédéfinie

Voici la liste des fonctions et des constantes prédéfinies dans le système, avec leurs types. Tous les opérateurs binaires sont accessibles en tant que fonction sous leur forme préfixée en les notant entre parenthèses (par exemple, `(+)` pour la fonction addition).

A.6.1 Unit

```
() : unit
```

La valeur unit (void).

A.6.2 Arithmétique

```
(+) : int ^a-> int ^0-> int
(-) : int ^a-> int ^0-> int
(*) : int ^a-> int ^0-> int
(/) : int ^a-> int ^0-> int
(%) : int ^a-> int ^0-> int
```

Les opérateurs standards pour l'addition, la soustraction, la multiplication, la division et le modulo.

```
(<) : int ^a-> int ^0-> bool
(<=) : int ^a-> int ^0-> bool
(>) : int ^a-> int ^0-> bool
(>=) : int ^a-> int ^0-> bool
(==) : int ^a-> int ^0-> bool
(<>) : int ^a-> int ^0-> bool
```

Les relations de comparaison entre entiers.

A.6.3 Booléens

`true : bool`
`false : bool`
Les constantes booléennes.

`(&&) : bool ^a-> bool ^0-> bool`
`(||) : bool ^a-> bool ^0-> bool`
`not : bool ^0-> bool`
Les opérateurs booléens standards (conjonction, disjonction et négation).

A.6.4 Références

`ref : 'a ^b-> 'a ref`
Crée une nouvelle référence.

`( !) : 'a ref ^0-> 'a`
Donne la valeur d'une référence.

`(:=) : 'a ref ^b-> 'a ^c-> unit`
Change la valeur d'une référence.

A.6.5 Listes

`[] : 'a list`
La liste vide.

`(::) : 'a ^b-> 'a list ^0-> 'a list`
L'opérateur cons.

`hd : 'a list ^0-> 'a`
`tl : 'a list ^0-> 'a list`
Les opérateurs tête et queue.

`null : 'a list ^0-> bool`
Teste si une liste est vide.

`(@) : 'a list ^b-> 'a list ^0-> 'a list`
Concaténation de listes.

A.6.6 Autres

`(^) : string ^a-> string ^0-> string`
Concaténation de chaînes de caractères.

`( ;) : unit ^b-> 'a ^0-> 'a`
Séquence.

```
print_int : int ^0-> unit
print_string : string ^0-> unit
print_newline : unit ^0-> unit
```

Ecriture d'un entier, d'une chaîne de caractères ou d'un retour à la ligne sur la sortie standard.

```
quit : unit ^0-> 'a
```

Quitte la session.

A.7 Sémantique big-step d'origine

Pour mémoire et pour information, nous reproduisons ici la sémantique "big-step" utilisée dans les premières versions de l'interpréteur. D'un point de vue performance, celle-ci n'est pas plus lente que la machine abstraite du Chapitre 5, car elle peut être implémentée directement de façon récursive terminale...

A.7.1 Syntaxe

La syntaxe est à peu de choses près la même que celle utilisée dans la version actuelle de MLOBJ (il manque les listes et les liaisons multiples).

$$
\begin{array}{rcl}
M, N & ::= & M N \\
& | & \texttt{let x = } M \texttt{ in } N \\
& | & \texttt{let rec x = } M \texttt{ in } N \\
& | & \texttt{fun x } \rightarrow M \\
& | & \texttt{x} \\
& | & i \in Int \\
& | & s \in String \\
& | & \texttt{if } M \texttt{ then } N \texttt{ else } N' \\
& | & \texttt{\{ \}} \\
& | & \texttt{\{ } M \texttt{, l = } N \texttt{ \}} \\
& | & M\texttt{.l} \\
& | & M\backslash\texttt{l}
\end{array}
$$

A.7.2 Valeurs à l'exécution

On suppose qu'on a un ensemble *Location* de *cases mémoires*. Celles-ci serviront aussi bien pour les références créées par `ref` que pour les valeurs récursives. L'ensemble des valeurs retournées par l'évaluation d'un terme est alors (où $(\mathtt{x}, M, E)$ désigne une clôture) :

$$
\begin{array}{rcl}
V & ::= & (\mathtt{x}, M, E) \\
 & | & u \in \mathit{Location} \\
 & | & \{l_1 = V_1, \ldots, l_n = V_n\} \text{ sans ordre et } l_i \neq l_j \\
 & | & \mathit{rec}(u) \\
 & | & \mathtt{()} \in \mathit{Unit} \\
 & | & i \in \mathit{Int} \\
 & | & b \in \mathit{Bool} \\
 & | & s \in \mathit{String} \\
 & | & \mathit{prim}(f) \text{ avec } f : V \times S \to \mathit{Res} \times S \\
S & ::= & \mathit{Location} \to V \cup \{\bullet\} \\
E & ::= & \emptyset \\
 & | & E, \mathtt{x} = V \\
\mathit{Res} & ::= & V \\
 & | & \mathtt{err}
\end{array}
$$

Une *mémoire* S est une application partielle de domaine fini, d'un ensemble infini de cases mémoires vers l'ensemble des valeurs (ou la valeur spéciale $\bullet$ (pointeur null), utilisée seulement pour `let rec`). Il est donc toujours possible de trouver une case mémoire $u \notin \mathit{dom}(S)$ (on dira que u est *fraîche*). La mise à jour d'une mémoire est définie par :

$$
S\{^u/_V\} = S' \quad \text{où} \quad \left\{ \begin{array}{ll} S'(v) = S(v) & \text{pour } v \neq u \\ S'(u) = V & \end{array} \right.
$$

avec $\mathit{dom}(S') = \mathit{dom}(S)$ si $u \in \mathit{dom}(S)$ et $\mathit{dom}(S') = \mathit{dom}(S) \cup \{u\}$ sinon.

Un environnement E est une suite de liaisons variables-valeurs. Enfin, un résultat Res est soit une valeur, soit `err` pour signifier une erreur d'exécution.

A.7.3 Sémantique big-step

La forme générale est :

$$E; S \vdash M \Rightarrow \mathit{Res}, S'$$

i.e. l'évaluation du terme M dans l'environnement E et la mémoire S, produit le résultat Res pour une mémoire modifiée S'.

Les règles de réduction sont listées ci-dessous ; nous ne les détaillerons pas.

Recherche dans l'environnement

$$\frac{}{E, \mathtt{x} = V; S \vdash \mathtt{x} \Rightarrow V, S}$$

$$\frac{E; S \vdash \mathtt{x} \Rightarrow V, S' \qquad \mathtt{x} \neq \mathtt{y}}{E, \mathtt{y} = V'; S \vdash \mathtt{x} \Rightarrow V, S'}$$

$$\frac{E; S \vdash \mathtt{x} \Rightarrow \mathit{rec}(u), S' \qquad S'(u) \neq \bullet}{E; S \vdash \mathtt{x} \Rightarrow S'(u), S'}$$

Règles principales

$$\frac{E;S \vdash M \Rightarrow (\mathtt{x}, M', E'), S' \qquad E;S' \vdash N \Rightarrow V, S'' \qquad E', \mathtt{x} = V; S'' \vdash M' \Rightarrow V', S'''}{E;S \vdash MN \Rightarrow V', S'''}$$

$$\frac{E;S \vdash M \Rightarrow prim(f), S' \qquad E;S' \vdash N \Rightarrow V, S''}{E;S \vdash MN \Rightarrow f(V, S'')} \quad \text{lorsque } f(V, S'') = (V', S''')$$

$$\frac{E;S \vdash M \Rightarrow V, S' \qquad E, \mathtt{x} = V; S' \vdash N \Rightarrow V', S''}{E;S \vdash \mathtt{let\ x\ =}\ M\ \mathtt{in}\ N \Rightarrow V', S''}$$

$$\frac{u \text{ fraîche pour } S \qquad E, \mathtt{x} = rec(u); S\{^u/_\bullet\} \vdash M \Rightarrow V, S' \qquad E, \mathtt{x} = rec(u); S'\{^u/_V\} \vdash N \Rightarrow V', S''}{E;S \vdash \mathtt{let\ rec\ x\ =}\ M\ \mathtt{in}\ N \Rightarrow V', S''}$$

$$\frac{}{E;S \vdash \mathtt{fun\ x}\ \rightarrow M \Rightarrow (\mathtt{x}, M, E), S}$$

$$\frac{i \in Int}{E;S \vdash i \Rightarrow i, S}$$

$$\frac{s \in String}{E;S \vdash s \Rightarrow s, S}$$

$$\frac{E;S \vdash M \Rightarrow true, S' \qquad E;S' \vdash N \Rightarrow V, S''}{E;S \vdash \mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ N' \Rightarrow V, S''}$$

$$\frac{E;S \vdash M \Rightarrow false, S' \qquad E;S' \vdash N' \Rightarrow V, S''}{E;S \vdash \mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ N' \Rightarrow V, S''}$$

$$\frac{}{E;S \vdash \mathtt{\{\ \}} \Rightarrow \{\}, S}$$

$$\frac{E;S \vdash M \Rightarrow \{l_i = V_i\}, S' \qquad E;S' \vdash N \Rightarrow V, S'' \qquad l \neq l_i}{E;S \vdash \mathtt{\{}\ M\mathtt{,\ l\ =}\ N\mathtt{\}} \Rightarrow \{l_i = V_i, l = V\}, S''}$$

$$\frac{E;S \vdash M \Rightarrow \{l_i = V_i\}, S'}{E;S' \vdash M\mathtt{.l}_i \Rightarrow V_i, S'}$$

$$\frac{E;S \vdash M \Rightarrow \{l_i = V_i\}, S'}{E;S \vdash M\backslash\mathtt{l}_i \Rightarrow \{l_1 = V_1, \ldots, l_{i-1} = V_{i-1}, l_{i+1} = V_{i+1}, \ldots, l_n = V_n\}, S'}$$

A.7.4 Erreurs à l'exécution

Axiomes non capturés par le système de type

Identificateur non défini (pourrait être détecté par une analyse statique) :

$$\frac{}{\emptyset; S \vdash \mathtt{x} \Rightarrow \mathtt{err}, S}$$

Vraie erreur à l'exécution (par exemple 1/0) :

$$\frac{E;S \vdash M \Rightarrow prim(f), S' \qquad E;S' \vdash N \Rightarrow V, S''}{E;S \vdash MN \Rightarrow f(V, S'')} \quad \text{lorsque } f(V, S'') = (\mathtt{err}, S''')$$

Axiomes capturés par le système de type

$$\frac{E;S \vdash M \Rightarrow V,S' \qquad V \neq (\mathtt{x},M',E') \qquad V \neq prim(f)}{E;S \vdash MN \Rightarrow \mathtt{err},S'}$$

$$\frac{E;S \vdash M \Rightarrow V,S' \qquad V \notin Bool}{E;S \vdash \mathtt{if}\ \ M\ \ \mathtt{then}\ \ N\ \ \mathtt{else}\ \ N' \Rightarrow \mathtt{err},S'}$$

$$\frac{E;S \vdash M \Rightarrow V,S' \qquad V \neq \{l_i = V_i\}}{E;S \vdash \{M\mathtt{,\ l\ =\ }N\} \Rightarrow \mathtt{err},S'}$$

$$\frac{E;S \vdash M \Rightarrow \{l_i = v_i\},S' \qquad E;S' \vdash N \Rightarrow V,S'' \qquad l = l_i}{E;S \vdash \{M\mathtt{,\ l\ =\ }N\} \Rightarrow \mathtt{err},S''}$$

$$\frac{E;S \vdash M \Rightarrow V,S' \qquad V \neq \{l_i = V_i\}}{E;S \vdash M\,\mathtt{.l} \Rightarrow \mathtt{err},S'}$$

$$\frac{E;S \vdash M \Rightarrow \{l_i = V_i\},S' \qquad l \neq l_i}{E;S \vdash M\,\mathtt{.l} \Rightarrow \mathtt{err},S'}$$

$$\frac{E;S \vdash M \Rightarrow V,S' \qquad V \neq \{l_i = V_i\}}{E;S \vdash M\backslash\mathtt{l} \Rightarrow \mathtt{err},S'}$$

$$\frac{E;S \vdash M \Rightarrow \{l_i = V_i\},S' \qquad l \neq l_i}{E;S \vdash M\backslash\mathtt{l} \Rightarrow \mathtt{err},S'}$$

Propagation des erreurs

$$\frac{E;S \vdash \mathtt{x} \Rightarrow \mathtt{err},S' \qquad \mathtt{x} \neq \mathtt{y}}{E,\mathtt{y} = V';S \vdash \mathtt{x} \Rightarrow \mathtt{err},S'}$$

$$\frac{E;S \vdash M \Rightarrow \mathtt{err},S'}{E;S \vdash MN \Rightarrow \mathtt{err},S'}$$

$$\frac{E;S \vdash M \Rightarrow (\mathtt{x},M',E'),S' \qquad E;S' \vdash N \Rightarrow \mathtt{err},S''}{E;S \vdash MN \Rightarrow \mathtt{err},S''}$$

$$\frac{E;S \vdash M \Rightarrow prim(f),S' \qquad E;S' \vdash N \Rightarrow \mathtt{err},S''}{E;S \vdash MN \Rightarrow \mathtt{err},S''}$$

$$\frac{E;S \vdash M \Rightarrow (\mathtt{x},M',E'),S' \qquad E;S' \vdash N \Rightarrow V,S'' \qquad E',\mathtt{x} = V;S'' \vdash M' \Rightarrow \mathtt{err},S'''}{E;S \vdash MN \Rightarrow \mathtt{err},S'''}$$

$$\frac{E;S \vdash M \Rightarrow \mathtt{err},S'}{E;S \vdash \mathtt{let\ \ x\ \ =}\ \ M\ \ \mathtt{in}\ \ N \Rightarrow \mathtt{err},S'}$$

$$\frac{E;S \vdash M \Rightarrow V,S' \qquad E,\mathtt{x} = V;S' \vdash N \Rightarrow \mathtt{err},S''}{E;S \vdash \mathtt{let\ \ x\ \ =}\ \ M\ \ \mathtt{in}\ \ N \Rightarrow \mathtt{err},S''}$$

$$\frac{u \text{ fraîche pour } S \qquad E,\mathtt{x} = rec(u);S\{^u/_\bullet\} \vdash M \Rightarrow \mathtt{err},S'}{E;S \vdash \mathtt{let\ \ rec\ \ x\ \ =}\ \ M\ \ \mathtt{in}\ \ N \Rightarrow \mathtt{err},S'}$$

$$\frac{u \text{ fraîche pour } S \qquad E,\mathtt{x} = rec(u);S\{^u/_\bullet\} \vdash M \Rightarrow V,S' \qquad E,\mathtt{x} = rec(u);S'\{^u/_V\} \vdash N \Rightarrow \mathtt{err},S''}{E;S \vdash \mathtt{let\ \ rec\ \ x\ \ =}\ \ M\ \ \mathtt{in}\ \ N \Rightarrow \mathtt{err},S''}$$

$$\frac{E;S \vdash M \Rightarrow \mathtt{err},S'}{E;S \vdash \mathtt{if}\ \ M\ \ \mathtt{then}\ \ N\ \ \mathtt{else}\ \ N' \Rightarrow \mathtt{err},S'}$$

$$\frac{E; S \vdash M \Rightarrow true, S' \qquad E; S' \vdash N \Rightarrow \texttt{err}, S''}{E; S \vdash \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ N' \Rightarrow \texttt{err}, S''}$$

$$\frac{E; S \vdash M \Rightarrow false, S' \qquad E; S' \vdash N' \Rightarrow \texttt{err}, S''}{E; S \vdash \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ N' \Rightarrow \texttt{err}, S''}$$

$$\frac{E; S \vdash M \Rightarrow \texttt{err}, S'}{E; S \vdash \texttt{\{}\ M\texttt{,}\ \texttt{l = } N\texttt{\}} \Rightarrow \texttt{err}, S'}$$

$$\frac{E; S \vdash M \Rightarrow \{l_i = V_i\}, S' \qquad E; S' \vdash N \Rightarrow \texttt{err}, S''}{E; S \vdash \texttt{\{}\ M\texttt{,}\ \texttt{l = } N\texttt{\}} \Rightarrow \texttt{err}, S''}$$

$$\frac{E; S \vdash M \Rightarrow \texttt{err}, S'}{E; S \vdash M\texttt{.l} \Rightarrow \texttt{err}, S'}$$

$$\frac{E; S \vdash M \Rightarrow \texttt{err}, S'}{E; S \vdash M\backslash\texttt{l} \Rightarrow \texttt{err}, S'}$$

A.7.5 Quelques primitives prédéfinies dans l'environnement initial

Pour créer et manipuler les références :

$$ref(V, S) = (u, S\{^u/V\}) \quad \text{avec } u \text{ fraîche pour } S$$

$$!(V, S) = \begin{cases} (S(u), S) & \text{si } V = u \in dom(S) \\ (\texttt{err}, S) & \text{sinon} \end{cases}$$

$$:= (V, S) = \begin{cases} (prim(:=_u), S) & \text{si } V = u \in dom(S) \\ (\texttt{err}, S) & \text{sinon} \end{cases}$$
$$\text{où } :=_u (V, S) = (\texttt{()}, S\{^u/_V\})$$

Addition :

$$+(V, S) = \begin{cases} (prim(+_i), S) & \text{si } V = i \in Int \\ (\texttt{err}, S) & \text{sinon} \end{cases}$$
$$\text{où } +_i (V, S) = \begin{cases} (i + j, S) & \text{si } V = j \in Int \\ (\texttt{err}, S) & \text{sinon} \end{cases}$$

et de même pour les autres fonctions arithmétiques, booléennes et de comparaison...