Chapitre 6

D'autres extensions du langage ML et de son système de typage

Dans ce chapitre nous présentons d'autres extensions du language ML et de son système de typage. Dans une première partie, nous considérons des extensions du système de typage qui ne peuvent pas s'effectuer sans ajout de nouvelles primitives. L'extension du langage préserve le noyau ML, dans le sens où l'on retrouve simplement ML en retirant les nouvelles constructions du langage. On pourrait également parler d'extensions "horizontales". Nous présenterons également des extensions "verticales", où seul le système de typage est modifié, la syntaxe et la sémantique opérationelle du langage restant celles de ML, voir même du lamda-calcul.

Par exemple, l'ajout des références a nécessité des modifications importantes dans le formalisation de la sémantique, mais le système de typage est inchangé (dans la version basée sur la restriction de la liaison à des valeurs). Inversement, l'ajout d'objets enregistrements a nécessité une extension importante du système de typage, alors que leur sémantique s'écrivait très naturellement dans le formalisme existant. Nous présentons ci-dessous d'autres extensions du langage ML, nous ne les traitons pas avec le même soucis de détail que les précédentes.

6.1 Extensions horizontales

La plupart des extensions que nous avons vues jusqu'à présent rentrent dans cette catégorie. Les plus évidentes sont celles qui peuvent être traitées par simple ajout de constantes.

6.1.1 Types dynamiques

Il n'y a pas une seule bonne raison d'introduire des types dynamiques dans un langage, mais de nombreuses mauvaises raisons. Les types dynamiques sont en quelques sortes un paliatif à de nombreuses déficience du système de typage. Une valeur de type dynamique est la paire d'une valeur avec une représentation de son type ; son type statique est

une nouvelle constante `Dyn` ne portant aucune information. La véritable information de type et la valeur d'origine peuvent être retrouvées à l'éxécution en explorant la représentation du type portée par la valeur dynamique.

La construction de valeurs de type dynamique permet, par exemple, de fabriquer des listes non homogènes, de sauver des données dans un fichier et de les relire de façon sure (puisque les donneées sont enregistrées avec leur type), de passer des arguments polymorphes à des fonctions (au prix de vérifications de types à l'exécution). Bien souvent, ces cas particuliers peuvent trouver des solutions mieux adaptées, mais toutes différentes.

Nous décrivons la forme la plus simple d'ajout de types dynamiques. Le langage est étendu avec deux constructions :

$$a ::= \ldots \mid \texttt{dynamic}\ (a : \sigma) \mid \texttt{typecase}\ a\ \texttt{of}\ (x, \sigma) \Rightarrow a\ \texttt{else}\ a$$

La première permet de construire des valeurs de type dynamique. La seconde extrait la "vraie" valeur contenue dans une valeur de type dynamique pourvu que son type soit suffisamment général ; elle retourne une valeur par défaut dans le cas contraire.

La sémantique des opérations sur les dynamiques est très proche de celle des types concrets, à la différence qu'elle fait intervenir les types du langage. Les contextes d'évaluation sont étendus à :

$$E ::= \ldots \mid \texttt{dynamic}\ (E, \sigma) \mid \texttt{typecase}\ E\ \texttt{of}\ (x, \sigma) \Rightarrow a\ \texttt{else}\ a \qquad (FV(\sigma) = \emptyset)$$

Dans leur formulation la plus simple, le filtrage des valeurs de type dynamique est exact, i.e. les règles de réduction sont

$$\begin{array}{l}(\texttt{typecase dynamic}\ (v, \sigma)\ \texttt{of}\ (x, \sigma) \Rightarrow a \ldots\ \texttt{else}\ a_0) \xrightarrow{\epsilon} (\texttt{fun}\ (x)\ a)\ v \\ (\texttt{typecase dynamic}\ (v, \sigma)\ \texttt{of}\ (x, \sigma') \Rightarrow a \ldots\ \texttt{else}\ a_0) \xrightarrow{\epsilon} a_0\end{array}$$

Les opérations sur les dynamiques ne peuvent pas être typées comme des ajouts de primitives, car elles font intervenir des schémas de types. Les règles de typage sont :

$$\text{(Dynamic)}\quad \frac{A \vdash a : \tau \qquad \bar{\alpha} \notin FV(A)}{A \vdash \texttt{dynamic}\ (a : \forall\, \bar{\alpha}.\, \tau) : \texttt{Dyn}}$$

$$\text{(Typecase)}\quad \frac{A \vdash a : \texttt{dynamic} \qquad A \oplus (x : \sigma) \vdash a_1 : \tau \qquad A \vdash a_2 : \tau}{A \vdash \texttt{typecase}\ a\ \texttt{of}\ (x : \sigma) \Rightarrow a_1\ \texttt{else}\ a_2 : \tau}$$

Exercice 49 *Les propriétés élémentaires du système de type de ML sont-elles conservées ?* □

Exercice 50 *Montrer la correction du typage vis à vis de la sémantique.* □

En fait on peut permettre une sémantique plus souple, et

$$(\texttt{typecase dynamic}\ (v,\sigma)\ \texttt{of}\ (x,\sigma') \Rightarrow a\ \texttt{else}\ a_0) \xrightarrow{\epsilon} \begin{cases} (\texttt{fun}\ (x)\ a)\ v & \text{if } \sigma \leq \sigma' \\ a_0 & \text{otherwise} \end{cases}$$

où $\sigma \leq \sigma'$ signifie que σ est plus général que σ'.

Il est aussi possible d'écrire `dynamic` (a) au lieu de `dynamic` $(a : \sigma)$; cette liberté, utile en pratique est plus délicate à formaliser. D'une part on perd la propriété de type principal. Par exemple le programme `fun` (x) `dynamic` (x) a pour types `Int` $\rightarrow$ `Dyn` et `Bool` $\rightarrow$ `Dyn` mais pas $\alpha \rightarrow$ `Dyn` car la condition de la règle Dynamic ne serait pas respectée. En pratique, on pourra par exemple échouer lorsque le programme n'admet pas de type principal. D'autre part, le type avec lequel est construite une valeur dynamique doit être fixé statiquement. Il est donc plus commode de voir cette proposition comme une synthèse de programme plutôt que comme une synthèse de types, la sémantique du langage étant définie pour les valeurs dynamiques explicitement typées.

Pour en savoir plus

Il est également possible d'étendre les schémas de types utilisés comme filtres avec des variables de motifs. On pourra consulter [**?**, **?**] à ce sujet.

Enfin, une autre extension de ML intéressante et assez proche consiste à avoir des types dynamiques sans jamais construire de valeurs dynamiques, c'est-à-dire que les types étiquetant les valeurs dynamiques sont remplacés par des arguments supplémentaires transportant des informations de typage et ajoutés au besoin [**?**].

6.1.2 Types existentiels

Les types dynamiques cachent complètement le type des valeurs ; ils permettent, par exemple, de contruire des listes hétérogènes. Nous proposons ici une autre façon plus systématique qui permet aussi de caché, mais partiellement, le type d'une valeur. Il s'agit des types existentiels. A la différence des types dynamiques, les types existentiels ne permettent pas de retrouver la partie cachée par un test à l'exécution.

Un exemple fréquent consiste à "empaqueter" ensemble dans une même structure une donnée avec des fonctions permettant de manipuler celle-ci. Par exemple, une paire $p_1 \equiv$ `(1, print_int)` d'un entier et d'une fonction d'impression aura le type `int * (int` $\rightarrow$ `unit)`. Dans ce cas le type révèle entièrement la structure et interdit donc de mélanger p_1 avec une autre paire $p_2 \equiv$ `(true, print_bool)`. Dans l'exemple ci-dessus, on aimerait ne retenir que le fait que la première composante de la paire est une valeur qui peut être passée en argument à la deuxième composante. On exprime cela en disant que ces paires ont le type $\exists\alpha.(\alpha \rightarrow \texttt{Unit}) * \alpha$; elles peuvent maintenant être mémorisées dans une structure homogène, par exemple une liste.

Il est facile de représenter le type $\exists\alpha.((\alpha \rightarrow \texttt{Unit}) * \alpha)$ en ML. Il suffit de définir un nouveau symbol de type, par exemple `pair`, mais d'arité 0, c'est-à-dire que la variable

de type α est implicitement quantifiée et donc cachée. Il est aussi commode d'introduire un constructeur `Pair` de type $(\alpha * \texttt{Unit}) * \alpha \to \texttt{pair}$. En particulier, la création d'une valeur de type existentielle s'écrit `fun` (x) `Pair` (x).

```
type pair = Pair of ('a * ('a → unit));;
let p1 = Pair (print_int, 1) : pair
and p2 = Pair (print_bool, true) : pair;;
```

Mais que faire de ces objets une fois créés ? Il est possible de les déplacer, de les mémoriser bien sûr, mais comment en extraire les composantes et appliquer le second argument au premier ? Comment aussi empêcher d'appliquer le premier argument de p_1 au second argument de p_2 ? On écrira :

```
let (Pair (f,x)) = p1 in f x;;
```

Dans le corps $f\ x$ de l'expression, la paire `(f,x)` a le type $(\omega \to \texttt{Unit} * \omega)$ où ω est un nouveau constructeur de type ω, différent de tous les autres. Le résultat `f x` est de type `Unit` dans lequel u n'apparaît plus. Plus généralement, à l'ouverture d'une valeur de type dynamique $\exists\alpha.\tau$ une nouvelle constante de type ω est introduite pour représenter la valeur inconnue de α. La règle de typage sera

$$\frac{A \vdash a_1 : \texttt{pair} \qquad (\texttt{pair} = \texttt{Pair of } \exists\alpha.\tau) \in A \qquad A \oplus (x : \tau[\omega/\alpha]) \vdash a_2 : \tau_2 \qquad \omega \notin \tau_2, A}{\texttt{let } x = \texttt{Pair } a_1 \texttt{ in } a_2 : \tau_2}$$

Nous avons affirmé plus haut les types existentiels n'avaient pas de coût à l'exécution. En effet le constructeur `Pair` est le seul dans son type, et il est donc superflu, comme le constructeur $(\_, \_)$ des produits.

Il est aussi possible de se passer de la déclaration des ces constructeurs, en considérant que tous les constructeurs sont prédéclarés : par exemple `pair` est remplacé par $(\_ : \exists\alpha.\alpha * (\alpha \to \texttt{Unit}))$. En fait, il faut faire attention à ne considérer que des déclarations de la forme $(\_ : \exists\alpha.\tau)$ pour des α-squelettes τ. Un α-squelette est un squelette de type τ (i.e. un type à trou) tels que α apparaît dans tous les sous-termes (différents de $\_$). Par exemple, $\alpha * (\_ \to \_)$ car α n'apparait pas dans $(\_ * \_)$, mais $\alpha * (\alpha \to \_)$ en est un. Le symbole pré-défini $(\exists\alpha.\alpha * (\alpha \to \_))$ d'arité 1 permet d'écrire `Unit` $(\exists\alpha.\alpha * (\alpha \to \_))$ au lieu de `pair`, ce que l'on abbrège évidemment en $\exists\alpha.\alpha * (\alpha \to \texttt{Unit})$ avec la convention usuelle.

Les règles de typage s'écrivent :

$$\frac{A \vdash a_1 : \tau}{A \vdash (a_1 : \exists\alpha.\tau)}\ (\exists\text{-Intro}) \qquad \frac{A \vdash a_1 : \exists\alpha.\tau \qquad A \oplus (x : \tau[\omega/\alpha]) \vdash a_2 : \tau_2 \qquad \omega \notin A, \tau_2}{\texttt{let } x = (a_1 : \exists\alpha.\tau) \texttt{ in } a_2 : \tau_2}\ (\exists\text{-Elim})$$

Pour en savoir plus

Introduits dans [**?**], les types existentiels que nous avons décrits ici sont très simples, car ils sont entièrement déclarés. Nous avons utilisé le formalisme des types concrets pour indiquer quand et comment (avec quels types) emballer ou ouvrir une structure

de donnée. Ceci permet en particulier de modifier au minimum le système de type de ML (il a quand même fallu introduire la notion de constantes inconnues). Il est difficle, d'aller plus loin dans l'intégration des types existentiels et en faire des types de première classe tout en gardant la synthèse de types. On pourra se reporter à [**?**] pour suivre une telle tentative, et découvrir ses limites !

Pour une preuve de correction on pourra se reporter à [**?**]. Dans le chapitre **??** nous verrons une autre façon de réaliser des types abstraits.

6.1.3 Types universels

Nous avons mentionné l'usage des types dynamiques pour passer des valeurs de types polymorphes comme argument des fonctions, comme l'illustre l'exemple ci-dessous.

```
let self_apply dx =
  typecase dx of (dx : All 'a. 'a → 'a) → x x
  else fail
in self_apply (dynamic (fun y → y : All 'a. 'a → 'a));;
```

Cette utilisation des valeurs de type dynamique est très inesthétique. En particulier, nous sommes obligés de prévoir un comportement anormal au cas où nous appliquons `xxx` a une valeur du mauvais type.

De façon analogue au types existentiels, nous écrivons :

```
type id = Id of All 'a. 'a → 'a;;

let self_apply ax =  let (Id x) = ax in x x
in self_apply (Id (fun x → x));;
```

Les types universels déclarés sont encore plus simples que les types existentiels déclarés. On pourra se reporter à [**?**] pour une formalisation plus précise ou [**?**].

6.2 Extensions verticales

Nous décrivons ici des extensions essentielles du système de typage, en général au prix de perdre la synthèse des types. Nous nous limitons au noyau de ML décrit dans le premier chapitre.

6.2.1 Et si les types étaient récursifs ...

On peut étendre l'algèbre des types en remplaçant les termes de types par des expressions rationnelles. Les propriétés essentielles du langage sont conservées, en particulier, les programmes bien typés admettent des types principaux, et les programmes bien typés ne sont jamais bloqués.

Alors que dans le langage sans constante avec types récursifs, tous les termes seraient typables. C'est un peu pour cette raison que le système des types récursifs a été décrié.

Cependant, cela n'est plus vrai si le langage possède des constantes. Par exemple l'entier 1 appliqué à lui même ne peut pas être un programme bien typé car le type des entiers est incompatible avec celui des fonctions.

Le combinateur de point fixe introduit dans la section 2.3, page 27 est définissable dans cette extension de ML, car le programme

$$\texttt{let}\ Y\ F = \texttt{let}\ G\ f\ x = F\ (f\ f)\ x\ \texttt{in}\ G\ G$$

est bien typé. Donc un système de typage avec types récursifs n'a pas besoin d'une construction de récursion (ou d'un combinateur de point fixe primitif). Autoriser les types récursifs dans le langage invalide évidemment la propriété de terminaison pour les programmes bien typés. D'autres constructions comme les types récursifs déclarés, les exceptions ou bien sûr les structures mutables l'invalidaient déjà.

6.2.2 Types d'ordre supérieur

La règle Gen-Let de ML peut être remplacée par les deux règles suivantes :

$$\text{(Gen)}\ \frac{A \vdash a : \tau \qquad \alpha \notin dom\,(A)}{A \vdash a : \forall\,\alpha.\,\tau} \qquad \text{(Let)}\ \frac{A \vdash a_1 : \sigma \qquad A \oplus (x : \sigma) \vdash a_2 : \tau}{A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}$$

Il est maintenant possible de dériver des jugements $A \vdash a : \sigma$ mais l'ensemble des programmes typables avec chacun des deux systèmes est identique. La règle Gen signifie que si un programme possède le type τ dans un contexte A dans lequel la variable α n'apparaît pas, alors le programme possède le type polymorphe $\forall\,\alpha.\,\tau$, c'est-à-dire intuitivement qu'il possède tous les types $\tau[\tau_0/\alpha]$.

La fonction $\texttt{let}\ f = \texttt{fun}\ (x)\ x\ \texttt{in}\ f\ f$est typable en ML, en donnant à f le type polymorphe $\forall\,\alpha.\,\alpha \to \alpha$, mais on ne peut pas typer $(\texttt{fun}\ (f)\ f\ f)\ (\texttt{fun}\ (x)\ x)$. La raison est qu'il faudrait donner à l'argument f dans la première expression le type de $\forall\,\alpha.\,\alpha \to \alpha$ de f, et donc typer $\texttt{fun}\ (f)\ f\ f$ avec $(\forall\,\alpha.\,\alpha \to \alpha) \to \alpha' \to \alpha'$. Or en dans les schémas de type de ML les variables quantifiés ne peuvent à apparaître qu'en tête des types. On dit que les types de ML sont *quantifiés en tête* ou que la quantification est *prénexe*. Le système F étend le système de typage de ML en permettant des quantificateurs en possition arbitraire.

$$\tau ::= \alpha \mid \tau \to \tau \mid \forall\,\alpha.\,\tau$$

Les règles de typage sont

$$\text{(Fun)}\ \frac{A \oplus (x : \tau_2) \vdash a : \tau_1}{A \vdash \texttt{fun}\ (x)\ a : \tau_2 \to \tau_1} \qquad \text{(App)}\ \frac{A \vdash a_1 : \tau_2 \to \tau_1 \qquad A \vdash a_2 : \tau_2}{A \vdash a_1\ a_2 : \tau_1} \qquad \text{(Var)}\ \frac{z : \tau \in A}{A \vdash z : \tau}$$

$$\text{(Inst)}\ \frac{A \vdash a : \forall\,\alpha.\,\tau}{A \vdash a : \tau[t_0/\alpha]} \qquad \text{(Gen)}\ \frac{A \vdash a : \tau \qquad \alpha \notin A}{A \vdash z : \forall\,\alpha.\,\tau}$$

Exercice 51 (*) *Écrire une dérivation de typage de* `fun` $(x)\ x\ x$ *dans ce système.* □

Nous ne ferons pas une étude détaillée du système de typage ci-dessus. Dans le système F les typages sont préservés par réduction et les seuls programmes bien typés irréductibles sont les valeurs. Le typage garantit donc la sécurité de l'évaluation. Dans le système sans constantes les programmes bien typés ne peuvent pas être réduits indéfiniment. Par contre, l'inférence de type n'est pas décidable dans le système F.

Pour cette raison, nous pouvons nous intéresser à des sous-ensembles du système F. Nous définissons les types de rangs k de la façon suivante :

$$\begin{aligned}\tau_0 &::= \alpha \mid \tau \to \tau \\ \tau_{k+1} &::= \tau_k \mid \tau_k \to \tau_{k+1} \mid \forall\, \alpha.\, \tau_{k+1}\end{aligned}$$

Intuitivement, le rang mesure la profondeur des flèches traversées avant d'y trouver un quantificateur. Un type de τ_1 peut avoir des quantificateurs en tête seulement (ce sont les schémas de types de ML). Dans un type τ_2, un type τ_1 (c'est-à-dire un schéma de type de ML) peut aussi apparaître à gauche d'une flèche, etc.

La synthèse de type est possible dans le système où les contextes de typages sont de rang 1 et les types sont de rang 2, mais pas au delà.

On pourra également se reporter à [**?**] où est présenté plus en détail cette restriction décidable.

6.2.3 Types intersection

Dans le système F, une expression qui possède le type polymorphe $\forall\, \alpha.\, \tau$ possède aussi, en fait, tous les types $\tau[\tau_0/\alpha]$. Les types intersection permettent de donner un type, que nous notons $\tau_1 \cap \tau_2$, à une expression qui a à la fois le type τ_1 et le type τ_2. Les types intersection sont de la forme :

$$\tau ::= \alpha \mid \tau \to \tau \mid \tau \cap \tau$$

Les règles de typages sont :

$$\text{(Fun)}\ \frac{A \oplus (x : \tau_2) \vdash a : \tau_1}{A \vdash \texttt{fun}\ (x)\ a : \tau_2 \to \tau_1} \qquad \text{(App)}\ \frac{A \vdash a_1 : \tau_2 \to \tau_1 \qquad A \vdash a_2 : \tau_2}{A \vdash a_1\ a_2 : \tau_1}$$

$$\text{(Var)}\ \frac{z : \forall\, \bar{\alpha}.\, \tau_0 \in A}{A \vdash z : \tau_0[\bar{\tau}/\bar{\alpha}]} \qquad \text{(}\wedge\text{-Elim-L)}\ \frac{A \vdash a : \tau_1 \wedge \tau_2}{A \vdash a : \tau_1} \qquad \text{(}\wedge\text{-Elim-R)}\ \frac{A \vdash a : \tau_1 \wedge \tau_2}{A \vdash a : \tau_2} \qquad \text{(}\wedge\text{-Intro)}\ \frac{A \vdash a : \tau_1 \qquad A \vdash a : \tau_2}{A \vdash z : \tau_1 \wedge \tau_2}$$

L'ensemble des programmes typables dans le système des types intersection est exactement l'ensemble des termes fortement normalisables. La synthèse de types pour ce système est donc indécidable, puisque savoir si un terme arbitraire est fortement normalisable est indécidable.

Il existe également des restrictions décidables intéressantes des types intersection. On définit le rang d'un type intersection de façon similaire au rang d'un type quantifié :

$$\begin{aligned} \tau_0 &::= \alpha \mid \tau \to \tau \\ \tau_{k+1} &::= \tau_k \mid \tau_k \to \tau_{k+1} \mid \forall\, \alpha.\, \tau_{k+1} \end{aligned}$$

La synthèse de type est décidable dans le système de type de rang 2 où les jugements de typage sont de rang 1 et les types de rang 2. Cette restriction décidable est détaillée dans [**?**].

On pourra également se reporter à [**?**] pour un système de type qui comporte à la fois des types quanifiés et des types intersection d'ordre arbitraire.