Chapter 4

Extensions simples de mini-ML

Nous décrivons dans ce chapitre quelques traits du “vrai” langage ML qui s’ajoutent facilement à mini-ML. D’autres traits plus difficiles à ajouter sont décrits dans les chapitres suivants.

4.1 Les n-uplets

Pour passer des paires de mini-ML aux n-uplets de ML, il suffit de généraliser le constructeur de valeurs $(\_, \_)$ et le constructeur de types $\_ \times \_$ comme suit:

Expressions:	$a ::= \ldots \mid (a_1, \ldots, a_n)$
Valeurs:	$v ::= \ldots \mid (v_1, \ldots, v_n)$
Types:	$\tau ::= \ldots \mid \tau_1 \times \ldots \times \tau_n$
Contextes:	$\Gamma ::= \ldots \mid (\Gamma, a_2, \ldots, a_n) \mid (v_1, \Gamma, a_3, \ldots, a_n) \mid \ldots \mid (v_1, v_2, \ldots, v_{n-1}, \Gamma)$

On se donne alors une famille d’opérateurs de projection $\mathtt{proj}_{i,n}$ $(1 \leq i \leq n)$ pour extraire la $i^{\text{ième}}$ composante d’un n-uplet. Leurs règles de typage et de réduction sont:

$$\frac{E \vdash a_1 : \tau_1 \qquad \ldots \qquad E \vdash a_n : \tau_n}{E \vdash (a_1, \ldots, a_n) : \tau_1 \times \ldots \times \tau_n}$$

$$TC(\mathtt{proj}_{i,n}) = \forall \alpha_1 \ldots \alpha_n.\ (\alpha_1 \times \ldots \times \alpha_n) \to \alpha_i$$

$$\mathtt{proj}_{i,n}(v_1, \ldots, v_n) \xrightarrow{\varepsilon} v_i$$

Les résultats des chapitres 2 et 3 s’étendent sans problèmes aux n-uplets. En particulier, les hypothèses H0, H1, H2 du chapitre 2 sont vérifiées.

Exercice 4.1 *(*)/(**) Une autre manière de traiter les n-uplets est de les encoder par des paires imbriquées: $(a_1, \ldots, a_n)$ est vu comme une abréviation pour $(a_1, (a_2, (a_3, \ldots, (a_{n-1}, a_n))))$.*

(*) Définir la projection $\mathtt{proj}_{i,n}$ en termes de `fst` et `snd`.

(**) On note ML_t le langage mini-ML avec tuples “primitifs” (non encodés) et ML_p le langage mini-ML avec paires “primitives” et tuples encodés comme expliqué précédemment. Formaliser la traduction T des programmes de ML_t dans ML_p, et montrer qu’elle commute avec la réduction: si

$a \to a'$ dans ML_t, alors $T(a) \xrightarrow{*} T(a')$ dans ML_p. Réciproquement, est-il vrai que si $T(a) \to a''$, alors il existe a' tel que $a \to a'$ et $a'' = T(a')$?

4.2 Les types concrets

Les types concrets, aussi appelés types sommes ou types variants, jouent un rôle crucial pour définir de nouvelles structures de données, en particulier des structures récursives (listes, arbres, expressions, etc.). Un type concret se présente sous la forme d'un certain nombre de cas, appelés *constructeurs*, portant éventuellement un *argument*.

Exemples: un type `num` regroupant nombres entiers et nombres en virgule flottante s'écrit

```
type num = Entier of int | Flottant of float
```

Le type des expressions arithmétiques est:

```
type expr = Constante of int
          | Variable of string
          | Add of expr * expr
          | Diff of expr * expr
          | Prod of expr * expr
          | Quotient of expr * expr
```

Un autre exemple est la syntaxe abstraite des expressions mini-ML, comme dans les exercices de programmation.

Les types concrets peuvent être paramétrés par un ou plusieurs types, ainsi les types `option` et `list`:

```
type 'a option = None | Some of 'a
type 'a list = Nil | Cons of 'a * 'a list
```

La forme générale d'une déclaration de type concret est:

$$\texttt{type}\ (\alpha_1, \ldots, \alpha_n)\ t = C_1\ \texttt{of}\ \tau_1 \mid \ldots \mid C_p\ \texttt{of}\ \tau_p$$

$\alpha_1 \ldots \alpha_n$ sont les paramètres du type t. (Dans le cas fréquent $n = 0$, on écrit juste `type` $t = \ldots$.)
$C_1 \ldots C_m$ sont les constructeurs du type t.
$\tau_1 \ldots \tau_n$ sont les types des arguments des constructeurs.
(On convient de représenter les constructeurs constants comme des constructeurs `of unit`, où `unit` est un type muni d'une seule valeur notée `()`.)
On impose que $\mathcal{L}(\tau_i) \subseteq \{\alpha_1, \ldots, \alpha_n\}$ pour tout i.

La déclaration ci-dessus étend le langage de la manière suivante:

Opérateurs:	$op ::= \ldots \mid C_1 \mid \ldots \mid C_p \mid F_t$
Expressions de types:	$\tau ::= \ldots \mid (\tau_1, \ldots, \tau_n)\ t$
Valeurs:	$v ::= \ldots \mid C_1(v) \mid \ldots \mid C_p(v)$

Contextes: $\Gamma ::= \ldots \mid C_1(\Gamma) \mid \ldots \mid C_p(\Gamma)$

L'opérateur F_t est l'opérateur de filtrage associé au type t. Il permet de discriminer sur une valeur de type t suivant son constructeur de tête. Le filtrage qui s'écrit en ML

$$\texttt{match}\ a\ \texttt{with}\ C_1(x_1) \to a_1 \mid \ldots \mid C_p(x_p) \to a_p$$

est vu comme l'application suivante de l'opérateur F_t:

$$F_t(a, (\texttt{fun}\ x_1 \to a_1), \ldots, (\texttt{fun}\ x_p \to a_p))$$

La règle de réduction de F_t est:

$$F_t(C_k(v), v_1, \ldots, v_n) \quad \stackrel{\varepsilon}{\to} \quad v_k\ v \quad \text{si } C_k \text{ est le } k^{\text{ième}} \text{ constructeur du type } t$$

Les types des opérateurs C_k et F_t sont:

$$\begin{array}{rcl} C_k & : & \forall \alpha_1 \ldots \alpha_n.\ \tau_i \to (\alpha_1, \ldots, \alpha_n)\ t \\ F_t & : & \forall \alpha_1, \ldots, \alpha_n, \beta.\ ((\alpha_1, \ldots, \alpha_n)\ t \times (\tau_1 \to \beta) \times \ldots \times (\tau_n \to \beta)) \to \beta \end{array}$$

Exemples: pour le type `num` défini précédemment, on a:

$$\begin{array}{rcl} \texttt{Entier} & : & \texttt{int} \to \texttt{num} \\ \texttt{Flottant} & : & \texttt{float} \to \texttt{num} \\ F_{\texttt{num}} & : & \forall \beta.\ (\texttt{num} \times (\texttt{int} \to \beta) \times (\texttt{float} \to \beta)) \to \beta \\ F_{\texttt{num}}(\texttt{Entier}(v), v_1, v_2) & \stackrel{\varepsilon}{\to} & v_1\ v \\ F_{\texttt{num}}(\texttt{Flottant}(v), v_1, v_2) & \stackrel{\varepsilon}{\to} & v_2\ v \end{array}$$

Pour le type α `list`, on a de même:

$$\begin{array}{rcl} \texttt{Nil} & : & \forall \alpha.\ \texttt{unit} \to \alpha\ \texttt{list} \\ \texttt{Cons} & : & \forall \alpha.\ (\alpha \times \alpha\ \texttt{list}) \to \alpha\ \texttt{list} \\ F_{\texttt{list}} & : & \forall \alpha, \beta.\ (\alpha\ \texttt{list} \times (\texttt{unit} \to \beta) \times ((\alpha \times \alpha\ \texttt{list}) \to \beta)) \to \beta \\ F_{\texttt{list}}(\texttt{Nil}(v), v_1, v_2) & \stackrel{\varepsilon}{\to} & v_1\ v \\ F_{\texttt{list}}(\texttt{Cons}(v), v_1, v_2) & \stackrel{\varepsilon}{\to} & v_2\ v \end{array}$$

Les résultats des chapitres 2 et 3 s'appliquent presque immédiatement à cette extension de mini-ML. En particulier, l'hypothèse H1 de la page 19 est satisfaite, ce qui garantit la préservation des types pendant la réduction.

Le seul point délicat est que nous avons maintenant des applications d'opérateurs qui sont des valeurs et donc ne se réduisent pas: les applications de constructeurs $C_k(v)$. Il faut donc modifier l'hypothèse H2 de la page 19 de la manière suivante:

H2' Si $\emptyset \vdash op\ v : \tau$ et $op\ v$ n'est pas une valeur, alors il existe a' telle que $op\ v \stackrel{\varepsilon}{\to} a'$ par une δ-règle.

Il est facile de voir que cette modification de H2 n'invalide pas la preuve du lemme de progression (proposition 2.6).

Exercice de programmation 4.1 *Ajouter les n-uplets et les types concrets à l'un des évaluateurs mini-ML écrits précédemment (programmes 1.1 ou 2.1).*

Exercice 4.2 *(*) Montrer que les booléens et l'expression conditionnelle* `if...then...else` *sont des cas particuliers de types concrets.*

Exercice 4.3 *(*) Justifier la restriction $\mathcal{L}(\tau_i) \subseteq \{\alpha_1, \ldots, \alpha_n\}$ sur les types des arguments de constructeurs dans la déclaration* `type`. *(On montrera que le typage n'est pas sûr si cette restriction est levée.)*

Exercice 4.4 *(**) On considère le type concret suivant:*

```
type t = C of t → t
```

Quelles sont les valeurs de ce type? Montrer qu'il existe deux fonctions totales `enrouler` : (`t` → `t`) → `t` *et* `dérouler` : `t` → (`t` → `t`). *Utiliser ces fonctions pour donner un codage des termes du λ-calcul pur (non typé) sous forme d'expressions de type* `t`. *Est-ce que mini-ML sans opérateur de point fixe* `fix` *mais avec les types concrets est normalisant?*

Exercice 4.5 *(***) Le langage ML offre des mécanismes de filtrage plus puissants que l'opérateur de filtrage F_t utilisé ci-dessus. En particulier, on peut tester non seulement sur le constructeur de tête d'une valeur, mais aussi sur d'autres parties de la valeur. Exemple:*

```
match l with Nil -> 0 | Cons(x, Nil) -> 1 | Cons(x, y) -> 2
```

Cette expression renvoie 0 pour les listes vides, 1 pour les listes à un élément, et 2 pour les autres listes. Pour faire le même calcul en mini-ML, il faut emboîter deux filtrages $F_{\texttt{list}}$ comme suit:

F_{list}`(l, (fun v → 0), (fun l1 →` F_{list}`(l1, (fun v' → 0), (fun l2 → 2))))`

On considère mini-ML étendu comme suit:

Expressions: $a ::= \ldots \mid (\texttt{match } a_1 \texttt{ with } p \to a_2 \mid \_ \to a_3)$
Motifs: $p ::= \_ \mid x \mid c \mid C(p) \mid (p_1, \ldots, p_n)$

Le motif _ filtre toutes les valeurs. Le motif x (où x est un identificateur) filtre également toutes les valeurs, et lie x à la valeur filtrée. Un motif restreint à une constante c filtre les valeurs égales à c uniquement. Le motif $C(p)$ filtre les valeurs de la forme $C(v)$ où de plus l'argument v est filtré par p. Enfin, le motif $(p_1, \ldots, p_n)$ filtre les n-uplets $(v_1, \ldots, v_n)$ tels que p_i filtre v_i pour $i = 1, \ldots, n$. La construction `match` a_1 `with` $p \to a_2 \mid \_ \to a_3$ *teste si p filtre la valeur de a_1; si oui, a_2 est évaluée après remplacement des identificateurs liés dans p par leurs valeurs; si non, a_3 est évaluée.*

1) Donner la règle de réduction de la construction `match`.

2) Donner la règle de typage de la construction `match`.

3) Montrer que pour toute construction `match`, *il existe une expression mini-ML équivalente n'utilisant que les prédicats de filtrage F_t.*

4.3 Les enregistrements déclarés

Les enregistrements (*records*) sont des n-uplets dont les composantes sont nommées (par des étiquettes) au lieu d'être repérées par position. En Caml, ils sont traités de manière similaire aux types concrets.

Exemple: le type des points dans l'espace s'écrit:

```
type point = { x : float; y : float; z : float }
```

Le type des paires peut se définir comme suit:

```
type ('a, 'b) paire = { fst : 'a; snd : 'b }
```

La forme générale d'une déclaration d'enregistrement est:

$$\texttt{type}\ (\alpha_1, \ldots, \alpha_n)\ t = \{\textit{étiq}_1 : \tau_1; \ldots ; \textit{étiq}_p : \tau_p\}$$

La déclaration ci-dessus étend le langage de la manière suivante:

Opérateurs:	$op ::= \ldots \mid M_t \mid .\textit{étiq}_1 \mid \ldots \mid .\textit{étiq}_p$
Expressions de types:	$\tau ::= \ldots \mid (\tau_1, \ldots, \tau_n)\ t$
Valeurs:	$v ::= \ldots \mid M_t(v)$
Contextes:	$\Gamma ::= \ldots \mid M_t(\Gamma)$

Les opérateurs $.\textit{étiq}$ sont les fonctions d'accès aux champs: l'expression Caml $a.\textit{étiq}$ est vue comme l'application d'opérateur $.\textit{étiq}(a)$.

L'opérateur M_t est l'opérateur de construction de l'enregistrement: l'expression Caml $\{\textit{étiq}_1 = a_1; \ldots ; \textit{étiq}_p = a_p\}$ est vue comme l'application d'opérateur $M_t(a_1, \ldots, a_p)$ si t est le type enregistrement dont les étiquettes sont $\textit{étiq}_1, \ldots, \textit{étiq}_p$.

Les règles de typage et de réduction pour ces opérateurs sont:

$$\begin{array}{rcl} M_t & : & \forall \alpha_1 \ldots \alpha_n.\ \tau_1 \times \tau_2 \times \ldots \times \tau_p \to (\alpha_1, \ldots, \alpha_n)\ t \\ .\textit{étiq}_k & : & \forall \alpha_1 \ldots \alpha_n.\ (\alpha_1, \ldots, \alpha_n)\ t \to \tau_k \\ .\textit{étiq}_k(M_t(v_1, \ldots, v_p)) & \stackrel{\varepsilon}{\to} & v_k \end{array}$$

Remarque: en fait, l'ordre des étiquettes dans l'expression Caml $\{\textit{étiq}_1 = a_1; \ldots ; \textit{étiq}_p = a_p\}$ n'est pas forcément le même que dans la déclaration du type t. Pour refléter ce fait, il faut déterminer le type t et la permutation σ sur $\{1, \ldots, p\}$ tels que t est déclaré comme $\{\textit{étiq}_{\sigma(1)} : \tau_1; \ldots ; \textit{étiq}_{\sigma(p)} : \tau_p\}$ et traduire l'expression Caml ci-dessus en $M_t(a_{\sigma^{-1}(1)}, \ldots, a_{\sigma^{-1}(p)})$.

Remarque: ce traitement des enregistrements fait qu'une étiquette donnée ne peut appartenir à plusieurs types enregistrement simultanément. Nous verrons au chapitre 6 un traitement beaucoup plus souple des enregistrements.

4.4 Les contraintes de types

En ML, l'expression $(a : \tau)$ s'évalue comme a, mais force a à avoir le type τ. En mini-ML, on peut voir $(a : \tau)$ comme une application d'opérateur $\mathtt{contr}_\tau(a)$. Les opérateurs $\mathtt{contr}_\tau$ (un par type τ) se typent et s'évaluent comme suit:

$$\begin{array}{rcl} \mathtt{contr}_\tau & : & \forall \alpha_1 \ldots \alpha_n.\ \tau \rightarrow \tau \text{ si } \alpha_1 \ldots \alpha_n = \mathcal{L}(\tau) \\ \mathtt{contr}_\tau(v) & \overset{\varepsilon}{\rightarrow} & v \end{array}$$

Si τ contient des variables de types, cette présentation ne force pas a à avoir exactement le type τ, mais assure que le type de a est une instance $\varphi(\tau)$. Ainsi, $(1 : \alpha)$ est correct et a le type `int`. Ce comportement est celui adopté en Caml, mais Standard ML par exemple exige que a ait exactement le type τ. Ce dernier comportement ne peut s'exprimer en terme d'applications d'opérateurs; il faut une règle de typage spéciale.

4.5 Les références et autres structures de données mutables

Voir le chapitre 5.

4.6 Les exceptions

Voir le chapitre 5.