Résumé Caml

Résumé de la syntaxe et des types. Dans la suite, on notera v_1, v_2, $\ldots v_n$ les *identifiants* (noms de variables), e_1, e_2, $\ldots e_n$, f, r les expressions, m_1, $\ldots$, m_n les motifs.

Expressions

Expression	Type	Contrainte	Remarques
$()$	unit		type unit
true, false	bool		booléens
1, 27, -12	int		constantes entières
`1.`, `-0.12`, `27e-19`	float		constantes flottantes
`'a'`, `'c'`	char		caractères
`"foo"`, `""`, `"hello"`	string		chaînes de caractères
$[\|e_1; e_2; \ldots; e_n\|]$	α vect	$\forall i,\ e_i : \alpha$	tableau
(e_1)			parenthésage d'une expression
let $v_1 = e_1$ in e_2	α	$e_2 : \alpha$ avec $v_1 = e_1$	liaison locale : v_1 n'est définie que dans e_2.
let rec $v_1 = e_1$ and $\ldots$ and $v_n = e_n$ in e	α	$e : \alpha$	définitions mutuellement récursives
e_1 || e_2	bool	e_1, e_2 : bool	"ou" booléen
e_1 && e_2	bool	e_1, e_2 : bool	"et" booléen
not e_1	bool	e_1 : bool	négation booléenne
$e_1 + e_2$	int	e_1 : int, e_2 : int	addition entière
$e_1 \square e_2$	int	e_1 : int, e_2 : int	opération entière ($\square \in \{/, -, *, mod\}$)
$e_1 +. e_2$	float	e_1 : float, e_2 : float	addition flottante
$e_1 \square e_2$	float	e_1 : float, e_2 : float	opération flottante ($\square \in \{/., -., *.\}$)
$\square\ e_1$	float	e_1 : float	fonction flottante ($\square \in \{\exp, \cos, \sin, \ldots\}$)
e_1 ^ e_2	string	e_1 : string, e_2 : string	concaténation de chaînes
$e_1.[e_2]$	char	e_1 : string, e_2 : int	accès à un caractère dans une chaîne
$e_1.[e_2] \leftarrow e_3$	unit	e_1 : string, e_2 : int, e_3 : char	modification d'un caractère dans une chaîne
make_vect e_1 e_2	τ vect	e_1 : int, $e_2 : \tau$	création de tableau
$e_1.(e_2)$	τ	$e_1 : \tau$ vect, e_2 : int	accès à un élément de tableau
$e_1.(e_2) \leftarrow e_3$	unit	$e_1 : \tau$ vect, e_2 : int, $e_3 : \tau$	modification d'un élément de tableau
$e_1, e_2, \ldots, e_n$	$(\tau_1 * \tau_2 * \cdots * \tau_n)$	$e_1 : \tau_1, \ldots, e_n : \tau_n$	n-uplet (produit cartésien)
$f\ e_1$	β	$f : \alpha \to \beta$, $e_1 : \alpha$	application de fonction
$f\ e_1\ e_2\ \ldots\ e_n$			raccourci pour $((f\ e_1)\ e_2)\ \ldots e_n$
if e_1 then e_2 else e_3	α	e_1 : bool, $e_2 : \alpha$, $e_3 : \alpha$	test sur les booléens. Évaluation uniquement de la branche choisie.
if e_1 then e_2	unit	e_1 : bool, e_2 : unit	équivalent à if e_1 then e_2 else ()
$E(e_1, \ldots, e_n)$	τ	$\forall i,\ e_i : \tau_i$	constructeur algébrique avec type $\tau = \|\ldots\| E$ of $\tau_1 * \cdots * \tau_n \|\ldots$
match e_0 with $\|\ m_1 \to e_1$ $\|\ \ldots$ $\|\ m_n \to e_n$	α	$\forall i \geq 0,\ e_i : \alpha, m_i : \tau$ $e_0 : \tau$	filtrage (voir définition des motifs)
fun $v_1 \to e_1$	$\alpha \to \beta$	$e_1 : \beta$ avec $v_1 : \alpha$	fonction d'une variable

let $f\ v_1 = e_1$ in e_2			similaire à let $f =$ fun $v_1 \to e_1$ in e_2
function $\mid m_1 \to e_1$ $\mid \ldots$ $\mid m_n \to e_n$	$\tau \to \alpha$	$\forall i,\ e_i : \alpha, m_i : \tau$	fonction qui filtre son argument directement
$e_1;\ e_2$	α	e_1 : unit, $e_2 : \alpha$	évalue e_1, puis s'évalue comme e_2
for $v_1 = e_1$ to e_2 do e_3 done	unit	e_1, e_2 : int, e_3 : unit	boucle for (sur les entiers) [1].
while e_1 do e_2 done	unit	e_1 : bool, e_2 : unit	boucle while
ref e_1	α ref	$e_1 : \alpha$	création d'une nouvelle référence
$r := e_1$	unit	$r : \alpha$ ref, $e_1 : \alpha$	affectation de référence
! r	α	$r : \alpha$ ref	contenu d'une référence
$[e_1; \ldots; e_n]$	α list	$\forall i,\ e_i : \alpha$	liste d'éléments
[]	α list		liste vide
$e_1 :: e_2$	α list	$e_1 : \alpha$, $e_2 : \alpha$ list	constructeur de liste

On rappelle que l'application de fonction est *curryfiée*, c'est-à-dire que l'expression `f x y z` signifie en fait `((f x) y) z`. De même on écrira `fun x y z -> e` pour `fun x -> (fun y -> (fun z -> e))`.

Liaisons globales

Tout d'abord, rappelons qu'une définition globale s'écrit let $v_1 = e_1$;;. L'identifiant v_1 réfère à la valeur de e_1 dans la suite du fichier. On peut également définir plusieurs identifiants en même temps via let $v_1 = e_1$ and ... and $v_n = e_n$;;.

Déclarations de Types

On peut déclarer de nouveaux types de la manière suivante (notez les définitions de types algébriques) :

```
type my_int = int;;

type point = (float * float);;

type 'a option =
    | None
    | Some of 'a;;
```

Motifs

Les motifs utilisés dans les filtrages correspondent à un sous-ensemble des expressions valides, comprenant les n-uplets, types algébriques (y compris listes), nombres, chaînes et booléens. Les motifs peuvent *introduire* des variables, par exemple la variable `y` dans :

```
let x = Some 42;;
let succ_x = match x with
    | Some y -> Some (y+1)  (* introduit "y" *)
    | None -> None;;
```

On peut également utiliser la construction `when` pour restreindre un motif, comme dans la fonction suivante qui cherche un élément dans une liste :

```
let rec mem x = function
    | [] -> false
    | y :: _ when y = x -> true
    | _ :: l -> mem x l;;
```

1. la version avec "downto" existe aussi