Programmation Fonctionnelle Avancée

http://www-lipn.univ-paris13.fr/~saiu/teaching/PFA-2010/

Syntaxe simplifiée (et idéalisée) de ocaml avec sémantique informelle (written by Jean-Vincent Loddo)

Constantes et motifs correspondants (données prédéfinies)

	Expressions de type t ::=	*Expressions de valeur* e ::=	*Expressions de motif* p ::=	*Sémantique informelle de l'expression*	*Identificateurs prédéfinis associés au type*
int	`int`	k	k	l'entier k	`+ * - / mod`
float	`float`	k. | $k_1.k_2$	k. | $k_1.k_2$	le rationnel k ou $k_1.k_2$	`+. *. -. /. **`
char	`char`	`'c'`	`'c'`	le caractère c	`int_of_char`
string	`string`	“$c_1 \ldots c_n$“	“$c_1 \ldots c_n$“	la chaîne de caractères $c_1 \ldots c_n$	`^`
bool	`bool`	`true` | `false`	`true` | `false`	le booléen vrai ou faux correspondant	`not` `&&` (ou `&`) `||` (ou `or`) `=` `==` `<>` (ou `!=`) `<` `>` `<=` `>=`
unit	`unit`	`()`	`()`	l'unique élément de unit={•} (résultat des commandes provoquant des effets de bord)	`print_int` `print_float` `print_char` `print_string` ..
product	t_1 `*` $\cdots$ `*` t_n ($n \geq 2$)	e_1 , $\ldots$, e_n	p_1 , $\ldots$, p_n	l'n-plet $(v_1, .., v_n)$ des valeurs représentées par e_1.. e_n	`fst` `snd` (n=2)
disjonction	| X_1 [`of` t_1] ⋮ | X_n [`of` t_n]	X_i [e]	X_i [p]	le terme ayant X à la racine et possédant éventuellement le sous-terme représenté par e comme unique fils	
list	t `list`	`[]`	`[]`	la liste vide	`@` `hd` `tl` `nth` `rev`
		e_1 `::` e_2	p_1 `::` p_2	la liste composé de la valeur v (représentée par e_1) suivie des valeurs de la liste ℓ (repr. par e_2)	`map` `iter` `exists` `find` `mem` `assoc` `sort` ..
record	{x_1 `:` t_1 `;` … `;` x_n `:` t_n}	{x_1 `=` e_1 `;` … `;` x_n `=` e_n}	{x_{i_1} `=` p_{i_1} `;` … `;` x_{i_k} `=` p_{i_k}}	l'enregistrement composé des champs $x_1 \ldots x_n$ affectés respectivement aux valeurs $v_1 \ldots v_n$ représentés par les expressions $e_1 \ldots e_n$. Dans la construction du motif $\{i_1, .., i_k\} \subseteq \{1, .., n\}$.	
exception	`exn`	`raise` X [e]	X [p]	exception représentée par le constructeur X du type somme exn (extensible par le toplevel)	
(.)	(t)	(e)	(p)	la valeur représentée par e	

Expressions de calcul

	Expressions de type t ::=	*Expressions de valeur* e ::=	*Sémantique informelle de l'expression*
function	t_1 -> t_2	function | p_1 -> e_1 ⋮ | p_n -> e_n	la fonction qui à tout $v \in t_1$ associe la valeur de t_2 représentée le premier e_i dont le motif p_i unifie avec la valeur v qui sera donnée en argument. L'expression sélectionnée e_i est évaluée dans l'environnement d'exécution augmenté par tous les liens résultant de l'unification (matching) de p_i avec v
apply		e_1 e_2	le résultat de l'application de la valeur fonctionnelle représentée par e_1 à l'argument représenté par e_2
bloc		let d in e_1	la valeur représentée par e_1 dans l'environnement d'exécution ρ augmenté par tous les liens provoqués par les définitions dans d
name	'a	x	la valeur de l'identificateur x dans l'environnement d'exécution ρ
sel		x.e	la valeur du champ x de l'enregistrement représenté par e dans l'environnement d'exécution ρ
try		try e_0 with | p_1 -> e_1 ⋮ | p_n -> e_n	la valeur de e_0 si la calcul se déroule normalement; sinon, l'exception levée par le calcul de e_0 sera unifiée avec le premier motif compatible et le résultat sera la valeur calculée par l'expression associée à ce motif

Expressions impératives (commandes de type unit ou liées au type unit)

e : := while e_0 do e_1 done
| for x = e_0 (to|downto) e_1 do e_2 done
| if e_0 then e_1
| e_1 ; e_2
| ref e
| !e
| e_0 := e_1

Autres types

t : := t x (application de l'expression de type t au type paramétré nommé x)

Définitions de noms (de valeurs)

	Définition de valeurs **d** ::=	*Sémantique informelle de la définition*
simple	`x = e`	ajoute le lien (x,v) à l'environnement d'exécution ρ, où v est le résultat de l'évaluation de e dans ρ
pattern	`p = e`	ajoute à l'environnement d'exécution ρ tous les liens résultants du matching entre le motif p et le résultat v de l'évaluation de e dans ρ
parallèle	$\mathbf{d}_1$ `and` $\mathbf{d}_2$	ajoute l'union des liens provoqués par les définitions d_1 et d_2 évaluées sur le même environnement d'exécution
récursive	`rec d`	ajoute à l'environnement d'exécution les liens provoqués par d considérant que les expressions contenues dans d peuvent déjà utiliser tous les noms définis par d, malgré qu'ils soient encore en cours de définition. (**idéalisation** : le *rec* doit en réalité être écrit juste après le *let*, et ne peut donc se trouver dans une sous-définition)

Instructions Top Level

	Définition de valeur/type **l** ::=	*Sémantique informelle*
valdef	`let d ;;`	ajoute à l'environnement d'exécution ρ tous les liens de la forme (nom,valeur) provoqués par la définition d.
typedef	`type x = t ;;`	enregistre dans l'environnement de *typage* que le nom x est synonyme du type t. Cette instruction est nécessaire pour toute utilisation successive d'un type enregistrement ou somme. Autrement dit, les types enregistrement et somme doivent être nommés de façon préliminaire à l'utilisation de leurs expressions ou motifs.
parameter	`type 'a x = t ;;`	enregistre dans l'environnement de *typage* que le nom x est synonyme (de la famille) du type t paramétré par 'a.
parameters	`type ('a`$_1$`,...,'a`$_n$`) x = t ;;`	enregistre dans l'environnement de *typage* que le nom x est synonyme (de la famille) du type t paramétré par 'a_1,..'a_n.
general form	`type` [`'a`|`('a`$_1$`,...,'a`$_n$`)`] $\mathbf{x}_1$ `= t`$_1$ `and` ⋮ `and` [`'a`|`('a`$_1$`,...,'a`$_m$`)`] $\mathbf{x}_k$ `= t`$_k$ `;;`	forme générale de la définition de types : plusieurs types mutuellement récursifs sont définis, chacun éventuellement paramétré par un ou plusieurs paramètres. Cette définition enregistre dans l'environnement de *typage* tous les noms définis, leur type (avec paramètres éventuels) et leur relation mutuelle.
exn	`exception X [of t] ;;`	étend le type exn par le nouveau constructeur
eval	`e ;;`	évalue l'expression e dans l'environnement d'exécution courant.
séq	$\mathbf{l}_1$ $\mathbf{l}_2$	composition séquentielle des deux instructions de top level $\mathbf{l}_1$ et $\mathbf{l}_2$

Sucre syntaxique (macros ou abréviations)

Expressions

	Abréviation d'expression $\mathbf{e}$	*Développement*
listes	$[\mathbf{e}_1 ; \cdots ; \mathbf{e}_n]$	$\mathbf{e}_1 :: \cdots :: \mathbf{e}_n :: []$
match	`match` $\mathbf{e}_0$ `with` `|` $\mathbf{p}_1$ `->` $\mathbf{e}_1$ $\vdots$ `|` $\mathbf{p}_n$ `->` $\mathbf{e}_n$	`(function` `|` $\mathbf{p}_1$ `->` $\mathbf{e}_1$ $\vdots$ `|` $\mathbf{p}_n$ `->` $\mathbf{e}_n$`)` $\mathbf{e}_0$
cond	`if` $\mathbf{e}_0$ `then` $\mathbf{e}_1$ `else` $\mathbf{e}_2$	`match` $\mathbf{e}_0$ `with true ->` $\mathbf{e}_1$ `| false->` $\mathbf{e}_2$
fun	`fun` $\mathbf{p}_1 \cdots \mathbf{p}_k$ `->` $\mathbf{e}$	`function` $\mathbf{p}_1$ `-> function` $\mathbf{p}_2$ `->` $\cdots$ `-> function` $\mathbf{p}_k$ `->` $\mathbf{e}$
let-fun	`let` $\mathbf{x}$ $\mathbf{p}_1 \cdots \mathbf{p}_k$ `=` $\mathbf{e}_0$ `in` $\mathbf{e}_1$	`let` $\mathbf{x}$ `= (fun` $\mathbf{p}_1 \cdots \mathbf{p}_k$ `->` $\mathbf{e}_0$`) in` $\mathbf{e}_1$

Motifs

	Abréviation de motif $\mathbf{p}$	*Développement*
listes	$[\mathbf{p}_1 ; \ldots ; \mathbf{p}_n]$	$\mathbf{p}_1 :: \ldots :: \mathbf{p}_n :: []$

Top Level

	Abréviation de toplevel $\mathbf{l}$	*Développement*
let-fun	`let` $\mathbf{x}$ $\mathbf{p}_1 \cdots \mathbf{p}_k$ `=` $\mathbf{e}$ `;;`	`let` $\mathbf{x}$ `= fun` $\mathbf{p}_1 \cdots \mathbf{p}_k$ `->` $\mathbf{e}$ `;;`

Récapitulatif des méta-identificateurs (catégories syntaxiques) utilisés

$\mathbf{l}$	instructions top level
$\mathbf{e}$	expressions
$\mathbf{p}$	motifs
$\mathbf{t}$	types
$\mathbf{d}$	définitions de noms (de valeurs)
$\mathbf{k}$	entiers
$\mathbf{c}$	caractères
$\mathbf{x}, \mathbf{a}$	identificateurs (noms) commençant par une lettre minuscule (ou bien _)
$\mathbf{X}$	identificateurs (noms) commençant par une lettre majuscule