Fondements de la programmation

TD5 : Pattern-matching

1 Fragment avec Algebraic Data Types (ADT) d'OCaml

Le mot "motif" se dit "pattern" en anglais, de même "filtrage par motif" se dit "pattern matching", j'utiliserais plus souvent les seconds à l'oral, car ce sont les termes techniques qui donneront des résultats sur internet.

Nous utiliserons le langage :

TYPE

$$\begin{array}{rlr} \tau ::= & \alpha & \text{(var. de type)} \\ & \rho & \text{(const. de type)} \\ & \texttt{Num} & \text{(tyNombres)} \end{array}$$

TYPEPRODUIT $\quad \vec{\tau} ::= \tau_1 * \cdots * \tau_k$

PROGRAMME

$$\begin{array}{rlr} r ::= & \epsilon & \text{(prog vide)} \\ & c;;\ r & \text{(commande)} \end{array}$$

COMMANDE

$$\begin{array}{rlr} c ::= & \texttt{let}\ p = e & \text{(let-motif)} \\ & \texttt{type}\ \rho = A_1\ \texttt{of}\ \vec{\tau_1} \mid \cdots \mid \texttt{A}_\texttt{n}\ \texttt{of}\ \vec{\tau_n} & \text{(type def)} \end{array}$$

EXPRESSION

$$\begin{array}{rlr} e ::= & x & \text{(variable)} \\ & n & \text{(nombre littéral)} \\ & A(e_1, \cdots, e_n) & \text{(structure)} \\ & \texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid \cdots \mid p_n \to e_n & \text{(filtrage)} \end{array}$$

MOTIF

$$\begin{array}{rlr} p ::= & n & \text{(motif entier)} \\ & _ & \text{(motif jocker)} \\ & x & \text{(motif variable)} \\ & A(p_1, \ldots, p_n) & \text{(structures)} \end{array}$$

VALEUR

$$\begin{array}{rlr} v ::= & n & \text{(valeur entière)} \\ & A(v_1, \ldots, v_n) & \text{(valeur struct)} \end{array}$$

CONTEXTVAR $\quad \Gamma ::= x_1{:}\tau_1\ ,\ \ldots\ ,\ x_n{:}\tau_n$

CONTEXTA $\quad \Delta ::= A_1{:}\vec{\tau_1}{\to}\rho_1\ ,\ \ldots\ ,\ A_n{:}\vec{\tau_n}{\to}\rho_n$

Priorité : (définition) < (filtrage)

Associativité droite : filtrage

Lieurs :

type-def : ρ dans τ_1,...,τ_n
ρ dans les commandes suiv.
$A_1 ... A_n$ dans les commandes suiv.
filtrage : $FV(p_i)$ dans e_i (pour chaque i)
let-motif : $FV(p)$ dans les commandes suiv.

où les tokens $\alpha \in [']{[a-z]}$, $x \in [a-z_][a-zA-Z0-9_]^*$, $\rho \in [a-z_][a-zA-Z0-9_]^*$, $A \in [A-Z][a-zA-Z0-9_]^*$, et $n \in [0-9]^+$
sont variables de types, variables de termes, constante de type, constructeurs, et entiers, de OCaml.

Attention : les variables de termes et constantes de types sont syntaxiquement identique (même token), mais s'utilisent à des endroits différents.

Attention : constructeurs commencent par des majuscules et les variables par des minuscules

1.1 Analyse de la grammaires

Cette grammaire contient beaucoup de non terminaux. On reconnaît `Type`, `Expression`, `Valeur` et `ContextVar` que l'on a déjà vu (le dernier s'appelait juste `Context`). Le type produit est juste là pour simplifier les écritures. Les autres sont importants :

— `Commande` : les commandes sont des instructions ne s'évaluant pas en une valeur. Essentiel en programmation impérative (en C, un if, une boucle, une affectation... sont autant de commandes), elle sont plus anecdotiques en OCaml, elles servent essentiellement à définir de nouveaux types et fonctions.

— `Programme` : Un programme, dans presque tous les langages, est une séquence de commandes, qui seront évaluées séquentiellement.
— `Motif` : un motif est "une valeur à trou" ou "une structure à trous" dont les trous sont des variables ou un "_", un motif va être filtré par le "*match*" qui permet de "*match*er" une valeur en récupérant, dans les variables, les éléments "*match*és".
— `ContextA` : on manipulera deux dictionnaires, les contextes habituels (`ContextVar`) et ces contextes qui associent à chaque constructeur $A_1 ... A_n$ leur type, lequel qui est toujours de la forme $\vec{\tau} \to \rho$ où $\vec{\tau} = \tau_1 * \cdots * \tau_k$ sont les types des arguments du constructeur et ρ est le type construit par celui-ci.

1.2 Unification (dirigée) de terme

Lorsque l'on veut pouvoir faire plusieurs substitutions simultanément, on travail avec des unificateurs de termes. Un unificateur de terme est un dictionnaire

$$\theta = [x_1 := v_1, ..., x_n := v_n]$$

associant des valeurs à des variables.

On peut appliquer un unificateur à un terme

$$e[x_1 := v_1, \ldots, x_n := v_n] = e[v_1/x_1, \ldots, v_n/x_n]$$

ce qui substitut v_1 à x_1, ... v_n à x_n dans e. [1]

On veut unifier un pattern p et une valeur v, cela consiste à chercher un unificateur θ tel que $p\theta = v$.

Cette unification peut échouer s'il n'existe pas d'unificateur, on écrit alors

$$p \lhd v \Uparrow \ .$$

Sinon, on peut donner un meilleur unificateur θ, noté

$$p \lhd v \Downarrow \theta \ .$$

Règles de l'unification réussie

$$\frac{}{n \lhd n \Downarrow \emptyset}\ (\lhd n) \qquad \frac{}{_ \lhd v \Downarrow \emptyset}\ (\lhd _) \qquad \frac{}{x \lhd v \Downarrow \{x := v\}}\ (\lhd x)$$

$$\frac{p_1 \lhd v_1 \Downarrow \theta_1 \quad \cdots \quad p_n \lhd v_n \Downarrow \theta_n \qquad \theta_1 ... \theta_n \text{ disjoints}}{A(p_1,...,p_n) \lhd A(v_1,...,v_n) \Downarrow \theta_1 \uplus \cdots \uplus \theta_n}\ (\lhd A)$$

Règles de l'unification échouée

$$\frac{n \neq n'}{n \lhd n' \Uparrow}\ (\lhd n2) \qquad \frac{p_i \lhd v_i \Uparrow}{A(p_1,...,p_n) \lhd A(v_1,...,v_n) \Uparrow}\ (\lhd A2) \qquad \frac{A \neq A'}{A(p_1,...,p_n) \lhd A'(v_1,...,v_k) \Uparrow}\ (\lhd A3)$$

1.3 Système de transition

Règles de transition des programmes/commandes

L'évaluation d'une commande doit se faire en prenant en compte la suite du programme car elle peut disparaître, comme (type) qui n'a aucun comportement calculatoire (il n'est là que pour typer) ; mais surtout l'expression peut agir sur le contexte en y substituant des variables comme dans (letR)

$$\frac{}{\texttt{type } \rho = A_1 \texttt{ of } \vec{\tau_1} \mid \cdots \mid A_n \texttt{ of } \vec{\tau_n} ;; \ r \mapsto r}\ (\text{tp-elim})$$

$$\frac{e \mapsto e'}{\texttt{let } p{=}e ;; \ r \mapsto \texttt{let } p{=}e' ;; \ r}\ (\text{letG}) \qquad \frac{p \lhd v \Downarrow \theta}{\texttt{let } p{=}v ;; \ r \mapsto r\theta}\ (\text{letD})$$

1. Normalement, il faudrait préciser que les substitutions se font simultanément, mais puisque nos valeur ne contiennent aucune variable, ce n'est pas important ici.

Règles de transition des expressions

Une seule règle car on a peu d'expressions aujourd'hui. Un match va chercher à unifier, dans l'ordre, [2] le terme avec chacun des motifs jusqu'à en trouver un qui match.

$$\frac{p_1 \triangleleft v \Uparrow \quad \cdots \quad p_{i-1} \triangleleft v \Uparrow \quad p_i \triangleleft v \Downarrow \theta}{\texttt{match}\ v\ \texttt{with}\ p_1 \to e_1 \mid \cdots \mid p_n \to e_n \mapsto e_i\theta}\ (\text{match-elim}) \qquad \frac{e_i \mapsto e_i'}{A((e_1), \ldots (e_i) \ldots, (e_n)) \mapsto A((e_1), \ldots (e_i') \ldots, (e_n))}\ (\text{Ai})$$

1.4 Types

Le système de type va paraître étrange à ceux qui ont déjà vu des systèmes de types (L2, L2 ici). C'est simplement que l'on va typer des objets qui ne sont pas des expressions.

Une **expression** `e` se typait jusque là dans un séquent $\Gamma \vdash e : \tau$ car il pouvait contenir des variables libres (d'où le Γ) et produisait une valeur d'un certain type (ici τ). Mais aujourd'hui on parle aussi de constructeurs A, qui auraient été introduit dans des déclarations de types, et qui peuvent aussi apparaître dans un terme. On rajoute donc, aux séquents typant les expressions, un contexte de constructeur Δ, une expression est donc typé dans un séquent

$$\Gamma\Delta \vdash e : \tau$$

Mais les autres sortes (petit nom des non-terminaux) de notre grammaires n'ont pas forcément de raison d'être typé ainsi. Un **programme**, par exemple, ne renvoi pas de valeur et donc n'a pas de type, mais prend toujours un contexte (les constructeurs et variables introduites précédemment). Un programme est donc typé dans un séquent

$$\Gamma\Delta \vdash r$$

Pour les **commandes**, on peut faire la même remarque, qu'elles ne produisent aucune valeur mais qu'elles ont besoin de contextes de valeurs Γ et de constructeurs Δ. Par contre, ce n'est pas suffisant pour les caractériser, en effet une commande va lier des variables et constructeurs à l'extérieur : lorsque l'on définit un type ou une fonction, ceux-ci sont alors disponibles pour les commandes suivantes. Il faut donc indiquer les variables et constructeurs qui seront liés en sortie de la commande dans de nouveaux contextes où Γ' et Δ'. Ainsi une commande est typé dans un séquent

$$\Gamma\Delta \vdash c \dashv \Gamma'\Delta'$$

Enfin, nous souhaitons typer les **patterns/motifs**. Un motif contient des constructeurs libres, il a donc besoin d'un contexte de constructeurs Δ. Un motif contient aussi des variables, mais celles-ci ne sont pas libres, elles servent de lieur à l'extérieur du motif. On pourrait traiter les motifs comme dans les expressions (c'est souvent ce qui est fait), mais pour bien indiquer que l'on indique ce qui sera lié **après**, on met le contexte de variables Γ après. Enfin, un motif représente un terme à trou, ce terme a un type, on associe donc un type au motif. D'où la forme du séquent permettant de typer un motif :

$$\Delta \vdash p : \tau \dashv \Gamma$$

Règles de typage des programmes

$$\frac{}{\Gamma\Delta \vdash}\ (\vdash \epsilon) \qquad \frac{\Gamma\Delta \vdash c \dashv \Gamma'\Delta' \qquad \Gamma'\Delta' \vdash r}{\Gamma\Delta \vdash c;; r}\ (\vdash ;;)$$

Règles de typage des commandes

$$\frac{\Delta \vdash p : \tau \dashv \Gamma' \qquad \Gamma\Delta \vdash e : \tau}{\Gamma\Delta \vdash \texttt{let}\ p = e \dashv (\Gamma, \Gamma')\Delta}\ (\vdash \text{let-motif})$$

$$\frac{}{\Gamma\Delta \vdash \texttt{type}\ \rho = A_1\ \texttt{of}\ \vec{\tau_1} \mid \cdots \mid A_n\ \texttt{of}\ \vec{\tau_n} \dashv \Gamma, (\Delta, A_1 : \vec{\tau_1} \to \rho, \ldots, A_n : \vec{\tau_n} \to \rho)}\ (\vdash \text{tp-decl})$$

2. En réalité, il est plus efficace que ça et est capable supprimer plein de patterns d'un coup en regardant les constructeurs.

Règles de typage des expressions

$$\frac{}{(\Gamma, x : \tau)\Delta \vdash x : \tau}\ (\vdash \text{id}) \qquad \frac{}{\Gamma\Delta \vdash n : \texttt{Num}}\ (\vdash \text{num})$$

$$\frac{\Delta = \Delta', A : \tau_1 * ... * \tau_n \to \rho \qquad \Gamma\Delta \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma\Delta \vdash e_n : \tau_n}{\Gamma\Delta \vdash A(e_1, ..., e_n) : \rho}\ (\vdash \text{A})$$

$$\frac{\Gamma \vdash e : \tau \qquad \Delta \vdash p_1 : \tau \dashv \Gamma_1 \qquad \Gamma, \Gamma_1\Delta \vdash e_1 : \tau' \qquad \cdots \qquad \Delta \vdash p_n : \tau \dashv \Gamma_n \qquad \Gamma, \Gamma_n\Delta \vdash e_n : \tau'}{\Gamma\Delta \vdash \texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid \cdots \mid p_n \to e_n : \tau'}\ (\vdash \text{match})$$

Règles de typage des motifs

$$\frac{}{\Delta \vdash n : \texttt{Num} \dashv}\ (\vdash n \text{ motif}) \qquad \frac{}{\Delta \vdash _ : \tau \dashv}\ (\vdash _) \qquad \frac{}{\Delta \vdash x : \tau \dashv x : \tau}\ (\vdash x \text{ motif})$$

$$\frac{\Delta = \Delta', A : \tau_1 * \cdots * \tau_n \to \rho \qquad \Delta \vdash \texttt{p}_1 : \tau_1 \dashv \Gamma_1 \qquad \cdots \qquad \Delta \vdash \texttt{p}_n : \tau_n \dashv \Gamma_n \qquad \Gamma_1 ... \Gamma_n \text{ disjoints}}{\Delta \vdash A(\texttt{p}_1, ..., \texttt{p}_n) : \rho \dashv \Gamma_1 \uplus ... \uplus \Gamma_n}\ (\vdash \text{A motif})$$

2 Questions

On donne, ici, énormément de questions. Il ne s'agit pas de les faire toutes en TD, mais de donner suffisamment d'exercices pour s'entraîner.

2.1 Analyse de la grammaires

Question A. Écrire une commande et un pattern correctes de plus de 10 caractères. .

Remarquez que les commandes ne lient pas seulement des variables dans ce qu'elle contiennent mais aussi dans toutes les commandes qui suivent. Les lieurs-commandes sont souvent définit comme ça, par exemple les déclarations de variables en C ou java sont ce genre de lieur-commandes.

Question B. Écrire quelques programmes alpha-équivalents à `let x = 2;; let y = x;;`. .

Plus inhabituel : le matching ou le let avec motif permettent de lier plein de variables d'un seul coup, et les variables liés sont celles que l'on trouve dans le motif.

Question C. Écrire quelques programmes alpha-équivalents à. .

```
type list = Cons of Num*list | Vide ;;
let Cons(x,Cons(y,_)) = toto ;;
let _ = Cons(x,Cons(y,Vide)) ;;
```

Question D. Écrire l'AST de. .

```
let _ = match x with
   | A(0,z) -> z
   | A(z,_) -> B(z,1)
   | B(m,n) -> match m with
     |0 -> 0
     |1 -> 12
   | _ -> 40
;;
```

2.2 Unification (dirigée) et transitions

Question E. Résolvez les unifications dirigées suivantes.

- `Toto(foo,A,B(20,_))` $\triangleleft$ `Toto(B(3,A),A,B(20,B(42,A)))`
- `Toto(foo,A,B(20,_))` $\triangleleft$ `Toto(B(3,A),A,B(42,B(20,A)))`
- `Toto(foo,A,B(bar,_))` $\triangleleft$ `Toto(B(3,A),A,B(42,B(20,A)))`
- `Toto(foo,A,B(foo,_))` $\triangleleft$ `Toto(B(3,A),A,B(42,B(20,A)))`

Question F. Correction de l'unification. Montrer que si p est aussi un terme (c'est à dire qu'il ne contient pas de _) et si $p \triangleleft v \Downarrow \theta$ alors $p\theta = v$:

$$\frac{p \triangleleft v \Downarrow \theta \qquad p \text{ est un terme}}{p\theta = v}$$

Question G. Réduire le programme suivant. .

```
type nlist = Cons of Num*nlist | Vide ;;
type llist = Consl of nlist*llist | Videl ;;
let toto = Cons(2,Cons(0,Vide));;
let Consl(x,Consl(y,_)) = Consl(toto,Consl(Cons(0,toto)),Vide) ;;
let _ = match x with
```

```
  | Cons(0,z) -> z
  | Cons(z,_) -> Cons(z,y)
  | Vide -> Vide
;;
```

2.3 Types

Question H. Écrire une dérivation de type pour le programme de la question précédente. Vous pouvez écrire des morceaux séparéments et faire des alias (sinon vous aurez des soucis de place).

Question I. Extensivité des contextes. .

Montrez que si $\Gamma\Delta \vdash c \dashv \Gamma'\Delta'$ alors $\texttt{dom}(\Gamma) \subseteq \texttt{dom}(\Gamma')$ et $\texttt{dom}(\Delta) \subseteq \texttt{dom}(\Delta')$.

$$\frac{\Gamma\Delta \vdash c \dashv \Gamma'\Delta'}{\texttt{dom}(\Gamma) \subseteq \texttt{dom}(\Gamma')} \qquad \frac{\Gamma\Delta \vdash c \dashv \Gamma'\Delta'}{\texttt{dom}(\Delta) \subseteq \texttt{dom}(\Delta')}$$

Question J. (long) Affaiblissement à gauche des variables de terme. .

Montrez que pour tout programme p, commande c, et terme e :

$$\frac{\Gamma\Delta \vdash r \qquad x \notin \texttt{dom}(\Gamma)}{(\Gamma, x : \tau)\Delta \vdash r}\ (\text{weak_prg}) \qquad \frac{\Gamma\Delta \vdash e : \tau' \qquad x \notin \texttt{dom}(\Gamma)}{(\Gamma, x : \tau)\Delta \vdash e : \tau'}\ (\text{weak_exp})$$

$$\frac{\Gamma\Delta \vdash c \dashv \Gamma'\Delta' \qquad x \notin \texttt{dom}(\Gamma')}{(\Gamma, x : \tau)\Delta \vdash c \dashv (\Gamma', x : \tau)\Delta'}\ (\text{weak_cmd}) \qquad \frac{\Gamma\Delta \vdash c \dashv \Gamma'\Delta' \qquad x \notin \texttt{dom}(\Gamma) \qquad x \in \texttt{dom}(\Gamma')}{(\Gamma, x : \tau)\Delta \vdash c \dashv \Gamma'\Delta'}\ (\text{weak_cmd2})$$

Remarquez les deux versions d'affaiblissement des commandes. Il s'agit de traiter séparément les cas où le x de l'affaiblissement estlié par un `let`

Question K. Échec de l'affaiblissement à droite. Montrez qu'il existe $\Delta \vdash p : \tau \dashv \Gamma$ et $x \notin \texttt{dom}(\Gamma)$ mais pas $\Delta \vdash p : \tau \dashv \Gamma, x : \tau'$

Question L. Valeurs libres. .

Montrez que pour toute valeur, si $\Gamma\Delta \vdash v : \tau$, alors $\emptyset\Delta \vdash v : \tau$.

Dans les exercices suivant on utilisera des séquents $\Delta \vdash v : \tau$ pour les valeurs en n'indiquant pas le Γ qui ne sert à rien.

Question M. (pas facile) Non-habitation de l'ensemble des types. .

Montrez qu'il existe un type τ et un contexte de constructeurs Δ, tel que pour toute valeur v, $\Delta \nvdash v : \tau$

Question N. (dificile) Condition d'habitation des types. .

Trouver des conditions simples sur τ et sur Δ tel qu'il existe une valeur $\Delta \vdash v : \tau$

Question O. (long) Substitution. Étant donné un certain contexte, dans une expression bien typée, on peut toujours remplacer une variable libre par une expression bien typée de même type.

Montrer que

$$\frac{\Gamma\Delta, x : \tau \vdash r \qquad \Gamma\Delta \vdash e : \tau}{\Gamma\Delta \vdash r[e/x]}\ (\text{subst_prg}) \qquad \frac{\Gamma\Delta, x : \tau \vdash e' : \tau' \qquad \Gamma\Delta \vdash e : \tau}{\Gamma\Delta \vdash e'[e/x] : \tau'}\ (\text{subst_expr})$$

$$\frac{\Delta \vdash p' : \tau' \dashv \Gamma, x : \tau \qquad \Delta \vdash v : \tau}{\Delta \vdash p'[v/x] : \tau' \dashv \Gamma}\ (\text{subst_motif})$$

Question P. (long) Décomposition. Réciproquement, on peut toujours remplacer une sous-expression par une variable de même type dans une expression bien typée plus grande.

Montrer que

$$\frac{\Gamma\Delta \vdash r[e/x] \qquad \Gamma\Delta \vdash e : \tau}{(\Gamma, x : \tau)\Delta \vdash r}\ (\text{dec_prg}) \qquad \frac{\Gamma\Delta \vdash e'[e/x] : \tau' \qquad \Gamma\Delta \vdash e : \tau}{(\Gamma, x : \tau)\Delta \vdash e' : \tau'}\ (\text{dec_expr})$$

$$\frac{\Gamma\Delta \vdash c[e/x] \dashv \Gamma'\Delta' \qquad \Gamma\Delta \vdash e : \tau \qquad x \notin \Gamma \qquad x \in \Gamma'}{(\Gamma, x : \tau)\Delta \vdash c \dashv \Gamma'\Delta'}\ (\text{dec_cmd})$$

$$\frac{\Gamma\Delta \vdash c[e/x] \dashv \Gamma'\Delta' \qquad \Gamma\Delta \vdash e : \tau \qquad x \notin \Gamma \qquad x \notin \Gamma'}{(\Gamma, x : \tau)\Delta \vdash c \dashv (\Gamma', x : \tau)\Delta'}\ (\text{dec_cmd2})$$

2.4 Typage de l'unification et du système de transition

Question Q. Typage de l'unification. Si $\Delta \vdash p : \tau \dashv \Gamma$ et si $\Delta \vdash v : \tau$ et $p \triangleleft v \Downarrow \theta$, alors $\mathtt{dom}(\Gamma) = \mathtt{dom}(\theta)$ et pour tout $x \in \mathtt{dom}(\Gamma)$, $\Delta\Gamma' \vdash \tau(x) : \Gamma(x)$. Autrement dit :

$$\frac{\Delta \vdash p : \tau \dashv \Gamma \qquad p \triangleleft v \Downarrow \theta}{\mathtt{dom}(\theta) = \mathtt{dom}(\Gamma)} \qquad \frac{\Delta \vdash p : \tau \dashv (\Gamma, x : \tau') \qquad \Delta \vdash v : \tau \qquad p \triangleleft v \Downarrow (\theta, x := v')}{\Delta \vdash v' : \tau'}$$

Question R. Complétude de l'unification. Montrer que pour tout $\Delta \vdash p : \tau \dashv \Gamma$ et $\Delta \vdash v : \tau$, soit $p \triangleleft v \Uparrow$, soit il existe θ tel que $p \triangleleft v \Downarrow \theta$.

Question S. Préservation. Montrer que

$$\frac{\Gamma\Delta \vdash r \qquad r \mapsto r'}{\Gamma\Delta \vdash r'}\ (\text{preserv_cmd}) \qquad \frac{\Gamma\Delta \vdash e : \tau \qquad e \mapsto e'}{\Gamma\Delta \vdash e' : \tau}\ (\text{preserv_exp})$$

Question T. Non-progression. Montrer que le théorème de progression est faux.