Types existentiels

Types abstraits de données

- Manipulation de domaines complexes.
- La manipulation est indépendante de la représentation particulière du domaine.
- On se donne :
  - Un nom pour domaine abstrait
  - Des opérations de construction d'une représentation concrète vers le domaine abstrait
  - Des opérations de test pour les différents constructions.
  - Des opérations d'accès d'un objet abstrait vers les composantes d'une représentation concrète.

Exemple : jour

Domaine :

jour (jours vus comme des entiers de 1 à 31)

Opérations de construction :

cons_jour : $\{x : \texttt{entier} \mid 1 \leq \texttt{x} \leq 31\} \rightarrow \texttt{jour}$

Opérations de construction étendue :

entier_to_jour : entier $\rightarrow$ jour

Opérations d'accès :

numéro_jour : jour $\rightarrow \{\texttt{x} : \texttt{entier} \mid 1 \leq \texttt{x} \leq 31\}$

Exemple : exemplaire

Domaine :

exemplaire

Opérations de construction :

cons_livre : livre $\rightarrow$ exemplaire

cons_video : video $\rightarrow$ exemplaire

Opérations de test :

est_livre,est_video : exemplaire $\rightarrow$ bool

Opérations d'accès :

ex_livre : $\{x : \texttt{exemplaire} \mid \texttt{est\_livre(x)}\} \rightarrow$ livre

ex_video : $\{x : \texttt{exemplaire} \mid \texttt{est\_video(x)}\} \rightarrow$ video

Exemple : argent

Domaine :

argent

Opérations de construction :

francs : entier $\rightarrow$ argent

euros : entier $\rightarrow$ argent

Opérations de test :

est_francs, est_euros : argent $\rightarrow$ bool

Opérations d'accès :

val_argent : argent $\rightarrow$ entier

Exemple : Les cartes de couleur

Domaine :

couleur

Constantes :

pique, coeur, carreau, trèfle : couleur

Domaine :

figure

Constantes :

as, roi, dame, valet, dix, neuf, huit, sept : figure

Domaine :

carte

Opérations de construction :

cons_carte : couleur $\times$ figure $\rightarrow$ carte

Opérations d'accès :

figure_carte : carte $\rightarrow$ figure

couleur_carte : carte $\rightarrow$ couleur

Domaine :

main

Opérations de construction :

cons_main : $\{\vec{n} \in \text{carte}^5 \mid \text{Diff}(\vec{n})\} \rightarrow$ main

Opérations de construction étendue :

carte^5_to_main : $\text{carte}^5 \rightarrow$ main

Opérations d'accès :

carte^1 : main $\rightarrow$ carte

$\vdots$

carte^5 : main $\rightarrow$ carte

Domaine :

$\texttt{carte\_tarot}$

Constantes :

$\texttt{excuse} : \texttt{carte\_tarot}$

Opérations de construction :

$\texttt{cons\_tarot\_simple} : \texttt{carte} \rightarrow \texttt{carte\_tarot}$

$\texttt{cons\_atout} : \{n : \texttt{entier} \mid 1 \leq \texttt{n} \leq 21\} \rightarrow \texttt{carte\_tarot}$

Opérations de construction étendue :

$\texttt{entier\_to\_carte\_tarot} : \texttt{entier} \rightarrow \texttt{carte\_tarot}$

Opérations de test :

$\texttt{est\_carte\_simple,est\_atout} : \texttt{carte\_tarot} \rightarrow \texttt{bool}$

Opérations d'accès :

$\texttt{couleur\_de\_tarot} : \{x : \texttt{carte\_tarot} \mid \texttt{est\_carte\_simple(x)}\} \rightarrow \texttt{couleur}$

$\texttt{figure\_de\_tarot} : \{x : \texttt{carte\_tarot} \mid \texttt{est\_carte\_simple(x)}\} \rightarrow \texttt{figure}$

$\texttt{atout\_de\_tarot} : \{x : \texttt{carte\_tarot} \mid \texttt{est\_atout(x)}\} \rightarrow \{n : \texttt{entier} \mid 1 \leq \texttt{n} \leq 21\}$

Forme générale d'un TDA

Domaine :

dom

Constantes :

$c_1, \ldots, c_n : dom$

Opérations de construction :

$cons\_dom_1 : \{x : d_1 \mid P_1(x)\} \rightarrow dom$

$\vdots$

$cons\_dom_m : \{x : d_m \mid P_m(x)\} \rightarrow dom$

Opérations de construction étendue :

$dom_1\_to\_dom : d_1 \rightarrow dom$

$\vdots$

$dom_m\_to\_dom : d_m \rightarrow dom$

Opérations de test :

$est\_c_1, \ldots, est\_c_n : dom \rightarrow bool$

$\vdots$

$est\_op_1, \ldots, est\_op_m : dom \rightarrow bool$

Opérations d'accès :

$acces_1 \ : \{x : dom \mid est\_op_1(x)\} \rightarrow \{x : d_1 \mid P_1(x)\}$

$\vdots$

$acces_m \ : \{x : dom \mid est\_op_m(x)\} \rightarrow \{x : d_m \mid P_m(x)\}$

Autres opérations :

$\vdots$

Quelques variantes

On peut omettre ou rajouter quelques items selon les cas :

- Si la propriété $P_i(x)$ est toujours $true$, alors on l'écrit pas.
- Si le domaine est construit à partir d'une seule constante ou d'une seule opération de construction, alors inutile d'écrire l'opération de test car elle est équivalente à $est(x) = true$.
- Si le domaine est construit à partir de m opérations de construction, alors $m - 1$ opérations de test suffisent.
- Si un domaine d_i est un produit cartésien $a_1 \times \ldots \times a_k$, alors on peut remplacer l'opération d'accès

  $$acces_i \ : \{x : dom \mid est\_op_i(x)\} \rightarrow \{x : d_i \mid P_i(x)\}$$

  par k opérations de la forme

$acces_{i_1} \ : \{x : dom \mid \ est\_op_i(x)\} \rightarrow$
$\{y : a_1 \mid y = proj_1(x) \text{ et } x : d_i \text{ et } P_i(x)\}$

$\vdots$

$acces_{i_k} \ : \{x : dom \mid \ est\_op_i(x)\} \rightarrow$
$\{y : a_k \mid y = proj_k(x) \text{ et } x : d_i \text{ et } P_i(x)\}$

Le Tarot en OCAML

```
(* type couleur *)

#type couleur = Pique | Coeur | Carreau | Trefle;;


(* Constructeurs pour les constantes de couleur *)

#let pique   = Pique;;
#let coeur   = Coeur;;
#let carreau = Carreau;;
#let trefle  = Trefle;;
```

```
(* type figure *)

#type figure =
 As | Roi | Dame | Cavalier | Valet | Dix | Neuf | Huit | Sep

(* Constructeurs pour les constantes de figure *)

#let as      =As;;
#let roi     =Roi;;
#let dame    =Dame;;
#let cavalier=Cavalier;;
#let valet   =Valet;;
#let dix     =Dix;;
#let neuf    =Neuf;;
```



```
#let huit    =Huit;;
#let sept    =Sept;;


(* type carte *)

#type carte = Carte of figure * couleur;;



(* opération de construction du type carte *)

#let cons_carte(f,c) = Carte(f,c);;
```



```
(* opérations d'accès du type carte *)
#let figure_carte(e) = match e with Carte(f,c)->f;;
#let couleur_carte(e) = match e with Carte(f,c)->c;;

(* type carte_tarot *)

#type carte_tarot =
   Excuse | Simple of carte | Atout of int;;

(* opérations de construction étendue du type carte_tarot *)

#let excuse              = Excuse;;
#let cons_tarot_simple(c) = Simple(c);;
#let cons_atout (n)       = Atout(n);;
```



```
(* opérations de construction du type carte_tarot *)

#let int_to_carte_tarot (n) =
     if 1 <= n & n <= 21
        then cons_atout (n)
        else failwith"numéro d'atout invalide";;


(* opérations de test du type carte_tarot *)

#let est_carte_simple (c) =
     match c with Simple(_) -> true | _ -> false;;
```

```
#let est_atout (c)        =
     match c with Atout(_)  -> true | _ -> false;;



(* opérations d'accès du  type carte_tarot *)

#let couleur_de_tarot (e) =
     match e with
     Simple(Carte(_,c)) -> c
     | _                -> failwith "Pas une carte simple";;
```



```
#let figure_de_tarot (e) =
     match e with
     Simple(Carte(f,_)) -> f
     | _                -> failwith "Pas une carte simple";;


#let atout_de_tarot (e) =
     match e with
     Atout(x) -> x
     | _      -> failwith "Pas un atout";;
```


Les profils de symboles de la signature

```
pique:                          -> couleur
coeur:                          -> couleur
carreau:                        -> couleur
trefle:                         -> couleur
as:                             -> figure
roi:                            -> figure
dame:                           -> figure
cavalier:                       -> figure
valet:                          -> figure
dix:                            -> figure
neuf:                           -> figure
huit:                           -> figure
```



```
sept:                                -> figure
cons_carte:        figure * couleur  -> carte = <fun>
figure_carte:      carte             -> figure = <fun>
couleur_carte:     carte             -> couleur = <fun>
excuse:                              -> carte_tarot
cons_tarot_simple: carte             -> carte_tarot = <fun>
cons_atout:        int               -> carte_tarot = <fun>
int_to_carte_tarot:int               -> carte_tarot = <fun>
est_carte_simple:  carte_tarot       -> bool = <fun>
est_atout:         carte_tarot       -> bool = <fun>
figure_de_tarot:   carte_tarot       -> figure = <fun>
couleur_de_tarot:  carte_tarot       -> couleur = <fun>
atout_de_tarot:    carte_tarot       -> int = <fun>
```

Codage d'autres opérations d'un TDA

- Abstrait : on utilise uniquement les opérations abstraites définies dans le TDA.
- Concret : on utilise les primitives du langage de programmation associées au domaines concrets utilisés dans le TDA.

Codage abstrait

```
#let points (c) =
    if c = excuse
    then 4.5
    else if est_carte_simple (c)
         then let f = figure_de_tarot(c) in
              if f = roi
              then 4.5
              else if f = dame
                   then 3.5
                   else if f = cavalier
                        then 2.5
                        else if f = valet
```



```
                              then 1.5
                              else 0.5
         else let n = atout_de_tarot(c) in
              if n=1 or n=21
                 then 4.5
                 else 0.5;;

val points : carte_tarot -> float = <fun>
```


Codage concret

```
#let points(c) = match c with
    Excuse                -> 4.5
  | Simple(Carte(f,_))    -> (match f with
                                Roi      -> 4.5
                              | Dame     -> 3.5
                              | Cavalier -> 2.5
                              | Valet    -> 1.5
                              | _        -> 0.5)
  | Atout(1)              -> 4.5
  | Atout(21)             -> 4.5
  | _                     -> 0.5;;
val points : carte_tarot -> float = <fun>
```

Types existentiels

Types : $A ::= \ldots \mid \exists\alpha.A$

Expressions : $M ::= \ldots \mid$ `pack` M `with` A `as` $\exists\alpha.A$

Règle de typage pour l'introduction de $\exists\alpha.A$:

$$\frac{\Gamma \vdash M : A\{\alpha/B\}}{\Gamma \vdash \texttt{pack}\ M\ \texttt{with}\ B\ \texttt{as}\ \exists\alpha.A : \exists\alpha.A}$$

Le terme M donne une implémentation d'un TDA ayant signature A. Le type B utilisé dans l'implémentation du TDA est caché si α a des occurrences libres dans le type A.

Exemples simples

$p_1 =$ `pack` 0 `with` int `as` $\exists\alpha.\alpha$

$p_2 =$ `pack` $(0, \lambda x : int.x + 0)$ `with` int `as` $\exists\alpha.(\alpha \times (\alpha \to \alpha))$

$p_3 =$ `pack` $(0, \lambda x : int.x + 0)$ `with` int `as` $\exists\alpha.(\alpha \times (\alpha \to int))$

$p_4 =$ `pack` $(true, \lambda x : bool.1)$ `with` $bool$ `as` $\exists\alpha.(\alpha \times (\alpha \to int))$

$p_5 =$ `pack` $(0, \lambda x : int.x + 0)$ `with` int `as` $\exists\alpha.(int \times (int \to int))$

p_1 ne sert à rien, p_2 et p_3 ont la même implémentation mais différente signature, p_3 et p_4 ont la même signature mais différente implémentation, p_5 ne cache rien.

On peut aussi écrire :

`let` $a = 0;;$

`let` $f = \lambda x : int.x + 0;;$

`let` $p =$ `pack` (a, f) `with` int `as` $\exists\alpha.(a : \alpha \times (f : \alpha \to \alpha));;$

Ou bien

`let` $z = \{a = 0; f = \lambda x : int.x + 0\};;$

`let` $p =$ `pack` z `with` int `as` $\exists\alpha.(a : \alpha \times (f : \alpha \to \alpha));;$

$z.a$ et $z.f$ sont les composantes de z.

Exemple : un TDA

Domaine :

type_pile

Constantes :

pile_vide : type_pile

Opération de construction :

cons_pile : entier $\times$ type_pile $\rightarrow$ type_pile

Opération de test :

est_pile_nv : type_pile $\rightarrow$ bool

Opérations d'accès

premier_pile : $\{x : \text{type\_pile} \mid \text{est\_pile\_nv}(x)\} \rightarrow$ entier

reste_pile : $\{x : \text{type\_pile} \mid \text{est\_pile\_nv}(x)\} \rightarrow$ type_pile

Exemple : une première implémentation

```
#type pile = intlist;;

#let pile_vide = [];;
val pile_vide : intlist

#let cons_pile(a,p) = a::p;;
val cons_pile : int *  intlist -> intlist = <fun>

#let est_pile_nv(p) =
     match p with a::l -> true | _ -> false;;
val est_pile_nv : intlist -> bool = <fun>
```



```
#let premier_pile(p) = match p with a::l -> a;;
val premier_pile : intlist -> int = <fun>

#let reste_pile(p) = match p with a::l -> l;;
val reste_pile : intlist -> intlist = <fun>
```


type_pile $=$

$\alpha \times (\text{int} \times \alpha \rightarrow \alpha) \times (\alpha \rightarrow \text{bool}) \times (\alpha \rightarrow \text{int}) \times (\alpha \rightarrow \alpha)$

$M = (\text{pile\_vide}, \text{cons\_pile}, \text{est\_pile\_nv},$
$\text{premier\_pile}, \text{reste\_pile})$

$\text{type\_pile}\{\alpha/\text{intlist}\} = \text{intlist} \times (\text{int} \times \text{intlist} \rightarrow \text{intlist}) \times$
$(\text{intlist} \rightarrow \text{bool}) \times (\text{intlist} \rightarrow \text{int}) \times (\text{intlist} \rightarrow \text{intlist})$

$$\frac{\Gamma \vdash M : \text{type\_pile}\{\alpha/\text{intlist}\}}{\Gamma \vdash \texttt{pack}\ M\ \texttt{with}\ \text{intlist}\ \texttt{as}\ \exists\alpha.\text{type\_pile} : \exists\alpha.\text{type\_pile}}$$

Exemple : une autre implémentation

```
#type pile_nv = {tete:int; reste:ma_pile} and
      ma_pile = Pile_Vide | P of pile_nv;;

type pile_nv = { tete : int; reste : ma_pile; } and
     ma_pile = Pile_Vide | P of pile_nv


# let cons_pile_nv(t,r) = { tete=t ; reste = r};;
val cons_pile_nv : int * ma_pile -> pile_nv = <fun>

# let tete_pile_nv(p) = p.tete;;
val tete_pile_nv : pile_nv -> int = <fun>
```



```
# let reste_pile_nv(p) = p.reste;;
val reste_pile_nv : pile_nv -> ma_pile = <fun>

# let pile_vide =Pile_Vide;;
val pile_vide : ma_pile = Pile_Vide

# let cons_pile(a,p) = P(cons_pile_nv(a,p));;
val cons_pile : int * ma_pile -> ma_pile = <fun>

# let est_pile_nv(p) =
           match   p with Pile_Vide -> false | _ -> true ;;
val est_pile_nv : ma_pile -> bool = <fun>
```



```
# let premier_pile(p) = match p with
     Pile_Vide -> failwith"Pile Vide"
   | P(x)      -> x.tete;;
val premier_pile : ma_pile -> int = <fun>

# let reste_pile(p) = match p with
   Pile_Vide -> failwith"Pile Vide"
 | P(x) -> x.reste;;
val reste_pile : ma_pile -> ma_pile = <fun>
```


$\texttt{type\_pile} =$

$\alpha \times (\texttt{int} \times \alpha \to \alpha) \times (\alpha \to \texttt{bool}) \times (\alpha \to \texttt{int}) \times (\alpha \to \alpha)$

$M = (\texttt{pile\_vide}, \texttt{cons\_pile}, \texttt{est\_pile\_nv},$
$\texttt{premier\_pile}, \texttt{reste\_pile})$

$\texttt{type\_pile}\{\alpha/\texttt{ma\_pile}\} = \texttt{ma\_pile} \times (\texttt{int} \times \texttt{ma\_pile} \to \texttt{ma\_pile}) \times$
$(\texttt{ma\_pile} \to \texttt{bool}) \times (\texttt{ma\_pile} \to \texttt{int}) \times (\texttt{ma\_pile} \to \texttt{ma\_pile}$

$$\frac{\Gamma \vdash M : \texttt{type\_pile}\{\alpha/\texttt{ma\_pile}\}}{\Gamma \vdash \texttt{pack}\ M\ \texttt{with}\ \texttt{ma\_pile}\ \texttt{as}\ \exists\alpha.\texttt{type\_pile} : \exists\alpha.\texttt{type\_pile}}$$

Règle de typage pour l'élimination de $\exists\alpha.A$:

$$\frac{\Gamma \vdash M : \exists\alpha.A \quad \Gamma, z : A \vdash N : C}{\Gamma \vdash \texttt{unpack}\ M\ \texttt{as}\ \beta\ \texttt{with}\ z : A\ \texttt{in}\ N : C}$$

où la variable β n'est pas dans C

On ouvre l'implémentation M dans le programme N en lui donnant un nom z et en supposant que le type caché est β.

Exemples simples

Soit p une implémentation du type $\exists\alpha.(a : \alpha, f : \alpha \to int)$

$\texttt{unpack}\ p\ \texttt{as}\ \beta\ \texttt{with}\ z : A\ \texttt{in}\ z.f(z.a) + 2$

$\texttt{unpack}\ p\ \texttt{as}\ \beta\ \texttt{with}\ z : A\ \texttt{in}\ z.a + 2$ (mal typé)

$(\texttt{unpack}\ p\ \texttt{as}\ \beta\ \texttt{with}\ z : A\ \texttt{in}\ z.a) + 2$ (mal typé)

Sémantique

Si $p =$ $\texttt{pack}\ M\ \texttt{with}\ B\ \texttt{as}\ \exists\alpha.A$, alors

$\texttt{unpack}\ p\ \texttt{as}\ \beta\ \texttt{with}\ y : A\ \texttt{in}\ N$ se réduit dans

$N\{y/M\}\{\alpha/\beta\}$ qui se comporte comme

let $y : A = M$ **in** N