Cours 5 : Typage avancé

2024-2025 (450f02-dirty)

Thèmes

- types fantômes, types singletons, types uniques, *type-state*
- types (algébriques) non uniformes, généralisés
- types enregistrements, types objets (au sens P.O.O.)

Les types dans OCaml

- le langage est fortement typé
- le système de types est très riche et expressif
- les types peuvent servir de spécification très fine
- on se rapproche des pré/post-conditions
- ces spécifications sont vérifiées par le compilateur
- Conclusion: il faut exploiter les types

le reste du monde

La plupart des constructions présentées existent ou peuvent être reproduites, plus ou moins complètement, dans d'autres langages fortement typés avec polymorphisme paramétrique (par exemple: C++, Java, Rust, F#, Haskell, . . .)

Types abstraits

Types abstraits

"Type abstrait" : ensemble de valeurs abstraites caractérisé par des opérations et leurs spécifications [Liskov & Zilles, 1974]

- Séparation entre usage et implémentation
- Importance de la notion d'invariant

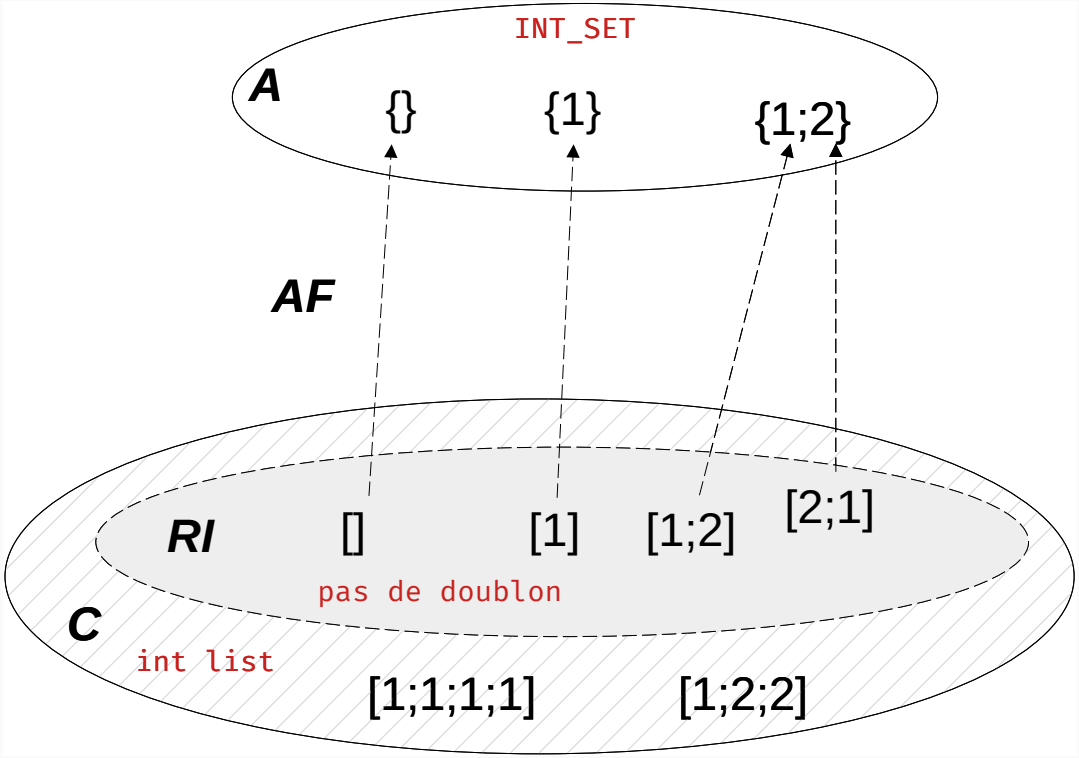
```
module type INT_SET = sig
  type t

  val empty : t
  val member : int -> t -> bool
  val add : int -> t -> t
end
```

Ex : représentation par des listes sans doublon.

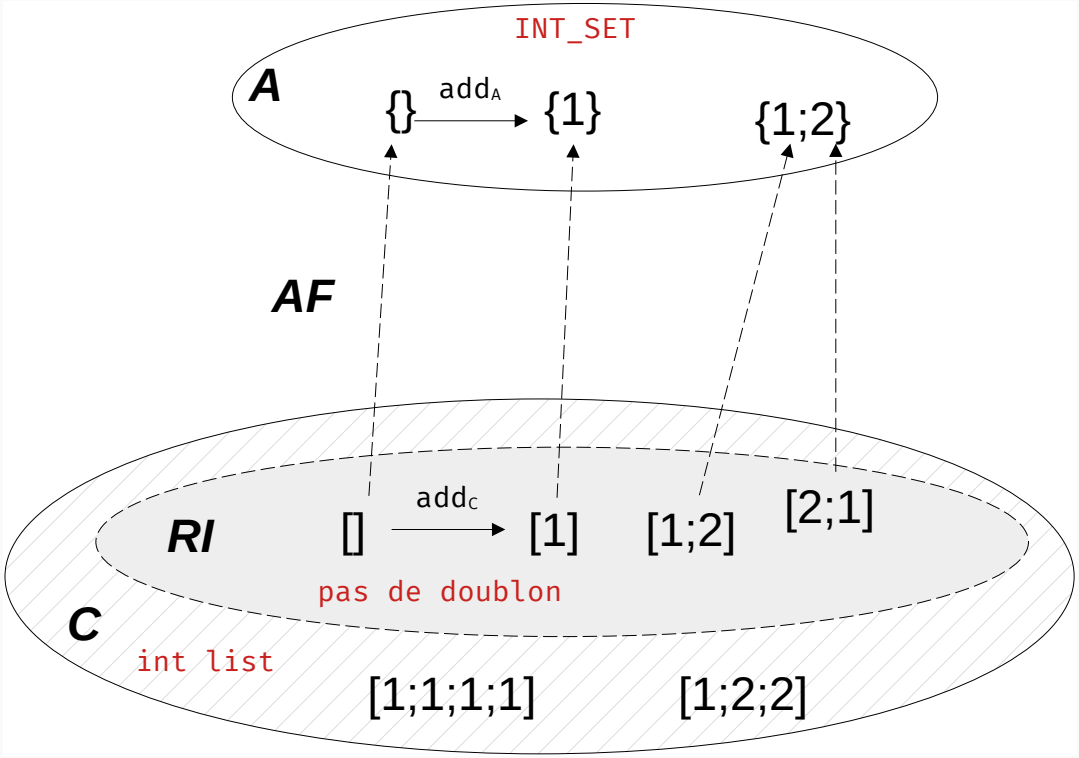
Représentation par un type concret

1. Le type abstrait (A) se situe dans le monde mathématique.
2. Choix d'un type concret C et d'une fonction d'abstraction $AF : C \rightarrow A$
   - AF pas implémentable mais à documenter
   - Surjective
   - Pas forcément injective (cf. $\{1; 2\}$)
   - Généralement partielle, de domaine de définition RI



Représentation par un type concret

1. Le type abstrait (A) se situe dans le monde mathématique.
2. Choix d'un type concret C et d'une fonction d'abstraction $AF : C \to A$
   - AF pas implémentable mais à documenter
   - Surjective
   - Pas forcément injective (cf. $\{1; 2\}$)
   - Généralement partielle, de domaine de définition RI



3. Une opération concrète est correcte si, quand elle respecte ses éventuelles préconditions, elle commute avec AF sur RI.

Invariant de représentation

RI = *invariant* de représentation $\rightsquigarrow$ souvent implémentable.

```
module NoDupList : INT_SET = struct
  type t = int list
  (* AF: la liste [a1; ...; an] représente l'ensemble {a1, ..., an}. *)

  (* RI: la liste ne contient pas de doublons. *)
  (* En phase de débogage : *)
  let check_rep l =
    if List.(length (sort_uniq Int.compare l) = length l) then l
    else failwith "RI"
  (* En phase d'exploitation : let check_rep l = l
       ou, mieux, emploi d'une version simplifiée et peu coûteuse *)

  let empty = check_rep []
  let member x s = List.mem x (check_rep s)
  let add x s = check_rep (if member x (check_rep s) then s else x :: s)
end
```

- Faut-il vérifier *RI* pour les arguments des fonctions *d'observation* (cf. deux `check_rep s` ci-dessus) ?
- Quid des fonctions privées ?

Types fantômes

Types fantômes

Définition

Un *type fantôme* est un type paramétré :

1. dont au moins un des paramètres *(variable fantôme)* n'apparaît pas dans la définition des *valeurs* de ce type
2. dont la définition est abstraite par une signature

Exemples

- ```
  module type S = sig type _ t ...
  module M : S = struct type _ t = int ...
  ```
- ```
  module type S = sig type (_,_) t ...
  module M : S = struct type ('a, 'b) t = Nil | Cons of 'b * ('a, 'b) t ...
  ```

Usage

- caractériser un état interne/caché (*type-state*)
  $\rightsquigarrow$ plutôt pour du code impératif (ou monade d'état, cf. cours 6)
- ne pénalise pas l'exécution (*zero cost abstraction*)

Exemple de type fantôme: spécification

- on veut imposer la lecture du premier caractère d'un fichier
- on définit l'interface `FichierLecture1Car`
- le paramètre du type `_ fichier`, prenant les valeurs `debut` et `fin`, définit l'état interne du fichier

```
module type FichierLecture1Car = sig
  type debut
  type fin
  type _ fichier
  val open : string -> debut fichier
  val read : debut fichier -> char * fin fichier
  val close : fin fichier -> unit
end
```

Exemple de type fantôme: réalisation

```
module Impl : FichierLecture1Car = struct
  type debut
  type fin
  type _ fichier = in_channel
  let open nom = open_in nom
  let read f = (input_char f, f)
  let close f = close_in f
end
```

Il est nécessaire d'imposer un usage purement séquentiel du fichier lu

- i.e. interdire:

```
let wrong = let f = Impl.open "toto" in (Impl.read f, Impl.read f, ...)
```

- mais autoriser:

```
let lire_char nom =
  let f = Impl.open nom in
  let (c, f) = Impl.read f in
  Impl.close f;
  c
```

Types non uniformes

Types non uniformes

Définition

Un type (récursif) non uniforme `'a t` fait apparaître des instances différentes du paramètre dans sa définition, **fonctions** de `'a`.

Exemples

- listes alternées:

```
type ('a, 'b) alt_list = | Nil | Cons of 'a * ('b, 'a) alt_list
```

- arbres binaires équilibrés:

```
type 'a perfect_tree = | Empty | Node of 'a * ('a * 'a) perfect_tree
```

Usage

- représenter des invariants de structure “descendants”
- meilleure spécification
- nécessite la “récursion polymorphe”

Un type non-uniforme: les arbres parfaits

Fonction qui calcule la profondeur d'un arbre parfait :

```
let rec depth = function
  | Empty -> 0
  | Node (_, sub) -> 1 + depth sub;;
```

Un type non-uniforme: les arbres parfaits

Fonction qui calcule la profondeur d'un arbre parfait :

```
let rec depth = function
  | Empty -> 0
  | Node (_, sub) -> 1 + depth sub;;
Error: This expression has type ('a * 'a) perfect_tree
but an expression was expected of type 'a perfect_tree
The type variable 'a occurs inside 'a * 'a
```

Un type non-uniforme: les arbres parfaits

Fonction qui calcule la profondeur d'un arbre parfait :

```
let rec depth : 'a perfect_tree -> int = function
  | Empty -> 0
  | Node (_, sub) -> 1 + depth sub;;
Error: This expression has type ('a * 'a) perfect_tree
but an expression was expected of type 'a perfect_tree
The type variable 'a occurs inside 'a * 'a
```

Un type non-uniforme: les arbres parfaits

Fonction qui calcule la profondeur d'un arbre parfait :

```
let rec depth : 'a . 'a perfect_tree -> int = function
  | Empty -> 0
  | Node (_, sub) -> 1 + depth sub;;
val depth : 'a perfect_tree -> int = <fun>
```

Récursion polymorphe : chaque application de `depth` doit avoir un type universellement quantifié.

Un type non-uniforme: les arbres parfaits

Fonction qui calcule la profondeur d'un arbre parfait :

```
let rec depth : 'a . 'a perfect_tree -> int = function
  | Empty -> 0
  | Node (_, sub) -> 1 + depth sub;;
val depth : 'a perfect_tree -> int = <fun>
```

Récursion polymorphe : chaque application de `depth` doit avoir un type universellement quantifié.

Variables de type en OCaml : attention aux confusions !

Sous `val` (ex : `val depth : 'a perfect_tree -> int`) une variable de type est forcément quantifiée universellement.

Sous `let` (ex : `let rec depth : 'a perfect_tree -> int`), une variable de type n'est *pas forcément* quantifiée universellement. Elle l'est si :

- Le typeur infère que c'est possible;
- Ou si on l'impose au typeur (ex : avec `'a .`) et que le typeur parvient à typer la fonction ainsi.

Types algébriques *généralisés*

Changement de notation !

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
```

Intuitivement les *constructeurs* `Empty` et `Node` sont des fonctions de profil :

```
Empty : 'a tree
Node : 'a tree * 'a * 'a tree -> 'a tree
```

OCaml offre une notation alternative pour les types algébriques qui reprend cette vision et explicite le type de retour de chaque constructeur :

```
type 'a tree =
  | Empty : 'a tree
  | Node : 'a tree * 'a * 'a tree -> 'a tree
```

Jusqu'ici, rien ne change p.r. à OCaml "de base"...

Changement de notation !

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
```

Intuitivement les *constructeurs* `Empty` et `Node` sont des fonctions de profil :

```
Empty : 'a tree
Node : 'a tree * 'a * 'a tree -> 'a tree
```

OCaml offre une notation alternative pour les types algébriques qui reprend cette vision et explicite le type de retour de chaque constructeur :

```
type _ tree =
  | Empty : _ tree
  | Node : 'a tree * 'a * 'a tree -> 'a tree
```

Jusqu'ici, rien ne change p.r. à OCaml "de base"...

Types algébriques généralisés (GADT)

Définition

Les types algébriques *généralisés* (GADT[1]) permettent de choisir les paramètres de type librement. Syntaxe et inférence sont distincts des autres types.

Deux nouvelles possibilités

- faire varier le type de retour des constructeurs:

```
type _ repr =
  | Int : int -> int repr
  | Add : (int -> int -> int) repr
```

- variables de type n'apparaissant pas dans le type de retour:

```
type showable = Showable : 'a * ('a -> string) -> showable
```

[1] *Generalized Algebraic Data Type*

Types algébriques généralisés

Usage

- généralisation des types non uniformes
- types singletons (une valeur, un type)
- types uniques
- permet d'exprimer le *Run Time Type Information*
- très expressif, associé au mécanisme d'**exhaustivité du filtrage**

Types singletons

Définition

Un type singleton est un type dont chaque valeur possible est associée à exactement une instance du GADT. Cela permet de « calculer dans les types » au moyen de ces valeurs.

```
type nat = Zero | Succ of nat
let deux = Succ (Succ Zero);;
val deux : nat = Succ (Succ Zero)
```

Types singletons

Définition

Un type singleton est un type dont chaque valeur possible est associée à exactement une instance du GADT. Cela permet de « calculer dans les types » au moyen de ces valeurs.

```
type nat = Zero | Succ of nat
let deux = Succ (Succ Zero);;
val deux : nat = Succ (Succ Zero)
```

```
type zero = Zero
type 'n succ = Succ of 'n
let deux = Succ (Succ Zero);;
val deux : zero succ succ = Succ (Succ Zero)
```

Types singletons

Définition

Un type singleton est un type dont chaque valeur possible est associée à exactement une instance du GADT. Cela permet de « calculer dans les types » au moyen de ces valeurs.

```
type nat = Zero | Succ of nat
let deux = Succ (Succ Zero);;
val deux : nat = Succ (Succ Zero)

type zero = Zero
type 'n succ = Succ of 'n
let deux = Succ (Succ Zero);;
val deux : zero succ succ = Succ (Succ Zero)

type zero = private Unused_zero
type 'n succ = private Unused2_succ
type _ nat = Zero : zero nat | Succ : 'a nat -> 'a succ nat
let deux = Succ (Succ Zero);;
val deux : zero succ succ nat = Succ (Succ Zero)
```

Variation du type de retour : application

```
(* listes de taille n *)
type (_, _) nlist =
  | Nil : ('b, zero) nlist
  | Cons : 'a * ('a, 'n) nlist -> ('a, 'n succ) nlist
```

Variation du type de retour : application

```
(* listes de taille n *)
type (_, _) nlist =
  | Nil : ('b, zero) nlist
  | Cons : 'a * ('a, 'n) nlist -> ('a, 'n succ) nlist

(* version totale de [hd] *)
let hd : ('a, _ succ) nlist -> 'a =
  function Cons (h, _) -> h;;
val hd : ('a, 'b succ) nlist -> 'a = <fun>
```

Variation du type de retour : application

```
(* ajouter un element en fin de liste *)
let rec snoc : 'a -> ('a,'m) nlist -> ('a,'m succ) nlist =
  fun a -> function
    | Nil -> Cons (a, Nil)
    | Cons (x,xs) -> Cons (x, snoc a xs);;
```

Variation du type de retour : application

```
(* ajouter un element en fin de liste *)
let rec snoc : 'a -> ('a,'m) nlist -> ('a,'m succ) nlist =
  fun a -> function
    | Nil -> Cons (a, Nil)
    | Cons (x,xs) -> Cons (x, snoc a xs);;

Error: This pattern matches values of type ('a, $0 succ) nlist
       but a pattern was expected which matches values of type ('a, zero) nlist
       The type constructor $0 would escape its scope
```

Pas un problème de récursion polymorphe, ici, mais de typage des motifs:

- Motif `Nil : ('a, zero) nlist`
- Motif `Cons (x, xs) : ('a, n succ) nlist` *pour un certain* `n` (= `$0`)

Variation du type de retour : application

```
let rec snoc : type m . 'a -> ('a, m) nlist -> ('a, m succ) nlist =
  fun a -> function
    | Nil -> Cons (a, Nil)
    | Cons (x,xs) -> Cons (x, snoc a xs);;
val snoc : 'a -> ('a, 'm) nlist -> ('a, 'm succ) nlist = <fun>
```

La notation `type m` (pas d'apostrophe !) stipule que :

- *(récursion polymorphe)* `m` est quantifié universellement ;
- *(type localement abstrait)* dans les motifs, `m` est égal aux types correspondants inférés dans les motifs ; ici :
  - motif `Nil : ('a, zero) nlist` : ajout de `zero = m`
  - motif `Cons (x, xs) : ('a, $0 succ) nlist` : ajout de `$0 succ = m` ;

  le typeur en déduit que les motifs sont bien typés (`zero = m = $0 succ`).

Notation quasi-systématique quand on fait de la récursion sur un GADT !

Arithmétique dans les types (Prolog !)

On a parfois besoin de raisonner sur plus que la simple application du successeur. Exemple : spécifier l'addition au niveau des types.

$$\frac{}{0 + n = n}\ \text{ADD\_ZERO} \qquad \frac{n + m = p}{(n + 1) + m = (p + 1)}\ \text{ADD\_SUCC}$$

```
type _ nat = Zero : zero nat | Succ : 'a nat -> 'a succ nat

type (_, _, _) add =
  | Add_zero : (zero, 'n, 'n) add
  | Add_succ : ('n, 'm, 'p) add -> ('n succ, 'm, 'p succ) add
```

Arithmétique dans les types : induction

Montrer que $\forall a \cdot a + 0 = a$:

$$\dfrac{\text{ADD\_ZERO}\ \dfrac{}{0 + 0 = 0} \qquad \dfrac{\dfrac{}{p + 0 = p \vdash p + 0 = p}\ \text{HYP}}{p + 0 = p \vdash (p + 1) + 0 = (p + 1)}\ \text{ADD\_SUCC}}{\forall a \cdot a + 0 = a}\ \text{IND.}$$

Arithmétique dans les types : induction

Montrer que $\forall a \cdot a + 0 = a$:

$$\dfrac{\text{ADD\_ZERO}\ \dfrac{}{0 + 0 = 0} \qquad \dfrac{\dfrac{}{p + 0 = p \vdash p + 0 = p}\ \text{HYP}}{p + 0 = p \vdash (p + 1) + 0 = (p + 1)}\ \text{ADD\_SUCC}}{\forall a \cdot a + 0 = a}\ \text{IND.}$$

```
let rec add_zero_right: type a. a nat -> (a, zero, a) add = fun a ->
  match a with
    | Zero -> Add_zero
    | Succ p -> Add_succ (add_zero_right p)
```

Arithmétique dans les types : induction

Montrer que $\forall a \cdot a + 0 = a$:

$$\dfrac{\text{ADD\_ZERO}\ \dfrac{}{0 + 0 = 0} \qquad \dfrac{\dfrac{}{p + 0 = p \vdash p + 0 = p}\ \text{HYP}}{p + 0 = p \vdash (p + 1) + 0 = (p + 1)}\ \text{ADD\_SUCC}}{\forall a \cdot a + 0 = a}\ \text{IND.}$$

```
let rec add_zero_right: type a. a nat -> (a, zero, a) add = fun a ->
  match a with
    | Zero -> Add_zero
    | Succ p -> Add_succ (add_zero_right p)
```

Rappel Coq :

```
Lemma add_zero_right: forall a, add a Zero a.
Proof.
    intro. induction a.
    - apply Add_zero.
    - apply Add_succ. apply IHm.
Qed.
```

Types uniques

Définition

Un type unique est un type associé à une seule donnée dans tout le programme. Un type unique est créé en même temps que la valeur correspondante.

Exemples

- directement avec des GADT :

```
type unique = Unique : 'a -> unique
let create_unique v = Unique v
```

Note : `∀ a . a -> unique ≅ (∃ a . a) -> unique` (`a ∉ unique`)

- avec un type abstrait et un module de première classe (cf. annexe)

```
module type TypeUnique = sig
  type unique
  val value : unique
end
(* create_unique : 'a -> (module TypeUnique) *)
let create_unique (type a) (v : a) =
  (module struct type unique = a let value = v end : TypeUnique)
```

Types uniques : exemple

- on considère un chiffrement asymétrique de type RSA (Rivest-Shamir-Adleman)
  - clé publique pour chiffrer, clé privée pour déchiffrer
- clés publiques/privées de type `key` de même nature (entiers), mais devraient être distinguées
- la donnée chiffrée de type `secret` devrait être associée à ses clés

Types uniques : déclarations

Interface “classique” :

```
module type RSA = sig


  (* clé *)
  type key
  (* secret produit *)
  type secret
  (* couple de clés publique/privée *)
  type key_pair = Kp of key * key
  (* creation d'une paire de cles publique/privee *)
  val create_key_pair : unit -> key_pair
  (* chiffrement a l'aide de la cle publique *)
  val encrypt : string -> key -> secret
  (* dechiffrement a l'aide de la cle privee *)
  val decrypt : secret -> key -> string
end
```

Types uniques : déclarations

Type fantôme (`key`) :

```
module type RSA = sig
  (* paramètres pour les clés *)
  type pub
  type priv
  (* clé publique ou privée (type fantôme) *)
  type 'pub_or_priv key
  (* secret produit *)
  type secret
  (* couple de clés publique/privée *)
  type key_pair = Kp of pub key * priv key
  (* creation d'une paire de cles publique/privee *)
  val create_key_pair : unit -> key_pair
  (* chiffrement a l'aide de la cle publique *)
  val encrypt : string -> pub key -> secret
  (* dechiffrement a l'aide de la cle privee *)
  val decrypt : secret -> priv key -> string
end
```

Types uniques : déclarations

Type fantômes (`key` et `secret`) + type existentiel :

```
module type RSA = sig
  (* paramètres pour les clés *)
  type pub
  type priv
  (* clé publique ou privée avec marqueur 'u pour l'unicité de cette clé *)
  type ('pub_or_priv, 'u) key
  (* secret produit avec la clé de marqueur 'u *)
  type 'u secret
  (* couple de clés publique/privée avec marqueur d'unicité 'u *)
  type key_pair = Kp : (pub, 'u) key * (priv, 'u) key -> key_pair
  (* creation d'une paire de cles publique/privee *)
  val create_key_pair : unit -> key_pair
  (* chiffrement a l'aide de la cle publique *)
  val encrypt : string -> (pub, 'u) key -> 'u secret
  (* dechiffrement a l'aide de la cle privee *)
  val decrypt : 'u secret -> (priv, 'u) key -> string
end
```

Type existentiel

Type de `Kp` : `∀ 'u . ((pub, 'u) key * (priv, 'u) key -> key_pair)`

`≅ (∃ 'u . (pub, 'u) key * (priv, 'u) key) -> key_pair`

Quand `Kp` est appliqué, `'u` est connu mais il est abstrait (oublié) dans `key_pair`.

On sait juste qu'il existe une instanciation pour `'u` mais on ne peut la laisser s'échapper sans briser la barrière d'abstraction :

```
module M(R :RSA) = struct
  open R
  let x = let Kp (a,b) = create_key_pair () in 1
end;;
module M : functor (R : RSA) -> sig val x : int end

module M(R :RSA) = struct
  open R
  let x = let Kp (a,b) = create_key_pair () in a
end;;
Error: This expression has type (pub, $Kp_'u) key
       but an expression was expected of type 'a
       The type constructor $Kp_'u would escape its scope
```

Types existentiels : implémentation “bidon”

```
module Bidon : RSA = struct
  type pub
  type priv
  type ('pub_or_priv, 'u) key = unit (* int dans une vraie implémentation *)
  type 'u secret = string
  type key_pair = Kp : (pub, 'u) key * (priv, 'u) key -> key_pair
  let create_key_pair () = Kp ((), ())
  let encrypt s pubk = s
  let decrypt secret privk = secret
end
```

Types existentiels : implémentation “bidon”

```
let () =
  let open Bidon in
  let msg = "message super important" in
  let Kp (pubk1, privk1) = create_key_pair () in
  let Kp (pubk2, privk2) = create_key_pair () in
  let secret = encrypt msg pubk1 in
  let clair = decrypt secret privk2 in (* erreur *)
  print_endline clair;;
Error: This expression has type (priv, $Kp_'u1) key
       but an expression was expected of type (priv, $Kp_'u) key
       Type $Kp_'u1 is not compatible with type $Kp_'u
```

Types existentiels : implémentation “bidon”

```
let () =
  let open Bidon in
  let msg = "message super important" in
  let Kp (pubk1, privk1) = create_key_pair () in
  let Kp (pubk2, privk2) = create_key_pair () in
  let secret = encrypt msg pubk1 in
  let clair = decrypt secret privk1 in
  print_endline clair;;
message super important
```

Enregistrements

Types enregistrements

- type enregistrement “classique”
- champs mutables ou non
- mise-à-jour impérative (`move_x : int -> t-> unit`)
- mise-à-jour fonctionnelle (`move_y : int -> t-> t`)

```
type t = { mutable x : int; y : int }

let create x y = { x = x; y = y }

let get_y p = p.y
(* filtrage : *)
let get_y { y; _ } = y

(* mise àjour impérative *)
let move_x d v = v.x <- v.x + d

(* mise-à-jour fonctionnelle *)
let move_y d v = { x = v.x; y = v.y + d }
(* ou mieux : *)
let move_y d v = { v with y = v.y + d }
let move_y d { x; y } = { x; y = y + d }
```

Types enregistrements *inlinés*

- type enregistrement “intégré” dans un type algébrique
- constructeurs avec arguments nommés plutôt que tuples anonymes

```
type t =
  | Point of {width: int; mutable x: float; mutable y: float}
  | ...

let v = Point {width = 10; x = 0.; y = 0.}

let scale l = function
  | Point p -> Point {p with x = l *. p.x; y = l *. p.y}
  | ....
let print = function
  | Point {x; y; _} -> Format.printf "%f/%f" x y
  | ....
let reset = function
  | Point p -> p.x <- 0.; p.y <- 0.
  | ...
let invalid = function
  | Point p -> p (* incorrect *)
  | ...
```

Types enregistrements polymorphes

- un type enregistrement peut être paramétrique, par exemple:

```
> type ('a, 'b) t = {x: 'a; f: 'b list -> 'b};;
type ('a, 'b) t = { x : 'a; f : 'b list -> 'b; }
> let test1 r e = r.f [e] = e;;
val test1 : ('a, 'b) t -> 'b -> bool = <fun>
> let test2 r = (r.f [0] = 0) && (r.f [true] = true);;
Error: This expression has type bool but an expression was expected
  of type int
```

- un type enregistrement peut contenir des champs *polymorphes*
- le champ `y` est maintenant polymorphe en `'b`:

```
> type 'a u = {x: 'a; f: 'b. 'b list -> 'b};;
type 'a u = { x : 'a; f : 'b. 'b list -> 'b; }
> let test1 r e = r.f [e] = e;;
val test1 : 'a u -> 'b -> bool = <fun>
> let test2 r = (r.f [0] = 0) && (r.f [true] = true);;
val test2 : 'a u -> bool = <fun>
```

Conclusion

- OCaml possède un langage de types très puissant
- nombreuses modélisations possibles, purement avec des types
- abstractions à faible coût
- beaucoup d'autres aspects non évoqués
  - Objets et classes (typage structurel)
  - Variants extensibles
  - Variants polymorphes
  - Types étiquettes et options

Annexe

Exemple de type unique avec un module de première classe

Une interface `KeyPair` spécifie les modules avec un type `unique` + 2 clés.

```
module type RSA = sig
 type pub
 type priv
  type ('typ, 'uniq) key
  type 'uniq secret
  (* Tout module qui implémente cette signature a un type existentiel car,
     étant abstrait, on sait juste qu'un type concret existe qui réalise
     "unique", mais on ne peut savoir lequel. *)
  module type KEY_PAIR = sig
      type unique
      val pubk : (pub, unique) key
      val privk : (priv, unique) key
    end
  (* creation d'une paire de cles publique/privee *)
  val create_key_pair : unit -> (module KEY_PAIR)
  (* chiffrement a l'aide de la cle publique *)
  val encrypt : string -> (pub, 'uniq) key -> 'uniq secret
  (* dechiffrement a l'aide de la cle privee *)
  val decrypt : 'uniq secret -> (priv, 'uniq) key -> string
end
```

Exemple de type unique: réalisation

- une réalisation “bidon”

```
module NullCrypt : RSA = struct
    type pub
    type priv
    type ('typ, 'uniq) key = unit
    type 'uniq secret = string
    module type KEY_PAIR = sig
      type unique
      val pubk : (pub, unique) key
      val privk : (priv, unique) key
    end
    let create_key_pair () = (module
                              struct
                                type unique
                                let pubk = ()
                                let privk = ()
                              end : KEY_PAIR)
    let encrypt by pk = by
    let decrypt se pk = se
  end
```

Exemple de type unique: réalisation

- une application classique
- garantie sans erreurs de clé

```
let _ =
  let msg = Bytes.of_string "message super important" in
  let (module Key1) = NullCrypt.create_key_pair () in
  let (module Key2) = NullCrypt.create_key_pair () in
  let msg_secret = NullCrypt.encrypt msg Key1.pubk in
  let msg_decode = NullCrypt.decrypt msg_secret Key1.privk (*erreur si mauvaise
  msg_decode = msg
```

Types algébriques généralisés: modélisation de problèmes

- on modélise, par des types, le problème *Homme-Loup-Mouton-Chou*
- *H*, sur une barque à 2 places, doit transporter *L*, *M* et *C*, de la rive gauche à la rive droite
- *M* ne peut rester seul (sans *H*) sur une rive avec *L* ou *C*

```
(* les positions: rive gauche ou rive droite *)
type g = private G
type d = private D

(* les 4 mouvements possibles des 4 entites h, l, m, c *)
type ('h, 'l, 'm, 'c, 'h1, 'l1, 'm1, 'c1) move =
  | H  : ('h, 'l, 'm, 'c, 'h1, 'l, 'm, 'c) move
  | HL : ('h, 'h, 'm, 'c, 'h1, 'h1, 'm, 'c) move
  | HM : ('h, 'l, 'h, 'c, 'h1, 'l, 'h1, 'c) move
  | HC : ('h, 'l, 'm, 'h, 'h1, 'l, 'm, 'h1) move

(* condition de securite: m ne mange pas c et n'est pas mange par l *)
type ('h, 'l, 'm, 'c) safe =
  | SafeHM : ('h, 'l, 'h, 'c) safe
  | SafeHLC : ('h, 'h, 'm, 'h) safe
```

Types algébriques généralisés: *SAT-solving*

- un chemin est soit terminé (tous sur la rive droite), soit un premier mouvement sûr suivi du chemin restant
- `exists_path` permet de tester l'absence de solution, par des clauses de **réfutation**, qui sont vérifiées par le système de types
- le dernier filtre est refusé, il existe bien une solution de longueur 7

```
(* les chemins, listes de mouvements surs *)
type ('h, 'l, 'm, 'c) path =
  | OK : (d, d, d, d) path
  | GD : (g, 'l, 'm, 'c, d, 'l1, 'm1, 'c1) move * (d, 'l1, 'm1, 'c1) safe
         * (d, 'l1, 'm1, 'c1) path -> (g, 'l, 'm, 'c) path
  | DG : (d, 'l, 'm, 'c, g, 'l1, 'm1, 'c1) move * (g, 'l1, 'm1, 'c1) safe
         * (g, 'l1, 'm1, 'c1) path -> (d, 'l, 'm, 'c) path

let exists_path (p : (g, g, g, g) path) =
  match p with
  | GD (_, _, OK) -> .
  | GD (_, _, DG (_, _, GD (_, _, OK))) -> .
  | GD (_, _, DG (_, _, GD (_, _, DG (_, _, GD (_, _, OK))))) -> .
  | GD (_, _, DG (_, _, GD (_, _, DG (_, _, GD (_, _, DG (_, _, GD (_, _, OK))))))
```

Types algébriques généralisés: résolution de problèmes

- un chemin est soit terminé (tous sur la rive droite), soit un premier mouvement sûr suivi du chemin restant
- `exists_path` permet de tester l'absence de solution, par des clauses de **réfutation**, qui sont vérifiées par le système de types
- le dernier filtre est refusé, il existe bien une solution de longueur 7

```
(* les chemins, listes de mouvements surs *)
type ('h, 'l, 'm, 'c) path =
  | OK : (d, d, d, d) path
  | GD : (g, 'l, 'm, 'c, d, 'l1, 'm1, 'c1) move * (d, 'l1, 'm1, 'c1) safe
         * (d, 'l1, 'm1, 'c1) path -> (g, 'l, 'm, 'c) path
  | DG : (d, 'l, 'm, 'c, g, 'l1, 'm1, 'c1) move * (g, 'l1, 'm1, 'c1) safe
         * (g, 'l1, 'm1, 'c1) path -> (d, 'l, 'm, 'c) path

let exists_path (p : (g, g, g, g) path) =
  match p with
  | GD (_, _, OK) -> .
  | GD (_, _, DG (_, _, GD (_, _, OK))) -> .
  | GD (_, _, DG (_, _, GD (_, _, DG (_, _, GD (_, _, OK))))) -> .
  | GD (_, _, DG (_, _, GD (_, _, DG (_, _, GD (_, _, DG (_, _, GD (_, _, OK))))))
```

Cours 6 : Monades

2024-2025 (450f02-dirty)

Introduction

Effet : tout ce que fait une fonction au-delà d'associer un résultat à une donnée en entrée.

P. ex. étant donnée une expression `e` qui s'évalue en une valeur `v`, si remplacer `e` par `v` (dans un calcul) change le comportement du programme, c'est que `e` produit aussi des effets.

Effets disponibles en OCaml

- état (y.c. d'ordre supérieur)
- exceptions
- entrées-sorties
- non-terminaison
- ...

Indisponibles

- non-déterminisme
- état polymorphe
- exceptions typées
- ...

Introduction

Pour distinguer un calcul *pur* $A \to B$ d'un calcul produisant *en plus* des effets, on donne à ce dernier un type $A \to T(B)$, où T décrit les effets à l'œuvre.

Une *monade* (Moggi 1989) $=$ un module implémentant un effet donné sous la forme de T et d'opérations sur T respectant des lois algébriques générales.

Approche monadique: les foncteurs

Interface

```
module type FONCTEUR = sig
  type 'a t
  val map : ('a -> 'b) -> ('a t -> 'b t)
end
```

Lois

$$\texttt{map id} = \texttt{id}$$

$$\texttt{map (f} \circ \texttt{g)} = \texttt{(map f)} \circ \texttt{(map g)}$$

Approche monadique: les monades

Interface

```
module type MONADE = sig
  include FONCTEUR
  (* injecte une valeur dans un calcul avec effets *)
  val return : 'a -> 'a t
  (* "bind" : composition de calculs avec effets *)
  val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
end
```

Lois

```
          map f x     =     x >>= (fun v -> return (f v))
      x >>= return    =     x
    return v >>= f    =     f v
  (f >>= g) >>= h     =     f >>= (fun x -> g x >>= h)
```

Monade “identité”

```
module Id : MONADE with type 'a t = 'a = struct
  type 'a t = 'a
  let map f x = f x
  let return x = x
  let (>>=) x f = f x
end
```

Monade “option” / “*maybe*” / “partialité” (I)

```
let mult_even xs ys =
  let trouve = List.find_opt (fun n -> n mod 2 = 0) in
  match trouve xs with
  | None -> None
  | Some x ->
    match trouve ys with
    | None -> None
    | Some y -> Some (x * y)

let xy = mult_even [ 1;3;5;8;5;4 ] [ 9;9;6;12;3 ]

val mult_even : int list -> int list -> int option = <fun>
val xy : int option = Some 48
```

Monade “option” / “*maybe*” / “partialité” (II)

```
module type MAYBE = MONADE with type 'a t = 'a option

module Maybe : MAYBE = struct
  type 'a t = 'a option
  let map f x = match x with
    | None -> None
    | Some v -> Some (f v)
  let return x = Some x
  let (>>=) x f = match x with
    | None -> None
    | Some v -> f v
end
```

Monade “option” / “*maybe*” / “partialité” (III)

```
let mult_even xs ys =
  let trouve = List.find_opt (fun n -> n mod 2 = 0) in
  let open Maybe in
  trouve xs >>= fun x ->
  trouve ys >>= fun y ->
  return (x * y)

let xy = mult_even [ 1;3;5;8;5;4 ] [ 9;9;6;12;3 ]

val mult_even : int list -> int list -> int option = <fun>
val xy : int option = Some 48
```

Bind

`>>=` est appelé “bind” car il associe une valeur à une variable pour un certain contexte, p. ex. dans `trouve xs >>= fun x -> ...`

S’il existait un opérateur `let*` (p. ex.) pouvant inspecter le contenu de ce que retourne `trouve xs`, on pourrait même écrire `let* x = trouve xs in ...`

Bind

`>>=` est appelé “bind” car il associe une valeur à une variable pour un certain contexte, p. ex. dans `trouve xs >>= fun x -> ...`

S’il existait un opérateur `let*` (p. ex.) pouvant inspecter le contenu de ce que retourne `trouve xs`, on pourrait même écrire `let* x = trouve xs in ...`

(OCaml permet de définir un tel opérateur depuis la v4.08.)

Map2

```
module type FONCTEUR = sig
  ...
  val map2 : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
end

...

module Maybe : MAYBE = struct
  ...
  let map2 f x y = match x, y with
    | None, _ | _, None -> None
    | Some x', Some y' -> Some (f x' y')
end

let mult_even xs ys =
  let trouve = List.find_opt (fun n -> n mod 2 = 0) in
  let open Maybe in
  map2 ( * ) (trouve xs) (trouve ys)
```

Monade d'environnement / "reader" (I)

```
type expr = Const of int | Var of string | Add of expr * expr

let rec eval e env = match e with
  | Const c -> c
  | Var v ->
      (* on ignore l'exception [Not_found] de [List.assoc] pour cet exemple *)
      List.assoc v env
  | Add (e1, e2) ->
      let v1 = eval e1 env in
      let v2 = eval e2 env in
      v1 + v2

let env1 = [ ("x", 13); ("t", 1); ("y", 14) ]

let e = Add (Const 1, Add (Const 2, Var "x"))

let _ = eval e env1
```

Monade d'environnement / "reader" (II)

```
module type READER = sig
  include MONADE
  type env

  (* chaque monade particulière a en plus ses fonctions propres *)

  (* récupère l'environnement *)
  val ask : env t
  (* modifie l'environnement localement pour un calcul *)
  val local : (env -> env) -> 'a t -> 'a t
  (* exécute un calcul pour un environnement donné *)
  val run : 'a t -> env -> 'a
end
```

Monade d'environnement / "reader" (III)

```
module Make_reader(Env : sig type t end): READER with type env = Env.t = struct
  type env = Env.t
  type 'a t = env -> 'a

  let return (v : 'a) : 'a t = fun _ -> v
  let ( >>= ) (x : 'a t) (f : 'a -> 'b t) : 'b t = fun env -> f (x env) env
  let map (f : 'a -> 'b) (x : 'a t) : 'b t = fun env -> f (x env)
  let map2 f (x : 'a t) (y : 'b t) : 'c t= fun env -> f (x env) (y env)

  let ask : env t = fun env -> env
  let local (ext : env -> env) (x : 'a t) : 'a t = fun env -> x (ext env)
  let run (x : 'a t) (env : env) : 'a = x env
end
```

Monade d'environnement / "reader" (IV)

```
module Reader = Make_reader(struct type t = (string * int) list end)
open Reader

let rec eval = function
  | Const c -> return c
  | Var v ->
      ask >>= fun env ->
      return (List.assoc v env)
  | Add (a, b) ->
      eval a >>= fun a' ->
      eval b >>= fun b' ->
      return (a' + b')

let _ = Reader.run (eval e) env1
```

Monade d'environnement / "reader" (V)

```
let rec eval = function
  | Const c -> return c
  | Var v -> map (List.assoc v) ask
  | Add (a, b) -> map2 ( + ) (eval a) (eval b)

let _ = Reader.run (eval e) env1

let _ =
  Reader.run
    (Reader.local (List.cons ("z", 15))
       (eval (Add (Var "x", Add (Var "y", Var "z"))))) env1
```

Permutations

- le type `'a list list` rend le programme peu clair
- abstraction des ens. de permutations difficile
- la structuration monadique est une solution

```
(* insertions : 'a -> 'a list -> 'a list list *)
let rec insertions e l =
 match l with
 | []   -> [ [e] ]                    (* insertion de e en fin           *)
 | t::q -> (e::l)                     (* insertion de e avant t          *)
           ::
           List.map (fun l -> t::l) (* on ajoute t en tete des ...      *)
                    (insertions e q) (* insertions de e apres t, i.e. dans q *)

(* permutations : 'a list -> 'a list list *)
let rec permutations l =
 match l with
 | []   -> [ [] ]
 | e::q -> List.(flatten (map (fun p -> insertions e p) (permutations q)))
```

Monade non-déterministe / “ensembliste” (I)

```
module type NON_DETERMINISTIC = MONADE with type 'a t = 'a list

module List_monad : NON_DETERMINISTIC = struct
  type 'a t = 'a list
  let map = List.map
  let map2 = List.map2
  let return x = [x]
  let rec (>>=) l f = match l with
    | [] -> []
    | t::q -> f t @ (q >>= f)
  (* ou encore : *)
  let (>>=) x f = List.flatten (List.map f x)
end
```

- quelle différence avec la monade “option” ?
- applicable aux permutations ? *oui mais… cf. inf.*

Monade non-déterministe / “ensembliste” (II)

Ici, un intérêt de l’approche monadique est de permettre de raisonner sur une seule permutation à la fois (la généralisation “non-déterministe” étant produite par `>>=`).

```
let rec permutations = function
  | [] -> return []
  | e::q -> permutations q >>= fun p -> insertions e p
```

Compréhension

Dans le cas de cette monade, `permutations q >>= fun p -> insertions e p` peut être vu intuitivement comme la définition d’une liste par “compréhension” : `[ insertions e p | p <- permutations q ]`.

PS : on ne peut toujours pas définir `insertions`.

Monades additives

Interface

```
module type MONADE_PLUS = sig
  include MONADE
  val zero : 'a t
  val (++) : 'a t -> 'a t -> 'a t
end
```

Lois (variables selon les publications)

```
          zero ++ a    =    a
          a ++ zero    =    a
       (a ++ b) ++ c   =    a ++ (b ++ c)
         zero >>= f    =    zero
 f >>= fun _ -> zero   =    zero
     (a ++ b) >>= f    =    (a >>= f) ++ (b >>= f)
```

Insertions

```
module type NON_DETERMINISTIC = MONADE_PLUS with type 'a t = 'a list

module List_monad : NON_DETERMINISTIC = struct
  ...
  let zero = []
  let (++) = (@)
end

let rec insertions e l = match l with
  | [] -> return [e]
  | t::q ->
      return (e::l)
      ++
      map (List.cons t) (insertions e q)

let rec permutations = function
  | [] -> return []
  | e::q -> permutations q >>= insertions e
```

Une monade ensembliste ? (I)

Attention, cette monade n'empêche pas la présence de doublons :

```
> permutations [ 1;2;2;3 ];;
- : int list list =
[[1; 2; 2; 3]; [2; 1; 2; 3]; [2; 2; 1; 3]; [2; 2; 3; 1]; [1; 2; 2; 3]; ...]

(* ou encore *)

> let rec insertions e l = match l with
   | [] -> return [e]
   | t::q ->
       return (e::l) ++ return (e::l) (* !!! *)
       ++
       map (List.cons t) (insertions e q)
> permutations [ 1;2;2;3 ];;
- : int list list =
[[1; 2; 3]; [1; 2; 3]; ...]
```

Une monade ensembliste ? (II)

Et si on implémente la monade avec des listes sans doublons ?

```
module NDet : NON_DETERMINISTIC = struct
  type 'a t = 'a list
  let add x l = if List.mem x l then l else x::l
  let (++) l1 l2 = List.fold_right add l1 l2
  let return x = [x]
  let zero = []
  let map f l = List.fold_right (fun x -> add (f x)) l []
  let map2 f xs ys = List.fold_right2 (fun x y -> add (f x y)) xs ys []
  let rec (>>=) x f = match x with
    | [] -> []
    | t::q -> f t ++ (q >>= f)
end
```

Pas une monade ! Pourquoi ?

Application à la recherche de solutions

- on cherche un élément d'un type `'a` qui satisfait une propriété `ok : 'a -> bool`
- on procède de proche en proche, en visitant différentes `positions`
- en utilisant une fonction `neighbors : 'a -> 'a list` qui donne les voisins d'un élément donné
- applications: rendre la monnaie, le compte est bon, labyrinthe, ...

```
(* search_list : ('a -> 'b list) -> 'a list -> 'b list *)
let rec search_list explore positions = match positions with
    | [] -> [] (* ECHEC ! *)
    | pos::queue -> match explore pos with
                    | [] -> search_list explore queue
                    | sol -> sol (* OK ! *)
(* search : ('a -> 'a list) -> ('a -> bool) -> 'a -> 'a list *)
let rec search neighbors ok pos =
  if ok pos then [ pos ] (* OK ! *) else
  match search_list (search neighbors ok) (neighbors pos) with
    | [] -> [] (* ECHEC ! *)
    | sol -> pos :: sol
```

Application de l'application: rendre la monnaie

- le type `pos` représente l'état courant du problème: la monnaie à rendre et l'ensemble des pièces disponibles dans la caisse
- le critère (`ok`) est que le montant à rendre soit nul
- les actions possibles au voisin d'une position donnée sont: soit rendre la première pièce disponible; soit "jeter" cette première pièce

```
type pos = int * int list (* monnaie à rendre, pièces dans la caisse *)
let ok (a_rendre, caisse : pos) = a_rendre = 0
let neighbors (a_rendre, caisse : pos) = match caisse with
  | []   -> []
  | p::q -> (if p <= a_rendre then [(a_rendre-p, q)] else []) @ [(a_rendre, q)]
let rendre_monnaie a_rendre caisse = (* pour affichage lisible du résultat *)
  let rec loop = function
    | [] -> failwith "impossible"
    | [ (_, _) ] -> []
    | (n1, _)::(((n2, _)::_) as q) ->
        let ps = loop q in
        let p = n1 - n2 in
        if p > 0 then p :: ps else ps
  in loop (search neighbors ok (a_rendre, caisse));;
rendre_monnaie 6 [ 1;1;2;1;2;2 ];;
- : int list = [1; 1; 2; 2]
```

Application à la recherche de solutions

1. utiliser le type `'a NDet.t` à la la place de `'a list`

   ```
   search_list : ('a -> 'b NDet.t) -> 'a NDet.t -> 'b NDet.t
   search : ('a -> 'a NDet.t) -> ('a -> bool) -> 'a -> 'a list NDet.t
   ```

2. utiliser `zero` et `++`:

   ```
   let rec search_list explore positions = match positions with
     | [] -> zero
     | pos::queue -> explore pos ++ search_list explore queue
   ```

3. reconnaître `bind`:

   ```
   let search_list explore positions = positions >>= explore
   ```

4. le traitement du résultat de `search_list` est également un `bind`:

   ```
   let rec search neighbors ok pos =
     if ok pos then return [ pos ]
     else neighbors pos >>=
          search neighbors ok >>= fun chemin ->
          return (pos::chemin)
   ```

   contrairement à la version sans monade, avec `NDet` (avec listes sans doublons), cette version renvoie *toutes* les solutions possibles !

Monades et encapsulation

- une monade `M` sert à encapsuler/représenter un “effet”, par dessus un calcul normal
- le type `'a M.t` est une valeur de type `'a`, obtenue au moyen de l’effet `M.t`
- le type `'a M.t` est le plus souvent abstrait, donc l’usage de la monade `M` est diffusif
- pour isoler l’usage de `M`, on définit une fonction `run : 'a M.t -> 'a result`, où `'a result` dépend de `'a`, mais pas de `M.t`
- la composition de monade, i.e. d’effets, est complexe, il faut *a minima* construire une bijection `'a M1.t M2.t <-> 'a M2.t M1.t`

Des monades pour tout représenter

- variable d'état (modifiable), entrées-sorties, exceptions, non-déterminisme, calcul probabiliste, environnement, journalisation, transactions, continuations, . . .
- utiles même dans un langage où les effets sont nativement présents comme OCaml
- utiles également en logique (double négation), etc
- d'autres monades seront couvertes en TD et TP

Cours 7 : Les continuations

2024-2025 (450f02-dirty)

Thèmes

- la notion de continuation, applicable à (presque) tous les langages
- la transformation CPS
- les continuations natives
- usages: inversion de contrôle, coroutines, processus utilisateurs, compilation, etc

La notion de continuation

- permet de représenter l'état d'un calcul en cours dans son contexte d'exécution, i.e. ce qui *continue* **l'exécution**
- supposons qu'on soit en train d'exécuter `fact 3` (déf. usuelle de `fact`)
- à un moment de l'exécution, on en est à `3 * (2 * (fact 1))`
- du point de vue du calcul courant (`fact 1`), la *continuation (courante)* peut être vue comme une fonction `fun fact_1 -> 3 * 2 * fact_1`
- attention : une continuation est relative à un contexte d'exécution
- p. ex. dans le programme `5 + fact 2`, la continuation de `fact 1` est `fun fact_1 -> 5 + 2 * fact_1`

Introduction de continuation: principe

- supposons `f : a -> b`, où l'on souhaite introduire une continuation
- `f` peut être appelée par `g : b -> c` quelconque
- on transforme `f` en `kf : a -> (b -> c) -> c`, intégrant ce futur
- le type `b` est devenu `(b -> c) -> c`, en fait très proche
- lien étroit avec la logique, où A est proche de $\neg\neg A = (A \to \bot) \to \bot$
- "inversion de contrôle" / principe d'Hollywood : "don't call us, we'll call you!"

Continuation-passing style

Introduction de continuation: exemples

Transformation systématique appelée *Continuation-Passing Style* (CPS)

```
> let f x = 2 * x;;
val f : int -> int = <fun>
> let g x = x + 5;;
val g : int -> int = <fun>
> let f' x k = k (2 * x);;
val f' : int -> (int -> 'a) -> 'a = <fun>
> let g' x k = k (x + 5);;
val g' : int -> (int -> 'a) -> 'a = <fun>

> let h x = g (f x);;
val h : int -> int = <fun>
(* expliciter les calculs intermédiaires *)
> let h x =
    let fx = f x in
    let gfx = g fx in
    gfx;;
val h : int -> int = <fun>
> h 6;;
- : int = 17
```

Introduction de continuation: exemples

Transformation systématique appelée *Continuation-Passing Style* (CPS)

```
> let f x = 2 * x;;
val f : int -> int = <fun>
> let g x = x + 5;;
val g : int -> int = <fun>
> let f' x k = k (2 * x);;
val f' : int -> (int -> 'a) -> 'a = <fun>
> let g' x k = k (x + 5);;
val g' : int -> (int -> 'a) -> 'a = <fun>

> let h x = g (f x);;
val h : int -> int = <fun>
(* expliciter les calculs intermédiaires puis introduire les continuations *)
> let h x =
    let fx = f x in
    let gfx = g fx in
    gfx;;
val h : int -> int = <fun>
> h 6;;
- : int = 17
```

```
> let h' x k =
    f' x (fun fx ->
      g' fx (fun gfx ->
        k gfx));;
val h' : int -> (int -> int) -> int = <fun>
> h' 6 (fun x -> x);;
- : int = 17
```

Introduction de continuation: exemples

```
> let rec fact n =
    if n <= 0 then 1
    else n * fact (n - 1);;
val fact : int -> int = <fun>
> let rec fact n =
    if n <= 0 then 1
    else
      let fact_n_1 = fact (n - 1) in
      n * fact_n_1;;
val fact : int -> int = <fun>
> let rec kfact n k =
    if n <= 0 then k 1
    else
      kfact (n - 1) (fun fact_n_1 ->
        k (n * fact_n_1));;
val kfact : int -> (int -> 'a) -> 'a = <fun>
> let fact' n = kfact n (fun fact_n -> fact_n);;
val fact' : int -> int = <fun>
```

Introduction de continuation: exemples

```
> let rec fact n =
    if n <= 0 then 1
    else n * fact (n - 1);;
val fact : int -> int = <fun>
> let rec fact n =
    if n <= 0 then 1
    else
      let fact_n_1 = fact (n - 1) in
      n * fact_n_1;;
val fact : int -> int = <fun>
> let rec kfact n k =
    if n <= 0 then k 1
    else
      kfact (n - 1) (fun fact_n_1 ->
        k (n * fact_n_1));;
val kfact : int -> (int -> 'a) -> 'a = <fun>
> let fact' n = kfact n (fun fact_n -> fact_n);;
val fact' : int -> int = <fun>
```

```
fact' 3
→ kfact 3 (fun fact_n -> fact_n)
                    k3 = id
→ kfact 2 (fun fn2 -> k3 (3 * fn2))
                      k2
→ kfact 1 (fun fn1 -> k2 (2 * fn1))
                      k1
→ kfact 0 (fun fn0 -> k1 (1 * fn0))
                      k0
→ k0 1
→ k1 (1 * 1)
→ k2 (2 * 1)
→ k3 (3 * 2)
→ 6
```

Introduction de continuation: exemples

```
> let rec fibo n =
    if n <= 1 then 1
    else fibo (n-1) + fibo (n-2);;
val fibo : int -> int = <fun>
> let rec fibo n =
    if n <= 1 then 1
    else
      let fib_n_1 = fibo (n-1) in
      let fib_n_2 = fibo (n-2) in
      fib_n_1 + fib_n_2;;
val fibo : int -> int = <fun>
> let rec kfibo n k =
    if n <= 1 then k 1
    else
      kfibo (n-1) (fun fib_n_1 ->
        kfibo (n-2) (fun fib_n_2 ->
          k (fib_n_1 + fib_n_2)));;
val kfibo : int -> (int -> 'a) -> 'a = <fun>
> let fibo' n = kfibo n (fun fib_n -> fib_n);;
val fibo' : int -> int = <fun>
```

Transformation CPS

- récursivité terminale ($-$ de pile *vs.* $+$ de mémoire)
- mieux contrôler/changer/linéariser le flôt de contrôle d'un programme
- utile par exemple pour écrire un compilateur (gestion du flôt de contrôle)

Continuations pour étendre un langage

Exemple: la sémantique des exceptions

- on a un langage composé de : valeurs (*val*), fonctions (`(fun v -> expr)`) et appel de fonction (`(fn ar)`)
- on souhaite implanter/représenter la sémantique des exceptions
- une exécution peut terminer (succès) ou lever une exception (échec)
- on ajoute 2 continuations : $\texttt{k}_-$, d'échec; $\texttt{k}_+$, de succès
- pour chacune des constructions du langage de départ
- auxquelles on ajoute le traitement des exceptions

$$
\begin{aligned}
[\![\texttt{try } expr_1 \texttt{ with Exc v -> } expr_2]\!]\ \texttt{k}_-\ \texttt{k}_+ &\triangleq [\![expr_1]\!]\ (\texttt{fun v -> } [\![expr_2]\!]\ \texttt{k}_-\ \texttt{k}_+)\ \texttt{k}_+ \\
[\![\texttt{raise (Exc v)}]\!]\ \texttt{k}_-\ \texttt{k}_+ &\triangleq \texttt{k}_-\ \texttt{v} \\
[\![\texttt{(fun v -> } expr\texttt{)}]\!]\ \texttt{k}_-\ \texttt{k}_+ &\triangleq \texttt{(fun v -> } [\![expr]\!]\ \texttt{k}_-\ \texttt{k}_+\texttt{)} \\
[\![\texttt{(}fn\ ar\texttt{)}]\!]\ \texttt{k}_-\ \texttt{k}_+ &\triangleq [\![ar]\!]\ \texttt{k}_-\ \texttt{(fun va -> } [\![fn]\!]\ \texttt{k}_-\ \texttt{(fun vf -> vf va))} \\
[\![val]\!]\ \texttt{k}_-\ \texttt{k}_+ &\triangleq \texttt{k}_+\ val
\end{aligned}
$$

Continuations natives

Continuations natives

- la transformation CPS est fastidieuse et invasive (changement du code source)
- les continuations natives existent dans de nombreux langages : `call/cc` en Scheme
- ou bien des primitives inspirées de celles-ci : `setjmp`/`longjmp` en C, `yield` en Python, Scala, Racket, Haskell, etc
- ces primitives permettent de faire comme si on avait ajouté des paramètres supplémentaires de continuation dans le code des fonctions
- on étudiera une librairie de continuations (dites délimitées)

La librairie Delimcc (Oleg Kiselyov)

- l'implantation des continuations implique la recopie d'une partie de la pile d'appels
- `new_prompt: unit -> 'a prompt`, pour créer un marqueur de pile
- `push_prompt: 'a prompt -> (unit -> 'a) -> 'a`, pour délimiter l'usage des continuations, en insérant un marqueur dans la pile
- `shift: 'a prompt -> (('b -> 'a) -> 'a) -> 'b`, pour capturer la continuation courante, i.e. la pile depuis le marqueur jusqu'à l'appel en cours
- l'exécution de :
  `push_prompt p (fun () -> ... (shift p (fun k -> expr)) ...)`
  1. suspend l'exécution en cours, lorsque doit être évaluée la sous-expression `(shift ...)` ci-dessus
  2. capture dans la variable `k` la continuation courante (notamment la pile). (appeler `(k arg)` *reprendrait* alors son évaluation sur *arg*)
  3. exécute *expr* (dépendant de `k` ou non, au choix), qui sera le résultat global de toute l'expression `push_prompt ...`
- dans d'autres langages, `push_prompt` et `shift` apparaissent sans marqueur de pile `p` et sont nommées `reset` et `shift`

Utilisation des continuations natives

- très proche des continuations explicites (mais **pas** équivalentes)
  - le type des expressions/fonctions ne change pas (pas de paramètre supplémentaire)
  - 2 types importants (et parfois $\neq$) : celui de l'expression `(push_prompt ...)` et celui de la sous-expression `(shift ...)`
  - pas d'introduction de récursivité terminale, pas d'économie de pile d'appels
- souvent appelées *exceptions reprenables (resumable exceptions)*
  - calcul interrompu, comme pour une exception
  - mais on peut le reprendre, en utilisant la continuation `k`
  - `push_prompt p (fun () -> ... (shift p (fun k -> expr)) ...)` est un "`try ... (raise (Exc k)) ... with Exc k -> expr`"
  - dans le traitement d'erreur *expr*, on peut fournir à `k` une valeur qui remplace le `(raise (Exc k))` et qui redémarre l'exécution "normalement" à l'endroit où le `raise` l'a arrêtée

Utilisation des continuations natives: factorielle

```
let fact n =
  let rec kfact n k =
    if n <= 0 then k 1
    else kfact (n-1) (fun fact_n_1 ->
           k (n * fact_n_1))
  in kfact n (fun fact_n -> fact_n)

fact 3
→ kfact 3 (fun fact_n -> fact_n)
                  k3 = id
→ kfact 2 (fun fn2 -> k3 (3 * fn2))
                  k2
→ kfact 1 (fun fn1 -> k2 (2 * fn1))
                  k1
→ kfact 0 (fun fn0 -> k1 (1 * fn0))
                  k0
→ k0 1 → k1 (1 * 1) → k2 (2 * 1)
→ id (3 * 2) → 6
```

```
let fact n =
  let p = new_prompt () in
  let rec loop n =
    if n <= 0 then shift p (fun k -> k 1)
    else n * loop (n-1)
  in push_prompt p (fun () -> loop n)

fact 3
→ push_prompt p (fun () -> loop 3)
(* marquage de la pile *)
→ loop 3
→ 3 * loop 2
→ 3 * (2 * loop 1)
→ 3 * (2 * (1 * loop 0))
→ 3 * (2 * (1 * shift p (fun k -> k 1))
      ⇝ k = fun n -> 3 * (2 * (1 * n))
→ k 1
→ 3 * (2 * (1 * 1))
→ 6
```

- seul l'usage de `k` (i.e. `(k 1)`) nécessite réellement sa capture avec `shift`
- les continuations natives construisent implicitement cette valeur

Utilisation des continuations natives: concaténation de listes

```
let rec append l1 l2 = match l1 with
  | [] -> l2
  | t1 :: q1 -> t1 :: append q1 l2
(* fonction recursive locale, sans l2 *)
let append l1 l2 =
  let rec loop = function
    | [] -> l2
    | t1 :: q1 -> t1 :: loop q1
  in loop l1
(* avec variable intermediaire *)
let append l1 l2 =
  let rec loop = function
    | [] -> l2
    | t1 :: q1 -> let app_q1_l2 = loop q1
                  in t1 :: app_q1_l2
  in loop l1
```

```
(* CPS *)
let append l1 l2 =
  let rec loop l1 k = match l1 with
    | [] -> k l2
    | t1 :: q1 ->
        loop q1 (fun app_q1_l2 ->
          k (t1 :: app_q1_l2))
  in loop l1 (fun app_l1_l2 -> app_l1_l2)
(* avec continuations natives ;
   attention, n'économise pas la pile *)
let append l1 l2 =
  let p = new_prompt () in
  let rec loop = function
    | [] -> shift p (fun k -> k l2)
    | t1 :: q1 -> t1 :: loop q1
  in push_prompt p (fun () -> loop l1)
```

- la continuation `k` correspond à la fonction `(fun l -> t1::...::tn::l)` où `[t1;...;tn]` est la liste `l1`
- tous les éléments t_i parcourus sont stockés dans la pile d'appels capturée par `k`

Utilisation des continuations natives: concaténation de listes

- usage très contraint, i.e. nécessairement `k : 'a list -> 'a list`

```
let append l1 l2 =
  let p = new_prompt () in
  let rec loop = function
    | [] -> shift p (fun k -> k l2)
    | t1 :: q1 -> t1 :: loop q1
  in push_prompt p (fun () -> loop l1)
```

- peut-on découpler le parcours de la concaténation à `l2` ?
- oui, à condition de changer les types :

```
type ('s, 'a) res = (* type de retour d'un programme sous push_prompt *)
  | Done of 'a                    (* sortie "nominale" *)
  | Wait of ('s -> ('s, 'a) res) (* sortie sur interruption (shift) *)
```

- `'s -> ('s, 'a) res` : type d'une continuation (l'appel de celle-ci doit renvoyer un `... res` comme `push_prompt ...`)
- `'s` : type retourné par `shift` ; polymorphe pour s'adapter à tout contexte (comme `raise` pour les exceptions)

Utilisation des continuations natives: concaténation de listes

```
type ('s, 'a) res =
  | Done of 'a
  | Wait of ('s -> ('s, 'a) res)

let append l1 l2 =
  let p = new_prompt () in (* doit être fait après définition de ... res *)
  let rec loop l1 = (* ne fait que parcourir l1 *)
    match l1 with
      | [] -> shift p (fun k -> Wait k)
      | t1 :: q1 -> t1 :: loop q1
  in
  let handle l1 l2 = (* concaténation à l2 hors de loop! *)
    match push_prompt p (fun () -> Done (loop l1)) with
      | Done _ -> assert false (* necessairement un shift dans loop *)
      | Wait k -> match k l2 with
                  | Done r -> r
                  | Wait _ -> assert false (* un seul shift *)
  in handle l1 l2
```

PS : `res` est inutilement général ici, la définition suivante suffirait :

```
type 'a res =
  | Done of 'a list
  | Wait of ('a list -> 'a res)
```

Application: *resumable exceptions*

- séparation des cas nominaux et des cas d'erreurs
- récupération des erreurs : compliqué avec exceptions ordinaires
- on souhaite par exemple lire la première ligne d'un fichier, avec entrée d'un autre nom et reprise en cas d'erreur
- traitements séparés:

```
(* cas nominal de lecture *)
let cas_nominal nom =
  Format.printf "tentative de lecture du fichier '%s'@." nom;
  let f = open_in (if Sys.file_exists nom then nom
                   else shift p (fun k -> BadName k)) in
  let l = input_line f in
  close_in f;
  Done l
(* cas d'erreur de lecture *)
let cas_erreur nom k =
  Format.printf "fichier '%s' inexistant, entrez un autre nom@." nom;
  k (read_line ())
```

- `shift (fun k -> BadName k)` remplace le nom du fichier à ouvrir
- `cas_nominal: string -> (string,string) res`
- `cas_erreur: string -> (string -> (...) res) -> (...) res`

Application: *resumable exceptions*

- programme principal combinant les différents traitements:

```
let lecture_ligne_interactive nom =
  match push_prompt p (fun () -> cas_nominal nom) with
  | Done l    -> l
  | BadName k ->
      match cas_erreur nom k with
        | Done l -> l
        | _ -> assert false (* pas de shift dans le traitement d'erreur *)
```

- le traitement d'erreur n'est pas réentrant
- il faudrait différents constructeurs (comme `BadName`) pour gérer des erreurs différentes
- on aurait alors la déclaration de type:

```
type ('shift1, 'shift2, ..., 'a) res = ...
```

Application: *resumable exceptions*

On souhaite maintenant lire un entier. Extension au traitement de plusieurs erreurs:

```
(* cas nominal de lecture *)
let cas_nominal nom =
  Format.printf "tentative de lecture du fichier '%s'@." nom;
  let f = open_in (if Sys.file_exists nom then nom
                   else shift p (fun k -> BadName k)) in
  let i = try input_binary_int f with _ -> shift p (fun k -> NotInt k) in
  close_in f;
  Done i
(* cas d'erreur de lecture: fichier inexistant *)
let cas_erreur_fichier nom k =
  Format.printf "fichier '%s' inexistant, entrez un autre nom@." nom;
  k (read_line ())
(* cas d'erreur de lecture: pas d'entier dans le fichier *)
let cas_erreur_entier k =
  Format.printf "contenu incorrect, entrez un entier@.";
  k (read_int ())
```

- `cas_nominal: string -> (string, string, int) res`
- `cas_erreur_fichier: string -> (string -> (...) res) -> (...) res`
- `cas_erreur_entier: (int -> (...) res) -> (...) res`

Application: *resumable exceptions*

- avec le type suivant qui permet de traiter 2 types d'erreur:

```
type ('shift1, 'shift2, 'a) res =
  | BadName of ('shift1 -> ('shift1, 'shift2, 'a) res)
  | NotInt of ('shift2 -> ('shift1, 'shift2, 'a) res)
  | Done   of 'a
```

- ou bien directement le type spécialisé:

```
type res =
  | BadName of (string -> res) | NotInt of (int -> res) | Done of int
```

- `BadName` correspond à l'erreur de fichier (pas de fichier)
- `NotInt` correspond à l'erreur de contenu de fichier (pas d'entier)
- `Done` correspond à l'exécution nominale, qui renvoie l'entier lu dans le fichier

Application: *resumable exceptions*

- les traitements d'erreurs doivent toujours rester sous la portée d'un `push_prompt`
- on a intérêt à factoriser ces traitements et éviter les `match ... with ...` imbriqués
- programme principal:

```
let traitement_erreur_fichier prog nom =
  match push_prompt p (fun () -> prog nom) with
    | BadName k -> cas_erreur_fichier nom k
    | resultat -> resultat
let traitement_erreur_entier prog =
  match push_prompt p (fun () -> prog ()) with
    | NotInt k -> cas_erreur_entier k
    | BadName _ -> assert false (* erreur recuperee plus tot *)
    | Done i   -> Done i
let lecture_ligne_interactive nom =
  match traitement_erreur_entier
        (fun () -> traitement_erreur_fichier cas_nominal nom) with
    | Done i -> i
    | _ -> assert false (* toutes les erreurs sont recuperees plus tot *)
```