Programmation Fonctionnelle

Introduction
Première utilisation de OCaml

Stefano Guerrini

LIPN - Institut Galilée, Université Paris Nord 13

Licence 3

2009–2010

Styles de programmation

- Style applicatif
  - Fondé sur l'évaluation d'expressions, où le résultat ne dépend que de la valeurs des arguments (et non de l'état de la mémoire).
  - Donne des programmes courts, faciles à comprendre.
  - Usage intensif de la récursion.
  - Langages typiques : Lisp, ML, Caml (Camlight, Ocaml).
- Style impératif
  - Fondé sur l'exécution d'instructions modifiant l'état de la mémoire.
  - Utilise une structure de contrôle et des structures de données.
  - Usage intensif de l'itération.
  - Langages typiques : Fortran, C, Pascal.
- Style objet
  - Un programme est vu comme une communauté de composants autonomes (objets) disposant de ses ressources et de ses moyens d'interaction.
  - Utilise des classes pour décrire les structures et leur comportement.
  - Usage intensif de l'échange de message (métaphore).
  - Langages typiques : Simula, Smalltalk, C++, Java.

La petite histoire

OCaml (Objective Caml) est le fruit de développements continus:

- 1975 Robin Milner propose ML comme métalangage (langage de script) pour l'assistant de preuve LCF. Il devient rapidement un langage de programmation à part entière.
- 1981 Premières implantations de ML
- 1985 Développement de Caml à l'INRIA et en parallèle, de Standard ML à Édimbourg, de "SML of New-Jersey", de Lazy ML à Chalmers, de Haskell à Glasgow, etc.
- 1990 Implantation de Caml-Light par X. Leroy and D. Doligez
- 1995 Compilateur vers du code natif + système de modules
- 1996 Objets et classes (OCaml)
- 2001 Arguments étiquetés et optionnels, variantes.
- 2002 Méthodes polymorphes, librairies partagées, etc.
- 2003 Modules récursifs, private types, etc.

Domaine d'utilisation du langage OCaml

Un langage d'usage général avec des domaines de prédilection.

- Domaines de prédilection
  - Calcul symbolique : Preuves mathématiques, compilation, interprétation, analyses de programmes.
  - Prototypage rapide. Langage de script. Langages dédiés. Programmation distribuée (bytecode rapide).
- Enseignement et recherche
- Industrie
  - Startups, CEA, EDF, France Telecom, Simulog,... Gros Logiciels
- Coq, Ensemble, ASTREE, (voir la bosse d'OCaml)
- Plus récemment, outils système
  - Unison
  - MLdonkey (peer-to-peer)
  - Libre cours (Site WEB)
  - Lindows (outils système pour une distribution de Linux)

Quelques mots clés

Le langage OCaml est:

- *fonctionnel*: les fonctions sont des valeurs de première classe, elles peuvent être retournées en résultat et passées en argument à d'autres fonctions.
- *à gestion mémoire automatique* (comme JAVA)
- *fortement typé*: le typage garantit l'absence d'erreur (de la machine) à l'exécution.
- *avec synthèse de types*: les types sont facultatifs et synthétisés par le système.
- *compilé ou interactif* (mode interactif = grosse calculette)

De plus le langage possède :

- *une couche à objets* assez sophistiquée,
- *un système de modules* très expressif.

Principale caractéristiques d'OCaml

- Langage fonctionnel
  - Une seule notion fondamentale: la notion de valeur. Les fonctions elles-mêmes sont considérées comme des valeurs.
  - Un programme est essentiellement constitué d'expressions et le calcul consiste en l'évaluation de ces expressions.
- Le système de typage
  - Le langage est typé statiquement.
  - Il possède un système d'inférence de types: le programmeur n'a besoin de donner aucune information de type: les types sont synthétisés automatiquement par le compilateur.
- Le polymorphisme paramétrique
  - Un type peut être non entièrement déterminé. Il est alors polymorphe.
  - Ceci permet de développer un code générique utilisable pour différentes structures de données tant que la représentation exacte de cette structure n'a pas besoin d'être connue dans le code en question.

Principale caractéristiques d'OCaml

- Mécanismes particuliers
  - OCaml possède un mécanisme de programmation par cas: "pattern matching" très puissant.
  - OCaml possède un mécanisme d'exceptions pour interrompre l'évaluation d'une expression. Ce mécanisme peut aussi être adopté comme style de programmation.
  - Il y a des types récursifs mais pas de pointeurs.
  - La gestion de la mémoire est différente de celle du C. OCaml intègre un mécanisme de récupération automatique de la mémoire.
- Autres aspects de OCaml
  - OCaml possède des aspects impératifs. La programmation impérative est possible mais non souhaitée.
  - OCaml interagit avec le C.
  - OCaml possède des aspects objets. Les fonctions peuvent manipuler des objets.
  - Les programmes peuvent être structurés en modules paramétrés (séparation du code et de l'interface).
  - OCaml exécute des processus léger (threads).
  - OCaml communique avec le réseau internet.

Principale caractéristiques d'OCaml

- Environnement de programmation
  - OCaml dispose de trois modes d'exécution.
    - ⋆ le mode interactif: c'est une boucle d'interaction. La boucle d'interaction d'OCaml utilise du code-octet pour exécuter les phrases qui lui sont envoyées.
    - ⋆ compilation vers du code-octet (bytecode) interprété par une machine virtuelle (comme Java).
    - ⋆ compilation vers du code machine exécuté directement par un microprocesseur.
  - OCaml dispose de bibliothèques.

Mode de compilation

- Noms des commandes
  - ocaml: lance la boucle d'interaction
  - ocamlrun: interprète de code-octet
  - ocamlc: compilateur en ligne de code-octet
  - ocamlopt: compilateur en ligne de code natif
  - ocamlmktop: constructeur de nouvelles boucles d'interaction
- Unités de compilation
  C'est la plus petite décomposition d'un programme pouvant être compilée.
  - Pour la boucle d'interaction, c'est une phrase du langage. On va voir plus loin e qu'est une phrase.
  - Pour les compilateurs en ligne, c'est un couple de fichiers (le source `.ml` et l'interface facultative `.mli`).

  Les fichiers objets en code-octet ont pour extension `.cmo`, `.cma` (bibliothèque objet), `.cmi` (interface compilée). Les fichiers objets en code natif ont les extensions `.cmx` et `.cmxa`.
  L'option `-c` indique au compilateur de n'engendrer que le code objet.
  L'option `-custom` de `ocamlrun` permet de créer un code autonome il contient le code-octet du programme plus l'interprète du code-octet).

Mode interactif

- L'option `-I catalogue` permet d'indiquer un chemin dans lequel se fera la recherche de fichiers sources ou compilés.
- La boucle possède plusieurs directives de compilation qui permettent de modifier de façon interactif son comportement. Ces directives commencent par # et se terminent par `;;`.
  - `#quit;;` sort de la boucle d'interaction
  - `#directory catalogue;;` ajoute un chemin de recherche
  - `#cd catalogue;;` change de catalogue courant
  - `#load "fichier objet";;` charge un fichier `.cmo`
  - `#use "fichier source";;` compile et charge un fichier `.ml` ou `.mli`
  - `#print depth profondeur;;` modifie la profondeur d'affichage
  - `#trace fonction;;` trace les arguments d'une fonction
  - `#untrace fonction;;` enlève cette trace
  - `#untrace all;;` enlève toute les traces

On peut lancer la boucle d'interaction sous emacs par `M-x run-caml`.

Expressions constantes

- Une fois entré dans la boucle OCaml on peut évaluer des expressions.

```
shell $ ocaml                    --> appel de la boucle
        Objective Caml version 3.11.1
# 3;;                            --> expression à évaluer
- : int = 3                      --> réponse
# #quit;;                        --> pour sortir de la boucle
shell $                          --> shell de système
```

- La réponse de l'évaluation d'une expression est sa valeur et son type.
- Les expressions plus simples sont les constants des types de base.

```
# 5.0;;
- : float = 5.
# true;;
- : bool = true
# "Hello, word!";;
- : string = "Hello, word!"
```


Les opérateur arithmétiques

- Sur les entier on a les opérateur arithmétiques infixes

```
# 3 + 4 * 2;;
- : int = 11
```

- Mais attention, les opérateur sont typés et ils ne sont pas "overloaded"

```
# 3.0 + 2.5;;
Characters 0-3:
  3.0 + 2.5;;
  ^^^
Error: This expression has type float but an expression was expected of type
         int
```

- Les opérateur arithmétiques sur les réel ont des symboles distingués.

```
# 3.0 +. 2.0;;
- : float = 5.
```

Les opérateur booléen et le if-then-else

- Les opérateurs booléens sont
  - `not`: négation
  - `&&` ou `&`: et séquentiel
  - `||` ou `or`: ou séquentiel

```
# not false && false;;
- : bool = false
# not (false && false);;
- : bool = true
# true = false;;
- : bool = false
# 3 = 3;;
- : bool = true
# 4 + 5 >= 10;;
- : bool = false
# 2.0 *. 4.0 >= 7.0;;
- : bool = true
```

- Expressions conditionnelles

```
# if (3 < 4) then 1 else 0;;
- : int = 1
#  if (3 > 4) then 1 else 0;;
- : int = 0
```

seulement une des branches then/else est évaluée.

Conversion de types

- Pour utiliser un entier dans une expression que contient un opérateur réel il faut utiliser une fonction de conversion explicite

```
# 3.0 +. 2;;
Characters 7-8:
  3.0 +. 2;;
         ^
Error: This expression has type int but an expression was expected of type
         float
# 3.0 +. float 2;;
- : float = 5.
```

- Les opérateur de comparaison `< > <= >= =` sont les mêmes pour les entiers et les réels

```
# 2.0 < 3.0;;
- : bool = true
```

```
# 2 < 3;;
- : bool = true
```

  - Mais attention, il faut toujours faire de conversion de type si nécessaire

```
# 2.0 < 3;;
Characters 6-7:
  2.0 < 3;;
        ^
Error: This expression has type int but an expression was expected of type
         float
# 2.0 < float 3;;
- : bool = true
```

Les chaînes de caractères

- literals

```
# "Hello, world!";;
- : string = "Hello, world!"
```

- concaténation

```
# "Hello, " ^ "world!";;
- : string = "Hello, world!"
```

- la bibliothèque `String`

```
# String.uppercase "Hello, world!";;
- : string = "HELLO, WORLD!"  \end{itemize}

# open String;;
# uppercase  "Hello, world!";;
- : string = "HELLO, WORLD!"
```


Les fonctions

- Les fonctions sont des expressions (`fun` est équivalent à `function`)

```
# fun x -> x * 2;;
- : int -> int = <fun>
# function x -> x *2;;
- : int -> int = <fun>
```

- Le type d'une fonction n'est plus un type de base
- Pour évaluer une fonction il faut lui donner les valeurs des arguments

```
# (fun x -> x * 2) 3;;
- : int = 6
```

- L'argument d'une fonction peut être une fonction

```
# (fun f -> f (f 2));;
- : (int -> int) -> int = <fun>
```

- Le résultat de l'évaluation d'une fonction peut être une autre fonction

```
# (fun f x -> f (f x)) (fun x -> x * x);;
- : int -> int = <fun>
```

- En ajoutant des arguments on peut évaluer la fonction obtenue

```
# (fun f x -> f (f x)) (fun x -> x * x) 2;;
- : int = 16
```

Les noms

- On peut associer un nom à des expressions pour le réutiliser après

```
# let x = 2 + 3;;
val x : int = 5
# x * 3;;
- : int = 15
```

- En particulier on peut donner un nom à les fonctions

```
# let f = (fun x -> x * 2);;
val f : int -> int = <fun>
# f 3;;
- : int = 6
```

- Pour définir une fonction OCaml fournit aussi une syntaxe simplifié

```
# let g f x = f (f (x * 2));;
val g : (int -> int) -> int -> int = <fun>
# let sq x = x * x;;
val sq : int -> int = <fun>
# let h = g sq;;
val h : int -> int = <fun>
# h 1;;
- : int = 16
```

- `let f x y = e` est équivalent à `let f = fun x y -> e`

Convention sur les noms

- Avec OCaml, les noms de variables et de fonctions doivent commencer par une minuscule.
- Les noms commençant par une majuscule sont réservés pour les constructeurs.

```
# let IsEven x = x mod 2 = 0;;
Characters 4-12:
  let IsEven x = x mod 2 = 0;;
      ^^^^^^^^
Error: Unbound constructor IsEven
# let isEven x = x mod 2 = 0;;
val isEven : int -> bool = <fun>
```

- Dans une déclaration, un nom `x` déclaré dans un `let` est connu dans toutes les expressions qui suivent la déclaration mais pas dans l'expression à la droite de `let x =`.

Porté du `let`

- Dans une déclaration, un nom `x` déclaré dans un `let` est connu dans toutes les expressions qui suivent la déclaration mais pas dans l'expression à la droite de `let x =`

```
# let z = z + 3;;
Characters 8-9:
  let z = z + 3;;
          ^
Error: Unbound value z
# let z = 2;;
val z : int = 2
# let z = z + 3;;
val z : int = 5
```

- Pour définir des fonctions récursives il faut noter qu'on est en train de donner une déclaration récursive en utilisant le construit `let rec`.

Les produits cartésiens

- Les valeurs des produits cartésiens binaires sont les couples

```
# (3, 4);;
- : int * int = (3, 4)
```

- Ce n'est pas nécessaire que les objets de la couple aient le même type

```
# let p = (true, "alpha");;
val p : bool * string = (true, "alpha")
```

- Les projections

```
# fst p;;
- : bool = true
# snd p;;
- : string = "alpha"
# p = (fst p, snd p);;
- : bool = true
```

- Les produits cartésiens n-aires

```
# (true, 3, "alpha", 4);;
- : bool * int * string * int = (true, 3, "alpha", 4)
```

  - les projections `fst` et `snd` sont prédéfinies seulement pour le cas binaire $n = 2$
  - il n'y a pas des projections prédéfinies pour chaque n
  - on peut définir les projections pour toutes les n-uples

```
# let p_1_3 (x, y, z) = x;;
val p_1_3 : 'a * 'b * 'c -> 'a = <fun>
```

Fonction *n*-aire ou sur *n*-uples

- Une fonction binaire

```
# let f x y = x + y;;
val f : int -> int -> int = <fun>
```

  en effet, c'est une fonction unaire qui renvoie une fonction unaire
- On peut définir des fonction binaire en utilisant les couples

```
# let g (x,y) = x + y;;
val g : int * int -> int = <fun>
```

  en effet, cette fonction est une fonction unaire sur une couple
- la définition précédente c'est équivalente à

```
# let g p = fst p + snd p;;
val g : int * int -> int = <fun>
```


La récursion

- Une définition de fonction peut être récursive, mais il faut noter que dans une définition récursive il faut ajouter le mot clé `rec` à `let`

```
# let rec exp x =
  if (x = 0) then 1
             else if (x mod 2 = 1) then (exp (x/2)) * (exp (x/2)) * 2
                                   else (exp (x/2)) * (exp (x/2));;
      val exp : int -> int = <fun>
```

- La définition précédente est très inefficace car elle évalue deux fois la même expression récursive dans chaque branche du cas $x > 0$.
- Pour éviter ce type de problèmes on peut définir des noms locaux

```
# let rec exp x =
  if (x = 0) then 1
             else let h = exp (x/2) in
                  if (x mod 2 = 1) then h * 2
                                   else h;;
```

- La fonction de Fibonacci

```
# let rec fib n = if n < 2 then 1 else (fib (n-1) + fib (n-2));;
val fib : int -> int = <fun>
```

Déclarations globales et déclarations locales

- Une déclaration globale associe un nom à une valeur

```
# let x = 3 * 4;;
val x : int = 12
```

- En réponse à une déclaration globale, la boucle affiche l'effet que la déclaration a eu sur l'environnement globale (en ajoute au type et à la valeur de l'expression).
- Dans l'exemple, `val x` indique que
  1. le nom `x` a été ajouté à l'environnement;
  2. que le nom `x` est associé à une valeur.
- Dans une déclaration locale

```
let x = 3 in x + 2;;
```

  le nom déclaré n'est connu que dans une expression (celle que suive le `in`).
- Une déclaration locale peut redéfinir localement un nom global

```
# let x = 2;;
val x : int = 2
# let x = 3 in x;;
- : int = 3
# x;;
- : int = 2
```


Déclarations et expressions

- Une déclaration locale est une expression et on peut l'utiliser comme une autre expression quelconque du même type

```
# 4 * let x = 2 in x + 4;;
- : int = 24
```

- Une déclaration globale ce n'est pas une expression

```
#  4 * let x = 2;;
Characters 14-16:
   4 * let x = 2;;
               ^^
Error: Syntax error
```

- Une déclaration globale peut être utiliser seulement dans des contextes particuliers, par exemple, comme commande pour la boucle.

Noms OCaml vs. variables C

- Les noms OCaml et les variables C sont des objets de natures complètement différentes.
  - En OCaml, une variable est un nom pour une valeur, alors qu'en C une variable est un nom pour une case mémoire.
  - En OCaml, il est donc hors de question de pouvoir changer la valeur d'une variable.
  - La déclaration globale (ou locale, voir plus loin) d'OCaml, n'a rien à voir avec l'affectation du C.
  - Pour obtenir des variables OCaml avec un comportement similaire aux variables C il faut utiliser
    - ⋆ des données d'un type particulier, les enregistrements à champs modifiables,
    - ⋆ et les références (ou pointers).

Déclarations simultanées

- Dans une déclaration globale ou locale on peut définir plusieurs variable au même temps et en parallèle (déclaration simultanée)

```
# let x = 3 and y = 4;;
val x : int = 3
val y : int = 4
# let x = y and y = x in (x,y);;
- : int * int = (4, 3)
```

- Dans une déclaration simultanée les expressions sont évaluées avant d'associer les noms de la déclaration aux valeurs des expressions correspondantes.
- Une déclaration simultanée n'est pas équivalent à une chaîne de déclaration

```
# let x = y in let y = x in (x,y);;
- : int * int = (4, 4)
# let y = x in let x = y in (x,y);;
- : int * int = (3, 3)
```

- Dans une chaîne de déclarations l'ordre est relevant, tandis que dans une déclaration simultanée l'ordre des déclarations n'a aucun influence sur le résultat

```
# let y = x and x = y in (x,y);;
- : int * int = (4, 3)
```

La récursion mutuelle

- Les déclarations récursives simultanées permettent de définir des fonctions mutuellement récursives

```
# let rec isOdd x = if (x = 0) then false else isEven (x-1)
and isEven x = if (x = 0) then true else isOdd (x-1);;
  val isOdd : int -> bool = <fun>
val isEven : int -> bool = <fun>
```

```
# let rec f x = if x <= 1 then 1 else g (x+2)
and g x = f (x-3) + 4;;
  val f : int -> int = <fun>
val g : int -> int = <fun>
```


Synthèse des types

- OCaml c'est un langage fortement typé
  - tout les expressions ont un type
  - les système vérifie que la construction des expressions respect le typage
  - il n'y a pas des conversions implicite de type
- Dans une expression ou dans la déclaration d'une variable il n'y a pas des indication des types (mais on verra que on pourrai l'ajouter)

En OCaml les types sont synthétisés.

- L'algorithme de typage prend une expression et
  - soit trouve un type pour l'expression
  - soit trouve qu'il y a un erreur de typage.

```
#  fun x -> x + (x & true);;
Characters 15-16:
   fun x -> x + (x & true);;
                  ^
Error: This expression has type int but an expression was expected of type
         bool
```

Synthèse du type d'une fonction

```
# fun x y -> if x then y+1 else y*y;;
- : bool -> int -> int = <fun>
```

Dans l'expression précèdent:

- la variable *y* est utilisée dans deux expression entiers (`y*y` et `y+1`) et donc c'est de type `int`;
- la variable `x` est utilisée comme conditionnel d'un `if` et donc c'est de type `bool`;
- avec ces hypothèses le if-then-else est bien typé parce que l'expression conditionnel est de type `bool` et les branches then et else ont le même type;
- le type du valeur renvoyé par le if-then-else (et donc de tout l'expression if-then-else) c'est le type de les branches then/else et donc c'est `int`;
- le type de tout l'expression `fun` est

  ```
  bool -> int -> int
  ```

  c'est-à-dire le type d'une fonction qui prend un argument de type `bool` (la variable `x`), un argument de type `int` (la variable `y`) et renvoie un résultat de type `int` (le résultat de le if-then-else).

Le polymorphisme

- Dans certain cas une expression est bien typé mais
  - le type des certains paramètres ou de la valeur de retour ne sont pas complètement précisé;
  - ces types ne sont pas nécessaire pour déterminer la valeur de l'expression à partir de ces paramètres.

```
# fun x -> x;;
- : 'a -> 'a = <fun>
# let eq x y = x = y;;
val eq : 'a -> 'a -> bool = <fun>
```

- `'a` (qu'on lit α) c'est une variable de type et indique que la fonction c'est bien définie pour un type quelconque α.
- Les fonctions dont l'un (ou plusieurs) des paramètres ou la valeur de retour est d'un type qu'il n'est pas nécessaire de préciser sont dites polymorphes.
- Le type d'une fonction polymorphe peut contenir plus d'une variable de type `'a`, `'b`, `'c`, ... (qu'on lit α, β, γ, ...)

```
# let first = function (x,y) -> x;;
val first : 'a * 'b -> 'a = <fun>
# let third (x, y, z) = z;;
val third : 'a * 'b * 'c -> 'c = <fun>
```

Utilisation de fonctions polymorphes

- Une fonction polymorphe peut être utilisé dans tour les contextes compatible avec le type de la fonction

```
# first (2,3);;
- : int = 2
# first ("toto",true);;
- : string = "toto"
```

```
# eq 2 3;;
- : bool = false
# eq true true;;
- : bool = true
```

- Si le contexte n'est pas compatible, on a un erreur de typage

```
# eq true 3;;
Characters 8-9:
  eq true 3;;
          ^
Error: This expression has type int but an expression was expected of type
         bool
```

Remarque

- Ce type d'erreur est trouvé par le compilateur avant d'évaluer la fonction.
- Le typage sert à garantir que on n'a pas des erreur à temps d'exécution du à une fonction qui reçoit une valeur de type incorrect.
- Le polymorphisme de OCaml assure une très grand flexibilité et praticité dans la définition des fonctions sans perdre les avantage du typage.

Des autres exemples de fonctions polymorphes

-
```
# let d x = (x, x);;
val d : 'a -> 'a * 'a = <fun>
# let g x = first (d x);;
val g : 'a -> 'a = <fun>
# g 3;;
- : int = 3
```

-
```
# let g = function f -> f 0;;
val g : (int -> 'a) -> 'a = <fun>
# g (function x->x+1);;
- : int = 1
# g (function x->if x=0 then "toto" else "tata");;
- : string = "toto"
```

- La fonction compose qui compose deux fonctions:

```
# let compose f g x = f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Les limites du polymorphisme en OCaml

- ```
  # let id x = x;;
  val id : 'a -> 'a = <fun>
  # ((id true),(id 2));;
  - : bool * int = (true, 2)
  ```
  1. la première occurrence de `id` a le type `bool -> bool`
  2. la seconde a le type `int -> int`
- Cependant

```
# let g f = ((f true),(f 2));;
Characters 23-24:
  let g f = ((f true),(f 2));;
                        ^
Error: This expression has type int but an expression was expected of type
         bool
```

- Les contraints qui OCaml pose au polymorphisme empêchent que le type de la fonction `f` soit polymorphe.
- La synthèse des types de OCaml ne permet pas de partir de deux type particuliers et de trouver un type plus générale:
  1. `(f true)` force que `f` soit de type `bool -> 'a`
  2. `(f 2)` force que `f` soit de type `int -> 'b`

  on ne peut pas conclure que: "`f:'a -> 'a`".

Synthèse de types polymorphes

- Si une occurrence d'une variable `f` a type τ_1 e une autre a type τ_2, le système cherche une type τ qui est un cas particulier de τ_1 et de τ_2.
- ```
  # fun f g x -> (f (g x), g (f x));;
  - : ('a -> 'a) -> ('a -> 'a) -> 'a -> 'a * 'a = <fun>
  ```
  1. dans `f (g x)` on a: `x: 'a` `g: 'a -> 'b` `f: 'b -> 'c`
  2. dans `g (f x)` on a: `x: 'd` `f: 'd -> 'e` `g: 'e -> 'f`
- Pour le typage de l'expression complète, il faut trouver des valeurs pour les variables de type tels que, pour le typage
  1. de `x`: `'a = 'd`
  2. de `g`: `'a -> 'b = 'e -> 'f` implique `'a = 'e` et `'f = 'b`
  3. de `f`: `'b -> 'c = 'd -> 'e` implique `'b = 'd` et `'c = 'e`
- Par conséquence la solution c'est

  `'a = 'd = 'b = 'f = 'e = 'c`
- Qui force

  `x: 'a` `f: 'a -> 'a` `g: 'a -> 'a`

Unification

- L'algorithme de synthèse des types est un algorithme d'unification de termes.
- Si dans une expression,
  - l'utilisation d'un identificateur x dans une contexte force le type τ_1 pour x,
  - tandis que l'utilisation de x dans une autre contexte force le type τ_2, il faut unifier τ_1 et τ_2,
  - alors il faut trouver une substitution pour les variables de type de τ_1 et τ_2 (une fonction θ qui replace toutes ou une partie des variables de τ_1 avec des autres expressions de type) tel que on obtient le même type τ en appliquant la substitution à τ_1 et τ_2 (c'est-à-dire $\theta(\tau_1) = \tau = \theta(\tau_2)$).
- Exemple:
  - $\tau_1 = (\alpha \to \beta) \to \gamma \times \alpha$
  - $\tau_2 = \delta \to \beta$
  - $\theta = \{\beta \mapsto \gamma \times \alpha, \delta \mapsto \alpha \to \gamma \times \alpha\}$
  - $\tau = \theta(\tau_1) = \theta(\tau_2) = (\alpha \to \gamma \times \alpha) \to \gamma \times \alpha$

Le type le plus général

- En utilisant l'unification, le compilateur calcule le type "le plus général" qu'on puisse donner aux identificateurs et à l'expression elle-même qui soit compatible avec les différentes constructions syntaxiques utilisées.
- Le "type plus général" τ d'un identificateur a la propriété que
  - s'il y a une autre type τ' compatible avec l'utilisation de l'identificateur,
  - alors il τ' c'est un cas particulier de τ, c'est-à-dire qu'il y a une substitution χ tel que $\tau' = \chi(\tau)$.
- Pour déterminer le type plus général l'algorithme d'unification de termes trouve toujours l'unificateur le plus général ou mgu (most general unifier), s'il existe.
- Une substitution θ qui unifie τ_1 et τ_2 (et donc $\tau = \theta(\tau_1) = \theta(\tau_2)$), c'est le mgu de τ_1 et τ_2 si
  1. pour chaque substitution θ' qui unifie τ_1 et τ_2 (et donc $\tau' = \theta'(\tau_1) = \theta'(\tau_2)$)
  2. il existe une substitution χ tel que $\chi \circ \theta = \theta'$ (et donc $\chi(\tau) = \tau'$).