Effets dans les langages fonctionnels

Ocaml vs Haskell

Martin Reynier Louis Castelbou

Sources:
https://fr.wikipedia.org/wiki/Curryfication

https://ocaml.org/learn/tutorials/functional_programming.fr.html

https://fr.wikipedia.org/wiki/Programmation_fonctionnelle

https://www-apr.lip6.fr/~chaillou/Public/DA-OCAML/book-ora028.html

http://pauillac.inria.fr/~levy/x/annexe-caml/node5.html

https://hackage.haskell.org/

https://wiki.haskell.org/Haskell

https://en.m.wikibooks.org/wiki/Haskell

1. Les effets de bords

On parle de fonction à effets de bord lorsqu'elle modifie un état en dehors de son environnement local, c'est-à-dire qu'elle a une interaction observable avec le monde extérieur autre que retourner une valeur.

La programmation impérative permet l'emploi des effets de bord dans le fonctionnement de ses programmes
A contrario la programmation fonctionnelle cherche à les minimiser et les isole souvent dans des structures prévues entre autres pour cela : les monades.

2. Le paradigme fonctionnel

La programmation fonctionnelle souhaite s'émanciper des effets de bord en interdisant toute opération d'affectation.

Un des mécanismes les plus intéressants des langages fonctionnels est l'usage des fonctions d'ordre supérieur. Une fonction est dite d'ordre supérieur lorsqu'elle peut prendre des fonctions comme arguments ou renvoyer une fonction comme résultat. On dit aussi que les fonctions sont des objets de première classe, ce qui signifie qu'elles sont manipulables aussi simplement que les types de base.

Certains langages fonctionnels autorisent la programmation impérative et donc la possibilité d'introduire localement des effets de bord. Ces langages sont regroupés sous le nom de langages fonctionnels impurs.

3. Ocaml, un langage fonctionnel impur

3.1 Application partielle et curryfication

La curryfication est la transformation d'une fonction à plusieurs arguments en une fonction à un argument qui retourne une fonction sur le reste des arguments.
C'est un concept que l'on doit à Haskell Curry.

```
# let multiply n list =
    let f x = n * x in
    List.map f list ;;
val multiply : int -> int list -> int list = <fun>
# let double = multiply 2 ;;
val double : int list -> int list = <fun>
# double [1;4;9] ;;
- : int list = [2; 8; 18]
# let multiply n = List.map (( * )n) ;;
val multiply : int -> int list -> int list = <fun>
# multiply 3 [1;2];;
- : int list = [3; 6]
```

Voici un exemple de curryfication. On utilise notamment l'application partielle de la fonction (*). On passe d'une fonction à deux arguments à une fonction à un argument.

```
# let rec map f = function
    |[] -> []
    |h::t -> f h :: map f t ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let l = [1; 2; 3] ;;
val l : int list = [1; 2; 3]
# let f x =
    x+1 ;;
val f : int -> int = <fun>
# map f l ;;
- : int list = [2; 3; 4]
```

En re-définissant la fonction map, on peut mieux constater l'utilisation de fonctions passées en arguments à d'autres fonctions, ce qui permet d'illustrer le concept des fonctions comme citoyens de première classe dans OCaml.

3.2 Les mécanismes impératifs d'OCaml.

Les références

```
# let x = ref 1 ;;
val x : int ref = {contents = 1}
# let f x =
    x:=3 ;;
val f : int ref -> unit = <fun>
# f x ;;
- : unit = ()
# x ;;
- : int ref = {contents = 3}
```

Équivalent des pointeurs en C.

Les structures itératives (Séquences)

```
# let x = ref 1 ;;
val x : int ref = {contents = 1}
# x:=!x+1;
  x:=!x*4;
  !x ;;
- : int = 8
```

La séquence permet l'évaluation ordonnée de gauche à droite d'une suite d'expressions séparées par un point-virgule.

Les structures itératives : les boucles For et While

La condition de répétition ou de sortie, d'une boucle n'a de sens que si une modification physique de la mémoire permet d'en changer la valeur. Les structures de boucle elles-mêmes sont des expressions du langage. Elles retournent donc une valeur : la constante () du type unit.

```
# let r = ref 1
  in while !r < 11 do
    print_int !r;
    print_string " ";
    r := !r+1
  done ;;
1 2 3 4 5 6 7 8 9 10
- : unit = ()
```

```
# for i=1 to 10 do
    print_int i;
    print_string " "
  done;
  print_newline(); ;;
1 2 3 4 5 6 7 8 9 10
- : unit = ()
```

4. Les effets en haskell

4.1 Les monades.

Le langage haskell étant purement fonctionnel, il interdit par définition les effets de bord.
Cependant un programme qui n'interagit pas avec son environnement se retrouve vite limité. Pour éviter cela, le langage haskell utilise des monades afin d'effectuer des effets en préservant le paradigme fonctionnel.

La monade est un type qui permet d'encapsuler des informations et de les utiliser à l'extérieur de la monade pour effectuer des opérations dessus.
Une monade est définie par un constructeur de type, une fonction return ainsi qu'une fonction bind (>>=).

```haskell
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

Elle doit aussi être une instance des classes Foncteur et Applicative.

4.2 La boucle While

Un exemple d'utilisation de monade en haskell est l'implémentation d'une boucle while.
En programmation impérative le while est dans la syntaxe même du langage.
En haskell, les boucles impliquant des effets, il faut les implémenter à l'aide de monades.

```haskell
whileM_ :: (Monad m) => m Bool -> m a -> m ()
whileM_ p f = go
    where go = do
            x <- p
            if x
                then f >> go
                else return ()
```

On voit ci dessus la façon dont while est implémenté dans la librairie Control.Monad.Loops.

4.3 Programmation Impérative avec “do”

On a vu plus haut l'implémentation de while en haskell. Cependant on n’y voit pas bien la présence de bind. C’est car ils sont imbriqués dans la syntaxe “do”.
where go = do x <- p …
correspond à
where go = p >>= (\x -> …)

Grâce à cela, on peut s'approcher d'une syntaxe impérative plus agréable tout en conservant la pureté fonctionnelle du langage.

5. Conclusion

Le style impératif et les effets de bords offrent une utilisation du langage plus intuitive.

D’un autre côté, les monades permettent de conserver les avantages d’un style purement fonctionnel.
Aussi cela offre plus de précision et de garantie avec les signature des fonctions monadiques. En effet une signature Int -> Int d’une fonction à effets en impératif n'exploite pas l'intégralité des effets de la fonction sur l’environnement. Alors qu’avec les monades nous savons qu’elle est exhaustive.