TME 9 : Interprète Forth avec des transformers

Dans ce TME nous étudions l'application pratique des *monad transformers* dans le cadre d'un petit interprète pour un langage orienté pile (*stack-based programming*) inspiré de Forth.

Contrairement au cours et au TD, l'objectif n'est pas de comprendre la conception des transformers mais plutôt leur utilisation pratique. Nous utilisons donc la bibliothèque `transformers` (cf. https://www.stackage.org/lts-22.6/package/transformers-0.6.1.0). Cette bibliothèque implémente les *transformers* que nous avons vus en cours (`IdentityT`, `MaybeT`, `WriterT`) ainsi que `StateT` (transformer pour la monade `State` vue en cours 8), `ExceptT` (transformer pour `Either err a`) pour le type d'erreur `err`, etc.

Description de l'interprète

Nous commençons par décrire l'implémentation en Haskell de l'interprète.

Architecture

L'architecture de l'interprète Forth est basée sur les modules suivants :

- le module `FState` implémente la machine Forth (`FMachine`) avec en particulier la pile de calcul `fStack`, la base des mots (*words*) `fWords` ainsi que le programme en cours d'exécution `fProg`, un simple liste d'instruction `FInstr`. Le module implémente aussi toutes les opérations élémentaires sur la `FMachine`, par exemple les opérations `fpush` et `fpop` pour la pile. Ces opérations sont pures et ne peuvent donc pas effectuer des effets de bord (dans `IO`). Cette partie du code correspond à de la programmation fonctionnelle "de base"
- le module `FInterp` implémente l'architecture principale de l'interprète, basée sur une pile de *transformers* (cf. explications ci-dessous). Ce module utilise `FState` pour tous les calculs purs, et sinon réalise les effets de bord de l'interprète (typiquement sorties sur la console) avec `IO` (contexte de base), le maintient d'état de la `FMachine` (avec `StateT`) et la gestion des erreurs (avec `ExceptT`). C'est dans ce module que l'on implémente les primitives Forth (comme `POP` ou encore `EMIT`, nous y reviendrons), donc les opérations élémentaires que l'on ne veut pas/peut pas implémenter directement en Forth.
- le module `FParser` contient la description de l'analyseur syntaxique (*parser*) pour les programmes Forth(-trans). Il utilise la bibliothèque `megaparsec` qui propose la construction des *parsers* dans un style monadique. Il s'agit d'une autre application pratique intéressante du concept de monade. En

interne `megaparsec` définit un *transformer* `ParsecT` pour pouvoir mixer le parsing et la réalisation d'opérations complexe pendant ce dernier (exemple : maintient d'un état, génération de rapports d'erreurs détaillés, pourquoi pas `IO`, etc.). Dans notre cas, on n'exploite pas ces possibilités (d'ailleurs la gestion d'erreur actuelle est un peu trop primitive).

- le module `Main` (dans `app`) contient le *main* de l'interprète, encapsulé dans le *transformer* `InputT` fourni par la bibliothèque `Haskeline` qui implémente un éditeur en mode ligne très flexible. Il s'agit d'une autre démonstration de l'utilisation pratique des *transformers*.

Fonctionnement

Du point de vue de l'utilisateur, l'interprète fonctionne avec un REPL (Read-Eval-Print-Loop) et voici une session d'exemple que vous pouvez tester après compilation.

```
ForthTrans Interpreter v0.2 -- Copyright (C) 2020-2024 PAF
----
Type !stack for showing current forth stack
Type !debug for debug mode / !nodebug to reverse
Type !reset for resetting forth machine
Type !q (or !quit) for quitting

> !stack
Machine stack: []
> 42 42 ;
 Ok.
> !stack
Machine stack: [42,42]
> :TEN 10 ;
 Ok.
> !stack
Machine stack: [42,42]
> TEN TEN TEN ;
 Ok.
> !stack
Machine stack: [10,10,10,42,42]
> !quit
Bye bye.
```

Pour comprendre le fonctionnement de Forth en détails, on pourra consulter avec intérêt la page suivante :

https://www.forth.com/starting-forth/

Mais les explications données ci-dessous permettent de comprendre les principes de base tels qu'implémentés dans le squelette.

Exemple : Empilement d'un nombre et dépilement

L'invite (*prompt*) de l'interprète attend une *phrase* en entrée.

En complément des commandes spéciales (commençant par `!`, comme `!stack` pour afficher l'état courant de la pile), les phrases Forth formées d'une suite d'instructions terminées par un point-virgule[1]. Un litéral entier est par exemple une instruction valide syntaxiquement.

Par exemple :

```
ForthTrans Interpreter v0.2 -- Copyright (C) 2020 PAF
----
...

> !stack
Machine stack: []
> 42 ;
 Ok.
> !stack
Machine stack: [42]
```

Une occurrence d'un entier correpond donc à un empilement de la valeur. Dans le code source, l'opération principal réalisant cet empilement se trouve dans le module `FInterp` :

```haskell
-- | interpretation des instructions
interpInstr :: FInstr -> Forth
interpInstr (FVal x) = hoistPure $ fpush x
interpInstr (FPrim prim) = interpPrim prim
-- ... etc ...
```

En interne l'instruction "empiler un entier `x`" se code `FVal x`. La fonction `hoistPure` permet de faire remonter une fonction sur la `FMachine` (donc une opération *pure*, sans effet de bord) qui est ici $fpushx$ pour "empiler `x`". Cette opération est implémentée dans le module `FState` dont voici un extrait :

```haskell
-- | Les types de valeurs
data FValue =
  FNone             -- pas de valeur
  | FBool Bool      -- booléens (pas en Forth standard, mais c'est mieux comme ça)
  | FInt Int        -- entiers machine
  -- etc
```

[1] Le point-virgule terminal est imposé par Forth qui date d'une époque un peu ancienne pour ce qui concerne les techniques de *parsing*. Cela s'explique également parce que Forth est un langage au *runtime* minimaliste, facile à implanter directement en *hardware*.

```haskell
-- | L'état de la machine Forth
data FMachine =
  FMachine { fStack :: [FValue]  -- <<- la pile de calcul
           , fWords :: Map String (Seq FProgram)  -- <<- la base des mots
           , fProg :: FProgram   -- <<- le programme courant
           }
  deriving (Eq)
-- ...


-- | empiler
fpush :: FValue -> FMachine -> (ForthResult, FMachine)
fpush x fm@(FMachine { fStack = xs }) = (Right FNone, fm { fStack = (x:xs) })
```

Cette fonction réalise simplement l'empilement et retourne la valeur `FNone` en retour (un `Left <erreur>` permet de retourner un code d'erreur).

Les opérations sur la `FMachine` adopte toutes une signature de la forme :

```haskell
op :: Param1 -> Param2 -> ... -> FMachine -> (ForthResult, FMachine)
```

Avec les `Param1`, `Param2`, ..., sont les paramètres potentiels de l'opération, la `FMachine` en entrée est l'état courant de la machine, et on retourne toujours un couple avec une valeur en sortie ou une erreur, et l'état suivant de la machine.

On peut voir un second exemple : `fpop` pour le dépilement :

```haskell
-- | dépiler
fpop :: FMachine -> (ForthResult, FMachine)
fpop fm@(FMachine { fStack = [] }) = (Left FErrStackEmpty, fm)
fpop fm@(FMachine { fStack = (x:xs) }) = (Right x, fm { fStack = xs })
```

On voit ici le cas d'erreur si on essaye de faire `POP` avec une pile vide. Par exemple dans l'interprète :

```
!stack
Machine stack: []
> POP ;
 Ko. Empty Stack
```

Le cas de l'instruction `POP` est un peu spécial, il s'agit d'un *mot* qui est un concept essentiel en Forth. Lorsque l'interprète reçoit une telle instruction dans la machine, sous la forme d'un `(FWord w)` (ici `(FWord "POP")`), voici l'opération réalisée (dans `FInterp`) :

```haskell
-- | interpretation des instructions
interpInstr :: FInstr -> Forth
-- ...
interpInstr (FWord w) = hoistPure $ fword w
interpInstr (FDef w prog) = hoistPure $ fdef w prog
```

(nous verrons `FDef` juste après)

avec (dans `FState`) :

```
fword :: String -> FMachine -> (ForthResult, FMachine)
fword word fm@(FMachine { fWords = fws, fProg = prog }) =
  case M.lookup word fws of
    Nothing -> (Left $ FErrNoSuchWord word, fm)
    Just Empty -> error "Empty word sequence"
    Just (p :<| _) -> (Right FNone, fm { fProg = p <> prog})
```

Cette opération un peu plus complexe va chercher dans la base des mots (une table associative) la première définition pour ce mot. Une subtilité des définitions de mots en Forth est qu'elles peuvent être redéfinies temporairement (par exemple un `POP` avec un message associé). On a choisit les séquences stricte ici pour éviter que la paresse de Haskell ne "remonte" dans notre interprète (ce qui pourrait être amusant ceci dit).

Regardons maintenant l'état initial de la `FMachine` (toujours dans `FState`) :

```
initialFMachine :: FMachine
initialFMachine = FMachine {
  fStack = []
  , fWords = M.fromList [("POP", mkSeq [[FPrim POP]])
                        ,("EMIT", mkSeq [[FPrim EMIT]])
                        ]
  , fProg = []
  }
```

On voit que le mot `POP` est préenregistré avec comme définition l'instruction `FPrim POP`. Donc `fword` place simplement comme prochaine instruction pour être exécutée ensuite. Et pour maintenant comprendre l'exécution de cette primitive, nous retournons dans le module `FInterp`.

```
-- | interpretation des instructions
interpInstr :: FInstr -> Forth
-- ...
interpInstr (FPrim prim) = interpPrim prim
-- ...
```

et voici l'interprète des primitives (toujours dans `FInterp`) :

```
-- | interprétation des primitives
interpPrim :: FPrimitive -> Forth
interpPrim POP = hoistPure $ fpop
interpPrim EMIT = do
  x <- hoistPure fpop
  liftIO $ putStr (show x)
  return FNone
```

Pour la primitive `POP`, on retourne simplement dans `FState` avec `pop`. On voit une seconde primitive `EMIT` qui réalise elle un effet de bord, essayons-là :

```
> !reset
> !stack
Machine stack: []
> 42 ;
 Ok.
> !stack
Machine stack: [42]
> EMIT ;
42 Ok.
> !stack
Machine stack: []
```

Cette fois-ci on a combiné un dépilement et un effet, l'affichage de l'entier `42` (juste avant `Ok`).

Pour résumer :

la partie pure d'une primitive (ou d'une instruction, mais on ne va pas rajouter d'instruction dans ce TME) est codée dans le module `FState` (avec des opérations `fpop`, `fpush`, ou d'autres à créer) et les effets sont codés dans une équation de `interpPrim` où on dispose de `IO` (avec `liftIO`), c'est donc en fait très simple, grâce aux *transformers* ! Bien sûr, la glue entre ces deux parties est un peu complexe, mais elle est codée une fois pour toute et ne devrait pas nécessiter trop de modification.

La dernière opération élémentaire de Forth est la définition d'un nouveau mot. C'est la seule construction syntaxique du langage et elle est très simple. Prenons l'exemple suivant :

```
> !reset
> !stack
Machine stack: []
> :TWOTENS 10 10 ;
 Ok.
> !stack
Machine stack: []
> TWOTENS ;
 Ok.
> !stack
Machine stack: [10,10]
> TWOTENS TWOTENS ;
 Ok.
> !stack
Machine stack: [10,10,10,10,10,10]
```

La phrase de définition est `: TWOTENS 10 10 ;`. Le deux-point démarre la

définition, suivi du nom du mot concerné (les mots sont en majuscules en Forth, mais notre interprète supporte aussi les miniscules) puis vient le corps de la définition qui n'est autre qu'une suite d'instructions terminée par un point-virgule. Cette définition n'a aucun impacte sur la pile mais est enregistrée dans la base des mots, comme nouvelle définition (soit en cachant une définition précédente, soit en créant une nouvelle entrée). Dans `FState` c'est la fonction `fdef` qui réalise ce travail (cf ci-dessus ou directement dans le fichier concerné, donc `src/FState.hs`).

On peut donc maintenant invoquer ce nouveau mot, qui a pour conséquence de placer deux entiers 10 sur la pile.

Et en gros voilà nous avons fait le tour des concepts de Forth. Ensuite, pour obtenir un interprète suffisamment riche, il faut ajouter des primitives, qui est le but des questions du TME.

Question 1 : primitive `DUP`

La primitive `DUP` permet de dupliquer le sommet de pile.

Par exemple :

```
> !reset
> !stack
Machine stack: []
> 42 ;
 Ok.
> !stack
Machine stack: [42]
> DUP ;
> !stack
Machine stack: [42,42]
```

En vous inspirant de la primitive `POP`, implémentez et testez `DUP`.

Question 2 : opérations arithmétiques et logiques

On souhaite ajouter les opérations arithmétiques et logiques à la machine forth :

- les mots primitifs `+`, `*`, `-` et `/` pour l'arithmétique (entière ou flottante)
- les mots primitifs `=`, `<`, `<=`, `>`, `>=`

Les primitives associées fonctionnent toutes selon les mêmes principes :

- sur la pile doit se trouver les opérandes de opérateurs : en tête de pile le deuxième opérande et ensuite le premier
- la primitive dépile les deux arguments, réalise l'opération souhaité en empile le résultat (ou signale une erreur).

Toutes ces primitives sont pures.

Voici un exemple d'utilisation :

```
> !reset
> !stack
Machine stack: []
> 42 3 -;
> !stack
Machine stack: [39]
```

Définir (et tester) ensuite en Forth (donc *pas* de façon primitive) le mot `SQUARE` qui élève le sommet de pile au carré.

```
> 8 SQUARE EMIT ;
 64 Ok.
```

Question 3 : les alternatives

Pour pouvoir faire des calculs intéressants (donc rendre notre langage Turing-compelt), il manque une primitive : les *alternatives*. En Forth c'est une opération assez complexe dont la forme est la suivante :

```
... IF <conséquent> THEN ...
```

Il s'agit de la version avec uniquement un `<conséquent>` (la partie *then* qui se trouve *avant* le mot spécial `THEN`). Le principe est simple : le `IF` dépile la tête de pile qui est soit `TRUE` soit `FALSE` (qu'il faudra également implémenter). Si c'est `TRUE` alors le `<conséquence>` est évalué, sinon le programme "saute" jusqu'à l'instruction située *après* le `THEN` (qui peut être le programme vide). Voici un exemple :

```
> 0 0 = IF "Hello" EMIT THEN "World" EMIT ;
 HelloWorld Ok.
> 1 0 = IF "Hello" EMIT THEN "World" EMIT ;
 World Ok.
```

Il existe bien sûr une variante avec alternant :

```
... IF <conséquent> ELSE <alternant> THEN ...
```

Les deux branches sont bien sûr exclusives.

```
> 0 0 = IF "Hello" EMIT ELSE "Bye" EMIT THEN "World" EMIT ;
 HelloWorld Ok.
> 1 0 = IF "Hello" EMIT ELSE "Bye" EMIT THEN "World" EMIT ;
 ByeWorld Ok.
```

On peut trouver ce style de programmation bizarre, mais quoiqu'il arrive il faut l'implémenter dans la machine.

Remarques :

- on se rappellera que le programme à exécuté est “chargé” dans la `FMachine`, donc on peut aller inspecter les instructions à l'avance.
- tous les mots qui sont utilisés comme marqueurs, par exemple `THEN` et `ELSE` doivent être défini comme des primitives qui génèrent des erreurs si elles sont exécutées.

On montrera finalement comment coder le calcul de la factorielle en Forth avec notre interprète. Par exemple :

```
> !stack
Machine stack: []
> 5 FACT ;
 Ok.
> !stack
Machine stack: [120]
```

Achievement : Turing completeness !

Défi : Compléter l'interprète Forth

Compléter l'interprète en ajoutant des primitives inspirée du document *Starting Forth*.

cf. https://www.forth.com/starting-forth/