Rapport de TIPE

Samuel GARDELLE

Mai 2021

1 Bibliographie

N°	Auteur	Titre	Source
1	Ken Thompson	Reflexions On Trusting Trust	1984 Turing Award Lecture
2	Conal Elliott	Compiling To Categories	Proceedings of the ACM on Programming Languages (ICFP)
3	Steve Awodey	Category Theory	Oxford Logic Guides (2^{nd} édition)
4	Erik Meijer et Daan Leijen	Parsec : Direct Style Monadic Parser Combinators For The Real World	User Modeling 2007, 11th International Conference

2 Introduction

L'étude de la compilation et des langages de programmation est utile du point de vue de la sécurité. S'assurer de la fiabilité d'un compilateur permet de vérifier qu'il ne déforme pas le sens de ce que le programmeur a souhaité exprimer. Cela est indispensable si l'on veut s'assurer qu'un système informatique fonctionne conformément à la manière dont on pense l'avoir conçu. S'il n'est pas fiable cela peut altérer la disponibilité, la confidentialité ou l'intégrité des données qu'il manipule.

Aussi, la simplicité du compilateur est importante. Pour deux raisons. D'abord parce que l'on peut éprouver plus facilement la fiabilité d'un compilateur simple que d'un compliqué.

Ensuite, parce que utiliser un compilateur ou un exécutable qui nous a été fourni par un tiers requiert obligatoirement qu'on lui fasse confiance. Pour ne pas dépendre d'une telle confiance il est obligatoire de soi-même re-faire son propre compilateur. Les raisons qui soutendent cette affirmation sont détaillés dans *Reflexions On Trusting Trust* de Ken Thompson (source 1).

2.1 But

L'informatique et la programmation permettent de mettre en oeuvre par des procédés physiques le calcul de fonctions mathématiques. Ce qui rend difficile ce calcul sont les limitations des machines qui l'effectuent (ordinateurs). Ces machines ne peuvent effectuer qu'un nombre limité d'opérations simples. Il faut donc traduire dans ce langage limité (appelé assembleur) la signification des fonctions mathématiques. Ce processus de traduction est la compilation.
Un programme qui automatise cette traduction est un compilateur. Pour rendre l'automatisation possible, il faut disposer d'un language standard pour exprimer des fonctions mathématiques.

Ce TIPE cherche donc à construire un compilateur simple.

Il se décompose en deux parties : d'abord une partie sur la théorie qui explique comment la compilation se déroule, puis une autre sur les aspects qui ont attrait à l'implémentation du compilateur.

Mes contributions sont les suivantes : la structure de CCC de l'assembleur pour ce qui est de la théorie, et l'implémentation en Haskell du compilateur pour ce qui est de la pratique. Les autres résultats théoriques ne sont pas de mon fait et sont sourcés.

2.2 Point de vue

Il faut choisir un langage qui sera compilé en assembleur. Ce langage doit permettre d'exprimer simplement des fonctions mathématiques calculables. On utilisera ici le λ-calcul qui se prette bien à la tâche (cf. Thèse de Church-Turing) dans sa version dite simplement typée. Il sera abrégé STLC. On ajoutera un opérateur d'addition et le type primitif $\mathbb{N}$ dont les termes sont les entiers naturels.

Il faut maintenant trouver une méthode pour le traduire en assembleur.

La théorie des catégories est une théorie qui permet d'étudier certains objets que les autres théories de l'algèbre ne traitent pas. Un tel objet est le STLC. Il est muni d'une structure particulière de catégorie cartésienne fermée (abrégé CCC). *Compiling to Categories* de Conal Elliott (source 2) utilise déjà cette propriété pour compiler le STLC vers d'autres langages qui ont cette structure de CCC.

L'idée de ce TIPE est de rechercher dans l'assembleur une structure de CCC. Ensuite, une fois cette structure exibée, on peut espérer pouvoir en déduire une compilation/transformation canonique entre les deux (un foncteur cartésien fermé dans le terminologie catégorique). Cette transformation correspond au processus de compilation.

La définition du STLC et des CCCs sont détaillées dans *Category Theory* de Steve Awodey p.140 et p.122 (source 3).

3 Structure de CCC de l'assembleur

L'assembleur que l'on va utiliser est l'LLVM IR. On ne peut pas donner de preuve du fait que l'assembleur ai la structure de CCC dans la mesure où LLVM IR n'est pas décrit par une sémantique formelle. La documentation officielle ne fait que donner un paragraphe explicatif pour chaque instruction qui fait office de "sémantique". Je décris ici la construction de la "catégorie assembleur". Le fait qu'elle vérifie bien les axiomes d'une CCC ne tiend donc qu'à l'intuition que l'on se fait des instructions.

Les objets de la "catégorie de l'assembleur" sont des noms de *registre* (r) munis d'un type (τ) (*typed register* dans la terminologie LLVM) . On les note $r : \tau$. On suppose qu'il y en a une infinité.

Les flèches sont des procédures (des suites d'instructions) qui supposent l'existence d'une valeur d'entrée (resp. de sortie) présente dans leur registre domaine (resp. codomaine).

La composition de $g : (b : \mu) \to (c : \nu)$ et $f : (a : \tau) \to (b : \mu)$ est la flèche $g \circ f : (a : \tau) \to (c : \nu)$ où $g \circ f$ est la procédure concaténée de f puis g. Pour se convaincre du caractère associatif on comprend bien que la concaténation est associative (comme le monoide libre). L'identité est la procédure qui ne fait rien (no-op). Elle est bien neutre pour la composition.

On a donc des objets, des flèches, une opération de composition associative avec un élément neutre : on a définit une catégorie.

Exhibons son caractère cartésien puis fermé. On aura alors une CCC. Les catégoriciens utilisent des diagrammes commutatifs pour décrire ces propriétés, que l'on peut retrouver en *Illustration 1* et *Illustration 2*.

Le produit de deux objets $a : \tau$ et $b : \mu$ est l'objet $a \times b : \tau \times \mu$ où $a \times b$ est un nouveau registre qui contient une paire de τ et μ. La projection π_τ (resp π_μ) est la procédure qui extrait la première composante (resp. seconde composante) de la paire et la place dans a (resp b).

Avant d'introduire l'exponentielle, il faut mentionner que LLVM permet la définition et l'appel de fonctions. On supposera que toutes les fonctions que l'on utilisera sont déjà définies dans le programme. La tâche incombe au compilateur de détecter quelles fonctions sont utilisées et de les insérer avant le reste du programme.

On dira qu'une fonction formelle est un couple contenant une adresse vers une fonction LLVM et l'adresse d'un pointeur (qui fait office de contexte comme pour une clôture lexicale). L'évaluation d'une fonction formelle est l'appel de cette fonction avec son paramètre mais aussi la valeur du pointeur. Son type est noté avec une flèche $\Rightarrow$.

Soient $y : \tau$ et $z : \mu$ deux objets : Quelque soit $x : \nu$ un objet, $g : (x \times y : \nu \times \tau) \to (z : \mu)$ une procédure : l'objet exponentiel $z^y : \tau \Rightarrow \mu$ est un registre qui contient une fonction formelle.

La transposée $\lambda g : (x : \nu) \to (z^y : \tau \Rightarrow \mu)$ est la procédure suivante : d'abord elle alloue un pointeur vers la valeur stockée dans le registre x. Ensuite elle écrit dans le registre z^y le couple contenant ce pointeur et l'adresse vers une fonction LLVM. Cette fonction LLVM est celle qui d'abord écrit dans le registre $x \times y$ le couple contenant la valeur de contexte x et celle du registre y. Ensuite cette fonction exécute g puis elle retourne la valeur contenue dans z.

L'évaluation $eval_{z^y} : (z^y \times y : (\tau \Rightarrow \mu) \times \tau) \to (z : \mu)$ est la procédure suivante : Elle lit la valeur du registre $z^y \times y$ et en extrait une fonction formelle et une valeur de type τ. Elle déréférence la valeur contenue dans le pointeur et appelle la fonction avec cette valeur et la valeur de type τ.

La transposée λg est en fait la version currifiée de g et $eval_{z^y}$ l'évaluation d'une fonction formelle. Cela définie l'exponentielle de deux objets.

4 Implémentation

Le fonctionnement du compilateur se décompose en plusieurs phases. La première phase est celle du parseur. Elle a pour tâche de transformer l'écriture d'un programme en un arbre syntaxique qui représente ce programme. La seconde est le calcul du type de ce programme et en même temps la vérification que ce programme est correctement typé. La troisième est la transformation de cet arbre qui représente un terme du STLC en un autre arbre qui correspond à la réification des CCCs. La quatrième est la génération du code : c'est la transformation de ce second arbre en une suite d'instructions qui représentent le langage de l'assembleur.

4.1 Parseur monadique

Il y a de nombreuses manières de créer un parseur. Celle que j'ai choisi d'utiliser est connue sous le nom de "parseur monadique" ou "parser combinators". Les monades sont des constructions utilisées en Haskell qui sont particulièrement adaptées pour traiter ce genre de problème. Elles permettent d'écrire des parseurs de manière simple. Le programme ressemble alors dans sa structure à la définition du langage STLC donnée avec la notation de Backus-Naur.

On retrouve dans le fichier *Parsing.hs* une librairie que j'ai fais en m'inspirant de *Parsec : Direct Style Monadic Parser Combinators For The Real World* (source 4) qui permet de créer de tels parseurs.

L'idée derrière la librairie est de concevoir un parseur qui retourne un élément de type a comme une fonction d'une chaine de caractères vers soit une erreur soit un couple (x, s) où x est l'élément de type a qui a été parsé et s le reste de la chaine de caractère qu'il reste à traiter.

```
newtype Parser a = Parser { parse :: String -> Either String (a,String) }
```

FIGURE 1 – Type d'un parseur

Les structures de monade, de foncteur, d'applicative et d'alternative qui sont définies dans le fichier *Parsing.hs* permettent de préciser comment on peut composer séquentiellement ou parrallèlement des parseurs et comment promouvoir ("lift") une fonction en une transformation de parseur. Haskell propose des notations pour les monades qui rendent alors naturel l'usage de cette librairie.

Le fichier *Parser.hs* utilise cette librairie dans le cas particulier du STLC.

4.2 Catamorphismes

Le parcours d'arbre ("top-down") se fait en général par des fonctions récursives qui déconstruisent leurs arguments selon chaque constructeur de l'arbre. Souvent, la fonction est appliquée récursivement sur les sous-arbres de chaque constructeur. Un tel algorithme est appelé catamorphisme et peut s'écrire de manière plus générale. Ce qui est spécifique à un catamorphisme particulier n'est alors que la manière dont il assemble les résultats des appels récursifs sur ses sous-arbres : son algèbre.

La librairie *recursion – schemes* propose l'opérateur *cata* qui à une algèbre associe son catamorphisme. Une algèbre étant ici une manière d'assembler les résultats des sous-arbres en un seul résultat. Par exemple, on peut implémenter l'évaluation avec un catamorphisme :

```
data Term = Add Term Term | Cst Int
data TermF f = AddF f f | CstF int
-- En pratique ce type est généré automatiquement
-- f est un foncteur

alg :: Algebra Int Int
alg (AddF a b) = a + b
alg (CstF k) = k

eval :: Term -> Int
eval = cata alg
```

FIGURE 2 – Évaluation d'un langage composé de constantes entières et de l'addition

Les catamorphismes permettent de grandement simplifier la plupart des parcours de l'arbre syntaxique. Notamment, le calcul du type d'un terme ou la substitution d'une variable par un terme en sont grandement simplifiés (cf. *Lambda.hs*).

4.3 Élimination des abstractions

L'élimination des abstractions est la procédure qui permet de passer d'un arbre syntaxique représentant un terme du STLC en un arbre syntaxique représentant une réification des CCCs. Ce processus est décrit dans *Compiling To Categories* de Conal Elliott (source 2) et fonctionne par induction en exploitant les axiomes des CCCs. Il s'implémente donc par une fonction récursive (cf. *Cat.hs*). Il s'agit de la seule fonction sur un arbre qui n'est pas implémentée avec un catamorphisme.

4.4 Génération du code

Lors de la génération du code, il faut pouvoir déclarer des fonctions et générer des noms de registre ou de fonction qui sont uniques. Ces deux effets de bord sont implémentés à l'aide de monades.

Pour accumuler les déclarations de fonctions, j'utilise la monade $Writer$ [$Declaration$]. Pour générer les noms de registres/fonctions je maintiens un couple de compteurs qui sont les noms de la dernière fonction ou registre généré. Pour générer de nouveaux noms on incrémente ces compteurs. J'utilise donc la monade $State$ (Int, Int).

Le type *Arrow* (cf. *Lib.hs*) représente les flèches de la catégorie de l'assembleur, et le générateur de code cherche à construire une flèche (une procédure) qui correspond au programme donné en entrée.

5 Limitations

Si cette manière de compiler est plus simple que d'autres il persiste des problèmes qui la rendent inutilisable en pratique. D'abord la version du λ-calcul que j'ai utilisé ne permet pas l'expression de la plupart des fonctions utiles. Cependant, en ajoutant des types somme et du pattern matching (structure co-cartésienne dans la théorie des catégories), la possibilité de définir des types, et des opérateurs de point fixe pour les termes et les types, on peut obtenir un langage de programmation utilisable. Pour ajouter ces constructions il faut identifier ce à quoi elles correspondent dans la théorie des catégories puis à quoi elles correspondent en assembleur. Cependant il persiste d'autres problèmes qui sont plus difficiles :

— la taille du code : actuellement le compilateur génère des programmes qui sont excessivement longs pour ce qu'ils font. Il serait intéressant d'étudier la capacité des passes d'optimisation d'LLVM à les réduire.

— il n'y a aucune garantie quand à la rapidité du code généré. S'il est trop long il est probablement lent, mais il pourrait aussi être lent pour d'autres raisons.

En revanche, le code du compilateur est relativement concis et découle directement de la théorie exposée. Il est donc de ce point de vue relativement simple.

6 Illustrations

Illustration 1

Propriété universelle du produit binaire :

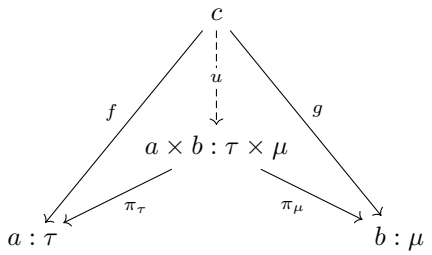
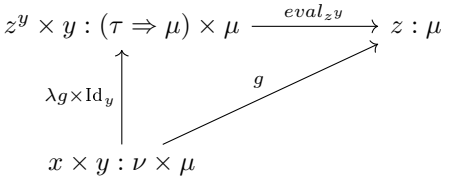


Illustration 2

Propriété universelle de l'exponentielle :



Lib.hs

```
{-# LANGUAGE DeriveFoldable, DeriveFunctor, DeriveTraversable
LambdaCase, NamedFieldPuns, TemplateHaskell, TypeFamilies #-}
module Lib where

import           Data.List                (intercalate)

-- Recursion schemes

import           Data.Functor.Foldable.TH

type Algebra f a = f a -> a

-- Types

type Gamma = [(Char,Type)]
type Typer = Gamma -> Type -- Une fonction qui résoud un type à partir d'un contexte

data Type = TFun Type Type
```

```
    | TInt
    | TProd Type Type deriving Eq

(~>) a b = TFun a b
(×) a b = TProd a b
infixr 6 ~>
infixr 7 ×

makeBaseFunctor ''Type

-- Lambda-termes

data Term = Lambda Type Char Term
    | Var Char
    | Add Term Term
    | Lit Int
    | App Term Term
    | Pair Term Term
    | PLeft Term
    | PRight Term deriving (Eq, Show)

makeBaseFunctor ''Term

-- Language des catégories

data Cat = Id Type
    | Compose Cat Cat
    | PiLeft Type
    | PiRight Type
    | Fork Cat Cat
    | AddC
    | CstC Int Type
    | AppC Type
    | Curry Cat
    | UnCurry Cat deriving (Show, Eq)

makeBaseFunctor ''Cat

-- Assembleur

type Register = (Int,Type)
data Arrow = Arrow
    { domain    :: Register
    , procedure :: [IST]
    , codomain  :: Register
    }
```

```
identity :: Register -> Arrow
identity r = Arrow r [NoOp] r

instance Semigroup Arrow where
  Arrow d1 p1 c1 <> Arrow d2 p2 c2 = Arrow d1 (p1 ++ [Move c1 d2] ++ p2) c2

data Val = Constant Int
    | Couple Val Val
    | RegContent Register
    | Parameter deriving Show

type LLVMFun = Int
type Declaration = (LLVMFun,[IST])
type FormFun = (LLVMFun,Int)

data IST = Assign Register Val
    | Move Register Register
    | Return Val
    | CallInto Val Val Register
    | AllocInto Register Type Register
    | DerefPointerInto Register Register
    | ProjLeftInto Register Register
    | ProjRightInto Register Register
    | AddInto Register Register Register
    | NoOp

type Program = (Arrow, [Declaration])
```

Main.hs

```
module Main where

import Lib
import Lambda
import Parsing
import Parser
import Cat
import CodeGen

import Control.Monad (forM)

-- TODO : de meilleurs messages d'erreur
```

```
main :: IO ()
main = do
  program <- getLine
  putStrLn ""

  let parsed = case parse expr program of
                Right (parsed,_) -> parsed
                Left e -> error e

  print parsed
  putStrLn "Parsing : OK.\n"

  putStrLn "Vérification du typage..."
  let ptype = resolve parsed
  print ptype
  putStrLn "Vérification du typage : OK.\n"

  putStrLn "Conversion..."
  let cat = convert parsed
  print cat
  putStrLn "Conversion : OK.\n"

  putStrLn "Génération du code..."
  let (arr,decls) = codegen cat

  putStrLn $ show arr
  putStrLn ""

  forM decls $ \d -> do
      putStrLn ""
      putStrLn $ show d

  putStrLn "Génération du code : OK.\n"
```

CodeGen.hs

```
{-# LANGUAGE FlexibleContexts, LambdaCase, RecordWildCards #-}
module CodeGen where

import           Data.Functor.Foldable

import           Control.Monad.State
```

```
import           Control.Monad.Writer
import           Control.Monad.State.Lazy
import           Control.Monad.Writer.Lazy

import           Cat
import           Lib

codegen :: Cat -> Program
codegen c =
  let g = gen c in
  let g' = evalStateT g (0,0) in
  let (arr,w) = runWriter g' in
  (arr,w)

type MyState = (Int,Int)
-- (i,j) | i = dernier registre, j = dernier nom de fonction

-- Un DSL pour la génération de code

createRegister :: MonadState MyState m => Type -> m Register
createRegister t = do
    modify (\(i,j) -> (i+1,j))
    i <- fst <$> get
    return $ (i,t)

createRegPair :: MonadState MyState m => Type -> Type -> m (Register,Register)
createRegPair tau mu = (,) <$> createRegister tau <*> createRegister mu

declareLLVMFun :: (MonadWriter [Declaration] m, MonadState MyState m) =>
                  [IST] -> m LLVMFun
declareLLVMFun proc = do
    modify (\(i,j) -> (i,j+1))
    j <- snd <$> get
    tell [(j,proc)]
    return j

createArrow :: MonadState MyState m =>
               Type -> Type -> (Register -> Register -> m [IST]) -> m Arrow
createArrow domType codomType builder = do
    (dom, codom) <- createRegPair domType codomType
    procedure <- builder dom codom
    return $ Arrow dom procedure codom

registerType :: Register -> Type
registerType = snd
```

```
-- Génération du code

gen :: (MonadWriter [Declaration] m, MonadState MyState m) => Cat -> m Arrow
gen = cata genM

genM :: (MonadWriter [Declaration] m, MonadState MyState m) => Algebra CatF (m Arrow)
genM = \case

  IdF t -> identity <$> createRegister t
  ComposeF g f -> (<>) <$> f <*> g

  PiLeftF (TProd a b) ->
      createArrow (TProd a b) a $ \dom codom ->
          return [ProjLeftInto dom codom]
  PiRightF (TProd a b) ->
      createArrow (TProd a b) b $ \dom codom ->
          return [ProjRightInto dom codom]

  ForkF f g -> do

    Arrow domF procF codomF <- f
    Arrow domG procG codomG <- g

    let a = registerType domF
        b = registerType codomF
        c = registerType codomG

    createArrow a (b × c) $ \dom codom ->
        return $ [Move dom domF, Move dom domG]
                 ++ procF ++ procG
                 ++ [Assign codom (Couple (RegContent codomF) (RegContent codomG))]

  AddCF -> createArrow (TInt × TInt) TInt $ \dom codom -> do
              (regA, regB) <- createRegPair TInt TInt
              return $
                    [ProjLeftInto dom regA,
                    ProjLeftInto dom regB, AddInto regA regB codom]

  CstCF k t -> createArrow t TInt $ \dom codom -> return [Assign codom (Constant k)]

  AppCF t@(TProd (TFun _ b) a) ->
    createArrow t b $ \dom codom -> do
      formFun <- createRegister (TInt × TInt)
      (fnAddr, ptrAddr) <- createRegPair TInt TInt
      param <- createRegister a
```

```
    return [ ProjLeftInto dom formFun,
             ProjRightInto dom param,

             ProjLeftInto formFun fnAddr,
             ProjRightInto formFun ptrAddr,

             CallInto (RegContent fnAddr) (Couple (RegContent param)
                  (RegContent ptrAddr)) codom
           ]

CurryF c -> do
  Arrow dom proc codom <- c

  let (TProd a b) = registerType dom
      c = registerType codom

  (ra,rb) <- createRegPair a b
  ptr <- createRegister TInt
  param <- createRegister (a × TInt)

  llvmfun <- declareLLVMFun $ [

        Assign param Parameter,
        ProjLeftInto param ra,
        ProjRightInto param ptr,

        DerefPointerInto ptr rb,
        Assign dom (Couple (RegContent ra) (RegContent rb))
      ] ++ proc ++ [ Return (RegContent codom) ]

  createArrow a (b ~> c) $ \dom codom -> do
        param <- createRegister a
        ptr <- createRegister TInt
        ret <- createRegister (TInt × TInt)
        return $ [
            Assign param Parameter,
            AllocInto param a ptr,
            Assign ret (Couple (Constant llvmfun) (RegContent ptr))
            ]

UnCurryF c -> do

  Arrow dom proc codom <- c
  let a = registerType dom
      (TFun b c) = registerType codom
```

```
    Arrow dom' proc' codom' <- gen $ AppC ((b ~> c) × b)
    param <- createRegister (a × b)
    rb <- createRegister b

    createArrow (a × b) c $ \dom codom -> return $ [
        Assign param Parameter,
        ProjLeftInto param dom,
        ProjRightInto param rb
      ] ++ proc ++
      [ Assign dom' (Couple (RegContent codom) (RegContent rb)),
        Return (RegContent codom)
      ]
```

Parser.hs

```
{-# LANGUAGE LambdaCase #-}
module Parser where

import           Control.Applicative ((<|>))
import           Lambda
import           Lib
import           Parsing
import           Prelude             hiding (lambda, not)


expr :: Parser Term
expr = Parser $ \case
    [] -> Left "Fin de chaine atteinte"
    x -> parse work x
  where work = do
          (chainl1 (unary <|> nullary) parseOp) <|> paren expr


parseType :: Parser Type
parseType = (chainr1 simpleKinds (char '>' >> return TFun))
  where simpleKinds = (char 'I' >> return TInt)
                      <|> (do { char '(' ; a <- parseType
                              ; char ',' ; b <- parseType
                              ; char ')' ; return $ a × b})
                      <|> paren parseType


-- Constructions nullaires
```

```
nullary :: Parser Term
nullary = nmbr <|> var

nmbr = Lit <$> natural
var = Var <$> alpha

-- Constructions unaires

unary :: Parser Term
unary = lambda <|> pleft <|> pright <|> paren expr

pleft = prefix "<" >> PLeft <$> expr
pright = prefix ">" >> PRight <$> expr
lambda = paren $ do
  char '\\'
  x <- alpha
  char ':'
  t <- parseType
  char '.'
  e <- expr
  return $ Lambda t x e

-- Constructions binaires

operations = [
    (Add,'+'),(App,' '),(Pair,',')
  ]

parseOp = anyOf $
  map (\(c,symb) -> char symb >> return c) operations
```

Parsing.hs

```
{-# LANGUAGE LambdaCase #-}
module Parsing where

import           Control.Applicative
import           Control.Monad
import           Data.Bifunctor

newtype Parser a = Parser { parse :: String -> Either String (a,String) }
```

```
instance Functor Parser where
  fmap f p = Parser $ second (first f) . parse p

instance Applicative Parser where
  pure x = Parser $ \s -> Right (x,s)
  a <*> b = Parser $ \s ->
    case parse a s of
      Right (f,rem) -> parse (f <$> b) rem
      Left err      -> Left err

instance Monad Parser where
  a >>= f = Parser $ \s ->
    case parse a s of
      Right (x,rem) -> parse (f x) rem
      Left err      -> Left err

instance Alternative Parser where
  empty = Parser . const $ Left "Erreur"
  a <|> b = Parser $ \s ->
    case parse a s of
      Left err -> parse b s
      x        -> x

instance MonadFail Parser where
  fail err = Parser $ \s -> Left err

instance MonadPlus Parser

prefix :: String -> Parser String
prefix pattern = Parser $
  \s -> case splitAt (length pattern) s of
          ([],[])              -> Left "Fin de chaine atteinte"
          (a,r) | a == pattern-> Right (a,r)
          _                    -> Left $ "Erreur, il était attendu : " ++ pattern

char :: Char -> Parser Char
char c = head <$> prefix [c]

anyOf :: [Parser a] -> Parser a
anyOf (x:xs) = x <|> anyOf xs
anyOf []     = empty

anyChar :: Parser Char
anyChar = Parser $ \case
  [] -> Left "Aucun charactère restant"
  x:xs -> Right (x,xs)
```

```
option :: Parser a -> Parser (Maybe a)
option p = (Just <$> p) <|> (pure Nothing)

charOf :: String -> Parser Char
charOf l = (anyOf . (map char) $ l) <*< (l ++ " étaients attendus")
digit = (charOf $ map (head.show) [0..9]) <*< "Un chiffre était attendu"
alpha = (charOf $ ['a'..'z'] ++ ['A'..'Z']) <*< "Une lettre était attendue"
alphaNum = (alpha <|> digit) <*< "Une lettre ou un chiffre était attendu"

many1 :: Parser a -> Parser [a]
many1 p = Parser $ \s ->
  case parse (many p) s of
    Right ([],_) -> Left "Élément non trouvé"
    x            -> x


natural :: Parser Int
natural = (read <$> many1 digit) <*< "Un entier naturel était attendu"

paren :: Parser a -> Parser a
paren p = do { char '(' ; r <- p ; char ')' ; return r }

identifier = many1 (digit <|> (charOf $ ['a'..'z']))

(<*<) :: Parser a -> String -> Parser a
(<*<) p err = Parser $ \s -> first (const err) (parse p s)
infixr 9 <*<

-- ce qui suit provient de la publication
-- "Parsec: Direct Style Monadic Parser Combinators For The Real World"

chainr1 p op = scan
  where scan = do { x <- p ; rest x }
        rest x = do { f <- op ;
                      y <- scan ;
                      ; return (f x y) } <|> return x

chainl1 p op = do { x <- p ; rest x }
  where rest x = do {
                  f <- op ;
                  y <- p ;
                  rest (f x y) } <|> return x
```

```
{-# LANGUAGE LambdaCase #-}
module Lambda (substituteTerm, resolve, resolveWithCtx) where

import           Data.Functor.Foldable
import           Data.List             (intersperse)
import           Lib

-- Substitution des termes

substituteTerm :: Term -> Char -> Term -> Term
substituteTerm t x = cata (subTerm t x)

subTerm :: Term -> Char -> Algebra TermF Term
subTerm e x = \case
  VarF x' | x == x' -> e
  LambdaF t a b | x == a -> error "Substitution de terme ambigue"
  t -> embed t

-- Typage des termes

resolve :: Term -> Type
resolve t = (cata typer t) []

resolveWithCtx :: Term -> Gamma -> Type
resolveWithCtx = cata typer

typer :: Algebra TermF Typer

typer (VarF x) = \l -> case lookup x l of
    Just t -> t
    _      -> error $ "La variable " ++ show x ++ " est libre."
                      ++ "\nPeut être confondez-vous avec les variables suivantes : "
                      ++ (intersperse ',' (map fst l))

typer (AddF a b) = \l -> case (a l,b l) of
                  (TInt, TInt) -> TInt
                  _ -> error "L'addition attend des opérandes entières"

typer (LitF k) = const TInt

typer (LambdaF t x y) = \l -> t ~> y ((x,t):l)

typer (AppF f x) = \l ->
  let tf = f l
      tx = x l in
```

```
  case tf of
    TFun a b | tx == a -> b
    _ -> error $ "Application illégale de " ++ show tf ++ " à  " ++ show tx

typer (PairF a b) = \l -> (a l) × (b l)

typer (PLeftF p) = \l ->
  case p l of
    TProd a _ -> a
    _         -> error "La projection gauche attend un type produit"

typer (PRightF p) = \l ->
  case p l of
    TProd _ b -> b
    _         -> error "La projection droite attend un type produit"
```

Cat.hs

```
{-# LANGUAGE LambdaCase #-}
module Cat where

import           Data.Functor.Foldable
import           Lambda
import           Lib

-- Résolution des types

resolveC :: Cat -> Type
resolveC = cata resC

resC :: Algebra CatF Type
resC = \case
  IdF t -> t ~> t
  ComposeF (TFun tc td) (TFun ta tb ) | tb == tc -> ta ~> td

  AppCF (TProd (TFun x y) a) | x == a -> ((x ~> y) × x) ~> y
  CurryF (TFun (TProd a b) c) -> a ~> b ~> c
  UnCurryF (TFun a (TFun b c)) -> a × b ~> c

  ForkF (TFun tx ty) (TFun tx' ty') | tx == tx' -> tx ~> (ty × ty')
  PiLeftF (TProd a b) -> a × b ~> a
  PiRightF (TProd a b) -> a × b ~> b
```

```
  AddCF -> (TInt × TInt) ~> TInt
  CstCF _ t -> t ~> TInt

-- Convertion STLC -> CCC

convert :: Term -> Cat

convert (Lambda tx x e) = elimAbs tx x e
convert _ = error "Seules les abstractions (les flèches) peuvent être converties"

-- Elimination des abstractions

elimAbs :: Type -> Char -> Term -> Cat

elimAbs tx x (Var v)
  | v == x = Id tx
  | otherwise = error $ "Variable libre: " ++ show v

elimAbs tx x (App u v) =
    Compose (AppC tapp) (Fork u' v')
  where tapp = undefined
        u' = elimAbs tx x u
        v' = elimAbs tx x v

elimAbs tx x (Lambda ty y u) = Curry $ elimAbs (tx × ty) x u'
  where u' = (substituteTerm (Var x) '!') .
             (substituteTerm (PLeft $ Var '!') x) .
             (substituteTerm (PRight $ Var '!') y) $ u

elimAbs tx x (Add u v) =
    Compose AddC (Fork u' v')
  where u' = elimAbs tx x u
        v' = elimAbs tx x v

elimAbs tx x (Lit n) = CstC n tx

elimAbs tx x (Pair u v) = Fork u' v'
  where u' = elimAbs tx x u
        v' = elimAbs tx x v

elimAbs tx x (PLeft u) = Compose (PiLeft tu) u'
  where u' = elimAbs tx x u
        tu = resolveWithCtx u [(x,tu)]
elimAbs tx x (PRight u) = Compose (PiRight tu) u'
  where u' = elimAbs tx x u
        tu = resolveWithCtx u [(x,tx)]
```