Laboratorul 6 – Lambda calcul cu definiții

În acest laborator vom adăuga codările Church din laboratorul 5 la interpretorul la care am ajuns la finalul laboratorului 4. Astfel vom obține un interpretor echivalent ca putere de expresie și de calcul cu cel din primul laborator.

Pentru a adăuga codările Church la limbajul deja definit, vom încerca să le încărcăm dintr-un fișier. Pentru a fi mai apropiate de definițiile din Haskell, vom folosi definiții care arată în felul următor:

```
id a := a ;
const x y := x ;
flip f x y := f y x ;
. g f x := g (f x) ;
```

Comparați acestea cu definițiile din `MyPrelude.hs` din Laboratorul 5 (am eliminat signaturile funcțiilor):

```
id a = a
const x _ = x
flip f x y = f y x
(g . f) x = g (f x)
```

Observați că:

- numele identificatorului este urmat de o listă de variabile pentru argumentele sale
- folosim `:=` în loc de `=` pentru operatorul de definire
- folosim `;` pentru a separa definițiile
- deși putem defini nume de operatori, aceștia sunt prefix, ca orice funcție

Pentru tipurile cu codări Church, putem folosi instanța corespunzătoare codificării Church pentru constructori și funcția de agregare, și celelalte funcții dependente de clasă pentru definiții coresunzătorare în limbajul MiniHaskell. De exemplu, codarea Church pentru Bool,

```
newtype CBool = CBool { getCBool :: forall a. a -> a -> a}

instance BoolClass CBool where
  true = CBool (\f t -> t)
  false = CBool (\f t -> f)
  bool f t b = getCBool b f t
```

poate fi tradusă în `MiniHaskell` astfel:

```
true f t := t ;
false f t := f ;
bool f t b := b f t ;
```

Observați că, deoarece nu mai avem nevoide de constructorul `CBool` (era folosit în Haskell doar pentru a împacheta tipul și a putea defini instanța de `BoolClass`), putem scrie `f` și `t` direct ca argumente ale definițiilor funcțiilor `true` și `false`. Tot din același motiv, nu mai avem nevoie de "proiecția" `getBool`.

Odată ce avem funcțiile din clasă definite, putem pur și simplu traduce celelalte funcții care au clasa ca dependință. De exemplu,

```
-- | if-then-else
ite :: BoolClass b => b -> a -> a -> a
ite b t e = bool e t b
```

se poate traduce în `MiniHaskell` astfel:

```
ite b t e := bool e t b ;
```

In cele ce urmează vom defini sintaxă și mod de folosire pentru definiții ca acestea.

Elemente noi de analiză lexicală și sintactică

Fără `Parsec`

Dacă nu folosiți Parsec, următoarele definiții trebuie adăugate la modulul de bază pentru analiza sintactică. Dacă folosiți Parsec, ele sunt deja disponibile.

```
-- | parsează virgulă, eliminând spațiile de după
semi :: Parser ()
semi = reserved ";"

-- | una sau mai multe instanțe, separate de punct-și-virgulă,
--   cu eliminarea spațiilor de după fiecare punct-și-virgulă
--   intoarce lista obiectelor parsate
semiSep1 :: Parser a -> Parser [a]
semiSep1 p
  = do
    a <- p
    as <- many (semi *> p)
    return (a : as)
```

```
-- | citește un fișier și aplică analiza sintactică specificată asupra
--    conținutului său
parseFromFile :: Parser a -> FilePath -> IO (Either String a)
parseFromFile parser file
  = do
    str <- readFile file
    case apply parser str of
      [] -> return $ Left "Cannot parse"
      (a,_):_ -> return $ Right a
```

Exercițiu (excluderea : ca simbol de operație)

Deoarece vrem să folosim `:=` ca separator între capul și corpul unei definiții de funcție, modificați definiția funcției `haskellOp` din modulul `Parsing` pentru a exclude simbolul `:` dintre simbolurile acceptate pentru operatori.

Acest lucru va ajuta ca analiza sintactică să aibă mai puține posibiltăți de ambiguitate.

Definiții

Vom defini un nou modul, `Program` cuprinzând următoarele:

- tipuri pentru reprezentarea definițiilor,
- analizor sintactic pentru definiții și programe
- tipuri pentru medii de evaluare și funcții pentru popularea lor
- evaluarea expresiilor folosind medii de evaluare

```
module Program where
import Exp
import Lab2 ( Parser, endOfInput, whiteSpace, reserved, semiSep1 )
import Parsing ( expr, var, parseFirst )
import Sugar ( desugarExp, desugarVar )
import Eval ( substitute )

import Control.Applicative ( Alternative(..) )
import System.IO ( stderr, hPutStrLn )
import qualified Data.Map.Strict as Map
```

Așa cum am spus mai sus, o definiție este formată din

- numele identificatorului care se definește (`defHead`);
- o listă de agumente pentru acel identificator (`defArgs`); și
- corpul funcției (`defBody`)

```
data Definition = Definition
  { defHead :: Var
  , defArgs :: [Var]
  , defBody :: ComplexExp
  }
  deriving (Show)
```

Exercițiu (analizor sintactic)

Scrieți un analizor sintactic pentru definiții.

```
definition :: Parser Definition
definition = undefined

-- >>> parseFirst definition "id := \\x -> x"
-- Just (Definition {defHead = Var {getVar = "id"}, defArgs = [], defBody
= CLam (Var {getVar = "x"}) (CX (Var {getVar = "x"}))})

-- >>> parseFirst definition "id x := x"
-- Just (Definition {defHead = Var {getVar = "id"}, defArgs = [Var {getVar
= "x"}], defBody = CX (Var {getVar = "x"})})

-- >>> parseFirst definition "const x y := x"
-- Just (Definition {defHead = Var {getVar = "const"}, defArgs = [Var
{getVar = "x"},Var {getVar = "y"}], defBody = CX (Var {getVar = "x"})})
```

Scrieți, de asemenea, un analizor sintactic pentru programe. Programele sunt secvențe de definiții separate prin `;` și terminate tot printr-un simbol `;`. Deoarece `program` va fi folosit pentru analiza sintactică a unui întreg fișier, asigurați-vă că săriți peste spațiile de la început și că nu va mai rămâne conținut nenalizat în urma recunoașterii unui program.

```
program :: Parser [Definition]
program = undefined

-- >>> parseFirst program "    id x := x ; const x y := x"
```

```
-- Nothing

-- >>> parseFirst program "    id x := x ; const x y := x ;"
-- Just [Definition {defHead = Var {getVar = "id"}, defArgs = [Var {getVar
= "x"}], defBody = CX (Var {getVar = "x"})},Definition {defHead = Var
{getVar = "const"}, defArgs = [Var {getVar = "x"},Var {getVar = "y"}],
defBody = CX (Var {getVar = "x"})}]
```

Evaluare folosind definiții

Exercițiu (funcția asociată unei definiții)

Definiți o funcție `definitionExp` care asociază un termen de tipul `ComplexExp` unuia de tipul `Definition` legând variabilele din defArgs prin lambda abstracții:

```
definitionExp :: Definition -> ComplexExp
definitionExp def = undefined

-- >>> definitionExp (Definition {defHead = Var {getVar = "const"},
defArgs = [Var {getVar = "x"},Var {getVar = "y"}], defBody = CX (Var
{getVar = "x"})})
-- CLam (Var {getVar = "x"}) (CLam (Var {getVar = "y"}) (CX (Var {getVar
= "x"})))
```

Exercițiu (mediu de evaluare)

Definim tipul unui mediu de evaluare ca o asociere de la nume de variabile la expresii.

```
type Environment = Map.Map IndexedVar Exp
```

Pentru a putea lucra cu `Map`-uri având chei de tipul `IndexedVar`, este necesar să avem o relație de ordine pe tipul `IndexedVar`. Pentru aceasta este suficient ca în fișierul `Exp.hs` să adăugați `Ord` la declarația `deriving` pentru `IndexedVar`.

Definiți o funcție care dintr-o listă de definiții (program) obține un `Environment` în care fiecărui identificator de funcție îi este asociată definiția de funcție obținută prin `definitionExp`. Observații:

- atât variabila cât si funcția vor trebui să fie "desugared" în variabile indexate /expresii simple.
- puteți folosi funcția `fromList` pentru a crea `Map`-ul.

```
programEnv :: [Definition] -> Environment
programEnv pgm = undefined
```

Exercițiu (normalizare folosind definiții)

Definiți o funcție de normalizare pentru expresii, similară cu funcția `normalize` din modulul `Eval`, dar care are ca argument auxiliar un `Environment` conținând contextul de definiții în care va fi evaluată expresia.

Singura diferență față de `normalize` va fi că atunci când se va încerca un pas de evaluare pentru o variabilă, în loc de a eșua ca mai înainte, se va verifica mai întâi dacă variabila este în contextul de definiții, și în caz afirmativ, pasul de evaluare va avea succes cu valoarea corespunzătoare variabilei.

```
normalizeEnv :: Environment -> Exp -> Exp
normalizeEnv = undefined
```

Exercițiu (Modulul `Main`)

Modificați definiția funcției `main` astfel:

- delegați `main` către o nouă funcție `execute` care are aceeași definiție ca `main`, dar care are un argument de tipul `Environment` (inițial `empty`) în plus, argument care va fi pasat la apelurile recursive ale lui `execute` (care vor înlocui apelurile recursive ale lui `main`)
- inlocuiți apelul către `normalize` cu unul către `normalizeEnv` care va folosi contextul de definiții pasat drept argument
- implementați funcționalitatea pentru comanda `:l` (`Load`) astfel
  - folosiți `parseFromFile` interpreta programul din fișierul dat ca argument lui `Load` și a obține lista de definiții
  - transformați lista de definiții într-un `Environment`
  - transmiteți acest environment funcției `execute` printr-un nou apel recursiv

Aceste modificări vor permite încărcarea fișierului `prelude.mhs` în timpul execuției și evaluarea expresiilor folosind contextul de definiții.

Exercițiu (completați `prelude.mhs`)

Adăugați definiții din Laboratorul 5 în `prelude.mhs`, precum și alte definiții dacă doriți, pentru a ajunge la funcționalitatea din Laboratorul 1.

Mai departe

În laboratoarele ce urmează ne vom concentra mai mult pe sistemul de tipuri, dar, dacă vreți să mai experimentați cu dezvoltarea interpretorului, iată unele idei:

Puteti adăuga mai multe facilități de Sugar/Desugar

Dacă denumiți argumentele corespunzătoare constructorilor din codările Church cu nume distinctive, puteți să încercați să le detectați în faza de Sugar și să încercați să restabiliți valori tradiționale (constante bool, constructori `Maybe`, liste, numere naturale).

Recursie

Deoarece folosim un mediu de evaluare, putem deja avea recursivitate nativ în limbaj, deoarece definițiile pot să se refere la ele însele sau la alte funcții care sunt definite după ele.

Dacă vrem să excludem / controlăm aceste comportamente vom putea face acest lucru prin sistemul de tipuri.

Importarea altor fișiere cu definiții

Puteți modifica notiunea de program astfel încât acesta să conțină nu doar definiții, ci și anumite directive, precum `import`.

De exemply, am putea spune că operația de `import` citește fișierul importat și adaugă definițiile obținute din această operație la definițiile aferente fișierului curent. Astfel putem obține un sistem rudimentar (dar posibil util) de module.