Universidad Nacional de Rosario
Facultad de Ciencias Exactas,
Ingeniería y Agrimensura

Lic. en Cs. de la Computación

Análisis de Lenguajes de Programación

Trabajo Práctico 3

Tomás Fernández De Luco F-3443/6
Ignacio Sebastián Moline M-6466/1

5 de noviembre de 2018

Ejercicio 1

Definiremos los siguientes entornos:
Γ_1 = x : B→B→B
Γ_2 = x : B→B→B, y : B→B
Γ_3 = x : B→B→B, y : B→B, z : B

Primero haremos las siguientes derivaciones de tipo: $\Gamma_3 \vdash$ (x z) : B → B y $\Gamma_3 \vdash$ (y z) : B.

$$\dfrac{\dfrac{}{\Gamma_3 \vdash x : B \to B \to B}\ \text{T-Var} \qquad \dfrac{}{\Gamma_3 \vdash z : B}\ \text{T-Var}}{\Gamma_3 \vdash (x\ z) : B \to B}\ \text{T-App}$$

$$\dfrac{\dfrac{}{\Gamma_3 \vdash y : B \to B}\ \text{T-Var} \qquad \dfrac{}{\Gamma_3 \vdash z : B}\ \text{T-Var}}{\Gamma_3 \vdash (y\ z) : B}\ \text{T-App}$$

Procederemos dar una derivación de tipo para el término S, teniendo los resultados de sus hojas demostradas previamente:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_3 \vdash (x\ z) : B \to B \qquad \Gamma_3 \vdash (y\ z) : B}{\Gamma_3 \vdash (x\ z)\ (y\ z) : B}\ \text{T-App}}{\Gamma_2 \vdash \lambda z : B.(x\ z)\ (y\ z) : B \to B}\ \text{T-Abs}}{\Gamma_1 \vdash \lambda y : B \to B.\lambda z : B.(x\ z)\ (y\ z) : (B \to B) \to B \to B}\ \text{T-Abs}}{\vdash \lambda x : B \to B \to B.\lambda y : B \to B.\lambda z : B.(x\ z)\ (y\ z) : (B \to B \to B) \to (B \to B) \to B \to B}\ \text{T-Abs}$$

Ejercicio 2

La función *infer* retorna un tipo *Either* ya que la inferencia de tipo puede fallar. Si hubo algún error, se propaga en la recursión y se termina devolviendo *Left MensajeDeError*. Si el tipo evalúa de manera correcta, se devuelve *Right Tipo*.

El operador (>>=) toma un argumento de tipo *Either String Type* y una función de tipo *(Type → Either String Type)*. Si el valor del primer argumento es un *Left x*, el operador funciona como la identidad. Si es un *Right v*, devuelve el resultado de aplicar la función argumento a *v*.

Ejercicio 3

Los cambios realizados en código son los siguientes:
Common.hs: Agregamos LLet a LamTerm y Let a Term.

```
data LamTerm = ...
| LLet String LamTerm LamTerm
```

```
data Term = ...
| Let Term Term
```

Parse.y: Agregamos los tokens TLet y TIn al listado.

```
LET     { TLet }
IN      { TIn }
```

Los pusimos al mismo nivel de la abstracción en el listado de precedencias.

```
%right VAR
%left '='
%right '->'
%right '\\' '.' LET IN
```

Agregamos una regla para Exp en la gramática.

```
Exp     :: { LamTerm }
        : '\\' VAR ':' Type '.' Exp     { Abs $2 $4 $6 }
        | LET VAR '=' Exp IN Exp        { LLet $2 $4 $6 }
```

Agregamos TLet y TIn al tipo de dato de los tokens.

```
data Token = ...
| TLet
| TIn
```

Agregamos los patterns para "*let*" e "*in*" en el lexer.

```
lexer cont s = case s of
...
    ("let", rest) -> cont TLet rest
    ("in", rest) -> cont TIn rest
```

Simplytyped.hs: Agregamos un pattern a conversion'.

```
conversion' b (LLet var lt1 lt2) = Let (conversion' b lt1) (conversion' (var:b) lt2)
```

Agregamos un pattern a sub.

```
sub i t (Let lt1 lt2)          = Let (sub i t lt1) (sub (i+1) t lt2)
```

Agregamos un pattern a eval.

```
eval e (Let lt1 lt2)              = let value = eval e lt1
                                    in eval e (sub 0 (quote value) lt2)
```

Agregamos un pattern a infer'.

```
infer' c e (Let lt1 lt2) = infer' c e lt1 >>= (\tlt1 -> infer' (tlt1 : c) e lt2)
```

Hicimos un cambio en eval (Lam t u :@: v), para evitar tener que agregar casos adicionales cuando se definan nuevos valores.

```
eval e (Lam t u :@: v)          = let value = eval e v
                                  in eval e (sub 0 (quote value) u)
```

Prettyprinter.hs: Agregamos un pattern a fv.

```
fv (Let lt1 lt2)      = fv lt1 ++ fv lt2
```

Agregamos la funcion isLet.

```
isLet :: Term -> Bool
isLet (Let _ _) = True
isLet _         = False
```

Agregamos un pattern a pp.

```
pp ii vs (Let lt1 lt2) = text "let " <>
                         text (vs !! ii) <>
                         text " = " <>
                         parensIf (isLam lt1 || isApp lt1 || isLet lt1 || isAs lt1)
                           (pp ii vs lt1) <>
                         text " in " <>
                         parensIf (isLam lt2 || isApp lt2 || isLet lt2 || isAs lt2)
                           (pp (ii + 1) vs lt2)
```

Ejercicio 4

Los cambios realizados en código son los siguientes:
Common.hs: Agregamos LAs a LamTerm y As a Term.

```
data LamTerm = ...
| LAs LamTerm Type
```

```
data Term = ...
| As Term Type
```

Parse.y: Agregamos los tokens TAs al listado.

```
AS      { TAs }
```

Lo pusimos en un nivel mayor que la abstracción en el listado de precedencias.

```
%right VAR
%left '='
%right '->'
%right '\\' '.' LET IN
%left AS
```

Agregamos una regla para Exp en la gramática.

```
Exp     :: { LamTerm }
        : '\\' VAR ':' Type '.' Exp     { Abs $2 $4 $6 }
        | LET VAR '=' Exp IN Exp        { LLet $2 $4 $6 }
        | Exp AS Type                   { LAs $1 $3 }
```

Agregamos TAs al tipo de dato de los tokens.

```
data Token = ...
| TAs
```

Agregamos un pattern para "*as*" en el lexer.

```
lexer cont s = case s of
...
    ("as", rest) -> cont TAs rest
```

Simplytyped.hs: Agregamos un pattern a conversion'.

```
conversion' b (LAs lt typ) = As (conversion' b lt) typ
```

Agregamos un pattern a sub.

```
sub i t (As lt typ)          = As (sub i t lt) typ
```

Agregamos un pattern a eval.

```
eval e (As lt typ)              = eval e lt
```

Agregamos un pattern a infer'.

```
infer' c e (As lt typ) = infer' c e lt >>= (\tlt -> if tlt == typ
                        then ret typ
                        else matchError typ tlt)
```

Prettyprinter.hs: Agregamos un pattern a fv.

```
fv (As lt typ)      = fv lt
```

Agregamos la funcion isAs.

```
isAs :: Term -> Bool
isAs (As _ _) = True
isAs _        = False
```

Agregamos un pattern a pp.

```
pp ii vs (As lt typ) = parensIf (isApp lt || isLam lt || isLet lt || isAs lt)
                       (pp ii vs lt) <>
                       text " as " <>
                       printType typ
```

Ejercicio 5

Dado el término: (let z = ((x : B: x) as B → B) in z) as B → B , su árbol de derivación de tipo es:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : B \vdash x : B}\ \text{T-Var}}{\vdash \lambda x : B.\ x : B \to B}\ \text{T-Abs}}{\vdash (\lambda x : B.\ \text{x})\ as\ B \to B : B \to B}\ \text{T-Ascribe} \qquad \dfrac{}{z : B \to B \vdash z : B \to B}\ \text{T-Var}}{\vdash let\ z = ((\lambda x : B.x)\ as\ B \to B)\ in\ z : B \to B}\ \text{T-Let}}{\vdash (let\ z = ((\lambda x : B.x)\ as\ B \to B)\ in\ z)\ as\ B \to B : B \to B}\ \text{T-Ascribe}$$

Ejercicio 6

Los cambios realizados en código son los siguientes:
Common.hs: Agregamos Unit a Type, LUnit a LamTerm, UnitT a Term y VUnit a Value.

```
data Type = ...
| Unit
```

```
data LamTerm = ...
| LUnit
```

```
data Term = ...
| UnitT
```

```
data Value = ...
| VUnit
```

Parse.y: Agregamos los tokens TVUnit y TUnit al listado.

```
VUNIT   { TVUnit }
UNIT    { TUnit }
```

Agregamos VUNIT a los Atoms y UNIT a los tipos, dentro de la gramática.

```
Atom    :: { LamTerm }
        : VAR                       { LVar $1 }
        | VUNIT                     { LUnit }
```

```
Type    : TYPE                      { Base }
        | Type '->' Type            { Fun $1 $3 }
        | UNIT                      { Unit }
```

Agregamos TVUnit y TUnit al tipo de dato de los tokens.

```
data Token = ...
| TVUnit
| TUnit
```

Agregamos los patterns para "*unit*" y "*Unit*" en el lexer.

```
lexer cont s = case s of
...
("unit", rest) -> cont TVUnit rest
("Unit", rest) -> cont TUnit rest
```

Simplytyped.hs: Agregamos un pattern a conversion'.

```
conversion' b  LUnit = UnitT
```

Agregamos un pattern a sub.

```
sub i t  UnitT                = UnitT
```

Agregamos un pattern a eval.

```
eval e  UnitT                    = VUnit
```

Agregamos un pattern a infer'.

```
infer' c e  UnitT = ret Unit
```

Agregamos un pattern a quote.

```
quote  VUnit          = UnitT
```

Prettyprinter.hs: Agregamos un pattern a fv.

```
fv  UnitT             = []
```

Agregamos un pattern a printType

```
printType  Unit          = text "Unit"
```

Agregamos un pattern a pp.

```
pp ii vs  UnitT = text "unit"
```

Ejercicio 7

La relación de evaluación extendida es:

$$\frac{t1 \to t1'}{(t1, t2) \to (t1', t2)}\ \text{E-Tuple1}$$

$$\frac{t2 \to t2'}{(v, t2) \to (v, t2')}\ \text{E-Tuple2}$$

$$\frac{}{fst(v1, v2) \to v1}\ \text{E-Fst}$$

$$\frac{}{snd(v1, v2) \to v2}\ \text{E-Snd}$$

Ejercicio 8

Los cambios realizados en código son los siguientes:
Common.hs: Agregamos Tuple a Type; LTuple, LFirst y LSecond a LamTerm; TupleT, First y Second a Term, VTuple a Value.

```
data Type = ...
| Tuple Type Type
```

```
data LamTerm = ...
|  LTuple LamTerm LamTerm
|  LFirst LamTerm
|  LSecond LamTerm
```

```
data Term = ...
| TupleT Term Term
| First Term
| Second Term
```

```
data Value = ...
| VTuple Value Value
```

Parse.y: Agregamos los tokens TFirst y TSecond al listado.

```
  FST     { TFirst }
  SND     { TSecond }
```

Los pusimos en un nivel mayor a AS en el listado de precedencias.

```
%right VAR
%left '='
%right '->'
%right '\\' '.' LET IN
%left AS
%right REC
%right SUC
%right FST SND
```

Agregamos reglas para FST y SND en la gramática para las expresiones, y una regla para construcción de tuplas.

```
Exp     :: { LamTerm }
        : '\\' VAR ':' Type '.' Exp     { Abs $2 $4 $6 }
        | LET VAR '=' Exp IN Exp        { LLet $2 $4 $6 }
        | Exp AS Type                   { LAs $1 $3 }
        | FST Exp                       { LFirst $2 }
        | SND Exp                       { LSecond $2 }
        | '(' Exp ',' Exp ')'           { LTuple $2 $4 }
```

```
Type    : ...
| '(' Type ',' Type ')'        { Tuple $2 $4 }
```

Agregamos TFirst y TSecond al tipo de dato de los tokens.

```
data Token = ...
| TFirst
| TSecond
```

Agregamos los patterns para "*fst*" y "*snd*" en el lexer.

```
lexer cont s = case s of
...
("fst", rest) -> cont TFirst rest
("snd", rest) -> cont TSecond rest
```

Simplytyped.hs: Agregamos patterns a conversion'.

```
conversion' b (LTuple lt1 lt2) = TupleT (conversion' b lt1) (conversion' b lt2)
conversion' b (LFirst lt) = First (conversion' b lt)
conversion' b (LSecond lt) = Second (conversion' b lt)
```

Agregamos patterns a sub.

```
sub i t (TupleT lt1 lt2)      = TupleT (sub i t lt1) (sub i t lt2)
sub i t (First lt)            = First (sub i t lt)
sub i t (Second lt)           = Second (sub i t lt)
```

Agregamos patterns a eval.

```
eval e (TupleT lt1 lt2)          = VTuple (eval e lt1) (eval e lt2)
eval e (First lt)                = case eval e lt of
                                       (VTuple v1 v2) -> v1
                                       _ -> error "Error de tipo de run-time,
                                            verificar type checker"
eval e (Second lt)               = case eval e lt of
                                       (VTuple v1 v2) -> v2
```

Agregamos patterns a infer'.

```
infer' c e (TupleT lt1 lt2) = infer' c e lt1 >>=
                              (\tlt1 -> infer' c e lt2 >>=
                                (\tlt2 -> ret (Tuple tlt1 tlt2)))
infer' c e (First lt) = infer' c e lt >>=
                        (\tlt -> case tlt of
                                     Tuple t1 t2 -> ret t1
                                     t -> tupleError t)
infer' c e (Second lt) = infer' c e lt >>=
                         (\tlt -> case tlt of
                                      Tuple t1 t2 -> ret t2
                                      t -> tupleError t)
```

Agregamos un pattern a quote.

```
quote (VTuple v1 v2)  = TupleT (quote v1) (quote v2)
```

Agregamos la función tuppleError.

```
tupleError :: Type -> Either String Type
tupleError t = err ("Funci n de tupla aplicado a un tipo " ++
                    render (printType t) ++ ".")
```

Prettyprinter.hs: Agregamos patterns a fv.

```
fv (TupleT lt1 lt2)  = fv lt1 ++ fv lt2
fv (First lt)        = fv lt
fv (Second lt)       = fv lt
```

Agregamos un pattern a printType.

```
printType (Tuple t1 t2) = text "(" <>
                          printType t1 <>
                          text ", " <>
                          printType t2 <>
                          text ")"
```

Agregamos patterns a pp.

```
pp ii vs (TupleT lt1 lt2) = parens ((pp ii vs lt1) <>
                                    text ", " <>
                                    (pp ii vs lt2))
pp ii vs (First lt) = text "fst " <>
                      parensIf (isApp lt || isLam lt || isLet lt || isAs lt || isTupleOp lt)
                          (pp ii vs lt)
pp ii vs (Second lt) = text "snd " <>
                       parensIf (isApp lt || isLam lt || isLet lt || isAs lt || isTupleOp lt)
                          (pp ii vs lt)
```

Definimos isTupleOp.

```
isTupleOp :: Term -> Bool
isTupleOp (First _)  = True
isTupleOp (Second _) = True
isTupleOp _          = False
```

Ejercicio 9

El árbol de derivación de tipo para el término fst (*unit as Unit*, $\lambda x : (B, B)$. *snd* x) es:

$$\dfrac{\dfrac{\dfrac{}{\vdash unit : Unit}\ \text{T-Unit}}{\vdash \text{unit as Unit: } Unit}\ \text{T-Ascribe} \qquad \dfrac{\dfrac{\dfrac{}{x : (B,B) \vdash x : (B,B)}\ \text{T-Var}}{x : (B,B) \vdash \text{snd x: } B}\ \text{T-Snd}}{\vdash \lambda x : (B,B).\ \text{snd x} : (B,B) \to B}\ \text{T-Abs}}{\vdash (\text{unit as Unit}, \lambda x : (B,B).\ snd\ x) : (Unit, (B,B) \to B)}\ \text{T-Pair}$$
$$\dfrac{}{\vdash fst\ (\text{unit as Unit}, \lambda x : (B,B).\ snd\ x) : Unit}\ \text{T-Fst}$$

Ejercicio 10

Los cambios realizados en código son los siguientes:
Common.hs: Agregamos Nat a Type; LZero, LSucc y LR a LamTerm; ZeroT, SuccT y RT a Term, VNat a Value. Agregamos el tipo de datos Natural.

```
data Type = ...
| Nat
```

```
data LamTerm = ...
|  LZero
|  LSucc LamTerm
|  LR LamTerm LamTerm LamTerm
```

```
data Term = ...
| ZeroT
| SuccT Term
| RT Term Term Term
```

```
data Value = ...
| VNat Natural
```

```
data Natural = Zero
| Succ Natural
```

Parse.y: Agregamos los tokens TSuc, TRec, TZero y TNat al listado.

```
  SUC     { TSuc }
  REC     { TRec }
  ZERO    { TZero }
  NAT     { TNat }
```

Los pusimos en orden correspondiente en un nivel mayor a AS y menor de FST y SND en el listado de precedencias.

```
%right VAR
%left '='
%right '->'
%right '\\' '.' LET IN
%left AS
%right REC
%right SUC
%right FST SND
```

Agregamos reglas para SUC y REC en la gramática para las expresiones, ZERO a los átomos y NAT a los tipos.

```
Exp     :: { LamTerm }
...
| SUC Exp                          { LSucc $2 }
| REC Atom Atom Exp                { LR $2 $3 $4 }
```

```
Atom    :: { LamTerm }
...
| ZERO                             { LZero }
```

```
Type    : TYPE                              { Base }
...
| NAT                              { Nat }
```

Agregamos TSuc, TRec, TZero y TNat al tipo de dato de los tokens.

```
data Token = ...
| TSuc
| TRec
| TZero
| TNat
```

Agregamos los patterns para $'0'$, "$succ$", "R" y "Nat" en el lexer.

```
lexer cont s = case s of
...
('0':cs) -> cont TZero cs
...
("succ", rest) -> cont TSuc rest
("R", rest) -> cont TRec rest
("Nat", rest) -> cont TNat rest
```

Simplytyped.hs: Agregamos patterns a conversion'.

```
conversion' b  LZero = ZeroT
conversion' b (LSucc lt) = SuccT (conversion' b lt)
conversion' b (LR lt1 lt2 lt3) = RT (conversion' b lt1) (conversion' b lt2)
                    (conversion' b lt3)
```

Agregamos patterns a sub.

```
sub i t  ZeroT                = ZeroT
sub i t (SuccT lt)            = SuccT (sub i t lt)
sub i t (RT lt1 lt2 lt3)      = RT (sub i t lt1) (sub i t lt2) (sub i t lt3)
```

Agregamos patterns a eval.

```
eval e  ZeroT                    = VNat Zero
eval e (SuccT lt)                = case eval e lt of
VNat n -> VNat (Succ n)
_      -> error "Error de tipo de run-time, verificar type checker"
eval e (RT lt1 lt2 lt3)          = case eval e lt3 of
VNat Zero     -> eval e lt1
VNat (Succ n) -> eval e ((lt2 :@: (quote (eval e (RT lt1 lt2 (quote (VNat n))))))
                         :@: (quote (VNat n)))
```

Agregamos patterns a infer'.

```
infer' c e  ZeroT = ret Nat
infer' c e (SuccT lt) = infer' c e lt >>=
  (\tlt -> case tlt of
    Nat -> ret Nat
    t -> succError t)
infer' c e (RT lt1 lt2 lt3) = infer' c e lt1 >>=
      (\tlt1 -> infer' c e lt2 >>=
        (\tlt2 -> case tlt2 of
          Fun t1A (Fun Nat t1B) -> if t1A == tlt1 && t1B == tlt1 then infer' c e lt3 >>=
            (\tlt3 -> case tlt3 of
              Nat -> ret tlt1
              t3  -> rError2 t3)
            else rError1 tlt1 (Fun t1A (Fun Nat t1B))
          t2 -> rError1 tlt1 t2))
```

Agregamos patterns a quote.

```
quote (VNat Zero)     = ZeroT
quote (VNat (Succ n)) = SuccT (quote (VNat n))
```

Agregamos succError, rError1, rError2

```
succError :: Type -> Either String Type
succError t = err ("Sucesor aplicado a un tipo " ++ render (printType t) ++ ".")

rError1 :: Type -> Type -> Either String Type
rError1 t1 t2 = err ("Error de tipado en operador R. Tipo del primer argumento: " ++
  render (printType t1) ++ ". Tipo del segundo argumento: " ++
  render (printType t2) ++ ".")

rError2 :: Type -> Either String Type
rError2 t = err ("Operador R con tercer argumento de tipo " ++
  render (printType t) ++ " cuando deberia ser Nat.")
```

Prettyprinter.hs: Agregamos patterns a fv.

```
fv  ZeroT           = []
fv (SuccT lt)       = fv lt
fv (RT lt1 lt2 lt3) = fv lt1 ++ fv lt2 ++ fv lt3
```

Agregamos un pattern a printType

```
printType  Nat          = text "Nat"
```

Agregamos patterns a pp.

```
pp ii vs  ZeroT = text "0"
pp ii vs (SuccT lt) =
  text "succ " <>
  parensIf (isLam lt || isApp lt || isLet lt || isAs lt || isTupleOp lt || isNatOp lt)
    (pp ii vs lt)
pp ii vs (RT lt1 lt2 lt3) =
  text "R " <>
  sep [parensIf (isLam lt1 || isApp lt1 || isLet lt1 || isAs lt1)
    (pp ii vs lt1),
  nest 1 (parensIf (isLam lt2 || isApp lt2 || isLet lt2 || isAs lt2)
    (pp ii vs lt2)),
  nest 1 (parensIf (isLam lt3 || isApp lt3 || isLet lt3 || isAs lt3)
```

Definimos isNatOp

```
isNatOp :: Term -> Bool
isNatOp (SuccT _)   = True
isNatOp (RT _ _ _) = True
isNatOp _           = False
```

Ejercicio 11

La función de Ackermann es:
Ack: N $\rightarrow$ N $\rightarrow$ N
Ack 0 n = n + 1
Ack m 0 = Ack (m 1) 1
Ack m n = Ack (m 1) (Ack m (n 1))

A grandes rasgos, los casos dependen de si el primer argumento es nulo. Si lo es, existe un caso base. Si no, hay dos posibilidades para llamadas recursivas en función de si el segundo argumento es nulo. Entonces, creemos que el patrón de Ack se puede captar mediante dos funciones que se definan a partir del operador R. De esta forma, se plantean las siguientes definiciones:

AckAux: (N $\rightarrow$ N) $\rightarrow$ N $\rightarrow$ N
AckAux f 0 = f 1
AckAux f n = f (AckAux f (n - 1))

Ack: N $\rightarrow$ N $\rightarrow$ N
Ack 0 = succ
Ack m = AckAux (Ack (m - 1))

La función Ack definida de esta forma se encarga de definir qué comportamiento tener con respecto al segundo argumento en función de si el primero es nulo o no. Si no lo es, se utiliza la función AckAux para que defina cuál será el segundo argumento de la llamada recursiva. Teniendo las definiciones, ahora se puede expresar a ambas funciones con el operador R:

def AckAux = λa : Nat$\rightarrow$ $Nat.$ λn : Nat.R (a (succ 0)) (λx : Nat.λy : Nat.a x) n

$defAck = \lambda m : Nat.R\ (\lambda n : Nat.\text{succ n})\ (\lambda f : Nat \rightarrow \text{Nat.}\ \lambda y : Nat.AckAux\ f)\ m$

Para ambas se tuvo que definir una función de recursión que tome dos valores e ignore el segundo, ya que el valor actual del argumento no es necesario en el cálculo del resultado, sólo lo que devuelva la llamada recursiva.