Taller de Programación
Algoritmo de Inferencia de Tipos

Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

13 de septiembre de 2016

Algoritmo de Inferencia

Entrada	una expresión U de λ sin anotaciones
Salida	el juicio de tipado $\Gamma \rhd M : \sigma$ más general para U, o bien una falla si U no es tipable
Estrategia	recursión sobre la estructura de U

Tipos de datos y funciones auxiliares

Expresiones de tipo:

$$\sigma \quad ::= \quad s \mid \mathit{Nat} \mid \mathit{Bool} \mid \sigma \to \tau$$

Tipos de datos y funciones auxiliares

Expresiones de tipo:

$$\sigma \quad ::= \quad s \mid \mathit{Nat} \mid \mathit{Bool} \mid \sigma \to \tau$$

En Haskell:

```
data Type = TVar Int
          | TNat
          | TBool
          | TFun Type Type
```

Tipos de datos y funciones auxiliares

Expresiones anotadas:

$$
\begin{array}{rcl}
M & ::= & x \\
& | & 0 \mid succ(M) \mid pred(M) \mid iszero(M) \\
& | & true \mid false \mid if\ M\ then\ P\ else\ Q \\
& | & \lambda x : \sigma . M \\
& | & M\, N
\end{array}
$$

Tipos de datos y funciones auxiliares

Expresiones sin anotar:

$$
\begin{array}{rcl}
M & ::= & x \\
 & \mid & \mathit{0} \mid \mathit{succ}(M) \mid \mathit{pred}(M) \mid \mathit{iszero}(M) \\
 & \mid & \mathit{true} \mid \mathit{false} \mid \mathit{if}\ \ M\ \ \mathit{then}\ \ P\ \ \mathit{else}\ \ Q \\
 & \mid & \lambda x.M \\
 & \mid & M\,N
\end{array}
$$

Tipos de datos y funciones auxiliares

```
type Symbol = String

data Exp a = VarExp Symbol
           | ZeroExp
           | SuccExp (Exp a)
           | PredExp (Exp a)
           | IsZeroExp (Exp a)
           | TrueExp
           | FalseExp
           | IfExp (Exp a) (Exp a) (Exp a)
           | LamExp Symbol a (Exp a)
           | AppExp (Exp a) (Exp a)
```

Tipos de datos y funciones auxiliares

```
type Symbol = String

data Exp a = VarExp Symbol
           | ZeroExp
           | SuccExp (Exp a)
           | PredExp (Exp a)
           | IsZeroExp (Exp a)
           | TrueExp
           | FalseExp
           | IfExp (Exp a) (Exp a) (Exp a)
           | LamExp Symbol a (Exp a)
           | AppExp (Exp a) (Exp a)

type AnnotExp = Exp Type
```

Tipos de datos y funciones auxiliares

```
type Symbol = String

data Exp a = VarExp Symbol
           | ZeroExp
           | SuccExp (Exp a)
           | PredExp (Exp a)
           | IsZeroExp (Exp a)
           | TrueExp
           | FalseExp
           | IfExp (Exp a) (Exp a) (Exp a)
           | LamExp Symbol a (Exp a)
           | AppExp (Exp a) (Exp a)

type AnnotExp = Exp Type

type PlainExp = Exp ()
```

Tipos de datos y funciones auxiliares

Contexto

```
emptyEnv :: Env
extendE  :: Env -> Symbol -> Type -> Env
removeE  :: Env -> Symbol -> Env
evalE    :: Env -> Symbol -> Type
joinE    :: [Env] -> Env
domainE  :: Env -> [Symbol]
```

Sustituciones

```
emptySubst :: Subst
extendS :: Int -> Type -> Subst -> Subst
```

Sustituciones

```
emptySubst :: Subst
extendS :: Int -> Type -> Subst -> Subst

class Substitutable a where
    (<.>) :: Subst -> a -> a
    instance Substitutable Type   -- subst <.> t
    instance Substitutable Env    -- subst <.> env
    instance Substitutable Exp    -- subst <.> e
```

Sustituciones

```
emptySubst :: Subst
extendS :: Int -> Type -> Subst -> Subst

class Substitutable a where
    (<.>) :: Subst -> a -> a
    instance Substitutable Type   -- subst <.> t
    instance Substitutable Env    -- subst <.> env
    instance Substitutable Exp    -- subst <.> e
```

Unificación

```
type UnifGoal = (Type, Type)
data UnifResult = UOK Subst | UError Type Type
mgu :: [UnifGoal] -> UnifResult
```

La función de inferencia

```
type TypingJudgment = (Env, AnnotExp, Type)
data Result a = OK a | Error String

inferType :: PlainExp → Result TypingJudgment
```

La función de inferencia

```
type TypingJudgment = (Env, AnnotExp, Type)
data Result a = OK a | Error String

inferType :: PlainExp → Result TypingJudgment
inferType (VarExp x) = ...
    ...
```

La función de inferencia

```
type TypingJudgment = (Env, AnnotExp, Type)
data Result a = OK a | Error String

inferType :: PlainExp → Result TypingJudgment
inferType (VarExp x) = ...
    ...
```

$$\mathbb{W}(x) \stackrel{\text{def}}{=} \{x : s\} \triangleright x : s, \quad s \text{ variable fresca}$$

La función de inferencia

```
type TypingJudgment = (Env, AnnotExp, Type)
data Result a = OK a | Error String

inferType :: PlainExp → Result TypingJudgment
infer' :: PlainExp
               → Int
               → Result (Int, TypingJudgment)
```

¡A programar!

Consigna

- Completar archivo `TypeInference.hs`
- Definir la función `inferType` utilizando `infer'`
- Definir la función `infer'` para los casos

  VarExp ZeroExp LamExp AppExp
- Usar *pattern matching* sobre Exp

¡A programar!

Consigna

- Completar archivo `TypeInference.hs`
- Definir la función `inferType` utilizando `infer'`
- Definir la función `infer'` para los casos

  VarExp ZeroExp LamExp AppExp
- Usar *pattern matching* sobre Exp

Tip

```
let x = expr1 in expr2
case expr of  Pattern1 -> res1
              ...
              Paternn -> resn
```

¡A programar!

Consigna

- Completar archivo `TypeInference.hs`
- Definir la función `inferType` utilizando `infer'`
- Definir la función `infer'` para los casos

  VarExp ZeroExp LamExp AppExp
- Usar *pattern matching* sobre Exp

Prueba

- Archivo `Examples.hs`
- `expr` n `:: String`
- `inferExpr :: String` $\rightarrow$ `Doc`

Ejemplo

```
infer'(SuccExp e) n =
```

Ejemplo

```
infer'(SuccExp e) n =

case infer' e n of
```

Ejemplo

```
infer'(SuccExp e) n =
case infer' e n of
    OK ( n', (env', e', t') ) ->
```

Ejemplo

```
infer'(SuccExp e) n =
case infer' e n of
    OK ( n', (env', e', t') ) ->
```

Ejemplo

```
infer'(SuccExp e) n =
case infer' e n of
    OK ( n', (env', e', t') ) ->
        case mgu [ (t', TNat) ] of
```

Ejemplo

```
infer'(SuccExp e) n =
case infer' e n of
    OK ( n', (env', e', t') ) ->
        case mgu [ (t', TNat) ] of
            UOK subst ->





            UError u1 u2 ->
```

Ejemplo

```
infer'(SuccExp e) n =
case infer' e n of
    OK ( n', (env', e', t') ) ->
        case mgu [ (t', TNat) ] of
            UOK subst ->









            UError u1 u2 ->
                uError u1
```

Ejemplo

```
infer'(SuccExp e) n =
case infer' e n of
    OK ( n', (env', e', t') ) ->
        case mgu [ (t', TNat) ] of
            UOK subst ->
                OK ( n', (



                         ) )
            UError u1 u2 ->
                uError u1
```

Ejemplo

```
infer'(SuccExp e) n =

case infer' e n of

    OK ( n', (env', e', t') ) ->

        case mgu [ (t', TNat) ] of

            UOK subst ->

                OK ( n', (

                          subst <.> env',
                          subst <.> SuccExp e',
                          TNat

                         ) )

            UError u1 u2 ->

                uError u1
```