

Taller de Programación

Algoritmo de Inferencia de Tipos

Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Septiembre 2017

Algoritmo de Inferencia

Entrada una expresión U de λ **sin anotaciones**

Salida el **juicio de tipado** $\Gamma \triangleright M : \sigma$ 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:

$$\sigma ::= s \mid \textit{Nat} \mid \textit{Bool} \mid \sigma \rightarrow \tau$$

Tipos de datos y funciones auxiliares

Expresiones de tipo:

$$\sigma ::= s \mid Nat \mid Bool \mid \sigma \rightarrow \tau$$

En Haskell:

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

Tipos de datos y funciones auxiliares

Expresiones anotadas:

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

Tipos de datos y funciones auxiliares

Expresiones sin anotar:

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

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

Tipos de datos y funciones auxiliares

Sustituciones y unificación

Sustituciones

```
emptySubst :: Subst
```

```
extendsS :: Int -> Type -> Subst -> Subst
```

Tipos de datos y funciones auxiliares

Sustituciones y unificación

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
```

Tipos de datos y funciones auxiliares

Sustituciones y unificación

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
             ...
             Patternn -> 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 → 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') ) ->
```

```
res@(Error _) ->
```

Ejemplo

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

```
res@(Error _) -> res
```


Ejemplo

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

```
res@(Error _) -> res
```

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 ->  
  
    res@(Error _) -> res
```

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 u2  
    res@(Error _) -> res
```

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 u2  
  res@(Error _) -> res
```

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 u2  
res@(Error _) -> res
```