Taller de Programación
Algoritmo de Inferencia de Tipos

Paradigmas de Lenguajes de Programación

2c2023

(1) Algoritmo de Inferencia $\mathbb{W}$

Resumen

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.

(1) Algoritmo de Inferencia $\mathbb{W}$

Resumen

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.

Ejemplos

- $\mathbb{W}(\text{True}) = \emptyset \rhd \text{True} : \text{Bool}$
- $\mathbb{W}(\text{Succ(x)}) = \{x : \text{Nat}\} \rhd \text{Succ}(x) : \text{Nat}$
- $\mathbb{W}(\text{x}) = \{x : t_0\} \rhd x : t_0$
- $\mathbb{W}(\lambda\ \text{x:Nat. } 0) = \text{ERROR}$
- $\mathbb{W}(\lambda\ \text{x.0}) = \emptyset \rhd (\lambda x : t_0.0) : t_0 \to \text{Nat}$

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

(2) Tipos de datos y funciones auxiliares

Tipos

Expresiones de tipo

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

(2) Tipos de datos y funciones auxiliares

Tipos

Expresiones de tipo

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

En Haskell

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

(3) Tipos de datos y funciones auxiliares

Expresiones

Expresiones anotadas

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

(4) Tipos de datos y funciones auxiliares

Expresiones

Expresiones sin anotar

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

(5) Tipos de datos y funciones auxiliares

Expresiones

En Haskell

```
type Symbol = String

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

(5) Tipos de datos y funciones auxiliares

Expresiones

En Haskell

```
type Symbol = String

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

           | LamExp ???
```

(6) Tipos de datos y funciones auxiliares

Expresiones

En Haskell (funciones lambda)

El constructor `LamExp` toma como segundo argumento el tipo.

$$\lambda x : \sigma . M$$

```
LamExp Symbol a (Exp a)
```

(6) Tipos de datos y funciones auxiliares

Expresiones

En Haskell (funciones lambda)

El constructor `LamExp` toma como segundo argumento el tipo.

$$\lambda x : \sigma . M$$

```
LamExp Symbol a (Exp a)
```

Anotadas vs no anotadas

Para diferenciar entre anotadas y no anotadas, a la segunda le pasamos `()` (Unit) como tipo

```
type AnnotExp = Exp Type
type PlainExp = Exp ()
```

(7) Tipos de datos y funciones auxiliares

Contextos

Contextos

Representamos al contexto con un tipo abstracto de datos.
Lo vamos construyendo a medida que vamos ejecutando el algoritmo (es una de las 3 partes del juicio de tipado).

En Haskell

Podemos realizar las siguientes operaciones:

```
emptyContext :: Context
extendC  :: Context -> Symbol -> Type -> Context
removeC  :: Context -> Symbol -> Context
evalC    :: Context -> Symbol -> Type
joinC    :: [Context] -> Context
domainC  :: Context -> [Symbol]
```

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

(8) Tipos de datos y funciones auxiliares

Sustitiuciones

Sustituciones

Las representamos con un tipo abstracto de datos. La sustitución vacía (o identidad) puede extenderse para formar nuevas sustituciones.
Para extender una sustitución, indicamos qué índice de variable se va a reemplazar por qué tipo.

(8) Tipos de datos y funciones auxiliares

Sustitiuciones

Sustituciones

Las representamos con un tipo abstracto de datos. La sustitución vacía (o identidad) puede extenderse para formar nuevas sustituciones.
Para extender una sustitución, indicamos qué índice de variable se va a reemplazar por qué tipo.

En Haskell

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

(9) Tipos de datos y funciones auxiliares

Sustitiuciones

Aplicando sustituciones

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

(10) Tipos de datos y funciones auxiliares

Unificación

Objetivos a unificar

Representamos los objetivos a unificar como una lista de pares de tipos. La función `mgu` toma esa lista y devuelve, o bien la sustitución resultante, o bien un error que indica qué par de tipos no se pudo unificar.

(10) Tipos de datos y funciones auxiliares

Unificación

Objetivos a unificar

Representamos los objetivos a unificar como una lista de pares de tipos. La función `mgu` toma esa lista y devuelve, o bien la sustitución resultante, o bien un error que indica qué par de tipos no se pudo unificar.

En Haskell

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

(11) La función de inferencia

En Haskell

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

inferType :: PlainExp → Result TypingJudgment
```

(11) La función de inferencia

En Haskell

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

inferType :: PlainExp → Result TypingJudgment
inferType e = ...
      ...
```

(11) La función de inferencia

En Haskell

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

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

(11) La función de inferencia

En Haskell

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

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

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

(11) La función de inferencia

En Haskell

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

inferType :: PlainExp → Result TypingJudgment
inferType e = case infer' e 0 of
     OK (_, tj) → OK tj
     Error s → Error s

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

(12) ¡A programar!

Consigna

- Completar archivo `TypeInference.hs`
- Definir la función `inferType` utilizando `infer'`
- Definir la función `infer'` para los casos
  ZeroExp VarExp AppExp LamExp
- Usar *pattern matching* sobre Exp

(12) ¡A programar!

Consigna

- Completar archivo `TypeInference.hs`
- Definir la función `inferType` utilizando `infer'`
- Definir la función `infer'` para los casos
  ZeroExp VarExp AppExp LamExp
- Usar *pattern matching* sobre Exp

Pista

```
let x = expr1 in expr2
case expr of  Patrón_1 -> res_1
              ...
              Patrón_n -> res_n
```

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

(13) Probando el código

- Cargar el archivo `Main.hs`
  - `inferExpr :: String → Doc`
    - Toma una cadena de texto, la convierte a algo de tipo `Exp` y se lo pasa a `inferType`
- `expr` *n* `:: String` (en el archivo `Examples.hs`)
  - Toma un número y devuelve una cadena de texto para pasarle a `inferExpr`

- `Main> inferExpr $ expr 1`
- `Main> inferExpr "succ(x)"`

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Ejemplo

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

infer' :: PlainExp -> Int -> Result (Int, TypingJudgment)
infer' (SuccExp e) n =
```

Ejemplo

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

infer' :: PlainExp -> Int -> Result (Int, TypingJudgment)
infer' (SuccExp e) n =
  case infer' e n of
    err@(Error _) ->
```

Ejemplo

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

infer' :: PlainExp -> Int -> Result (Int, TypingJudgment)
infer' (SuccExp e) n =
  case infer' e n of
    err@(Error _) -> err
    OK ( n', (c', e', t') ) ->
```

Ejemplo

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

infer' :: PlainExp -> Int -> Result (Int, TypingJudgment)
infer' (SuccExp e) n =
  case infer' e n of
    err@(Error _) -> err
    OK ( n', (c', e', t') ) ->
        case mgu [  ] of
```

Ejemplo

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

infer' :: PlainExp -> Int -> Result (Int, TypingJudgment)
infer' (SuccExp e) n =
  case infer' e n of
    err@(Error _) -> err
    OK ( n', (c', e', t') ) ->
        case mgu [ (t', TNat) ] of
```

Ejemplo

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

infer' :: PlainExp -> Int -> Result (Int, TypingJudgment)
infer' (SuccExp e) n =
  case infer' e n of
    err@(Error _) -> err
    OK ( n', (c', e', t') ) ->
        case mgu [ (t', TNat) ] of
            UError u1 u2 ->
            UOK subst ->
```

Ejemplo

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

infer' :: PlainExp -> Int -> Result (Int, TypingJudgment)
infer' (SuccExp e) n =
  case infer' e n of
    err@(Error _) -> err
    OK ( n', (c', e', t') ) ->
        case mgu [ (t', TNat) ] of
            UError u1 u2 -> uError u1 u2
            UOK subst ->
                OK ( n', (

                          ))
```

Ejemplo

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

infer' :: PlainExp -> Int -> Result (Int, TypingJudgment)
infer' (SuccExp e) n =
  case infer' e n of
    err@(Error _) -> err
    OK ( n', (c', e', t') ) ->
        case mgu [ (t', TNat) ] of
            UError u1 u2 -> uError u1 u2
            UOK subst ->
                OK ( n', (c',

                          ))
```

Ejemplo

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

infer' :: PlainExp -> Int -> Result (Int, TypingJudgment)
infer' (SuccExp e) n =
  case infer' e n of
    err@(Error _) -> err
    OK ( n', (c', e', t') ) ->
        case mgu [ (t', TNat) ] of
            UError u1 u2 -> uError u1 u2
            UOK subst ->
                OK ( n', (c',
                          SuccExp e',
                          ))
```

Ejemplo

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

infer' :: PlainExp -> Int -> Result (Int, TypingJudgment)
infer' (SuccExp e) n =
  case infer' e n of
    err@(Error _) -> err
    OK ( n', (c', e', t') ) ->
        case mgu [ (t', TNat) ] of
            UError u1 u2 -> uError u1 u2
            UOK subst ->
                OK ( n', (c',
                          SuccExp e',
                          TNat))
```

Ejemplo

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

infer' :: PlainExp -> Int -> Result (Int, TypingJudgment)
infer' (SuccExp e) n =
  case infer' e n of
    err@(Error _) -> err
    OK ( n', (c', e', t') ) ->
        case mgu [ (t', TNat) ] of
            UError u1 u2 -> uError u1 u2
            UOK subst ->
                OK ( n', (subst <.> c',
                          subst <.> SuccExp e',
                          TNat))
```