Taller de Programación
Algoritmo de Inferencia de Tipos

Paradigmas de Lenguajes de Programación

19 de Mayo de 2020

(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

- $\mathbb{W}(\text{true}) =$

(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

- $\mathbb{W}(\mathsf{true}) = \emptyset \rhd \mathit{true} : \mathit{Bool}$
- $\mathbb{W}(\mathsf{succ(x)}) =$

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

(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

- $\mathbb{W}(\text{true}) = \emptyset \rhd \mathit{true} : \mathit{Bool}$
- $\mathbb{W}(\text{succ(x)}) = \{x : \mathit{Nat}\} \rhd \mathit{succ}(x) : \mathit{Nat}$
- $\mathbb{W}(\text{x}) =$

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

(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

- $\mathbb{W}(\text{true}) = \emptyset \rhd \mathit{true} : \mathit{Bool}$
- $\mathbb{W}(\text{succ(x)}) = \{x : \mathit{Nat}\} \rhd \mathit{succ}(x) : \mathit{Nat}$
- $\mathbb{W}(\text{x}) = \{x : s_0\} \rhd x : s_0$
- $\mathbb{W}(\lambda\ \text{x:Nat. 0}) =$

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

(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

- $\mathbb{W}(\text{true}) = \emptyset \rhd \mathit{true} : \mathit{Bool}$
- $\mathbb{W}(\text{succ(x)}) = \{x : \mathit{Nat}\} \rhd \mathit{succ}(x) : \mathit{Nat}$
- $\mathbb{W}(\text{x}) = \{x : s_0\} \rhd x : s_0$
- $\mathbb{W}(\lambda\ \text{x:Nat. 0}) = \text{ERROR}$
- $\mathbb{W}(\lambda\ \text{x.0}) =$

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

(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

- $\mathbb{W}(\text{true}) = \emptyset \rhd \mathit{true} : \mathit{Bool}$
- $\mathbb{W}(\text{succ(x)}) = \{x : \mathit{Nat}\} \rhd \mathit{succ}(x) : \mathit{Nat}$
- $\mathbb{W}(\text{x}) = \{x : s_0\} \rhd x : s_0$
- $\mathbb{W}(\lambda\ \text{x:Nat. } 0) = \text{ERROR}$
- $\mathbb{W}(\lambda\ \text{x.0}) = \emptyset \rhd (\lambda x : s_0.0) : s_0 \to \mathit{Nat}$

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

(2) Tipos de datos y funciones auxiliares

Tipos

Expresiones de tipo

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

(2) Tipos de datos y funciones auxiliares

Tipos

Expresiones de tipo

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

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 `()` como tipo

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

(7) Tipos de datos y funciones auxiliares

Contexto

Contexto

(7) Tipos de datos y funciones auxiliares

Contexto

Contexto

El contexto de tipado es el conjunto de pares que asigna a cada variable un único tipo.
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]
```

(8) Tipos de datos y funciones auxiliares

Sustitiuciones y unificación

Sustituciones

(8) Tipos de datos y funciones auxiliares

Sustitiuciones y unificación

Sustituciones

Funciones que asignan tipos a variables de tipo.
Útiles para cuando aún no podemos inferir el tipo de una expresión: Le asignamos una variable de tipo temporal y, si más adelante obtenemos más información podemos **sustituirla**.

(8) Tipos de datos y funciones auxiliares

Sustitiuciones y unificación

Sustituciones

Funciones que asignan tipos a variables de tipo.
Útiles para cuando aún no podemos inferir el tipo de una expresión: Le asignamos una variable de tipo temporal y, si más adelante obtenemos más información podemos **sustituirla**.

Unificación

(8) Tipos de datos y funciones auxiliares

Sustitiuciones y unificación

Sustituciones

Funciones que asignan tipos a variables de tipo.
Útiles para cuando aún no podemos inferir el tipo de una expresión: Le asignamos una variable de tipo temporal y, si más adelante obtenemos más información podemos **sustituirla**.

Unificación

Encontrar una sustitución que permita unificar pares de tipos cuando ya sé que ambos deben referirse al mismo tipo.
El Unificador Más General (MGU) me asegura no perder soluciones.

(8) Tipos de datos y funciones auxiliares

Sustitiuciones y unificación

Sustituciones

Funciones que asignan tipos a variables de tipo.
Útiles para cuando aún no podemos inferir el tipo de una expresión: Le asignamos una variable de tipo temporal y, si más adelante obtenemos más información podemos **sustituirla**.

Unificación

Encontrar una sustitución que permita unificar pares de tipos cuando ya sé que ambos deben referirse al mismo tipo.
El Unificador Más General (MGU) me asegura no perder soluciones.

Ejemplo

$\mathbb{W}(\text{x}) = \{x : s_0\} \rhd x : s_0 \qquad \mathbb{W}(\text{y}) = \{y : s_1\} \rhd y : s_1$
$\mathbb{W}(\text{if true then y else x}) = \ldots$

(8) Tipos de datos y funciones auxiliares

Sustitiuciones y unificación

Sustituciones

Funciones que asignan tipos a variables de tipo.
Útiles para cuando aún no podemos inferir el tipo de una expresión: Le asignamos una variable de tipo temporal y, si más adelante obtenemos más información podemos **sustituirla**.

Unificación

Encontrar una sustitución que permita unificar pares de tipos cuando ya sé que ambos deben referirse al mismo tipo.
El Unificador Más General (MGU) me asegura no perder soluciones.

Ejemplo

$\mathbb{W}(\text{x}) = \{x : s_0\} \rhd x : s_0$ $\quad$ $\mathbb{W}(\text{y}) = \{y : s_1\} \rhd y : s_1$

$\mathbb{W}(\text{if true then y else x}) = \ldots$ Tengo que unificar s_0 y s_1

(9) Tipos de datos y funciones auxiliares

Sustitiuciones y unificación

En Haskell (Sustituciones)

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

(9) Tipos de datos y funciones auxiliares

Sustitiuciones y unificación

En Haskell (Sustituciones)

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

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

(9) Tipos de datos y funciones auxiliares

Sustitiuciones y unificación

En Haskell (Sustituciones)

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

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

En Haskell (Unificación)

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

(10) La función de inferencia

En Haskell

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

inferType :: PlainExp → Result TypingJudgment
```

(10) La función de inferencia

En Haskell

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

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

(10) 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) = ...
      ...
```

(10) 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) = ...
     ...
```

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

(10) 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)
```

(11) ¡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

(11) ¡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

Tip

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

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

(12) Probando el código

- Cargar el archivo `Main.hs`
  - `inferExpr :: String` $\rightarrow$ `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

```
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', (c', e', t') ) ->

    res@(Error _) ->
```

Ejemplo

```
infer' (SuccExp e) n =

case infer' e n of

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

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

Ejemplo

```
infer' (SuccExp e) n =

case infer' e n of

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

        case mgu [ ??? ] of
```

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

Ejemplo

```
infer' (SuccExp e) n =

case infer' e n of

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

        case mgu [ (t', TNat) ] of



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

Ejemplo

```
infer' (SuccExp e) n =

case infer' e n of

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

        case mgu [ (t', TNat) ] of

            UOK subst ->

                        (

                         c',

                        )

            UError u1 u2 ->

                uError u1 u2

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

Ejemplo

```
infer' (SuccExp e) n =

case infer' e n of

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

        case mgu [ (t', TNat) ] of

            UOK subst ->

                        (

                         c',
                         SuccExp e,

                        )

            UError u1 u2 ->

                uError u1 u2

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

Ejemplo

```
infer' (SuccExp e) n =

case infer' e n of

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

        case mgu [ (t', TNat) ] of

            UOK subst ->

                        (

                         c',
                         SuccExp e',

                        )

            UError u1 u2 ->

                uError u1 u2

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

Ejemplo

```
infer' (SuccExp e) n =

case infer' e n of

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

        case mgu [ (t', TNat) ] of

            UOK subst ->

                              (

                               c',
                               SuccExp e',
                               t'

                              )

            UError u1 u2 ->

                uError u1 u2

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

Ejemplo

```
infer' (SuccExp e) n =

case infer' e n of

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

        case mgu [ (t', TNat) ] of

            UOK subst ->

                              (

                               c',
                               SuccExp e',
                               TNat

                              )

            UError u1 u2 ->

                uError u1 u2

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

Ejemplo

```
infer' (SuccExp e) n =

case infer' e n of

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

        case mgu [ (t', TNat) ] of

            UOK subst ->

                        (

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

                        )

            UError u1 u2 ->

                uError u1 u2

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

Ejemplo

```
infer' (SuccExp e) n =

case infer' e n of

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

        case mgu [ (t', TNat) ] of

            UOK subst ->

                OK ( n', (

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

                         ) )

            UError u1 u2 ->

                uError u1 u2

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