Trabajo Práctico 2
Inferencia de tipos

Paradigmas de Lenguajes de Programación — 2^{do} cuat. 2020

Fecha de entrega: 29 de Octubre de 2020

Introducción

Este trabajo consiste en implementar en Haskell el algoritmo de inferencia de tipos $\mathbb{W}$ visto en clase. Además se pide definir e implementar una extensión para listas y su función zipWith (similar a la de Haskell, aunque sin currificar).

El código del TP incluye un parser, y una función de inferencia `inferExpr` a medio implementar. El objetivo es que al completar esta implementación podamos obtener juicios de tipado a partir de expresiones no anotadas, como por ejemplo:

```
*Main> inferExpr "0"
{} >> 0 : Nat

*Main> inferExpr "\\x → succ(f x)"
{f:t1 → Nat} >> \x:t1 → succ(f x) : t1 → Nat
```

Representación en Haskell del cálculo lambda con Nat y Bool

Las expresiones de tipo del cálculo lambda, incluyendo variables de tipo

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

se representarán en Haskell mediante el tipo `Type`:

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

Las expresiones del cálculo lambda

$$M ::= 0 \mid \mathsf{true} \mid \mathsf{false} \mid x \mid \mathsf{succ}(M) \mid \mathsf{pred}(M) \mid \mathsf{iszero}(M) \mid \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M \mid M\ M \mid \lambda x : \sigma.M$$

se representarán mediante el tipo `Exp`:

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

`Symbol` está definido como `String`, y lo usamos para representar a las variables mediante un nombre. `Exp a` representa a las expresiones cuyas anotaciones tienen tipo `a`. Cuando la expresión no tiene su tipo anotado, le podemos pasar `()`, es decir Unit, en el lugar del tipo `Type`:

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

Usando la función `parseLC` se puede obtener la expresión correspondiente a un término representado por un string, por ejemplo:

```
*Main> parseLC "succ(0)"
SuccExp ZeroExp

*Main> parseLC "\\x → succ(f x)"
LamExp "x" () (SuccExp (AppExp (VarExp "f") (VarExp "x")))
```

Contextos de tipado

Los contextos de tipado para los juicios que iremos construyendo estarán representados por el tipo `Context`, que está definido como listas de pares: `[(Symbol, Type)]`. Es decir, que a cada variable libre representada por `Symbol` le corresponderá un tipo `Type`.

```
data Context = C [(Symbol,  Type)] deriving Show
```

Cuentan con las siguientes operaciones sobre contextos implementadas:

`emptyContext :: Context`
: Representa el contexto vacío. A partir de él se pueden crear nuevos contextos.

`extendC :: Context -> Symbol -> Type -> Context`
: Permite agregar un nuevo par `(Symbol, Type)` a un contexto.

`removeC :: Context -> Symbol -> Context`
: Borra del contexto a la variable asociada al símbolo pasado por parámetro.

`evalC :: Context -> Symbol -> Type`
: Dado un contexto y el nombre de una variable, devuelve su tipo asociado.

`joinC :: [Context] -> Context`
: Une los contextos de la lista pasada por parámetro.

`domainC :: Context -> [Symbol]`
: Devuelve una lista con todas las variables contenidas en el contexto.

Sustituciones

Representamos a las sustituciones mediante el tipo `Subst`, que se define como funciones: `Int -> Type`. Es decir, funciones que asignan tipos a variables de tipo, ya que estas variables están indicadas mediante un entero.

Cuentan con las siguientes operaciones definidas sobre sustituciones:

`emptySubst :: Subst`
: Define la sustitución vacía.

`extendS :: Int -> Type -> Subst -> Subst`
: Extiende la sustitución asignándole el tipo pasado por parámetro a la variable de tipo correspondiente al entero pasado por parámetro.

Las sustituciones pueden aplicarse sobre tipos, contextos o expresiones, mediante la operación `<.>`:

```
class Substitutable a where
      (<.>) :: Subst → a → a

instance Substitutable Type -- subst <.> t
instance Substitutable Context -- subst <.> c
instance Substitutable Exp -- subst <.> e
```

Unificación

Para realizar unificaciones, pueden utilizar la función `mgu`:

```
type UnifGoal = (Type, Type)
data UnifResult = UOK Subst | UError Type Type

mgu :: [UnifGoal] → UnifResult
```

Esta función recibe una lista de los pares de tipos a unificar y, si la unificación es exitosa, devuelve la sustitución resultado. En caso contrario, devuelve un error informando qué par no pudo ser unificado.

Inferencia de tipos

El objetivo de este taller es completar la implementación de la función que realiza inferencia de tipos para las expresiones no anotadas, y que devuelve juicios de tipado. Los juicios de tipado están representados por tuplas de tres elementos: un contexto de tipado, una expresión anotada y un tipo. La función `inferType`, (en el archivo `TypeInference.hs`) que realiza la inferencia de tipos, recibe una expresión sin anotaciones y devuelve un resultado, que puede ser un juicio de tipado o un error en el caso de que la expresión no sea tipable.

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

inferType :: PlainExp → Result TypingJudgment
```

La función `inferType` usa la función auxiliar `infer'` para administrar el uso de variables de tipo frescas. Esta función toma una expresión no anotada y un entero (que representa el número de la última variable fresca usada) y devuelve un resultado que, si no es un error, además de contener el juicio de tipado contiene el número de la próxima variable fresca.

```
inferType e = case infer' e 0 of
      OK (_, tj) → OK tj
      Error s → Error s
```

Resolver

Ejercicio 1

Completar la implementación de la función `infer'` en el archivo `TypeInference.hs` para los casos `ZeroExp`, `VarExp`, `AppExp` y `LamExp`, según el algoritmo $\mathbb{W}$ visto en clase. Esta función hace pattern matching sobre `Exp` y usa recursión explícita. El caso de `succExp` ya viene implementado como ejemplo, y se cuenta con la función `uError` para mostrar errores de unificación.

Para probar el código hay que cargar el archivo `Main.hs`. Luego se puede usar la función `inferExpr :: String -> Doc`, que toma una expresión no anotada como un string, la parsea y le aplica `inferType`.

Se pueden usar como ejemplos para probar las expresiones definidas en `Examples.hs`, las cuales se pueden llamar mediante `expr n` donde `n` es un entero entre `1` y `22`.

Por ejemplo:

`inferExpr "succ(x)"` debe devolver `x:Nat >> succ(x) : Nat`.

`inferExpr (expr 1)` debe devolver `x:t0 >> x : t0`.

Ejecutando `runTest testEj1` se testea este ejercicio usando las 22 expresiones definidas. También pueden correrse grupos de tests individuales para cada caso (por ejemplo `runTest testZeroExp`), definidos en `Main.hs`.

Ejercicio 2 (opcional)

Completar la implementación de la función `infer'` para los casos `PredExp`, `IsZeroExp`, `TrueExp`, `FalseExp` e `IfExp`. Los tests pueden correrse con `runTest testEj2`.

Ejercicio 3

Se agregan los siguientes tipos y términos al cálculo lambda:

$$\sigma ::= \ldots \mid [\sigma] \qquad M ::= \ldots \mid [\,]_\sigma \mid M :: M \mid \mathsf{zip}\ M\ \mathsf{and}\ M\ \mathsf{with}\ x, y\ \rightsquigarrow\ M$$

El tipo $[\sigma]$ representa a las listas de términos de tipo σ. El término $[\,]_\sigma$ es la lista vacía de elementos de tipo σ. Los términos de la forma $M :: N$ representan la construcción de una lista, agregando el elemento M a la lista N. El término $\mathsf{zip}\ M\ \mathsf{and}\ N\ \mathsf{with}\ x, y\ \rightsquigarrow\ O$ es similar al `zipWith` de Haskell:

dadas dos listas M y N, no necesariamente del mismo tipo, y un término O que puede tener libres a las variables x e y, es la lista que resulta de recorrer simultáneamente a M y N, ligando las variables x e y en O con los elementos que se están recorriendo en cada paso.

Las reglas de tipado para los constructores de listas y para zipWith son las siguientes:

$$\frac{}{\Gamma \rhd [\,]_\sigma : [\sigma]}(\text{T-Empty}) \qquad \frac{\Gamma \rhd M : \sigma \quad \Gamma \rhd N : [\sigma]}{\Gamma \rhd M :: N : [\sigma]}(\text{T-Cons})$$

$$\frac{\Gamma \rhd M : [\sigma] \quad \Gamma \rhd N : [\tau] \quad \Gamma, x : \sigma, y : \tau \rhd O : \rho}{\Gamma \rhd \mathsf{zip}\ M\ \mathsf{and}\ N\ \mathsf{with}\ x, y\ \rightsquigarrow\ O : [\rho]}(\text{T-ZipWith})$$

Representamos el tipo de las listas en Haskell mediante `TList Type`. Las nuevas expresiones estarán dadas por los siguientes constructores:

```
data Exp = ...
         | EmptyListExp a
         | ConsExp (Exp a) (Exp a)
         | ZipWithExp (Exp a) (Exp a) Symbol Symbol (Exp a)
```

a) Definir la extensión del algoritmo $\mathbb{W}$ para los dos constructores de listas y para las expresiones zipWith, según las reglas de tipado dadas. La solución debe seguir el formato presentado en la teórica (o en el machete).

b) Completar la implementación de la función `infer'` para estos tres casos, según la extensión definida en el punto a. En este caso se puede correr `runTest testEj3` para ejecutar los tests.

Tests

Para instalar HUnit usar: `> cabal install hunit` o bien `apt install libghc-hunit-dev`.

Para instalar cabal ver: https://wiki.haskell.org/Cabal-Install

Pautas de Entrega

Se debe enviar un e-mail conteniendo el código fuente en Haskell a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;TP-I]` seguido inmediatamente del **nombre del grupo**.
- El código Haskell debe acompañar el e-mail y lo debe hacer en forma de archivo adjunto (puede adjuntarse un `.zip` o `.tar.gz`).
- La extensión al algoritmo $\mathbb{W}$ del ejercicio 3 puede entregarse en un archivo aparte (pdf) ya sea hecho en LaTeXo escrito a mano y escaneado.

El código debe poder ser ejecutado en Haskell2010. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté **adecuadamente** comentado (son comentarios adecuados los

que ayudan a entender lo que no es evidente o explican decisiones tomadas; no son adecuadas las traducciones al castellano del código). Los objetivos a evaluar son:

- Corrección.
- Declaratividad.
- Prolijidad: evitar repetir código innecesariamente y usar adecuadamente las funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).
- Uso adecuado de funciones de alto orden, currificación y esquemas de recursión: es necesario para los ejercicios que usen las funciones que vimos en clase y aprovecharlas, por ejemplo, usar zip, map, filter, take, takeWhile, dropWhile, foldr, foldl, listas por comprensión, etc, cuando sea necesario y no volver a implementarlas.

Para el desarrollo de este TP, **se permite utilizar recursión explícita sobre estructuras de tipo `Exp`**, dado que este tipo tiene demasiados constructores para poder escribir código claro que lo recorra usando `fold`. Se permite utilizar listas por comprensión y esquemas de recursión definidos en el preludio de Haskell y los módulos `Prelude`, `Data.Char`, `Data.Function`, `Data.List`, `Data.Maybe`, `Data.Ord` y `Data.Tuple`. Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas. Pueden escribirse todas las funciones auxiliares que se requieran, pero estas no pueden usar recursión explícita (ni mutua, ni simulada con `fix`) sobre tipos que no sean `Exp`.

Importante: se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

Referencias del lenguaje Haskell

Como principales referencias del lenguaje de programación Haskell, mencionaremos:

- **The Haskell 2010 Language Report:** el reporte oficial de la última versión del lenguaje Haskell a la fecha, disponible online en: http://www.haskell.org/onlinereport/haskell2010.
- **Learn You a Haskell for Great Good!**: libro accesible, para todas las edades, cubriendo todos los aspectos del lenguaje, notoriamente ilustrado, disponible online en: http://learnyouahaskell.com/chapters.
- **Real World Haskell**: libro apuntado a zanjar la brecha de aplicación de Haskell, enfocándose principalmente en la utilización de estructuras de datos funcionales en la "vida real", disponible online en http://book.realworldhaskell.org/read.
- **Hoogle**: buscador que acepta tanto nombres de funciones y módulos, como signaturas y tipos *parciales*, online en http://www.haskell.org/hoogle.
- **Hayoo!**: buscador de módulos no estándar (i.e. aquellos no necesariamente incluídos con la plataforma Haskell, sino a través de **Hackage**), online en http://holumbus.fh-wedel.de/hayoo/hayoo.html.