Práctica 4
Inferencia de tipos

Favio E. Miranda Perea (favio@ciencias.unam.mx)
Pablo G. González López (pablog@ciencias.unam.mx)

Miércoles 9 de octubre de 2019

Fecha de entrega: Miércoles 23 de octubre de 2019 a las 23:59:59.

1. Mini Haskell (`MinHs`)

`MinHs` es un pequeño lenguaje de programación que implementa los conceptos y mecanismos esenciales del paradigma de programación funcional vistos hasta ahora en clase.

Las expresiones de este lenguaje son:

```
data Expr = V Identifier | I Int | B Bool
  | Fn Identifier Expr
  | Succ Expr | Pred Expr
  | Add Expr Expr | Mul Expr Expr
  | Not Expr
  | And Expr Expr | Or Expr Expr
  | Lt Expr Expr | Gt Expr Expr | Eq Expr Expr
  | If Expr Expr Expr
  | Let Identifier Expr Expr
  | App Expr Expr
```

Ya hemos implementado la semántica dinámica y parte de la semántica estática. Sin embargo, cuando agregamos el núcleo funcional, suspendimos el análisis estático. Si se ingresa una expresión inválida a nuestro evaluador, este comenzará la evaluación y eventualmente devolverá un error de ejecución. Queremos evitar evaluar expresiones inválidas por lo que tendremos que completar las reglas del análisis estático.

1.1. Problema

Existen dos clasificaciones de los lenguajes de programación fuertemente tipados. Los estrictamente tipados, donde el programador está obligado a hacer anotaciones de tipo en el programa (como *Pascal* o *Java*) y los lenguajes tipados pero sin anotaciones de tipos (como *Haskell* o *ML*). Ambas generan problemas

relacionados con respecto al proceso de revisar si los tipos de un programa dado son correctos.

- **Problema de la verificación de tipos**. Dada una expresión e con anotaciones de tipos, un contexto Γ y un tipo T, verificar si $\Gamma \vdash e : T$.
- **Problema de la inferencia de tipos.** Dada una expresión e, y si e^a es su equivalente con anotaciones de tipos, tratar de encontrar Γ y T tales que $\Gamma \vdash e^a : T$.

1.2. Algoritmo de inferencia

1.2.1. Tipos

Para definir el algoritmo de inferencia, debemos extender la categoría de tipos como sigue:

```
type Identifier = Int
infix :->

data Type = T Identifier
          | Integer | Boolean
          | Type :-> Type

type Substitution = [(Identifier, Type)]
```

Definiremos variables de tipos con el constructor `T` y su conjunto de identificadores como `Int`. El operador `:->` representará al tipo función.

La sustitución de tipos de define como un conjunto (representado con una lista) de duplas de identificadores de tipo y tipos.

Agrega esta definición e implementa las siguientes funciones en un módulo llamado `Type`:

1. (0,5 puntos) `vars`. Función que devuelve el conjunto de variables de tipo.

   ```
   vars :: Type -> [Identifier]
   ```

   Ejemplo:

   ```
   *Main> vars (T 1 :-> (T 2 :-> T 1))
   [1, 2]
   *Main> vars (Integer :-> (Boolean :-> Integer))
   []
   ```

2. (0,5 puntos) `subst`. Función que aplica la sustitución a un tipo dado.

   ```
   subst :: Type -> Substitution -> Type
   ```

   Ejemplo:

```
*Main> subst (T 1 :-> (T 2 :-> T 1)) [(3, T 4),
    (5, T 6)]
T1 -> (T2 -> T1)
*Main> (T 1 :-> (T 2 :-> T 1)) [(1, T 2), (2, T
    3)]
T2 -> (T3 -> T2)
```

3. (1 puntos) `comp`. Función que realiza la composición de dos sustituciones.

```
comp :: Substitution -> Substitution ->
    Substitution
```

Ejemplo:

```
*Main> comp [(1, T 2 :-> T 3), (4, T 5)] [(2, T
    6)]
[(1, T6 -> T3), (4, T5), (2, T6)]
```

4. (0,25 puntos) `simpl`. Función que elimina sustituciones redundantes ($T_0 := T_0$) en una sustitución.

```
simpl :: Substitution -> Substitution
```

Ejemplo:

```
*Main> simpl [(1, T 2 :-> T 3), (4, T 5)]
[(1, T2 -> T3), (4, T5)]
*Main> simpl [(0, T 0), (1, T 2 :-> T 3), (4, T 5
    :-> T 6), (6, T 6)]
[(1, T2 -> T3), (4, T5 -> T6)]
```

1.2.2. Definición

El algoritmo consta de dos pasos, el primero consiste en construir una derivación de tipos con restricciones. Para este propósito definimos una relación cuaternaria $R|\Gamma \vdash e : T$, cuyo significado es $\Gamma \vdash e : T$ *se cumple cuando el conjunto de restricciones* R *es satisfacible.*

Las siguientes reglas definen a esta relación y permiten construir sus derivaciones:

- Tipado de variables.

$$\frac{x : T \in \Gamma}{\emptyset|\Gamma \vdash x : T}$$

- Tipado de constantes.

$$\frac{}{\emptyset|\emptyset \vdash n : Integer}$$

$$\frac{}{\emptyset|\emptyset \vdash b : Boolean}$$

- Tipado de funciones.

$$\frac{R|\Gamma, x : S \vdash e : T}{R|\Gamma \vdash fn(x.e) : S \to T}$$

- Tipado de operadores unarios.

$$\frac{R|\Gamma \vdash e : S}{R, S = Integer|\Gamma \vdash succ(e) : Integer}$$

  Similar para $pred()$ y $not()$.

- Tipado de operadores n-narios.

$$\frac{\begin{array}{c} R_1|\Gamma_1 \vdash e_1 : T_1 \\ R_2|\Gamma_2 \vdash e_2 : T_2 \\ vars(R_1 \cup \Gamma_1 \cup T_1) \cap vars(R_2 \cup \Gamma_2 \cup T_2) = \emptyset \\ S = \{S_1 = S_2 | x : S_1 \in \Gamma_1, x : S_2 \in \Gamma_2\} \end{array}}{R_1, R_2, S, T_1 = Integer, T_2 = Integer|\Gamma_1, \Gamma_2 \vdash add(e_1, e_2) : Integer}$$

  Similar para $mul()$, $and()$, $or()$, $lt()$, $gt()$, $eq()$ e $if()$.

- Tipado de aplicaciones.

$$\frac{\begin{array}{c} R_1|\Gamma_1 \vdash e_1 : T_1 \\ R_2|\Gamma_2 \vdash e_2 : T_2 \\ vars(R_1 \cup \Gamma_1 \cup T_1) \cap vars(R_2 \cup \Gamma_2 \cup T_2) = \emptyset \\ S = \{S_1 = S_2 | x : S_1 \in \Gamma_1, x : S_2 \in \Gamma_2\} \\ Z\ fresh \end{array}}{T_1 = T_2 \to Z, S, R_1, R_2|\Gamma_1, \Gamma_2 \vdash e_1 e_2 : Z}$$

- Tipado de expresión *let*.

$$\frac{\begin{array}{c} R_1|\Gamma_1 \vdash e_1 : T_1 \\ R_2|\Gamma_2, x : S \vdash e_2 : T_2 \\ vars(R_1 \cup \Gamma_1 \cup T_1) \cap vars(R_2 \cup \Gamma_2 \cup T_2) = \emptyset \\ S = \{S_1 = S_2 | x : S_1 \in \Gamma_1, x : S_2 \in \Gamma_2\} \end{array}}{R_1, R_2, S, T_1 = S|\Gamma_1, \Gamma_2 \vdash let(e_1, x.e_2) : T_2}$$

donde $Z\ fresh$ significa que Z es una variable de tipo nueva, es decir, una variable que no figura en ninguna de las premisas anteriores de la regla.

Para implementar el algoritmo modelaremos el contexto y el conjunto de restricciones como se muestra a continuación:

```
type Ctxt = [(Identifier, Type)] -- Identifier definido
    en el modulo Sintax.

type Constraint = [(Type, Type)]
```

Agrega estas definiciones e implementa las siguientes funciones en un módulo llamado `Static`:

1. (0,5 puntos) `subst`. Función que aplica una sustitución (de variables de tipo) a un contexto dado.

```
subst :: Ctxt -> Substitution -> Ctxt --
    Substitution definido en el modulo Type
```

   Ejemplo:

```
*Main> subst [("x", T 0)] [(0, T 1), (1, T 2)]
[("x", T1)]
```

2. (0,25 puntos) `find`. Función que busca el tipo de una variable en un contexto dado. Devuelve el tipo en caso de encontrarlo y `Nothing` en otro caso.

```
find :: Identifier -> Ctxt -> Maybe Type --
    Identifier definido en el modulo Sintax
```

   Ejemplo:

```
*Main> find "x" [("x", T 0), ("y", Boolean)]
Just T0
*Main> find "z" [("x", T 0), ("y", Boolean)]
Nothing
```

Para modelar la creación de variables de tipo *frescas*, agregaremos a la entrada y salida del algoritmo un conjunto de variables de tipo que nos ayude a encontrarlas rápidamente. La idea es que el conjunto de entrada representará todas las variables de tipo que se han usado hasta ese momento durante la ejecución del algoritmo, y el conjunto de salida será este mismo además de todas las variables de tipo que se usaron para inferir el tipo de la expresión actual.

Implementa las siguientes funciones:

1. (1 punto) `fresh`. Dado un conjunto de variables de tipo, obtiene una variable de tipo *fresca*, es decir, que no aparece en este conjunto.

```
fresh :: [Type] -> Type
```

   Ejemplo:

```
*Main> fresh [T 0, T 1, T 2, T 3]
T4
*Main> fresh [T 0, T 1, T 3, T 4]
T2
```

2. (5 puntos) `infer'`. Dada una expresión, infiere su tipo implementando las reglas descritas anteriormente. Devolviendo el contexto y el conjunto de restricciones donde es válido. Utiliza el conjunto de variables de tipo para crear variables de tipo *frecas* durante la ejecución.

```
infer' :: ([Type], Expr) -> ([Type], Ctxt,
    Type, Constraint)
```

Ejemplo:

```
*Main> infer' ([], Fn "x" (V "x"))
([T0], [], T0 -> T0, [])
*Main> infer' ([], Add (V "x") (V "x"))
([T0, T1], [("x", T0), ("x", T1)], Integer, [(T0,
    T1), (T0, Integer), (T1, Integer)])
*Main> infer' ([], Fn "x" (Fn "x" (Fn "x" (V "x")
    )))
([T0, T1, T2], [], T2 -> (T1 -> T0 -> T0), [])
*Main> infer' ([], Let "x" (B True) (And (V "x")
    (Let "x" (I 10) (Eq (I 0) (Succ (V "x"))))))
([T0, T1], [], Boolean, [(T1, Integer), (Integer,
    Integer), (Integer, T1), (T0, Boolean), (
    Boolean, Boolean), (Boolean, T0)])
```

Finalmente, cuando tenemos la derivación $R|\Gamma \vdash e : T$ debemos intentar resolver el conjunto de restricciones para obtener el tipo de la expresión. Esto es obtener el unificador más general (μ). Si es posible obtener μ, el tipo de la expresión será $\Gamma_\mu \vdash e : T_\mu$, en otro caso no es posible tipar la expresión y podríamos decir que está mal formada.

La implementación del algoritmo de unificación de Martelli-Montanari se encuentra en el módulo `Unifier` del sitio del curso. La función μ toma un conjunto de restricciones, y en caso de ser soluble devuelve una lista con una sustitución. En otro caso, lanza un error indicando qué ecuación no se pudo resolver.

Integra este módulo al proyecto e implementa las siguientes funciones:

1. (1 puntos) `infer`. Dada una expresión, infiere su tipo devolviendo el contexto donde es válido.

```
infer :: Expr -> (Ctxt, Type)
```

Ejemplo:

```
*Main> infer (Let "x" (B True) (And (V "x") (Let
    "x" (I 10) (Eq (I 0) (Succ (V "x"))))))
([], Boolean)
*Main> infer (Sintax.Let "x" (Sintax.V "p") (
    Sintax.Or (Sintax.Let "x" (Sintax.I 10) (
    Sintax.Eq (Sintax.I 0) (Sintax.Succ (Sintax.V
     "x")))) (Sintax.V "q")))
```

```
([("p", T0), ("q", Boolean)], Boolean)
```

Atención

Será un **requisito de entrega** integrar correctamente los módulos `Type` y `Static` al analizador sintáctico.

¡Suerte!