Práctica 4
Inferencia de tipos

Sandra del Mar Soto Corderi
Edgar Quiroz Castañeda

October 27, 2019

1 Desarrollo

1.1 Tipos

Lo relacionado al módulo `Type`. La creación de los tipos y los alias de tipos fue directa.

```
-- | Se extiende la categoría de tipos.
type Identifier = Int
infix :->

data Type = T Identifier
    | Integer | Boolean
    | Type :-> Type deriving (Show, Eq)

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

1.1.1 `vars`

El tipo `Type` es un tipo inductivo. Así que la función se definió recursivamente. Simplemente, si se tiene una variable, se agrega al conjunto de variables. Si se tiene un tipo concreto (`Boolean, Integer`), no se hace nada. Si se tiene una función, se hace recursión sobre las subestrucutras.

```
-- | Devuelve el conjunto de variables de tipo
vars :: Type -> [Identifier]
vars (T t) = [t]
vars (t1 :-> t2) = List.union (vars t1) (vars t2)
vars _ = []
```

1.1.2 `subst`

En este caso hay que hacer una doble inducción sobre la sustitución (que es una lista) y el tipo.

Primero se hace recursión sobre el tipo. Usando el mismo principio inductivo, si se tiene una variable, entonces se busca en la sustitución. Si se tiene un tipo concreto, no se hace nada. Si se tiene una función, se hace recursión.

Luego, cuando se tiene una variable de tipo, se busca la variable recursivamente en la sustitución.

```
-- | Aplica la sustitución a un tipo dado
subst :: Type -> Substitution -> Type
subst e@(T t) s =
    case s of
    [] -> e
    ((x, t'): ss) ->
        if x == t then t' else subst e ss
subst (t1 :-> t2) s = (subst t1 s) :-> (subst t2 s)
subst t _ = t
```

1.1.3 `comp`

Traducción literal de la definición de composición de sustituciones.

Esto es, que $\sigma \circ \rho$ corresponde a

$$(\sigma \circ \rho)(x) = \begin{cases} \rho(\sigma(x)) \text{ si } x \in dom(\sigma) \\ \rho(x) \text{ en otro caso} \end{cases}$$

Si se piensa en las sustituciones como listas (como en nuestra implementación) entonces esto corresponde a

- Remplazar cada tupla $(x, e) \in \sigma$ por (x, e_ρ)
- Añadir las tuplas de ρ cuyas variables no estaban definidas para σ.

En código esto sería

```
-- | Realiza la composición de dos sustituciones
comp :: Substitution -> Substitution -> Substitution
comp s1 s2 =
    List.union
    [(x, subst t s2) | (x, t) <- s1]
    [(x, t) | (x, t) <- s2, List.notElem x [y | (y, _) <- s1]]
```

1.1.4 `simpl`

Para esto se definió una función auxiliar `redundant` que indica si una sustitución es redundante.

Entonces, para quitar sustituciones redundantes sólo hay que aplicar un filtro usando esta función.

```
-- | Elimina sustituciones redundantes
simpl:: Substitution -> Substitution
simpl s = filter (\x -> not (redundant x)) s

-- | Verifica si una tupla de una sustitución es reduntante. Auxiliar.
redundant :: (Identifier, Type) -> Bool
redundant (i, T t) = i == t
redundant _ = False
```

1.2 Inferencia

Lo relacionado al módulo `Static`. Los tipos y alias utilizados se crearon tal cuál se especificó en la descripción de la práctica.

```
-- | Definiendo el contexto para tipos de variables
type Ctxt = [(Sintax.Identifier, Type.Type)]

-- | Definiendo las restricciones para inferir los tipos
type Constraint = [(Type.Type, Type.Type)]
```

1.2.1 `subst`

Únicamente se realizó la sustitución sobre cada una de las tuplas del contexto usando la función `Type.subst` definida anteriormente.

```
-- | Realiza sustituciones de variables de tipo
subst :: Ctxt -> Type.Substitution -> Ctxt
subst [] _ = []
subst ((x, t): cs) s = (x, Type.subst t s) : subst cs s
```

1.2.2 `find`

Se realiza la búsqueda recusrivamente sobre el contexto. De no encontrar nada, se regresa `Nothing`.

```
-- | Busca el tipo de una variable en un contexto
find :: Sintax.Identifier -> Ctxt -> Maybe Type.Type
find _ [] = Nothing
find x ((y, t) : cs) =
    if x == y
    then Just t
    else find x cs
```

1.2.3 `fresh`

Para esto, se usó la función `Type.vars` que regresa los identificadores de las variables en una expresión.

De entre ellos, se obtuvo el más grande, y se sumó uno para obtener un identificador que no estaba presente en ese conjunto.

```
-- | Obtiene una variable que no figure en la lista
fresh :: [Type.Type] -> Type.Type
fresh s =
    Type.T ((foldl max 0 (foldr (\ x y -> List.union (Type.vars x) y) [] s)) + 1)
```

1.2.4 `infer'`

Como está especificado en la descripción de la práctica, esta función está dividida en seis partes principales, cada una expresada con un juicio.

Cada parte está compuesta por varias instancias de la regla correspondiente. Al ser todos los casos análogos, sólo se va a describir el caso general de cada sección.

- Variables

  Se obtiene un nuevo tipo para la variable, la cuál se agrega al catálogo de variables, así como al contexto.

  Y se añade un conjunto de restricciones vacío.

  ```
  -- varialbles
  infer' (nv, (Sintax.V x)) =
      let t = fresh nv; nv' = nv `List.union` [t]
          in (nv' ,[(x, t)] ,t , [])
  ```

- Constantes

  Sólo se da directamente el tipo. No se modificada nada más.

  ```
  -- T
  infer' (nv, (Sintax.T e)) = (nv, [], Type.T, [])
  ```

- Funciones

  Primero, es necesario tener el tipo, el contexto, las restricciones y el catálogo de tipos del cuerpo de la función. Esto se logra con una llamada recursiva.

  Luego, se necesita el tipo de la variable ligada.

  Si está en el cuerpo, entonces su tipo está en el contexto. Así que sólo hay que buscarlo y borrarlo.

  Si no figuraba en el cuerpo, entonces no estará en el contexto. Así que habrá que crear un nuevo tipo, y agregarlo al catálogo de tipos. No se debe modificar nada más.

  En ambos casos, el tipo de la función es un tipo función que recibe el tipo de la variable ligada y regresa el tipo del cuerpo de la función.

  ```
  -- functions
  infer' (nv, (Sintax.Fn x e)) =
  let (nv', g, t, r) = infer' (nv, e)
      in case find x g of
          Just t' -> (nv', filter (\(i, t) -> i /= x) g, t' Type.:-> t, r)
  ```

```
        Nothing ->
        let t' = fresh nv'; nv'' = nv' `List.union` [t']
            in (nv'', g, t' Type.:-> t, r)
```

- Operadores

  Primero, se necesita el tipo, restricciones, contexto y catálogo de tipo de los operados, así como el tipo que regresa la operación.

  Y se debe agregar a las restricciones que todos los diferentes tipos que pueda tener una variable en realidad son el mismo.

  El catálogo de tipos es la unión de los catálogos. El contexto en la unión de los contextos. Las restricciones son la unión de las restricciones, pero además hay que agregar que todos los operandos son del tipo requerido.

  Entonces, si se tienen $e_1, \ldots, e_n$ operandos y una función F que regresa un tipo T y que recibe n operandos de tipos $U_1, \ldots, U_1$. Entonces, el código para procesar esta función es

```
-- op
infer' (nv, (Sintax.T e1 ... en)) =
    let (nv1, g1, t1, r1) = infer' (nv, e1);
        ...;
        (nvn, gn, tn, rn) = infer' (nv{n-1}, en);
        s = [(s1, s2) | (x, s1) <- g1, (y, s2) <- g2, x == y]
        `List.union`
            ...
        `List.union`
            [(s{n-1}, sn) | (x, s{n-1}) <- g{n-1}, (y, sn) <- gn, x == y];
        in (
            nvn,
            g1 `List.union` ... `List.union` gn,
            T,
            r1 `List.union` ... `List.union` rn `List.union` s
            `List.union` [(t1, U1), ..., (tn, Un)]
            )
```

  Hay que notar que se está omitiendo conseguir los tipo T y U_i, lo cuál varia según el operador particular.

- Aplicaciones

  Se requiere toda la información de las dos subexpresiones, lo cuál se obtiene con llamadas recursivas.

  Como en la mayoría de los casos, hay que añadir como restricciones que todos los posibles tipos de una variable son en realidad el mismo.

  Se requiere un nuevo tipo que represente el tipo que regresa la aplicación.

  Y finalmente, como restricción extra hay que añadir que el tipo de la expresión a la izquierda sea un tipo función que tomo el tipo de la expresión a la derecha y regresa el tipo de la aplicación.

```
infer' (nv, (Sintax.App e1 e2)) =
let (nv1, g1, t1, r1) = infer' (nv, e1);
    (nv2, g2, t2, r2) = infer' (nv1, e2);
    s = [(s1, s2) | (x, s1) <- g1, (y, s2) <- g2, x == y];
    z = fresh nv2; --item para operadores
    nv' = nv2 `List.union` [z]
    in (
        nv',
        g1 `List.union` g2,
        z,
        r1 `List.union` r2 `List.union` s `List.union` [(t1, t2 Type.:-> z)
        )
```

- Ligado de variables

  Es similar al procesamiento de funciones.

  Sólo que hay que tomar en cuenta dos subexpresiones en lugar de una.

```
infer' (nv, (Sintax.Let x e1 e2)) =
let (nv1, g1, t1, r1) = infer' (nv, e1);
    (nv2, g2, t2, r2) = infer' (nv1, e2);
    (nv3, g3, t3) =
    case find x g2 of
        Just t' -> (nv2, filter (\(i, t) -> i /= x) g2, t')
        Nothing -> (nv2 `List.union` [t'], g2, t') where t' = fresh nv2;
    s = [(s1, s3) | (x, s1) <- g1, (y, s3) <- g3, x == y];
    in (
        nv3,
        g1 `List.union` g3,
        t2,
        r1 `List.union` r2 `List.union` s `List.union` [(t1, t3)]
        )
```

1.2.5 `infer`

Usando `infer'`, se obtiene el tipo sin restricciones, las restricciones y el contexto.

Usando la función `Unifier.μ`, se intenta unificar las restricciones.

Si esto se logra, se aplica el unificador tanto al contexto como al tipo de la expresión, que es lo que regresa la función.

```
infer :: (Sintax.Expr) -> (Ctxt, Type.Type)
infer e =
    let (_, g, t, r) = infer' ([], e); umg = Unifier.μ r
    in (subst g (head umg), Type.subst t (head umg))
```

1.3 Integración

1.3.1 Modificaciones a `Semantic`

Parte de la funcionanlidad que se añadió antes pertenecía al este módulo. Así que fue necesario partirlo en tres partes. Los módulos `Static`, `Dynamic` y `Type`.

Y se convirtió al módulo `Semantic` en una especie de proxy cuya única función es envolver estos tres módulos para no tener que realizar modificaciones ni a `Parser` ni a `Main`.

Tanto `Static` como `Type` son el tema de la sección anterior.

Así que sólo falta discutir `Dynamic`.

Ahí se guardaron todas las funciones para evaluar expresiones. En estas, se tuvo que añadir el caso para el operador `Fix`, que anteriormente no era parte del lenguaje.

Incluyendo aquellas que hacían uso del análisis estático para revisar los tipos antes de realizar la evaluación. Por lo que `Dynamic` hace uso de `Static`.

1.3.2 Modificaciones a `Sintax`

Únicamente se agregaron los casos necesarios para `Fix`.

1.3.3 Modificaciones a `Main`

Al tener `Type` y `Static` funciones con el mismo nombre, al ser envueltas en `Semantic` tuvieron que ser importados de manera calificada.

Así que al usar recursos de este módulo en `Main`, hubo que envolver los nombres.

2 Dificultades

La única gran dificultad fue la parte del algoritmo de inferencia de tipos que teníamos que implementar.

2.1 `infer'`

Hubo que tener mucho cuidado con el manejo de los contextos en las funciones y con el operador `let`.

Además, encontrar al manera adecuada de procesar el operador `if` fue tardado, al no haber un ejemplo sobre operadores ternarios en la descripción de la práctica.

Además, surgió el inconveniente que es posible tener varios tipos para una misma variable en un mismo contexto, así que para borrarla no siempre bastaba con eliminar la primera aparición.