Práctica 4

Implementación del algoritmo $\mathcal{W}$ para `MinHs`

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

Miércoles 10 de octubre de 2018

Fecha de entrega: Miércoles 24 de octubre de 2018 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.

Basados en la definición descrita en las notas de clase [1], las expresiones de este lenguaje serán las siguientes:

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

Y sus posibles tipos serán:

```
infixr :->

data Type
  = Integer
  | Boolean
  | T Identifier
  | Type :-> Type
  deriving Eq
```

Estas definiciones, junto con la función de sustitución para tipos, ya se encuentran implementadas en los módulos `Syntax.MinHs` y `Syntax.Type`.

También se brinda la definición de `UMinHs`:

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

que es un lenguaje semejante a `MinHs` con la diferencia que en `UMinHs` los tipos de los parámetros de las funciones y la variable ligada de la expresión `let` no se da explícitamente.

Trabajaremos con ambos lenguajes transformando sus expresiones de uno al otro. Ciertamente, el caso complicado será transformar expresiones de `UMinHs` a `MinHs`, y para realizar esto nos apoyaremos del algoritmo $\mathcal{W}$.

2. El algoritmo $\mathcal{W}$

$\mathcal{W}$ es un algoritmo para inferir tipos basado en sus usos. Formaliza la intuición de que un tipo puede ser inferido por la acción que realiza.

Para entender esto, analicemos el siguiente ejemplo:

```
def foo(s : String) = s.length

def bar(x1, x2) = foo(x1) + x2
```

Observando la definición de `bar`, es fácil inferir que su tipo debe ser `(String, Int) =>Int`. Basta con leer el cuerpo de la función y buscar el uso que se le da a los parámetros `x1` y `x2`. `x1` se *pasa* a la función `foo`, que espera un `String`. Por lo tanto, `x1` debe ser de tipo `String`, además que `foo` devuelve un valor de tipo `Int`. Con esto en mente, el operador `+` de la clase `Int` espera valores de tipo `Int` como parámetros, por lo que `x2` debe ser de tipo `Int`. Finalmente, sabemos que el operador `+` devuelve un nuevo valor de tipo `Int`, por lo que ahora sabemos cuál es el tipo de retorno de `bar`.

Este proceso es justo lo que el algoritmo $\mathcal{W}$ hace, examina el cuerpo de una función y calcula un conjunto de restricciones en función de cómo se utiliza cada valor. Esto es lo que hicimos cuando notamos que `foo` espera un parámetro de tipo `String`. Una vez que tenemos el conjunto de restricciones, el algoritmo las unifica y encuentra el tipo de la expresión. Si la expresión estaba bien tipada, las restricciones se podrán unificar, de lo contrario las restricciones serán contradictorias o in satisfacibles dados los tipos disponibles.

2.1. Definición

El algoritmo $\mathcal{W}$ se define recursivamente del siguiente modo:

Variables: $\mathcal{W}(x)$ devuelve $\{x : X\} \vdash x : X$ con X una variable de tipo nueva.

Aplicación: $\mathcal{W}(e_1 e_2)$ devuelve $(\Gamma_1 \cup \Gamma_2)\mu \vdash (e_1^a e_2^a)\mu : X\mu$, donde:

- $\mathcal{W}(e_1)$ devuelve $\Gamma_1 \vdash e_1^a : T_1$
- $\mathcal{W}(e_2)$ devuelve $\Gamma_2 \vdash e_2^a : T_2$
- $\mu = umg(\{S_1 = S_2 \mid x : S_1 \in \Gamma_1, x : S_2 \in \Gamma_2\} \cup \{T_1 = T_2 \to X\})$ con X una variable de tipo nueva.

Abstracción: Si $\mathcal{W}(e)$ devuelve $\Gamma \vdash e^a : S$ entonces:

- Si Γ tiene una declaración para x, digamos $x : R \in \Gamma$, entonces $\mathcal{W}(\lambda x.e)$ devuelve $\Gamma \setminus \{x : R\} \vdash \lambda x : R.e^a : R \to S$
- Si Γ no tiene una declaración para x, entonces $\mathcal{W}(\lambda x.e)$ devuelve $\Gamma \vdash \lambda x : X.e^a : X \to S$, con X una variable de tipo nueva.

Análogamente se define para las expresiones `let` y `fun`.

Operador Unario: $\mathcal{W}(succ(e))$ devuelve $\Gamma\mu \vdash succ(e^a)\mu : Integer$, donde:

- $\mathcal{W}(e)$ devuelve $\Gamma \vdash e^a : T_1$
- $\mu = umg(\{T_1 = Integer\})$

Análogamente se define para los demás operadores unarios.

Operador Binario: $\mathcal{W}(e_1 + e_2)$ devuelve $(\Gamma_1 \cup \Gamma_2)\mu \vdash (e_1^a + e_2^a)\mu : Integer$, donde:

- $\mathcal{W}(e_1)$ devuelve $\Gamma_1 \vdash e_1^a : T_1$
- $\mathcal{W}(e_2)$ devuelve $\Gamma_2 \vdash e_2^a : T_2$
- $\mu = umg(\{S_1 = S_2 \mid x : S_1 \in \Gamma_1, x : S_2 \in \Gamma_2\} \cup \{T_1 = Integer, T_2 = Integer\})$

Análogamente se define para los demás operadores binarios.

Para implementar el algoritmo modelaremos los juicios del siguiente modo:

```
type Ctx = [(Identifier, Type)]
type VType = Type

data Judgement = Assertion (Ctx, MH.Expr, Type)
```

Cada `Assertion` contendrá el contexto con las declaraciones de los tipos de las variables, la expresión en `MinHs` y el tipo de dicha expresión.

Implementa las siguientes funciones:

1. (1.5 puntos) `erase`. Dada una expresión tipada devuelve una expresión sin tipar.

   ```
   erase :: MH.Expr -> UMH.Expr
   ```

   Ejemplo:

   ```
   *Main> MH.FunF "potd" "x" Nat Nat (HM.If (MH.Eq (MH.V "x") (HM.I 0))
   (HM.I 1) (MH.Mul (MH.I 2) (MH.App (MH.V "potd") (MH.Pred (MH.V "x")))))
   potd(x.if(eq(V "x", I 0), I 1, mul(I 2, (V "potd" pred(V "x")))))
   ```

2. (0.5 puntos) `newVType`. Dada una lista de variables de tipo, genera una variable nueva.

   ```
   newVType :: [VType] -> VType
   ```

   Ejemplo:

   ```
   *Main> newVType [T 0, T 1]
   T2
   *Main> newVType [T 0, T 2, T 3]
   T1
   ```

3. (8 puntos) `w`. Dada una expresión sin tipar, realiza la inferencia de tipos usando el algoritmo $\mathcal{W}$ para devolver un juicio con la expresión tipada.

   ```
   w :: UMH.Expr -> Judgement
   ```

   Ejemplos en la documentación de la función.

¡Suerte!

Referencias

[1] Favio E. Miranda Perea; Lourdes Del Carmen González Huesca. Nota de clase 7: El cálculo lambda con tipos simples y el lenguaje pcf. (7), oct 2019.