Compiladores 2022-2
Facultad de Ciencias UNAM
Práctica 5: Inferencia de tipos

Lourdes del Carmen González Huesca Juan Alfonso Garduño Solís
Nora Hilda Hernández Luna

4 de mayo de 2022
Fecha de entrega: 13 de mayo del 2022

1. Lenguajes

El lenguaje resultante de aplicar los procesos definidos en las prácticas 2, 3 y 4 al lenguaje fuente es el lenguaje `L8` que se define con la gramática siguiente:

```
<programa> ::= <expr>

<expr> ::= <const>
         | <var>
         | (quot <const>)
         | (begin <expr> <expr>*)
         | (primapp <prim> <expr>*)
         | (if <expr> <expr> <expr>)
         | (lambda ([<var> <type>]*) <expr>)
         | (let ([<var> <type> <expr>])  <expr>)
         | (letrec ([<var> <type> <expr>]) <expr>)
         | (list <expr>*)
         | (<expr> <expr>*)

<const> ::=  <boolean>
         | <integer>
         | <char>

<boolean> ::= #t | #f

<integer> ::= <digit> | <digit><integer>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> ::= <car> | <car><var> | <car><digit> | <car><digit><var>

<car> ::= a | b | c | ... | z
```

```
<prim> ::= + | - | * | / | length | car | cdr

<type> ::= Bool | Int | Char | List | Lambda
```

Y definido con *Nanopass* como sigue:

```
(define-language L8
  (terminals
   (variable (x))
   (primitive (pr))
   (constant (c))
   (type (t)))
  (Expr (e body)
    x
    (quot c)
    (begin e* ... e)
    (primapp pr e* ...)
    (if e0 e1 e2)
    (lambda ([x* t*] ...) body)
    (let ([x t e]) body)
    (letrec ([x t e]) body)
    (letfun ([x t e]) body)
    (list e* ...)
    (e0 e1 ...)))
```

Este es el lenguaje de entrada para esta práctica.

Ejercicios

Para simplificar la inferencia de tipos se aplicarán cambios al lenguaje.

1. *(2 pts.)* Definir el proceso `curry`. Este proceso se encarga de currificar las expresiones `lambda` así como las aplicaciones de función.

   Para este proceso es necesario definir un nuevo lenguaje `L9` en el que el constructor `lambda` recibe un único argumento y la aplicación se simplifica a `(e0 e1)`.

   ```
   entrada : (curry '(foo x y))
   salida : '((foo x) y)
   ```

   ```
   entrada : (curry '(lambda ([x Int] [y Int]) (+ x y)))
   salida : '(lambda ([x Int]) (lambda ([y Int]) (+ x y)))
   ```

2. *(1 pt.)* Definir el proceso `type-const`. Este proceso se encarga de colocar las anotaciones de tipos correspondientes a las constantes de nuestro lenguaje.

Para este proceso es necesario definir un nuevo lenguaje `L10` en el que se agregue el constructor `(const t c)`, en donde `t` debe corresponder al tipo de la constante y se elimina el constructor `(quot c)`.

```
entrada : (type-const '(quot 5))
salida : '(const Int 5)
```

2. Inferencia de Tipos

El propósito de la inferencia de tipos es que dada una expresión e del lenguaje encontrar T el tipo correspondiente a e bajo un contexto Γ.
Primero se extiende el sistema de tipos agregando los tipos para el correcto tipado de `list` y `lambda`:

`T ::= Bool | Int | Char | List | Lambda | (T → T) | (List of T)`

Se puede observar que el constructor `(T → T)` es para funciones currificadas (de un único argumento) y hace que nuestro sistema de tipos permita construir una infinidad de funciones, es decir, existen una infinidad de tipos. Aunque se incorpora el tipo función se conserva el tipo `Lambda` para simplificar la implementación.

Por otro lado el tipo `(List of T)` modela listas homogéneas. Con esto surge la duda ¿Cuál es el tipo de la lista vacía, representada en nuestro lenguaje como `(list)`? La lista vacía es el único caso de polimorfismo del lenguaje, debido a las anotaciones de tipos en los argumentos del constructor `lambda`. Para resolverlo se le asocia el tipo `List` (`(list) : List`) siendo `List` un "subtipo"de `(List of T)`, es decir, `List` es unificable con `(List of T)` para cualquier tipo T del lenguaje.

El resto de los tipos modelan los tipos básicos del lenguaje como lo han hecho en las prácticas anteriores.

Ahora se define Γ como un conjunto de parejas de variables con su tipo $\Gamma = \{x_0 : \mathsf{T_0}, ..., x_n : \mathsf{T_n}\}$. Puede modelarse también como una función tal que $\Gamma(x_i) = \mathsf{T_i}$.

2.1. Algoritmo $\mathcal{J}$

En el curso de lenguajes de programación es usual abordar o estudiar el algoritmo de inferencia de tipos $\mathcal{W}$. Sin embargo en la práctica este algoritmo resulta muy ineficiente pues aplica sustituciones con demasiada frecuencia. En el desarrollo de un compilador o interprete es necesario que el algoritmo con el cual se infieren los tipos de una expresión sea eficiente. Es por eso que se utilizará una versión simplificada del Algoritmo $\mathcal{W}$, el Algoritmo $\mathcal{J}$.

> As it stands, $\mathcal{W}$ is hardly an efficient algorithm; substitutions are applied too often. It was formulated to aid the proof of soundness. We now present a simpler algorithm $\mathcal{J}$ which simulates $\mathcal{W}$ in a precise sense.
>
> (Robin Milner)

El Algoritmo $\mathcal{J}$ es una versión simplificada pero sobre todo eficiente del Algoritmo $\mathcal{W}$ y difiere en dos formas. Primero se adopta la idea de encontrar las sustituciones pero solo aplicarlas en caso de ser necesarias. Y en segundo lugar se busca el tipo asociado a la expresión, no la expresión tipada correspondiente.

Estos algoritmos fueron definidos sobre el lenguaje `Exp`. En esta práctica solo se mostrarán las reglas correspondientes a las expresiones de Cálculo Lambda puro (λ^U) con un constructor `let`. El resto de las reglas así como la definición completa del lenguaje pueden consultarse en el artículo original de Robin Milner.

Consideremos la siguiente definición:

$$e ::= x \mid \lambda x.e \mid (e\, e) \mid \texttt{let}\ x = e\ in\ e$$

Se define el Algoritmo $\mathcal{J}$ para el lenguaje anterior mediante las siguientes reglas de inferencia:

$$\frac{x : \mathsf{S} \in \Gamma \quad \mathsf{T} = inst(\mathsf{S})}{\Gamma \vdash x : \mathsf{T}}\ \lambda var$$

$$\frac{\Gamma \vdash \mathtt{e_0} : \mathsf{T_0} \quad \Gamma \vdash \mathtt{e_1} : \mathsf{T_1} \quad \mathsf{X} = newvar \quad unify(\mathsf{T_0} = \mathsf{T_1} \to \mathsf{X})}{\Gamma \vdash (\mathtt{e_0}\ \ \mathtt{e_1}) : \mathsf{X}}\ \lambda app$$

$$\frac{\mathsf{X} = newvar \quad \Gamma, x : \mathsf{X} \vdash \mathtt{e} : \mathsf{S}}{\Gamma \vdash \lambda x.\mathtt{e} : (\mathsf{X} \to\ \mathsf{S})}\ \lambda abs$$

$$\frac{\Gamma \vdash \mathtt{e_0} : \mathsf{T}' \quad \Gamma, x : \mathsf{T} \vdash \mathtt{e_1} : \mathsf{S} \quad \mathtt{unify}(\mathsf{T} = \mathsf{T}')}{\Gamma \vdash \mathtt{let\ x = e_0\ in\ e_1} : \mathsf{S}}\ \lambda let$$

En donde *inst* es una función que dice si un tipo es instancia de otro, esto es para tratar el polimorfismo del lenguaje, *unify* es una función que dice si dos tipos pueden ser unificados y *newvar* representa una variable de tipo que no figura en el resto de las expresiones, una variable nueva.

Estas reglas no son suficientes para utilizar el algoritmo $\mathcal{J}$ como método de inferencia de tipos en nuestro lenguaje. Se requiere definir nuevas reglas para el tipado de las expresiones. El lenguaje que se ha diseñado en las practicas anteriores carece de polimorfismo, eso simplifica la implementación del algoritmo de inferencia y no es necesario implementar la función *inst* ni contar con variables de tipos. La función *unify* se define en la siguiente sección.

2.2. Unify

Como se vio en la sección anterior, en el caso de la regla de aplicación, se debe calcular la sustitución y aplicarse solo en los casos en los que sea necesario, estos casos necesarios son cuando se tiene que cambiar una variable de tipo por un tipo definido. Como el sistema de tipos definido para esta práctica no tiene variables de tipo nunca será necesario aplicar la sustitución.

La función `(unify t1 t2)` se reduce a verificar si `t1` es unificable con `t2` sin regresar el unificador. La función se define con el siguiente código:

```
(define (unify t1 t2)
  (if (and (type? t1) (type? t2))
    (cond
      [(equal? t1 t2) #t]
      [(and (equal? 'List t1) (list? t2)) (equal? (car t2) 'List)]
      [(and (equal? 'List t2) (list? t1)) (equal? (car t1) 'List)]
      [(and (list? t1) (list? t2))
        (and (unify (car t1) (car t2)) (unify (caddr t1) (caddr t2)))]
      [else #f])
    (error "Se esperaban 2 tipos")))
```

Hay que prestar especial atención en los casos de Listas, pues como se mencionó anteriormente, el tipo `List` es un subtipo de `(List of T)` para cualquier tipo T del sistema. Esto se modela en la función `unify` al regresar `#t` cuando se intenta unificar `List` con `(List of T)`.

2.3. Sistema de Tipos

El algoritmo $\mathcal{J}$ visto en secciones anteriores infiere el tipo para expresiones de lenguaje `Exp`. En esta práctica se busca extender el algoritmo para que infiera el tipo de las expresiones de nuestro Lenguaje.

Las siguientes reglas de inferencia definen el algoritmo $\mathcal{J}$ para el lenguaje `L10`. Cada regla corresponde a un constructor del lenguaje.

$$\frac{x : \mathsf{T} \in \Gamma}{\Gamma \vdash x : \mathsf{T}}\ var$$

$$\frac{}{\Gamma \vdash (\texttt{const}\ \mathsf{T}\ \mathsf{c}) : \mathsf{T}}\ const$$

$$\frac{\Gamma \vdash \mathsf{e_i} : \mathsf{S_i} \quad \Gamma \vdash \mathsf{e_n} : \mathsf{T} \quad 0 \leq i < n}{\Gamma \vdash (\texttt{begin}\ \mathsf{e_0}\ \ldots\ \mathsf{e_n}) : \mathsf{T}}\ begin$$

$$\frac{\Gamma \vdash \mathsf{e_1} : \mathsf{Int} \quad \Gamma \vdash \mathsf{e_2} : \mathsf{Int} \quad \texttt{arit}(\oplus)}{\Gamma \vdash (\texttt{primapp}\ \oplus\ \mathsf{e_1}\ \mathsf{e_2}) : \mathsf{Int}}\ arit$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{List\ of\ T}}{\Gamma \vdash (\texttt{primapp car}\ \mathsf{e}) : \mathsf{T}}\ car$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{List\ of\ T}}{\Gamma \vdash (\texttt{primapp cdr}\ \mathsf{e}) : \mathsf{List\ of\ T}}\ cdr$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{List\ of\ T}}{\Gamma \vdash (\texttt{primapp lenght}\ \mathsf{e}) : \mathsf{Int}}\ length$$

$$\frac{\Gamma \vdash \mathsf{e_0} : \mathsf{Bool} \quad \Gamma \vdash \mathsf{e_1} : \mathsf{T} \quad \Gamma \vdash \mathsf{e_2} : \mathsf{T'} \quad \texttt{unify}(\mathsf{T} = \mathsf{T'})}{\Gamma \vdash (\texttt{if}\ \mathsf{e_0}\ \mathsf{e_1}\ \mathsf{e_2}) : \mathsf{T}}\ if$$

$$\frac{\Gamma, x : \mathsf{T} \vdash \mathtt{e} : \mathsf{S}}{\Gamma \vdash (\mathtt{lambda}\ ([\mathtt{x}\ \mathsf{T}])\ \mathtt{e}) : (\mathsf{T} \rightarrow \mathsf{S})}\ lambda$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathsf{T}' \quad \Gamma, x : \mathsf{T} \vdash \mathtt{e}_1 : \mathsf{S} \quad \mathtt{unify}(\mathsf{T} = \mathsf{T}')}{\Gamma \vdash (\mathtt{let}\ ([\mathtt{x}\ \mathsf{T}\ \mathtt{e}_0])\ \mathtt{e}_1) : \mathsf{S}}\ let$$

$$\frac{\Gamma, x : \mathsf{T} \vdash \mathtt{e}_0 : \mathsf{T}' \quad \Gamma, x : \mathsf{T} \vdash \mathtt{e}_1 : \mathsf{S} \quad \mathtt{unify}(\mathsf{T} = \mathsf{T}')}{\Gamma \vdash (\mathtt{letrec}\ ([\mathtt{x}\ \mathsf{T}\ \mathtt{e}_0])\ \mathtt{e}_1) : \mathsf{S}}\ letrec$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : (\mathsf{T}' \rightarrow \mathsf{S}') \quad \Gamma, x : (\mathsf{T} \rightarrow \mathsf{S}) \vdash \mathtt{e}_1 : \mathsf{R} \quad \mathtt{unify}((\mathsf{T} \rightarrow \mathsf{S}) = (\mathsf{T}' \rightarrow \mathsf{S}'))}{\Gamma \vdash (\mathtt{letfun}\ ([\mathtt{x}\ (\mathsf{T} \rightarrow \mathsf{S})\ \mathtt{e}_0])\ \mathtt{e}_1) : \mathsf{R}}\ letfun$$

$$\frac{}{\Gamma \vdash (\mathtt{list}) : \mathsf{List}}\ empty$$

$$\frac{\Gamma \vdash \mathtt{e_i} : \mathsf{T_i} \quad \mathsf{T} = part(\{\mathsf{T_0}, ..., \mathsf{T_n}\}) \quad unify(\mathsf{T} = \mathsf{T_i}) \quad 0 \leq i \leq n}{\Gamma \vdash (\mathtt{list}\ \mathtt{e}_0\ ...\ \mathtt{e_n}) : \mathsf{List\ of\ T}}\ list$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathsf{T}_0 \quad \Gamma \vdash \mathtt{e}_1 : \mathsf{T}_1 \quad unify(\mathsf{T}_0 = \mathsf{T}_1 \rightarrow \mathsf{R})}{\Gamma \vdash (\mathtt{e}_0\ \ \mathtt{e}_1) : \mathsf{R}}\ app$$

Ejercicios

1. *(5 pts.)* Implementar la función `J` con base en las reglas definidas para el algoritmo $\mathcal{J}$ sobre el lenguaje `L10`. Esta función recibe una expresión del lenguaje y un contexto inicial, y regresa el tipo correspondiente a la expresión.

   Se recomienda utilizar `nanopass-case` para la definición de ésta función.

   ```
   entrada : (J (parser-L10 '(lambda ([x Int]) x)) '())
   salida : '(Int → Int)
   ```

2. *(2 pts.)* Definir el proceso `type-infer` que se encarga de quitar la anotación de tipo `Lambda` y sustituirlas por el tipo `'(T → T)` que corresponda a la definición de la función. Y sustituye las anotaciones de tipo `List` por el tipo `(List of T)` **de ser necesario**.

   Para este proceso no es necesario definir un nuevo lenguaje. Como se mencionó anteriormente el tipo `Lambda` no será eliminado del lenguaje para simplificar la implementación y el tipo `List` para tipar la lista vacía.

   Hint: Utilizar la función `J`. Recordando que las asignaciones `let` ya no tienen funciones.

   ```
   entrada : (type-infer (letrec ([foo Lambda (lambda ([x Int]) x)])
                                 (foo 5)))
   salida : (letrec ([foo (Int → Int) (lambda ([x Int]) x)]) (foo 5))
   ```

```
entrada : (type-infer (let ([x List (list)]) x))
salida : (let ([x List (list)]) x)
```

```
entrada : (type-infer (let ([x List (list 1 2 3 4)]) x))
salida : (let ([x (List of Int) (list 1 2 3 4)]) x)
```

Punto extra

Para obtener un punto extra responde lo siguiente en un archivo de texto anexo a los scripts de esta práctica:

- El algoritmo $\mathcal{W}$ utiliza como método de unificación de tipos el algoritmo de Martelli-Montanari. ¿Cuál otro algoritmo podrías utilizar para unificar un conjunto de ecuaciones de tipos?
- Tanto el algoritmo $\mathcal{W}$ como el algoritmo $\mathcal{J}$ infieren el tipo de una expresión. ¿A qué clase de complejidad pertenecen ambos algoritmos? ¿Y porqué se utiliza $\mathcal{J}$ en la implementación?[1]
- Muestra una ejecución donde el algoritmo $\mathcal{W}$ es más ineficiente que el algoritmo $\mathcal{J}$.

Para las dos primeras preguntas deberás dar referencias que apoyen tus afirmaciones.

Entrega

- La entrega será en el *classroom* del curso. Basta entregar los archivos comprimidos y el código bien documentado. Indicando los integrantes del equipo.
- Esta práctica debe realizarse en equipos de a lo más 3 personas.
- En los scripts entregados, deberán considerar incluir lo siguiente como parte de la documentación:
  - Los nombres y números de cuenta de los integrantes del equipo.
  - Descripción detallada del desarrollo de la práctica.
  - En caso de haber sido necesario, la especificación formal de los nuevos lenguajes definidos, utilizando gramáticas.
  - Comentarios, ideas o críticas sobre la práctica.
- Se recibirá la práctica hasta las 23:59:59 horas del día fijado como fecha de entrega, cualquier práctica recibida después no será tomada en cuenta.
- **Cualquier práctica total o parcialmente plagiada, será calificada automáticamente con cero y no se aceptarán más prácticas durante el semestre.**

[1] Puedes revisar el artículo donde definen a $\mathcal{W}$.