Universidad Nacional Autónoma de México

Facultad de Ciencias

Práctica 5.
Inferencia de Tipos.

Francisco Arturo Licón Colón
314222095

Kevin Miranda Sánchez
314011163

Victor Hugo Molina Bis
314311212

Compiladores.
Profesor: Lourdes del Carmen González Huesca.
27 de octubre de 2019

1. Desarrollo del programa

1. **Ejercicio 1.** *Definir el proceso curry. Este proceso se encarga de currificar las expresiones lambda así como las aplicaciones de función.*

   Para este ejercicio definimos un nuevo lenguaje L9.
   En este lenguajes el constructor lambda recibe un único parámetro y la aplicación se simplifica a (e0 e1).

   La definicón de este lenguaje es la siguiente.

```
(define-language L9
  (extends L8)
  (Expr (e body)
        (- (lambda ([x* t*] ...) body))
        (+ (lambda ([x t]) body))))
```

   Este nuevo lenguaje se define con la siguiente gramática.

   $< programa > ::= < expr >$
   $< expr > ::= < const >$
   $\quad | < var >$
   $\quad | (\text{quot} < const >)$
   $\quad | (\text{ begin} < expr > < expr >^*)$
   $\quad | (\text{ primapp} < prim > < expr >^*)$
   $\quad | (\text{ if} < expr > < expr > < expr >)$
   $\quad | (\text{ lambda } ([< var > < type >]) < expr >)$
   $\quad | (\text{ let } ([< var > < type > < expr >]) < expr >)$
   $\quad | (\text{ letrec } ([< var > < type > < expr >]) < expr >)$
   $\quad | (\text{ list} < expr >^*)$
   $\quad | (< expr > < expr >^*)$

   $< const > ::= < boolean >$
   $\quad | < integer >$
   $\quad | < char >$

   $< boolean > ::= \#\ \text{t} \mid \#\ \text{f}$

   $< integer > ::= < digit > \mid < digit > < integer >$

   $< digit > ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

   $< var > ::= < car > \mid < car > < var > \mid < car > < digit > \mid < car > < digit > < var >$

   $< char > ::= \text{a} \mid \text{b} \mid \text{c} \mid ... \mid \text{z}$

   $< prim > ::= + \mid - \mid * \mid / \mid \text{length} \mid \text{car} \mid \text{cdr}$

   $< type > ::= \text{Bool} \mid \text{Int} \mid \text{Char} \mid \text{List} \mid \text{Lambda}$

El proceso curry se define de la siguiente manera.

```
(define-pass curry : L8 (ir) -> L9 ()
 (Expr : Expr (ir) -> Expr ()
   [(lambda ([,x* ,t*] ...) ,[body])
   (let f ([x* x*] [t* t*])
    (if (not(null? x*))
      `(lambda ([,(car x*) ,(car t*)]) ,(f (cdr x*) (cdr t*)))
     body
     ))]))
```

2. **Ejercicio 2.** *Definir el proceso type-const. Este proceso se encarga de colocar las anotaciones de tipos correspondientes a las constantes de nuestro lenguaje.*

   Para este ejercicio definimos un nuevo lenguaje L10, donde se agrega el constructor (const t c) y se elimina el constructor (quot c).

   La definicón de este lenguaje es la siguiente.

```
(define-language L10
  (extends L9)
  (Expr (e body)
    (- (quot c))
    (+ (const t c))))
```

   Este nuevo lenguaje se define con la siguiente gramática.

   $< programa > ::= < expr >$
   $< expr > ::= < const >$
   $\quad | < var >$
   $\quad | ($ const $< type > < const >)$
   $\quad | ($ begin $< expr > < expr >^*)$
   $\quad | ($ primapp $< prim > < expr >^*)$
   $\quad | ($ if $< expr > < expr > < expr >)$
   $\quad | ($ lambda $([< var > < type >]) < expr >)$
   $\quad | ($ let $([< var > < type > < expr >]) < expr >)$
   $\quad | ($ letrec $([< var > < type > < expr >]) < expr >)$
   $\quad | ($ list $< expr >^*)$
   $\quad | (< expr > < expr >^*)$

   $< const > ::= < boolean >$
   $\quad | < integer >$
   $\quad | < char >$

   $< boolean > ::=$ # t | # f

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

   $< digit > ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

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

   $< char > ::=$ a | b | c | ... | z

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

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

El proceso type-const se define de la siguiente manera.

```
(define-pass type-const : L9 (ir) -> L10 ()
 (Expr : Expr (ir) -> Expr ()
  [(quot ,c)
  (if (integer? c)
    `(const Int ,c)
    (if (boolean? c)
      `(const Bool ,c)
      (if (char? c)
        `(const Char ,c)
        (error "Consatnte inválida")))))]))
```

3. **Ejercicio 1 (Inferencia de tipos).** *Implementar la función J con base a las reglas definidas para el algoritmo J sobre el lenguaje L10. Esta función recibe una expresión del lenguaje y un contexto inicial y regresa el tipo correspondiente a la expresión.*

   Para este ejercicio fue necesario definir 3 funciones.

   a) unify: Verifica si dos tipos son unificables.

```
(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")))
```

   b) part: Encuentra el tipo más particular en una lista de tipos.

```
(define (part lst)
    (if (equal? (car lst) 'List)
      (part (cdr lst))
      (car lst)))
```

   c) lookup: Busca un símbolo en la lista y regresa su tipo.

```
(define lookup
 (lambda (symbol env)
  (if(symbol? symbol)
    (let recursiveLookup([symbol symbol] [env env])
     (if(equal? symbol (car (car env)))
      (cdr (car env))
      (if(null? (cdr env))
        (error "El símblolo no pertence al contexto")
        (recursiveLookup symbol (cdr env)))))
    (error "No es un símbolo")))
```

La función J se define de la siguiente manera.

```
(define (J expr ctx)
 (nanopass-case (L10 Expr) expr
  [,x (lookup x ctx)]
  [(const ,t ,c) t]
  [(list) 'List]
  [(begin ,e* ... ,e) (J `,e ctx)]
  [(primapp ,pr ,[e*] ...)        (if (or (arit? pr) (equal? pr 'length))
     'Int
     (if (equal? pr 'car)
      (last (last e*))
      (if (equal? pr 'cdr)
       e*
       (error "Operador inválido")))))]
  [(if ,e0 ,e1 ,e2)
   (if (unify (J `,e1 ctx) (J `,e2 ctx))
    (J `,e1 ctx)
     (error "Los tipos no son unificables"))]
  [(lambda ([,x ,t]) ,body) (let ([nctx (set-add ctx (cons x t))]) `(,t → ,(J `,body nctx)))]
  [(let ([,x ,t ,e]) ,body)
   (if (unify (J `,e ctx) (let ([nctx (set-add ctx (cons x t))]) (J `,body nctx)))
    (let ([nctx (set-add ctx (cons x t))]) (J `,body nctx))
    (error "Los tipos no son unificables"))]
  [(letrec ([,x ,t ,e]) ,body)
   (if (unify (let ([nctx (set-add ctx (cons x t))]) (J `,e nctx)) (let ([nctx (set-add ctx (cons x t))]) (J
`,body nctx)))
     (let ([nctx (set-add ctx (cons x t))]) (J `,body nctx))
     (error "Los tipos no son unificables"))]
  [(letfun ([,x ,t ,e]) ,body)
   (if (unify (J `,e ctx) t)
    (let ([nctx (set-add ctx (cons x t))]) (J `,body nctx))
    (error "Los tipos no son unificables"))]
  [(list ,[e*] ... )
   (let f ([e e*])
    (if (null? e)
      `(List of ,(part e*))
      (if (unify (car e) (part e*))
        (f (cdr e))
        (error "Listas Homogeneas"))))]
  [(,e0 ,e1 ...)
   (if (unify (J `,e0 ctx) (J `,e1 ctx))
    (J `,e1 ctx)
    (error "Los tipos no son unificables"))]
  [else "Expr incorrecta"]))
```

4. **Ejercicio (Inferencia de tipos).** *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.*

El proceso type-infer se define de la siguiente manera.

```
(define-pass type-infer : L10 (ir) -> L10()
 (Expr : Expr(ir) -> Expr()
  [(letrec ([,x ,t ,e]) ,body)
  (if (equal? t 'Lambda)
```

```
    `(letrec ([,x ,(J `,e '()) ,e]) ,body)
    `(letrec ([,x ,t ,e] ,body)))]
[letfun ([,x ,t ,e]) ,body)
  (if (equal? t 'Lambda)
    `(letfun ([,x ,(J `,e '()) ,e]) ,body)
    `(letfun ([,x ,t ,e] ,body)))]))
```

2. Comentarios

1. Las primeras dos funciones fueron my sencillas de realizar, la segunda prácticamente se realizó en la practica pasada, por lo que solo se debía cambiar un poco dicha función.

2. El algortimo J fue un poco confuso de realizar al principio, pero después de entenderlo, la implementación resultó sencilla.

3. En los casos en el que el contexto de una expresión dependía de otra, la cual se encontraba antes fue un poco confusa. La implementación se realizó con un let, en el cual se definía el nuevo contexto, se debía concatenar el contexto a la primer parte de la expresión; dicha concatenación no se podía realizar con **append**, por lo que se optó por ocupar **set-add**, además, dada la definición de la funcio **lookup**, se debía colocar **cons** al inicio del segundo parámetro, al final, solo se aplicaba la función.

4. En el algoritmo J, en el caso de listas, la implementación no se realizó como se hizo en clase porque ésta resultó un poco confusa; se utilizó una forma la cual ha sido recurrente en nuestras prácticas anteriores, y el resultado en los casos definidos en el archivo fueron correctos.