Compiladores 2020-1
Facultad de Ciencias UNAM
Práctica 5: Inferencia de Tipos

Jaime Arturo García Campos Alejandro Tonatiuh Valderrama Silva

27 de Octubre, de 2019

Desarrollo

Antes de iniciar el desarrollo de esta práctica, recuperamos las funciones y predicados declarados en la práctica 4. Esto para continuar con la implementación de un parser que poco a poco va transformando tipos en expresiones.

Gramática Original

La gramática que describe el lenguaje utilizado para el desarrollo de esta práctica es la 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 | String
```

Ejercicio 1

Implementamos el proceso **curry**, el cual recibe una expresión en L8 y devuelve una expresión en L9 (donde el constructor lambda recibe un único parámetro y la aplicación se simplifica a e0 e1)

```
(define-pass curry : L8 (ir) -> L9 ()
  (definitions
    (define (currify vars types body)
      (cond
        [(empty? vars) (unparse-L9 body)]
        [else `(lambda ([,(car vars) ,(car types)])
                 ,(currify (cdr vars) (cdr types) body))])))
  (Expr : Expr (ir) -> Expr ()
        [(lambda ([,x* ,t*] ...) ,[body])
         (parser-L9 (currify x* t* body))]))
```

Ejercicio 2

Definimos el proceso **type-const** que coloca las anotaciones de tipos correspondientes a las constantes de nuestro leguaje. Este proceso recibe una expresión en L9 y devuelve una expresión en L10 (en el cual se agrega al constructor (const t c), donde t corresponde al tipo de la constante y se elimna al constructor (quot c). Definimos este lenguaje después de este proceso).

```
(define-pass type-const : L9 (ir) -> L10 ()
  (definitions
    (define (type c)
      (cond
        [(integer? c) `(const Int ,c)]
        [(char? c) `(const Char ,c)]
        [(boolean? c) `(const Bool ,c)])))
  (Expr : Expr (ir) -> Expr ()
        [(cuote ,c) (parser-L10 (type c))]))
```

Ejercicio 3

Para implementar la Función J, tomamos reglas definidas para el algoritmo en el PDF de la práctica. Esta función trabaja sobre L10 y recibe una expresión y un contexto para devolver el tipo correspondiente a al expresión.

```
(define (J exp context)
  (nanopass-case (L10 Expr) exp
     [,x (let ([searched-variable (findf (lambda (y) (equal? (car y) x)
        ) context)])
           (if searched-variable
              (cdr searched-variable)
              (error "Variable not in context")))]
     [(const ,t ,c) t]
     [(list) 'List]
     [(begin ,e* ... ,e) (first (append
                                  (list (J e context))
                                  (foldl append '() (list (map (lambda
                                     (x) (J x context)) e*)))))]
     [(primapp ,pr ,e* ... ,e0)
      (cond
        [(arit? pr) (if (and (equal? (J e0 context) 'Int) (equal? (
           first (map (lambda (x) (J x context)) e*)) 'Int))
                                    'Int
                                    (error "Binary operators must have
                                       Integer parameters"))]
        [(lst? pr) (let ([type-lst-operators (J e0 context)])
                     (if (c-type? type-lst-operators)
                         (match pr
                           ['car (third type-lst-operators)]
                           ['cdr type-lst-operators]
                           ['length 'Int])
                         (error "Type does not match the list
                            operator")))])]
     [(if ,e0 ,e1 ,e2) (let ([t0 (J e0 context)]
                             [t1 (J e1 context)]
                             [t2 (J e2 context)])
                         (if (and (equal? 'Bool (J e0 context)) (unify
                             t1 t2))
                           t1
                           (error "Different type between expresions")
                             ))]
     [(lambda ([,x ,t]) ,body) (let* ([new-context (set-add context (
        cons x t))]
                                      [s (J body new-context)])
                                `(,t '    ,s))]
```

```
    [(let ([,x ,t ,e]) ,body) (let* ([new-context (set-add context (
       cons x t))]
                                     [t0 (J e context)]
                                     [s (J body new-context)])
                                (if (unify t t0)
                                  s
                                  (error "The type doesn't correspond
                                      to the value")))]
    [(letrec ([,x ,t ,e]) ,body) (let* ([new-context (set-add context
       (cons x t))]
                                        [t0 (J e new-context)]
                                        [s (J body new-context)])
                                (if (unify t t0)
                                  s
                                  (error "The type doesn't correspond
                                      to the value")))]
    [(letfun ([,x ,t ,e]) ,body) (let* ([new-context (set-add context
       (cons x t))]
                                        [t0 (J e context)]
                                        [s (J body new-context)])
                                (if (and (unify t t0) (equal? (cadr t
                                   ) '   ))
                                  s
                                  (error "The type doesn't correspond
                                      to the value")))]
    [(list ,e* ... ,e) (let* ([types (append (list (J e context))(map
       (lambda (x) (J x context)) e*))]
                              [t (part types)]
                              [eqt (map (lambda (x) (unify t x)) types
                                 )])
                        (if (foldr (lambda (x y) (and x y)) #t eqt)
                        `(List of ,t)
                        (error "Lists must be homogeneous")))]
    [(,e0 ,e1) (let* ([t0 (J e0 context)]
                      [R (third t0)]
                      [t1 (J e1 context)])
                 (if (unify t0 (t1 '    R))
                   R
                   (error "Types can not be inferred. Can not unify"))
                    )]
    [else (error "Types can not be inferred")]))
```

Ejercicio 4

Este ejercicio se encarga de sustituir las Lambdas por su tipo, es decir, $(T \to T)$, asi como tambien las listas $(List\ of\ T)$. Además, para este ejercicio podemos notar que sólo hay que construir el contexto con base en las funciones que pueden agregar nuevas variables a éste. Pero también hay que verificar las expresiones que pueden contener a las expresiones que pueden modificar el contexto y esas son: *begin, if y apliación de función.*

```
;Funcion auxiliar que obtiene el contexto general
(define (get-abs-context e)
    (nanopass-case (L10 Expr) e
                [(begin ,e* ... ,e) (append (map (lambda (x) (
                   get-abs-context x)) e*)
                                            (get-abs-context e))]
                [(if ,e0 ,e1 ,e2) (append (get-abs-context e0)
                                          (get-abs-context e1)
                                          (get-abs-context e2))]
                [(lambda ([,x ,t]) ,body) (set-add (get-abs-context
                   body)
                                                   (cons x t))]
                [(let ([,x ,t ,e]) ,body) (append (set-add (
                   get-abs-context body) (cons x t))
                                                  (get-abs-context e))]
                [(letrec ([,x ,t ,e]) ,body) (append (set-add (
                   get-abs-context body) (cons x t))
                                                    (get-abs-context e))]
                [(letfun ([,x ,t ,e]) ,body) (append (set-add (
                   get-abs-context body) (cons x t))
                                                    (get-abs-context e))]
                [(,e1 ,e2) (append (get-abs-context e1)
                                   (get-abs-context e2))]
                [else '()] ;Si no se ha agregado nada, entonces
                   lidiamos con un contexto vacio
                ))

;Solo se sustituyen expresiones que sean del tipo Lambda o List.
(define-pass type-infer : L10 (ir) -> L10 ()
  (definitions
   (define context (get-abs-context ir)))
  (Expr : Expr (ir) -> Expr()
    [(lambda ([,x ,t]) ,[body]) (if (or (equal? t 'List) (equal? t '
       Lambda))
                                    `(lambda ([,x ,(J body context)]) ,
                                       body)
                                    ir)]
    [(let ([,x ,t ,[e]]) ,[body]) (if (equal? t 'List)
```

```
                                        `(let ([,x ,(J e context) ,e]) ,
                                           body)
                                        ir)]
    [(letrec ([,x ,t ,[e]]) ,[body]) (if (or (equal? t 'List) (equal? t
        'Lambda))
                                        `(letrec ([,x ,(J e context) ,e
                                           ]) ,body)
                                        ir)]
    [(letfun ([,x ,t ,[e]]) ,[body]) (if (or (equal? t 'List) (equal? t
        'Lambda))
                                        `(letfun ([,x ,(J e context) ,e
                                           ]) ,body)
                                        ir)]))
```

Lenguajes Usados

Definición de L8

Este es el lenguaje que se toma como punto de partida para esta práctica, que es la salida de la práctica anterior. En ***nano-pass***:

```
(define-language L8
  (terminals
   (variable (x))
   (primitive (pr))
   (constant (c))
   (type (t)))
  (Expr (e body)
    x
    (cuote c)
    (begin e* ... e)
    (primapp pr e* ... e0)
    (if e0 e1 e2)
    (lambda ([x* t*] ...) body)       ;En L8 debe ser multiparametrico,
        sino 'curry' no tiene sentido
    ;(lambda ([x t]) body)
    (let ([x t e]) body)
    (letrec ([x t e]) body)
    (letfun ([x t e]) body)
    (list e* ...)
    (e0 e1 ...)))
```

Definición formal:

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

```
<expr> ::= <list>
         | <string>
         | (cuote <quotable>)
         | (<primitiva> <const> <const>*)
         | (begin <expr> <expr>*)
         | (if <expr> <expr>)
         | (if <expr> <expr> <expr>)
         | (lambda ([<var> <type>]) <expr>)
         | (let ([<var> <type> <expr>])  <expr>)
         | (letrec ([<var> <type> <expr>]) <expr>)
         | (letfun   ([<var> <type> <expr>]) <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

<list> ::= empty | (cons <const> <list>)

<string> ::= "" | " <char> <string> "

<char> ::=  a | b | c | ... | z | ... | @ | # | $ | % | & | ...

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

<type> ::= Bool | Int | Char | List | String

<quotable> ::= q
```

Definición de L9

En ***nano-pass***:

```
(define-language L9
```

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

Definición formal:

```
<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 | String
```

Definición de L10

En ***nano-pass***:

```
(define-language L10
```

```
  (extends L9)
  (Expr (e body)
        (- (cuote c))
        (+ (const t c))))
```

Definición formal:

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

<expr> ::= <const>
         | <var>
         | (const <type> <constant>)
         | (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 | String
```

Predicados

```

```

```
(define (variable? x) (and (symbol? x) (not (primitive? x)) (not (
   constant? x))))

(define (primitive? x) (memq x '(+ - * / length car cdr)))

(define (constant? x)
  (or (integer? x)
      (char? x)
      (boolean? x)))

(define (primitiva? pr) (memq pr '(+ - * / length car cdr)))

;encapsulamos los elementos que pueden ser quoteados
(define (quotable? q) (or (constant? q) (and (symbol? q) (not (
   primitive? q)))))

;; SISTEMA DE TIPOS

;; Int | Char | Bool | Lambda | List | (List of T) | (T     T)
(define (type? x) (or (b-type? x) (c-type? x)))
;;Verifica si es un tipo basico
(define (b-type? x) (memq x '(Bool Char Int List String Lambda)))
;; Verifica si es un tipo compuesto
(define (c-type? x) (if (list? x)
   (let* (
      [f (car x)]
      [s (cadr x)]
      [t (caddr x)])
   (or (and (equal? f 'List) (equal? s 'of) (type? t))
      (and (type? f) (equal? s '   ) (type? t))))
   #f))

;; Verifica si es una primitiva aritmtica
(define (arit? x) (memq x '(+ - * /)))

;; Verifica si es una primitiva de listas
(define (lst? x) (memq x '(length car cdr)))
```

Comentarios

- Nos pareció que esta práctica fue un tanto confusa. Nos costó un poco de trabajo asimilar como pasan las reglas, del algoritmo J, puestas en el PDF a código. Así como también probrar dos de esas reglas: *letfun* y *app*.

- Respecto a las reglas de la aplicación de función que vienen en el PDF, no nos quedó muy claro (a la hora de implementar) como es que se podía unificar t_0 con $t_1 \to R$.
- Como se puede observar en la definición de $L8$, éste fue modificado para que las lambdas no estuvieran ya currificadas desde el principio.
- Otra cosa que nos confunde es que en la aplicación suponemos (o al menos eso entendimos) que todas las funciones ya vienen con "$flechita$" y no es así.

.