Lenguajes de Programación

Proyecto final: Parte 2

Karla Ramírez Pulido Manuel Soto Romero Alejandro Hernández Mora
Silvia Díaz Gómez Pedro Ulises Cervantes González

Semestre 2021-2

Facultad de Ciencias, UNAM

Fecha de inicio: 23 de agosto de 2021

Fecha de entrega: 7 de septiembre de 2021

1. Objetivos

Implementar un verificador de tipos y definiciones de funciones recursivas para el lenguaje ***RCFWBAE-Typed***[1]. Para llevar a cabo esta tarea, se debe completar el cuerpo de las funciones faltantes dentro de los archivos ***grammars.rkt***, ***parser.rkt***, ***desugar.rkt***, ***interp.rkt*** y ***verifier.rkt***[2].

La gramática del lenguaje `CFWBAE` se presenta a continuación:

```
<expr>  ::= <id>
          | <num>
          | <bool>
          | <char>
          | <string>
          | <list>
          | {<op> <expr>+}
          | {if <expr> <expr> <expr>}
          | {if0 <expr> <expr> <expr>}
          | {cond {<expr> <expr>}+ {else <expr>}}
          | {with {{<id> : <type> <expr>}+} <expr>}
          | {with*{{<id> : <type> <expr>}+} <expr>}
          | {rec  {{<id> : <type> <expr>+} <expr>}
          | {fun  {{<id> : <type>}+} : <type> <expr>}
          | {<expr>}{(<expr>)*}
<id>    ::= a | b | c | ...
<num>   ::= 1 | 2 | 3 | ...
<bool>  ::= true | false
<char>  ::= 'a' | 'b' | 'c' | ...
<string>::= "a" | "aa" | "ab" | ...
<list>  ::= empty | {lst <expr>*}
```

[1] Que es el lenguaje resultante de la parte 1 del proyecto final, más algunas modificaciones mencionadas en este documento.

[2] Puedes encontrar el esqueleto de estos archivos en la especificación del proyecto en Classroom. También puedes tomar como base los archivos de la entrega de la parte 1 del proyecto final.

```
<op>    ::= + | - | * | / | modulo | expt | add1 | sub1
          | < | <= | = | > | >= | not | and | or | zero?
          | num? | bool? | char?  | string? | list?
          | cons | car | cdr | append | length | empty?
          | string-append | string-length
 <type> ::= number | boolean | char | string | (<type>+ -> <type>)
```

En *Racket*, se define la gramática anterior por medio del los siguientes tipos abstractos de datos.

```
;;Data-type que define al tipo de dato Type
(define-type Type
  [numberT]
  [booleanT]
  [charT]
  [stringT]
  [funT (params (listof Type?))])

;; Definición del tipo Type-Context
(define-type Type-Context
  [phi]
  [gamma (id symbol?) (tipo Type?) (rest Type-Context?)])

;; Definición del tipo Param
(define-type Param
  [param (param symbol?) (tipo Type?)])

;; Definición del tipo Binding
(define-type BindingS
  [bindingS (id symbol?) (type Type?) (value SRCFWBAE-Typed?)])

;; Definición del tipo Binding
(define-type Binding
  [binding (id symbol?) (value RCFWBAE-Typed?)])

;;Definición del tipo condition para la definición de cond.
(define-type Condition
  [condition (test-expr SRCFWBAE-Typed?) (then-expr SRCFWBAE-Typed?)]
  [else-cond (else-expr SRCFWBAE-Typed?)])

;;Definición del tipo Lista para la definición de listas.
(define-type ListDef
  [mtList]
  [aList (cabeza SRCFWBAE-Typed?) (cola ListDef?)])

;; Definición del tipo SRCFWBAE-TypedL
(define-type SRCFWBAE-Typed
  [idS    (i symbol?)]
  [numS   (n number?)]
```

```
[boolS  (b boolean?)]
[charS  (c char?)]
[stringS (s string?)]
[listS  (l ListDef?)]
[iF0    (condicion SRCFWBAE-Typed?) (then SRCFWBAE-Typed?) (else SRCFWBAE-Typed?)]
[iFS    (condicion SRCFWBAE-Typed?) (then SRCFWBAE-Typed?) (else SRCFWBAE-Typed?)]
[opS    (f procedure?) (args (listof SRCFWBAE-Typed?))]
[condS  (cases (listof Condition?))]
[withS  (bindings (listof bindingS?)) (body SRCFWBAE-Typed?)]
[withS* (bindings (listof bindingS?)) (body SRCFWBAE-Typed?)]
[recS   (bindings (listof bindingS?)) (body SRCFWBAE-Typed?)]
[funS   (params (listof param?)) (rType Type?) (body SRCFWBAE-Typed?)]
[appS   (fun SRCFWBAE-Typed?) (args (listof SRCFWBAE-Typed?))])
```

2. Ejercicios

El lenguaje debe mantener el mismo comportamiento del lenguaje ***CFWBAE*** del proyecto final (parte 1). Es decir, que las instrucciones existentes previamente no deben cambiar a excepción de la inclusión de algunos tipos[3].

2.1. Definiciones recursivas

Debes implementar definiciones recursivas utilizando cajas. Las cajas son una herramienta nativa de *Racket*, sirven como contenedor para almacenar valores que posteriormente podemos modificar. Las cajas tienen 3 operaciones básicas.

1. Nombrar un contenedor vacío.

```
(let ([fact (box 1)])
  ...
```

2. Mutar un contenedor existente.

```
(begin
  (let ([fact (box 1)])
    (set-box! fact
      (lambda(n)
        (if (zero? n)
          1
          (* n (fact (sub1 n))))))
    ...))
```

[3]Se explica detalladamente en la ***subsección*** *2.2*.

3. Consultar su contenido.

```
(let ([fact (box 1)])
  (begin
    (set-box! fact
      (lambda(n)
        (if (zero? n)
          1
          (* n (fact (sub1 n))))))
    ((unbox fact) 5 )))
```

Para poder interpretar correctamente el contenido de una caja que almacena una expresión recursiva es necesario modificar el ambiente de evaluación, para considerar identificadores cuyo valor sea el contenido de una caja. La modificación mencionada se logra a través los siguientes tipos abstractos de datos en *Racket*.

```
;; Función auxiliar para la definición de ambientes con cajas
;; boxed-RCFWBAE-Value?: any -> boolean
(define (boxed-RCFWBAE-Value? v)
  (or (box? v) (RCFWBAE-Value? (unbox v))))

;; Data-type que representa un ambiente de evaluación con cajas
(define-type Env
  [mtSub]
  [aSub   (name symbol?) (value RCFWBAE-Value?) (rest-env Env?)]
  [aRecSub (name symbol?) (value boxed-RCFWBAE-Value?) (rest-env Env?)])
```

Con esta modificación, la función `interp` recibe una expresión del lenguaje y en lugar de un caché de sustituciones `DefrdSub`, recibe un ambiente de evaluación `Env`. No olvides agregar el caso en el que recibimos un `aRecSub` en la función `lookup`.

Se recomienda revisar la *Nota de clase 6: Recursión*, que se encuentra en la página del curso.

1. (0.25 pts.) Modifica las funciones `parse` y `desugar` para que consideren las definiciones recursivas, de acuerdo con la nueva gramática del lenguaje *RCFWBAE-Typed*. Esto es, que tome una cadena de símbolos y lo transforme a un *AST* con azúcar sintáctica (`parse`) y después sin azúcar sintáctica (`desugar`).

   Por ejemplo:

```
>(parse '{rec ([fac : (number -> number)
                {fun {(n : number)} : (number -> number)
                  {if {zero? n}
                    1
                    {* n {fac ({- n 1})}}}}]
                [n : number 5])
              {fac (n)}})
```

```
(recS (list
  (bindingS 'fac (funT (list (numberT) (numberT)))
    (funS (list (param 'n (numberT))) (funT (list (numberT) (numberT)))
      (iFS (opS #<procedure:zero?> (list (idS 'n)))
        (numS 1)
        (opS #<procedure:*> (list (idS 'n) (appS (idS 'fac) (list (opS #<procedure:->
           (list (idS 'n) (numS 1)))))))))
  (bindingS 'n (numberT) (numS 5)))
    (appS (idS 'fac) (list (idS 'n))))

> (desugar (parse ...
(rec (list
  (binding 'fac (fun (list (param 'n (numberT)))
    (iF (op #<procedure:zero?> (list (id 'n)))
      (num 1)
      (op #<procedure:*> (list (id 'n) (app (id 'fac) (list (op #<procedure:-> (list
         (id 'n) (num 1)))))))))
  (binding 'n (num 5)))
    (app (id 'fac) (list (id 'n))))
```

2. (3 pts.) Modifica la función `interp` para que reciba un *AST* de un expresión recursiva y la evalúe.

```
>(interp (desugar (parse
    '{rec ([fac : (number -> number) {fun {(n : number)} : (number -> number) {if
       {zero? n} 1 {* n {fac ({- n 1})}}}}]
          [n : number 5])
      {fac (n)}})) (mtSub))
(numV 120)
```

2.2. Verificador de tipos

3. (0.25 pts.) Modifica la función `parse` para que reconozca los tipos en parámetros e identificadores, de acuerdo a la nueva gramática del lenguaje *RCFWBAE-Typed*.

4. (6.0 pts.) Completar el cuerpo de la función `(typeof sexpr context)`, que se encuentra dentro del archivo `verificador.rkt`.

```
;; typeof SCFBWAE-> Type-Context -> Type
(define (typeof sexpr context)...)
```

Dicha función debe tomar una expresión del tipo ***SCFBWAE-Typed***, verificar que no tenga errores de tipos y devolver el tipo del valor de retorno de cada expresión. En caso de que el programa no cumpla con alguna de las dos condiciones anteriores se debe mandar un error de tipos.

- **Operaciones aritméticas y lógicas (op):**
  Se debe verificar que el valor de cada parámetro recibido en la lista tenga el tipo correspondiente según el operador. Para los operadores `+`, `-`, `*`, `/`, `=`, `modulo`, `expt`, `add1`, `sub1`, `<`, `<=`, `>`, `>=`, `>`, `zero?` los parámetros deben tener tipo `numberT`. Para los operadores `and`, `or` y `not` los parámetros deben tener tipo `booleanT`. Para los predicados (excepto `zero?` y `empty?`), los parámetros no deben corresponder a un tipo en específico; esto es porque se utilizan precisamente para verificar si un argumento es de algún tipo en particular. Para los operadores `car`, `cdr`, `length` y `empty?`, los parámetros no deben tener un tipo en específico. Para los operadores `string-append` y `string-length`, los parámetros deben tener tipo `stringT`.
- **Condicionales (if, cond):**
  Se debe verificar que las condicionales en ambos casos sea algo de tipo `booleanT` o mandar un error de tipos en otro caso. Además en ambos casos los tipos de las expresiones a evaluar deben ser el mismo para todas las posibles expresiones que se van a ejecutar; es decir que en un `if` la condicional tiene un tipo `booleanT` y las expresiones `then-expression` y `else-expression` deben tener el mismo tipo. Análogamente esto debe suceder con las expresiones de tipo `cond`, conde las `test-expression` tengan tipo `booleanT` y tanto las `then-expression` como las `else-expression` tengan el mismo tipo. Si las expresiones no tienen el mismo tipo se debe mandar un error de tipos[4].
- **Funciones (fun):**
  Se debe guardar dentro del contexto el tipo de cada uno de los parámetros recibidos en la función, para que al aplicar la función, se puedan obtener los tipos de cada uno de éstos. La función tendrá tipo `funT` $(a_1, a_2, ..., r)$, donde cada a_i representa el tipo del parámetro recibido en la posición i, y r es el tipo del valor de retorno de la función.
- **Asignaciones locales simples, anidadas (`with`, `with*`) y definiciones recursivas(`rec`):**
  Se debe verificar que los parámetros recibidos se evalúen al mismo tipo al que fueron declarados.
- **Aplicaciones de función (app):**
  Verificar que la función tenga el tipo `funT` y que los parámetros de la función tengan el tipo correspondiente a la declaración de la función.

Debes integrar el verificador de tipos junto con el intérprete, de manera que después de construir el árbol de sintaxis abstracta con azúcar sintáctica (es decir, después de aplicar la función `parse` a la entrada) se ejecute el verificador de tipos. Después de hacer la verificación de tipos, si no hubo ningún error de tipos debes quitar el azúcar sintáctica e interpretar la expresión; devolviendo el valor final de la interpretación.

[4] Esto es una decisión de diseño de lenguaje, que tiene sus diferentes implicaciones, tanto ventajas como desventajas. Por esta razón, no todos los lenguajes funcionan de esta manera.

2.3. Puntos extra

1. **1 punto extra:** Modifica la gramática del lenguaje, las sintaxis abstractas y el verificador de tipos, para que las listas sean homogéneas. Es decir, que debes considerar que una lista contiene elementos de un sólo tipo, por lo que también debes verificar que todos los elementos de ésta sean de dicho tipo.

2. **1 punto extra:** Modifica el lenguaje para que sea de evaluación perezosa. Es decir, que debes incluir la implementación de putos estrictos en la evaluación de las expresiones. Agrega al menos un par de ejemplos donde construyas una lista infinita y operes con ella.

La implementación de los puntos extra son sobre el total del porcentaje del proyecto final y depende totalmente de tu equipo. Tampoco debe de intervenir con el resto de los requerimientos del proyecto, de manera que la implementación de los puntos extra no deben afectar el desempeño de las pruebas unitarias proporcionadas, ni cambiar de ninguna manera la sintaxis del lenguaje (excepto claro, en el caso de las listas heterogéneas). Para el caso de las listas heterogéneas, debes incluir (y/o cambiar) las pruebas unitarias para que considere los tipos, de acuerdo con tu nueva gramática; ningún otro cambio es permitido en la gramática del lenguaje. Si es necesario explicar algo sobre la implementación debes incluirlo en un documento de tipo **PDF** elaborado en LaTeX. Puedes apoyarte en las notas de clase y el libro en el que está basado el curso.

3. Requerimientos

Se deben subir los archivos requeridos a la plataforma Google Classroom antes de las 23:59 hrs. del día de la fecha de entrega, conforme lo como lo indican los lineamientos de entrega. No olvides incluir el archivo `ReadMe.txt` con los datos de tus compañeros de equipo. Sólo es necesario que una persona suba la práctica, los demás compañeros deberán subir como tarea el contenido del archivo `ReadMe.txt`, de manera que se pueda ver el equipo que integran desde la plataforma Google Classroom.

El orden en el que aparezcan las funciones en los archivos solicitados, debe ser el orden especificado en este archivo PDF, de lo contrario podrán penalizarse algunos puntos.

La idea es que reutilices funciones de los ejercicios que ya se programaron anteriormente.

¡Que tengas éxito en tu proyecto!.