

Fundamentos de lenguajes de programación

Semántica de los Conceptos Fundamentales de Lenguajes de Programación

carlos.andres.delgado@correounivalle.edu.co

Carlos Andrés Delgado S. Carlos Alberto Ramírez

Facultad de Ingeniería. Universidad del Valle

Noviembre de 2016

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

1 Tipos y Programación

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

1 Tipos y Programación

2 Chequeo de tipos

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

1 Tipos y Programación

2 Chequeo de tipos

- Uno de los conceptos fundamentales de cualquier lenguaje de programación es el concepto de **tipo**.
- Los tipos definen las estructuras de almacenamiento disponibles para representar información; además, el tipo determina cómo interpreta el compilador el contenido de la memoria.
- Un error de tipo se produce cuando se aplica una operación a un dato con características diferentes a las esperadas.

Segun la forma en que los lenguajes se comportan frente a los diversos tipos, se clasifican de dos formas:

- **Lenguajes fuertemente tipados:** requieren definiciones de los tipos y no permiten errores en los tipos. Ej: Pascal.
- **Lenguajes debilmente tipados:** tienen cierta flexibilidad para errores de tipos (pero sí existen los errores de tipos). Ej: Scheme.

- El análisis de tipos es un paso del procesamiento de un lenguaje de programación.
- El análisis de tipos se realiza para evitar que se presenten errores de tipos en tiempo de ejecución.

Para realizar el análisis de tipos se tienen en cuenta los siguientes aspectos:

- 1 Se define un conjunto de tipos para el lenguaje y lo que significa “un valor expresado v es de tipo t ”.
- 2 El analizador asigna un tipo para cada expresión en el programa.
 - si a una expresión e se asigna un tipo t , entonces siempre que e es interpretada, su valor sera de tipo t .
 - Esta propiedad, hace que un sistema de tipos sea adecuado (*sound*).

- 3 El analizador debe inspeccionar también cada invocación de una operación en el programa, con el fin de verificar que los operandos sean del tipo adecuado.

Si no se puede determinar si los argumentos son del tipo apropiado, entonces se dice que esta invocación posiblemente genera un error de tipo.

- 4 Si se detectan errores de tipo, el analizador puede tomar algunas acciones, las cuales son, normalmente, parte del diseño del lenguaje:
 - puede rehusarse a ejecutar el programa,
 - aplicar medidas correctivas.

- En el diseño del lenguaje, se definen cuáles tipos habrán, si un valor puede tener más de un tipo, y si los tipos pueden ser determinados en tiempo de ejecución.
- En algunos lenguajes, en cada ejecución, los valores incluyen una etiqueta para indicar su tipo.
- Esto es llamado tipado *dinámico*.

- Scheme es un lenguaje tipado dinámico: las etiquetas son revisadas por `number?`, `string?`, etc.
- Las desventajas del tipado dinámico son:
 - Insertar y revisar las etiquetas puede adicionar tiempo en ejecución.
 - No soporta abstracción de datos.

Lenguajes no tipados

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

- En otros lenguajes, los valores pueden representar, por ejemplo, un entero y un caracter.
- Dichos lenguajes tienen un modelo de ejecución *no tipado*.
- Los lenguajes con modelos de ejecución no tipado normalmente no detectan operaciones inapropiadas en tiempo de ejecución.
- Si las operaciones son aplicadas con datos incorrectos, los resultados no son especificados.

Lenguajes no tipados

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

- En dichos lenguajes se podría hacer la multiplicación de dos caracteres.
- El resultado de esa operación será lo que el hardware haga con la representación de los caracteres.
- Esto es llamado un diseño *político*.

- Los lenguajes tipados evitan las dificultades anteriores analizando el programa antes de la ejecución para determinar cualquier llamado particular en el programa que pueda resultar en una operación no apropiada en tiempo de ejecución.
- Esto es llamado *chequeo de tipos estático*.
- Si un error es detectado, el analizador podría producir una advertencia, insertar un código de revisión en tiempo de ejecución o rechazar el programa.

Los tipos para el lenguaje que se está implementando tienen una la siguiente estructura:

$\langle \text{tipo-exp} \rangle ::= \text{int}$
 $\text{int-type-exp } ()$

$\langle \text{tipo-exp} \rangle ::= \text{bool}$
 $\text{bool-type-exp } ()$

$\langle \text{tipo-exp} \rangle ::= (\{ \langle \text{tipo-exp} \rangle \}^{*(*)} \rightarrow \langle \text{tipo-exp} \rangle)$
 $\text{proc-type-exp } (\text{arg-texps result-texp})$

- Los tipos del lenguaje incluyen tipos base para enteros, booleanos y tipos para procedimientos.
- El tipo de un procedimiento consiste de los tipos de sus argumentos (separados por `*`) y el tipo de su resultado.
- Cuando se permite explicitar los tipos en los programas, se suele hacer con *expresiones de tipos*.

La propiedad: “un valor expresado v es de tipo t ” se define inductivamente sobre t de la siguiente manera:

- Un valor expresado es de tipo `int` sii es un entero.
- Un valor expresado es de tipo `bool` sii es un booleano.
- Un valor expresado es de tipo $(t_1 * \dots * t_n \rightarrow t)$ sii es un `ProcVal` que espera exáctamente n argumentos, y cuando tiene n argumentos de tipos t_1, \dots, t_n , éste retorna un valor de tipo t .

- Así mismo, en éste lenguaje, cada valor expresado tiene como máximo un tipo.
- Sin embargo, no siempre es posible determinar el tipo de un valor en tiempo de ejecución, ya que no siempre es posible determinar el tipo del valor retornado por un procedimiento.

Se pueden usar los tipos definidos para describir valores de Scheme. Por ejemplo:

```
(even? g) = boo
```

```
(int -> bool)
```

tipo de even?

```
(int * int -> int)
```

tipo de +

```
(int -> (int -> int))
```

tipo de (lambda (x)

```
(lambda (y) (+ x y)))
```

```
((int -> int) * int -> bool)
```

tipo de (lambda (f x)

```
(even? (f (+ x 1))))
```

donde esos son los tipos de los *valores* de las expresiones y no de las expresiones mismas; hasta ahora no se ha dicho lo que significa que una expresión tenga un tipo.

Cual es el tipo de dato de las siguientes expresiones del Lenguaje del curso?

1 $-(32, 26) \rightarrow \text{int}$

2 $\text{proc}(x) (x \ 26) \rightarrow (int \rightarrow t) \rightarrow t$

3 $\text{proc}(x) \text{ if } x \text{ then } 1 \text{ else } 3 \rightarrow (bool \rightarrow int)$

4 $\text{proc}(f, x) \text{ if } (f \ x) \text{ then } 1 \text{ else } 2$

5 $\text{proc}(x) \text{ proc}(y) \text{ if } y \text{ then } x \text{ else } 3$

$\rightarrow ((tx \rightarrow bool) * tx \rightarrow int)$
 $\rightarrow (int \rightarrow (bool \rightarrow int))$

- El lenguaje que se está diseñando, será *fuerte y estáticamente tipado*, lo que significa que ningún programa que pasa el chequeo de tipos tendrá un error de tipos.
- Un error de tipo será:
 - 1 un intento de aplicar un entero o un booleano a un argumento,
 - 2 un intento de aplicar un procedimiento o primitiva a un número errado de argumentos,
 - 3 un intento de aplicar una primitiva que espera un entero a un no-entero, ó
 - 4 un intento de usar un no-booleano como la prueba de una expresión condicional.
- Los errores como una división por cero, no serán considerados como errores de tipo.

1. `proc (x) - (x, 3)` $(int \rightarrow int)$
2. `proc (f) proc (x) - ((f x), 1)` $((\lambda x \rightarrow int) \rightarrow (\lambda x \rightarrow int))$
3. `proc (x) x` $(\lambda x \rightarrow \lambda x)$
4. `proc (x) proc (y) (x y).` $((\lambda y \rightarrow \lambda) \rightarrow (\lambda y \rightarrow \lambda))$
5. `proc (x) (x 3)`
6. `proc (x) (x x)`
7. `proc (x) if x then 88 else 99`
8. `proc (x) proc (y) if x then y else 99`
9. `(proc (p) if p then 88 else 99
33)`
10. `(proc (p) if p then 88 else 99
proc (z) z)`
11. `proc (f)
proc (g)
proc (p)
proc (x) if (p (f x)) then (g 1) else -((f x), 1)`

1. `proc (x) -(x,3)`
2. `proc (f) proc (x) -((f x), 1)`
3. `proc (x) x`
4. `proc (x) proc (y) (x y).`
5. `proc (x) (x 3)`
6. `proc (x) (x x)`
7. `proc (x) if x then 88 else 99`
8. `proc (x) proc (y) if x then y else 99`
9. `(proc (p) if p then 88 else 99 33)`
10. `(proc (p) if p then 88 else 99 proc (z) z)`
11. `proc (f)`
`proc (g)`
`proc (p)`
`proc (x) if (p (f x)) then (g 1) else -((f x),1)`

t_x
 $((int \rightarrow t) \rightarrow t)$

(\dots)

$(bool \rightarrow int)$

$(bool \rightarrow (int \rightarrow int))$

int

error de tipos $(t_x \rightarrow t_x) \neq bool$

$\rightarrow ((t_x \rightarrow int) \rightarrow (int \rightarrow int) \rightarrow (int \rightarrow bool) \rightarrow (t_x \rightarrow int))$

f g p x

Tipos de las expresiones

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

- Se definirá un procedimiento `type-of-expression` que dado una expresión *exp* y un *ambiente de tipos* *tenv* (que asocia cada variable con un tipo), asigna a *exp* un tipo *t* con la propiedad que:
Siempre que *exp* es ejecutado en un ambiente en el cual cada variable tiene el tipo especificado para él por *tenv*, el valor resultante tendrá tipo *t*.
- El análisis estará basado en el principio de que si se conoce los tipos de los valores de cada una de las variables en una expresión, se puede deducir el tipo del valor de la expresión.

Tipos de las expresiones

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

EL procedimiento `type-of-expression` se comportará de la siguiente manera:

- Si la expresión es un número, entonces el resultado será siempre un entero y si es una expresión booleana (`true` o `false`) el resultado será booleano.
- Si la expresión es una variable, entonces el resultado es del tipo especificado por *tenv*.

Tipos de las expresiones

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

```
(type-of-expression  $\ll n \gg$  tenv) = int
```

```
(type-of-expression  $\ll true \gg$  tenv) = bool
```

```
(type-of-expression  $\ll false \gg$  tenv) = bool
```

```
(type-of-expression  $\ll id \gg$  tenv) = (apply-env tenv id)
```

Tipos de las expresiones

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

Una expresión de la forma:

Si **<condicion1>**

y **<condicion2>**

y **<condicion3>**

...

y **<condicionN>**

entonces

<consecuencia>

Se conoce como regla de tipado o especificación condicional. En adelante se omitirán los “Si”, “y” y “entonces”, ya que ellos están implícitos en el formato de la especificación.

Tipos de las expresiones

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

- Si la expresión es una aplicación (de la forma $(rator\ rand_1 \dots rand_n)$), el tipo del resultado se infiere del tipo del operador y los tipos de los operandos.
- El tipo del operador debe ser (obligatoriamente) un tipo procedimiento.
- Si el tipo del operador es $(t_1 * t_2 * \dots * t_n \Rightarrow t)$, entonces debe haber exactamente n operandos, y el tipo del i -ésimo operando debe ser t_i . *Evaluación*
- Si las condiciones se mantienen, entonces el resultado de la aplicación será el tipo resultado del procedimiento, *t*.

Tipos de las expresiones

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

Regla de tipamiento para aplicación de procedimientos o primitivas:

$$(\text{type-of-expression } \ll rator \gg \text{ tenv}) = (t_1 * t_2 * \dots * t_n \rightarrow t)$$

$$(\text{type-of-expression } \ll rand_1 \gg \text{ tenv}) = t_1$$

$$(\text{type-of-expression } \ll rand_2 \gg \text{ tenv}) = t_2$$

...

$$(\text{type-of-expression } \ll rand_n \gg \text{ tenv}) = t_n$$

$$(\text{type-of-expression } \ll (rator \ rand_1 \ rand_2 \ \dots \ rand_n) \gg \text{ tenv}) \\ = \underline{t}$$

Tipos de las expresiones

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

La regla de tipamiento para las expresiones condicionales es:

if · exp

$$(\text{type-of-expression } \ll \text{test} - \text{exp} \gg \text{ tenv}) = \text{bool}$$

$$(\text{type-of-expression } \ll \text{true} - \text{exp} \gg \text{ tenv}) = t$$

$$(\text{type-of-expression } \ll \text{false} - \text{exp} \gg \text{ tenv}) = t$$

$$(\text{type-of-expression } \ll \text{if } \text{test} - \text{exp} \text{ then } \text{true} - \text{exp} \text{ else } \text{false} - \text{exp} \gg \text{ tenv}) = t$$

Tipos de las expresiones

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

Para el caso de una expresión que crea un procedimiento ($\text{proc } (x_1, \dots, x_n) \text{ exp}$), se dice que el procedimiento resultante tiene tipo $(t_1 * t_2 * \dots * t_n \rightarrow t)$, si:

- Cada x_i está asociado con el tipo t_i .
- Tiene exactamente n argumentos.
- Se puede mostrar que cuando el cuerpo exp del procedimiento sea ejecutado, producirá un valor de tipo t .

Tipos de las expresiones

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

La regla de tipado para creación de procedimientos es:

$$\frac{(\text{type-of-expression } \ll exp \gg [x_1=t_1, \dots, x_n = t_n] \text{ tenv}) = t}{(\text{type-of-expression } \ll \text{proc } (x_1, \dots, x_n) \text{ exp} \gg \text{ tenv}) = ((t_1 * t_2 * \dots * t_n \rightarrow t))}$$

Tipos de las expresiones

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

- La anterior regla lleva a un problema:

Si se trata de encontrar el tipo de una expresión `proc`,
¿cómo se encuentran los tipos t_1, \dots, t_n de las variables
ligadas?.

- La respuesta es: No hay donde encontrarlos.
- Existen dos estrategias para esta situación:
 - *chequeo de tipos*: se requiere que el programador brinde información sobre los tipos de las variables.
 - *inferencia de tipos*: inferencia de tipos: el analizador de tipos, infiere el tipos de las variables de la forma en que dichas variables son usadas en el programa.

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

1 Tipos y Programación

2 Chequeo de tipos

Chequeo de tipos

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

- En un lenguaje con chequeo de tipos, se requiere que el programador incluya los tipos de todas las variables ligadas.
- Para hacer chequeo de tipos en el lenguaje del curso, se necesita cambiar la forma en que se escriben procedimientos (`proc-exp`) y recursion (`letrec-exp`).

Chequeo de tipos

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

La gramática para expresiones con tipo será ahora:

$$\langle \text{expresión} \rangle ::= \text{proc } (\{ \langle \text{tipo-exp} \rangle \langle \text{identificador} \rangle \}^{*(,)}) \langle \text{expresión} \rangle$$

`proc-exp (arg-texps ids body)`

$$\langle \text{expresión} \rangle ::= \text{letrec}$$

$\{ \langle \text{type-exp} \rangle \langle \text{identificador} \rangle$
 $\{ \{ \langle \text{type-exp} \rangle \langle \text{identificador} \rangle \}^{*(,)} = \langle \text{expresión} \rangle$
 $\text{in } \langle \text{expresión} \rangle$

`letrec-exp
(result-texps proc-names arg-texpss
idss bodies letrec-body)`

Chequeo de tipos

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

Se deben añadir las siguientes producciones a la especificación de la gramática:

```
( expression
  ("proc" "(" (separated-list type-exp identifier "
    ,") ")"
    expression)
  proc-exp)
(expression
  ("letrec"
    (arbno type-exp identifier
      "(" (separated-list type-exp identifier ",") "
        )"
        "=" expression) "in" expression)
  letrec-exp)
```

Chequeo de tipos

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

Así mismo, se añaden las producciones correspondientes a las expresiones `type-exp` a la especificación de la gramática:

```
(type-exp ("int") int-type-exp)
(type-exp ("bool") bool-type-exp)
(type-exp
  ("(" (separated-list type-exp "*" ) "->" type-exp
    ")")
  proc-type-exp)
```

De esta manera se pueden escribir programas como:

```
letrec
  int fact (int x) =
    if zero?(x) then 1 else *(x,(fact sub1(x)))
in
  (fact 5)
```

Aquí, el tipo resultado de `fact` es `int`, pero el tipo de `fact` como tal es `(int -> int)`.

Se pueden escribir programas como:

```
let
  sum = proc (bool b, int x, int y)
           if b then +(x, y) else -(x, y)
  m = 4
  n = 3
in
  (sum true m n)
```

Aquí, el tipo resultado de `sum` es `int`, pero el tipo de `sum` como tal es `(bool*int*int -> int)`.

- Ya se ha definido qué es una *expresión de tipo*.
- Una expresión de tipo denota un *tipo*. Pero, qué es un tipo?.

Chequeo de tipos

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

- Ya se ha definido qué es una *expresión de tipo*.
- Una expresión de tipo denota un *tipo*. Pero, qué es un tipo?.
- Ahora se definirán los valores (tipos) que pueden denotar las expresiones de tipo.

- Ya se ha definido qué es una *expresión de tipo*.
- Una expresión de tipo denota un *tipo*. Pero, qué es un tipo?.
- Ahora se definirán los valores (tipos) que pueden denotar las expresiones de tipo.
- Para ello se definirá el tipo abstracto de datos “tipo”. Un “tipo” es o un *tipo atómico* con un nombre, o un *tipo procedimiento*, con una lista de argumentos con tipo y un tipo resultado.

El tipo abstracto de datos tendrá la siguiente definición:

```
(define-datatype type type?
  (atomic-type
    (name symbol?))
  (proc-type
    (arg-types (list-of type?))
    (result-type type?)))

(define int-type (atomic-type 'int))
(define bool-type (atomic-type 'bool))
```

Las constantes `int-type` y `bool-type` son abreviaciones.

- Ahora, se debe definir el procedimiento `expand-type-expression` que calcula el tipo denotado por una expresión de tipo.
- El revisor (*checker*) llamará a `expand-type-expression` siempre que necesite utilizar el tipo denotado por una expresión de tipo.

El procedimiento `expand-type-expression` estará definido así:

```
(define expand-type-expression
  (lambda (texp)
    (cases type-exp texp
      (int-type-exp () int-type)
      (bool-type-exp () bool-type)
      (proc-type-exp (arg-texps result-texp)
        (proc-type
         (expand-type-expressions arg-texps)
         (expand-type-expression result-texp))))))

(define expand-type-expressions
  (lambda (texps)
    (map expand-type-expression texps)))
```

- El revisor (*checker*) se implementará como un procedimiento que recibe una expresión y un ambiente de tipos y retorna el tipo correspondiente a ella o un error.
- El revisor corresponde al procedimiento `type-of-expression` y tendrá cláusulas que implementan las reglas de producción para las expresiones con tipo.

- Las primeras cláusulas del procedimiento `type-of-expression` tienen que ver con literales y variables.
- Si la expresión corresponde a una expresión de un literal o un booleano se retorna el tipo `int-type` o `bool-type` respectivamente.
- Si la expresión corresponde a una variable, se retorna el tipo de la variable en el ambiente de tipos en el que se chequea la expresión.
- Se usa el procedimiento `apply-tenv`, similar a `apply-env` pero con distinto mensaje de error.

El procedimiento type-of-expression:

```
(define type-of-expression
  (lambda (exp tenv)
    (cases expression exp
      (lit-exp (number) int-type)
      (true-exp () bool-type)
      (false-exp () bool-type)
      (var-exp (id) (apply-tenv tenv id))
      ...)))
```

- La siguiente cláusula del procedimiento `type-of-expression` implementa la regla para expresiones condicionales (`if-exp`).

$$(\text{type-of-expression } \langle\langle \text{test} - \text{exp} \rangle\rangle \text{ tenv}) = \text{bool}$$
$$(\text{type-of-expression } \langle\langle \text{true} - \text{exp} \rangle\rangle \text{ tenv}) = t$$
$$(\text{type-of-expression } \langle\langle \text{false} - \text{exp} \rangle\rangle \text{ tenv}) = t$$

$$(\text{type-of-expression}$$
$$\langle\langle \text{if test} - \text{exp then true} - \text{exp else false} - \text{exp} \rangle\rangle \text{ tenv}) =$$

Ahora, el procedimiento `type-of-expression`:

```
(define type-of-expression
  (lambda (exp tenv)
    (cases expression exp
      ...
      (if-exp (test-exp true-exp false-exp)
        (let ((test-type (type-of-expression test-exp
          tenv))
              (false-type (type-of-expression false-exp
          tenv))
              (true-type (type-of-expression true-exp
          tenv))))
          (check-equal-type! test-type bool-type test-exp)
          (check-equal-type! true-type false-type exp)
          true-type))
      ...)))
```

El procedimiento `check-equal-type!` verifica si sus dos primeros argumentos tienen igual tipo. El tercer argumento es usado para el reporte de errores.

```
(define check-equal-type!  
  (lambda (t1 t2 exp)  
    (if (not (equal? t1 t2))  
        (eopl:error 'check-equal-type!  
          "Types didn't match: ~s != ~s in ~%~s"  
            (type-to-external-form t1)  
            (type-to-external-form t2)  
            exp))))
```

El procedimiento `check-equal-type`! usa el procedimiento `type-to-external-form` para convertir un tipo en una estructura de lista, fácil de leer, por ejemplo `(int * (int -> bool) -> int)`.

```
(define type-to-external-form
  (lambda (ty)
    (cases type ty
      (atomic-type (name) name)
      (proc-type (arg-types result-type)
        (append
          (arg-types-to-external-form arg-types)
          '(->)
          (list (type-to-external-form result-type)))))))
```

Chequeo de tipos

Checker

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

```
(define arg-types-to-external-form
  (lambda (types)
    (if (null? types)
        '()
        (if (null? (cdr types))
            (list (type-to-external-form (car types)))
            (cons
              (type-to-external-form (car types))
              (cons '*
                    (arg-types-to-external-form (cdr types))))))))
```

- Retomamos, la regla de tipamiento para creación de procedimientos:

$$\frac{(\text{type-of-expression } \ll exp \gg [x_1=t_1, \dots, x_n = t_n] \text{ tenv}) = t}{(\text{type-of-expression } \ll \text{proc } (x_1, \dots, x_n) \text{ exp} \gg \text{ tenv}) = ((t_1 * t_2 * \dots * t_n \rightarrow t))}$$

- Esta regla es implementada con ayuda del procedimiento `type-of-proc-exp`.

- El procedimiento `type-of-proc-exp` recibe una expresión `proc ($t_1\ x_1, t_2\ x_2, \dots, t_n\ x_n$) exp` y convierte los tipos t_1, \dots, t_n en una lista de tipos `arg-types`.
- Posteriormente, revisa el cuerpo en el ambiente especificado y liga el tipo resultado a `result-type`.
- Finalmente, construye un procedimiento con tipo, con las partes apropiadas.

La definición del procedimiento `type-of-proc-exp` es la siguiente:

```
(define type-of-proc-exp
  (lambda (texps ids body tenv)
    (let ((arg-types (expand-type-expressions texps)))
      (let ((result-type
              (type-of-expression body
                (extend-tenv ids arg-types tenv))))
        (proc-type arg-types result-type)))))
```

Ahora, el procedimiento `type-of-expression`:

```
(define type-of-expression
  (lambda (exp tenv)
    (cases expression exp
      ...
      (proc-exp (texps ids body)
        (type-of-proc-exp texps ids body tenv))
      ...)))
```

- Para aplicaciones de primitivas o de procedimientos teníamos la siguiente regla de tipamiento:

$$(\text{type-of-expression } \ll rator \gg \text{ tenv}) = (t_1 * t_2 * \dots * t_n \rightarrow t)$$
$$(\text{type-of-expression } \ll rand_1 \gg \text{ tenv}) = t_1$$
$$(\text{type-of-expression } \ll rand_2 \gg \text{ tenv}) = t_2$$

...

$$(\text{type-of-expression } \ll rand_n \gg \text{ tenv}) = t_n$$

$$(\text{type-of-expression } \ll (rator \ rand_1 \ rand_2 \ \dots \ rand_n) \gg \text{ tenv}) =$$

- `type-of-expression` encuentra el tipo del operador y luego llama a un procedimiento auxiliar `type-of-application` para aplicar la regla de tipamiento.
- El procedimiento `type-of-application` primero revisa si el tipo del operador es un tipo procedimiento.
- Luego revisa si el número de argumentos esperados por el procedimientos es igual al número dado y después, en el ciclo `for-each`, revisa si el tipo de cada argumento esperado es igual al tipo del operando correspondiente.
- Si la revisión tiene éxito, entonces el tipo de la aplicación es le tipo resultado del procedimiento.

La definición del procedimiento type-of-application es la siguiente:

```
(define type-of-application
  (lambda (rator-type rand-types rator rand exp)
    (cases type rator-type
      (proc-type (arg-types result-type)
        (if (= (length arg-types) (length rand-types))
            (begin
              (for-each
               check-equal-type!
               rand-types arg-types rand)
              result-type)
            (eopl:error 'type-of-expression
              (string-append
               "Wrong number of arguments in expression ~s
               : "
               "~%expected ~s~%got ~s")
               exp
               (map type-to-external-form arg-types)
               (map type-to-external-form rand-types))))))
    (else
     (eopl:error 'type-of-expression
```

- Para determinar el tipo del operador cuando se trata de aplicación de primitivas se utiliza el procedimiento auxiliar `type-of-primitive`.
- Este procedimiento recibe un valor del tipo de dato primitiva y crea un tipo `proc-type` de acuerdo a ese valor.

La definición del procedimiento `type-of-primitive` es la siguiente:

```
(define type-of-primitive
  (lambda (prim)
    (cases primitive prim
      (add-prim ()
        (proc-type (list int-type int-type) int-type))
      (incr-prim ()
        (proc-type (list int-type) int-type))
      (zero-test-prim ()
        (proc-type (list int-type) bool-type))
      ...)))
```

Ahora, se añaden las cláusulas correspondientes al procedimiento `type-of-expression`:

```
...  
  (primapp-exp (prim rand)  
    (type-of-application  
      (type-of-primitive prim)  
      (types-of-expressions rand tenv)  
      prim rand exp))  
  (app-exp (rator rand)  
    (type-of-application  
      (type-of-expression rator tenv)  
      (types-of-expressions rand tenv)  
      rator rand exp))  
...
```


El procedimiento `types-of-expressions`:

```
(define types-of-expressions
  (lambda (rands tenv)
    (map (lambda (exp) (type-of-expression exp tenv))
         rands)))
```

- La regla de tipamiento para expresiones `let` es:

$$(\text{type-of-expression } \ll e_1 \gg \text{ tenv}) = t_1$$
$$(\text{type-of-expression } \ll e_2 \gg \text{ tenv}) = t_2$$
$$\dots$$
$$(\text{type-of-expression } \ll e_n \gg \text{ tenv}) = t_n$$
$$\frac{(\text{type-of-expression } \ll body \gg [x_1=t_1, \dots, x_n = t_n] \text{ tenv}) = t}{(\text{type-of-expression } \ll \text{let } x_1 = e_1 \dots x_n = e_n \text{ in } body \gg \text{ tenv})}$$

- Esta regla es implementada con ayuda del procedimiento `type-of-let-exp`.

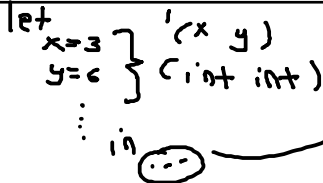
- El procedimiento `type-of-let-exp` encuentra el tipo de las expresiones en las declaraciones del `let` y extiende el ambiente de tipos con las variables en el `let` y los tipos encontrados.
- Luego, `type-of-let-exp` determina el tipo de la expresión correspondiente a su cuerpo en el ambiente de tipos extendido.

La definición del procedimiento `type-of-let-exp` es la siguiente:

```
(define type-of-let-exp
  (lambda (ids rands body tenv)
    (let ((tenv-for-body
          (extend-tenv
            ids
            (types-of-expressions rands tenv)
            tenv)))
      (type-of-expression body tenv-for-body))))
```

Handwritten example of a let expression and its type signature:

```
let
  x=3 } '(x y)
  y=6 } (int int)
  ...
  in ...
```



Así mismo, se añade la cláusula respectiva a la definición del procedimiento `type-of-expression`:

```
(define type-of-expression
  (lambda (exp tenv)
    (cases expression exp
      ...
      (let-exp (ids rands body)
        (type-of-let-exp ids rands body tenv))
      ...)))
```

- Una expresión letrec se ve de esta manera:

letrec
 $\underbrace{t_1}_{p_1} (t_{11} \ x_{11}, \dots, t_{1n_1} \ x_{1n_1}) = e_1$
 $\underbrace{t_2}_{p_2} (\underbrace{t_{21}}_{x_{21}}, \dots, t_{2n_2} \ x_{2n_2}) = e_2$
 ...
 in *body*

- En la cual se declaran un conjunto de procedimientos p_1, p_2, \dots con cuerpos e_1, e_2, \dots
- El tipo de p_i será $(t_{i1} * t_{i2} * \dots * t_{in} \rightarrow t_i)$.

- El cuerpo del `letrec` y cada uno de los cuerpos de los procedimientos deben ser revisados en un ambiente donde cada variable tenga el tipo correcto.
- Como el procedimiento p_i tiene tipo $(t_{i1} * t_{i2} * \dots * t_{in} \rightarrow t_i)$, el cuerpo del `letrec` debe ser revisado en el ambiente:

$$\text{tenv}_{body} = [\begin{array}{l} p_1 = (t_{11} * t_{12} \dots \rightarrow t_1), \\ p_2 = (t_{21} * t_{22} \dots \rightarrow t_2), \\ \dots \\ \end{array}] \text{tenv}$$

- Cada cuerpo de los procedimientos, se debe revisar en el ambiente:

$$\text{tenv}_i = [x_{i1} = t_{i1}, x_{i2} = t_{i2}, \dots] \text{tenv}_{body}$$

- Luego, la regla de tipamiento para expresiones letrec es:

$$\begin{array}{l}
 (\text{type-of-expression } \ll e_1 \gg \text{tenv}_1) = t_1 \\
 (\text{type-of-expression } \ll e_2 \gg \text{tenv}_2) = t_2 \\
 \dots \\
 (\text{type-of-expression } \ll e_n \gg \text{tenv}_n) = t_n \\
 (\text{type-of-expression } \ll \text{body} \gg \text{tenv}_{\text{body}}) = t \\
 \hline
 (\text{type-of-expression } \\
 \quad \ll \text{letrec} \\
 \quad \quad t_1 \ p_1 \ (t_{11} \ x_{11}, \dots, t_{1n_1} \ x_{1n_1}) = e_1 \\
 \quad \quad t_2 \ p_2 \ (t_{21} \ x_{21}, \dots, t_{2n_2} \ x_{2n_2}) = e_2 \\
 \quad \quad \dots \\
 \quad \quad \text{in } \text{body} \gg \\
 \text{tenv}) = t
 \end{array}$$

- La cláusula correspondiente a una expresión letrec en `type-of-expression` es:

```
(letrec-exp (result-texps proc-names texpss  
             idss bodies  
             letrec-body)  
  (type-of-letrec-exp  
   result-texps proc-names texpss idss bodies  
   letrec-body tenv))
```

- Donde se utiliza el procedimiento auxiliar `type-of-letrec-exp`.

- `type-of-letrec-exp` es un procedimiento que, inicialmente, convierte las expresiones de tipo de los argumentos y el resultado a tipos.
- Luego liga la lista de los tipos de los procedimientos a la variable `the-proc-types` y el ambiente del cuerpo de `letrec` a `tenv-for-body`.
- Finalmente, computa el tipo del cuerpo de cada procedimiento y lo compara con el tipo del resultado especificado.

La siguiente es la definición del procedimiento
`type-of-letrec-exp`:

```
(define type-of-letrec-exp
  (lambda (result-texps proc-names texpss idss bodies
          letrec-body tenv)
    (let ((arg-typess (map (lambda (texps)
                           (expand-type-expressions
                            texps))
                          texpss))
          (result-types (expand-type-expressions
                          result-texps)))
      (let ((the-proc-types
              (map proc-type arg-typess result-types)))
        (let ((tenv-for-body ;~ type env for all
                proc-bodies
                (extend-tenv proc-names the-proc-types
                             tenv)))
          (for-each
            (lambda (ids arg-types body result-type)
              (check-equal-type!
               (type-of-expression
                body
```

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

?

Fundamentos
de lenguajes
de
programación

Carlos Andrés
Delgado S.
Carlos Alberto
Ramírez

Tipos y
Programación

Chequeo de
tipos

■ Inferencia de tipos.