

Lenguajes de Programación

Tarea III

Andrea Itzel González Vargas
Karla Esquivel Guzmán
Carlos Gerardo Acosta Hernández
Facultad de Ciencias UNAM

Problema I

$$\begin{array}{c} \Gamma [x \leftarrow \text{number}], [\text{fib} (\text{number} \rightarrow \text{number})] \vdash (- x 1) : \text{number} \checkmark \\ \Gamma [x \leftarrow \text{number}], [\text{fib} (\text{number} \rightarrow \text{number})] \vdash (- x 2) : \text{number} \checkmark \\ \hline \Gamma [x \leftarrow \text{number}] \vdash x : \text{number} \quad \Gamma [x \leftarrow \text{number}] \vdash 1 : \text{number} \checkmark \\ \Gamma [x \leftarrow \text{number}] \vdash (\text{fib} (- x 1)) : \text{number} \\ \Gamma [x \leftarrow \text{number}] \vdash (\text{fib} (- x 2)) : \text{number} \\ \hline \Gamma [x \leftarrow \text{number}] \vdash x : \text{number} \checkmark \\ \Gamma [x \leftarrow \text{number}] \vdash 0 : \text{number} \checkmark \\ \Gamma [x \leftarrow \text{number}] \vdash (= x 1) : \text{bool} \quad \Gamma [x \leftarrow \text{number}] \vdash 1 : \text{number} \checkmark \\ \Gamma [x \leftarrow \text{number}] \vdash (+ (\text{fib} (- x 1)) (\text{fib} (- x 2))) : \text{number} \\ \hline \Gamma [x \leftarrow \text{number}] \vdash (= x 0) : \text{boolean} \\ \Gamma [x \leftarrow \text{number}] \vdash 0 : \text{number} \checkmark \\ \Gamma [x \leftarrow \text{number}] \vdash (\text{if} (= x 1) 1 (+ (\text{fib} (- x 1)) \text{fib} (- x 2)))) : \text{number} \\ \hline \Gamma [x \leftarrow \text{number}] \vdash (\text{if} (= x 0) 0 (\text{if} (= x 1) 1 (+ (\text{fib} (- x 1)) \text{fib} (- x 2)))) : \text{number} \\ \hline \Gamma \vdash \text{fib} (x : \text{number}) : \text{number} (\text{if} (= x 0) 0 (\text{if} (= x 1) 1 (+ (\text{fib} (- x 1)) \text{fib} (- x 2)))) : (\text{number} \rightarrow \text{number}) \\ \Gamma \vdash \text{empty?} : (\text{list} \rightarrow \text{bool}) \quad \Gamma \vdash l : \text{list} \checkmark \\ \hline \Gamma \vdash (\text{empty? } l) : \text{bool} \end{array}$$

Problema II

$\boxed{1} (+ \boxed{2} 1 \boxed{3} (\text{first } \boxed{4} (\text{cons } \boxed{5} \text{ true } \boxed{6} \text{ empty})))$

Retricciones

$\boxed{1}$ = number si $\boxed{2}$ = $\boxed{3}$ = number
 $\boxed{2}$ = number
 $\boxed{3}$ = number si $\boxed{4}$ = nlist
 $\boxed{4}$ = nlist si $\boxed{5}$ = number y $\boxed{6}$ = nlist
 $\boxed{5}$ = number si $\boxed{5}$ contiene un numeral, pero $\boxed{5}$ = boolean!!
 Por lo tanto esta mal formado el programa

Problema III

$\boxed{1} \{ \text{fun } \{f : C1\} : C2$
 $\quad \boxed{2} \{ \text{fun } \{x : C3\} : C4$
 $\quad \quad \boxed{3} \{ \text{fun } \{y : C5\} : C6$
 $\quad \quad \quad \boxed{4} \text{ cons } x \boxed{5} \{f \boxed{6} \{f y\}\}\}\}\}$

Acción	Stack	Sustitución
Inicio	$\boxed{1} = [f] \rightarrow \boxed{2}$ $\boxed{2} = [x] \rightarrow \boxed{3}$ $\boxed{3} = [y] \rightarrow \boxed{4}$ $[\text{cons}] = [x] \times [5] \rightarrow \boxed{4}$ $[f] = [6] \rightarrow \boxed{5}$ $[f] = [y] \rightarrow \boxed{6}$	Vacio
Paso 3	$\boxed{2} = [x] \rightarrow \boxed{3}$ $\boxed{3} = [y] \rightarrow \boxed{4}$ $[\text{cons}] = [x] \times [5] \rightarrow \boxed{4}$ $[f] = [6] \rightarrow \boxed{5}$ $[f] = [y] \rightarrow \boxed{6}$	$\boxed{1} \mapsto [f] \rightarrow \boxed{2}$
Paso 3	$\boxed{3} = [y] \rightarrow \boxed{4}$ $[\text{cons}] = [x] \times [5] \rightarrow \boxed{4}$ $[f] = [6] \rightarrow \boxed{5}$ $[f] = [y] \rightarrow \boxed{6}$	$\boxed{1} \mapsto [f] \rightarrow \boxed{2}$ $\boxed{2} \mapsto [x] \rightarrow \boxed{3}$
Paso 3	$[\text{cons}] = [x] \times [5] \rightarrow \boxed{4} = \text{number} \times \text{list} \rightarrow \text{list}$ $[f] = [6] \rightarrow \boxed{5}$ $[f] = [y] \rightarrow \boxed{6}$	$\boxed{1} \mapsto [f] \rightarrow [x] \rightarrow [y] \rightarrow \boxed{4}$ $\boxed{2} \mapsto [x] \rightarrow [y] \rightarrow \boxed{4}$ $\boxed{3} \mapsto [y] \rightarrow \boxed{4}$

Paso 5	$[[x]] = \text{number}$ $[[5]] = \text{list}$ $[[4]] = \text{list}$ $[[f]] = [[6]] \rightarrow [[5]]$ $[[f]] = [[y]] \rightarrow [[6]]$	$[[1]] \mapsto [[f]] \rightarrow [[x]] \rightarrow [[y]] \rightarrow [[4]]$ $[[2]] \mapsto [[x]] \rightarrow [[y]] \rightarrow [[4]]$ $[[3]] \mapsto [[y]] \rightarrow [[4]]$
Paso 3	$[[5]] = \text{list}$ $[[4]] = \text{list}$ $[[f]] = [[6]] \rightarrow [[5]]$ $[[f]] = [[y]] \rightarrow [[6]]$	$[[1]] \mapsto [[f]] \rightarrow \text{number} \rightarrow [[y]] \rightarrow [[4]]$ $[[2]] \mapsto \text{number} \rightarrow [[y]] \rightarrow [[4]]$ $[[3]] \mapsto [[y]] \rightarrow [[4]]$ $[[x]] \mapsto \text{number}$
Paso 3	$[[4]] = \text{list}$ $[[f]] = [[6]] \rightarrow \text{list}$ $[[f]] = [[y]] \rightarrow [[6]]$	$[[1]] \mapsto [[f]] \rightarrow \text{number} \rightarrow [[y]] \rightarrow [[4]]$ $[[2]] \mapsto \text{number} \rightarrow [[y]] \rightarrow [[4]]$ $[[3]] \mapsto [[y]] \rightarrow [[4]]$ $[[x]] \mapsto \text{number}$ $[[5]] \mapsto \text{list}$
Paso 4	$[[f]] = [[6]] \rightarrow \text{list}$ $[[f]] = [[y]] \rightarrow [[6]]$	$[[1]] \mapsto [[f]] \rightarrow \text{number} \rightarrow [[y]] \rightarrow \text{list}$ $[[2]] \mapsto \text{number} \rightarrow [[y]] \rightarrow \text{list}$ $[[3]] \mapsto [[y]] \rightarrow \text{list}$ $[[x]] \mapsto \text{number}$ $[[5]] \mapsto \text{list}$ $[[4]] \mapsto \text{list}$
Paso 3	$[[6]] \rightarrow \text{list} = [[y]] \rightarrow [[6]]$	$[[1]] \mapsto [[6]] \rightarrow \text{list} \rightarrow \text{number} \rightarrow [[y]] \rightarrow \text{list}$ $[[2]] \mapsto \text{number} \rightarrow [[y]] \rightarrow \text{list}$ $[[3]] \mapsto [[y]] \rightarrow \text{list}$ $[[x]] \mapsto \text{number}$ $[[5]] \mapsto \text{list}$ $[[4]] \mapsto \text{list}$ $[[f]] \mapsto [[6]] \rightarrow \text{list}$
Paso 5	$[[6]] = [[y]]$ $\text{list} = [[6]]$	$[[1]] \mapsto [[6]] \rightarrow \text{list} \rightarrow \text{number} \rightarrow [[6]] \rightarrow \text{list}$ $[[2]] \mapsto \text{number} \rightarrow [[6]] \rightarrow \text{list}$

		$\begin{aligned} &[[3]] \mapsto [[y]] \rightarrow \text{list} \\ &[[x]] \mapsto \text{number} \\ &[[5]] \mapsto \text{list} \\ &[[4]] \mapsto \text{list} \\ &[[f]] \mapsto [[6]] \rightarrow \text{list} \end{aligned}$
Paso 4	list = $[[6]]$	$\begin{aligned} &[[1]] \mapsto [[6]] \rightarrow \text{list} \rightarrow \\ &\text{number} \rightarrow [[6]] \rightarrow \text{list} \\ &[[2]] \mapsto \text{number} \rightarrow [[6]] \rightarrow \\ &\text{list} \\ &[[3]] \mapsto [[6]] \rightarrow \text{list} \\ &[[x]] \mapsto \text{number} \\ &[[5]] \mapsto \text{list} \\ &[[4]] \mapsto \text{list} \\ &[[f]] \mapsto [[6]] \rightarrow \text{list} \\ &[[y]] \mapsto [[6]] \end{aligned}$
Paso 4	Vacio	$\begin{aligned} &[[1]] \mapsto \text{list} \rightarrow \text{number} \rightarrow \\ &\text{list} \rightarrow \text{list} \\ &[[2]] \mapsto \text{number} \rightarrow \text{list} \rightarrow \\ &\text{list} \\ &[[3]] \mapsto \text{list} \rightarrow \text{list} \\ &[[x]] \mapsto \text{number} \\ &[[5]] \mapsto \text{list} \\ &[[4]] \mapsto \text{list} \\ &[[f]] \mapsto \text{list} \rightarrow \text{list} \\ &[[y]] \mapsto \text{list} \\ &[[7]] \mapsto \text{list} \end{aligned}$

Problema IV

En ninguno de los dos casos cambia la regla de tipado. Para justificarlo, primero recordemos las reglas correspondientes.

- Definición de funciones:

$$\frac{\Gamma[i \leftarrow \tau_1] \vdash b : \tau_2}{\Gamma \vdash \{fun \{i : \tau_1\} : \tau_2 \ b\} : (\tau_1 \rightarrow \tau_2)}$$

- Aplicación de funciones:

$$\frac{\Gamma \vdash f : (\tau_1 \rightarrow \tau_2) \quad \Gamma \vdash a : \tau_1}{\Gamma \vdash \{f\ a\} : \tau_2}$$

Veamos que para determinar el tipo de las expresiones, nunca se hace alusión directa a la evaluación. En las funciones se habla en cambio de tipos de entrada y tipos de salida que no restringimos a ser el mismo. Podríamos pensar para nuestro lenguaje perezoso que en la aplicación de función si recibimos una expresión en lugar de un valor del tipo de entrada esperado tendríamos que hacer ciertas modificaciones a la regla de tipado, sin embargo ya sabemos que uno de los puntos estrictos de un lenguaje perezoso ocurre precisamente en una aplicación de función, por lo que se está forzando la evaluación y la función recibe el resultado de dicha evaluación que debe cumplir el tipo especificado. Esto para la entrada. Para el tipo de salida, si exigimos un resultado, aún con un lenguaje perezoso no podemos devolver una expresión sin evaluar, por lo que no necesitamos modificar la regla de tipado. Tanto el argumento de la función como el resultado tendrá la misma forma para ambas evaluaciones. Por lo tanto, no es necesario cambiar la regla original para la versión perezosa del lenguaje.

Problema V

Explícito

En el polimorfismo explícito los tipos de los objetos deben ser declarados en su inicialización para cada uno antes de ser operados. Esto nos lleva a una programación repetitiva, pesada y mucho más restrictiva. El control del código puede disminuir con el uso de este polimorfismo por la obligatoriedad de hacer presente el tipo. Sin embargo, en su uso se encuentran ciertas ventajas. Para empezar, la creación de nuevas clases de objetos sin afectar la forma ya existente de la que procede. También la facilidad con la que se recicla el código es mayor y por lo general los lenguajes proveen de ciertas herramientas para evitar la presencia explícita del tipo más que al momento de inicializarlo.

Implícito

Uno de los ejes en los lenguajes de programación con polimorfismo implícito es la utilización de código en común si la abstracción semántica lo permite. Ello se traduce en una de las ventajas más grandes sobre el otro, la simplicidad y la versatilidad del código. Es resumido en “nunca repitas el código”. Si encontramos una o más secciones de código repetidas es porque el autor del código falló en identificar una abstracción en común para diferentes problemas. Sin embargo, para vislumbrar un par de desventajas podemos considerar el siguiente escenario: Bajo el contexto de programación en *Java*, consideremos una lista con capacidad para almacenar objetos de cualquier tipo. Para evitar la repetición del código, es posible hacer uso de la jerarquía de tipos de acuerdo al supertipo más general que es *Object*. Ahora, al momento de tomar elementos de la lista para operar con ellos, nos veremos obligados a hacer un *cast* a su tipo específico para el contexto particular en el que será utilizado. Por supuesto, obtendremos un código desordenado y perderemos precisión

sobre las tareas que los objetos son capaces de realizar, de hecho puede que nos veamos afectados por un nuevo factor de incertidumbre. Sin embargo, este tipo de desventajas es para situaciones o circunstancias menos generales, por lo que no significa una contraparte fuerte en comparación con las ventajas que ofrece.

Problema VI

DSL - Lenguajes de Dominio Específico

Ventajas	Desventajas
<ul style="list-style-type: none">■ Permiten ofrecer soluciones expresadas en el mismo nivel de abstracción del dominio del problema.■ Ofrecen una mayor oportunidad de extrapolación de un grupo desarrollador especializado en el dominio de algún sistema específico que necesite actualizarse a un grupo desarrollador más general. Esto implica la siguiente ventaja.■ Las herramientas de trabajo que provee el DSL permite que los desarrolladores tengan facilidad para introducirse o desenvolverse en el área del dominio sin la necesidad de que se especialicen en ella.■ Puesto que la semántica del lenguaje permanece limitada por el dominio, la aparición de errores de este tipo se mantiene igualmente acotada.	<ul style="list-style-type: none">■ El costo de aprender un nuevo lenguaje es grande si se considera el limitado universo de aplicaciones que tiene un DSL.■ Supone una dificultad mayor integrar las aplicaciones programadas en un DSL con otros componentes realizados con un lenguaje de propósito general para algún sistema.■ Los desarrolladores no expertos en el área a la que está orientado el dominio del lenguaje encontrarán dificultades para crear o modificar código.■ Esta baja oferta de expertos o conocedores de un DSL particular provocará un alza en la valoración monetaria, posiblemente desproporcionada respecto del esfuerzo real, del código escrito en dicho lenguaje.

GPL - Lenguajes de Propósito General

Ventajas	Desventajas
<ul style="list-style-type: none">■ Se puede abarcar una gran cantidad de problemas de diversas áreas.■ Suelen ser más fáciles de aprender para tareas generales y las soluciones que ofrecen suelen ser más fácilmente traducidas a otro GPL.■ La cantidad de código ya escrito y disponible para el público es superior a los DSL por lo que se cuenta ya con una diversidad importante de bibliotecas y documentación que sirven de apoyo para el desarrollo.	<ul style="list-style-type: none">■ Por la naturaleza general del dominio, puede que no sea la mejor opción para un problema de un dominio específico.■ Por lo general están deprovistos de funcionalidades especializadas para un dominio particular, por lo que puede volverse más laboriosa la implementación de soluciones para problemas pertenecientes a un área específica.■ Debido a la diversidad de áreas en que son aplicables, la adquisición de un nivel de habilidad que cubra cualquier posibilidad para el usuario de estos lenguajes es prácticamente imposible.

Ejemplos DSL:

1. Algol: Actualmente una familia de lenguajes de programación imperativos, fue desarrollado a mitad de la década de los años 50's principalmente como herramienta de investigación en las ciencias de la computación. Significó una gran influencia para muchos otros lenguajes de programación posteriores a su aparición y constituyó el método estándar de descripción de algoritmos utilizado en la academia por muchos años. Su nombre indica su fuerte orientación a la implementación de algoritmos (**A**lgorithm **L**anguage).

```
PROC factorial = (INT upb n)LONG LONG INT:(  
  LONG LONG INT z := 1;  
  FOR n TO upb n DO z *:= n OD;  
  z  
); ~
```

La versión iterativa de factorial implementada en *Algol 68*.

2. Lisp: A pesar de ser ahora una familia de lenguajes, su aparición en 1958 lo hace el segundo lenguaje de programación de alto nivel más viejo (después de Fortran). Lisp fue originalmente creado como una notación matemática pragmática para programas de computadora, influenciada en la notación del cálculo lambda. Su nombre refiere a procesamiento de listas ("list-processing").

```
(defun list-reverse (L)
  (if (null L)
      nil
      (list-append (list-reverse (rest L))
                    (list (first L)))))
```

La función recursiva list-reverse recibe una lista L y devuelve una nueva lista con los elementos de L al revés.

3. SQL: Es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones en ellas. Los orígenes de SQL están ligados a los de las bases de datos relacionales en los años 70's.

```
SELECT * FROM TABLE1;
```

Esta consulta nos permite obtener todas las tuplas de la tabla TABLE1.