Resumen: Paradigmas de Lenguajes de Programación

Zamboni, Gianfranco

15 de febrero de 2018

Índice general

0.1. Introducción . . . 1
0.1.1. Aspectos del lenguaje . . . 1
0.1.2. Paradigmas . . . 1

1. Parádigma Funcional **3**
1.1. Programación Funcional . . . 3
1.1.1. Tipos . . . 3
1.1.2. Tipo Función . . . 4
1.1.3. Inducción/Recursion . . . 5
1.1.4. Parametrización . . . 5
1.1.5. Tipos algebraicos . . . 6
1.1.6. Tipos algebraicos recursivos . . . 7
1.1.7. Esquemas de recursión . . . 8
1.2. Cálculo Lambda Tipado . . . 10
1.2.1. Expresiones de Tipos de λ^b . . . 10
1.2.2. Sistema de tipado . . . 12
1.2.3. Semántica operacional . . . 13
1.2.4. Semántica operacional de λ^b . . . 15
1.2.5. Extensión Naturales (λ^{bn}) . . . 15
1.2.6. Simulación de lenugajes imperativos . . . 16
1.2.7. Extensión con recursión . . . 19

A. Programación funcional en Haskell **21**
A.0.1. Otros tipos útiles . . . 22

B. Extensiones del lenguaje λ^b **23**
B.0.1. Extensión con Registros $\lambda^{\cdots r}$ . . . 23
B.0.2. Extensión con Declaraciones Locales ($\lambda^{\cdots let}$) . . . 24
B.0.3. Extensión con tuplas . . . 24

0.1. Introducción

Paradigma Marco filosófico y teórico de una escuela científica o disciplina en la que se formulan teorías, leyes y generalizaciones y se llevan a cabo experimentos que les dan sustento.

Lenguaje de programación Es un lenguaje usado para comunicar instrucciones a una computadora. Estas instrucciones describen cómputos que llevará a cabo la computadora.

Un lenguaje de programación es computacionalmente completo si puede expresar todas las funciones computables.

Paradigma de lenguaje de programación Marco filosófico y teórico en el que se formulan soluciones a problemas de naturaleza algorítmica. Lo entendemos como un estilo de programación en el que se escriben soluciones a problemas en términos de algoritmos.

Su ingrediente básico es el modelo de cómputo que es la visión que tiene el usuario de cómo se ejecutan sus programas.

0.1.1. Aspectos del lenguaje

Sintaxis Descripción del conjunto de secuencias de símbolos considerados como programas válidos.

Semántica

Descripción del significado de instrucciones y expresiones puede ser informal (e.g. Castellano) o formal (basado en técnicas matemáticas).

Semántica operacional Un programa es un mecanismo que dado un elemento del conjunto de partida, sigue una sucesión de pasos para calcular el elemento correspondiente del conjunto de llegada.

Semántica axiomática Interpreta a un programa como un conjunto de propiedades verdaderas que indican los estados que puede llegar a tomar.

Semántica denotacional Un programa es un valor matemático (función) que relaciona cada elemento de un conjunto de partida (expresiones que lo componen) con un único elemento de otro conjunto de llegada (significado de las expresiones).

Sistema de tipo Es una herramienta que nos permite analizar código para prevenir errores comunes en tiempo de ejecución (e.g. evitar sumar booleanos, aplicar función a número incorrecto de argumentos, etc). En general, requiere anotaciones de tipo en el código fuente.

Además sirve para que la especificación de un programa sea más clara.

Hay dos tipos de análisis de tipos:

- **Estático**: Análisis de tipos en tiempo de compilación.
- **Dinámico**: Análisis de tipos en tiempo de ejecución.

0.1.2. Paradigmas

Paradigma Imperativo

Estado global Se usan variables que representan celdas de memoria en distintos momentos del tiempo. En ellas vamos almacenando resultados intermedios del problema.

Asignación Es la acción que modifica las celdas de memoria

Control de flujo Es la forma que tenemos de controlar el orden y la cantidad de veces que se repite un cómputo dentro del programa. En este paradigma, la repetición de cómputos se basa en la iteración.

Por lo general, los lenguajes de este paradigma son eficientes ya que el modelo de ejecución usado y la arquitectura de las computadoras (a nivel procesador) son parecidos. Sin embargo, el bajo nivel de abstracción que nos proveen hacen que, por lo general, la implementación de un problema sea difícil de entender.

Paradigma Funcional

No tiene un estado global. Un cómputo se expresa a través de la aplicación y composición de funciones y los resultados intermedios (salida de las funciones) son pasados directamente a otras funciones como argumentos. Todas las expresiones de este paradigma son tipadas y usa la recursión para repetir cómputos.

Ofrece un alto nivel de abstracción, es declarativo, usa una matemática elegante y se puede usar razonamiento algebraico para demostrar correctitud de programas.

Paradigma Lógico

Los programas son predicados de la lógica proposicional y la computación esta expresada a través de proof search. No existe un estado global y los resultados intermedios son pasados por unificación. La repetición se basa en la recursión.

Ofrece un alto nivel de abstracción, es muy declarativo y, al ser predicados, tiene fundamentos lógicos robustos pero su ejecución es muy lenta.

Paradigma Orientado a Objetos

La computación se realiza a través del intercambio de mensajes entre objetos. Tiene dos enfoques: basados en clases o basados en prototipos.

Ofrece alto nivel de abstracción y arquitecturas extensibles pero usa una matemática de programas compleja.

Capítulo 1

Parádigma Funcional

1.1. Programación Funcional

Los dos aspectos fundamentales de la programación son:

- Transformación de la información.
- Interacción con el medio (cargar datos, interfaces gráficas, etc).

La programación funcional se concentra en el primer aspecto.

Valor Entidad matemática abstracta con ciertas propiedades.

Expresión Secuencia de símbolos utilizada para denotar un valor. Hay dos tipos de expresiones:

- **Atómicas ó formas formales**: Son las expresiones más simples y denotan un valor.
- **Compuestas**: Expresiones que se construyen combinando otras expresiones.

Puede haber expresiones incorrectas (mal formadas) debido a errores sintácticos (expresiones mal escritas) o a errores de tipo (expresiones que denotan operaciones sobre tipos incorrectos).

En funcional, computar significa tomar una expresión y reducirla hasta que sea atómica.

Transparencia referencial El valor que denota una expresión solo depende de los símbolos que la constituyen. Esto nos permite indicar. Esto nos permite hacer uso de un programa sin considerar la necesidad de considerar los detalles de su ejecución y nos permite demostrar propiedades usando las propiedades de las subexpresiones y métodos de deducción lógica.

1.1.1. Tipos

Son una forma de particionar el universo de valores de acuerdo a ciertas propiedades. Hay:

- **Tipos básicos** (Int, Bool, Float) ó primitivos que son los que ya vienen definidos en el lenguaje por literales y representan valores
- **Tipos compuestos** (pares, listas) que son aquellos que se definen a partir de otros tipos.

Cada tipo de dato tiene asociado operaciones que no tienen significado para otros tipos.

A toda expresión bien formada se le puede asignar un tipo que sólo depende los componentes de la expresión (strong-typing). Dada una expresión, se puede deducir su tipo a partir de su constitución.

Notación

`e :: A` se lee "la expresión `e` tiene tipo `A`" y significa que el valor denotado por `e` pertenece al conjunto de valores denotado por `A`.

Propiedades deseables de un lenguaje funcional

Se busca que un lenguaje le asigne un tipo de manera automática al mayor número posible de expresiones con sentido y que no le asigne ningún tipo al mayor número posible de expresiones mal formadas. Además, se busca que el tipo de la expresión se mantenga si es reducida.

Otra cosa a tener en cuenta, es que los tipos ofrecidos por el lenguaje deben ser descriptivos y razonablemente sencillos de leer.

Inferencia de tipos Dada una expresión e determinar si tiene tipo o no y, si lo tiene, cuál es ese tipo según las reglas.

Chequeo de tipos Dada una expresión `e` y un tipo `A`, determinar si `e :: A` o no.

1.1.2. Tipo Función

Un programa en el paradigma funcional es una función descripta por un conjunto de ecuaciones (expresiones) que definen uno o más valores. Estas ecuaciones son evaluadas (reducidas) hasta llegar a una expresión atómica que nos indique el valor de las mismas.

Funciones Las funciones son valores especiales que representan transformación de datos. En haskell el tipo de una función se escribe: `->`. Las funciones se aplican a elementos de un conjunto de entrada definido por el tipo de entrada de la función y devuelve un elemento del tipo de salida.

Al ser valores, las funciones pueden ser argumentos y resultados de otras funciones, pueden almacenarse y pueden ser estructuras de datos.

Funciones de alto orden Son funciones que manipulan otras funciones.

Lenguaje Funcional Puro Lenguaje de expresiones con transparencia referencial y funciones como valores, cuyo modelo de cómputo es la reducción realizada mediante el reemplazo de iguales por iguales.

Polimorfismo paramétrico Cuando una función tiene un parámetro que puede ser instanciado de diferentes maneras en diferentes usos. Esta propiedad se da dentro de los sistemas de tipos.

Dada una expresión que puede ser tipada de infinitas maneras, el sistema puede asignarle un tipo que sea más general que todos ellos, y tal que en cada uso pueda transformarse en uno particular.

Hay funciones que a pesar de poseer polimorfismo paramétrico, no aceptan cualquier clase de tipo, sino que requieren que los tipos con las que son llamadas tengan ciertas propiedades. Por ejemplo, que tengan relaciones de igualdad (`Eq`), relación de order (`Ord`), que se comporten como números (`Num`) o que puedan ser mostrados en pantalla (`Show`)

Evaluación

Por lo general, dependiendo del orden de evaluación del lenguaje, el tipo de evaluación se clasifica en:

Evaluación Estricta Si una parte de una expresión se indefine, entonces la expresión se indefine. La evaluación eager, en la que un en lenguaje computa una expresión apenas es definida, es de este tipo.

Evaluación no Estricta Puede pasar que una expresión esté definida a pesar de que alguna de sus partes se haya indefinido. La evaluación lazy, en la que un lenguaje solo computa una expresión cuando de esta depende el valor de otra expresión, es de este tipo.

Haskell usa evaluación lazy de izquierda a derecha, resolviendo primero las partes más externas de la expresión y luego, si es necesario, sus partes.

Currificación Correspondencia entre cada función de múltiples parámetros y una de alto orden que retorna una función intermedia que completa el trabajo. Por cada f definida como:

```
f :: (a,b) -> c
f (x,y) = e
```

existe un función f' tal que se puede escribir:

```
f' :: a -> (b -> c)
(f' x) y = e
```

La currificación nos da mayor expresividad y la posibilidad de realizar evaluación parcial. Además, nos permite tratar el código de manera más modular al momento de inferir tipos y transformar programas.

Evaluación parcial Se evalúan las funciones parcialmente, lo que nos permite llamarlas con menos parámetros de los que necesitan. Esto nos devuelve una función con las expresiones asociadas a los valores pasados como parámetros y que toma como parámetros los parámetros faltantes de la función original.

1.1.3. Inducción/Recursion

La inducción es un mecanismo que nos permite definir conjuntos infinitos, probar propiedades sobre sus elementos y definir funciones recursivas sobre ellos con garantía de terminación.

Principio de extensionalidad: Dadas dos expresiones A y B, si A y B denotan el mismo valor, entonces A puede ser remplazada por B y B por A sin que esto afecte al resultado de una equación.

Inducción estructural

Una definición inductiva de un conjunto $\mathcal{R}$ consiste en dar condiciones de dos tipos:

- reglas base ($z \in \mathcal{R}$) que afirman que algún elemento simple x pertenece a $\mathcal{R}$
- reglas inductivas ($y_1 \in \mathcal{R}, \ldots, y_n \in \mathcal{R} \Rightarrow y \in \mathcal{R}$) que afirman que un elemento compuesto y pertenece a $\mathcal{R}$ siempre que sus partes $y_1, \ldots, y_n$ pertenezcan a $\mathcal{R}$ (e y no satisface otra regla de las dadas)

y pedir que $\mathcal{R}$ sea el menor conjunto (en sentido de la inclusión) que satisfaga todas las reglas dadas.

Funciones recursivas

Sea `S` un conjunto inductivo, y `T` uno cualquiera. Una definición recursiva estructural de una función `f :: S -> T` es una definición de la siguiente forma:

- Por cada elemento base `z`, el valor de `(f z)` se da directamente usando valores previamente definidos
- Por cada elemento inductivo `y`, con partes inductivas `y1`, ..., `yn`, el valor de `(f y)` se da usando valores previamente definidos y los valores `(f y1)`, ..., `(f yn)`.

Principio de inducción

Sea S un conjunto inductivo, y sea P una propiedad sobre los elementos de S. Si se cumple que:

- para cada elemento $z \in S$ tal que z cumple con una regla base, $P(z)$ es verdadero, y
- para cada elemento $y \in S$ construido en una regla inductiva utilizando los elementos $y_1, ..., y_n$, si $P(y_1), ..., P(y_n)$ son verdaderos entonces $P(y)$ lo es

entonces $P(x)$ se cumple para todos los $x \in S$.

1.1.4. Parametrización

Dado un conjunto de funciones que se comportan de la misma manera buscamos encontrar alguna forma de crear una función que las genere automáticamente.

Esquema de funciones Dado un conjunto de funciones "parecidas", el esquema de estas funciones son los que no permiten parametrizar correctamente alguno de los parámetros.

La parametrización nos permitirá crear definiciones más concisas y modulares, reutilizar código y demostrar propiedades generales de manera más fácil.

1.1.5. Tipos algebraicos

Definición de tipos

Hay dos formas de definir un tipo de dato:

- **De manera algebraica:** Establecemos qué *forma* tendrá cada *elemento* y damos un mecanismo único para inspeccionar cada elemento.
- **De manera abstracta:** Determinamos cuales serán las *operaciones* que manipularán los elementos, **SIN** decir cuál será la forma exacta del tipo ni de las operaciones que definimos.

Tipos algebraicos en Haskell

Los definimos mediante **constantes** llamadas *constructores* cuyos nombres comienzan con mayúscula. Los constructores no tienen asociada una regla de reducción y pueden tener argumentos.

Para implementarlos en Haskell, usamos la clausula `data` que introduce un nuevo tipo algebraico, los nombres de su constructores y sus argumentos.

Ejemplos:

```
data Sensacion = Frio | Calor
data Shape = Circle Float | Rect Float Float
```

Los tipos algebraicos pueden tener argumentos. Esto nos permite definir tipos que contienen al conjunto de elementos de otro tipo más los elementos del tipo que se están definiendo.

Ejemplo:

```
data Maybe = Nothing | Just a
```

`Maybe` tiene todos los elementos del tipo a con `Just` adelante más el elemento *Nothing*

Son considerados tipos algebraicos porque:

- toda combinación válida de constructores y valores es elemento del tipo algebraico (y solo ellas lo son)
- y porque dos elementos de un tipo algebraico son iguales si y solo si están construidos utilizando los mismos constructores aplicados a los mismos valores.

Al principio de esta sección, dijimos que además de establecer la forma que tiene el tipo, debemos dar un mecanismo único de inspección. En Haskell, este mecanismo es el **Pattern Matching**.

Pattern Matching

El pattern matching es la búsqueda de patrones especiales (en nuestro caso, los constructores de nuestro tipo) dentro de una expresión en el lado izquierdo de una ecuación que, si tiene éxito, nos permita inspeccionar el valor de la misma.

Si el pattern matching resulta exitoso, entonces ligas las variables del patrón.

Tipos especiales

Tupla Este tipo es un tipo algebraico con sintaxis especial. Una tupla es un estructura que posee varios elementos de distintos tipos. Por ejemplo: `(Float,Int)` es una tupla cuyo primer elemento es un `Float` y tiene como segundo elemento a un `Int`.

Maybe El tipo `Maybe`, definido en el último ejemplo, nos permite expresar la posibilidad de que el resultado sea erróneo, sin necesidad de usar casos especiales. De esta forma, logramos evitar el uso de $\perp$ hasta que el programador lo decida, permitiendo controlar errores.

Either El tipo `Either` representa la unión disjunta de dos conjuntos (los elementos de uno se identifican con `Left` y los del otro con `Right`. Sirve para mantener el tipado fuerte y poder devolver elementos de distintos tipos o para representar el origen de un valor.

```
data Either = Left a | Right b
```

Expresividad

Los tipos algebraicos no pueden representar cualquier cosa, por ejemplo, los números racionales son pares de enteros (numerador, denominador) cuya igualdad puede no depender de los valores con los que fueron construidos o incluso pueden llegar a no ser validos. Esto es así porque no todo par de enteros es un número racional, por ejemplo el (1,0).

Además recordemos que la igualdad de dos elementos de un tipo algebraico solo se da si estos fueron construidos exactamente de la misma forma. Si seguimos con el ejemplo de los racionales, sabemos que hay racionales iguales con distinto numerador y denominador como el (4,2) y el (2,1), sin embargo estos dos pares no podrían ser nunca iguales si fuesen tomados como un tipo algebraico.

Clases de tipos algebraicos

Enumerativos Solo constructores sin argumentos.

Productos Un único constructor con varios argumentos.

Sumas Varios constructores con argumentos.

Recursivos Utilizan el tipo definido como argumento.

1.1.6. Tipos algebraicos recursivos

Un tipo algebraico recursivo tiene al menos menos uno de los constructores con el tipo que se define como argumento y es la concreción, en Haskell, de un conjunto definido inductivamente.

Cada constructor define un caso de una definición inductiva de un conjunto. Si tiene al tipo definido como argumento, entonces es un caso inductivo, si no, es un caso base.

En estos caso, el pattern matching nos da una forma de realizar analizar los casos y de acceder a los elementos inductivos que forman a un elemento dado. Por esta razón, se pueden definir funciones recursivas.

A estos tipos, les damos un significado a través de funciones definidas recursivamente. Estas funciones manipulan simbólicamente al tipo. Sin embargo, estas manipulaciones, por si solas no tienen un significado, sino que el significado se lo dan las propiedades que dichas manipulaciones deben cumplir.

Enteros Notación unaria para expresar tipos enteros.

```
data N = Z | S N
```

Listas Definición equivalente a las listas de Haskell

```
data List a = Nil | Cons a (List a)
```

Árboles Un árbol es un tipo algebraico tal que al menos un elemento compuesto tiene dos componentes inductivas.

```
data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)
```

1.1.7. Esquemas de recursión

Cuando tenemos un conjunto de funciones que manipulan ciertas estructuras de manera similar, podemos abstraer este comportamiento en funciones de alto orden que nos facilitarán su escritura.

A continuación, veremos unos ejemplos de esquemas sobre listas:

Map Dada una lista `l`, aplica una función `f` a cada elemento de `l`.

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = (f x) : (map f xs)
```

Ejemplo:

```
doble x = x + x
dobleL = map doble
```

`dobleL` calcula el doble de cada elemento de una lista.

Filter Dada una lista `l` y un predicado `p`, selecciona todos los elementos de `l` que cumplen `p`.

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) | (p x)     = x : (filter p xs)
                | otherwise = filter p xs
```

Ejemplo

```
masQueCero = filter (>0)
```

`masQueCero` se queda con todos los elementos mayores de una lista

Fold La función fold es la función que expresa el patrón de recursión estructural sobre listas como función de alto orden. Dada una lista `l` y una función `f` que denota un valor que depende de todos los elementos de la lista `l` y un valor inicial `z`, aplica y combina las soluciones parciales obtenidas por `f` de manera "iterativa". Hay dos tipos de fold: `foldr` (acumula desde la derecha) y `foldl` (acumula desde la izquierda).

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z [] = z
foldr f z (x:xs) = f x (foldr f z xs)


foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z [] = z
foldl f z (x : xs) = foldl f (f z x) xs
```

Ejemplos

```
map f = foldr (\x rec -> (f x): rec) []
filter p = foldr (\x rec -> if (p x) then x:rec else rec) []
```

Recursión primitiva Recordemos de Logica y Computabilidad: una función h es recursiva primitiva si h es de la forma:

$$h(x_1, \ldots, x_n, 0) = f(x_1, \ldots, x_n)$$
$$h(x_1, \ldots, x_n, t+1) = g(h(x_1, \ldots, x_n, t), x_1, \ldots, x_n, t)$$

Es decir, el caso recursivo de h no solo depende de la descomposición de sus parámetros, sino que, además, depende de sus parámetros.

En Haskell, podemos definir una función que dada una lista `l`, un caso base `z` y un caso recursivo primitivo `f`, aplique la definición de `z` y `f` a la lista:

```
recr :: b -> (a -> [a] -> b -> b) -> [a] -> b
recr z _ []= z
recr z f (x:xs) = f x xs (recr z f xs)
```

En listas, este tipo de esquemas es difícil de ver. Como ejemplo, escibimos la función `insertar` de una lista con recursión primitiva:

```
-- Insert con pattern matching
insert :: a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x<y then (x:y:ys) else (y:insert x ys)

-- Insert con recursión primitiva
insert x = recr [x] (\y ys zs -> if x<y then (x:y:ys) else (y:zs))
```

En el segundo caso, `insert` es una función que agrega el elemento `x` a una lista `xs` que se le pase como parámetro.

Divide & Conquer La técnica de Divide & Conquer consiste en dividir un problema en problemas más fáciles de resolver y luego, combinando los resultados parciales, lograr obtener un resultado general. En este caso, `DivideConquer` es un tipo de función, es decir define una familia de funciones, que toman como parámetro 4 funciones y un elemento de tipo `a` y devuelve un elemento de tipo `b`:

```
type DivideConquer a b  = (a -> Bool) -> (a -> b) -> (a -> [a]) ->
                          ([b] -> b) -> a -> b
```

Las funciones que toma como parámetro son:

- `esTrivial :: a -> Bool` que devuelve verdadero si elemento de tipo `a` es el caso base del problema.
- `resolver :: a -> b` que resuelve el problema cuando el elemento de tipo `a` es el caso trivial
- `repartir :: a -> [a]` que divide al elemento de tipo `a` en la cantidad de subproblemas necesarios para resolver el problema.
- `combinar :: [b] -> b` que resuelve todos los subproblemas obtenidos por `repartir` y combina sus soluciones para obtener el resultado final.

Ejemplo
Vamos a definir el Divide & Conquer para listas:

```
divideConquerListas :: DivideConquer [a] b
-- Esto significa que DivideConquerLista es de tipo
-- ([a] -> Bool) -> ([a] -> b) -> ([a] -> [[a]]) -> ([b] -> b)
-- -> [a] -> b

divideConquerListas esTrivial resolver repartir combinar l =
        if (esTrivial l) then resolver l
        else combinar (map dc (repartir l))
where dc = divideConquerListas esTrivial resolver repartir combinar
```

Otros esquemas de recursión Los esquemas de recursión que nombramos, no son los únicos que existen y además, pueden ser definidos para otros tipos recursivos, no solo para listas.

La función fold y como definirla

Todo tipo algebraico tiene asociado un patrón de inducción estructural. En particular, dado un tipo algebraico recursivo `T`, podemos definir la función `foldrT:: * -> a` donde * son los parámetros de la función. A continuación damos algunas propiedades que debe cumplir para asegurarnos de la definimos correctamente:

- Por cada constructor recursivo debe tomar una función que tome como parámetros a cada elemento del constructor que no sea del tipo `T` y un parámetro de tipo `a` por cada elemento del tipo `T` del constructor. Esta función devuelve un elemento del tipo `a` y es la que resolverá recursivamente el caso planteado usando la segunda clase de parámetros.
- Por cada constructor base de `T` debe tomar un parámetro de tipo a que será el elemento devuelto por la función si cae en alguno de dichos casos.
- Por último, si la función está bien implementada, si remplazamos cada parámetro por el contructor correspondiente que tiene asignado, la función resultante debería ser la función identidad del tipo `T`.

Al momento de definir `fold` ayuda mucho plantear el esquema de recursión del tipo.

1.2. Cálculo Lambda Tipado

El cálculo lambda es un modelo de computación turing completo basado en **funciones** introducido por **Alonzo Church**. Este modelo consiste en un conjunto de expresiones o terminos que representan abstracciones o aplicaciones de funciones y cuyos valores pueden ser determinados aplicando ciertas reglas sintacticas hasta obtener lo que se dice su forma normal, una expresión que, a falta de reglas no puede ser reducida de ninguna manera. En nuestro caso, estamos estudiando cálculo lambda tipado, es decir que habrá expresiones que, a pesar de estar bien formadas, no tendrán sentido.

1.2.1. Expresiones de Tipos de λ^b

El primer lenguaje lambda que usamos en la materia tiene dos **tipos** $Bool$ y $\sigma \to \theta$ que son los tipos de los valores booelanos y las funciones que van de un tipo σ a un tipo θ, respectivamente. Y lo notamos:

$$\sigma, \theta \ ::= \ Bool \mid \sigma \to \theta$$

Una vez que definamos por completo el lenguaje lambda para estos dos tipos, esto es definir reglas de sintaxis, de tipado y de reducción de expresiones, vamos a extender el lenguaje con los naturales y, luego, con otros tipos de interés, como abstracciones de memoria y comandos.

Términos de λ^b

Ahora debemos definir los **términos** que nos permitirán escribir las expresiones válidas del tipado. Sea $\mathcal{X}$ un conjunto infinito enumerable de variables y $x \in \mathcal{X}$. Los **términos** de λ^b están dados por:

$$\begin{aligned} M, P, Q \ ::= \ & true \\ & \mid false \\ & \mid if\ M\ then\ P\ else\ Q \\ & \mid M\ N \\ & \mid \lambda x : \sigma . M \\ & \mid x \end{aligned}$$

Esto significa que dados tres términos M, P y Q, los términos válidos del lenguaje son:

- $true$ y $false$ que representan las **constantes de verdad**.
- $if\ M\ then\ P\ else\ Q$ que expresa el **condicional**.
- $M\ N$ que indica la **aplicación** de la función denotada por el termino M al argumento N.
- $\lambda x : \sigma . M$ que es una **función** (abstracción) cuyo parámetro formal es x y cuyo cuerpo es M
- x, una **variable de términos**.

Variables ligadas y libres

Por como definimos el lenguaje, una variable x puede ocurrir de dos formas: **libre** o **ligada**. Decimos que x ocurre **libre** si no se encuentra bajo el alcance de una ocurrencia de λx. Caso contrario ocurre ligada.

Por ejemplo:

$$\lambda x : Bool.if\ true\ then\ \underbrace{x}_{ligada}\ else\ \underbrace{y}_{libre}$$

Para conseguir las variables ligadas de una expresión, vamos a definir la función FV que toma como parámetro una expresión y devuelve el conjunto de variables libres de la misma.

$$\begin{aligned} FV(x) &\stackrel{def}{=} x \\ FV(true) = FV(false) &\stackrel{def}{=} \phi \\ FV(if\ M\ then\ P\ else\ Q) &\stackrel{def}{=} FV(M) \cup FV(P) \cup FV(Q) \\ FV(M\ N) &\stackrel{def}{=} FV(M) \cup FV(N) \\ FV(\lambda x : \sigma.M) &\stackrel{def}{=} FV(M) \backslash \{x\} \end{aligned}$$

Reglas de sustitución

Una de las operaciones que podemos realizar sobre las expresiones del lenguaje es la **sustitución** que, dado un término M, sustituye todas las ocurrencias **libres** de una variable x en dicho término por un término N. La notamos:

$$M\{x \leftarrow N\}$$

Esta operación nos sirve para darle semántica a la aplicación de funciones y es sencilla de definir, sin embargo debemos tener en cuenta algunos casos especiales.

α-equivalencia Dos terminos M y N que difieren solamente en el nombre de sus variables ligadas se dicen α-equivalentes. Esta relación es una relación de equivalencia. Técnicamente, la sustitución está definida sobre clases de α-equivalencia de términos

Captura de variables El primer problema se da cuando la sustitución que deseamos realizar sustituye una variable por otra con el mismo nombre que alguna de las variables ligadas de la expresión. Por ejemplo:

$$(\lambda z : \sigma.x)\{x \leftarrow z\} = \lambda z : \sigma.z$$

En estos casos, si realizamos la sustitución cambiariamos el significado de la expresión (en el caso mostrado, estariamos convirtiendo la función constante que devuelve x en la función identidad). Por esta razón debemos asegurarnos que cuando realizemos la operación $\lambda y : \sigma.M\{x \leftarrow N\}$, la variable ligada y sea renombrada de tal manera que **no** ocurra libre en N.

Entonces, teniendo en cuenta lo mencionado, definimos el comportamiento de la operación:

$$\begin{aligned} x\{x \leftarrow N\} &\stackrel{def}{=} N \\ a\{x \leftarrow N\} &\stackrel{def}{=} a \text{ si } a \in \{true, false\} \cup \mathcal{X} \backslash \{x\} \\ (if\ M\ then\ P\ else\ Q)\{x \leftarrow N\} &\stackrel{def}{=} if\ M\{x \leftarrow N\}\ then\ P\{x \leftarrow N\}\ else\ Q\{x \leftarrow N\} \\ (M_1\ M_2)\{x \leftarrow N\} &\stackrel{def}{=} M_1\{x \leftarrow N\}\ M_2\{x \leftarrow N\} \\ (\lambda y : \sigma.M)\{x \leftarrow N\} &\stackrel{def}{=} \lambda y : \sigma.M\{x \leftarrow N\}\ x \neq y,\ y \notin\ FV(N) \end{aligned}$$

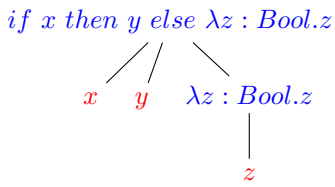
La condición $x \neq y,\ y \notin FV(N)$ está para que efectivamente no se produzca la situación mencionada en el parrafo anterior. Y **siempre** puede cumplirse, solo hay que renombrar las variables de manera apropiada.

Árbol sintáctico

Dada una expresión M, su árbol sintáctico es un árbol que tiene como raíz a M y como hijos de la raíz a todos los subtérminos válidos de la expresión.

Ejemplos El árbol sintáctico de *true* es:

El árbol sintáctico de $if\ x\ then\ y\ else\ \lambda z : Bool.z$ es:



1.2.2. Sistema de tipado

El sistema de tipado es un sistema formal de deducción (o derivación) que utiliza axiomas y reglas de tipado para caracterizar un subconjunto de los términos. A estos términos los llamamos **términos tipados**.

Como dijimos, vamos a estudiar lenguajes de cálculo lambda tipado, por lo que para que una expresión sea considerada una expresión válida del lenguaje no solo debe ser sintácticamente correcta sino que debemos poder inferir su tipo a través del sistema de tipado que definamos. Y si no es posible inferir el tipo de una expresión con el sistema dado, entonces no la consideraremos una expresión válida del lenguaje.

Contexto de tipado : Es un conjunto de pares $x_i : \sigma_i$, anotado $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ que nos indica los tipos de cada variable de un programa.

Dado un contexto de tipado Γ, un **juicio de tipado** es una expresion $\Gamma \triangleright M : \sigma$ que se lee "el término M tiene tipo σ asumiendo el contexto de tipado Γ".

Axiomas de tipado de λ^b

$$\frac{}{\Gamma \triangleright true : Bool}(\text{T-True}) \qquad \frac{}{\Gamma \triangleright false : Bool}(\text{T-False})$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \triangleright x : \sigma}(\text{T-Var})$$

Los axiomas **T-True** y **T-False** nos dicen, que no importa el contexto en el que se encuentren los valores *true* y *false*, respectivamente, ambos valores serán de tipo *Bool*. El axioma **T-Var**, nos dice que una variable libre x es de σ en un contexto Γ entonces el par $x : \sigma$ se encuentra en Γ

Reglas de tipado de λ^b

$$\frac{\Gamma \triangleright M : Bool \quad \Gamma \triangleright P : \sigma \quad \Gamma \triangleright Q : \sigma}{\Gamma \triangleright if\ M\ then\ P\ else\ Q : \sigma}(\text{T-If})$$

$$\frac{\Gamma, x : \sigma \triangleright M : \tau}{\Gamma \triangleright \lambda x : \sigma . M : \sigma \to \tau}(\text{T-Abs}) \qquad \frac{\Gamma \triangleright M : \sigma \to \tau \quad \Gamma \triangleright N : \sigma}{\Gamma \triangleright M\ N : \tau}(\text{T-App})$$

T-If nos dice que si $if\ M\ then\ P\ else\ Q$ es de tipo σ en Γ, entonces M es de tipo $Bool$ y P y Q son de tipo σ en Γ.

T-Abs indica que si $\lambda x : \sigma . M$ es de tipo $\sigma \to \tau$ en Γ, entonces M es de tipo τ y x en de tipo σ en Γ.

T-App significa que si $M\ N$ es de tipo $\sigma \to \tau$ en Γ, entonces M es de tipo τ en el contexto $\Gamma, x : \sigma$. Este es el contexto formado por la unión disjunta entre Γ y $x : sigma$, o en castellano, el contexto que remplaza el tipo de x en $Gamma$ por σ.

Resultados básicos

Si $\Gamma \triangleright M : \sigma$ puede derivarse usando los axiomas y reglas de tipados decimos que el juicio es **derivable**. Además, si el juicio se puede derivar para algún Γ y σ, entonces decimos que M es **tipable**.

Unicidad de tipos Si $\Gamma \triangleright M : \sigma$ y $\Gamma \triangleright M : \tau$ son derivables, entonces $\sigma = \tau$

Weakening + Strengthening Si $\Gamma \triangleright M : \sigma$ es derivable y $\Gamma \cap \Gamma'$ contiene a todas las variables libres de M, entonces $\Gamma' \triangleright M : \sigma$

Sustitución Si $\Gamma, x : \sigma \triangleright M : \tau$ y $\Gamma \triangleright N : \sigma$ son derivables, entonces $\Gamma \triangleright M\{x \leftarrow N\} : \tau$ es derivable.

Demostración de juicios de tipado

Dado un sistema tipado, queremos ver si un juicio de tipado es correcto. Para hacer esto, iremos aplicando, al juicio, las reglas del sistema hasta llegar a sus axiomas o hasta llegar a una contradicción o incertidumbre. Si pasa lo primero, entonces el juicio es correcto, si pasa lo segundo, el juicio está mal.

1.2.3. Semántica operacional

Ya definimos cuales serán los términos y expresiones válidas de nuestro lenguaje. El siguiente paso, es definir algún mecanismo que nos permita inferir el significado o **valor** de un término.

Para lograr este objetivo definimos lo que se llama **semántica operacional**, un mecanismo que interpreta a los **términos como estados** de una máquina abstracta y define una **función de transición** que indica, dado un estado, cual es el siguiente.

De esta forma, el significado de un término M es el estado final que alcanza la máquina al comenzar con M como estado inicial.

Tenemos dos formas de definir la semántica:

- **Small-step**: La función de transición describe un paso de computación, descomponiendo los términos compuestos en términos más simples y especificando el orden el que deben ser reducidos.
- **Big-step** (o **Natural Semantics**): La función de transición, en un paso, evalúa el termino a su resultado.

Nosotros vamos a usar la primer opción. Y la formulamos a través de **juicios de evaluación**

$$M \to N$$

que se leen "*el término M reduce, en un paso, al término N*".

Para establecer el significado de estos juicios, vamos a definir **axiomas de evaluación** y **reglas de evaluación**. Los axiomas nos indicarán cuales juicios de evaluación son siempre derivables y las reglas nos dirán que juicios son derivables dado un contexto. Las reglas de la semántica asumen que las expresiones están bien tipadas.

Expresiones Booleanas

Los valores de las expresiones booleanas son:

$$V \ ::= \ true \mid false$$

y son usados para reducir el término $if\ M_1\ then\ M_2\ else\ M_3$ mediante los siguientes axiomas:

$$\frac{}{if\ true\ then\ M_1\ else\ M_2 \to M_1}\text{(E-IfTrue)}$$

$$\frac{}{if\ false\ then\ M_1\ else\ M_2 \to M_2}\text{(E-IfFalse)}$$

$$\frac{M_1 \to M_1'}{if\ M_1\ then\ M_2\ else\ M_3 \to if\ M_1'\ then\ M_2\ else\ M_3}\text{(E-If)}$$

Estas reglas nos indican que dado un término del tipo $if\ M_1\ then\ M_2\ else\ M_3$, si $M_1 = true$, entonces podemos remplazar la expresión por M_2, si $M_1 = false$ entonces podemos remplazar la expresión por M_3 y si M_1 es una expresión reducible a M_1', entonces podemos remplazar la expresión por $if\ M_1'\ then\ M_2\ else\ M_3$.

Con estas reglas definimos la estrategia de evaluación del condicional que se corresponde el orden habitual en lenguajes de programación:

1. Primero evaluamos la guarda del condicional
2. y una vez que la guarda sea un valor, evaluamos la expresión del *then* o del *else* según corresponda.

Propiedades

Determinismo Si $M \to M'$ y $M \to M''$ entonces $M' = M''$, esto quiere decir que el valor que representa M no cambia con las reducciones que le apliquemos.

Valores en forma normal Una **forma normal** es un término que no puede evaluarse más. Consideraremos que terminamos de evaluar un término cuando conseguimos su forma normal.

Todos los valores tiene una forma normal, sin embargo hay que tener en cuenta que como estamos definiendo un lenguaje tipado, habrá formas normales que no representen ningún valor.

Evaluación en muchos pasos

El juicio de **evualuación de muchos pasos** $\twoheadrightarrow$ es la clausura reflexiva, transitiva de $\to$. Es decir, la menor relación tal que:

1. Si $M \to M'$, entonces $M \twoheadrightarrow M'$
2. $M \twoheadrightarrow M$ para todo M
3. Si $M \twoheadrightarrow M'$ y $M' \twoheadrightarrow M''$, entonces $M \twoheadrightarrow M''$

Unicidad de formas normales Si $M \twoheadrightarrow U$ y $M \twoheadrightarrow V$ con U y V formas normales, entonces $U = V$

Terminación Para todo M existe una forma normal N tal que $M \twoheadrightarrow N$

1.2.4. Semántica operacional de λ^b

En la sección 1.2.3 definimos el comportamiento de las expresiones booleanas, sin embargo, nos falta definir como reducir términos del tipo $\lambda x : \sigma.M$ y $M\ N$.

Lo primero a tener en cuenta, es que vamos a considerar a los términos de $\lambda x : \sigma.M$ como valores, sin si M es reducible o nó. Entonce, nuestro conjunto de valores del lenguaje sería:

$$V\ ::=\ true \mid false \mid \lambda x : \sigma.M$$

Por lo que todo término bien tipado y cerrado (sin variables libres) evalúa a alguna de estos términos. Si es de tipo $Bool$ evalúa a $true$ o $false$, si es de tipo $\sigma \to \tau$ evalúa a $\lambda x : \sigma M$. A las reglas y axiomas definidos para los tipos booleanos agregamos los siguientes:

$$\frac{M_1 \to M_1'}{M_1\ M_2 \to M_1'\ M_2}(\text{E-App1 / } \mu)$$

$$\frac{M_2 \to M_2'}{V_1\ M_2 \to V_1\ M_2'}(\text{E-App2 / } v)$$

$$\frac{}{(\lambda x : \sigma.M)\ V \to M\{x \leftarrow V\}}(\text{E-App2 / } \beta)$$

Estado de error Es un estado que **no es** un valor pero en el que la computación está trabada. Representa el estado en el cual el sistema de runtime de una implementación real generaría una excepción.

El sistema de tipado, nos garantiza que si un término cerrado está bien tipado entonces evalúa a un valor.

Corrección La corrección de un término nos asegura dos cosas: **Progreso** y **Preservación**.

El **progreso** asegura que si M es un término cerrado y bien tipado, entonces M es un valor o existe M' tal que $M \to M'$. En otras palabras, nos asegura que la evaluación no puede trabarse para términos cerrados y bien tipados que no son valores. Y si un programa termina, entonces nos devuelve un valor.

La **preservación** asegura que la evaluación de un término M cerrado y bien tipado preserva tipos. Es decir, no importa cuanta veces se reduzca M, el término resultante siempre es del tipo original.

$$\text{Si } \Gamma \triangleright M : \sigma \text{ y } M \to N \text{ entonces } \Gamma \triangleright N : \sigma$$

Extendiendo el lenguaje Cuando queramos extender el lenguaje, debemos realizar los mismos pasos que realizamos para definir el lenguaje λ^b, esto es decir, agregar el nuevo tipo al conjunto de tipos, definir los términos de ese tipo, sus reglas de tipado y sus reglas semánticas, asegurándonos de que las nuevas reglas no interfieran con las ya definidas. Esto es, no debemos definir reglas que las contradigan o que den nuevas formas de inferir algo que ya se podía inferir con otras reglas.

En el apéndice de extensiones, muestro algunas extensiones que servirán como ejemplo.

Macros Hay expresiones del lenguaje que usaremos con demasiada frecuencia, para estas expresiones podremos definir macros que simplificaran su escritura. Algunos ejemplos son:

$$Id_{Bool} \stackrel{def}{=} \lambda x : Bool.x$$

$$and \stackrel{def}{=} \lambda x : Bool.\lambda y : Bool.if\ x\ then\ y\ else\ false$$

1.2.5. Extensión Naturales (λ^{bn})

Tipos

$$\sigma, \tau\ ::=\ Bool \mid Nat \mid \sigma \to \tau$$

Términos

$$M \ ::= \ \ldots \mid 0 \mid succ(M) \mid pred(M) \mid isZero(M)$$

Los términos significan:

- $succ(M)$: evaluar M hasta arrojar un número e incrementarlo.
- $pred(M)$: evaluar M hasta arrojar un número y decrementar.
- $iszero(M)$: evaluar M hasta arrojar un número, luego retornar $true/false$ según sea cero o no.

Axiomas y reglas de tipado

$$\frac{}{\Gamma \triangleright 0 : Nat}(\text{T-Zero})$$

$$\frac{\Gamma \triangleright M : Nat}{\Gamma \triangleright succ(M) : Nat}(\text{T-Succ}) \qquad \frac{\Gamma \triangleright M : Nat}{\Gamma \triangleright pred(M) : Nat}(\text{T-Pred})$$

$$\frac{\Gamma \triangleright M : Nat}{\Gamma \triangleright isZero(M) : Bool}(\text{T-IsZero})$$

Valores

$$V \ ::= \ \ldots \mid \underline{n} \text{ donde } \underline{n} \text{ abrevia } succ^n(0)$$

Axiomas y reglas de evaluación

$$\frac{M_1 \to M_1'}{succ(M_1) \to succ(M_1')}(\text{E-Succ})$$

$$\frac{}{pred(0) \to 0}(\text{E-PredZero}) \qquad \frac{}{pred(succ(\underline{n})) \to \underline{n}}(\text{E-PredSucc})$$

$$\frac{M_1 \to M_1'}{pred(M_1) \to pred(M_1')}(\text{E-Pred})$$

$$\frac{}{isZero(0) \to true}(\text{E-IsZeroZero}) \qquad \frac{}{isZero(succ(\underline{n})) \to false}(\text{E-isZeroSucc})$$

$$\frac{M_1 \to M_1'}{isZero(M_1) \to isZero(M_1')}(\text{E-isZero})$$

1.2.6. Simulación de lenugajes imperativos

Los lenguajes imperativos se caracterizan por su capacidad de asignar y modificar variables dentro de un programa. Esto lo hace a través de comandos, expresiones del lenguaje cuyo objetivo es crear un efecto sobre el estado de la computadora.

Queremos extender el lenguaje λ para que poder simular comandos y efectos sobre la memoria.

En un lenguaje imperativo **todas** las variables son **mutables**, es decir, que hay operaciones que pueden modificar su valor. Para lograr esto, hace uso de tres operaciones básicas:

- **Asignación:** $x := M$ almacena en la referencia x el valor de M
- **Alocación (Reserva de memoria)** $ref\ M$ genera una referencia fresca cuyo contenido es el valor de M

- **Derreferenciación (Lectura):** $!x$ sigue la referencia x y retorna su contenido.

Notemos que una vez que agreguemos estas expresiones al lenguaje lambda, este dejará de ser un lenguaje funcional **puro** (un lenguaje en el todas sus expresiones carecen de efecto).

Nos gustaría agregar las expresiones mencionadas a nuestro lenguaje, para esto primero debemos asignarles un tipo.

Asignacion Lo primero que debemos tener en cuenta, es que la igualdad ($x := M$) es una expresión de la cual no nos interesa saber su valor sino el efecto que tiene la misma sobre el contexto. Entonces, debemos definir un nuevo tipo que nos permita identificar cuando una expresión evaluada solo fue evaluada para generar un efecto. Nombraremos este tipo $Unit$ y su conjunto de valores será solo el valor $unit$. Podemos decir que este tipo cumple el rol de *void* en C.

Macro punto y coma (;) En lenguajes con efectos laterales, como el que estamos definiendo, esta macro nos servirá para definir el orden de evaluación de varias expresiones en **secuencia**.

$$M_1; M_2 \stackrel{def}{=} (\lambda x : Unit.M_2)\ M_1 \qquad x \notin FV(M_2)$$

Por como definimos las reglas semánticas del lenguaje, esto significa que primero se evalúa M_1 y luego M_2.

Extensión con Referencias (λ^{bnu})

Referencias Una referencia es una abstracción de una porción de memoria que se encuentra en uso. Usaremos el tipo $Ref\ \sigma$ para diferenciar las expresiones que representan referencias.

Representación Representaremos las posiciones con **direcciones simbólicas** o *locations* usando etiquetas l, l_1 y definiremos a la **memoria** o *store* como una función parcial μ que dada una dirección nos devuelve el valor almacenado en ella. Y notaremos:

- $\mu[l \to V]$ es el store resultante de **pisar** $\mu(l)$ con V.
- $\mu \oplus (l \to V)$ es el **store extendido** resultante de ampliar μ con una nueva asociación $l \to V$ asumiendo que $l \notin Dom(\mu)$.

Uso en semántica Ahora necesitamos una forma de usar estas nuevas definiciones en nuestras evaluaciones, por lo que agregaremos las etiquetas al conjunto de valores y, a partir de ahora, los juicios de evaluación, tendrán la siguiente forma:

$$M|\mu \to M'|\mu'$$

Esto significa que una expresión M reduce a M' y que afecta a μ de tal forma que pasa a ser μ', así reflejaremos los cambios de estado de la memoria.

Notemos que, a pesar de que agregamos las etiquetas l como términos y valores, estás son solo producto de la formalización y **no** se pretende que sean usadas por el programador.

Uso en tipado Con la posibilidad de modificar la memoria durante la ejecución de un programa, se hace necesaria la definición de un contexto que nos permita inferir el tipo del valor almacenado en las posiciones usadas. Introducimos el **contexto de tipado** Σ para direcciones como una función parcial de direcciones a tipos. Y los juicios de tipado serán de la siguiente forma:

$$\Gamma|\Sigma \triangleright M : \sigma$$

Indicando esto, que M es de tipo σ en el contexto Γ cuando el estado de la memoria se corresponde con el contexto de tipado Σ.

La extension

Tipos

$$\sigma, \tau \ ::= \ Bool \ | Nat \mid Unit \mid Ref \ \sigma \mid \sigma \to \tau$$

Términos

$$M \ ::= \ \ldots \mid unit \mid ref \ M \mid !M \mid M \ := \ N \mid l$$

Axiomas y reglas de tipado

$$\frac{}{\Gamma | \Sigma \triangleright unit : Unit} (\text{T-Unit}) \qquad \frac{\Gamma | \Sigma \triangleright M_1 : \sigma}{\Gamma | \Sigma \triangleright ref \ M_1 : Ref \ \sigma} (\text{T-Ref})$$

$$\frac{\Gamma | \Sigma \triangleright M_1 : Ref \ \sigma}{\Gamma \triangleright !M_1 : \sigma} (\text{T-DeRef})$$

$$\frac{\Gamma | \Sigma \triangleright M_1 : Ref \ \sigma \quad \Gamma | \Sigma \triangleright M_2 : \sigma}{\Gamma \triangleright M_1 \ := \ M_2 : Unit} (\text{T-Assing})$$

$$\frac{\Sigma(l) = \sigma}{\Gamma | Signa \triangleright l : Ref \ \sigma} (\text{T-Loc})$$

Valores

$$V \ ::= \ \ldots \mid unit \mid l$$

Axiomas y reglas semánticas

$$\frac{M_1 | \mu \to M_1' | \mu'}{M_1 \ M_2 | \mu \to M_1' \ M_2 | \mu'} (\text{E-App1}) \qquad \frac{M_2 | \mu \to M_2' | \mu'}{V_1 \ M_2 | \mu \to V_1 \ M_2' | \mu'} (\text{E-App2})$$

$$\frac{}{(\lambda x : \sigma . M) \ V | \mu \to M\{x \leftarrow V\} | \mu'} (\text{E-AppAbs})$$

$$\frac{M_1 | \mu \to M_1' | \mu'}{!M_1 | \mu \to !M_1' | \mu'} (\text{E-DeRef}) \qquad \frac{\mu(l) = V}{!l | \mu \to V | \mu} (\text{E-DerefLoc})$$

$$\frac{M_1 | \mu \to M_1' | \mu'}{M_1 \ := \ M_2 | \mu \to M_1' \ := \ M_2 | \mu'} (\text{E-Assign1})$$

$$\frac{M_2 | \mu \to M_2' | \mu'}{V \ := \ M_2 | \mu \to V \ := \ M_2' | \mu'} (\text{E-Assign2})$$

$$\frac{}{l \ := \ V | \mu \to unit | \mu[l \to V]} (\text{E-Assign})$$

$$\frac{M_1 | \mu \to M_1' | \mu'}{ref \ M_1 | \mu \to ref \ M_1' | \mu'} (\text{E-Ref}) \qquad \frac{l \notin Dom(\mu)}{ref \ V | \mu \to l | \mu \oplus (l \to V)} (\text{E-RefV})$$

Correción de tipos en un lenguaje con referencias

Al agregar referencia, hay consecuencias. Una de ellas es que no todo término cerrado y bien tipado termina. Por lo que debemos reformular las definiciones de corrección del lenguaje, es decir, debemos indicar que significa el **progreso** y la **preservación** cuando hay referencias.

Preservación La preservación nos aseguraba que no importa cuantas veces reduzcamos una expresión, esta debería mantener su tipo. Sin embargo, con las asignaciones podemos cambiar el tipo de ciertos valores durante la ejecución de un programa, lo que implica que la expresión podria cambiar su tipo. Para definir la preservación precisamos una noción de compatibilidad entre el store y el contexto de tipado que nos permita asegurar que si los valores no cambian su tipo, entonces la expresión mantiene su tipo.

Decimos que $\Gamma|\Sigma \triangleright \mu$ si y solo si $Dom(\Sigma) = Dom(\mu)$ y $\Gamma|Sigma \triangleright \mu(l) : \Sigma(l)$ para todo $l \in Dom(\mu)$. Es decir, μ es compatible con Σ si ambas funciones tiene el mismo dominio, y es cierto que los tipos de cada etiqueta de μ coinciden con los tipos que se les asignó en Σ.

Entonces definimos la preservación de la siguiente manera:

> **Si** $\Gamma|\Sigma \triangleright M : \sigma$ y $M|\mu \to N|\mu'$ y $\Gamma|\Sigma \triangleright \mu$ **entonces existe un** Σ' **que contiene a** Σ **tal que** $\Gamma|\Sigma' \triangleright N : \sigma$ y $\Gamma|\Sigma' \triangleright \mu'$

La nueva definición nos dice que dada una expresión M de tipo σ y un contexto $\Gamma\Sigma$ compatible con μ, si pasa que cuando reducimos $M|\mu \to N|\mu'$, mu' es compatible con Σ', entonces la reducción tendrá el mismo tipo que M.

Para que μ' sea compatible con Σ' puede haber dos posibilidad: O qué $\mu' = \mu$ y $\Sigma = \Sigma'$ o qué μ' sea una extensión de μ, es decir que se haya creado una referencia nueva, en cuyo caso ninguno de los tipos fue modificado y Σ' es Σ extendido con el tipo de la nueva referencia.

Progreso El progreso nos aseguraba que dada una expresión, entonces su ejecución termina en un valor o no termina. Para el nuevo lenguaje, hay que tener en cuenta el contexto de tipado:

> Si M es cerrado y bien tipado en un contexto de tipado de memoria Σ, entonces
>
> - M es un valor
> - o bien para cualquier memoria μ que sea compatible con Σ, existe M' y μ' tal que $M|\mu \to M'|\mu'$

Esto quiere decir que solo se puede asegurar progreso cuando μ es compatible con Σ.

1.2.7. Extensión con recursión

Queremos dar al lenguaje λ, la capacidad de interpretar expresiones recursivas. Definimos, entonces, la función $fixM$ que dado para función M devuelve el punto fijo de dicha función, es decir, un valor x tal que Mx evalúa a x.

Veamos un ejemplo, supongamos que tenemos la función $f(n) = \textbf{If } n = 0 \textbf{ then } 1 \textbf{ else } n * f(n-1)$. Cuando evaluamos f en 0, obtenemos su definición para el valor O, cuando la evaluamos en 1, la definimos para 0 y 1, con cada valor que tengamos, la iremos definiendo para ese valor y para todos los anteriores.

Podríamos pensar que cuando evaluamos $n \to$ ínf, obtenemos una función que se define para todos los valores naturales, es decir obtenemos la función factorial, propiamente dicha.

Términos

$$M := \ldots \mid fix\ M$$

Regla de tipado

$$\frac{\Gamma \triangleright M : \sigma \to \sigma}{\Gamma \triangleright fix\ M : \sigma}(\text{T-Fix})$$

Reglas de evaluación

$$\frac{M_1 \to M_1'}{fix\ M_1 \to fix\ M_1'}(\text{E-Fix})$$

$$\frac{}{fix\ (\lambda x : \sigma . M) \to M\{x \leftarrow fix\ \lambda x : \sigma . M\}}(\text{E-FixBeta})$$

Apéndice A

Programación funcional en Haskell

Tipos elementales

```
1               -- Int          Enteros
'a'             -- Char         Caracteres
1.2             -- Float        Números de punto flotante
True            -- Bool         Booleanos
[1,2,3]         -- [Int]        Listas
(1, True)       -- (Int, Bool)  Tuplas, pares
length          -- [a] -> Int   Funciones
length [1,2,3]  -- Int          Expresiones
\x -> x         -- a -> a       Funciones anónimas
```

Guardas

```
signo n | n >= 0    = True
        | otherwise = False
```

Pattern Matching

```
longitud [] = 0
longitud (x:xs) = 1 + (longitud xs)
```

Polimorfismo paramétrico

```
todosIguales :: Eq a => [a] -> Bool
todosIguales [] = True
todosIguales [x] = True
todosIguales (x:y:xs) = x == y && todosIguales(y:xs)
```

Clases de tipo

```
Eq a    -- Tipos con comparación de igualdad
Num a   -- Tipos que se comportan como los números
Ord a   -- Tipos orden
Show a  -- Tipos que pueden ser representados como strings
```

Definición de listas

```haskell
[1,2,3,4,5]                 -- Por extensión
[1 .. 4]                    -- Secuencias aritméticas
[ x | x <- [1..], esPar x ] -- Por compresión

-- Las listas pueden ser infinitas, solo hay que tener cuidado cuando
↪  las usamos. Ejemplo de lista infinita:

infinitosUnos :: [Int]
infinitosUnos = 1 : infinitosUnos

puntosDelCuadrante :: [(Int, Int)]
puntosDelCuadrante = [ (x, s-x) | s <- [0..], x <-[0..s] ]
```

Funciones de alto orden

```haskell
mejorSegun :: (a -> a -> Bool) -> [a] -> a
mejorSegun _ [x] = x
mejorSegun f (x : xs) | f x (mejorSegun f xs) = x
                      | otherwise = mejorSegun f xs
```

A.0.1. Otros tipos útiles

Formula

```haskell
data Formula = Proposicion String | No Formula
                                  | Y Formula Formula
                                  | O Formula Formula
                                  | Imp Formula Formula

foldFormula :: (String -> a) -> (Formula -> a) ->
               (Formula -> Formula -> a) -> (Formula -> Formula -> a)
               -> (Formula -> Formula -> a) -> Formula -> a
foldFormula fp fn fy fo fImp form = case form of :
                Proposicion s -> fp s
                No sf -> fn (rec sf)
                Y sf1 sf2 -> fy (rec sf1) (rec sf2)
                O sf1 sf2 -> fo (rec sf1) (rec sf2)
                Impl sf1 sf2 -> fImpl (rec sf1) (rec sf2)
        where rec = foldForm fp fn fy fo fImp
```

Rosetree

```haskell
data Rosetree = Rose a [Rosetree]
-- Hay varias formas de definir el fold para esta estructura
foldRose :: (a -> [b] -> b) -> Rosetree a -> b
foldRose f ( Rose x l ) = f x ( map ( foldRose f ) l )

foldRose2 :: ( a -> c -> b) -> ( b -> c -> c ) -> c
            -> Rosetree a -> b
foldRose2 g f z (Rose x l) =
          g x ( foldr f z ( map ( foldRose g f z ) l ) )
```

Apéndice B

Extensiones del lenguaje λ^b

B.0.1. Extensión con Registros $\lambda^{\cdots r}$

Tipos

$$\sigma, \tau \ ::= \ \ldots \mid \{l_i : \sigma_i \ ^{i \in 1..n}\}$$

El tipo $\{l_i : \sigma_i^{i \in 1..n}\}$ representan las estructuras con n atributos tipados, por ejemplo: $\{nombre : String, edad : Nat\}$

Términos

$$M \ ::= \ \ldots \mid \{l_i = M_i \ ^{i \in 1..n}\} \mid M.l$$

Los términos significan:

- El registro $\{l_i = M_i \ ^{i \in 1..n}\}$ evalua $\{l_i = V_i \ ^{i \in 1..n}\}$ donde V_i es el valores al que evalúa M_i para $i \in 1..n$.
- $M.l$: Proyecta el valor de la etiqueta l del registro M

Axiomas y reglas de tipado

$$\frac{\Gamma \triangleright M_i : \sigma_i \text{ para cada } i \in 1..n}{\Gamma \triangleright \{l_i = M_i \ ^{i \in 1..n}\} : \{l_i : \sigma_i \ ^{i \in 1..n}\}} \text{(T-RCD)}$$

$$\frac{\Gamma \triangleright \{l_i = M_i \ ^{i \in 1..n}\} : \{l_i : \sigma_i \ ^{i \in 1..n}\} \qquad j \in 1..n}{\Gamma \triangleright M.l_j : \sigma_j} \text{(T-Proj)}$$

Valores

$$V \ ::= \ \ldots \mid \{l_i = V_i \ ^{i \in 1..n}\}$$

Axiomas y reglas de evaluación

$$\frac{j \in 1..n}{\{l_i = V_i \ ^{i \in 1..n}\}.l_j \to V_j} \text{(E-ProjRcd)}$$

$$\frac{M \to M'}{M.l \to M'.l} \text{(E-Proj)}$$

$$\frac{M_j \to M'_j}{\{l_i = V_i \ ^{i \in 1..j-1}, l_j = M_j, l_i = M_i \ ^{i \in j+1..n}\} \to \{l_i = V_i \ ^{i \in 1..j-1}, l_j = M'_j, l_i = M_i \ ^{i \in j+1..n}\}} \text{(E-RCD)}$$

B.0.2. Extensión con Declaraciones Locales ($\lambda^{\dots let}$)

Con esta extensión, agregamos al lenguaje el término $let\ x : \sigma = M\ in\ N$, que evalúa M a un valor, liga x a V y, luego, evalúa N. Este término solo mejora la legibilidad de los programas que ya podemos definir con el lenguaje hasta ahora definido.

Términos

$$M\ ::=\ \dots \mid let\ x : \sigma = M\ in\ N$$

Axiomas y reglas de tipado

$$\frac{\Gamma \rhd M : \sigma_1 \qquad \Gamma, x : \sigma_1 \rhd N : \sigma_2}{\Gamma \rhd let\ x : \sigma_1 = M\ in\ N : \sigma_2}(\text{T-Let})$$

Axiomas y reglas de evaluación

$$\frac{M_1 \to M_1'}{let\ x : \sigma = M_1\ in\ M_2 \to let\ x : \sigma = M_1'\ in\ M_2}(\text{E-Let})$$

$$\frac{}{let\ x : \sigma = V_1\ in\ M_2 \to M_2\{x \leftarrow V_1\}}(\text{E-LetV})$$

Construcción *let* recursivo (Letrec)

Una construcción alternativa para definir funciones recursivas es

$$letrec\ f : \sigma \to \sigma = \lambda x : \sigma.M\ in\ N$$

Y *letRec* se puede definir en base a *let* y fix (definido en 1.2.7) de la siguiente forma:

$$let\ f : \sigma \to \sigma = (fix\ \lambda f : \sigma \to \sigma.\lambda x : \sigma.M)\ in\ N$$

B.0.3. Extensión con tuplas

Tipos

$$\sigma, \tau\ ::= \dots \mid \sigma \times \tau$$

Términos

$$M,\ N\ ::=\ \dots \mid <M, N> \mid \pi_1(M) \mid \pi_2(M)$$

Axiomas y reglas de tipado

$$\frac{\Gamma \rhd M : \sigma \qquad \Gamma \rhd N : \tau}{\Gamma \rhd <M, N> : \sigma \times \tau}(\text{T-Tupla})$$

$$\frac{\Gamma \rhd M : \sigma \times \tau}{\Gamma \rhd \pi_1(M) : \sigma}(\text{T-}\pi_1) \qquad \frac{\Gamma \rhd M : \sigma \times \tau}{\Gamma \rhd \pi_2(M) : \tau}(\text{T-}\pi_2)$$

Axiomas y reglas de evaluación

$$\frac{M \to M'}{<M, N> \to <M', N>}(\text{E-Tuplas}) \qquad \frac{N \to N'}{<V, N> \to <V, N'>}(\text{E-Tuplas1})$$

$$\frac{M \to M'}{\pi_1(M) \to \pi_1(M')}(\text{E-}\pi_1) \qquad \frac{}{\pi_1(<V_1, V_2>) \to V_1}(\text{E-}\pi_1')$$

$$\frac{M \to M'}{\pi_2(M) \to \pi_2(M')}(\text{E-}\pi_2) \qquad \frac{}{\pi_2(<V_1, V_2>) \to V_2}(\text{E-}\pi_2')$$