Paradigmas de Lenguajes de Programación

Zamboni, Gianfranco

6 de marzo de 2018

Índice

1. Introducción

Este apunte lo hice a partir de las clases de PLP en el Verano del 2018.

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.

1.1. Aspectos del lenguaje

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

1.1.1. 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).

1.1.2. 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.

1.2. Paradigmas

1.2.1. 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.

1.2.2. 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.

1.2.3. 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.

1.2.4. 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.

Parte I
Parádigma Funcional

2. 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.

2.0.1. 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.

2.1. 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,) 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.

2.1.1. 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`.

2.1.2. 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 tipable `e` y un tipo `A`, determinar si `e :: A` o no.

2.2. 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`)

2.2.1. 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.

2.3. 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.

2.3.1. 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.

2.3.2. 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)`.

2.3.3. 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 $x \in S$.

2.4. 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.

2.5. Tipos algebraicos

2.5.1. 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.

2.5.2. 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**.

2.5.3. 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.

2.5.4. 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
```

2.5.5. 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.

2.5.6. 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.

2.6. 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)
```

2.7. 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:

2.7.1. 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.

2.7.2. 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

2.7.3. 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) []
```

2.7.4. 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.

2.7.5. 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.

2.7.6. 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.

3. 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.

3.1. 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.

3.1.1. 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**.

3.1.2. 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.

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

3.1.3. 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:

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

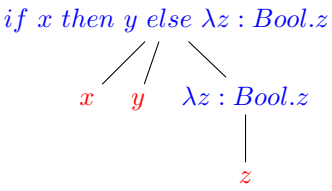
La condición $x \neq y,\ y \notin\ FV(N)$ está para que efectivamente no se produzca la situación mencionada en el parrafo anterior. Y **siempre** puede cumplirse, solo hay que renombrar las variables de manera apropiada.

3.1.4. 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:



3.2. 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 $\Gamma \rhd M : \sigma$ que se lee "el término M tiene tipo σ asumiendo el contexto de tipado Γ".

3.2.1. Axiomas de tipado de λ^b

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

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

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

3.2.2. Reglas de tipado de λ^b

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

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

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.

3.2.3. Resultados básicos

Si $\Gamma \rhd M : \sigma$ 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 $\Gamma \rhd M : \sigma$ y $\Gamma \rhd M : \tau$ son derivables, entonces $\sigma = \tau$

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

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

3.2.4. 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.

3.3. 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.

3.3.1. 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.

3.3.2. 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.

3.3.3. 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 $M \twoheadrightarrow N$

3.4. Semántica operacional de λ^b

En la sección 3.3.1 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:

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

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

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

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.

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

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:

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

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

3.5. 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

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

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

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

Valores

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

Axiomas y reglas de evaluación

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

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

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

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

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

3.6. 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.

3.6.1. 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:

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

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.

3.6.2. 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

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

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

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

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

Valores

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

Axiomas y reglas semánticas

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

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

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

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

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

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

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

3.6.3. 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.

3.6.4. 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 $\Gamma|\Sigma \rhd \mu$ si y solo si $Dom(\Sigma) = Dom(\mu)$ y $\Gamma|Sigma \rhd \mu(l) : \Sigma(l)$ 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** $\Gamma|\Sigma \rhd M : \sigma$ y $M|\mu \to N|\mu'$ y $\Gamma|\Sigma \rhd \mu$ **entonces existe un** Σ' **que contiene a** Σ **tal que** $\Gamma|\Sigma' \rhd N : \sigma$ y $\Gamma|\Sigma' \rhd \mu'$

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.

3.6.5. 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.

3.7. 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

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

Reglas de evaluación

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

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

4. Inferencia de tipos

Queremos modificar el lenguaje de cálculo lambda para que las expresiones no necesiten las notaciones de tipos explicitas. Para esto debemos definir términos sin información de tipos en los que la información faltante pueda ser **inferida** de manera sencilla. Esto es, debemos convertir dichos términos en términos bien tipados del cálculo lambda sin ningún problema.

Este nuevo lenguaje, nos evitará la sobrecarga de tener que declarar y manipular todos los tipos al momento de escribir un programa. Sin embargo, debemos tener en cuenta que, durante la compilación de los mismos, hay que hacer la inferencia de tipos, es decir, el compilador se deberá encargar de pasar el lenguaje que definamos a uno lambda tipado antes de poder compilar el programa.

Términos

El lenguaje sin tipos tendrá todos los términos del lenguaje λ con el que estuvimos trabajando hasta ahora, con la diferencia de que si en ellos había una notación de tipo, entonces la obviamos:

$$\begin{aligned} M \ ::= \ & x \\ & \mid \ true \mid false \mid if \ M \ then \ P \ else \ Q \\ & \mid \ 0 \mid succ(0) \mid isZero(M) \\ & \mid \ \lambda x.M \mid M \ N \mid fix \ M \end{aligned}$$

Ahora, si bien la mayoría de los términos son iguales a los términos originales, necesitariamos alguna forma de convertir los términos del lambda cálculo a términos no tipados y viceversa. Para el primer caso, definimos la función Erase que dado un término del lambda cálculo, **elimina** las anotaciones de tipos de las abstracciones que contenga. Por ejemplos:

$$\text{Erase}(\lambda x : Nat.\lambda f : Nat \to Nat.f \ x) = \lambda x.\lambda f.f \ x$$

Chequeos de tipo Realizar el chequeo de tipo es determinar, para un término estándar (del lenguaje λ tipado) M, si existe Γ y σ tales que $\Gamma \rhd M : \sigma$ es derivable. Osea que nos indica si M es un término tipable o no.

Este chequeo es facil de realizar, ya que solo hay que seguir la estructura sintáctica de M para reconstruir una derivación de juicio.

Definición de inferencia

En cambio, con la inferencia de tipos, dado un término U sin notaciones de tipo, se trata hallar un término estándar (con anotaciones de tipos) M tal que:

1. $\Gamma \rhd M : \sigma$ para algún Γ y σ, y
2. $\text{Erase}(M) = U$

Lo que estamos diciendo es que queremos encontrar una expresión bien tipada M del lenguaje lambda que sea equivalente a U. Si encontramos este M, U será de tipo σ, sino U será una expresión no tipable en nuestro lenguaje.

4.0.1. Variables de tipo

Supongamos que tenemos la expresión $U = \lambda x.x$. En este caso, si queremos tipar U, nos damos cuenta que puede ser la función identidad de cualquier tipo. Como cualquiera de estas expresiones es igual de válida necesitamos escribir esto en nuestra solución, para eso usamos las **variables de tipo**.

Una **variables de tipo** s es una variable que representa una expresión de tipo arbitraria e indica que no importa por que expresión de tipo la remplacemos, tendremos una solución válida. Esto nos permitirá escribir que la expresión M resultante de inferir los tipos de U será $M = \lambda x : \mathbf{s}.x$ donde **s** puede ser cualquier tipo de nuestro lenguaje.

Debemos agregar esta nueva expresión a las expresiones de tipo del cálculo lambda:

$$\sigma \ ::= \ \mathrm{s} \mid Nat \mid Bool \mid \sigma \to \tau$$

4.1. Sustitución de tipos

Una función S de sustitución es una función que mapea variables de tipo en expresiones de tipo y puede ser aplicada a expresiones de tipos ($S\sigma$), términos (SM) y contextos de tipado ($S\Gamma$).

Describimos S usando la notación $\{\sigma_1/t_1, \ldots, \sigma_n/t_n\}$ indicando que la variable t_i debe ser remplazada por σ_1. Además, definimos el **conjunto soporte** de S al conjunto $\{t_1, \ldots, t_n\}$ como el conjunto que representa las variables que afecta S.

Por ejemplo, si $S = \{Bool/t\}$, entonces $S(\lambda x : \mathrm{t}.x) = \lambda x : Bool.x$ y el tipo soporte de S es $\{t\}$.

La sustitución cuyo soporte es $\varnothing$, es la **sustitución identidad**.

Si tenemos dos juicios de tipado $\Gamma \rhd M : \sigma$ y $\Gamma' \rhd M' : \sigma'$ tales que $\Gamma' \rhd M' : \sigma'$ es el resultado de aplicar alguna función de sustitución S a $\Gamma \rhd M : \sigma$, entonces decimos que $\Gamma' \rhd M' : \sigma'$ es instancia de $\Gamma \rhd M : \sigma$

Composición de sustituciones La composición de sustituciones de S y T, denotada $S \circ T$, es la sustitución que se comporta como sigue:

$$(S \circ T)(\sigma) = S(T\sigma)$$

Preorden de sustituciones Una sustitución S es **más general** que T si existe una sustitución U tal que $T = U \circ S$, es decir, si T es una instancia de S.

4.1.1. Unificación

El algoritmo de inferencia que vamos a proponer analiza un término (sin notaciones de tipos) a partir de sus subtérminos. Una vez obtenida la información para cada uno de los subtérminos debe determinar si la información de cada uno de ellos es consistente (**consistencia**)y, si lo es, sintetizar la información del término original a partir de esta (**Síntesis**).

Para realizar la síntesis debemos **compatibilizar** la información de tipos de cada subtérmino, por cada variable x del término tenemos que tomar los tipos que le asigno cada súbtermino y unificarlos. Es decir, debemos encontrar una sustitución S que nos permita remplazar los tipos que dió cada subexpresión por un tipo único. Veamos un ejemplo:

Sea $M = x\ y + x\ (y + 1)$, del primer súbtermino $x\ y$ ténemos que $x :: \text{s} \to \text{t}$ e $y :: \text{s}$, del subtérmino $x\ (y+1)$ tenemos que $x :: Nat \to \text{u}$ e $y :: Nat$. Ahora, x e y pueden tener un solo tipo en M, por lo que necesitamos una sustitución que nos permita unificar $\text{s} \to \text{t}$ con $Nat \to \text{u}$ y s con Nat. En este caso podemos definir $S = \{Nat/\text{s},\ \text{u}/\text{t}\}$, concluyendo que $x :: Nat \to \text{t}$ e $y : Nat$.

Ecuación de unificación Es una expresión de la forma $\sigma_1 \doteq \sigma_2$ cuya solución es una sustitución tal que $S\sigma_1 = S\sigma_2$. Por lo general tendremos un conjunto de ecuaciones de unificación y la solución a dicho conjunto será la sustitución que unifica todas las expresiones.

En el ejemplo anterior, las ecuaciones de unificación hubiesen sido $\{\text{s} \to \text{t} \doteq Nat \to \text{u},\ \text{s} \doteq Nat\}$

Diremos que una sustitución S es un **unificador más general (MGU)** de $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$, si es solución de ese conjunto y es más general que cualquier otra de sus soluciones.

Teorema Si $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$ tiene solución, entonces existe un MGU y además es único salvo renombre de variables.

4.1.2. Algoritmo de unificación de Martelli-Montanari

Dado un conjunto de ecuaciones de unifiación $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$, vamos a presentar un algoritmo no-deterministico que consiste en **reglas de simplificación** que reescriben conjuntos de pares de tipos a unificar (*goals*).

Las secuencias que terminan en un *goal* vacío son **exitosas**, el resto, son **fallidas**. Si una secuencia es exitosa, entonces los pasos en los que realizamos sustituciones serán soluciones parciales al problema y la composición de todas ellas será el MGU.

4.1.3. Reglas de reducción

1. **Descomposición**

   $\{\sigma_1 \to \sigma_2 \doteq \tau_1 \to \tau_2\} \cup G \mapsto \{\sigma_1 \doteq \tau_1,\ \sigma_2 \doteq \tau_2\} \cup G$

2. **Eliminación de par trivial**

   $\{Nat \doteq Nat\} \cup G \mapsto G$

   $\{Bool \doteq Bool\} \cup G \mapsto G$

   $\{\text{s} \doteq \text{s}\} \cup G \mapsto G$

3. **Swap** Si σ no es una variable,

   $\{\sigma \doteq \text{s}\} \cup G \mapsto \{\text{s} \doteq \sigma\} \cup G$

4. **Eliminación de variable** Si $s \notin FV(\sigma)$

   $\{\text{s} \doteq \sigma\} \cup G \mapsto_{\sigma/s} G[\sigma/s]$

5. **Falla**

   $\{\sigma \doteq \tau\} \cup G \mapsto$ `falla`, con $(\sigma, \tau) \in T \cup T^{-1}$ y $T = \{(Bool, Nat), (Nat, \sigma_1 \to \sigma_2), (Bool, \sigma_1 \to \sigma_2\}$. Acá, la notación T^{-1} se refiere al conjunto con cada tupla de T invertida.

6. **Occur Check** Si $s \neq \sigma$ y $s \in FV(\sigma)$

   $\{\text{s} \doteq \sigma\} \cup G \mapsto$ `falla`

4.1.4. Propiedades del algoritmo

El algoritmo de Martinelli-Montanari siempre termina. Sea G un conjunto de pares, entonces:

- Si G tiene un unificador, el algoritmo termina exitosamente y retorna un MGU.
- Si G no tiene un unificador, el algoritmo termina con `falla`.

4.1.5. Ejemplo de aplicación

$$\{(Nat \to r) \to (r \to u) \doteq t \to (s \to s) \to t\} \mapsto^1 \{Nat \to r \doteq t,\ r \to u \doteq (s \to s) \to t\}$$
$$\mapsto^3 \{t \doteq Nat \to r,\ r \to u \doteq (s \to s) \to t\} \mapsto^4_{Nat \to r/t} \{r \to u \doteq (s \to s) \to (Nat \to r)\}$$
$$\mapsto^1 \{r \doteq (s \to s),\ u \doteq Nat \to r\} \mapsto^4_{s \to s/r} \{u \doteq Nat \to (s \to s)\} \mapsto^4_{Nat \to (s \to s)/u} \varnothing$$

Entonces, el MGU es
$\{Nat \to (s \to s)/u\} \circ \{s \to s/r\} \circ \{Nat \to r/t\} = \{Nat \to (s \to s)/u,\ s \to s/r,\ Nat \to (s \to s)/t\}$

4.2. Función de inferencia $\mathbb{W}$

Vamos a definir una función $\mathbb{W}$ que dada una expresión U sin notación de tipos, nos devolverá un juicio de tipado con una expresión tipada M que corresponde a U. Esta función, la ejecutaremos de manera recursiva sobre las sub-expresiones de U y sustituirá, si es posible, los tipos de cada una de ellas para que tengan "sentido" en U.

4.2.1. Propiedades deseables de $\mathbb{W}$

Dado un término U, $\mathbb{W}(U)$ nos devolverá, si tiene éxito, una terna de tres elementos que serán un contexto de tipado Γ una expresión M y un σ (notamos $\mathbb{W}(U) = \Gamma \rhd M : \sigma$).

Queremos que $\mathbb{W}$ sea **correcto** y **completo**.

Correctitud $\mathbb{W}(U) = \Gamma \rhd M : \sigma$ implica que $\text{Erase}(M) = U$ y $\Gamma \rhd M : \sigma$ es derivable. Osea que M es una expresión de tipo σ en un contexto Γ tal que si le borramos las notaciones de tipo, se convierte en U.

Completitud Si $\Gamma \rhd M : \sigma$ es derivable y $\text{Erase}(M) = U$, entonces: $\mathbb{W}(U)$ tiene éxito y produce un juicio $\Gamma' \rhd M' : \sigma'$ que es instancia del mismo. En otras palabras, si U se puede obtener a partir de una expresión M, entonces $\mathbb{W}$ deberá devolver el juicio de tipado que corresponde a M o uno más general (esto es con variables de tipos, si resulta que U podría ser de otros tipos).

4.2.2. Algoritmo de inferencia

El objetivo es definir $\mathbb{W}$ por recursión sobre la estructura de U, por lo que definirla, primero, para las construcciones más simples y luego para las expresiones compuetas. Además, el algoritmo se valdrá del algoritmo de unificación para combinar los resultados de los pasos recursivos y, así, obtener un tipado consistente.

4.2.3. Constantes y variables

$$\mathbb{W}(true) \stackrel{def}{=} \varnothing \rhd true : Bool$$
$$\mathbb{W}(false) \stackrel{def}{=} \varnothing \rhd false : Bool$$
$$\mathbb{W}(x) \stackrel{def}{=} \{x : s\} \rhd x : s,\ s \text{ variable fresca}$$
$$\mathbb{W}(0) \stackrel{def}{=} \varnothing \rhd 0 : Nat$$

4.2.4. Caso *succ*

$\mathbb{W}(succ(U)) \stackrel{def}{=} S\Gamma \rhd S\ succ(M) : Nat$

- $\mathbb{W}(U) = \Gamma \rhd M : \tau$
- $S = MGU\{\tau \doteq Nat\}$

4.2.5. Caso *pred*

$\mathbb{W}(pred(U)) \stackrel{def}{=} S\Gamma \rhd S\ pred(M) : Nat$

- $\mathbb{W}(U) = \Gamma \rhd M : \tau$
- $S = MGU\{\tau \doteq Nat\}$

4.2.6. Caso *isZero*

$\mathbb{W}(isZero(U)) \stackrel{def}{=} S\Gamma \rhd S\ isZero(M) : Bool$

- $\mathbb{W}(U) = \Gamma \rhd M : \tau$
- $S = MGU\{\tau \doteq Nat\}$

4.2.7. Caso *ifThenElse*

$\mathbb{W}(if\ U\ then\ V\ else\ W) \stackrel{def}{=} S\Gamma_1 \cup S\Gamma_2 \cup S\Gamma_3 \rhd S\ (if\ M\ then\ P\ else\ Q) : S\sigma$

- $\mathbb{W}(U) = \Gamma_1 \rhd M : \rho$
- $\mathbb{W}(V) = \Gamma_2 \rhd P : \sigma$
- $\mathbb{W}(W) = \Gamma_3 \rhd Q : \tau$
- $S = MGU\{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_i \ \wedge\ x : \sigma_2 \in \Gamma_j,\ i \neq j\} \cup \{\sigma \doteq \tau\ \ \rho \doteq Bool\}$

4.2.8. Caso aplicación

$\mathbb{W}(U\ V) \stackrel{def}{=} S\Gamma_1 \cup S\Gamma_2 \rhd S\ (M\ N) : St$

- $\mathbb{W}(U) = \Gamma_1 \rhd M : \tau$
- $\mathbb{W}(V) = \Gamma_2 \rhd N : \rho$
- $S = MGU\{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_i \ \wedge\ x : \sigma_2 \in \Gamma_j,\ i \neq j\} \cup \{\tau \doteq \rho \to t\}$ con t variable fresca

4.2.9. Caso abstracción

$$\mathbb{W}(\lambda x.U) \stackrel{def}{=} \Gamma\backslash\{x:\tau\} \rhd \lambda x:\tau.M : \tau \to \rho$$

Sea $\mathbb{W}(U) = \Gamma \rhd M : \rho$, si Γ tiene información de tipos para x, es decir $x : \tau \in \Gamma$ para algún τ, entonces:

$$\mathbb{W}(\lambda x.U) \stackrel{def}{=} \Gamma\backslash\{x:\tau\} \rhd \lambda x:\tau.M : \tau \to \rho$$

Si Γ no tiene información de tipos para x ($x \notin \mathrm{Dom}(\Gamma)$), entonces elegimos una variable fresca s y

$$\mathbb{W}(\lambda x.U) \stackrel{def}{=} \Gamma \rhd \lambda x:s.M : s \to \rho$$

4.2.10. Caso *fix*

$\mathbb{W}(fix\ (U)) \stackrel{def}{=} S\Gamma \rhd S\ fix\ (M) : St$

- $\mathbb{W}(U) = \Gamma_1 \rhd M : \tau$
- $S = MGU\{\tau \doteq t \to t\}$ con t variable fresca

4.2.11. Complejidad del algoritmo

Tanto la unificación como la inferencia para cálculo lambda se puede realizar en tiempo lineal. Sin embargo, el tipo principal asociado a un término sin anotaciones puede ser exponencial en el tamaño del término.

4.2.12. Extensión del algoritmo a nuevos tipos

Para extender el algoritmo a otros tipos debemos agregar los casos correspondientes al nuevo tipo teniendo en cuenta que los llamados recursivos devuelven un contexto, un término y un tipo sobre los que no podemos asumir nada. Si la nueva regla tiene tipos iguales o contextos repetidos, debemos unificarlos. Y si la regla liga alguna variable, entonces vamos a poder dividir en dos casos: Si alguno de los contextos recursivos tiene información sobre esa variable, entonces sacamos su tipo del contexto que la contenga, sino le asignamos una variable fresca de tipo. Si la regla tiene restricciones adicionales, se incorporan como posibles fallas. Veamos un ejemplo para la extensión de listas (definida en el ejercicio 17 de la práctica 2).

4.2.13. Extensión del algoritmo para listas

$\mathbb{W}([]) = \varnothing \rhd [\ ]_t : t$ con t variable fresca.
$\mathbb{W}(U_1 :: U_2) = S\Gamma_1 \cup S\Gamma_2 \rhd S(M_1 :: M_2) : S[\sigma_1]$

- $\mathbb{W}(U_i) = \Gamma_i \rhd M_i : \sigma_i$
- $S = MGU\{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_i \ \wedge\ x : \sigma_2 \in \Gamma_j,\ i \neq j\} \cup \{[\sigma_1] \doteq \sigma_2\}$

$\mathbb{W}(\text{case } U \text{ of } \{[\ ] \rightsquigarrow U_2 \mid h :: t \rightsquigarrow U_3\}) =$
$S\Gamma_1 \cup S\Gamma_2 \cup S\Gamma_3\backslash\{h,t\} \rhd S(\text{case } M_1 \text{ of } \{[\ ] \rightsquigarrow M_2 \mid h :: t \rightsquigarrow M_3\}) : S\sigma_2$

- $\mathbb{W}(U_i) = \Gamma_i \rhd M_i : \sigma_i$ con $i = 1, 2, 3$
- $S = MGU\{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_i \ \wedge \ x : \sigma_2 \in \Gamma_j,\ i \neq j\} \cup \{\sigma_1 \doteq [t_1],\ t_1 \doteq \tau_h,\ \tau_t \doteq \sigma_1,\ \sigma_2 \doteq \sigma_3\}$ con

$$\tau_h = \begin{cases} \sigma_h & \text{si } h : \sigma_h \in \Gamma_3 \\ t_2 & \text{sino} \end{cases} \qquad \text{y} \qquad \tau_t = \begin{cases} \sigma_t & \text{si } t : \sigma_t \in \Gamma_3 \\ t_2 & \text{sino} \end{cases}$$

5. Subtipado

Muchos lenguajes nos ofrecen la posibilidad de trabajar con subtipos. Esto significa que definen una relación entre sus tipos que, en ciertos casos, nos permite usar a un elemento de un tipo como si fuese un elemento de otro. Por ejemplo, si tenemos una función que toma dos $Float$ (reales) y nos devuelve su suma, a esa función podriamos pasarle un Nat (natural) o un Int (entero) y podríamos ejecutarla sin problema ya que estos tipos, en realidad, son subconjuntos de los reales.

Hasta ahora, para nuestro lenguaje λ, definimos un sistema de tipado que descarta todos los programas que no tipen, sin embargo la definición de tipado que tenemos es demasiado rígida y no nos permite definir programas con este estilo. Por esta razón, vamos a extender nuestro sistema y definir nuevas reglas de tipado que nos permitan incluir este tipo de expresiones en el conjunto de expresiones tipables del lenguaje.

Principio de sustitutividad Primero definimos la relación $\sigma <: \tau$, que indica que en todo contexto donde se espera una expresión de tipo τ, podemos utilizar una de tipo σ en su lugar **sin** que ello genere un error. Y además agregamos la regla de subtipado (**Subsumption**) T-Subs:

$$\frac{\Gamma \rhd M : \sigma \qquad \sigma <: \tau}{\Gamma \rhd M : \tau}(\text{T-Subs})$$

Esta regla es la que nos dice que si tenemos una expresión de tipo σ tal que $\sigma <: \tau$, entonces también podremos considerar a M como una expresión de tipo τ.

El tipo máximo Además, vamos a agregar, a nuestro lenguaje, el tipo Top que contendrá a todos los tipos del lenguaje y lo llamaremos **supertipo universal** porque todo tipo es subtipo de Top:

$$\frac{}{\sigma <: Top}(S - Top)$$

Tipos con constructores invariantes: Son aquellos tipos que no pueden ser remplazados por ningún otro.

Tipos con constructores covariantes: Son aquellos que tipos cuyo subtipos se consiguen remplazando sus argumentos por un subtipo del argumento.

Tipos con constructores contravariantes: Son aquellos que tipos cuyo subtipos se consiguen remplazando sus argumentos por un supertipo del argumento.

5.1. Reglas de subtipado

5.1.1. Tipos básicos

$$\frac{}{Nat <: Float}(\text{S-NatFloat}) \qquad \frac{}{Int <: Float}(\text{S-IntFloat}) \qquad \frac{}{Bool <: Nat}(\text{S-BoolNat})$$

5.1.2. Subtipado del tipo función

Buscamos una función $g : \sigma \to \tau$ que pueda remplazar a otra función $f : \sigma' \to \tau'$ en cualquier contexto sin generar ningún error. Lo primero que tenemos que tener en cuenta es que g debe estar definida para todos los elementos del dominio de f, osea $Dom(f) \subseteq Dom(g)$ pues si pasara que existe un valor x tal $f(x)$ está definida y $g(x)$ no lo está, entonces obtendríamos un error. Entonces, $\sigma' <: \sigma$.

Por otro lado, g no debería poder devolver como resultado ningún valor que no esperamos que f no devuelva ($Im(g) \subseteq Im(f)$). Podemos aclarar esto con un ejemplo, supongamos que tenemos la función esPar :: $Nat \to Nat$, cuando usemos esta función, el contexto en el que la usemos estará esperando conseguir un natural de su evaluación. Si la función g devuelve algo que no sea un Nat, entonces obtendriamos un error. Sin embargo, si $g :: Nat \to Bool$, entonces podremos realizar el remplazo sin ningún problema porque $\{0, 1\}$ es un subconjunto de los naturales. Luego $\tau <: \tau'$.

Y la regla de subtipado queda:

$$\frac{\sigma' <: \sigma \qquad \tau <: \tau'}{\sigma \to \tau <: \sigma' \to \tau'} \text{(S-Func)}$$

El constructor de tipos función es **contravariante** en su primer argumento y **covariante** en el segundo.

Un programa P, deberá **coercionar** (transformar) el argumento que le pasan a la función para que coincida con el tipo de la nueva función, ejecutarla y luego coercionar su resultado al tipo del resultado que espera P.

5.1.3. Reglas de subtipado de términos

Las reglas de tipado sin subtipos son dirigidas por sintaxis, por lo que es inmediato implementar un algoritmo de chequeo de tipos a partir de ellas. Con el agregado de la regla T-Subs, el chequeo también pasa a estar dirigidas por la semántica de la relación <:, por lo que el algoritmo deja de ser tan directo.

Un juicio de subtipado $\Gamma \mapsto M : \sigma$, nos dice que $\Gamma \rhd M : \sigma$ y que existe existe τ tal que $\Gamma \mapsto M : \tau$ con $\tau <: \sigma$. Entonces las nuevas reglas quedarían así:

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

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

La mayoría de las reglas de subtipado son similares a las reglas de tipo y la única regla que usa la relación <: de manera explicita es T-App, porque los términos de aplicación son los únicos que al ser evaluados remplazan expresiones y, efectivamente, esta regla es la que nos da el poder que estabamos buscando.

5.1.4. Relación de preorden

La relación de subtipado <: define una relación de preorden, es decir es reflexiva y transitiva por lo que podriamos escribir las siguientes reglas:

$$\frac{}{\sigma <: \sigma}(\text{S-Refl}) \qquad \frac{\sigma <: \tau \quad \tau <: \rho}{\sigma <: \rho}(\text{S-Trans})$$

Sin embargo, esta forma de descrbir la relación no es dirigida por semántica y no está claro como hacer un algoritmo de chequeo de subtipos que use estas reglas. Por esta razón vamos a considerar los siguientes tres axiomas:

$$\frac{}{Nat <: Nat}(\text{S-NatNat}) \qquad \frac{}{Bool <: Bool}(\text{S-BoolBool}) \qquad \frac{}{Float <: Float}(\text{S-FloatFloat})$$

Y ya con eso alcanza para derivar la reflexibidad de cualquier tipo del lenguaje, en el lenguaje λ básico. Si agregasemos un nuevo tipo escalar (sin parámetros), entonces debemos declarar explicitamente que ese tipo es subtipo de si mismo, sino el preorden se rompe.

Por otro lado, la transitividad se puede demostrar aplicando varios pasos de subtipado, por lo que directamente no es necesaria.

5.1.5. Algoritmo de chequeo de tipos

Entonces logramos escribir todas las reglas del sistema de subtipado de manera tal que son dirigidas por la sintaxis y podemos definir el algoritmos $subtype(S, T)$ que nos indica si S es subtipo de T. Al algoritmo mostrado le faltan los axiomas de Nat, $Bool$ y $Float$ que son triviales (hay que poner cada una de esas comparaciones una por una):

```
subtype(S, T) =
  if T==Top
    then true
    else
     if S==S1 → S2 and T==T1 → T2
        then subtype(T1, S1) and subtype(S2, T2)
        else
         if S=={kj : Sj, j ∈ 1..m} and T=={li : Ti, i ∈ 1..n}
            then {li, i ∈ 1..n} ⊆ {kj, j ∈ 1..m} and
               ∀ i ∃ j kj = li and subtype(Sj, Ti)
            else false
```

5.2. Subtipado de referencias

Queremos encontrar el tipo $Ref\ \tau$ que sea subtipo de $Ref\ \sigma$. Supongamos que $\tau <: \sigma$, si intentamos subtipar una referencia $M : Ref\ \sigma$ con $Ref\ \tau$, entonces cuando realicemos una asignación podremos usar un valor de tipo τ y no tendremos error. Ahora, cuando derreferenciemos M estarémos esperando algo de tipo σ, pero como τ es más general que σ puede tener valores que no son de ese tipo, por lo que si un contexto esperaba algo del primer tipo se obtendría un error.

Si $\sigma <: \tau$ y $M : Ref\ \sigma$, entonces podemos guardar en M un elemento de tipo σ, sin embargo cuando querramos derreferenciar M, como $Ref\ \sigma$ es subtipo de $Ref\ \tau$, podremos usar la derreferencia del segundo tipo. El problema vuelve a ser el mismo, el contexto va a estar esperando un valor de tipo τ, pero σ es más general, por lo que el valor almacenado en M puede no ser de este tipo, lo que llevaría a un error.

A continuación dos ejemplos que muestran cada caso usando los tipos $Int <: Float$:

$$let\ r = \text{ref } 3\ in\ r := 2{,}1;$$
$$!r$$

Este es el primer caso, como definimos r como una referencia de enteros en el let, cuando derreferenciemos r esperamos conseguir un entero, sin embargo la regla covariante, hace que con la asignación podamos asignar a r un $Float$

$$let\ r = \text{ref } 2{,}1\ in\ !r \tag{1}$$

Y este es el segundo caso, en el que definimos a r como una referencia de $Float$, sin embargo, como r es subtipable a $Ref\ Int$, podemos usar la derreferenciación de enteros para derreferenciarla, lo que provocaría el error en el programa.

Entonces, la regla de subtipado no es ni contravariante ni covariente, es variante. La única "sustitución"que podemos hacer es cuando σ y τ son el mismo tipo.

$$\frac{\sigma <: \tau}{Ref\ \tau <: Ref\ \sigma}$$

5.2.1. Refinando el tipo *Ref*

Extendemos el lenguaje, con los siguiente tipos $Source\ \sigma$ y $Sink\ \sigma$ que representan las referencias de lectura y las de escritura, respectivamente.

Reglas de tipado

$$\frac{\Gamma|\Sigma \rhd M : Source\ \sigma}{\Gamma|\Sigma \rhd !M : \sigma}\text{(T-DeRefSource)}$$

$$\frac{\Gamma|\Sigma \rhd M : Sink\ \sigma \qquad \Gamma|\Sigma \rhd N : \sigma}{\Gamma|\Sigma \rhd M := N : Unit}\text{(T-AssignSink)}$$

Reglas de subtipado

$$\frac{\sigma <: \tau}{Source\ \sigma <: Source\ \tau}\text{(S-Source)} \qquad \frac{\tau <: \sigma}{Sink\ \sigma <: Sink\ \tau}\text{(S-Sink)}$$

$$\frac{}{Ref\ \tau <: Source\ \tau}\text{(S-RefSource)} \qquad \frac{}{Ref\ \tau <: Sink\ \tau}\text{(S-RefSink)}$$

La regla S-Source es covariante, si esperamos leer de una referencia de tipo τ, entonces podemos esperar una referencia de un tipo más especifico que τ.

La regla S-Sink es contravariante, ya que cuando querramos guardar un valor de tipo τ, podremos guardarlo en una referencia de este tipo o en una de un tipo más general.

Además, $Source\ \tau$ y $Sink\ \tau$ son mas generales que $Ref\ \tau$ ya que siempre podremos remplazar referencias de lecturas o de escritura por referencias de lectura y escritura.

Parte II
Paradigma orientado a objetos

6. Objetos y el modelo de cómputo

En el parádigma orientado a objetos, todo programa es una simulación representada por una entidad u **objeto** que asocia los objetos físicos o conceptuales de un dominio del mundo real en objetos del dominio del programa. Estos objetos tienen las características y capacidades del mundo real que nos interesa modelar y se comunican entre si a través de intercambios de mensajes.

Los mensajes intercambiados son solicitudes para que el objeto **receptor** del mismo lleve a cabo una de sus operaciones. El **receptor** determinara si puede llevar a cabo dicha operación y, si puede hacerlo la ejecutará.

6.1. Objetos

Entonces un objeto es una entidad del programa que puede recibir un conjunto de mensajes (al que llamaremos **interfaz** o **protocolo**) que le permite determinar como llevar a cabo ciertas operaciones. Internamente, estará compuesto por un conjunto de **colaboradores internos** (tambien llamados **atributos** o **variables internas**) que determinan su **estado interno** y por un conjunto de **métodos** que describen (implementan) las operaciones que puede realizar y, si estas afectan a su estado interno, como lo hacen.

Principio de ocultamiento de la información El estado de un objeto es **privado** y solamente puede ser consultado o modificado por sus propios métodos, por lo que su implementación no depende de los detalles de implementación de otros objetos. Y la única forma que tenemos de interactuar con el mismo es enviándole los mensajes definidos en su interfaz.

Method dispatch Es el método mediante el cuál, un proceso, estable la asociación entre el mensaje y el método a ejecutar. Es decir, cuando un objeto recibe un mensaje, el **method dispatch** se encarga de hallar la **declaración del método** que se pretende ejecutar. Este método puede ser **estático** (realizado en tiempo de compilación) o **dinámico** (realizado en tiempo de ejecución).

Corrientes de organización Por lo general,tratamos de agrupar los objetos en conjuntos compuestos por objetos que se comportan de manera similar para conseguir programas más concisos. Esto se puede hacer de dos formas: Mediante clasifiación o mediante prototipado.

7. Clasificación

Se usan **clases** que modelan **conceptos abstractos** del dominio del problema a resolver y definen el comportamiento y la forma de un conjunto de objetos (sus **instancias**). Todo **objeto** es una instancia de una clase.

Componentes de una clase Todas las clases tienen un **nombre** que usado para referenciarse a la misma. Dentro de ellas se definen las variables de instancias (colaboradores internos) de los objetos) y los métodos que saben responder esas instancias (sus nombres, sus parámetros y su cuerpo).

7.1. Self/This

Todas las clases tienen definida una pseudovariable que, durante la evaluación de un método, referencia al receptor del mensaje que activó dicha evaluación. No puede ser modificada por medio de una asignación y se liga automáticamente al receptor cuando comienza la evaluación del método.

```
!classDefinition: #Node
        instanceVariableNames: 'leftchild, rightchild'
        ...
sum:
        ^ (self leftchild) sum + (self rightchild) ! !

!classDefinition: #Leaf
        instanceVariableNames: 'value'
        ...
sum:
        ^self value ! !
```

Vemos que los métodos acceden a sus variables de instancia, enviándose a si mismos el mensaje asociado a cada una de ellas. En muchos lenguajes, para facilitar la escritura de un programa, la mención de `self` se hace implícitamente.

7.2. Jerarquía de clases

Cuando escribimos un programa en este paradigma, es común que creemos nuevas clases que extiendan a las ya existentes con nuevas variables de instancia o clase o que modifiquen el comportameiento de unos o varios métodos.

Para evitar tener que escribir toda una clase de cero, hacemos que la clase que estamos creando **herede** los atributos y los métodos de la clase pre-existente (la **super-clase**) que queremos extender. De esta forma, la nueva clase tendrá todo lo que tenía la super-clase y, además, las modificaciones que nosotros querramos agregarle.

La herencia define una relación transitiva por lo que si una clase A tiene como super-tipo a otra clase B, entonces el super-tipo C de B, entonces C tambien es supertipo de A. Llamaremos **ancestros** a todos los supertipos de A y **descendientes** a todos los tipos que tienen a A como ancestro.

7.3. Tipos de herencia

Hay dos tipos de herencia: **simple** y **múltiple**. La herencia simple permite que una clase tenga una única clase padre y la única clase que no tiene padre es la clase `Object` que es la clase de la que heredan todas las demás. Mientras que la herencia múltiple deja que una clase tenga varios padres.

La mayoría de los lenguajes orientados a objetos utilizan la primera ya que la herencia múltiple complica el proceso de method dispath (que asocia los mensajes de un objeto con sus respectivos métodos). Para ver por qué pasa esto, supongamos que tenemos dos clases A y B incomparables y una clase C que es subclase de A y B. Si A y B definen (o heredan) dos métodos diferentes para un mismo mensaje m, entonces cuando enviemos dicho mensaje a C deberíamos saber cual de los dos elegir.

Hay dos soluciones posibles a este problema:

- Podemos establecer un **orden de búsqueda** sobre las superclases de un clase estableciendo, de esta forma, un nivel de prioridad sobre algunas de ellas.
- O obligar al programado a **redefinir** el método en C si C hereda dos métodos distintos para el mismo mensaje.

7.3.1. Method Dispatch

Como dijimos, el **method Dispatch** es el método mediante el cual asociamos un mensaje a su método correspondiente en el objeto. Por lo general, este método se realiza de manera dinámica, es decir se realizan durante tiempo de ejecución dependiendo del contexto sin embargo, hay situaciones en las que realizar este proceso de manera estática es necesario.

Un ejemplo de esto, es cuando el lenguaje nos permite hacer uso de `super`, una pseudovariable que **referencia al objeto que recibe el mensaje** y **cambia** su proceso de activación al momento de recibir un mensaje. Cuando usamos una expresión de la forma `super msg` en el cuerpo de un método m, el **method lookup** (la búsqueda del método realizada por el method dispath), comienze a realizarse desde el padre de la **clase anfitriona** de m.

Algunos lenguajes, además, nos permiten pasarle como parámetro una clase a partir de la que empezar de la siguiente forma: `super[A] msg`, siempre y cuando A sea un ancestro de la clase anfitriona del método.

8. Prototipado

Lo lenguajes basados en prototipado de objetos se caracterizan por la ausencia de clases. Proveen constructores para la creación de objetos particulares y la herramientas necesarias para crear procedimientos que generen objetos.

En este tipo de paradigma, creamos instancias concretas que se interpretan como representantes canónicos de instancias (llamados **prototipos**) y, a partir de ellos, generamos otras intancias (**clones**) que pueden ser modificados sin afectar al prototipo.

Cuando hacemos esto hacemos lo que se llama una **shallow copy**, es decir copiamos cada atributo de un objeto A en otro B. Es decir, si en A tenemos una referencia a C, entonces en B tendremos una referencia a C, no copiaremos C a ningún otro objeto.

8.1. Cálculo de objetos no tipado (ς cálculo)

Usaremos un lenguaje cuya única estructura computacional son los **Objetos**. Estos objetos son una colección de atributos nombrados (**registros**) que están asociados a métodos con una única variable ligada (que representa a `self`/`this`) y un cuerpo que produce un resultado.

Todos los objetos proveen dos operaciones:

Envío de mensajes: Que nos permite invocar un método para que el objeto ejecute.

Redefinición de un método: Que nos permite reemplazar el cuerpo de un atributo por otro.

8.1.1. Sintaxis

a, b	$::=$	x	Variables
	$\mid$	$[l_i = \varsigma(x_i)b_i^{i \in 1..n}]$	Objetos
	$\mid$	$a.l$	Selección/ Envío de mensajes
	$\mid$	$a.l \Leftarrow \varsigma(x)b$	Redefinición de un método.

El objeto $[\,]$ es el objeto vacío y no proporciona ningún método.

En este lenguajes, como todos los átributos son métodos, simulamos los colaboradores internos de un objeto con métodos que no utilizan el parámetro `self`. Por ejemplo:

$$o \stackrel{def}{=} [l_1 = \varsigma(x_1)[\,],\ l_2 = \varsigma(x_2)x_2.l_1]$$

$o.l_1$ retorna un objeto vacío. Y $o.l_2$ envia el mensaje l_1 a `self` (representado por el parámetro x_2).

Notación Cuando un objeto tenga un atributo de la forma $l = \varsigma(x)b$ y x no se usa en b podemos escribir $l = b$ y a la reasignación $o.l \Leftarrow \varsigma(x)b$ como $o.l := b$.

Variables libres ς es un ligador de variables, cuando lo usamos en una expresión de la forma $\varsigma(x)b$ siempre liga la variable x que se le pasa como párametro a `self`. Osea que cuando x aparece en b será remplazada por `self`.

De manera análoga a FV del cálculo λ definimos fv para objetos y diremos que un término a es **cerrado** si $\text{fv}(a) = \varnothing$:

$$
\begin{array}{ll}
\mathrm{fv}(\varsigma(x)b) & = \mathrm{fv}(b)\backslash\{x\} \\
\mathrm{fv}(x) & = \{x\} \\
\mathrm{fv}([l_i = \varsigma(x_i)b_i^{i\in 1..n}]) & = \bigcup^{1\in 1..n} \mathrm{fv}(\varsigma(x)b) \\
\mathrm{fv}(a.l) & = \mathrm{fv}(a) \\
\mathrm{fv}(a.l \Leftarrow \varsigma(x)b) & = \mathrm{fv}(a.l) \cup \mathrm{fv}(\varsigma(x)b)
\end{array}
$$

Sustitución La función de sustitución de variables libres para objetos está definida de la siguiente forma:

$$
\begin{array}{lll}
x\{x \leftarrow c\} & = c & \\
y\{x \leftarrow c\} & = y & \text{si } x \neq y \\
([l_i = \varsigma(x_i)b_i^{i\in 1..n}])\{x \leftarrow c\} & = [l_i = (\varsigma(x_i)b_i)\{x \leftarrow c\}^{i\in 1..n}] & \\
(a.l)\{x \leftarrow c\} & = (a\{x \leftarrow c\}).l & \\
(a.l \Leftarrow \varsigma(x)b)\{x \leftarrow c\} & = (a\{x \leftarrow c\}).l \Leftarrow (\varsigma(x)b)\{x \leftarrow c\} & \\
(\varsigma(y)b)\{x \leftarrow c\} & = (\varsigma(y')(b\{y \leftarrow y'\}\{x \leftarrow c\})) & \text{si } y' \notin \mathrm{fv}(\varsigma(y)b) \cup \mathrm{fv}(c) \cup \{x\}
\end{array}
$$

Notemos que en el último caso, remplazamos y por y' por si $y = x$ asegurandonos, de esta manera, que no cambiamos el significado de la expresión.

α-converión En objetos decimos que dos métodos son equivalentes, si tienen el mismo cuerpo salvo renombre de variables, es decir: $\varsigma(x)b$ y $\varsigma(y)(b\{x \leftarrow y\})$ con $y \notin \mathrm{fv}(b)$ son equivalentes.

Además, dos objetos o_1 y o_2 son considerados equivalentes ($o_1 \equiv o_2$) si solo difieren en el orden se sus componentes. Si

$$
\begin{aligned}
o_1 &\stackrel{def}{=} [l_1 = [\,],\ l_2 = \varsigma(x_2)x_2.l_1] \\
o_2 &\stackrel{def}{=} [l_2 = \varsigma(x_3)x_3.l_1, l_1 = [\,]]
\end{aligned}
$$

son equivalentes porque ambos objetos tiene los atributos l_1 y l_2 y $\varsigma(x_2)x_2.l_1 =_\alpha \varsigma(x_3)x_3.l_1$.

8.1.2. Semántica operacional

Todos los objetos son considerados valores.

$$V \ ::= \ [l_i = \varsigma(x_i)b_i^{1\in 1..n}]$$

A diferencía del cálculo λ, usaremos el método de reducción **big-step** para evaluar expresiones, que en un solo paso nos permite saber el valor que representa.

$$\frac{}{v \longrightarrow v}[\mathrm{Obj}]$$

$$\frac{a \longrightarrow v' \qquad v' \equiv [l_i = \varsigma(x_i)b_i^{i\in 1..n}] \qquad b_j\{x_j \leftarrow v'\} \longrightarrow v \qquad j \in 1..n}{a.l_j \longrightarrow v}[\mathrm{Sel}]$$

$$\frac{a \longrightarrow [l_i = \varsigma(x_i)b_i^{i\in 1..n}] \qquad j \in 1..n}{a.l_j \Leftarrow \varsigma(x)b \longrightarrow [l_j = \varsigma(x)b,\ l_i = \varsigma(x_i)b_i^{i\in 1..n-\{j\}}]}[\mathrm{Upd}]$$

La regla Obj nos dice que un objeto no reducen.

Sel nos indica que el resultado de enviar un mensaje es el valor que obtenemos al remplazar el párametro del método por el mismo objeto (esto es la ligación a `self`).

Upd es el comportamiento de la redifinición, que devuelve un objeto con los mismos atributos que a pero remplazando el $j-$ésimo atributo por la nueva definición.

Ejemplo de reducción

$$\dfrac{\dfrac{\dfrac{}{[\,]\{x \leftarrow o\} \longrightarrow [\,]}\,[\text{Obj}]}{(x.a)\{x \leftarrow o\} \longrightarrow [\,]}\,[\text{Sel}]}{[a = [\,],\ b = \varsigma(x)x.a].b \longrightarrow [\,]}\,[\text{Sel}]$$

Indefinición Similar al cálculo lambda, podemos definir expresiones que se indefinen pero, en este caso, no es necesario que introduzcamos ninguna estructura nueva. Simplemente podemos $[a = \varsigma(x)x.a].a$ y con esto ya alcanza.

Codificación de funciones (Cálculoa λ) Las expresiones son objetos con un atributo *val* que nos indica su valor, las funciones, además tiene el atributo *arg* que representa al argumento de la función. El argumento de una función permanecerá indefinido hasta que aparezca en una aplicación.

$$\begin{aligned}
[[x]] &\stackrel{def}{=} x \\
[[M\ N]] &\stackrel{def}{=} [[M]].arg := [[N]] \\
[[\lambda x.M]] &\stackrel{def}{=} [val = \varsigma(y)[[M]]\{x \leftarrow y.arg\},\ arg = \varsigma(y)y.arg]
\end{aligned}$$

Cuando querramos representar un método que espera parámetros, usaremos la definición de función para escribirlo: $\varsigma(x)[[\lambda x.M]]$. Y podemos hacer abuso de notacion y escribir $\lambda(X)M$ y $M(N)$, en vez de $[[M\ N]]$.

8.1.3. Traits

Un trait es una colección de métodos que parametrizan cierto comportamientos. Estos objetos no especifican variables de estado ni acceden a su estado.

El trait y sus métodos por si solo no son utilizables, ya que el trait no provee los estados necesarios para evaluarlos correctamente. Solo lo usaremos para definir métodos que pueden ser evaluados por varios objetos con el objetivo de no tener que repetir siempre las mismas definiciones.

Los vamos a representar como una colección de **pre-metodos** (que no usan el parámetro self). Por lo que un trait tendrá la forma:

$$\mathtt{t} = [l_i = \lambda y_i.b_i^{i \in 1..n}]$$

Y, además, definimos *new* como un constructor de objetos que crea un objeto con las mismas etiquetas que el trait y que asocia a cada una de ellas un método que invoca al método del trait con el primer parámetro ligado a `self`:

$$new \stackrel{def}{=} \lambda z.[l_i = \varsigma(s)z.l_i(s)^{i \in 1..n}]$$

Veamos un ejemplo, definimos el trait `CompT`:

$$\texttt{CompT} \stackrel{def}{=} [\; eq = \varsigma(t)\lambda(x)\lambda(y) if\ (x.comp(y)) == 0\ then\ \texttt{true}\ else\ \texttt{false},\ leq = \varsigma(t)\lambda(x)\lambda(y) if\ (x.comp(y)) < 0\ then\ \texttt{true}\ else\ \texttt{false}\;]$$

Entonces *new* `CompT` reduce a:

$$new\ \texttt{CompT} \longrightarrow [\; eq = \varsigma(x)\lambda(y) if\ (x.comp(y)) == 0\ then\ \texttt{true}\ else\ \texttt{false},\ leq = \varsigma(x)\lambda(y) if\ (x.comp(y)) < 0\ then\ \texttt{true}\ else\ \texttt{false}\;]$$

Clase Cuando un trait provee un método *new*, diremos que es un **trait completo** o **clase**.

$$\texttt{C} \stackrel{def}{=} [\; new = \varsigma(z)[l_i = \varsigma(s)z.l_i(s)^{i \in 1..n}],\ leq = \varsigma(t)\lambda(x)\lambda(y) if\ (x.comp(y)) < 0\ then\ \texttt{true}\ else\ \texttt{false}\;]$$

Herencia Cuando queremos que una clase "herede", lo que hacemos es crear un nuevo trait que contenga todas las etiquetas del trait original y asocie cada una de esas etiquetas al método correspondiente del trait original y le agregamos los atributos que deseamos para extenderlo. Además, modificamos el constructor *new* para que tome en cuenta los nuevos atributos.

Por ejemplo, la clase contador:

$$\texttt{Contador} \stackrel{def}{=} [\; new = \varsigma(z)[inc = \varsigma(s)z.inc(s),\ v = 0,\ get = \varsigma(s)z.get(s)],\ inc = \lambda(s)s.v := s.v + 1,\ get = \lambda(s)s.v\;]$$

Y su sublcase `ContadorR`:

$$
\begin{array}{ll}
\texttt{ContadorR} \stackrel{def}{=} [& new = \varsigma(z)[\\
& \quad inc = \varsigma(s)z.inc(s), \\
& \quad v = 0, \\
& \quad get = \varsigma(s)z.get(s), \\
& \quad reset = \varsigma(\lambda(s))z.reset(s) \\
&], \\
& inc = \texttt{Contador}.inc \\
& get = \texttt{Contador}.get \\
& reset = \lambda(s)s.v := 0, \\
&]
\end{array}
$$

Otras consideraciones del lenguaje El lenguaje que definimos se parece mucho al funcional. En la versión imperativa, donde se mantiene un store con referencias a objetos, se ofrece la función `Clone`(a) que crea un nuevo objeto con las mismas etiquetas de a y cada componente comparte los métodos con las componentes de a.

Además, el lenguaje no nos deja agregar o eliminar dinámicamente métodos en un objeto. No nos deja extraer los métodos de los objetos.

Hay otras versiones de este cálculo que incluyen sistemas de tipado.

Parte III
Apéndices

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

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

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

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

A.1. Otros tipos útiles

Formula

```
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

```
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 ) )
```

B. Extensiones del lenguaje λ^b

B.1. 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 s 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

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

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

Axiomas y reglas de subtipado

$$\frac{\{l_i | i \in 1..n\} \subseteq \{k_j | j \in 1..m\} \quad k_j = l_i \Rightarrow \sigma_j <: \tau_i}{\{k_j : \sigma_j | j \in 1..m\} <: \{l_i : \sigma_i | i \in 1..n\}} \text{(S-Rcd)}$$

Esta regla nos dice que un registro N es subtipo de otro registro M, si el conjunto de etiquetas de M está contenido en el conjunto de etiquetas de N y, además, si los tipos de cada una de esas etiquetas, en M, es más general que en N.

Una de las consecuancias de esta regla es que $\sigma < \{\}$ para todo tipo registro σ. Esto es porque $\{\}$ no tiene etiquetas, osea que su conjunto de etiquetas es el conjunto $\varnothing$ que está contenido en todos los conjuntos.

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.2. Declaraciones Locales ($\lambda^{\cdots let}$)

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

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

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

$$\frac{M_1 \to M'_1}{let\ x : \sigma = M_1\ in\ M_2 \to let\ x : \sigma = M'_1\ in\ M_2}\text{(E-Let)}$$

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

B.2.1. 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 3.7) de la siguiente forma:

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

B.3. Tuplas

Tipos

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

Términos

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

Axiomas y reglas de tipado

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

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

Valores

$$V\ ::=\ \ldots \mid <V,V>$$

Axiomas y reglas de evaluación

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

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

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

B.4. Árboles binarios

Tipos

$$\sigma,\tau\ ::= \ldots \mid AB_\sigma$$

Términos

$$M,\ N\ ::=\ \ldots \mid \text{Nil}_\sigma \mid \text{Bin}(M,N,O) \mid \text{raiz}(M) \mid \text{der}(M) \mid \text{izq}(M) \mid \text{esNil}(M)$$

Axiomas y reglas de tipado

$$\frac{}{\Gamma \rhd \text{Nil}_\sigma : AB_\sigma}(\text{T-Nil}) \qquad \frac{\Gamma \rhd M : AB_\sigma \qquad \Gamma \rhd N : \sigma \qquad \Gamma \rhd O : AB_\sigma}{\Gamma \rhd \text{Bin}(M,N,O) : AB_\sigma}(\text{T-Bin})$$

$$\frac{\Gamma \rhd M : AB_\sigma}{\Gamma \rhd \text{raiz}(M) : \sigma}(\text{T-raiz}) \qquad \frac{\Gamma \rhd M : AB_\sigma}{\Gamma \rhd \text{der}(M) : AB_\sigma}(\text{T-der})$$

$$\frac{\Gamma \rhd M : AB_\sigma}{\Gamma \rhd \text{izq}(M) : AB_\sigma}(\text{T-izq}) \qquad \frac{\Gamma \rhd M : AB_\sigma}{\Gamma \rhd \text{isNil}(M) : \mathit{Bool}}(\text{T-isNil})$$

Valores

$$V\ ::=\ \ldots \mid \text{Nil} \mid \text{Bin}(V,V,V)$$

Axiomas y reglas de evaluación

$$\frac{M \to M'}{\text{Bin}(M, N, O) \to \text{Bin}(M', N, O)}\text{(E-Bin1)} \qquad \frac{N \to N'}{\text{Bin}(V, N, O) \to \text{Bin}(V, N', O)}\text{(E-Bin2)}$$

$$\frac{O \to O'}{\text{Bin}(V_1, V_2, O) \to \text{Bin}(V_1, V_2, O')}\text{(E-Bin3)}$$

$$\frac{M \to M'}{\text{raiz}(M) \to \text{raiz}(M')}\text{(E-Raiz1)} \qquad \frac{}{\text{raiz}(\text{Bin}(V_1, V_2, V_3)) \to V_2}\text{(E-Bin3)}$$

$$\frac{M \to M'}{\text{der}(M) \to \text{der}(M')}\text{(E-Der1)} \qquad \frac{}{\text{der}(\text{Bin}(V_1, V_2, V_3)) \to V_3}\text{(E-Der2)}$$

$$\frac{M \to M'}{\text{izq}(M) \to \text{izq}(M')}\text{(E-Izq1)} \qquad \frac{}{\text{izq}(\text{Bin}(V_1, V_2, V_3)) \to V_1}\text{(E-Izq2)}$$

$$\frac{}{\text{isNil}(M) \to \text{izq}(M')}\text{(E-isNil1)} \qquad \frac{}{\text{isNil}(\text{Bin}(V_1, V_2, V_3)) \to false}\text{(E-isNilBin)}$$

$$\frac{}{\text{isNil}(\text{Bin}(V_1, V_2, V_3)) \to true}\text{(E-isNilNil)}$$

C. Javascript

Declaración de variables

```
// Declaración de variables
let miVar = 1;
var suVar = 2;

// Declaración de constante, no pueden ser modificadas.
const miConstante = 3;
```

Y es **case-sensitive**, es decir `unavariable` y `unaVariable` no son las mismas variables.

Tipos

- `number`: Los números, no hay distinción entre enteros y punto flotantes. Contiene a las constantes `-Infinity`,`+Infinity`, `NaN`.
- `boolean`: Los literales `true` y `false` con las operaciones `&&`, `!` y `||`.
- `string`: Secuencias de cero o más carácteres entre comillas simples o dobles.
- `null`: Un único valor `null` (nada, valor desconocido).
- `undefined`: Un único valor `undefined` (el valor no está definido).

Podemos usar `typeof` para saber el nombre del tipo de la expresión.

- `Arrays: [], [1,2,true], new Array()`
  `-[-] , push(-) , pop() , shift() , unshift(_)`

Tipado El tipado se hace de manera **dinámica** (en tiempo de ejecución) y **débil** (se pueden comparar cosas que no son del mismo tipo porque hay conversión automática).

Por ejemplo:

```
let a = 1  // a = 1
a += '1'
// a = '11' (El entero, se convierte automaticamente en un string)
```

```
1 == '1' // true
1 === '1' // false
0 == false // true
0 === false // false
1 == true // true
```

```
false == '' // true
false === 'false' // false
null == undefined // true
null === undefined // false
```

Flujos de control

```
if (cond) { ... } else { ... }

while (cond) {
    //cuerpo
}

do {
    //cuerpo
} while (cond);
```

Definición de funciones

```
function nombre(arg1, ..., argn){
    //cuerpo
}

let nombre = function(arg1, ..., argn){
    //cuerpo
}

let nombre = (arg1, ..., argn) => {
    //cuerpo
}
```

Objetos

```
let o = {
    a : 1,
    b : function(n) {
        return this.a + n // this hace referencia a o
    }
}

o.a // 1 Proyeccion del atributo a

o['a'] // 1 (Proyeccion del atributo a)

o.b(1) // 2  (Proyeccion y ejecucion del metodo b)


o.b = function() { return this.a } // Redefinimos o.b

o.c = true // Agrega el atributo c a o.
o['c'] = false // Redefinimos el atributo c de o.
delete o.a // Elimina el atributo a de o.
```

```
'c' in o // true (Checkea si c es una propiedad de o)
'd' in o // false

// Iteracion sobre todas las propiedades de o.
for(let p in o){
    ...
}

let p = o // Creamos una referencia a o.

let f = o.b
// Extraemos el metodo b de f. Dejamos las variable this desligada.

let o2 = { i: f, a: true}
// Crea el objeto o2, con la función f en su atributo i.

o2.f() // true

let o3 = Object.assign({}, o, o2)
// Copia las propiedade de o y o2 en o3. Hace un shallow copy, es decir
↪  si una atributo de o2 u o3 es una referencia, entonces en o, ese
↪  atributo va a ser una referencia al mismo objeto que en o2
```

Herencia Todos los objetos tiene una propiedad privada llamada `[[Prototype]]` cuyo valor es `null` u otro objeto que es su **prototipo**. Esta propiedad induce una cadena de prototipado sin ciclos que finaliza con `null`.

Cuando intentamos acceder a un método inexistente de un objeto, el mismo se busca en la cadena de prototipado del mismo hasta encontrar el primer objeto de la cadena que lo define. Si llegamos a `null` y el método no fue encontrado, entonces hay un error.

```
Object.setPrototypeOf(o, prot) // Hace que el prototipo de o sea prot.
Object.getPrototypeOf(o) // Devuelve el objeto que es protipo de o.

o.__proto__ // Otra forma de conseguir el prototipo de o.
o.__proto__ = o1 // Otra forma de setear el prototipo de o
```

Cuando ejecutamos un método de la cadena de prototipado, el método evaluado liga `this` al objeto llamó al método, es decir, si tenemos:

```
let o1 = { a: 1}
let o2 = { b: function(){
          return this.a
    }
}

o1.__proto__= o2

o1.b();
// Busca b en o1 y no lo encuentra, el proximo objeto en la cadena es
 ↪  o2, encuentra b y liga this = o1, luego ejecuta el método.
```

Para crear una copia de un objeto y asignarle a esa copia el objeto original como prototipo usamos `Object.create`.

Además javascript provee el prototipo `Object.prototype` que es prototipo de todos los objetos y provee metodos básicos como `hasOwnProperty()` que indica si el objeto contiene una propiedad no heredada y `toString()` que devuelve un string que representa al receptor.

Cadenas de prototipado

```
let o1 = { ... }
// o1 ---> Object.prototype ---> null

let o2 = Object.create(o1)
// o2 ---> o1 ---> Object.prototype ---> null

let o = Object.create(null)
// o ---> null
```

Constructores Son funciones que generan objetos. Cuando declaramos un constructor `C`, javascript crea un objeto llamado `C.prototype` que se asigna como prototipo de todos los objetos creados con dicha función. Por ejemplo:

```
// Constructor Punto, tiene la siguiente cadena de prototipado:
// Punto --> Function.prototype --> Object.prototype --> null
function Punto(x,y){
    this.x = x;
    this.y = y;
    this.mvx = function(d){
        this.x += d;
    }
}

// Objeto creado con el constructo punto
o = new Punto(1,2)
// o = Object{ x: 1, x:2, mvx: mvx()} y su cadena de prototipado es:
// o --> Punto.prototype --> Object.prototype --> null
```