Paradigmas de Lenguajes de Programación

Iván Arcuschin

13 de diciembre de 2016

Contents

1 Aspectos de un Lenguaje

1.1 Sintaxis

Es una descripción del conjunto de secuencias de símbolos considerados como programas válidos.

1.2 Semántica

Es la descripción del significado de instrucciones y expresiones. Puede ser formal o informal. La semántica formal puede ser axiomática, operacional o denotacional.

1.3 Sistema de Tipos

Se utiliza para prevenir errores en tiempo de ejecución. En general, requiere anotaciones de tipo en el código fuente. Puede ser:

- Chequeo de tipos estático: se analiza en tiempo de compilación.
- Chequeo de tipos dinámico: se analiza en tiempo de ejecución.

2 Paradigma Imperativo

2.1 Características generales

- Posee un *estado global* y mecanismos de *asignación* y *control de flujo*.
- Computación expresada a través de modificación reiterada del estado global (memoria).
- Utiliza variables como abstracción de celdas de memoria.
- Los resultados intermedios se almacenan en memoria.
- Repetición basada en iteración.
- Tiende a ser **eficiente**: la máquina hace exactamente lo que le pedimos, nada más y nada menos.
- Tiene un **bajo nivel de abstracción**: la diferencia entre implementación y especificación es muy grande.
- La **matemática/lógica de programas tiende a ser más compleja**: ya que el programa se piensa para que lo ejecute la máquina y no para que lo entienda un humano. Acá se ve la diferencia entre semántica Operacional vs Denotacional.

3 Paradigma Funcional

3.1 Características generales

- Computación expresada a través de la aplicación y composición de funciones.
- No hay un estado global.
- Los estados intermedios (salidas de las funciones) son pasados directamente a otras funciones como argumentos.
- Repetición basada en recursión.
- Expresiones tipadas.
- Tiene un **alto nivel de abstracción**: tiende a ser una *especificación ejecutable*.
- Es **declarativo**: la máquina se encarga de buscar el *cómo* y nosotros elegimos el *qué*.
- La matemática y razonamiento algebraico tienden a ser más elegantes.
- En el pasado: ejecución más lenta. Hoy en día no está tan claro que sea así.

3.2 Fundamentos

- Se enfoca en la transformación de la información, y no tanto en la interacción con el medio.
- **Valores:** entidades (matemáticas) abstractas con ciertas propiedades.
- **Expresiones:** cadenas de símbolos utilizadas para denotar valores. Se dividen en:
  - Atómicas: llamadas también formas normales. Ejemplo: `2, False, (3, True)`.
  - Compuestas: se *arman* combinando subexpresiones. Ejemplo: `(1+1), (2==1)`.

  Y pueden estar bien o mal formadas, dependiendo de las reglas sintácticas y de las reglas de asignación de tipos.
- **Funciones:** son valores especiales que representan "transformaciones de datos". Nótese que al ser valores, pueden ser argumento de otras funciones, resultado, almacenarse en estructuras de datos, ser estructuras de datos, etc. Llamamos funciones de **alto orden** a las funciones que trabajan con funciones.
- **Transparencia referencial:** el valor de una expresión depende sólo de los elementos que la constituyen. Esto implica, por ejemplo, que los detalles de ejecución no influencian el comportamiento del programa. Además, existe la posibilidad de demostrar propiedades usando las propiedades de las subexpresiones y métodos de deducción lógica.
- **Ecuaciones orientadas:** tienen la forma $e_1 = e_2$, donde e_1 y e_2 expresan el mismo valor. Decimos que están orientadas porque se puede utilizar e_2 para reemplazar una aparición de e_1 en alguna expresión (pero no al revés).
- **Programa funcional:** es un conjunto de ecuaciones que definen una o más funciones. Para usarlo, resolvemos la reducción de la aplicación de una función a sus datos (reducción de una expresión).

Decimos que un **Lenguaje funcional puro** es aquel que es un "lenguaje de expresiones con transparencia referencial y funciones como valores, cuyo modelo de cómputo es la reducción realizada mediante el remplazo de iguales por iguales".

3.3 Tipos

Toda expresión válida denota un valor. Además, todo valor pertenece a un conjunto, y los tipos denotan conjuntos. Es decir, toda expresión debería tener un tipo para ser válida.

Asignación de tipos: se puede realizar de dos formas:

- De manera explícita, con la sintaxis `e :: A`, que dice que la expresión `e` tiene tipo `A`.
- De manera implícita. Se infiere por el contexto en el cual se usa la expresión.

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

Chequeo de tipos: dada una expresión `e` y un tipo `A`, determinar si `e` tipa `A` (`e :: A`) según las reglas, o no.

Sistema de tipado fuerte (strong typing): acepta una expresión si, y solo si, ésta tiene tipo según las reglas.

Sistema de Hindley-Milner es un sistema de tipos que propone:

- Tipos básicos:
  - Enteros: `Int`
  - Caracteres: `Char`
  - Booleanos: `Bool`
- Tipos compuestos:
  - Tuplas: `(A,B)`
  - Listas: `[A]`
  - Funciones: `(A -> B)`
- **Polimorfismo:** permite que una expresión tenga más de un sólo tipo. El polimorfismo paramétrico soluciona esto usando **variables de tipos**, ejemplo: `id :: a -> a`.

3.3.1 Tipos algebraicos

Es un conjunto de valores junto con las operaciones que se les pueden realizar.

3.3.2 Tipo abstracto (TAD)

3.3.3 Pattern matching

es un mecanismo para acceder a los parámetros con los que fue construido un Tipo Algebraico. **Pattern** es una expresión utilizando constructores y variables sin repetir. **Matching** es la operación que dice si una expresión es de la forma de un patrón.

3.3.4 Tipos algebraicos recursivos

representan conjuntos inductivos, al menos un constructor de tener al menos un parámetro de tipo si mismo.

3.4 Currificación

Establece una correspondencia entre cada función de múltiples parámetros y una de alto orden que retorna una función intermedia que completa el trabajo. Es decir, por cada `f'` definida como:

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

siempre se puede escribir

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

O sea, que hay una correspondencia entre los tipos `(a,b) -> c` y `a -> (b -> c)`. Y se pueden hacer funciones `curry :: ((a,b) -> c) -> (a -> (b -> c))` y `uncurry :: (a -> (b -> c)) -> ((a,b) -> c)` que transformen funciones de un tipo al otro.

Nota: tener en cuenta que decir si algo está currificado o no puede ser una cuestión de interpretación. El ejemplo canónico es la función `distance :: (Int, Int) -> Int` que calcula la distancia euclidea de un punto al origen. Uno podría querer currificar el primer parámetro, pero entonces quizás sería menos obvio que la función espera un punto.

3.5 Haskell

3.5.1 Repaso

- La evaluación consiste en aplicar ecuaciones (orientadas de izquierda a derecha).
- Definición de funciones por casos.
- Tipos básicos: `Int`, `Bool`, `Float`, `Pares`, `Listas`, `Funciones`, etc.
- No toda evaluación termina.
- Utiliza evaluación *Lazy o Normal*: una subexpresión se evalua sólo si es necesario.
- Polimorfismo paramétrico.
- Alto orden
- Currificación

3.5.2 Esquemas de recursión

3.5.3 Monadas

Una mónada es un tipo paramétrico `M a` con operaciones

```
return :: a -> M a
(>>=) :: M a -> (a -> M b) -> M b
```

que satisfacen las siguientes leyes

```
return x >>= k                = k x
m >>= \x -> return x          = m
m >>= \x -> (n >>= \y -> p) = (m >>= \x -> n) >>= \y -> p
    (siempre que x no aparezca en p)
```

El objetivo de la Mónada es incorporar *efectos* a un valor.

- El tipo `M a` incorpora la información necesaria.

- `return x` representa a `x` con *efecto nulo*.
- `(>>=)` *secuencia* efectos con *dependencia* de datos.

Es una forma de abstraer comportamientos específicos en un cómputo. Cada mónada se diferencia de las demás por sus operaciones adicionales:

- `Maybe` tiene `fail`
- `State` tiene `incrementar`
- `Output` tiene `imprimir`
- etc.

Haskell define una clase para las mónadas

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

Por ejemplo, para definir `Maybe` como una mónada se escribe

```
instance Monad Maybe where
  return x = Just x
  m >>= k = case m of
    Nothing -> Nothing
    Just x -> k x
```

Funciones generales Pueden definirse muchas funciones de uso general usando sólo la interfase de mónadas

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f mx = do
  x <- mx
  return (f x)

liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
liftM2 f mx my = do x <- mx
  y <- my
  return (f x y)
```

Estas funciones pueden usarse para definir funciones específicas para la aplicación, por ejemplo:

```
(</>) :: Monad m => m Float -> m Float -> m Float
(</>) = liftM2 (/)

eval :: Monad m => E -> m Float
eval (Cte n) = return n
eval (Div e1 e2) = eval e1 </> eval e2
```

3.6 Lambda cálculo

Fue introducido por *Alonzo Church* en 1934, y presenta un modelo de computación basado en **funciones**.

La formulación original es sin tipos, pero posteriormente (1941) se introduce el **Lambda cálculo tipado**, que es el que vamos a estudiar.

Empezamos con el Lambda cálculo tipado con expresiones booleanas y luego lo vamos extendiendo con otras construcciones.

3.6.1 Expresiones de tipos

Denotan los diferentes **tipos**. Por ejemplo, en el Lambda cálculo tipado con expresiones booleanas (de ahora en más λ^b) tenemos:

$$\sigma, \tau ::= Bool | \sigma \to \tau$$

dónde,

- $Bool$ es el tipo de los booleanos,
- $\sigma \to \tau$ es el tipo de las funciones de tipo σ en τ,
- con σ y τ dos tipos (no necesariamente distintos).

3.6.2 Términos

Denotan las posibles expresiones que podemos construir. Por ejemplo, en λ^b tenemos:

$$M ::= true \mid false \mid if\ M\ then\ P\ else\ Q \mid M\ N \mid \lambda x : \sigma.M \mid x$$

dónde,

- $true$ y $false$ son las constantes de verdad,
- $if\ M\ then\ P\ else\ Q$ es el condicional,
- $M\ N$ es la aplicación de la función denotada por el término M al argumento N,
- $\lambda x : \sigma.M$ es una función cuyo parámetro formal es x y cuyo cuerpo es M.
- x es una variable de términos.

Ejemplo: $(\lambda f : Bool \to Bool.f\ true)\ (\lambda y : Bool.y)$

3.6.3 Sustitución

Dentro de un término, una variable puede estar **libre** o **ligada**. Decimos que x ocurre libre si no se encuentra bajo el alcance de una ocurrencia de λx. En caso contrario, está ligada. Formalmente, tenemos que:

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

Luego, el proceso de sustitución $M\{x \leftarrow N\}$ consiste en sustituir todas las ocurrencias *libres* de x en el término M por el término N. Esto se utilizar para darle semántica a la aplicación de funciones.

Cuando realizamos este proceso, asumimos que la variable ligada se renombró de tal manera que no ocurre libre en N. Por ejemplo:

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

Está mal, ya que convertimos la función constante $\lambda z : \sigma.x$ en la función identidad. Entonces, deberíamos haber utilizado:

$$\lambda w : \sigma.x\{x \leftarrow Z\} = \lambda w : \sigma.z$$

Una vez que entendemos esto, podemos definir el concepto de $\alpha - equivalencia$. Decimos que dos términos M y N son $\alpha - equivalentes$ si difieren solamente en el nombre de sus variables ligadas. Ejemplo: $\lambda z : Bool.z =_\alpha \lambda y : Bool.y$, pero $\lambda x : Bool.y \neq_\alpha \lambda x : Bool.z$.

En los casos de sustitución que haya conflictos, podemos renombrar apropiadamente para que ande todo.

3.6.4 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 llamados *tipados*. Es decir, es un sistema mediante el cual podemos deducir el tipo de un término.

Sin embargo, el objetivo del sistema de tipos de lambda cálculo es asegurar que si un término es cerrado, está bien tipado y termina, entonces evalúa a un valor (a menos que se mantengan formas normales que no sean valores para representar errores).

La correctitud del sistema de tipos se basa en el progreso y la preservación. El progreso indica que si se tiene un término M cerrado y bien tipado, entonces es un valor, o existe un M' tal que el primero evalúa en el segundo. La preservación se refiere a que el progreso preserva tipos.

En resumen:

- Corrección = Progreso + Preservación.
- **Progreso**: si M es cerrado y bien tipado entonces
  - M es un valor
  - o bien existe M' tal que $M \to M'$.

  Esto quiere decir que la evaluación no puede trabarse para términos cerrados y bien tipados que no son valores.
- **Preservación**: si $\Gamma \rhd M : \sigma$ y $M \to N$, entonces $\Gamma \rhd N : \sigma$. Es decir que la evaluación preserva tipos.

Ahora presentamos algunas nociones.

Un *contexto de tipado* Γ es un conjunto de pares $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ donde cada x_i es distinto. Los x_i representan las variables y los σ_i sus tipo asignado.

Un *juicio de tipado* es una expresión de la forma $\Gamma \rhd M : \sigma$ que significa que el término M tiene tipo σ asumiendo el contexto de tipado Γ. Estos juicios se generan utilizando *axiomas y reglas de tipado*. Ejemplo de axioma:

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

Ejemplo de regla:

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

Entonces, M es *tipable* si $\Gamma \rhd M : \sigma$ puede derivarse usando los axiomas y reglas de tipados, para algún Γ y σ.

Propiedades básicas:

- **Unicidad de tipos**: si $\Gamma \rhd M : \sigma$ y $\Gamma \rhd M : \tau$ son derivables, entonces $\sigma = \tau$.
- **Weakening+Strenghening**: 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.

3.6.5 Semántica operacional

Habiendo definido la sintaxis de λ^b, nos interesa formular cómo se evalúan o ejecutan los términos. Esto se puede hacer de varias formas: operacional, denotacional y axiomática. Nosotros vamos a usar *operacional*.

La semántica operacional consiste en interpretar a los *términos como estados* de una máquina abstracta y definir una *función de transición* que indica, dado un estado, cual es el estado siguiente. De esta forma, el *significado* de un término M es el estado final que alcanza la máquina empezando desde M. Hay principalmente dos formas de definir la función de transición:

- **Small-step**: la función de transición describe un paso de computación.
- **Big-step**: la función de transición, en un paso, evalúa el término a su resultado.

Al igual que con el Sistema de tipado, vamos a usar axiomas y reglas para formular **juicios de evaluación** $M \to N$ que indican que el término M reduce, en un paso, al término N. Ejemplo de axioma:

$$\frac{}{if\ true\ then\ M_2\ else\ M_3 \to M_2} \text{ (E-IfTrue)}$$

Ejemplo de regla:

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

Cuando dado un término M, no existe N tal que $M \to N$, decimos que M no puede reducirse más y está en **forma normal**.

Lo último que nos falta agregar es un conjunto de **Valores** que denota las expresiones que pueden ser un resultado válido de un cómputo. Para λ^b, tenemos que $V ::= true \mid false$.

Propiedades básicas:

- **Determinismo del juicio de evaluación en un paso**: si $M \to M'$ y $M \to M''$, entonces $M' = M''$.
- Todo valor está en forma normal, pero no vale el recíproco.
- Si un termino está en forma normal pero no es un valor, entonces decimos que es un **estado de error**.

El **Juicio de evaluación en muchos pasos** $\twoheadrightarrow$ es la clausura reflexiva, transitiva de $\rightarrow$. Es decir, la menor relación tal que:

- Si $M \rightarrow M'$, entonces $M \twoheadrightarrow M'$.
- $M \twoheadrightarrow M$, para todo M.
- Si $M \twoheadrightarrow M'$ y $M' \twoheadrightarrow M''$, entonces $M \twoheadrightarrow M''$.

Propiedades de evaluación en muchos pasos:

- **Unicidad de formas normales**: si $M \twoheadrightarrow U$ y $M \twoheadrightarrow V$, entonces $U = V$.
- **Terminación**: para todo M existe una forma normal N tal que $M \twoheadrightarrow N$.
- Si un término está bien tipado, y termina, entonces evalúa a un valor.

3.6.6 Extensiones

La versión más usada en la materia del Lambda Cálculo es λ^{bn}: que tiene $Bool$, Nat y funciones, pero existen varias extensiones que aportan nuevas herramientas.

Cuando definimos una extensión del Cálculo λ tenemos que re-definir (o indicar que es lo que se agrega):

- Expresiones válidas
- Tipos
- Reglas de tipado
- Valores
- Reglas de semántica

Lambda Cálculo λ^{bn} Este es el más común de todos. Las expresiones válidas son:

$$M ::= x \mid true \mid false \mid if\ M\ then\ M\ else\ M \mid \lambda x : \sigma . M \mid M\ M \mid 0 \mid succ(M) \mid pred(M) \mid iszero(M)$$

Los tipos:

$$\sigma ::= Bool \mid Nat \mid \sigma \rightarrow \rho$$

Las reglas de tipado:

$$\frac{}{\Gamma \rhd true : Bool}\ \text{(T-True)} \qquad \frac{}{\Gamma \rhd false : Bool}\ \text{(T-False)} \qquad \frac{x : \sigma \in \Gamma}{\Gamma \rhd x : \sigma}\ \text{(T-Var)}$$

$$\frac{}{\Gamma \rhd 0 : Nat}\ \text{(T-Zero)} \qquad \frac{\Gamma \rhd M : Nat}{\Gamma \rhd iszero(M) : Bool}\ \text{(T-IsZero)}$$

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

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

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

Los valores:

$$V ::= true \mid false \mid \lambda x : \sigma.M \mid \underline{n}$$

donde $\underline{n}$ abrevia $succ^n(0)$.

Las reglas de semántica:

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

$$\frac{}{if\ true\ then\ M_2\ else\ M_3 \to M_2}\ \text{(E-IfTrue)} \qquad \frac{}{if\ false\ then\ M_2\ else\ M_3 \to M_3}\ \text{(E-IfFalse)}$$

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

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

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

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

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

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

Registros Llamada $\lambda^{\cdots r}$. Permite utilizar descriptores de campos de la pinta: $\{nombre : String, edad : Nat\}$.

Agregamos al conjunto de tipos:

$$\sigma :: \ldots \mid \{I_i : {\sigma_i}^{i \in 1..n}\}$$

Y al conjunto de expresiones:

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

Informalmente:

- El registro $\{I_i = {M_i}^{i \in 1..n}\}$ evalúa a $\{I_i = {V_i}^{i \in 1..n}\}$ donde V_i es el valor al que evalúa M_i, $i \in 1..n$.
- $M.I$: evaluar M hasta que arroje $\{I_i = {V_i}^{i \in 1..n}\}$, luego proyectar el campo correspondiente.

Entonces, las reglas de tipado son:

$$\frac{\Gamma \rhd M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \rhd \{I_i = {M_i}^{i \in 1..n}\} : \{I_i : {\sigma_i}^{i \in 1..n}\}} \text{ (T-Rcd)}$$

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

Los valores se expanden a:

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

Y las reglas de semántica operacional son:

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

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

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

Declaraciones locales Esta extensión nos ayuda a mejorar la legibilidad. Agregamos a las expresiones:

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

Informalmente, $let\ x : \sigma = M\ in\ N$ evalúa M a un valor V, liga la variable x a V y evalúa N. Esta extensión no agrega tipos nuevos.

Las reglas de tipado son:

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

Y la semántica operacional:

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

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

Referencias En la extensión anterior $let\ x : Nat = \underline{2}\ in\ M$

- x es una variable declarada con valor 2.
- El valor de x permanece *inalterado* a lo largo de la evaluación de M.
- En este sentido, x es *inmutable*: no existe una operación de asignación.

En programación imperativa pasa todo lo contrario: Todas las variables son mutables. Entonces, vamos a extender Cálculo Lambda Tipado con variables mutables (y lo llamamos λ^{bnru}, pero la r es de referencias).

Las operaciones básicas que vamos a manejar son:

- 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.
- Asignación: $x := M$ almacena en la referencia x el valor de M.

Estas 3 operaciones son **comandos**: expresiones que no interesan por su valor, sino por su **efecto**. Entonces, decimos que se evalúan para causar un efecto, y lo expresamos con un nuevo valor *unit*.

En este momento nos estamos alejando de lo que serían un lenguaje funcional puro, en el que las expresiones son puras en el sentido de carecer de efectos.

Los tipos de λ^{bnu} son:

$$\sigma :: Bool \mid Nat \mid \sigma \to \rho \mid Unit \mid Ref\sigma$$

Las expresiones nos quedan:

$$M :: x \mid \lambda x : \sigma.M \mid M\ N \mid unit \mid ref\ M \mid !M \mid M := N \mid l$$

Notar que $Unit$ es un tipo unitario: el único valor posible de una expresión de ese tipo es *unit*. Cumple un rol similar a *void* en C o Java. Por otro lado, $Ref\sigma$ es el tipo de las referencias a valores de tipo σ.

En primer lugar, podemos introducir una forma de evaluar varias expresiones en **secuencia**:

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

Entonces, la evaluación de $M_1; M_2$ consiste en primero evaluar M_1 y luego M_2. Este comportamiento funciona debido a las reglas de evaluación definidas previamente para λ^{bn}.

El conjunto de valores son:

$$V ::= true \mid false \mid 0 \mid n \mid unit \mid \lambda x : \sigma.M \mid l$$

Y las reglas de tipado:

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

$$\frac{\Gamma \rhd M_1 : Ref\ \sigma}{\Gamma \rhd !M_1 : \sigma}\ \text{(T-DeRef)} \qquad \frac{\Gamma \rhd M_1 : Ref\ \sigma \quad \Gamma \rhd M_2 : \sigma}{\Gamma \rhd M_1 := M_2 : Unit}\ \text{(T-Assign)}$$

Para formalizar la semántica operacional, primero debemos preguntarnos: ¿Qué es una referencia? Rta: Es una abstracción de una porción de memoria que se encuentra en uso. Entonces debemos formalizar el concepto de memoria o *store*:

- Usaremos direcciones (simbólicas) o "locations" $l, l_i \in \mathcal{L}$ para representar referencias, con lo que el *store* no es más que una función parcial de direcciones a valores.
- Usamos las letras μ, μ' para referirnos a stores.
- Notación:
  - $\mu[l \mapsto V]$ es el store resultante de pisar $\mu(l)$ con V.
  - $\mu \oplus (l \mapsto V)$ es el store extendido resultante de ampliar μ con una nueva asociación $l \mapsto V$ (asumimos $l \notin Dom(\mu)$).
- Además, $\Gamma \rhd l :?$ depende de los valores que se almacenen en la dirección l. Esta situación es parecida a las variables libres, con lo que precisamos un "contexto de tipado" para direcciones: Σ función parcial de direcciones a tipos. Con esto, armamos un nuevo juicio de tipado: $\Gamma|\Sigma \rhd M : \sigma$.

- Y los juicios de evaluación, en vez de ser de la forma $M \to M'$, ahora son $M \mid \mu \to M' \mid \mu'$. Que se lee como: "Teniendo la expresión M, el store μ reduce la expresión a M', y nos queda el store μ'".

Entonces, las reglas de tipado nos terminan quedando:

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

$$\frac{\Gamma|\Sigma \rhd M_1 : Ref\ \sigma}{\Gamma|\Sigma \rhd !M_1 : \sigma} \text{ (T-DeRef)} \qquad \frac{\Gamma|\Sigma \rhd M_1 : Ref\ \sigma \quad \Gamma|\Sigma \rhd M_2 : \sigma}{\Gamma|\Sigma \rhd M_1 := M_2 : Unit} \text{ (T-Assign)}$$

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

Y la semántica operacional:

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

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

Notar que hasta ahora las reglas no modifican el store.

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

$$\frac{M_1 \mid \mu \to M_1' \mid \mu'}{M_1 := M_2 \mid \mu \to M_1' := M_2 \mid \mu'} \text{ (E-Assign1)} \qquad \frac{M_2 \mid \mu \to M_2' \mid \mu'}{V := M_2 \mid \mu \to V := M_2' \mid \mu'} \text{ (E-Assign2)}$$

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

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

Preservación y progreso Habíamos visto antes que *Corrección = Preservación + Progreso*. Sin embargo, al introducir el sistema de tipos para las direcciones (Σ) rompimos ambos conceptos.

Esto se puede ver teniendo en cuenta que $\Gamma|\Sigma \rhd M : \sigma$ y $M \mid \mu \to M' \mid \mu'$ implica $\Gamma|\Sigma \rhd M' : \sigma$. Pero si suponemos, por ejemplo, que: $M = !l, \Gamma = \emptyset, \Sigma(l) = Nat, \mu(l) = true$, entonces $\Gamma|\Sigma \rhd M : Nat$ y $M \mid \mu \to true \mid \mu$, pero $\Gamma|\Sigma \rhd true : Nat$ no vale.

Para la *preservación*, precisamos una noción de compatibilidad entre el store y el contexto de tipado para store. Entonces, debemos tipar los stores. Con este objetivo introducimos un nuevo juicio de tipado: $\Gamma|\Sigma \rhd \mu$, y se define del siguiente modo: $\Gamma|\Sigma \rhd \mu$ sii

1. $Dom(\Sigma) = Dom(\mu)$ y
2. $\Gamma|\Sigma \rhd \mu(l) : \Sigma(l)$ para todo $l \in Dom(\mu)$.

Y tenemos una nueva formulación de *preservación*. Si pasa:

- $\Gamma|\Sigma \rhd M : \sigma$
- $M \mid \mu \to N \mid \mu'$

- $\Gamma|\Sigma \rhd \mu$

implica que existe $\Sigma' \supseteq \Sigma$ tal que $\Gamma|\Sigma' \rhd N : \sigma$ y $\Gamma|\Sigma' \rhd \mu'$. Donde el Σ' se utiliza ya que Σ puede haber crecido en dominio por posibles reservas de memoria.

Para el *progreso*. Si M es cerrado y bien tipado (i.e. $\emptyset|\Sigma \rhd M : \sigma$, para algún Σ, σ) entonces:

- M es un valor,
- o bien para cualquier store μ tal que $\emptyset|\Sigma \rhd \mu$, existe M' y μ' tal que $M \mid \mu \to M' \mid \mu'$

Recursión Una ecuación recursiva es aquella que tiene la pinta $f = \ldots f \ldots f \ldots$ (se usa a si misma en la definición). Esta extensión permite construir funciones recursivas.

Agregamos a las expresiones:

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

No se precisan nuevos tipos, pero sí una regla de tipado:

$$\frac{\Gamma \rhd M : \sigma_1 \to \sigma_1}{\Gamma \rhd fix\ M : \sigma_1}\ \text{(T-Fix)}$$

No hay valores nuevos. La semántica operacional es:

$$\frac{M_1 \to M_1'}{fix\ M_1 \to fix\ M_1'}\ \text{(E-Fix)} \qquad \frac{}{fix\ (\lambda x : \sigma.M) \to M\{x \leftarrow fix\ (\lambda x : \sigma.M)\}}\ \text{(E-FixBeta)}$$

3.6.7 Inferencia de tipos

Es un procedimiento que consiste en transformar términos *sin* información de tipos en términos **tipables**. Para ello, debemos inferir la información de tipos faltante. Esto trae como beneficios:

- El programador puede obviar algunas declaraciones de tipos.
- En general, evita la sobrecarga de tener que declarar y manipular *todos* los tipos.
- Todo ello sin impactar la perfomance del programa: la inferencia de tipos se realiza en tiempo de compilación.

Función de borrado Llamamos $\textsc{Erase}(\cdot)$ a la función que dado un término *elimina* las anotaciones de tipos de las abstracciones. Ejemplo: $\textsc{Erase}(\lambda x : Nat.\lambda f : Nat \to Nat.f\ x) = \lambda x.\lambda f.f\ x$

Entonces queremos, dado un término U *sin* anotaciones de tipos, hallar un término estándar M tal que:

- $\Gamma \rhd M : \sigma$, para algún Γ y σ, y
- $\textsc{Erase}(M) = U$

Notar que este problema es más difícil que el chequeo de tipos para un término estándar M que hacíamos utilizando los axiomas y reglas del sistema de tipado.

Variables de tipos Para poder escribir una *única* expresión que englobe muchas soluciones de inferencia vamos a usar variables de tipo.

Dichas variables las vamos a representar con la letra "s". Si tenemos una expresión que utiliza s, basta con sustituir s por cualquier expresión de tipos para arrojar una solución válida.

Sustitución de tipos Es una función S o T que mapea variables de tipo en expresiones de tipo. Entonces, nuestra función de sustitución S puede aplicarse a:

- Una expresión de tipos σ (escribimos $S\sigma$)
- Una término M σ (escribimos SM)
- Un contexto de tipado $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ (escribimos $S\Gamma$) y se define como $S\Gamma \stackrel{\text{def}}{=} \{x_1 : S\sigma_1, \ldots, x_n : S\sigma_n\}$

Notas sobre la sustitución:

- El conjunto $\{t \mid St \neq t\}$ se llama **soporte** de S. Este conjunto representa las variables que S "afecta".
- Usamos la notación $\{\sigma_1/t_1, \ldots, \sigma_n/t_n\}$ para la sustitución con soporte $\{t_1, \ldots, t_n\}$.
- La sustitución cuyo soporte es $\emptyset$ es la sustitución identidad (Id).

Instancia de un juicio de tipado Un juicio de tipado $\Gamma' \rhd M' : \sigma'$ es una instancia de $\Gamma \rhd M : \sigma$ si existe una substitución de tipos S tal que $S\Gamma \subset \Gamma'$, $M' = SM$ y $\sigma' = S\sigma$. Como propiedad: si $\Gamma \rhd M : \sigma$ es derivable, entonces cualquier instancia del mismo también lo es.

Función de inferencia $\mathbb{W}(\cdot)$ Entonces, queremos definir una función $\mathbb{W}(\cdot)$ que dado un término U sin anotaciones verifica:

- **Corrección:** $\mathbb{W}(U) = \Gamma \rhd M : \sigma$ implica
  - $\textsc{Erase}(M) = U$
  - $\Gamma \rhd M : \sigma$ es derivable
- **Completitud:** si $\Gamma \rhd M : \sigma$ es derivable y $\textsc{Erase}(M) = U$, entonces
  - $\mathbb{W}(U)$ tiene éxito y,
  - produce un juicio $\Gamma' \rhd M' : \sigma'$ tal que $\Gamma \rhd M : \sigma$ es instancia del mismo. Se dice que $\mathbb{W}(\cdot)$ computa un **tipo principal**.

3.6.8 Unificación

La idea de nuestro algoritmo de inferencia va a ser analizar un término (sin anotaciones de tipo) a partir de sus subtérminos. El problema es que, una vez obtenida la información inferida de los subtérminos, debemos:

- **Consistencia:** determinas si la información de cada subtérmino es consistente.
- **Síntesis:** sintetizar la información del término original a partir de la información de los subtérminos. Esto involucra *compatibilizar* o *unificar* la información de tipos cuando dos subtérminos tienen una misma variable.

El proceso de determinar si existe un sustitución S tal que dos expresiones de tipos σ, τ son unificables (i.e. $S\sigma = S\tau$) se llama **unificación**.

Algunas propiedades de sustituciones

- **Composición:** $(S \circ T)(\sigma) = S(T(\sigma))$
- **Igualdad:** $S = T$ si tienen el mismo soporte y $St = Tt$ para todo t en el soporte de S
- $S \circ Id = Id \circ S = S$
- $S \circ (T \circ U) = (S \circ T) \circ U$
- Una sustitución S es **más general** que T si existe U tal que $T = U \circ S$.

Unificador más general (MGU) Una ecuación de unificación es una expresión de la forma $\sigma_1 \doteq \sigma_2$. Una sustitución S es una solución de un conjunto de ecuaciones de unificación $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$ si $S\sigma_i = S\sigma'_i$.

Una sustitución S es un **MGU** de $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$ si

- Es solución de $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$
- Es más general que cualquier otra solución de $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$

Algoritmo de unificación de Martelli-Montanari

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

El algoritmo de Martelli-Montanari es un algoritmo de unificación no-determinístico. Consiste en reglas de simplificación que reescriben conjuntos de pares de tipos a unificar (*goals*).

$$G_0 \mapsto G_1 \mapsto_{S_1} G_2 \mapsto \ldots \mapsto_{S_k} G_n$$

Las secuencias que terminan en el goal vació son exitosas. Si la secuencia es exitosa el MGU es $S_k \circ \cdots \circ S_1$.

Las reglas de este algoritmo son:

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:** $\{s \doteq s\} \cup G \mapsto G$
3. **Swap:** si σ no es una variable, $\{\sigma \doteq s\} \cup G \mapsto \{s \doteq \sigma\} \cup G$
4. **Eliminación de variable:** $\{s \doteq \sigma\} \cup G \mapsto_{\sigma/s} G[\sigma/s]$
5. **Falla:** $\{\sigma \doteq \tau\} \cup G \mapsto$ falla, con $(\sigma, \tau) \in T \cup T^{-1}$ y $T = \{(Boo, Nat), (Nat, \sigma_1 \to \sigma_2), (Bool, \sigma_1 \to \sigma_2)\}$,
6. **Occur check:** si $s \neq \sigma$ y $s \in FV(\sigma)$, entonces $\{s \doteq \sigma\} \cup G \mapsto$ falla

Teorema. *Propiedades del algoritmo*

- *El algoritmo de Martelli-Montanari siempre termina*
- *Sea G un conjunto de pares*
  - *Si G tiene un unificador, el algoritmo termina exitosamente y retorna un MGU.*
  - *Si G no tiene unificador, el algoritmo termina con* `falla`.
- *Tanto la unificación como la inferencia se pueden hacer en tiempo lineal. Pero el tipo asociado a un término sin anotaciones puede ser exponencial en el tamaño del término.*

3.6.9 Algoritmo de inferencia

El algoritmo que vamos a presentar se basa en definir $\mathbb{W}(U)$ por recursión sobre la estructura de U, y utiliza el algoritmo de unificación para ir sintetizando la información de las subexpresiones de U.

Presentamos las reglas para los casos comunes:

Caso constantes y variables

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

Caso succ

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

donde

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

El caso de *pred* es similar.

Caso iszero

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

donde

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

Caso ifThenElse

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

donde

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

Caso abs

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

donde

- $\mathbb{W}(U) = \Gamma \rhd M : \rho$
- $\tau' = \begin{cases} \alpha & \text{si } x : \alpha \in \Gamma \\ s \text{ variable fresca} & \text{en otro caso} \end{cases}$
- $\Gamma' = \Gamma \ominus \{x\}$

Caso app

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

donde

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

Conclusiones de las reglas de inferencia

- Los llamados recursivos devuelven un contexto, un término anotado y un tipo. No podemos asumir nada sobre ellos.
- Cuando la regla tiene tipos iguales, unificar.
- Si hay contextos repetidos en las premisas, unificarlos.
- Cuando la regla liga variables:
  - Obtener su tipo del Γ obtenido recursivamente.
  - Si no figuran: variable fresca.
  - Sacarlas del Γ del resultado (y del que se vaya a unificar), ya que ya no son variables libres.
- Decorar los términos según corresponda.
- Si la regla tiene restricciones adicionales, se incorporan como posibles casos de falla.

4 Paradigma Lógico

4.1 Características generales

- Se basa en el uso de la lógica como un lenguaje de programación: los programas son predicados.
- La computación se expresa a través de "proof search". Para esto, se especifican ciertos **hechos** y **reglas**, así como un objetivo o **goal** a probar. Luego, un motor de inferencia trata de probar que el objetivo es consecuencia de los hechos y reglas.
- No hay un estado global.
- Los resultados intermedios son pasados a través de unificación.
- Repetición basada en recursión.
- Tiene un **alto nivel de abstracción**: tiende a ser una *especificación ejecutable*.
- Es **declarativo**: la máquina se encarga de buscar el *cómo* y nosotros elegimos el *qué*.
- Fundamento lógico robusto. Utiliza técnicas de *Resolución*.
- Ejecución lenta en comparación con otros paradigmas.

4.2 Repaso lógica proposicional

Sintaxis Dado un conjunto $\mathcal{V}$ de variables proposicionales, definimos inductivamente el conjunto de formulas proposicionales de la siguiente manera:

- Una variable proposicional $P_0, P_1, \ldots$ es una proposición.
- Si A, B son proposiciones, entonces:
  - $\neg A$ es una proposición.
  - $A \wedge B$ es una proposición.
  - $A \vee B$ es una proposición.
  - $A \supset B$ es una proposición.
  - $A \iff B$ es una proposición.

Semántica Una **valuación** es una función $v : \mathcal{V} \to \{T, F\}$ que asigna valores de verdad a las variables proposicionales.

Una valuación **satisface** una proposición A si $v \models A$ donde

$$
\begin{aligned}
v \models P &\text{ sii } v(P) = T \\
v \models \neg A &\text{ sii } v \not\models A \text{ (que es lo mismo que } \neg v \models A) \\
v \models A \wedge B &\text{ sii } v \models A \text{ y } v \models B \\
v \models A \vee B &\text{ sii } v \models A \text{ o } v \models B \\
v \models A \supset B &\text{ sii } v \not\models A \text{ o } v \models B \\
v \models A \iff B &\text{ sii } (v \models A \text{ sii } v \models B)
\end{aligned}
$$

Tautologías y satisfacibilidad Una proposición A es:

- Una tautología si $v \models A$ para toda valuación v.
- Satisfacible si existe una valuación v tal que $v \models A$.
- Insatisfacible si no es satisfacible.

Un conjunto de proposiciones S es

- Satisfacible si existe una valuación v tal que para todo $A \in S$, se tiene $v \models A$.
- Insatisfacible si no es satisfacible.

Teorema. *Una proposición A es una tautología sii $\neg A$ es insatisfacible.*

Proof.
$\Longrightarrow$. Si A es tautología, para toda valuación v, $v \models A$. Entonces $v \not\models \neg A$.
$\Longleftarrow$. Si $\neg A$ es insatisfacible, para toda valuación v, $v \not\models \neg A$. Entonces $v \models A$. □

Forma Normal Conjuntiva (FNC) Un literal es una variable proposicional P o su negación $\neg P$.

Una proposición A está en FNC si es una conjunción

$$C_1 \wedge \cdots \wedge C_n$$

donde cada C_i (llamado cláusula) es una disyunción

$$B_{i1} \vee \cdots \vee B_{in}$$

y cada B_{ij} es un literal.
Entonces, una FNC es una *conjunción de disyunciones de literales.*

Teorema. *Para toda proposición A puede hallarse una proposición A' en FNC que es lógicamente equivalente a A.*

Notación conjuntista para FNC Dado que tanto $\vee$ como $\wedge$

- Son conmutativos.
- Son asociativos.
- Son idempotentes.

Podemos asumir que

- Cada cláusula C_i es distinta.
- Cada cláusula puede verse como un conjunto de literales distintos.

Consecuentemente, para una FNC podemos usar la notación

$$\{C_1, \ldots, C_n\}$$

donde cada C_i es un conjunto de literales

$$\{B_{i1}, \ldots, B_{in}\}$$

4.3 Método de resolución para lógica proposicional

El método de **Resolución** fue introducido en 1965 y se basa en el principio de demostración por refutación: probar que A es válido mostrando que $\neg A$ es insatisfacible.

Además, se basa en el hecho que el conjunto de el conjunto de cláusulas

$$\{C_1, \ldots, C_m, \{A, P\}, \{B, \neg P\}\}$$

es lógicamente equivalente a

$$\{C_1, \ldots, C_m, \{A, P\}, \{B, \neg P\}, \{A, B\}\}$$

En consecuencia, el conjunto de cláusulas

$$\{C_1, \ldots, C_m, \{A, P\}, \{B, \neg P\}\}$$

es insatisfacible sii

$$\{C_1, \ldots, C_m, \{A, P\}, \{B, \neg P\}, \{A, B\}\}$$

es insatisfacible.

Las cláusula $\{A, B\}$ se llama resolvente de las cláusulas $\{A, P\}$ y $\{B, \neg P.\}$. El resolvente de las cláusulas $\{P\}$ y $\{\neg P\}$ es la cláusula vacía y se anota $\square$.

Entonces, la regla de resolución nos queda:

$$\frac{\{A_1, \ldots, A_m, Q\}\ \{B_1, \ldots, B_n, \neg Q\}}{\{A_1, \ldots, A_m, B_1, \ldots, B_n\}}$$

En el método de resolución, cada **paso de resolución** consiste en agregar a un conjunto S la resolvente C de dos cláusulas C_1, C_2 que pertenecen a S (asumimos que C no pertenecía a S). Lo importante es que cada paso de resolución preserva la insatisfacibilidad, por lo que S es insatisfacible sii $S \cup \{C\}$ es insatisfacible.

Por último, un conjunto de cláusulas se llama una **refutación** si contiene a la cláusula vacía ($\square$), que es insatisfacible.

El método de resolución trata de construir una secuencia de conjuntos de cláusulas, obtenidas usando pasos de resolución hasta llegar a una refutación.

$$S_1 \implies S_2 \implies \cdots \implies S_n \ni \square$$

Con lo cual, se sabe que el conjunto inicial de cláusulas es insatisfacible.

Terminación de la regla de resolución La aplicación reiterada de esta regla siempre termina (suponiendo que cada resolvente que se agrega es nuevo). Esto se puede ver ya que:

- El resolvente se forma con los literales distintos que aparecen en el conjunto de cláusulas de partida S.
- Hay una cantidad finita de literales en el conjunto de cláusulas de partida S.

Teorema. *Dado un conjunto finito S de cláusulas, S es insatisfacible sii tiene una refutación.*

Recapitulando Para probar que A es una tautología:

1. Calculamos la FNC de $\neg A$
2. Aplicamos el método de resolución.
3. Si hallamos una refutación, $\neg A$ es insatisfacible, y por lo tanto A es una tautología.
4. Si no, $\neg A$ es satisfacible, y por lo tanto A no es una tautología.

4.4 Repaso lógica de primer orden

Sintaxis Un lenguaje de primer orden $\mathcal{L}$ consiste en:

- Un conjunto numerable de constantes $c_0, c_1, \ldots$
- Un conjunto numerable de símbolos de función con aridad $n > 0$, $f_0, f_1, \ldots$
- Un conjunto numerable de símbolos de predicado con aridad $n \geq 0$, $P_0, P_1, \ldots$.

Sea $\mathcal{V}$ un conjunto numerable de variables. El conjunto de $\mathcal{L}$-términos se define inductivamente como:

- Toda constante de $\mathcal{L}$ y toda variable es un $\mathcal{L}$-término
- Si $t_1, \ldots, t_n \in \mathcal{L}$-términos y f es un símbolo de función de aridad n, entonces $f(t_1, \ldots, t_n) \in \mathcal{L}$-términos

El conjunto de $\mathcal{L}$-fórmulas atómicas se define inductivamente como:

- Todo símbolo de predicado de aridad 0 es una $\mathcal{L}$-fórmula atómica
- Si $t_1, \ldots, t_n \in \mathcal{L}$-términos y P es un símbolo de predicado de aridad n, entonces $P(t_1, \ldots, t_n) \in \mathcal{L}$-fórmulas atómicas

El conjunto de $\mathcal{L}$-fórmulas se define inductivamente como:

- Toda $\mathcal{L}$-fórmula atómica es una $\mathcal{L}$-fórmula
- Si $A, B \in \mathcal{L}$-fórmulas, entonces $(A \wedge B)$, $(A \vee B)$, $(A \supset B)$, $(A \iff B)$ y $\neg A$ son $\mathcal{L}$-fórmulas
- Para toda variable x_i y cualquier $\mathcal{L}$-fórmula A, $\forall x_i.A$ y $\exists x_i.A$ son $\mathcal{L}$-fórmulas

Las variables pueden ocurrir libres o ligadas. Los cuantificadores ligan variables. Usamos $FV(A)$ y $BV(A)$ para referirnos a las variables libres y ligadas respectivamente de A.
Una fórmula A se dice **rectificada** si

- $FV(A)$ y $BV(A)$ son disjuntos y
- Cuantificadores distintos de A ligan variables distintas.

Toda fórmula se puede rectificar (renombrando variable ligadas) a una fórmula lógica equivalente. Una sentencia es una fórmula cerrada (sin variables libres).

Semántica Dado un lenguaje de primer orden $\mathcal{L}$, una estructura para $\mathcal{L}$ es un par (M, I) donde

- M (dominio) es un conjunto no vacío.
- I (función de interpretación) asigna funciones y predicados sobre M a símbolos de $\mathcal{L}$ de la siguiente manera:
  1. Para toda constante c, $I(c) \in M$
  2. Para toda función f de aridad $n > 0$, $I(f) : M^n \to M$
  3. Para todo predicado P de aridad $n \geq 0$, $I(P) : M^n \to \{T, F\}$

Sea (M, I) una estructura para $\mathcal{L}$. Una asignación es una función $s : \mathcal{V} \to M$, y la relación $s \models_{(M,I)} A$ establece que la asignación s satisface la fórmula A en la estructura (M, I).
Entonces,

- Una fórmula A es satisfacible en (M, I) sii existe una asignación s tal que $s \models_{(M,I)} A$.
- Una fórmula A es satisfacible sii existe una estructura (M, I) tal que A es satisfacible en (M, I).
- Una fórmula A es válida en (M, I) sii $s \models_{(M,I)} A$, para toda asignación s.
- Una fórmula A es válida sii es válida en toda estructura (M, I).
- Luego, A es válida sii $\neg A$ es insatisfacible.

Teorema (Teorema de Church)**.** *No existe un algoritmo que pueda determinar si una fórmula de primer orden es válida.*

Como consecuencia, el método de resolución que mostramos no es un procedimiento efectivo, sino que es un algoritmo de semi-decision:

- *Si una sentencia es insatisfacible hallará una refutación,*
- *Pero si es satisfacible puede que no se detenga.*

4.5 Método de resolución para lógica de primer orden

Cuando trabajamos en lógica de primer orden también tenemos una FNC en notación de conjuntos, Forma clausal, pero requiere tener en cuenta los cuantificadores ($\forall$ y $\exists$). Podemos llevar una fórmula de primer orden a Forma clausal con los siguientes pasos:

1. **Eliminar implicaciones**.
2. **Pasar a Forma normal negada**.
3. **Pasar a Forma normal prenexa (opcional)**.
4. **Pasar a Forma normal de Skolem (puede hacerse antes que el paso anterior)**.
5. **Pasar a Forma normal conjuntiva**.
6. **Distribuir cuantificadores universales**.

Todos estos pasos preservan validez lógica, salvo la *Skolemización*, que preserva satisfacibilidad. Vamos cada paso en detalle.

Eliminar implicaciones Consiste en escribir la fórmula en términos de $\wedge, \vee, \neg, \forall, \exists$.

Forma normal negada (FNN) Se define inductivamente como:

- Para cada fórmula atómica A, A y $\neg A$ están en FNN.
- Si $A, B \in$ FNN entonces $(A \wedge B), (A \vee B) \in$ FNN.
- Si $A \in$ FNN, entonces $\forall x.A, \exists x.A \in$ FNN.

Y podemos definir las siguientes reglas para hacer el pasaje más sencillo:

- $\neg(A \wedge B)$ pasa a $(\neg A \vee \neg B)$
- $\neg(A \vee B)$ pasa a $(\neg A \wedge \neg B)$
- $\neg\neg A$ pasa a A
- $\neg\forall x.A$ pasa a $\exists x.\neg A$
- $\neg\exists x.A$ pasa a $\forall x.\neg A$

Forma normal prenexa Consiste en pasar todos los cuantificadores ($\forall$ y $\exists$) al principio de la fórmula. También contamos con reglas para ayudar a hacer el pasaje:

- $(\forall x.A) \wedge B$ pasa a $\forall x.(A \wedge B)$
- $(A \wedge \forall x.B)$ pasa a $\forall x.(A \wedge B)$
- $(\exists x.A) \wedge B$ pasa a $\exists x.(A \wedge B)$
- $(A \wedge \exists x.B)$ pasa a $\exists x.(A \wedge B)$
- $(\forall x.A) \vee B$ pasa a $\forall x.(A \vee B)$
- $(A \vee \forall x.B)$ pasa a $\forall x.(A \vee B)$
- $(\exists x.A) \vee B$ pasa a $\exists x.(A \vee B)$
- $(A \vee \exists x.B)$ pasa a $\exists x.(A \vee B)$

Forma normal de Skolem El objetivo de este paso es

- Eliminar los cuantificadores existenciales ($\exists$)
- Sin alterar la satisfacibilidad (y por lo tanto la insatisfacibilidad).

Para esto, introducimos *testigos* (parámetros), tal que:

- Todo cuantificador existencial se reemplaza por una constante o función de Skolem. Ejemplo $\exists x.P(x)$ pasa a $P(c)$, con c una nueva constante.
- Cada ocurrencia de una subfórmula $\exists x.B$ se reemplaza en la fórmula A por $B\{x \leftarrow f(x_1, \ldots, x_n)\}$, donde f es un símbolo de función nuevo y las $x_1, \ldots, x_n$ son las variables de las que depende x en B. Por ejemplo $\forall x.\exists y.P(x, y)$ pasa a $\forall x.P(x, g(x))$.

Nota: no es posible eliminar los cuantificadores existenciales sin alterar la validez. Ejemplo: $\exists x.(P(a) \supset P(x))$ es válida pero $P(a) \supset P(b)$ no lo es.

La forma normal de Skolem de A (escrito como $SK(A)$) se puede definir recursivamente. Sea A' cualquier subfórmula de A:

- Si A' es una fórmula atómica o su negación, $SK(A') = A'$.
- Si A' es de la forma $(B \star C)$ con $\star \in \{\wedge, \vee\}$, entonces $SK(A') = (SK(B) \star SK(C))$.
- Si A' es de la forma $\forall x.B$, entonces $SK(A') = \forall x.SK(B)$.
- Si A' es de la forma $\exists x.B$ y $\{x, y_1, \ldots, y_m\}$ son las variables libres de B (que se ligan en A, dado que A es sentencia), entonces
  - Si $m > 0$, crear un nuevo símbolo de función de Skolem, f_x de aridad m y definir $SK(A') = SK(B\{x \leftarrow f_x(y_1, \ldots, Y_m)\})$
  - Si $m = 0$, crear una nueva constante de Skolem c_x y definir $SK(A') = SK(B\{x \leftarrow c_x\})$

  Notar que dado que A está rectificada, cada f_x y c_x es única.

Tip: siempre conviene skolemizar de afuera hacia adentro, para evitar cambios innecesarios.

Forma normal conjuntiva (FNC) Este paso es idéntico a como hacíamos cuando estábamos trabajando con fórmulas proposicionales. Llevamos la fórmula a la forma

$$C_1 \wedge \cdots \wedge C_r$$

donde cada C_i es una disyunción de literales.

Distribuir cuantificadores universales Consiste en distribuir los cuantificadores sobre cada conjunción usando la fórmula válida $\forall x.(A \wedge B) \iff \forall x.A \wedge \forall x.B$ arrojando una conjunción de cláusulas

$$\forall x_1, \ldots, x_n C_1 \wedge \cdots \wedge \forall y_1, \ldots, y_m C_r$$

donde cada C_i es una disyunción de literales.
Por último, lo simplificamos escribiendo $\{C_1, \ldots, C_r\}$

Regla de resolución para lógica de primer orden Ahora que tenemos nuestra fórmula en correctamente Forma Clausal veamos como hacemos para aplicar resolución.

Cómo motivación, consideremos la fórmula $(\forall x.P(x)) \wedge \neg P(a)$, que claramente es insatisfacible. Sin embargo, no podemos aplicar la regla de resolución estandar, ya que $P(x)$ y $P(a)$ no son idénticos... pero si son unificables.

Definimos una nueva regla de resolución, como

$$\frac{\{B_1, \ldots, B_k, A_1, \ldots, A_m\}\ \{\neg D_1, \ldots, \neg D_j, C_1, \ldots, C_n\}}{\sigma(\{A_1, \ldots, A_m, C_1, \ldots, C_n\})}$$

donde σ es el MGU de $\{B_1, \ldots, B_k, D_1, \ldots, D_j\}$. Algunas observaciones:

- Asumimos que las cláusulas $\{B_1, \ldots, B_k, A_1, \ldots, A_m\}$ y $\{\neg D_1, \ldots, \neg D_j, C_1, \ldots, C_n\}$ no tienen variables en común; en caso contrario se renombran las variables.
- Notar que $\sigma(B_1) = \cdots = \sigma(B_k) = \sigma(D_1) = \cdots = \sigma(D_j)$
- La cláusula $\sigma(\{A_1, \ldots, A_m, C_1, \ldots, C_n\})$ se llama resolvente, igual que antes.

Análogamente al caso proposicional, tenemos las nociones de cláusula vacía, paso de resolución y refutación. Además, tenemos el siguiente resultado.

Teorema (Teorema de Herbrand-Skolem-Gödel)**.** *Cada paso de resolución en lógica de primer orden preserva satisfacibilidad.*

Regla de resolución binaria Incorpora una regla adicional que es útil en algunos casos: *factorización*

$$\frac{\{B_1, \ldots, B_k, A_1, \ldots, A_m\}}{\sigma(\{B_1, A_1, \ldots, A_m\})}$$

donde σ es el MGU de $\{B_1, \ldots, B_k\}$.

4.6 Resolución lineal

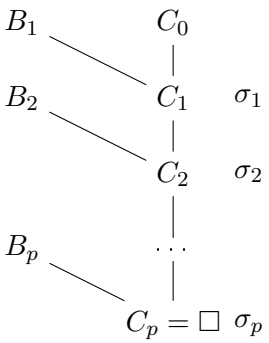
Si bien el método de resolución general es completo, hallar refutaciones es un proceso muy caro en el caso general ya que el espacio de búsqueda producido puede ser enorme. Además, hay un alto grado de no-determinismo

- ¿Qué cláusulas elegimos? Hace falta una Regla de búsqueda.

- ¿Qué literales eliminamos? Hace falta una Regla de selección.

Estas reglas ayudan a reducir el espacio de búsqueda, y es deseable que estas restricciones no renuncien a la completitud del método.

Una secuencia de pasos de resolución a partir de S es **lineal** si es de la forma:



donde C_0 y cada B_i es un elemento de S o algún C_j con $j < i$. Es decir, para cada nuevo paso de resolución tengo que usar el último resolvente que generé y algún elemento de S.

Observaciones de la resolución lineal:

- En general, reduce el espacio de búsqueda considerablemente.
- Preserva completitud.
- Sin embargo, sigue siendo altamente no-determinístico.

4.7 Cláusulas de Horn

Es un tipo de subclase de fórmulas para las cuales existen procesos eficientes para producir refutaciones sin perder completitud.

Una cláusula de Horn es una disyunción de literales que tiene *a lo sumo* un literal positivo. Para conjuntos de cláusulas de Horn puede la resolución SLD, que tiene buenas propiedades.

Dividimos estas cláusulas en 3 grupos:

- *Goal*: todos los literales negativos.
- *Definición*: tiene exactamente un literal positivo. Puede ser:
  - *Regla*: un literal positivo, y todos los demás negativos.
  - *Hecho*: sólo un literal positivo.

Tener en cuenta que no todas las fórmulas de primer orden pueden expresarse como una cláusula de Horn.

4.8 Resolución SLD

Es una variante de la resolución lineal.

Sea $S = P \cup \{G\}$ un conjunto de cláusulas de Horn (con nombres de variables disjuntos) tal que

- P es un conjunto de cláusulas de definición y
- G es una cláusulas *goal* o negativa.

Decimos que S son las cláusulas de entrada. P se conoce como el programa o base de conocimientos y G es el objetivo o meta.

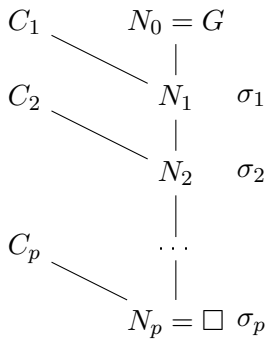
Luego, una secuencia de pasos de resolución SLD para S es una secuencia $\langle N_0, \ldots, N_p \rangle$ de cláusulas negativas que satisfacen:

- N_0 es el goal G.
- Para todo N_i en la secuencia, $0 < i < p$, si N_i es

$$\neg A_1, \ldots, \neg A_{k-1}, \neg A_k, \neg A_{k+1}, \ldots, \neg A_n$$

  entonces hay alguna cláusula de definición C_i de la forma $\{A, \neg B_1, \ldots, \neg B_m\}$ en P tal que A_k y A son unificables con MGU σ, y si
  - $m = 0$, entonces N_{i+1} es $\{\sigma(\neg A_1, \ldots, \neg A_{k-1}, \neg A_{k+1}, \ldots, \neg A_n)\}$
  - $m > 0$, entonces N_{i+1} es $\{\sigma(\neg A_1, \ldots, \neg A_{k-1}, \neg B_1, \ldots, \neg B_m, \neg A_{k+1}, \ldots, \neg A_n)\}$

En el caso de que $\langle N_0, \ldots, N_p \rangle$ sea una refutación, entonces $N_p = \square$. Y por ser lineal, tiene que tener la forma:



Observaciones de la resolución SLD:

- No se especifica regla de búsqueda ni de selección.
- La sustitución respuesta es $\sigma_p \circ \cdots \circ \sigma_2 \circ \sigma_1$
- **Corrección:** Si un conjunto de cláusulas de Horn tiene refutación SLD, entonces es insatisfacible.
- **Completitud:** Dado un conjunto de cláusulas de Horn $P \cup \{G\}$ tal como se describió, si $P \cup \{G\}$ es insatisfacible, existe una refutación SLD cuya primera cláusula es G.

4.9 Prolog

Es un lenguaje de programación lógico presentado en el año 1971. Los programas se escriben en un subconjunto de la lógica de primer orden, y el mecanismo teórico en el que se basa es el método de resolución.

En particular, utilizar resolución SLD con las siguientes restricciones:

- **Regla de búsqueda:** se seleccionan las cláusulas de programa de arriba hacia abajo, en el orden en que fueron introducidas.
- **Regla de selección:** seleccionar el átomo de más a la izquierda.

Estas dos reglas se definen como la *estrategia*. Cada estrategia determina un árbol de búsqueda o árbol SLD.

Dado que $(A_1 \wedge \cdots \wedge A_n) \supset B \iff \neg(A_1 \wedge \cdots \wedge A_n) \supset B \iff \neg A_1 \vee \cdots \vee \neg A_n \vee B$, tenemos que las cláusulas P en Prolog se escriben

- `B :- A_1, ..., A_n.` para $\{B, A_1, \ldots, \neg A_n\}$ (reglas)
- `B.` para $\{B\}$ (hechos)

Búsqueda de refutaciones SLD en Prolog

- Recorre el árbol SLD en profundidad (**DFS**), lo cual permite implementarlo de manera muy *eficiente*:
  - Se usa una pila para representar los átomos del goal.
  - Se hace un push del resolverte del átomo del tpo de la pila con la cláusula de definición.
  - Se hace un pop cuando el átomo del top de la pila no unifica con ninguna cláusula de definición más (luego, el átomo que queda en el tope se unifica con la siguiente cláusula de definición).
- **Desventaja:** puede no llegar a encontrar una refutación SLD aún si existe.

Operador Cut Es una anotación que permite poder el árbol SLD, y se encuentra presente por cuestiones de eficiencia. En Prolog se escribe el símbolo `!`.

- Cuando se selecciona un cut, tiene éxito inmediatamente.
- Si, debido a backtracking, se vuelve a este cut, su efecto es el de hacer fallar el goal que le dio origen. De esta forma, forzamos determinismo (a lo sumo una solución).
- En particular, todas las variables que estaban ligadas al momento de pasar por el cut, quedan ligadas por el resto de la ejecución del programa y no pueden cambiarse.

Operador Not (negación por falla) Antes veamos algunas definiciones:

- Decimos que un árbol SLD **falla finitamente** si es finito y no tiene ramas de éxito.
- Dado un programa P, definimos el **conjunto de falla finita** de P como $\{B \mid B$ átomo cerrado ('ground') y existe un árbol SLD que falla finitamente con B como raíz $\}$

Entonces, la **negación por falla** se define como:

$$\frac{B \text{ átomo cerrado} \quad B \text{ en conjunto de falla finita de } P}{\neg B}$$

El operador `not` nos permite utilizar la negación por falla y se define como:

```
not(G) :- call(G), !, fail.
not(G).
```

La clave está en combinar el operador `!` y `fail`. Una vez que el `Goal` tiene éxito, el cut nos prohíbe de volver atrás en el árbol SLD, y el `fail` hace que la solución actual falle. Como no hay más ramas que explorar, toda la consulta falla.

Notar que es distinto a la negación lógica. Si la cláusulas no es cerrada, `not(G)` tiene éxito si no existe *ninguna* valuación tal que `G` se satisface.

Además, observar que el `not(G)` nunca instancia variables de `G`:

- Si `G` tiene éxito, `fail` falla y descarta la sustitución.
- Caso contrario, `not(G)` tiene éxito inmediatamente (sin afectar `G`).

5 Paradigma Orientado a Objetos

5.1 Características generales

- Computación a través del intercambio de mensajes entre objetos.
- Los objetos se agrupan en clases, y estas se agrupan en jerarquías.
- Tiene un **alto nivel de abstracción**: los conceptos centrales son los de objetos, clases y mensajes.
- Tiene una **arquitectura extensible**: jerarquía de clases, polimorfismo de subtipos y binding dinámico.
- La matemática y razonamiento algebraico tiende a ser más compleja.
- La arquitectura de los programas se basan en **módulos** deducidos de objetos del dominio.

5.2 Fundamentos

- Programa: Modelo computable
- Modelo: Simulación
- Cómo: Envío de mensajes
- Objeto: Representación de un ente del dominio del problema
- Mensaje: Conjunto de colaboraciones entre objetos

El resultado es que los **objetos colaboran** entre sí enviándose **mensajes**.

Notar que el concepto de objeto no aparece al principio de la lista. El concepto principal es el de Modelo. Que todo sea un objeto es una consecuencia, no un objetivo.

5.3 Mensajes

Los mensajes definen al objeto, ya que nos dicen qué saben hacer los objetos. Mediante estos mensajes que saben responder, los objetos definen su comportamiento, su esencia.

Definimos la **esencia** de un objeto como el conjunto minimal de mensajes tales que, al sacar uno de ellos el objeto deja de representar al ente del dominio del problema.

En cuanto a la comunicación entre objetos:

- Dirigida: el receptor está presente y es alcanzable (visible) al momento del envío.
- Sincrónica: el envío de un mensaje es bloqueante.
- Siempre hay respuesta.
- Anónima: el receptor no conoce al emisor, la respuesta no varía en función del emisor.

Los mensajes tienen asociada una lista de colaboradores (parámetros/argumentos en el paradigma imperativo) que se envían con él.

5.4 Variantes del paradigma

5.4.1 Prototipado

En los lenguajes de prototipado orientados a objetos, todos los entes son de similar jerarquía, no hay abstracciones. Los objetos pueden tener padres (o prototipos) en otros objetos.

Permite trabajar con los objetos sin necesidad de tener clases. Con lo que podemos definir comportamiento por objeto (y crear objetos únicos).

Crear objetos equivale a clonar objetos. Permite modelar sin generalizar ni abstraer.

5.4.2 Clasificación

En los lenguajes de clasificación orientados a objetos, por cada entidad del problema a modelar existen una o más instancias de esa entidad, y la clase que las define. Esto exige pensar un nombre para dicha clase que represente correctamente lo que se está modelando.

Su esencias es saber crear instancias y definir el comportamiento de ellas.

En estos casos el tipado estático es una complicación, y el tipado dinámico un feature. Si el lenguaje es estáticamente tipado, es un problema pensar el protocolo a implementar por la clase de antemano. Además, el **Method LookUp** (búsqueda y llamado del código a ejecutar) se complica, mientras que con tipado dinámico se vuelve más flexible y permite varias optimizaciones.

Notar que como todo es un objeto, por lo tanto las clases también lo son. Pero los objetos solo existen cuando su clase existe. Para eso existe la clase `ProtoObject`, que no tiene clase padre, y la clase `MetaClass`, que es instancia de si misma.

Relación de subclasificación Permite decir que hay conceptos más abstractos que otros. Una subclase es algo más concreto que una super clase.

Este mecanismo se suele llamar *herencia*, y está pensado para construir teoría, no para ahorrar código.

5.5 Smalltalk

Self Es la forma en la que un objeto se refiere a si mismo. Es una pseudo-variable:

- Una palabra reservada.
- Siempre está y no se puede asignar.
- Representa al objeto que recibió el mensaje y está ejecutando el método.

Super Tiene exactamente las mismas propiedades que `self`. La única diferencia es que el *Method lookup* se realiza en la clase padre.

Tipos de mensajes Tenemos

- **Unarios**
- **Binarios**
- **Keyword**

Se evalúan de izquierda a derecha por pasadas: primero unarios, luego binarios y finalmente keywords.

Polimorfismo Dos objetos son polimórficos con respecto a un conjunto de mensajes si ambos responden a estos mensajes con la misma semántica.

Ejemplo: `1 + 2` son polimórficos con `1.0 + 2.0`, pero no con `'1' + '2'`.

Máximas Se necesita más delegación cuando:

- Tenemos código repetido.
  - Si es en objetos distintos, falta un objeto. **Forwarding**.
  - Si es en el mismo objeto, falta un método. **Delegación**.
- Hay código condicional. Faltan objetos. **Polimorfismo**.

Bloque, Closure y Full Closure Veamos las diferencias entre los 3:

- Un bloque es simplemente un pedazo de código (o función) cerrado (sin variables libres).
- Closure agrega *Binding Lexicográfico* de variables a los bloques. De esta forma, las variables libres del Closure se ligan con las variables que tenían el mismo nombre cuando fue creado.
- Full Closure agrega además un control apropiado de flujo. Es decir, si por ejemplo hacemos un return dentro de un full closure, es como si estuviéramos haciendo un return en el scope en el que fue creado.

Revisar, no está muy claro

5.6 Sistema de tipos

Algunos errores de tipos habituales en un lenguaje orientado a objetos suelen ser:

- Métodos que no reciben la cantidad y tipo de parámetros correctos.
- Asignaciones que no respetan el tipo declarado de las variables y campos.
- Invocación a métodos inexistentes.

Tener en cuenta que las nociones de clase y tipo se mantienen separadas:

- Clase: inherentemente de implementación (ej. variables de instancia privadas, código fuente de los métodos, etc.)
- Tipo de un objeto, la interfaz pública del mismo: nombres de todos los métodos, junto con el tipo de los argumentos y del resultado.

El tipo responde a la especificación de un objeto, mientras que la clase a su implementación. Es decir, varias clases pueden instanciar objetos con el mismo tipo.

5.7 Subtipado

Introducimos ahora la noción de **Juicio de subtitpado** $\sigma <: \tau$, que se entiende como "En todo contecxto donde se espera una expresión de tipo τ, puede utilizarse una de tipo σ en su lugar sin que ello genere un error". Ejemplo, si `D` es subclase de `C`, se espera que $DType <: CType$.

Ahora bien, si $\Gamma \rhd M : \sigma$ y $\sigma <: \tau$, siguiendo el principio de subtipado, podemos armar una nueva regla de tipado llamada **Subsumption**:

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

Ahora, podemos empezar a armar axiomas y reglas de subtipado. Para los tipos base asumimos que subtipan de la siguiente forma: $Bool <: Nat$, $Nat <: Int$, $Int <: Float$. Las reglas más básicas son:

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

El tipo Top Se puede ver como representando la clase `Object` en Samlltalk.

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

Notar que $Top \to Top <: Top$.

Subtipado de funciones Para las funciones, tenemos la siguiente regla de subtipado:

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

Observar que el sentido de $<:$ se da "vuelta" para el tipo del argumento, pero no para el tipo del resultado. Entonces, decimos que el constructor de tipos de función es **contravariante** en su primer argumento y **covariante** en el segundo.

Esto es así ya que si un contexto/programa P espera una expresión f de tipo $\sigma' \to \tau'$ puede recibir otra de tipo $\sigma \to \tau$ si se dan las condiciones indicadas:

- Toda aplicación de f está aplicada a argumentos de tipo σ'
- Los mismos se coercionan a argumentos del tipo σ. Es decir que σ' es más específico que σ. Que es lo mismo que decir que σ es más general que σ'. En un ejemplo concreto: si esperamos una función $Alumno \to Bool$, pueden pasarnos $Persona \to Bool$, pero no $AlumnoDC \to Bool$, porque podríamos usar la función con alumnos que no son del DC.
- Luego se aplica la función, cuyo tpo real es $\sigma \to \tau$.
- Finalmente se coerciona el resultado a τ', el tipo del resultado que P está esperando. Esto quiere decir que τ es más específico que τ'. Que es lo mismo que decir que τ' es más general que τ. En un ejemplo concreto: si esperamos una función $Nat \to Alumno$, pueden pasarnos $Nat \to AlumnoDC$, pero no $Nat \to Persona$, porque esa función podría devolver una persona que no es alumno.
- Luego se aplica la función, cuyo tpo real es $\sigma \to \tau$.

Subtipado de registros Recordemos las reglas de tipado para registros:

$$\frac{\Gamma \rhd M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \rhd \{I_i = M_i{}^{i \in 1..n}\} : \{I_i : \sigma_i{}^{i \in 1..n}\}} \text{ (T-Rcd)}$$

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

Cuando pensamos en registros, lo primero que queremos hacer es lograr que subtipen a lo "ancho". Es decir, que valga por ejemplo, $\{nombre : String, edad : Int\} <: \{nombre : String\}$, ya que en todos los contextos donde esperemos un registro con la etiqueta *nombre* no nos importa si el registro además tiene otras etiquetas. Con este objetivo, introducimos la siguiente regla de subtipado:

$$\frac{}{\{I_i : \sigma_i \mid I \in 1..n+k\} <: \{I_i : \sigma_i \mid I \in 1..n\}} \text{ (S-RcdWidth)}$$

Sin embargo, esta regla sola tiene una limitación. En los casos como $\{a : Alumno, b : Int\} <: \{a : Persona\}$, la regla anterior no funcionaría, ya que si bien es la misma etiqueta "a", el tipo es distinto. Pero nos gustaría que funcionara, ya que esperamos una Persona, y **un Alumno es una Persona** ($Alumno <: Persona$). Entonces, introducimos la regla de subtipado de registros en "profundidad":

$$\frac{\sigma_i <: \tau_i \quad i \in I = \{1..n\}}{\{I_i : \sigma_i\}_{i \in I} <: \{I_i : \tau_i\}_{i \in I}} \text{ (S-RcdDepth)}$$

Por último, si tenemos un caso como $\{b : Int, a : Alumno\} <: \{a : Persona\}$ las reglas anteriores no funcionan. Esto tiene sentido que funcione, ya que los registros son descriptores de campos, y no deberían depender del orden dado. Entonces, agregamos la siguiente regla:

$$\frac{\{k_j : \sigma_j \mid j \in 1..n\} \text{ es permutación de } \{I_i : \tau_i \mid i \in 1..n\}}{\{k_j : \sigma_j \mid j \in 1..n\} <: \{I_i : \tau_i \mid i \in 1..n\}} \text{ (S-RcdPerm)}$$

Notar que (S-RcdPerm) puede usarse en combinación con (S-RcdWidth) y (S-Trans) para eliminar campos en cualquier parte del registro.

Ahora, combinemos las 3 reglas en una sola (*one rule to rule them all*):

$$\frac{\{I_i \mid i \in 1..n\} \subset \{k_j \mid j \in 1..m\} \quad k_j = I_i \implies \sigma_j <: \tau_i}{\{k_j : \sigma_j \mid j \in 1..m\} <: \{I_i : \tau_i \mid i \in 1..n\}} \text{ (S-Rcd)}$$

Subtipado de referencias Lo interesante de las referencias es que no son contravariantes ni covariantes, son **invariantes**. Es decir: solo podemos comparar referencias de tipos equivalentes:

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

Ejemplo de que no es covariante: si $Bool <: Int$ entonces según la regla $Ref\ Bool <: Ref\ Int$ pero el término $let\ x\ =\ Ref\ True\ in\ x\ :=\ 3$ tipa y falla.

Ejemplo de que no es contravariante: si $Bool <: Int$ entonces según la regla $Ref\ Int <: Ref\ Bool$ pero el término $let\ x\ =\ Ref\ 3\ in\ (if\ !x\ then\ \ldots else \ldots)$ tipa y falla.

Algoritmo de subtipado Hasta ahora, las reglas de tipado **sin** subtipado eran dirigidas por sintaxis. Esto hace que sea inmediato implementar un algoritmo de chequeo de tipos a partir de ellas.

Sin embargo, al agregar **Subsumption**, ya no son dirigidas por sintaxis. Un análisis rápido determina que el único lugar donde se precisa subtipar es al aplicar una función a un argumento. Esto sugiere la siguiente modificación:

$$\frac{\Gamma \rhd M : \sigma \to \tau \quad \Gamma \rhd N : \rho \quad \rho <: \sigma}{\Gamma \rhd M\ N : \tau} \text{ (T-App)}$$

Pero igual hay que ver como se implementa en el algoritmo de subtipado.

Las reglas de subtipado no son dirigida por sintaxis, y el problema viene de las reglas `S-Refl` y `S-Trans`.

Observando que se puede probar $\sigma <: \sigma$ y transitividad, siempre que se tenga la reflexividad para los tipos escalares: $Nat <: Nat$, $Bool <: Bool$ y $Float <: Float$, agregamos esas 3 reglas y no consideramos explícitamente a `S-Refl` y `S-Trans`.

Luego, el algoritmo de chequeo de subtipos (obviando los axiomas de Nat, Bool, Float) nos queda:

```
subtype(S, T) =
  if T == Top:
    return true
```

```
elif S == S1->S2 and T == T1->T2:
  return subtype(T1, S1) and subtype(S2, T2)
elif S=={kj : Sj, j in 1..m} and T ={li : Ti, i in 1..n}:
  return {li, i in 1..n} subset {kj, j in 1..m} and
          ForAll i Exists j kj = li and subtype(Sj,Ti)
else:
  return false
```

A Finales tomados

Copio y pego los finales que están en cuba wiki. Trato de resolver los que tienen información suficiente.

A.1 07/03/2016 - Melgratti

1) Sea $fx = x(fx)$. Dar el resultado de evaluar $f(\backslash x \to 1 : x)$. Que representa esta función?

La función f representa la función fix: una extensión de cálculo lambda que permite construir funciones recursivas.

Entonces, $f(\backslash x \to 1 : x)$ construye una lista infinita de 1s.

2) Las reglas e-app1, e-app2, e-appAbs definen la regla de evaluación utilizada por haskell?

No. El mecanismo de reducción de haskell es *Lazy*. Las reglas e-app1, e-app2, e-appAbs establecen que la aplicación $M\ N$ recién se hace cuando N es un valor. Eso impediría que cosas como `let f x = 42 in f (1/0)` funcionen.

3) que pasa en el algoritmo de inferencia de if-then-else si eliminas la unificación de los contextos?

Lo mismo que pasaría en cualquier otra regla que tiene subexpresiones cuya inferencia da contextos distintos: puede pasar que una variable en una subexpresión se use como tipo σ y en otra subexpresión como tipo τ. Ejemplo: `if true then x+1 else x==false`

4) indicar por que no seria correcto una regla de subtipado de registros (era bastante fácil).

5) Dado un programa en prolog ver si con consultas ground vs el mismo programa con una modificación (cambiaba un hecho y ahora tenia un NOT) tenia soluciones distintas.

6) Verdadero y Falso de resolución

7) Un seguimiento de Smalltalk, bastante fácil. Jugar un poco con self y super.

A.2 26/03/2016 - Melgratti

1) sea $fab = (b : (-a))b$. Que representa $fixf1$?

2) uno que te daba unos tipos y tenias q dar las reglas semánticas

3) inferir el tipo de f g (f g) (o algo así)

4) Te daba un programa en Prolog y tenias que hacer el árbol. El programa tenia un NOT

5) Verdadero y Falso sobre resolución (en particular skolemizacion)

6) Un seguimiento de Smalltalk

B Práctica 8 - Ejercicios para el final

Ejercicio 1

1. Para armar el esquema de recursión estructural, primero tenemos que identificar los constructores:

- `Nil`: toma 0 parámetros.
- `Tern a (AT a) (AT a) (AT a)`: toma 4 parámetros (una raíz y los 3 árboles hijos).

Por lo que, si asumimos que nuestro tipo de salida es `b`, el esquema `foldAT` va a recibir 3 parámetros:

- Un `b`, que es el que devolvemos cuando el `AT a` es `Nil`.
- Y una función `f::a->b->b->b->b` que dada una raíz, y el resultado recursivo en los 3 árboles hijos, nos devuelve un `b`.
- El árbol `AT a` sobre el cual se va a hacer el *fold*.

Entonces, la función queda:

```
foldAT::b->(a->b->b->b->b)->AT a->b
foldAT base _ Nil = base
foldAT base f (Tern a izq cent der) = f a (recu izq) (recu cent) (
    recu der)
     where recu = foldAT base f
```

2. El esquema de recursión primitiva es similar al anterior, pero en el paso recursivo tenemos acceso a cual es el árbol que estamos procesando.

```
recAT::b->(AT a->b->b->b->b)->AT a->b
recAT base f Nil = base
recAT base f arbol@(Tern a izq cent der) = f arbol (recu izq) (recu
    cent) (recu der)
     where recu = recAT base f
```

3. Usando recursión explicita tenemos que:

```
esSubarbol::Eq a => AT a->AT a->Bool
esSubarbol Nil _ = True
esSubarbol uno otro@(Tern a izq cent der) = uno == otro || recu izq
    || recu cent || recu der
     where recu = esSubarbol uno
```

Notemos que en la definición necesitamos si o si el árbol "otro" ya que lo necesitamos para el caso en que el árbol "uno" no es es subArbol de izq, cent o der. Entonces, podemos reescribirla usando `recAT`:

```
esSubarbol::Eq a => AT a->AT a->Bool
esSubarbol Nil _ = True
esSubarbol uno = recAT True (\otro recuIzq recuCent recuDer->uno ==
    otro || recuIzq || recuCent || recuDer)
```

REVISAR

Ejercicio 2

```
funcionizar::Eq a => [a]->[b]->a->Maybe b
funcionizar _ [] _ = Nothing
funcionizar [] _ _ = Nothing
funcionizar (x:xs) (y:ys) p | p == x = y
                            | otherwise = funcionizar xs ys p
```

Ejercicio 3

```
inversaAcotada::Eq b => (a->b)->[a]->b->Maybe a
inversaAcotada f dom = funcionizar (map f dom) dom
```

Ejercicio 4

1. Igual que en el Ejercicio 1, identificamos los constructores:

- `Cte Float`: toma 1 parámetro.
- `Suma expr expr`: toma 2 parámetro.
- `Div expr expr`: toma 2 parámetro.

Si asumimos que nuestro tipo de salida es `b`, el esquema `foldExpr` va a recibir 4 parámetros:

- Una función `f::Float->b`, que se va a usar cuando la `expr` es constante.
- Una función `g::b->b->b`, que se va a usar cuando la `expr` es una suma.
- Una función `h::b->b->b`, que se va a usar cuando la `expr` es una división.
- La `expr` sobre la cual se va a hacer el *fold*.

Entonces, la función queda:

```
foldExpr::(Float->b)->(b->b->b)->(b->b->b)->expr->b
foldExpr f _ _ (Cte x) = f x
foldExpr f g h (Suma e1 e2) = g (recu e1) (recu e2)
    where recu = foldExpr f g h
foldExpr f g h (Div e1 e2) = h (recu e1) (recu e2)
    where recu = foldExpr f g h
```

2. Primero con recursión explicita

```
eval::expr->Maybe (Float,String)
eval (Cte x) = Just (x, "")
eval (Suma e1 e2) =
    case (eval e1) of
      Nothing->Nothing
      Just (res1, traza1)->
        case (eval e2) of
          Nothing->Nothing
          Just (res2, traza2)->Just (res1 + res2, traza1 ++ traza2)
eval (Div e1 e2) =
    case (eval e1) of
      Nothing->Nothing
      Just (res1, traza1)->
```

```
        case (eval e2) of
          Nothing->Nothing
          Just (res2, traza2)->if res2 != 0 then Just (res1 / res2,
              traza1 ++ traza2) else Nothing
```

3. Ahora usando `foldExpr`

```
eval::expr->Maybe (Float,String)
eval = foldExpr (\x->Just (x, ""))
                (\r1 r2->
                  case r1 of
                    Nothing->Nothing
                    Just (res1, traza1)->
                      case r2 of
                        Nothing->Nothing
                        Just (res2, traza2)->Just (res1 + res2,
                            traza1 ++ traza2))
                (\r1 r2->
                  case r1 of
                    Nothing->Nothing
                    Just (res1, traza1)->
                      case r2 of
                        Nothing->Nothing
                        Just (res2, traza2)->if res2 != 0 then Just (
                            res1 / res2, traza1 ++ traza2) else
                            Nothing)
```

4. Nos dan como pista el tipo:

```
data Evaluation = Ev (String->Maybe (Float,String))
```

Y el objetivo es poder definir `eval` como `eval expr = applyEv (evalM expr) ""`. Además, sabemos que:

```
applyEv :: Evaluation -> String -> Maybe (Float, String)
applyEv (Ev f) s = f s
```

Y que `evalM::expr->Evaluation`.

Usando la técnica de los recuadros para abstraer las partes comunes entre `Suma` y `Div`, tenemos las siguientes funciones comunes:

```
instance Monad Evaluation where
  -- return :: (String -> Maybe (Float, String) -> Evaluation
  return f = Ev f
  -- (>>=) :: Evaluation -> ((String -> Maybe (Float, String)) ->
      Evaluation) -> Evaluation
  (>>=) (Ev f) g = g f
```

Y las funciones específicas:

```
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
liftM2 f mx my = do x <- mx
  y <- my
  return (f x y)
```

```
(</>) :: Monad m => m Float -> m Float -> m Float
(</>) = mx my = do
  x <- mx
  y <- my
  if y == 0 then Nothing else return (x/y)

(<+>) :: Monad m => m Float -> m Float -> m Float
(<+>) = liftM2 (+)
```

Con lo que ya podemos escribir

```
evalM::expr->Evaluation
evalM (Cte x) = returnM (x, "")
evalM (Suma e1 e2) =
  evalM e1 >>= \v1 ->
  evalM e2 >>= \v2 ->
  return v1 <+> v2
evalM (Div e1 e2) =
  evalM e1 >>= \v1 ->
  evalM e2 >>= \v2 ->
  return v1 </> v2

evalM::expr->Evaluation
evalM (Cte x) = returnM (x, "")
evalM (Suma e1 e2) =
  evalM e1 >>= \(r1, traza1) ->
  evalM e2 >>= \(r2, traza2) ->
  returnM (r1 + r2, traza1 ++ traza2)
evalM (Div e1 e2) =
  evalM e1 >>= \(r1, traza1) ->
  evalM e2 >>= \(r2, traza2) ->
  if r2 == 0
    then failM
    else returnM (r1 / r2, traza1 ++ traza2)
```

Que, agregandole *syntatic sugar*, es lo mismo que:

```
evalM::expr->Evaluation
evalM (Cte x) = returnM (x, "")
evalM (Suma e1 e2) = do
  v1 <- evalM e1
  v2 <- evalM e2
  return v1 <+> v2
evalM (Div e1 e2) = do
  v1 <- evalM e1
  v2 <- evalM e2
  return v1 </> v2

evalM::expr->Evaluation
evalM (Cte x) = returnM (x, "")
evalM (Suma e1 e2) = do
  (r1, traza1) <- evalM e1
  (r2, traza2) <- evalM e2
  returnM (r1 + r2, traza1 ++ traza2)
```

```
evalM (Div e1 e2) = do
  (r1, traza1) <- evalM e1
  (r2, traza2) <- evalM e2
  if r2 == 0
    then failM
    else returnM (r1 / r2, traza1 ++ traza2)

evalM (Cte x) = returnM x
evalM (Div e1 e2) = do
  v1 <- evalM e1
  v2 <- evalM e2
  return v1 </> v2
```

Ejercicio 5

Sin usar `fix`, tenemos:

```
iterate::(a->a)->a->[a]
iterate sig base = base:(ite sig (sig base))
```

Por lo que con `fix`:

```
fix :: (a -> a) -> a
fix f = f (fix f)

iterate::(a->a)->a->[a]
iterate = fix (\rec sig base -> base:(rec sig (sig base)))
```

Ejercicio 6

Las expresiones que nos plantea el enunciado son:

$$M ::= \ldots \mid raise_\sigma\ M \mid try\ M_1, M_2, \ldots, M_n\ with\ N$$

Y nos dicen que los valores no se modifican.
Tampoco vamos a modificar los tipos, aunque si agregamos 2 reglas de tipado:

$$\frac{}{\Gamma \rhd raise_\sigma\ M : \sigma}\ \text{(T-Raise)}$$

$$\frac{\Gamma \rhd N : Nat \to \sigma_n \quad \Gamma \rhd M_i : \sigma_i, \text{ con } 1 \leq i \leq n}{\Gamma \rhd try\ M_1, M_2, \ldots, M_n\ with\ N : \sigma_n}\ \text{(T-Try)}$$

Y las reglas de semántica:

$$\frac{}{try\ V_1, \ldots, V_n\ with\ N \to V_n}\ \text{(E-NoError)}$$

$$\frac{M_i \to M_i'}{try\ V_1, \ldots, V_{i-1}, M_i, \ldots, M_n\ with\ N \to try\ V_1, \ldots, V_{i-1}, M_i', \ldots, M_n\ with\ N}\ \text{(E-UnPaso)}$$

$$\frac{}{try\ V_1, \ldots, V_{i-1}, raise_\sigma T, M_{i+1}, \ldots, M_n\ with\ N \to N\ T}\ \text{(E-Error)}$$

Ejercicio 7

Claramente queremos hacer algo como

$$pred \stackrel{\text{def}}{=} \lambda n : Nat.if\ iszero(n)\ then\ 0\ else\ search(\lambda x : Nat.succ(y) == x)$$

donde usamos el search para buscar el Nat anterior a n. Nos falta definir el predicado de igualdad entre Nat (==):

$$\frac{\Gamma \rhd M : Nat \quad \Gamma \rhd N : Nat}{\Gamma \rhd M == N : Bool}\ \text{(T-Igual)}$$

$$\frac{M \to M'}{M == N \to M' == N}\ \text{(E-Igual1)} \qquad \frac{N \to N'}{V == N \to V == N'}\ \text{(E-Igual2)}$$

$$\frac{}{0 == succ(V) \to false}\ \text{(E-IgualZeroSucc)} \qquad \frac{}{succ(V) == 0 \to false}\ \text{(E-IgualSuccZero)}$$

$$\frac{}{0 == 0 \to true}\ \text{(E-IgualZeroZero)} \qquad \frac{}{succ(V_1) == succ(V_2) \to V_1 == V_2}\ \text{(E-IgualSuccSucc)}$$

Ejercicio 8

La extensión básica de listas consiste en:

Agregar algunas partes del ejercicio 17, práctica 2

Para ayudarnos, vamos a escribir primero el map usando el fix, pero en Haskell:

```
fix :: (a -> a) -> a
fix f = f (fix f)

map :: (a->b) -> [a] -> [b]
map = fix (\rec f l -> case l of
                  [] -> []
                  (x:xs) -> (f x):(rec f xs))
```

Entonces, lo traducimos a cálculo lambda:

$$map_{\sigma,\tau} \stackrel{\text{def}}{=} fix\ (\lambda rec : \rho \to \rho.\lambda f : \sigma \to \tau.\lambda l : [\sigma].case\ l\ of\ \{[] \to [] \mid h :: t \to (f\ h) :: ((rec\ f)\ t)\})$$

Revisar el tipo de rec

Ejercicio 9

$$\mathbb{W}(let\ x = U\ in\ V) = \Gamma \rhd S(let\ x = M\ in\ N')$$

donde,

- $\mathbb{W}(U) = \Gamma_1 \rhd M : \tau$
- $\mathbb{W}(V\{x \leftarrow U\}) = \Gamma_2 \rhd N : \sigma$
- $S = MGU(\{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_1, x : \sigma_2 \in \Gamma_2\}$
- $N' = copiarAnotaciones(V, N)$
- $\Gamma = S\Gamma_1 \cup S\Gamma_2$

Notar que N' y el predicado *copiarAnotaciones* son necesarios porque el N inferido es el que tiene la x reemplazada por U. Si directamente usamos N en la respuesta no sólo estamos anotando los tipos, sino también modificando la expresión.

Falta usar el algoritmo nuevo con unos casos

Ejercicio 10

Usar el algoritmo del ejercicio 9 en algunos casos

Ejercicio 11

TODO

Ejercicio 12

Se agrega el término $map_{\sigma,\tau}$, con su regla de inferencia:

$$\mathbb{W}(map) = \emptyset \rhd map_{a,b} : (a \to b) \to [a] \to [b]$$

siendo a y b variables de tipo frescas.
Nos piden tipar la expresión `map map`. Dado que es una aplicación, tenemos que:

$$\mathbb{W}(map_1\ map_2) \overset{\text{def}}{=} \emptyset \rhd \ldots$$

donde

- $\mathbb{W}(map_1) = \emptyset \rhd map_{a,b} : (a \to b) \to [a] \to [b] = \tau$
- $\mathbb{W}(map_2) = \emptyset \rhd map_{c,d} : (c \to d) \to [c] \to [d] = \rho$
- t variable fresca
- Entonces, $S = MGU\{\tau \doteq \rho \to t\} \cup \{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_1, x : \sigma_2 \in \Gamma_2\}$ equivale a $S = MGU\{(a \to b) \to ([a] \to [b]) \doteq ((c \to d) \to ([c] \to [d])) \to t\}$

Hacemos paso a paso la unificación:

1. Descomposición: $\{(a \to b) \doteq (c \to d) \to ([c] \to [d]), [a] \to [b] \doteq t\}$.
2. Eliminación de variable: $\{(a \to b) \doteq (c \to d) \to ([c] \to [d])\}$ con $S_1 = \{t/([a] \to [b])\}$
3. Descomposición $\{a \doteq c \to d, b \doteq [c] \to [d]\}$
4. Eliminación de variables: $\emptyset$ con $S_2 = \{a/(c \to d), b/([c] \to [d])\}$.

Entonces, $S = S_2 \circ S_1$, y $St = [c \to d] \to [[c] \to [d]]$.
Y nos queda:

$$\mathbb{W}(map\ map) \overset{\text{def}}{=} \emptyset \rhd map_{c \to d,[c] \to [d]}\ map_{c,d} : [c \to d] \to [[c] \to [d]]$$

Ejercicio 13

a.

$$\mathbb{W}(U\ V) = \Gamma \rhd S(M\ N)$$

donde,

- $\mathbb{W}(U) = \Gamma_1 \rhd M : \tau$
- $\mathbb{W}(V) = \Gamma_2 \rhd N : \rho$
- t y s variables frescas, tales que $s <: \rho$
- $S = MGU(\{\tau \doteq s \to t\} \cup \{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_1, x : \sigma_2 \in \Gamma_2\}$
- $\Gamma = S\Gamma_1 \cup S\Gamma_2$

Revisar

b. aplicar el algoritmo del punto a

Ejercicio 14

1. Pasar $\forall x \forall y \exists z(P(x,z) \land P(y,z))$ a Forma normal de Skolem. Por pasos:

- Forma normal negada: idem.
- Forma prenexa: idem.
- Forma normal de Skolem: $\forall x \forall y(P(x, f(x,y)) \land P(y, f(x,y)))$

2. Pasar $\forall x \forall y((\exists z(P(x,z)) \land (\exists z P(y,z))))$ a Forma normal de Skolem. Por pasos:

- Forma normal negada: idem.
- Forma prenexa: $\forall x \forall y \exists w \exists v(P(x,w) \land P(y,v))$.
- Forma normal de Skolem: $\forall x \forall y(P(x, f(x,y)) \land P(y, g(x,y)))$

Ejercicio 15

a. ¿Cuál es la relación entre el árbol de resolución y el árbol de búsqueda de Prolog?

El árbol de resolución sólo contiene los resultados exitosos que se van encontrando en el árbol de búsqueda.

En este sentido, el árbol de resolución equivale a una refutación, mientras que el árbol de búsqueda utiliza las reglas de búsqueda y selección de prolog para ir explorando las soluciones mediante *backtracking*.

REVISAR

b. ¿Qué hay que verificar antes de llamar al algoritmo mgu para unificar dos cláusulas?

Hay que verificar que no compartan variables. Si este es el caso, hay que hacer un renombre de variables.

REVISAR

c. Calcular el resolvente entre las siguientes cláusulas: $\{P(x,y)\}, \{\neg P(y, f(y))\}$.

Primero renombramos a $\{P(x,w)\}, \{\neg P(y, f(y))\}$.

Luego, el resolvente es $\square$ usando $\{x \leftarrow y, w \leftarrow f(y)\}$.

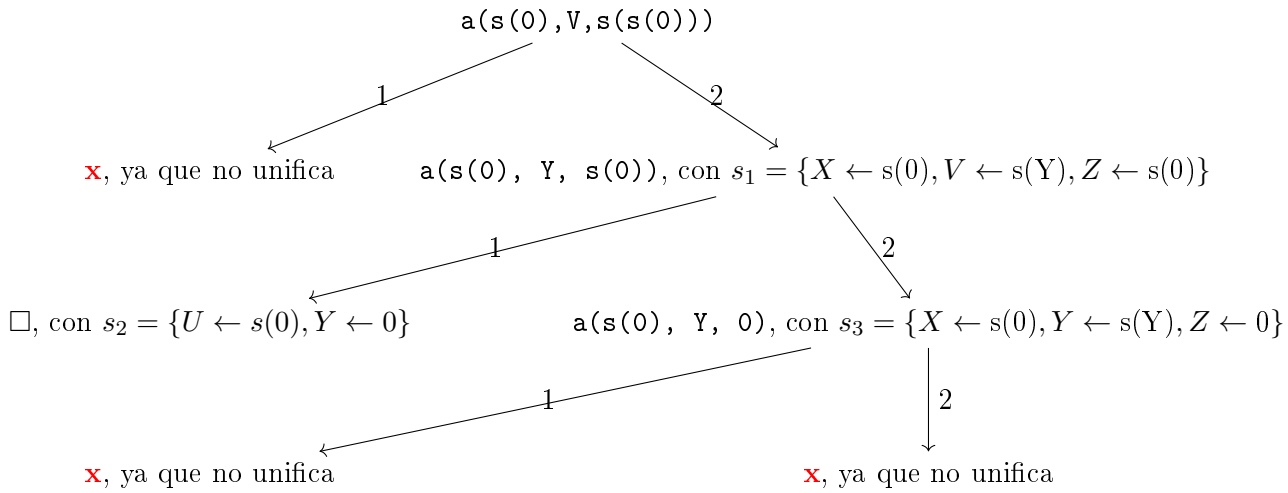
Ejercicio 16

Nuestra base de conocimientos:

1. `a(U,0,U)`
2. `a(X,s(Y),s(Z)):-a(X,Y,Z)`

Nuestro goal: `a(s(0),V,s(s(0)))`

a.



Luego, nos queda una sola sustitución resultado: $s = s_2 \circ s_1 = \{X \leftarrow s(0), V \leftarrow s(0), Z \leftarrow s(0), U \leftarrow s(0)\}$, con lo que nuestra única solución es $V = s(0)$.

b. La base de conocimientos pasadas a cláusulas:

1. $\{a(U, 0, U)\}$
2. $\{\neg a(X, Y, Z), a(X, s(Y), s(Z))\}$

El goal pasado a cláusula: $\{\neg a(s(0), V, s(s(0)))\}$. Entonces,

- Por 2, con $s_1 = \{X \leftarrow s(0), V \leftarrow s(Y), Z \leftarrow s(0)\}$, tenemos la cláusula $4 = \{\neg a(s(0), Y, s(0))\}$.
- Por 1, con $s_2 = \{U \leftarrow s(0), Y \leftarrow 0\}$, tenemos la cláusula $\square$.

Ejercicio 17

Seguimiento de `(c2 new) m3.`

receptor	mensaje	resultado
C2	new (de C2)	unC2
C2	new (de C1)	unC2
C2	new (de Object)	unC2
unC2	m3	23
unC2	m1 (de C1)	23
unC2	m2 (de C2)	23

Ejercicio 18

Ni idea cuales son los tipos Source y Sink

Ejercicio 19

Probar que si $M \twoheadrightarrow U$ y $M \twoheadrightarrow V$ con U una forma normal, entonces $V \twoheadrightarrow U$.

Proof. Recordemos que $\twoheadrightarrow$ es la clausura reflexiva, transitiva de $\rightarrow$. Es decir, la menor relación tal que:

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

Si $V = U$, es trivialmente cierto (por la segunda propiedad). Veamos el caso en que $V \neq U$.

Recordemos que cualquier término N sólo puede reducir a una forma normal: esto es así ya que el proceso de reducción en un paso es determinístico, y la definición de forma normal es una expresión que no se puede seguir reduciendo.

Luego, sabemos que $M \twoheadrightarrow V$. Como $V \neq U$, y M sólo puede reducir a una forma normal (U), entonces V no es forma normal.

Entonces, existe W forma normal tal que $V \twoheadrightarrow W$.

Ahora bien, como $M \twoheadrightarrow V$, y $V \twoheadrightarrow W$, por la primera propiedad, $M \twoheadrightarrow W$.

Pero W es forma normal, y $M \twoheadrightarrow W$, con lo que $W = U$, y $V \twoheadrightarrow U$, que era lo que queríamos ver. □

Ejercicio 20

Ni idea cuales son los tipos Source y Sink

Ejercicio 21

1. No vale en ninguno de los dos sentidos. Uno de es una referencía y el otro una tupla.

2. El único tipo que se me ocurre es Top, ya que vale $Ref\ Top <: Top$, aunque no vale $Top <: Ref\ Top$.

Ejercicio 22

TODO

Ejercicio 23

TODO

Ejercicio 24

1. Tenemos las siguientes fórmulas:

$$1.\forall X.b(X) \supset \forall Y.\neg a(Y,Y) \supset a(X,Y)$$
$$2.\forall X.b(X) \supset \forall Y.a(Y,Y) \supset \neg a(X,Y)$$

2. Queremos ver que $\neg\exists X.b(X)$, por lo que lo negamos y tratamos de demostrar que la negación es insatisfacible:

$$3.\exists X.b(X)$$

Quitamos implicaciones:

$$1.\forall X.\neg b(X) \vee \forall Y.a(Y,Y) \vee a(X,Y)$$
$$2.\forall X.\neg b(X) \vee \forall Y.\neg a(Y,Y) \vee \neg a(X,Y)$$
$$3.\exists X.b(X)$$

Pasamos a Forma normal negada:

$$1.\forall X.\neg b(X) \vee \forall Y.a(Y,Y) \vee a(X,Y)$$
$$2.\forall X.\neg b(X) \vee \forall Y.\neg a(Y,Y) \vee \neg a(X,Y)$$
$$3.\exists X.b(X)$$

Pasamos a Forma prenexa:

$$1.\forall X.\forall Y.\neg b(X) \vee a(Y,Y) \vee a(X,Y)$$
$$2.\forall X.\forall Y.\neg b(X) \vee \neg a(Y,Y) \vee \neg a(X,Y)$$
$$3.\exists X.b(X)$$

Pasamos a Forma de Skolem:

$$1.\forall X.\forall Y.\neg b(X) \vee a(Y,Y) \vee a(X,Y)$$
$$2.\forall X.\forall Y.\neg b(X) \vee \neg a(Y,Y) \vee \neg a(X,Y)$$
$$3.b(c)$$

Y ya están en FNC porque sólo hay $\vee$.
En Forma clausal:

$$1.\{\neg b(X), a(Y,Y), a(X,Y)\}$$
$$2.\{\neg b(X), \neg a(Y,Y), \neg a(X,Y)\}$$
$$3.\{b(c)\}$$

Hacemos la resolución:

- 4. $\{a(Y,Y), a(c,Y)\}$, usando 3 y 1, con $s_1 = \{X \leftarrow c\}$
- 5. $\{\neg a(Y,Y), \neg a(c,Y)\}$, usando 2 y 1, con $s_2 = \{X \leftarrow c\}$
- 6. Acá vamos a usar resolución binaria para mostrar que 4 y 5 son insatisfacibles.
  $\{a(c,c)\}$ factorizando 4, con $s_3 = \{Y \leftarrow c\}$
- 7. $\{\neg a(c,c)\}$ factorizando 5, con $s_4 = \{Y \leftarrow c\}$
- 8. $\square$, usando 6 y 7.

Ejercicio 25

a. Analizemos que pasa con cada predicado cuando se instancian o no las variables. Para `inorder1`:

- Si tanto el árbol como la lista están instanciadas: verifica si la lista corresponde al recorrido inorder del árbol.
- Si sólo el árbol está instanciado: calcula su recorrido inorder, y funciona ya que cuando llegamos al `append`, `LI` y `LR` ya están instanciadas.
- Si ninguno está instanciado: genera todos los pares (Árbol, Recorrido inorder), pero expande siempre la rama derecha, debido a que el backtracking nunca llega a `inorder1(I,LI)`.
- Si sólo la lista está instanciada: puede llegar a generar algún el árbol que tiene esa lista como su recorrido inorder, pero después se cuelga buscando más soluciones. De vuelta, el backtracking siempre lo hace en `inorder1(D,LD)`.

Para `inorder2`:

- Si tanto el árbol como la lista están instanciadas: verifica si la lista corresponde al recorrido inorder del árbol.
- Si sólo el árbol está instanciado: calcula su recorrido inorder, y se cuelga buscando más soluciones.
- Si ninguno está instanciado: idem `inorder1`.
- Si sólo la lista está instanciada: genera cada árbol que tiene esa lista como su recorrido inorder.

b. fiaca

Ejercicio 26

Seguimiento del código:

```
|cuentaA cuentaB|
cuentaA := CuentaBancaria new.
cuentaB := (CuentaVip new: 50) depositar: 40.
cuentaB transferir: 70 a: cuentaA.
```

Separados por lineas de código con la barra horizontal:

receptor	mensaje	resultado
CuentaBancaria	new (de CuentaBancaria)	unaCuenta
CuentaBancaria	new (de Object)	unaCuenta
unaCuenta	inicializar	unaCuenta
CuentaVip	new (de CuentaVip)	unaCuentaVip
CuentaVip	new (de CuentaBancaria)	unaCuentaVip
CuentaVip	new (de Object)	unaCuentaVip
unaCuentaVip	inicializar	unaCuentaVip
unaCuentaVip	fijarTope	unaCuentaVip
unaCuentaVip	depositar (de CuentaVip)	unaCuentaVip
unaCuentaVip	depositar (de CuentaBancaria)	unaCuentaVip
balance (0)	+	40
cuentaB	transferir: a: (de CuentaVip)	cuentaB
cuentaB	transferir: a: (de CuentaBancaria)	cuentaB
cuentaB	puedeExtraer (de CuentaVip)	True
balance (40)	+	90
90	≥	True
True	ifTrue:	_
cuentaB	extraer	cuentaB
cuentaB	puedeExtraer (de CuentaVip)	True
balance (40)	+	90
90	≥	True
True	ifTrue:	_
balance (40)	-	-30
cuentaA	depositar	cuentaA
balance (0)	+	70

Ejercicio 27

1. Probar que si $\Gamma \rhd M : \sigma$ es derivable y $\Gamma \cap \Gamma'$ contiene a todas las variables libres de M, entonces $\Gamma' \rhd M : \sigma$.

Proof. Como $\Gamma \rhd M : \sigma$ es derivable, significa que M es bien formado y tiene tipo σ. Además, $\Gamma \rhd M : \sigma$ si las variables libres de M tienen el tipo correcto en Γ.

Luego, como sabemos que $\Gamma \cap \Gamma'$ contiene a todas las variables libres de M, significa que las mismas se encuentran también en Γ' con el mismo tipo. Entonces, $\Gamma' \rhd M : \sigma$. □

2. (Weakening) Si $\Gamma \rhd M : \sigma$ es derivable y $x \notin dom(\Gamma)$, entonces $\Gamma \cup \{x : \tau\} \rhd M : \sigma$ es derivable.

Proof. Igual que antes, como $\Gamma \rhd M : \sigma$, las variables libres de M tienen el tipo correcto en Γ. Y sabemos que como $x \notin dom(\Gamma)$, entonces x no es una variable libre de M, por lo que no influencia en su tipado. Luego, $\Gamma \cup \{x : \tau\} \rhd M : \sigma$. □

3. (Strengthening) Si $\Gamma \cup \Gamma' \rhd M : \sigma$ es derivable y $FV(M) \subseteq Dom(\Gamma)$, entonces $\Gamma \rhd M : \sigma$ es derivable. Al unir contextos siempre se asume que tienen dominios disjuntos.

Proof. Con la última aclaración del enunciado es medio trivial. Si los dominios de Γ y Γ' son disjuntos ($\Gamma \cap \Gamma' = \emptyset$), entonces cada variable libre de M está en Γ, o bien está en Γ'.

Nos dicen que $FV(M) \subseteq Dom(\Gamma)$, por lo que sabemos que todas las variables libres de M están en Γ, y por lo tanto no están en Γ'.

Entonces, nos alcanza con usar Γ para tipar M ($\Gamma \rhd M : \sigma$). □

Ejercicio 28

Probar la propiedad de Unicidad: si $\Gamma \rhd M : \sigma$ y $\Gamma \rhd M : \tau$ son derivables, entonces $\sigma = \tau$.

Probar usando inducción estructural como pide el enunciado

Ejercicio 29

Probar el Lema se sustitución para subtipado: si $\Gamma \cup \{x : \sigma\} \rhd M : \tau$ y $\Gamma \rhd N : \sigma$ entonces $\Gamma \rhd M\{x \leftarrow N\} : \tau$

TODO

Ejercicio 30

TODO