Capítulo 1

Expresiones aritméticas tipadas

Hasta el momento hemos diseñado un lenguaje de programación sencillo no tipado, hemos descrito su sintaxis y las posibles estrategias de evaluación. Ahora vamos a añadir un sistema de tipos estático a nuestro programa. Recordemos que la sintaxis de nuestro programa era la siguiente:

```
t::=
     true
     false
     if t then t else t
     0
     succ t
     pred t
     iszero t
```

Al evaluar un término, el resultado podía ser un valor o bien el término podía quedarse "atascado" (como sería el caso de `pred true`). Los posibles valores de nuestro lenguaje de programación eran los siguientes:

```
v::=
     true
     false
     nv

nv::=
     0
     succ nv
```

Los términos atascados representan programas erróneos. Si somos capaces de distinguir aquellos términos cuyo resultado será un valor numérico de aquellos cuyo resultado será un valor booleano, podremos predecir que la evaluación de un término no se quedará atascada, sin necesidad de llegar a evaluarlo. Para ello introducimos dos tipos en nuestro sistema, `Nat` y `Bool`. En general, utilizaremos las meta-variables `S` , `T`, `U`, etc. para referirnos a tipos.

En nuestro lenguaje tipado, el término `if true then false else true` tiene tipo `Bool`, mientras que `pred (succ (pred (succ 0)))` tiene tipo `Nat`. Nuestro

análisis de tipos será *conservador* debido a su naturaleza estática, es decir, no podremos tipar términos como `if (iszero 0) then 0 else false` o `if true then 0 else false` a pesar de que su evaluación no se queda atascada en ningún momento.

1.1. La relación de tipado

La relación de tipado `t:T` se define por medio de un conjunto de reglas de inferencia que asignan tipos a los términos. Comencemos con las reglas de tipado de los booleanos.

Nuevas formas sintácticas:

```
T::=
     Bool
```

Reglas de tipado:

$$\texttt{true} : \texttt{Bool} \qquad (\texttt{T} - \texttt{True})$$

$$\texttt{false} : \texttt{Bool} \qquad (\texttt{T} - \texttt{False})$$

$$\frac{\texttt{t}_1\texttt{:Bool t}_2\texttt{:T t}_3\texttt{:T}}{\texttt{if t}_1\texttt{ then t}_2\texttt{ else t}_3\texttt{:T}} \qquad (\texttt{T} - \texttt{If})$$

En la regla `T-If` la guarda debe evaluarse a `Bool` y ambas ramas del condicional deben evaluarse al mismo tipo `T`.

Las nuevas formas sintácticas de la relación de tipado para los números son las siguientes:

```
T::=
     ...
     Nat
```

Y las nuevas reglas de tipado:

$$\texttt{0} : \texttt{Nat} \qquad (\texttt{T} - \texttt{Zero})$$

$$\frac{\texttt{t}_1\texttt{:Nat}}{\texttt{succ t}_1\texttt{:Nat}} \qquad (\texttt{T} - \texttt{Succ})$$

$$\frac{\texttt{t}_1\texttt{:Nat}}{\texttt{pred t}_1\texttt{:Nat}} \qquad (\texttt{T} - \texttt{Pred})$$

$$\frac{\texttt{t}_1\texttt{:Nat}}{\texttt{iszero t}_1\texttt{:Bool}} \qquad (\texttt{T} - \texttt{IsZero})$$

Definición: Se dice que un término `t` es *tipable* o está *bien tipado* si existe cierto tipo `T` tal que `t: T`.

El siguiente lema refleja algunas afirmaciones que pueden resultar de utilidad a la hora de razonar sobre la relación de tipado.

Lema *Inversión (o generación) de la relación de tipado*:

1. Si `true: R` entonces `R = Bool`.
2. Si `false: R` entonces `R = Bool`.

3. Si `if t1 then t2 else t3: R`, entonces `t1: Bool t2: R` y `t3: R`.
4. Si `0: R` entonces `R = Nat`.
5. Si `succ t1: R` entonces `R = Nat` y `t1: Nat`.
6. Si `pred t1: R` entonces `R = Nat` y `t1: Nat`.
7. Si `iszero t1: R` entonces `R = Bool` y `t1: Nat`.

De la misma manera que existía el árbol de derivación para la evaluación de un término, también existe el árbol de derivación para una relación de tipado. Por ejemplo, el árbol de derivación del término `if iszero 0 then 0 else pred 0: Nat` sería el siguiente:

$$\dfrac{\dfrac{}{\texttt{0:Nat}}\,\texttt{T-Zero}}{\dfrac{}{\texttt{iszero 0: Bool}}\,\texttt{T-IsZero}} \quad \dfrac{}{\texttt{0: Nat}}\,\texttt{T-Zero} \quad \dfrac{\dfrac{}{\texttt{0:Nat}}\,\texttt{T-Zero}}{\dfrac{}{\texttt{pred 0: Nat}}\,\texttt{T-Pred}}$$

$$\dfrac{\texttt{iszero 0: Bool} \qquad \texttt{0: Nat} \qquad \texttt{pred 0: Nat}}{\texttt{if iszero 0 then 0 else pred 0 : Nat}}\,\texttt{T-If}$$

Teorema *Unicidad del tipado*: En nuestro lenguaje de programación, cada término `t` tiene como máximo un tipo. Es decir, si `t` es tipable, entonces su tipo es único.

1.2. Seguridad = Progreso + Preservación

En el caso particular de nuestro lenguaje, diremos que se trata de un lenguaje seguro en el caso de que ningún término bien tipado pueda quedarse atascado. Para demostrar la seguridad de nuestro lenguaje utilizaremos los dos teoremas siguientes:

- *Progreso:* un término bien tipado no está atascado, es decir, o es un valor o se le puede aplicar alguna regla de evaluación.
- *Preservación:* si un término bien tipado se evalúa por paso corto, entonces el término resultante también está bien tipado.

Para probar el teorema del progreso, utilizaremos el siguiente lema:

Lema *Formas canónicas*:

1. Si `v` es un valor de tipo `Bool`, entonces `v` es `true` o `false`.
2. Si `v` es un valor de tipo `Nat`, entonces v es un número (`0` o `succ nv`).

Teorema *Progreso:* Si `t` es un término bien tipado`t: T`, entonces `t` es un valor o existe cierto `t'` tal que $\texttt{t} \rightarrow \texttt{t'}$.

Prueba: Por inducción en la derivación de `t: T`. Asumimos que la propiedad se cumple para todas las subderivaciones y analizamos la regla utilizada en la raíz del árbol de derivación.

1. Los casos `T-True`, `T-False` y `T-Zero` son inmediatos, pues en dichos casos `t` es un valor.

2. Caso `T-If`:

   $$\texttt{t = if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \qquad t_1 \texttt{ : Bool} \qquad t_2\texttt{: T} \qquad t_3\texttt{: T}$$

   Por la hipótesis de inducción, existen tres posibilidades:

   a) t_1 `= True`, en cuyo caso podemos aplicar la regla de derivación `E-IfTrue`.
   b) t_1 `= False`, en cuyo caso podemos aplicar la regla de derivación `E-IfFalse`.
   c) t_1 no es una constante, en cuyo caso existe un t_1`'` tal que $t_1 \rightarrow t_1$`'` y podemos aplicar la regla `E-If` de forma que `t` $\rightarrow$ `if` t_1`'` `then` t_2 `else` t_3.

3. Caso `T-Succ`:

   $$\texttt{t = succ } t_1 \qquad t_1\texttt{: Nat}$$

   Por la hipótesis de inducción existen dos posibilidades:

   a) t_1 `= nv`, en cuyo caso `t` también es un valor numérico.
   b) t_1 no es una constante, en cuyo caso existe un t_1`'` tal que $t_1 \rightarrow t_1$`'` y podemos aplicar la regla `E-Succ` de forma que `t` $\rightarrow$ `succ` t_1`'`.

4. Caso `T-Pred`:

   $$\texttt{t = pred } t_1 \qquad t_1\texttt{: Nat}$$

   Esta vez, por la hipótesis de inducción, existen 3 posibilidades:

   a) t_1 `= 0`, en cuyo caso podemos aplicar la regla `E-PredZero`.
   b) t_1 `= succ nv`, en cuyo caso podemos aplicar la regla `E-PredSucc`.
   c) t_1 no es una constante, en cuyo caso existe un t_1`'` tal que $t_1 \rightarrow t_1$`'` y podemos aplicar la regla `E-Pred` de forma que `t` $\rightarrow$ `pred` t_1`'`.

5. Caso `T-IsZero`:

   $$\texttt{t = iszero } t_1 \qquad t_1\texttt{: Nat}$$

   Similar al caso anterior.

Teorema *Preservación:* Si `t: T` y `t` $\rightarrow$ `t'`, entonces `t': T`.

Prueba: de nuevo por inducción en la derivación `t: T`.

1. Caso `T-True`: `t = true` y `T = Bool`.
   En este caso, la única regla del árbol de derivación es `T-True`, `t` es un valor, y la propiedad se cumple por defecto.
   Este mismo razonamiento se puede aplicar en los casos `T-False` y `t-Zero`.
2. Caso `T-If`: `t = if` t_1 `then` t_2 `else` t_3 t_1`:Bool` t_2`:T` t_3`:T`
   Por ser t_1`:Bool`, existen tres posibilidades:

   a) `t`$_1$ `= True`, en cuyo caso podemos aplicar la regla de derivación `E-IfTrue`, de lo cual se deduce que `t` $\rightarrow$ `t`$_2$ y dado que `t`$_2$ `= T` la propiedad se cumple.
   b) `t`$_1$ `= False`, igual que caso anterior.
   c) `t`$_1$ no es una constante, en cuyo caso existe un `t`$_1$`'` tal que `t`$_1$ $\rightarrow$ `t`$_1$`'` y podemos aplicar la regla `E-If` de forma que `t' = if t`$_1$`' then t`$_2$ `else t`$_3$ y, por la hipótesis de inducción, `t`$_1$`': Bool` y `t': T`.

3. Caso `T-Succ`: `t = succ t`$_1$ `T = Nat` `t`$_1$`: Nat`.
   La única regla de derivación aplicable en este caso es `E-Succ`, que nos dice que existe un `t`$_1$`'` tal que `t`$_1$ $\rightarrow$ `t`$_1$`'`. Dado que `t`$_1$`: Nat`, por la hipótesis de inducción podemos deducir que `t`$_1$`': Nat`, con lo cual `succ t`$_1$`: Nat`.

Capítulo 2

Lambda-cálculo tipado

En el tema anterior diseñamos un lenguaje de programación con tipado estático para expresiones aritméticas. Dicho sistema constaba de dos tipos `Bool` y `Nat`. Los términos atascados (como `if 0 then 1 else 2`) no pertenecen a ninguno de los dos tipos anteriores.

En este tema, construiremos un sistema de tipos para un lambda-cálculo puro enriquecido únicamente con booleanos (por simplicidad, vamos a ignorar los números). Es decir, enumeraremos un conjunto de reglas de tipado para variables, abstracciones y aplicaciones que:

1. Mantenga la seguridad del tipado, es decir, cumpla los teoremas de preservación y progreso.

2. No sea demasiado conservador.

En nuestro sistema, al que denominaremos $\lambda_{\rightarrow}$, no será posible determinar el tipo términos como (`if <calculo complicado> then true else (`λ`x.x)`) sin antes evaluar el cálculo complicado para comprobar si su resultado es `true` o `false`.

Dado que en el lambda-cálculo puro las funciones son un tipo en sí mismo, el sistema de tipos del lambda-cálculo enriquecido con booleanos debe incluir el tipo función. A la hora de definir un tipo función debemos tener en cuenta el tipo de sus parámetros y de su resultado. Por ello, el conjunto de tipos de nuestro sistema será el siguiente:

```
T::=
     Bool
     T → T
```

El *constructor de tipos* $\rightarrow$ es asociativo por la derecha, es decir $\mathtt{T}_1 \rightarrow \mathtt{T}_2 \rightarrow \mathtt{T}_3$ equivale a $\mathtt{T}_1 \rightarrow (\mathtt{T}_2 \rightarrow \mathtt{T}_3)$.

2.1. La relación de tipado

A la hora de tipar una abstracción λ`x.t`, lo primero que debemos preguntarnos es qué tipo tiene el argumento de dicha abstracción. Para responder a esta pregunta existen dos alternativas: anotar la abstracción con el tipo de su argumento o analizar

el cuerpo de la abstracción para intentar deducir el tipo en base a las operaciones realizadas con el argumento. Por ahora, nosotros vamos a optar por la primera alternativa, de forma que la abstracción se escribirá $\lambda \mathtt{x:T}_1\,.\,\mathtt{t}_2$.

En general, los lenguajes de programación en los que los términos se anotan con su correspondiente tipo se denominan *explícitamente tipado*, mientras que los que lo infieren a partir del comportamiento de los términos se denominan *implícitamente tipados*.

La sintaxis de nuestro lenguaje se define como:

```
t::=
      x
      λx:T.t
      t t

v::=
     λx:T.t

T::=
      T → T

Γ ::=
     ∅
     Γ, x:T
```

Sus reglas de evaluación son idénticas al lambda-cálculo puro excepto por las anotaciones de tipos en las abstracciones:

$$\frac{\mathtt{t}_1 \rightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \rightarrow \mathtt{t}_1'\ \mathtt{t}_2} \qquad \mathtt{E-App1}$$

$$\frac{\mathtt{t}_2 \rightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \rightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \mathtt{E-App2}$$

$$(\lambda \mathtt{x} : \mathtt{T}_1.\mathtt{t}_{12})\ \mathtt{v}_2 \rightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \mathtt{E-AppAbs}$$

Por último, las reglas de tipado son las siguientes:

$$\frac{\Gamma,\mathtt{x:T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x:T}_1.\mathtt{t}_2 : \mathtt{T}_1 \rightarrow \mathtt{T}_2} \qquad \mathtt{T-Abs}$$

$$\frac{\mathtt{x:T} \in \Gamma}{\Gamma \vdash \mathtt{x:T}} \qquad \mathtt{T-Var}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \rightarrow \mathtt{T}_{12} \quad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\mathtt{t}_2 : \mathtt{T}_{12}} \qquad \mathtt{T-App}$$

La regla `T-Abs` indica que, una vez conocido el tipo de los argumentos, el tipo del resultado de la función vendrá determinado por el tipo de su cuerpo $\mathtt{t}_2$ una vez asumido que en él todas las apariciones de las variables tienen tipo adecuado. En dicha fórmula, Γ es un conjunto de asunciones acerca del tipo de las variables libres que pueden aparecer en el término analizado.

Γ se denomina *contexto de tipado* y se trata de una secuencia de variables con su tipo separadas por coma. Al añadir una nueva variable `x` a un determinado Γ (Γ, `x:T`), debemos asegurarnos de que no exista otra variable con el mismo nombre en el contexto, en cuyo caso tendremos que renombrar la variable actual para evitar

posibles conflictos. En general, el dominio de un contexto $dom(\Gamma)$ es el conjunto de variables ligadas en dicho contexto.

La regla de tipado para variables `T-Var` es muy sencilla, pues una variable tiene el tipo indicado en el contexto Γ.

En general, las reglas de tipado de las constantes booleanas y los condicionales son las mismas que en el tema anterior:

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \; T - \texttt{If}$$

Una vez definidas las reglas de tipado, podemos establecer que el tipo de expresiones como:

```
if true then (λx:Bool.x) else (λx:Bool.not x): Bool → Bool
```

A través de las reglas de tipado podemos construir árboles de derivación, como hicimos con los lenguajes definidos en los temas anteriores. Por ejemplo, el árbol de derivación que demuestra que el término `(λx:Bool.x) true` tiene tipo `Bool` dado un contexto de tipado vacío.

$$\dfrac{\dfrac{\dfrac{x{:}\texttt{Bool} \in x{:}\texttt{ Bool}}{x{:}\texttt{Bool} \vdash x{:}\texttt{Bool}}\,\texttt{T-Var}}{\vdash \lambda x{:}\texttt{Bool}.x : \texttt{Bool} \to \texttt{Bool}}\,\texttt{T-Abs} \qquad \dfrac{}{\vdash \texttt{true: Bool}}\,\texttt{T-True}}{\vdash (\lambda x{:}\texttt{Bool}.x)\ \texttt{true} : \texttt{Bool}}\,\texttt{T-App}$$

Ejercicio: Dibujar el árbol de derivación del término

```
f: Bool → Bool ⊢ f (if false then true else false) : Bool
```

2.2. Propiedades del tipado

Para demostrar la seguridad de nuestro sistema de tipado debemos demostrar que se cumplen los teoremas de progreso y preservación. Ambos teoremas se prueban por inducción y precisan de la enumeración de una serie de lemas y teoremas que constituyen las propiedades de la relación de tipado.

Teorema *Unicidad del tipado*: en un determinado entorno de tipado Γ, un término `t` cuyas variables libres pertenezcan al dominio de Γ tendrá como mucho un tipo. Es decir, si el término es tipable, entonces su tipo es único.

Lema *Formas canónicas*: Si `v` es un valor de tipo `Bool`, entonces `v` es `true` o `false`.

Teorema *Progreso*: Si `t` es un término cerrado y bien tipado (⊢ `t: T`), entonces `t` es un valor o existe un término `t'` tal que `t` → `t'`.

Prueba: por inducción en la derivación del tipado.

Lema *Permutaciones*: Si $\Gamma \vdash$`t: T` y $\triangle$ es una permutación de Γ, entonces $\triangle \vdash$`t: T`

Lema *Debilitamiento*: Si $\Gamma \vdash$`t: T` y `x` $\notin dom(\Gamma)$, entonces Γ,`x:S`$\vdash$`t: T`

Lema *Preservación de los tipos bajo sustitución*: Si Γ, `x:S`$\vdash$`t: T` y $\Gamma \vdash$`s:S`, entonces $\Gamma \vdash$`[x` $\mapsto$ `s]t:T`.

Prueba: por inducción en la derivación de Γ, `x:S`$\vdash$`t: T`. Asumimos que la propiedad se cumple en las subderivaciones y analizamos la última regla de tipado aplicada.

1. Caso `T-Var`:

   `t = z`
   `z:T` $\in$ (Γ, `x:S`)

   Dos subcasos a considerar:

   a) `z = x`, en cuyo caso `[x` $\mapsto$ `s]z = s`. Por lo tanto, para que la propiedad se cumpla, ha de cumplirse que $\Gamma \vdash$`s:S`, que es una de las suposiciones del lema.

   b) `z` $\neq$ `x`, en cuyo caso `[x` $\mapsto$ `s]z = z`, y el resultado es inmediato.

2. Caso `T-Abs`:

   `t` = λ`y:`T_2`.`t_1
   `T` = T_1 $\rightarrow$ T_2
   Γ`,x:S,y:`T_2 $\vdash$ t_1`:`T_1

   Vamos a asumir que `x` $\neq$ `y` y que `y`$\notin FV($`s`$)$. Utilizando el lema de la permutación, obtenemos Γ`,y:`T_2`,x:S` $\vdash$ t_1`:`T_1. Utilizando ahora el lema del debilitamiento en la derivación ($\Gamma \vdash$`s:S`) obtenemos (Γ,`y:`$T_2 \vdash$`s:S`). Ahora, por la hipótesis de inducción tenemos que Γ,`y:`$T_2 \vdash$`[x` $\mapsto$ `s]`t_1`:`T_1. Aplicando la regla de tipado `T-Abs` tenemos que Γ,`y:`$T_2 \vdash$ λ`y:`T_2`.[x` $\mapsto$ `s]` t_1`:`T_2 $\rightarrow$ T_1, lo cual es el resultado buscado dado que por la definición de sustitución `[x` $\mapsto$ `s]t` = λ`y.`T_2`. [x` $\mapsto$ `s]`t_1.

3. Caso `T-App`:

   `t` = t_1 t_2
   Γ`, x:S` $\vdash$ t_1`:`T_2 $\rightarrow$ T_1
   Γ`, x:S` $\vdash$ t_2`:`T_2
   `T` = T_1

   Por la hipótesis de inducción se cumple $\Gamma \vdash$ `[x` $\mapsto$ `s]`t_1`:`T_2 $\rightarrow T_1$ y $\Gamma \vdash$ `[x` $\mapsto$ `s]`t_2`:`T_2. Por la regla de tipado `T-App` sabemos que $\Gamma \vdash$ `[x` $\mapsto$ `s]`t_1 `[x` $\mapsto$ `s]`t_2`: T`.

4. Caso `T-True`:

$$\texttt{t = true}$$
$$\texttt{T = Bool}$$

En este caso, $\texttt{[x} \mapsto \texttt{s]t = true}$ y el resultado, $\texttt{[x} \mapsto \texttt{s]t: T}$ es inmediato.

5. Caso `T-False`: análogo al caso anterior.
6. Caso `T-If`:

$$\texttt{t = if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$$
$$\Gamma\texttt{,x:S} \vdash t_1\texttt{: Bool}$$
$$\Gamma\texttt{,x:S} \vdash t_2\texttt{: T}$$
$$\Gamma\texttt{,x:S} \vdash t_3\texttt{: T}$$

Utilizando la hipótesis de inducción en cada uno de los subtérminos obtenemos:

$$\Gamma \vdash \texttt{[x} \mapsto \texttt{s]}t_1\texttt{: Bool}$$
$$\Gamma \vdash \texttt{[x} \mapsto \texttt{s]}t_2\texttt{: T}$$
$$\Gamma \vdash \texttt{[x} \mapsto \texttt{s]}t_3\texttt{: T}$$

Y aplicando `T-If` llegamos al resultado buscado.

Teorema *Preservación*: Si $\Gamma \vdash \texttt{t: T}$ y $\texttt{t} \rightarrow \texttt{t'}$, entonces $\Gamma \vdash \texttt{t': T}$

Prueba: muy similar a la prueba del teorema de preservación de expresiones aritméticas.

2.3. Borrado y tipabilidad

Tal como hemos definido las reglas de evaluación de nuestro lambda-cálculo tipado, las anotaciones de tipado no tienen ninguna influencia en la evaluación de los términos. De hecho, muchos lenguajes de programación usan las anotaciones de tipado durante la etapa de comprobación de tipos y las eliminan antes de llegar a la etapa de ejecución, de forma que las anotaciones de tipos no aparecen en la versión compilada del programa.

Para ello, los lenguajes de programación hacen uso de una función de *borrado* que transforma términos tipados en los correspondientes términos no tipados en el lambda-cálculo puro. La función de *borrado* se define de la siguiente manera:

$$\begin{array}{lcl} \mathit{erase}(\texttt{x}) & = & \texttt{x} \\ \mathit{erase}(\lambda\texttt{x} : \texttt{T}_1.\texttt{t}_2) & = & \lambda\texttt{x.erase}(\texttt{t}_2) \\ \mathit{erase}(\texttt{t}_1\ \texttt{t}_2) & = & \mathit{erase}(\texttt{t}_1)\ \mathit{erase}(\texttt{t}_2) \end{array}$$

Es de esperar que ambas semánticas del lambda cálculo tipado funcionen de forma equivalente, es decir, que el resultado de evaluar un término tipado y otro cuyas anotaciones de tipos hayan sido borradas sea el mismo. Para garantizar dicha equivalencia, podemos enunciar el siguiente teorema, que viene a formalizar el hecho de que "la evaluación conmuta con el borrado", es decir, que las operaciones pueden ser realizadas en cualquier orden.

Teorema :

1. Si`t` $\rightarrow$ `t'` en la evaluación tipada, entonces *erase*(`t`)$\rightarrow$*erase*(`t'`).
2. Si *erase*(`t`)$\rightarrow$`m'` en la evaluación tipada, entonces existe un término tipado `t'` tal que `t` $\rightarrow$ `t'` y *erase*(`t'`) = `m'`.

Lo que este teorema nos viene a decir es que la semántica de "alto nivel" que los programadores escriben tiene una equivalencia directa con la semántica de una implementación de bajo nivel del lenguaje.

2.4. El isomorfismo de Curry-Howard

El tipo "$\rightarrow$" utiliza dos reglas de tipado:

1. Una *regla de introducción* (`T-Abs`) que describe cómo se pueden *crear* elementos del tipo.

2. Una *regla de eliminación* (`T-App`) que describe cómo se pueden *usar* elementos del tipo.

La terminología de introducción y eliminación se usa muy amenudo al hablar de sistemas de tipos. Se origina debido a una conexión entre la lógica y la teoría de tipos conocida como *isomorfismo de Curry-Howard* o *correspondencia de Curry-Howard*.

Para entender el isomorfismo de Curry-Howard debemos definir brevemente la lógica constructiva. La mayor diferencia entre ésta y la lógica clásica es la eliminación de ciertos axiomas como la *ley del tercero excluído* que establece que, dada una proposición Q, o bien se cumple Q o bien se cumple $\neg Q$. Para probar $Q \vee \neg Q$ en lógica constructiva, debemos probar o bien Q o bien $\neg Q$.

La idea central de dicho isomorfismo es la siguiente: en lógica constructiva, una prueba de la proposición P consiste en proporcionar una evidencia concreta de P. Curry y Howard observaron, por ejemplo, que una prueba de la proposición $P \rightarrow Q$ puede ser interpretada como un procedimiento que, a partir de una prueba de P construye una prueba de Q. De forma similar, una prueba de $P \wedge Q$ consiste en una prueba de P y una prueba de Q. Dichas observaciones dieron lugar a la siguiente correspondencia:

LÓGICA	LENGUAJES DE PROGRAMACIÓN
proposiciones	tipos
proposición $P \rightarrow Q$	tipo `P` $\rightarrow$ `Q`
proposición $P \wedge Q$	tipo `P` $\times$ `Q`
proposición $P \vee Q$	tipo `P` + `Q`
prueba de proposición P	término `t` del tipo `P`
proposición P probable	tipo `P` tiene elementos

Según la tabla anterior, un término en el lambda cálculo con tipado simple es una prueba de una proposición lógica correspondiente a su tipo. La correspondencia de Curry-Howard se denomina también la analogía de *proposiciones como tipos*.

2.5. Estilo Curry vs. Estilo Church

Acabamos de ver dos estilos diferentes de definir la semántica del lambda-cálculo con tipos simples:

1. Como una relación de evaluación definida sobre la sintaxis tipada.
2. Como una compilación a un lambda cálculo no tipado más una relación de evaluación definida sobre los términos no tipados.

A la hora de definir un lenguaje de programación, también existen dos alternativas:

1. Estilo Curry: definir los términos, definir la semántica que indica cómo serán evaluados y por último definir el sistema de tipos que rechazará ciertos términos del lenguaje.
2. Estilo Church: definir los términos, identificar los términos bien tipados y definir la semántica de los términos bien tipados.

Históricamente, el estilo Curry se suele utilizar en los lenguajes implícitamente tipados, mientras que el estilo Church se suele utilizar en los lenguajes explícitamente tipados.

Capítulo 3

Extensiones simples

En este capítulo, enriqueceremos el lambda-cálculo tipado del capítulo anterior $\lambda_{\rightarrow}$ con numerosos elementos típicos de los lenguajes de programación de alto nivel.

3.1. Tipos básicos

Todos los lenguajes de programación contienen un conjunto de tipos básicos (valores simples y no estructurados como los números, booleanos y caracteres) junto con una serie de operaciones para manipularlos. En los capítulos anteriores hemos analizado detenidamente dos de dichos tipos básicos, los booleanos y los naturales. Los restantes (`Float`, `String`, ...) pueden ser introducidos en el lenguaje de forma análoga.

Vamos a suponer que nuestro lambda cálculo tipado viene equipado con un conjunto A de tipos básicos *no interpretados* sin operaciones primitivas definidas sobre los mismos. En general, utilizaremos las metavariables `A`, `B`, `C`, etc. para referirnos a un tipo básico no especificado. En este contexto, A puede ser entendido como el conjunto de tipos atómicos o tipos básicos de nuestro lenguaje.

3.2. El tipo Unit

Este tipo está formado por un único elemento, el término constante (o valor) `unit`. El tipo `Unit` resulta de gran interés en los lenguajes con efectos laterales. En ciertas expresiones de dichos lenguajes lo que resulta de interés es el efecto lateral y no el tipo devuelto por la expresión. Un ejemplo podría ser el caso de una expresión cuya finalidad fuera la de escribir un dato en un fichero; lo realmente importante es el efecto colateral, por lo que `Unit` podría ser un tipo adecuado para dicha expresión. El uso del tipo `Unit` sería similar al del tipo `void` en lenguajes como C o Java.

A continuación, mostramos las extensiones al lenguaje $\lambda_{\rightarrow}$ necesarias para definir el tipo `Unit`.

Nuevas formas sintácticas:

```
t::= ...
     unit

v::= ...
```

```
      unit

T::= ...
     Unit
```

Nueva regla de tipado:

$\Gamma \vdash$ `unit: Unit`

3.3. Formas derivadas: secuenciamiento y máscaras

En lenguajes con efectos laterales, en muchas ocasiones resulta de utilidad evaluar dos o más expresiones en una secuencia. La notación $\texttt{t}_1\texttt{;t}_2$ implica la evaluación de $\texttt{t}_1$ desechando su resultado y a continuación la evaluación de $\texttt{t}_2$.

Existen dos formas de formalizar el secuenciamiento en nuestro lenguaje:

1. Añadir $\texttt{t}_1\texttt{;}\ \texttt{t}_2$ al conjunto de formas sintácticas del lenguaje.

   Definir a continuación dos reglas de evaluación:

$$\frac{\texttt{t}_1 \rightarrow \texttt{t}_1'}{\texttt{t}_1\texttt{;t}_2 \rightarrow \texttt{t}_1'\texttt{;t}_2} \quad \texttt{E} - \texttt{Seq}$$
$$\texttt{unit;t}_2 \rightarrow \texttt{t}_2 \quad \texttt{E} - \texttt{SeqNext}$$

   Definir por último una regla de tipado

$$\frac{\Gamma \vdash \texttt{t}_1\texttt{:Unit} \qquad \Gamma \vdash \texttt{t}_2\texttt{:T}_2}{\Gamma \vdash \texttt{t}_1\texttt{;t}_2\texttt{:T}_2}$$

2. Considerar $\texttt{t}_1\texttt{;t}_2$ como una *abreviación* del término $(\lambda\texttt{x:Unit.t}_2)\texttt{t}_1$, donde la variable `x` es elegida *fresca* (diferente de las variables de $\texttt{t}_2$). En este caso no es necesario añadir ningún elemento nuevo (construcción sintáctica o reglas de evaluación y tipado) al lenguaje.

- **Teorema** [*El secuenciamiento es una forma derivada*]: Sean

  1. λ^E el *lenguaje externo*, es decir, el lambda cálculo con tipos simples más el tipo `Unit`, el constructor de secuencia, las reglas de evaluación `E-Sec` y `E-SecNext`, y la regla de tipado `T-Seq`.
  2. λ^I el *lenguaje interno*, es decir, el lambda cálculo con tipos simples más el tipo `Unit`.

  Si definimos la *función de elaboración* $e \in \lambda^E \rightarrow \lambda^I$ como la función que traduce del lenguaje externo al interno, sustituyendo cada aparición de $\texttt{t}_1\texttt{;t}_2$ por $(\lambda\texttt{x:Unit.t}_2)\texttt{t}_1$, entonces para cada término $\texttt{x} \in \lambda^E$ ha de cumplirse:

  - $\texttt{t} \rightarrow_E \texttt{t'}$ si y sólo si $e(\texttt{t}) \rightarrow_I e(\texttt{t}')$.
  - $\Gamma \vdash_E \texttt{t:T}$ si y sólo si $\Gamma \vdash_I e(\texttt{t:T})$.
    Demostración: cada dirección de la implicación se prueba por inducción en la estructura de `t`.

Las formas derivadas suelen denominarse *azúcar sintáctico (syntactic sugar)*, pues facilitan la escritura y lectura de los términos. Una construcción se denomina azúcar sintáctico si puede ser eliminada del lenguaje sin que ello afecte a la potencialidad de lo que el lenguaje puede hacer. La operación de transformar una forma derivada en su representación interna se denomina *quitar el azúcar (desugaring)*.

Otra forma derivada que también resulta de utilidad es la máscara para las anotaciones de variables. Cuando un término es de la forma λ`x:S.t` y la variable `x` no se utiliza en el cuerpo del término `t` es posible utilizar una máscara en lugar de un nombre de variable explícito. En este caso escribiremos λ`_:S.t`.

3.4. Adscripción

En ciertas ocasiones puede resultar conveniente indicar explícitamente el tipo de un determinado término, para lo cual añadiremos a nuestro lenguaje la siguiente forma sintáctica:

```
t::= ...
     t as T
```

Será necesario añadir también dos reglas de evaluación, que sencillamente descartarán la anotación de tipo una vez el término se haya evaluado a un valor:

$$\texttt{v}_1\ \texttt{as T} \rightarrow \texttt{v}_1 \quad (\texttt{E} - \texttt{Ascribe})$$

$$\frac{\texttt{t}_1 \rightarrow \texttt{t}_1'}{\texttt{t}_1\ \texttt{as T} \rightarrow \texttt{t}_1'\ \texttt{as T}} \quad (\texttt{E} - \texttt{Ascribe1})$$

Y una regla de tipado que compruebe que el tipo del término es verdaderamente igual al tipo indicado en la adscripción.

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}}{\Gamma \vdash \texttt{t}_1\ \texttt{as T} : \texttt{T}} \quad (\texttt{T} - \texttt{Ascribe})$$

La adscripción puede resultar útil para la programación desde diferentes puntos de vista. Se trata de una forma de documentación que facilitará la lectura de los programas; también es una herramienta útil durante la depuración, pues permite dividir una expresión en subexpresiones de cara a analizar el tipo de cada una de ellas por separado.

En capítulos posteriores enumeraremos otras utilidades de la adscripción.

- **Ejercicio:** Demostrar que la inscripción es una forma derivada. Encontrar la expresión equivalente en λ^I y demostrar que las reglas de evaluación y tipado se cumplen en nuestra definición de bajo nivel.

  `(`λ`x:T.x)t`

  La estrategia de evaluación de llamada por valor implica que `t` se evaluará a un valor antes de aplicar abstracción, lo cual es compatible con las reglas de evaluación. Además, según la regla de tipado de la aplicación `T-App`, para que el término esté bien tipado, `t` ha de ser de tipo `T`.

3.5. Sentencia let

Al escribir expresiones complejas, puede resultar de utilidad asignar identificadores a expresiones para mejorar la claridad del código y evitar repeticiones de código. Nuestra sentencia `let` es muy similar a la del lenguaje ML, su forma sintáctica es la siguiente:

```
t::= ...
     let x = t in t
```

Siguiendo una estrategia de evaluación por valor, el término asignado a la variable debe ser evaluado a un valor antes de sustituir dicho valor en el cuerpo de la sentencia `let`, lo cual queda reflejado en las reglas de evaluación:

$$\texttt{let x = v}_1\ \texttt{in t}_2 \rightarrow [\texttt{x} \mapsto \texttt{v}_1]\texttt{t}_2 \quad (\texttt{E} - \texttt{LetV})$$

$$\frac{\texttt{t}_1 \rightarrow \texttt{t}_1'}{\texttt{let x = t}_1\ \texttt{in t}_2 \rightarrow \texttt{let x = t}_1'\ \texttt{in t}_2} \quad (\texttt{E} - \texttt{Let})$$

Según nueva regla de tipado, para calcular el tipo de la sentencia `let` se debe calcular el tipo de la variable ligada, extender el contexto de tipado con dicha anotación y utilizar el contexto enriquecido para calcular el tipo del cuerpo, que será a su vez el tipo de la sentencia.

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1 \qquad \Gamma, \texttt{x:T}_1 \vdash \texttt{t}_2 : \texttt{T}_2}{\Gamma \vdash \texttt{let x = t}_1\ \texttt{in t}_2 : \texttt{T}_2} \quad (\texttt{T} - \texttt{Let})$$

La sentencia `let` también es una forma derivada, aunque con ciertas matizaciones. En principio, la sentencia $\texttt{let x=t}_1\ \texttt{in t}_2$ puede traducirse como $(\lambda \texttt{x:T}_1.\texttt{t}_2)\texttt{t}_1$ en el lenguaje interno. Sin embargo, la abstracción incluye el tipo de la variable ligada, mientras que la sentencia `let` no. Al *quitarle el azúcar* a la forma derivada, el compilador deberá utilizar la regla de tipado `T-Let` para determinar el tipo de la variable ligada. Esto significa que la sentencia `let` es "menos derivada" que las otras formas derivadas que hemos visto, ya que su regla de tipado debe ser incluida en el lenguaje.

3.6. Pares

La mayor parte de los lenguajes de programación ofrecen diversas alternativas para construir estructuras de datos complejas, siendo la más sencilla los pares de valores. Para añadir pares a nuestro lambda-cálculo con tipado simple tendremos que añadir un conjunto de convenciones sintácticas:

```
t::= ...
     {t,t}
     t.1
     t.2

v::= ...
     {v,v}

T::= ...
     T1×T2
```

El nuevo tipo del lenguaje se llama *producto*. También tendremos que añadir un conjunto de reglas que definan la evaluación:

$$\{\mathtt{v}_1, \mathtt{v}_1\}.1 \rightarrow \mathtt{v}_1 \quad (\mathtt{E-PairBeta1})$$

$$\{\mathtt{v}_1, \mathtt{v}_1\}.2 \rightarrow \mathtt{v}_2 \quad (\mathtt{E-PairBeta2})$$

$$\frac{\mathtt{t}_1 \rightarrow \mathtt{t}_1'}{\mathtt{t}_1.1 \rightarrow \mathtt{t}_1'.1} \quad (\mathtt{E-Proj1})$$

$$\frac{\mathtt{t}_1 \rightarrow \mathtt{t}_1'}{\mathtt{t}_1.2 \rightarrow \mathtt{t}_1'.2} \quad (\mathtt{E-Proj2})$$

$$\frac{\mathtt{t}_1 \rightarrow \mathtt{t}_1'}{\{\mathtt{t}_1,\mathtt{t}_2\} \rightarrow \{\mathtt{t}_1',\mathtt{t}_2\}} \quad (\mathtt{E-Pair1})$$

$$\frac{\mathtt{t}_2 \rightarrow \mathtt{t}_2'}{\{\mathtt{v}_1,\mathtt{t}_2\} \rightarrow \{\mathtt{v}_1,\mathtt{t}_2'\}} \quad (\mathtt{E-Pair2})$$

El uso de metavariables `t` y `v` en las reglas anteriores define de modo unívoco el orden de evaluación (de izquierda a derecha) de los pares. Por ejemplo, la sentencia:

```
{pred 4,if true then false else false}.1
```

se evalúa de la siguiente forma:

```
{pred 4,if true then false else false}.1 →
{3,if true then false else false}.1 →
{3,false}.1 →
3
```

El haber añadido $\mathtt{v}_1,\mathtt{v}_2$ a la lista de posibles valores de nuestro sistema implica que, cuando un par es pasado como argumento a una abstracción, va a ser evaluado totalmente antes de comenzar a ejecutar el cuerpo de la abstracción. Por ejemplo:

```
(λx:Nat×Nat.x.2){pred 4,pred 5} →
(λx:Nat×Nat.x.2){3,pred 5} →
(λx:Nat×Nat.x.2){3,4} →
{3,4}.2 →
4
```

Por último, también tendremos que introducir una serie de reglas de tipado.

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{t}_1,\mathtt{t}_2 : \mathtt{T}_1 \times \mathtt{T}_2} \quad (\mathtt{T-Pair})$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \times \mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1.1 : \mathtt{T}_{11}} \quad (\mathtt{T-Proj1})$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \times \mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1.2 : \mathtt{T}_{12}} \quad (\mathtt{T-Proj2})$$

3.7. Tuplas

Las tuplas son una generalización de los productos binarios a productos n-arios. Por ejemplo, {`1,2,true`} es una tupla de tres elementos, dos números y un booleano, cuyo tipo es {`Nat,Nat,Bool`}.

A la hora de generalizar la notación, escribiremos $\{\mathtt{t}_i{}^{i=1..n}\}$ para referirnos a una tupla de `n` términos y $\{\mathtt{T}_i{}^{i=1..n}\}$ para referirnos a su tipo. Siguiendo esta notación,

`n` podría ser 0, en cuyo caso {} sería la tupla vacía. Igualmente, el término `5` sería diferente del término {`5`}, siendo sus tipos `Nat` y {`Nat`}, respectivamente.

Para añadir tuplas a nuestro lenguaje, añadiremos las siguientes convenciones sintácticas:

```
t::= ...
     {t_i^{i=1..n}}
     t.i

v::= ...
     {v_i^{i=1..n}}

T::=
     {T_i^{i=1..n}}
```

Añadiremos también las siguientes reglas de evaluación:

$$\{\mathtt{v}_i^{i=1..n}\}.\mathtt{j} \rightarrow \mathtt{v}_j \qquad (\mathtt{E-ProjTuple})$$

$$\frac{\mathtt{t}_1 \rightarrow \mathtt{t}_1'}{\mathtt{t}_1.\mathtt{i} \rightarrow \mathtt{t}_1'.\mathtt{i}} \qquad (\mathtt{E-Proj})$$

$$\frac{\mathtt{t}_j \rightarrow \mathtt{t}_j'}{\{\mathtt{v}_i^{i=1..j-1}, \mathtt{t}_j, \mathtt{t}_k^{k=1..n}\} \rightarrow \{\mathtt{v}_i^{i=1..j-1}, \mathtt{t}_j, \mathtt{t}_k^{k=1..n}\}} \qquad (\mathtt{E-Tuple})$$

Donde la regla `E-Tuple` indica que los términos de una tupla se irán evaluando de izquierda a derecha.

Y, por último, las siguientes reglas de tipado:

$$\frac{\forall \mathtt{i} \Gamma \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{t}_i^{i=1..n}\} : \{\mathtt{T}_i^{i=1..n}\}} \qquad (\mathtt{T-Tuple})$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{T}_i^{i=1..n}\}}{\Gamma \vdash \mathtt{t}_1.\mathtt{j} : \mathtt{T}_j} \qquad (\mathtt{T-Proj})$$

3.8. Registros

Generalizar las tuplas a registros etiquetados resulta bastante sencillo. Basta con anotar cada campo $\mathtt{t}_i$ con una etiqueta $\mathtt{l}_i$ tomada de un conjunto predeterminado L. Por ejemplo, {`x=5`} y {`partno=5524,const= 30.27`} son ambos registros con tipos {`x:Nat`} y {`partno:Nat,const:Float`}, respectivamente. Todas las etiquetas de un registro han de ser diferentes entre si.

Las extensiones a la sintaxis, semántica y relación de tipado necesarias para incluir los registros en nuestro lenguaje son enumeradas a continuación:

```
t::= ...
     {l_i=t_i^{i=1..n}}
     t.l

v::= ...
     {l_i=v_i^{i=1..n}}
```

$$\begin{array}{l}\texttt{T::=}\\ \quad \{\texttt{l}_\texttt{i}:\texttt{T}_\texttt{i}^{\,\texttt{i=1..n}}\}\end{array}$$

$$\{\texttt{l}_\texttt{i} = \texttt{v}_\texttt{i}^{\,\texttt{i=1..n}}\}.\texttt{l}_\texttt{j} \to \texttt{v}_\texttt{j} \qquad (\texttt{E}-\texttt{ProjRcd})$$

$$\frac{\texttt{t}_1 \to \texttt{t}_1'}{\texttt{t}_1.\texttt{l} \to \texttt{t}_1'.\texttt{l}} \qquad (\texttt{E}-\texttt{Proj})$$

$$\frac{\texttt{t}_\texttt{j} \to \texttt{t}_\texttt{j}'}{\{\texttt{l}_\texttt{i}=\texttt{v}_\texttt{i}^{\,\texttt{i=1..j-1}},\texttt{l}_\texttt{j}=\texttt{t}_\texttt{j},\texttt{l}_\texttt{k}=\texttt{t}_\texttt{k}^{\,\texttt{k=1..n}}\} \to \{\texttt{l}_\texttt{i}=\texttt{v}_\texttt{i}^{\,\texttt{i=1..j-1}},\texttt{l}_\texttt{j}=\texttt{t}_\texttt{j},\texttt{l}_\texttt{k}=\texttt{t}_\texttt{k}^{\,\texttt{k=1..n}}\}} \qquad (\texttt{E}-\texttt{Rcd})$$

$$\frac{\forall \texttt{i}\, \Gamma \vdash \texttt{t}_\texttt{i}:\texttt{T}_\texttt{i}}{\Gamma \vdash \{\texttt{l}_\texttt{i}=\texttt{t}_\texttt{i}^{\,\texttt{i=1..n}}\}:\{\texttt{l}_\texttt{i}:\texttt{T}_\texttt{i}^{\,\texttt{i=1..n}}\}} \qquad (\texttt{T}-\texttt{Rcd})$$

$$\frac{\Gamma \vdash \texttt{t}_1:\{\texttt{l}_\texttt{i}:\texttt{T}_\texttt{i}^{\,\texttt{i=1..n}}\}}{\Gamma \vdash \texttt{t}_1.\texttt{l}_\texttt{j}:\texttt{T}_\texttt{j}} \qquad (\texttt{T}-\texttt{Proj})$$

Las tuplas pueden ser consideradas como un caso especial de los registros. Para ello, basta con considerar el conjunto de etiquetas de las tuplas como la sucesión de números `1,2,3,...`. En este caso, el tipo {`Bool,Nat,Bool`} sería una abreviación de {`1:Bool,2:Nat,3:Bool`}. Siguiendo esta convención, podríamos combinar campos etiquetados con campos no etiquetados; por ejemplo, el tipo {`a:Bool,Nat,c:Bool`} sería una abreviación de {`a:Bool,2:Nat,c:Bool`}. Si muchos lenguajes de programación utilizan notaciones distintas para las tuplas y los registros es por una razón meramente pragmática: que el compilador los implementa de manera diferente.

En algunos lenguajes de programación el orden de los campos de un registro no afecta a su significado, es decir, {`partno=5524,const= 30.27`} y {`const=30.27, partno=5524`} tienen el mismo significado y el mismo tipo. Por el momento, ese no va a ser el caso en nuestro lenguaje de programación, en el que el orden de los campos sí influirá en el significado y tipado del registro.

3.8.1. Pattern matching

Haciendo uso de los registros, podemos añadir una forma de pattern matching a nuestro lambda-cálculo añadiendo para ello una nueva categoría sintáctica, los *patrones* y una nueva construcción al conjunto de términos:

```
p::= x
     {li=pi^{i=1..n}}

t::= ...
     let p=t in t
```

Un patrón puede ser de la forma `x`, {`x,y`} o {`x,{y,z}`}.

Las reglas de evaluación hacen uso de una función auxiliar `match` que, dados un patrón `p` y un valor `v` o bien falla o bien mapea los campos de `v` en los campos de `p`. Por ejemplo, `match({x,y},{5,true})` da lugar a la sustitución `[x` $\mapsto$`5, y` $\mapsto$`true]`, `match(x,{5,true})` da lugar a `[x` $\mapsto$ {`5,true`}`]` mientras que `match({x},{5,true})` falla.

$$\texttt{match}(\texttt{x},\texttt{v}) = [\texttt{x} \mapsto \texttt{v}] \qquad (\texttt{M}-\texttt{Var})$$

$$\frac{\textit{para cada } \texttt{i} \quad \texttt{match}(\texttt{p}_\texttt{i},\texttt{v}_\texttt{i})=\sigma_i}{\texttt{match}(\{\texttt{l}_\texttt{i}=\texttt{p}_\texttt{i}^{\,\texttt{i=1..n}}\},\{\texttt{l}_\texttt{i}=\texttt{v}_\texttt{i}^{\,\texttt{i=1..n}}\})=\sigma_1 \circ \ldots \circ \sigma_n} \qquad (\texttt{M}-\texttt{Rcd})$$

La regla `M-Var` indica que el mapeo de variables siempre funciona. La regla `M-Rcd` indica que, para poder mapear $\{\texttt{l}_\texttt{i}=\texttt{p}_\texttt{i}^{\,\texttt{i=1..n}}$ con $\{\texttt{l}_\texttt{i}=\texttt{v}_\texttt{i}^{\,\texttt{i=1..n}}$ debemos mapear cada sub-patrón con cada sub-valor y calcular el valor final de la sustitución como la composición de todas las sustituciones parciales.

Finalmente, la regla de evaluación `E-LetV` utiliza `match` para calcular la sustitución adecuada de variables en `p`.

$$\texttt{letp} = \texttt{v}_1\texttt{int}_2 \rightarrow \texttt{match}(\texttt{p},\texttt{v}_1)\texttt{t}_2 \qquad (\texttt{T}-\texttt{LetV})$$

$$\frac{\texttt{t}_1 \rightarrow \texttt{t}_1'}{\texttt{letp}=\texttt{t}_1\texttt{int}_2 \rightarrow \texttt{letp}=\texttt{t}_1'\texttt{int}_2} \qquad (\texttt{M}-\texttt{Let})$$

3.9. Sumas

En muchos programas resulta necesario trabajar con colecciones de datos heterogéneas. Por ejemplo, un nodo en un árbol binario puede ser una hoja o un nodo intermedio con dos hijos; de forma similar, en una lista las celdas pueden ser `nil` o estar compuestas por una cabeza y una cola. En lo que a tipos se refiere, el mecanismo que permite dar soporte a esta forma de programación son los *tipos variantes*.

En esta sección vamos a analizar el tipo variante más sencillo que existe, el *tipo suma*. Un tipo suma describe a un conjunto de valores que pueden pertenecer a dos tipos de datos dados. Por ejemplo, para representar dos tipos de registros diferentes en una libreta de direcciones podemos utilizar los siguientes tipos:

```
PhysicalAddr = {firstLast:String,addr:String}
VirtualAddr = {name:String,email:String}
```

Si queremos manipular ambos tipos de forma homogénea (por ejemplo, si queremos crear una lista conteniendo ambos tipos de registros) podemos crear un tipo suma:

```
Addr = PhysicalAddr + VirtualAddr
```

Para crear elementos de este tipo a partir de elementos de los tipos constituyentes podemos utilizar los tokens `inl` y `inr`. Estos tokens inyectan elementos de los tipos `PhysicalAddr` y `VirtualAddr` en la parte izquierda y derecha del tipo `Addr`, respectivamente.

```
inl : PhysicalAddr -> PhysicalAddr+VirtualAddr
inr : VirtualAddr -> PhysicalAddr+VirtualAddr
```

Así pues, los elementos del tipo $\texttt{T}_1\texttt{+}\texttt{T}_2$ serán elementos del tipo $\texttt{T}_1$ etiquetados con el token `inl` más elementos del tipo $\texttt{T}_2$ etiquetados con el token `inr`.

Para poder utilizar los elementos de tipos sumas introduciremos un constructor `case` que nos permitirá determinar cuando un elemento proviene de la parte derecha o izquierda de la suma. Por ejemplo, para extraer un nombre de un elemento del tipo `Addr` podemos escribir la siguiente función de tipo `Addr`→`String`:

```
getName = λa:Addr.
   case a of
      inl x => x.firstLast
      |inr y => y.name
```

Para definir los tipos sumas en nuestro lenguaje debemos añadir los siguientes elementos sintácticos:

```
t::= ...
     inl t
     inr t
     case t of inl x => t | inr x => t

v::= ...
     inl v
     inr v

T::= ...
      T+T
```

Debemos añadir también las siguientes reglas de evaluación:

$$\texttt{case (inl } v_0\texttt{) of inl } x_1 \texttt{ => } t_1 \texttt{ | inr } x_2 \texttt{ => } t_2 \rightarrow [x_1 \mapsto v_0]t_1 \qquad (\texttt{E-CaseInl})$$

$$\texttt{case (inr } v_0\texttt{) of inl } x_1 \texttt{ => } t_1 \texttt{ | inr } x_2 \texttt{ => } t_2 \rightarrow [x_2 \mapsto v_0]t_2 \qquad (\texttt{E-CaseInr})$$

$$\frac{t_0 \rightarrow t_0'}{\texttt{case } t_0 \texttt{ of inl } x_1 \texttt{ => } t_1 \texttt{ | inr } x_2 \texttt{ => } t_2 \rightarrow \texttt{case } t_0' \texttt{ of inl } x_1 \texttt{ => } t_1 \texttt{ | inr } x_2 \texttt{ => } t_2} \qquad (\texttt{E-Case})$$

$$\frac{t_1 \rightarrow t_1'}{\texttt{inl } t_1 \rightarrow \texttt{inl } t_1} \qquad (\texttt{E-Inl})$$

$$\frac{t_1 \rightarrow t_1'}{\texttt{inr } t_1 \rightarrow \texttt{inr } t_1} \qquad (\texttt{E-Inr})$$

Por último, añadiremos las siguientes reglas de tipado:

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{inl } t_1 : T_1 + T_2} \qquad (\texttt{T-Inl})$$

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \texttt{inr } t_2 : T_1 + T_2} \qquad (\texttt{T-Inr})$$

$$\frac{\Gamma \vdash t_0 : T_1 + T_2 \qquad \Gamma x_1 : T_1 \vdash t_1 : T \qquad \Gamma x_2 : T_2 \vdash t_2 : T}{\Gamma \vdash \texttt{case } t_0 \texttt{ of inl } x \texttt{ => } t_1 \texttt{ | inr } x_2 \texttt{ =>} t_2 : T} \qquad (\texttt{T-Case})$$

La regla `T-Case` nos indica que la guarda de la sentencia `case` debe tener tipo suma y que las dos ramas de la sentencia tienen el mismo tipo, tal como ocurría con la sentencia `if`.

- **Ejercicio:** La sentencia `if` puede considerarse un caso particular de la sentencia `case` en la que no se pasa ninguna información a cada una de las ramas. Definir `true`, `false` y `if` como formas derivadas utilizando tipos sumas y `Unit`.

Sumas y unicidad de tipos

La mayor parte de las propiedades del tipado del lambda cálculo $\lambda_{\rightarrow}$ se cumplen al añadir los tipos sumas, sin embargo hay una que falla: la unicidad de los tipos. El problema surge a la hora de tipar las construcciones `inl` e `inr`. Según la regla de tipado T-Inl, una vez que sabemos que $\texttt{t}_1 : \texttt{T}_1$, podríamos deducir que $\texttt{inl t}_1 : \texttt{T}_1\texttt{+T}_2$ para cualquier tipo $\texttt{T}_2$. Es decir, `inl 5` puede ser interpretado como de tipo `Nat+Nat` o `Nat+Bool`. Al fallar el teorema de la unicidad de tipos, el algoritmo de tipado se complica pues ya no consiste únicamente en ir aplicando reglas de tipado. Llegados a este punto, tenemos varias opciones:

1. Complicar el algoritmo de tipado para que infiera el valor de $\texttt{T}_2$.
2. Refinar el lenguaje de tipos para representar uniformemente todos los valores posibles de $\texttt{T}_2$.
3. Obligar al programador a que indique explícitamente el tipo $\texttt{T}_2$.

La tercera alternativa es la más sencilla y la que adoptaremos por el momento (hasta que estudiemos la inferencia de tipos y el subtipado) pero requiere de un conjunto de extensiones a la sintaxis y reglas de los tipos sumas. Concretamente, las extensiones necesarias son las siguientes:

```
t::= ...
     inl t as T
     inr t as T
     case t of inl x => t | inr x => t

v::= ...
     inl v as T
     inr v as T

T::= ...
      T+T
```

$$\texttt{case (inl } v_0 \texttt{ as } T_0\texttt{) of inl } x_1 \texttt{ => } t_1 \texttt{ | inr } x_2 \texttt{ => } t_2 \rightarrow [x_1 \mapsto v_0]t_1 \quad (\texttt{E} - \texttt{CaseInl})$$

$$\texttt{case (inr } v_0 \texttt{ as } T_0\texttt{) of inl } x_1 \texttt{ => } t_1 \texttt{ | inr } x_2 \texttt{ => } t_2 \rightarrow [x_2 \mapsto v_0]t_2 \quad (\texttt{E} - \texttt{CaseInr})$$

$$\frac{t_0 \rightarrow t_0'}{\texttt{case } t_0 \texttt{ of inl } x_1 \texttt{ => } t_1 \texttt{ | inr } x_2 \texttt{ => } t_2 \rightarrow \texttt{case } t_0' \texttt{ of inl } x_1 \texttt{ => } t_1 \texttt{ | inr } x_2 \texttt{ => } t_2} \quad (\texttt{E} - \texttt{Case})$$

$$\frac{t_1 \rightarrow t_1'}{\texttt{inl } t_1 \texttt{ as } T \rightarrow \texttt{inl } t_1 \texttt{ as} T} \quad (\texttt{E} - \texttt{Inl})$$

$$\frac{t_1 \rightarrow t_1'}{\texttt{inr } t_1 \texttt{ as } T \rightarrow \texttt{inr } t_1 \texttt{as} T} \quad (\texttt{E} - \texttt{Inr})$$

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{inl } t_1 \texttt{ as } T_1\texttt{+}T_2 : T_1\texttt{+}T_2} \quad (\texttt{T} - \texttt{Inl})$$

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \texttt{inr } t_2 \texttt{ as } T_1\texttt{+}T_2 : T_1\texttt{+}T_2} \quad (\texttt{T} - \texttt{Inr})$$

$$\frac{\Gamma \vdash t_0 : T_1\texttt{+}T_2 \qquad \Gamma x_1 : T_1 \vdash t_1 : T \qquad \Gamma x_2 : T_2 \vdash t_2 : T}{\Gamma \vdash \texttt{case } t_0 \texttt{ of inl x => } t_1 \texttt{ | inr } x_2 \texttt{ =>} t_2 : T} \quad (\texttt{T} - \texttt{Case})$$

En lugar de escribir `inl t` o `inr t` escribimos `inl t as T` o `inr t as T` donde `T` es el tipo suma.

3.10. Variantes

Las sumas binarias se generalizan a variantes anotadas de forma similar a como los productos se generalizaban a registros anotados. En lugar de T_1+T_2 escribiremos $<l_1:T_1,l_2:T_2>$. En lugar de `inl t as` T_1+T_2 escribiremos $<l_1=t>$ `as` $<l_1:T_1,l_2:T_2>$. Por último, en lugar de anotar las ramas de la sentencia `case` con `inl` e `inr`, las anotaremos con las etiquetas correspondientes del tipo variante.

Vamos a transformar el ejemplo de la sección anterior `getAddr` para trabajar con tipos variantes:

```
Addr = < physical:PhysicalAddr, virtual:VirtualAddr >
a = <physical=pa> as Addr;
> a : Addr

getName = λa:Addr.case a of
      <physical=x> => x.firstLast
      | <virtual=y> => y.name;
> getName: Addr → String
```

La definición formal de las variantes aparece a continuación. Cabe destacar que, al igual que ocurría con los registros, el orden de las etiquetas es significativo:

$$
\begin{array}{l}
\texttt{t::= ...} \\
\qquad \texttt{<l=t> as T} \\
\qquad \texttt{case t of <} l_i\texttt{=}x_i \texttt{> =>} \; t_i^{\,i=1..n} \\
\texttt{T::= ...} \\
\qquad \texttt{<}l_i\texttt{:}T_i^{\,i=1..n}\texttt{>}
\end{array}
$$

$$\texttt{case (<}\, l_j = v_j \,\texttt{>}\ \texttt{as T) of}\ \ <\, l_i = x_i \,>\texttt{=>}\, t_i^{\,i=1..n} \;\rightarrow\; [x_j \mapsto v_j]t_j \qquad (\mathrm{E-CaseVariant})$$

$$\frac{t_0 \rightarrow t_0'}{\texttt{case } t_0 \texttt{ of <}l_i\texttt{=}x_i\texttt{> =>}\, t_i^{\,i=1..n} \rightarrow \texttt{case } t_0' \texttt{ of <}l_i\texttt{=}x_i\texttt{> =>}\, t_1^{\,i=1..n}} \qquad (\mathrm{E-Case})$$

$$\frac{t_i \rightarrow t_i'}{\texttt{<}l_i\texttt{=}t_i\texttt{> as T} \rightarrow \texttt{<}l_i\texttt{=}t_i'\texttt{> as T}} \qquad (\mathrm{E-Variant})$$

$$\frac{\Gamma \vdash t_j : T_j}{\Gamma \vdash \texttt{<}l_j\texttt{=}t_j\texttt{> as <}l_j\texttt{=}T_j^{\,j=1..n}\texttt{>:<}l_j\texttt{=}T_j^{\,j=1..n}\texttt{>}} \qquad (\mathrm{T-Variant})$$

$$\frac{\Gamma \vdash t_0 : \texttt{<}l_j\texttt{=}T_j^{\,j=1..n}\texttt{>} \qquad \textit{para cada } \texttt{i } \Gamma, x_i : T_i \vdash t_i : T}{\Gamma \vdash \texttt{case } t_0 \texttt{ of <}l_i\texttt{=}x_i\texttt{> =>}\, t_i^{\,i=1..n} : T} \qquad (\mathrm{T-Case})$$

Opciones

Los *valores opcionales* son una construcción muy útil basada en variantes. Por ejemplo, un valor del tipo:

```
OptionalNat = <none:Unit,some:Nat>
```

podría ser el valor trivial `unit` etiquetado con `none` o un número etiquetado con `some`. Por ejemplo, el siguiente tipo representa mapeos parciales de números a números:

```
Table = Nat → OptionalNat
```

Usando dicho tipo, la tabla vacía podría definirse como una función constante que devuelve `none` para cualquier entrada posible:

```
emptyTable: λn:Nat. <none=unit> as OptionNat;
> emptyTable : Table
```

La función `extendTable` añade o sobreescribe el valor asignado al número `m` en otra tabla:

```
extendTable =
   λt:Table.λm:Nat.λv:Nat.
      λn:Nat.
         if equal n m then <some=v> as OptionalNat
         else t n;
>extendTable : Table → Nat → Nat → Table
```

Si queremos comprobar el valor asignado por un objeto de tipo `Table` a un determinado número podemos usar una sentencia case:

```
lookup = λn.case t n of
     <none=u> => 999
     |<some=v> => v;
```

en la sentencia anterior devolvemos `999` en caso de que `t` esté indefinido en `n`.

Muchos lenguajes de programación proporcionan soporte para las opciones. OCaml proporciona un tipo `Option` mientras que en lenguajes como C++ o Java el valor `null` hace que los "tipos referencia" sean una forma oculta de definir opciones.

Enumeraciones

Las enumeraciones o tipos enumerados son una categoría especial de tipos variantes en los que el tipo de todos los campos es `Unit`. Por ejemplo, un tipo que represente los días laborables de la semana se define como:

```
Weekday = <monday:Unit,tuesday:Unit,wednesday:Unit,
           thursday:Unit,friday:Unit>;
```

Los elementos del tipo anterior son de la forma `<monday=unit> as Weekday`. De hecho, el tipo `Weekday` tiene únicamente cinco elementos. Para definir operaciones sobre las enumeraciones podemos utilizar la sentencia `case`:

```
nextBusinessDay = λw.Weekday.
   case w of <monday=x> => <tuesday=unit> as Weekday
      | case w of <tuesday=x> => <wednesday=unit> as Weekday
      | case w of <wednesday=x> => <thursday=unit> as Weekday
      | case w of <thursday=x> => <friday=unit> as Weekday
      | case w of <friday=x> => <monday=unit> as Weekday
```

Variantes de un solo campo

Un caso especial de variantes son las variantes de un solo campo:

```
V = <l:T>
```

En principio, este tipo de variantes podría parecer poco interesante. Su principal cualidad es que no se puede acceder directamente al elemento contenido en la variante, es decir, no se pueden realizar con él las operaciones típicas del tipo `T`. Vamos a ilustrar la utilidad de las variantes de un solo campo con el siguiente ejemplo.

Imaginemos que estamos escribiendo un programa que tiene que llevar a cabo cálculos financieros en diferentes divisas. Si todas las divisas se representan a través del tipo `Float` entonces tendríamos funciones como las siguientes:

```
dollars2euros = λd:Float. timesFloat d 1.34
> dollars2euros : Float → Float
euros2dollars = λe:Float.timesFloat e 0.74
> euros2dollars : Float → Float
```

Tras haber realizado numerosas operaciones, especialmente si hubiese más de dos divisas implicadas, podríamos perder la cuenta de si la cantidad almacenada en una determinada variable o celda de una base de datos está en euros o en dólares. Sin embargo, si utilizaramos los siguientes tipos:

```
DollarAmount = <dollars:Float>
EuroAmount = <euros:Float>
```

podríamos definir versiones seguras de las funciones de conversión que aceptasen únicamente cantidades en la moneda correcta:

```
dollars2euros =
   λd:DollarAmount.
      case d of <dollars=x> =>
      <euros = timesFloat x 1.34> as EuroAmount;
> dollars2euros : DollarAmount → EuroAmount
```

Una vez hecho esto, el sistema de tipos se encargará no sólo de garantizar que la cantidad de entrada es del tipo correcto, sino que también especificará la moneda del tipo de salida.

Las sumas y variantes se denominan también *uniones disjuntas*. El tipo $\texttt{T}_1\texttt{+T}_2$ es la unión de los tipos $\texttt{T}_1$ y $\texttt{T}_2$ porque incluye todos los elementos de ambos tipos. Es disjunto porque un término de dicho tipo contendrá un elemento o bien de $\texttt{T}_1$ o bien de $\texttt{T}_2$

3.11. Recursividad

Otra característica de la mayor parte de los lenguajes de programación es la posibilidad de definir funciones recursivas. En el lambda cálculo puro, las funciones recursivas se definían por medio del operador de punto fijo `fix`.

En el lambda cálculo con tipos simples, las funciones recursivas pueden definirse de forma similar. Por ejemplo, la función `iseven` devuelve `true` cuando se aplica a un número par y `false` en caso contrario:

```
 ff = λie:Nat→Bool.
      λx:Nat.
        if iszero x then true
        else if iszero (pred 0) then false
        else ie (pred (pred x));
> ff : (Nat → Bool) → Nat → Bool
iseven = fix ff
> iseven : Nat → Bool
iseven 7
> false : Bool
```

Si la función `ff` se aplica a una función `ie` que aproxime el funcionamiento de `iseven` hasta un cierto valor de `n`, entonces aproximará el funcionamiento de `iseven` hasta un valor `n+2`. Al aplicar `fix` a `ff` devolvemos su punto fijo, es decir, una función que produce el resultado correcto para todos los valores posibles de `n`.

El problema que tenemos ahora es que `fix` no puede definirse en nuestro lambda cálculo con tipos simples, dado que es imposible tiparlo en un sistema con las características del actual. Por lo tanto, en lugar de definir `fix` como un término de nuestro lenguaje, debemos añadir una nueva primitiva a nuestro lenguaje con unas reglas de evaluación que emulen el funcionamiento de combinador no tipado `fix` y su correspondiente regla de tipado.

```
t::= ...
      fix t
```

$$\texttt{fix}(\lambda \texttt{x} : \texttt{T}_1.\texttt{t}_2) \rightarrow [\texttt{x} \mapsto (\texttt{fix}(\lambda \texttt{x} : \texttt{T}_1 : \texttt{t}_2))]\texttt{t}_2 \quad (\texttt{E} - \texttt{FixBeta})$$

$$\frac{\texttt{t}_1 \rightarrow \texttt{t}_1'}{\texttt{fix t}_1 \rightarrow \texttt{fix t}_1'} \quad (\texttt{E} - \texttt{Fix})$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1 \rightarrow \texttt{T}_1}{\Gamma \vdash \texttt{fix t}_1 : \texttt{T}_1} \quad (\texttt{T} - \texttt{Fix})$$

Ejercicio: Definir `equal`, `plus`, `times` y `factorial` usando `fix`.

El operador `fix` suele utilizarse para construir funciones a partir de un punto fijo de funciones de funciones a funciones. Sin embargo, en la regla `T-Fix` el tipo $\texttt{T}_1$ no tiene por qué ser un tipo función. Por ejemplo, si $\texttt{T}_1$ es un tipo registro, se puede utilizar `fix` para definir funciones mutuamente recursivas, como en el siguiente ejemplo:

```
ff = λieio:{iseven:Nat→Bool,isodd:Nat→Bool}.
     {iseven = λx:Nat.
          if iszero x then true
          else ieio.isodd (pred x),
     isodd = λx:Nat.
          if iszero x then false
          else ieio.iseven (pred x)};
> f : {iseven:Nat→Bool,isodd:Nat→Bool} →
     {iseven:Nat→Bool,isodd:Nat→Bool}
```

Aplicando el operador punto fijo a la función `ff` el resultado es el siguiente:

```
r = fix ff
> r : {iseven:Nat→Bool,isodd:Nat→Bool}
```

Y proyectando la primera componente obtenemos la función `iseven`.

Por último, podemos añadir una forma derivada para reflejar el caso, de uso muy común, en el que queramos asignarle un nombre al resultado de una definición recursiva. La forma derivada será la siguiente:

```
letrec x:T1=t1 in t2
= let x = fix (λx:T1.t1) in t2
```

Haciendo uso de la forma derivada `letrec`, la definición de `iseven` sería la siguiente:

```
letrec iseven : Nat → Bool =
   λx:Nat.
      if iszero x then true
      else if iszero (pred x) then false
      else iseven (pred (pred x))
in
   iseven 7;
> false : Bool
```

Ejercicio: Reescribir las funciones `plus, times` y `factorial` utilizando la forma derivada `letrec` en lugar de `fix`.

3.12. Listas

Las características de tipado estudiadas hasta el momento en el curso pueden clasificarse en *tipos básicos* como `Bool` o `Unit` o como `constructores de tipos`, como → o ×. Un nuevo constructor de tipos es `List` que, dado un tipo `T` permite definir listas de longitud finita sobre elementos de `T`.

Las formas sintácticas a añadir a nuestra gramática son las siguientes:

```
t::= ...
     nil[T]
     cons[T] t t
     isnil[T] t
     head[T] t
     tail[T] t
v::= ...
     nil[T]
     cons[T] v v
T::= ...
     List T
```

La lista vacía con elementos de tipo `T` se representa como `nil[T]`. La lista resultante de añadir un elemento t_1 al frente de la lista t_2 se representa como `cons[T]` t_1 t_2.

Las nuevas reglas de evaluación son las siguientes:

$$\frac{\texttt{t}_1 \rightarrow \texttt{t}_1'}{\texttt{cons [T] t}_1\ \texttt{t}_2 \rightarrow \texttt{cons [T] t}_1'\ \texttt{t}_2} \quad (\texttt{E} - \texttt{Cons1})$$

$$\frac{\texttt{t}_2 \rightarrow \texttt{t}_2'}{\texttt{cons [T] v}_1\ \texttt{t}_2 \rightarrow \texttt{cons [T] v}_1\ \texttt{t}_2'} \quad (\texttt{E} - \texttt{Cons2})$$

$$\texttt{isnil[S] (nil[T])} \rightarrow \texttt{true} \quad (\texttt{E} - \texttt{IsNilNil})$$

$$\texttt{isnil[S] (cons[T] v}_1\ \texttt{v}_2\texttt{)} \rightarrow \texttt{false} \quad (\texttt{E} - \texttt{IsNilCons})$$

$$\frac{\texttt{t}_1 \rightarrow \texttt{t}_1'}{\texttt{isnil [T] t}_1 \rightarrow \texttt{isnil [T] t}_1'} \quad (\texttt{E} - \texttt{IsNil})$$

$$\texttt{head[S] (cons[T] v}_1\ \texttt{v}_2\texttt{)} \rightarrow v_1 \quad (\texttt{E} - \texttt{HeadCons})$$

$$\frac{\texttt{t}_1 \rightarrow \texttt{t}_1'}{\texttt{head [T] t}_1 \rightarrow \texttt{head [T] t}_1'} \quad (\texttt{E} - \texttt{Head})$$

$$\texttt{tail[S] (cons[T] v}_1\ \texttt{v}_2\texttt{)} \rightarrow v_2 \quad (\texttt{E} - \texttt{TailCons})$$

$$\frac{\texttt{t}_1 \rightarrow \texttt{t}_1'}{\texttt{tail [T] t}_1 \rightarrow \texttt{tail [T] t}_1'} \quad (\texttt{E} - \texttt{Tail})$$

Por último, añadiremos las siguientes reglas de tipado:

$$\Gamma \vdash \texttt{nil[T}_1\texttt{]} : \texttt{List T}_1 \quad (\texttt{T} - \texttt{Nil})$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1 \qquad \Gamma \vdash \texttt{t}_2 : \texttt{List T}_1}{\Gamma \vdash \texttt{cons[T}_1\texttt{] t}_1\ \texttt{t}_2 : \texttt{List T}_1} \quad (\texttt{T} - \texttt{Cons})$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{List T}_1}{\Gamma \vdash \texttt{isnil[T}_1\texttt{] t}_1 : \texttt{Bool}} \quad (\texttt{T} - \texttt{IsNil})$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{List T}_1}{\Gamma \vdash \texttt{head[T}_1\texttt{] t}_1 : \texttt{T}_1} \quad (\texttt{T} - \texttt{Head})$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{List T}_1}{\Gamma \vdash \texttt{tail[T}_1\texttt{] t}_1 : \texttt{List T}_1} \quad (\texttt{T} - \texttt{Tail})$$

- **Ejercicio:** Cumple nuestra definición de listas el teorema de Progreso?