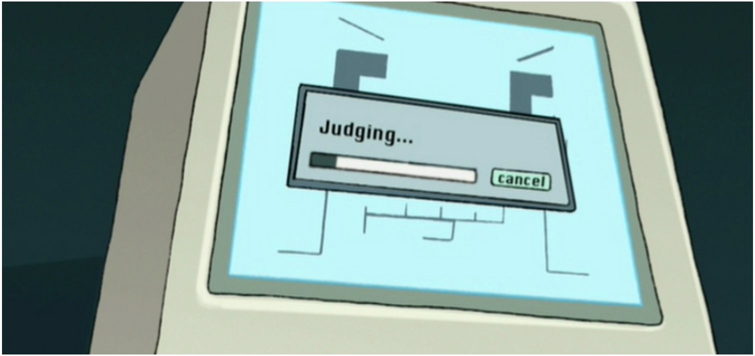
Sistemas de Tipos

Mauro Jaskelioff - 22/09/2017



Sistemas de Tipos

- Los sistemas de tipos son una forma de **método formal liviano**.
  - Automáticos,
  - pero, en gral, con un poder expresivo limitado.
- Un sistema de tipos es *"un método sintáctico para probar la ausencia de ciertos comportamientos mediante la clasificación de frases de acuerdo a los valores que computan"*.

Acerca de Sistemas de Tipos

- Los sistemas de tipos proveen un chequeo **estático** (i.e. en tiempo de compilación).
  - La idea es garantizar la ausencia de ciertos errores.
- Lo hacen **clasificando** términos de acuerdo a la **clase** de valor que computan.
  - Un sistema de tipos provee una **aproximación estática** al comportamiento de ejecución.

Ejemplo

- Si sabemos que dos fragmentos de programa exp_1 y exp_2 computan enteros (**clasificación**),
- podemos sumarlos en forma segura:

$$exp_1 + exp_2$$

- También sabemos que el resultado es un entero.
- No sabemos exactamente cuáles son los enteros involucrados, pero al menos sabemos que son enteros (**aproximación estática**)

“Tipado Dinámico”

- Los lenguajes con “tipado dinámico”, según nuestra definición, no tienen un sistema de tipos.
- Mas bien, deberían ser llamados lenguajes con **chequeo dinámico**.
- Por ejemplo, en un lenguaje con chequeo dinámico, $exp_1 + exp_2$ se evaluaría así:
  - evaluar exp_1 y exp_2
  - Sumar los resultados dependiendo del tipo de los resultados (suma entera, suma de punto flotante,...), o mostrar un error si la suma no es posible.

Conservatividad

- Los sistemas de tipos son necesariamente **conservadores**:
  - Algunos programas que se comportan correctamente serán rechazados.
- Por ejemplo,

  $$\textbf{if } \textit{test} \textbf{ then } S \textbf{ else } (\textit{error de tipo})$$

  es usualmente rechazado, aunque *test* sea siempre verdadero.
- En general, no se puede saber con antelación (estáticamente) el resultado de *test*.

Errores de run-time

- Un sistema de tipos excluye estáticamente **ciertos** errores de ejecución (**run-time errors**).
- Sin embargo, hay errores que son chequeados durante la ejecución.
- Por ejemplo, lo más usual es que un sistema de tipos:
  - **Verifique** que las operaciones aritméticas se hagan sobre números.
  - **No verifique** que el divisor no sea cero, o que el índice de un arreglo esté en rango.
- Hacer estas verificaciones estáticamente es posible con **tipos dependientes**.

Asignación de Tipos

- Inferencia de tipos
  - dada una expresión, determinar si tiene tipo o no, y cuál es ese tipo.
- Chequeo de tipos
  - dada una expresión e y un tipo A, determinar si $e : A$.
- Sistema de tipado fuerte
  - sistema que acepta una expresión si, y sólo si, ésta tiene tipo.

¿Para qué sirven los sistemas de tipos?

- Detectar errores
- Especificación rudimentaria. Documentación
- Abstracción
- Optimización
- Lenguaje seguros

¿Qué es un lenguaje seguro?

- No hay mucho consenso sobre esto.
- Según Pierce:

  > *Un lenguaje es seguro si protege sus abstracciones*

- Por ejemplo, varios lenguajes proveen la abstracción de arreglos, junto con operaciones para leerlo y modificarlo.
  - La abstracción no debería proveer maneras de escribir fuera del arreglo.
- Otro ejemplo: las variables locales deben accederse sólo en su alcance.
- Si un lenguaje no es seguro, es más difícil entender que hace un programa.

Seguridad y tipado estático

- ¡Lenguaje seguro no es igual a tipado estático!
- La seguridad puede ser obtenida tanto por tipado estático como por chequeos dinámicos (en tiempo de ejecución).
- Por ej: Scheme es un lenguaje seguro con chequeo dinámico.
- Incluso los lenguajes tipados estáticamente suelen usar chequeos dinámicos.
  - Rango del índice de un arreglo
  - conversiones de tipos (por ej. Java)
  - división por cero
  - falla de pattern-matching

Ejemplos de Lenguajes (in)Seguros

	Estático	Dinámico
Seguro	ML, Haskell, Java	Lisp, Scheme, Python, Perl
Inseguro	C, C++	

- Lenguajes como C y C++ proveen un tipado estático débil, que no puede ofrecer ninguna garantía.
- La seguridad hace que los programas sean más portables y, en general más predecibles.
- La seguridad no es absoluta. Los lenguajes frecuentemente proveen puertas de escape para poder interactuar con código escrito en un lenguaje inseguro.

Semántica estática y dinámica

- Un sistema de tipos **estáticamente** prueba propiedades acerca del comportamiento **dinámico** de los programas.
- Para poder precisar estas propiedades y **probar** que el sistema de tipos cumple con lo prometido es necesario formalizar:
  - **La semántica estática**
  - **La semántica dinámica**

Ejemplo: Expresiones Aritméticas

Recordemos la sintaxis abstracta de nuestro lenguaje de expresiones aritméticas:

$$
\begin{aligned}
t ::= {} & \mathit{true} \\
\mid {} & \mathit{false} \\
\mid {} & \textbf{if } t \textbf{ then } t \textbf{ else } t \\
\mid {} & 0 \\
\mid {} & \mathit{succ}\ t \\
\mid {} & \mathit{pred}\ t \\
\mid {} & \mathit{iszero}\ t
\end{aligned}
$$

Valores

Los valores son

$$\begin{array}{rcl} v & ::= & \mathit{true} \\ & \mid & \mathit{false} \\ & \mid & \mathit{nv} \end{array}$$

donde *nv* son los valores numéricos

$$\begin{array}{rcl} \mathit{nv} & ::= & 0 \\ & \mid & \mathit{succ}\ \mathit{nv} \end{array}$$

- Todos los valores son **formas normales**

Semántica Dinámica

- daremos la semántica dinámica **operacionalmente**
- Definimos la relación de evaluación $\rightarrow \subseteq \mathcal{T} \times \mathcal{T}$

$$\texttt{if true then } t_2 \texttt{ else } t_3 \rightarrow t_2 \ (\textsc{E-IfTrue})$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \rightarrow t_3 \ (\textsc{E-IfFalse})$$

$$\frac{t_1 \rightarrow {t_1}'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \rightarrow \texttt{if } {t_1}' \texttt{ then } t_2 \texttt{ else } t_3} \ (\textsc{E-If})$$

Semántica Dinámica (cont.)

$$\frac{t_1 \to t_1'}{\texttt{succ}\ t_1 \to \texttt{succ}\ t_1'} \quad \text{(E-Succ)}$$

$$\texttt{pred}\ 0 \to 0 \quad \text{(E-PredZero)}$$

$$\texttt{pred}\ (\texttt{succ}\ nv_1) \to nv_1 \quad \text{(E-PredSucc)}$$

$$\frac{t_1 \to t_1'}{\texttt{pred}\ t_1 \to \texttt{pred}\ t_1'} \quad \text{(E-Pred)}$$

$$\texttt{iszero}\ 0 \to \texttt{true} \quad \text{(E-isZeroZero)}$$

$$\texttt{iszero}\ (\texttt{succ}\ nv_1) \to \texttt{false} \quad \text{(E-IsZeroSucc)}$$

$$\frac{t_1 \to t_1'}{\texttt{iszero}\ t_1 \to \texttt{iszero}\ t_1'} \quad \text{(E-IsZero)}$$

Términos Atascados

- Los valores son formas normales,
  - `true`
  - $\texttt{succ}\ (\texttt{succ}\ 0)$
- pero **no todas las formas normales son valores**!
  - $\texttt{if}\ 0\ \texttt{then}\ \texttt{pred}\ 0\ \texttt{else}\ 0$
- Las formas normales que no son valores son los **términos atascados**.
  - Modelan errores de ejecución.
- El sistema de tipos debe eliminar **todos** los términos atascados, **garantizando** que los términos bien tipados nunca se atasquen.

Tipos

- Definimos la sintaxis de los tipos.
- El lenguaje sólo tiene dos tipos: booleanos y naturales

$$\begin{aligned} T ::= & \ \texttt{Bool} \\ \mid & \ \texttt{Nat} \end{aligned}$$

- La relación de tipado : relaciona términos con tipos.
- Un término t se dice bien tipado si existe T tal que $t : T$.
- La relación de tipado $t : T$ se define mediante reglas de inferencia.

Reglas de Tipado

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

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

$$\frac{t_1 : \texttt{Bool} \quad t_2 : T \quad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \qquad \text{(T-If)}$$

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

$$\frac{t_1 : \texttt{Nat}}{\texttt{succ } t_1 : \texttt{Nat}} \qquad \text{(T-Succ)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{pred } t_1 : \texttt{Nat}} \qquad \text{(T-Pred)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{iszero } t_1 : \texttt{Bool}} \qquad \text{(T-IsZero)}$$

Estilo: Curry vs. Church

- Nosotros definimos la semántica dinámica sobre todos los términos, y después definimos la semántica estática.
  - Se dice que la semántica es al "estilo de Curry"
- Otra alternativa es empezar por la semántica estática, y sólo después considerar la semántica dinámica sobre los términos bien tipados.
  - Este enfoque es lo que se conoce como "estilo de Church".

Derivación de Tipado

- Una derivación de tipado es un árbol de instancias de las reglas de tipado que muestra que un término tiene un determinado tipo.
- Ejemplo:

$$\dfrac{\dfrac{\dfrac{}{0 : \texttt{Nat}}\ \textsc{T-Zero}}{\texttt{iszero}\ 0 : \texttt{Bool}}\ \textsc{T-IsZero} \qquad \dfrac{}{0 : \texttt{Nat}}\ \textsc{T-Zero} \qquad \dfrac{\dfrac{}{0 : \texttt{Nat}}\ \textsc{T-Zero}}{\texttt{pred}\ 0 : \texttt{Nat}}\ \textsc{T-Pred}}{\texttt{if iszero}\ 0\ \texttt{then}\ 0\ \texttt{else pred}\ 0 : \texttt{Nat}}\ \textsc{T-If}$$

- Ejercicio: Dar el árbol de derivación de

$$\texttt{succ}\ (\texttt{if true then succ}\ 0\ \texttt{else}\ 0) : \texttt{Nat}$$

Seguridad = Progreso + Preservación

- La **seguridad** de un lenguaje es a veces caracerizada por el *slogan*.

  Los programas bien tipados no se rompen

  donde "romperse" significa atascarse.
- En general, para probar seguridad se divide la prueba en:
  - **Progreso:** Un término bien tipado no está atascado (es un valor o puede dar un paso.)
  - **Preservación:** Si un término bien tipado hace un paso de evaluación, entonces el resultado está bien tipado (Subject Reduction.)
- Progreso + preservación aseguran que ningún término bien tipado se atasque durante la evaluación.

Formas canónicas

- Una **forma canónica** es una forma normal cerrada.
- Antes de probar Progreso es conveniente analizar las formas de los términos de cada tipo.
- 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 valor numérico.
- Prueba: Análisis de casos sobre los valores.

Probando Progreso y Preservación

Teorema (Progreso) Sea t un término bien tipado (es decir, existe T tal que $t : T$). Entonces t es un valor, o bien existe t' tal que $t \to t'$.

Prueba: Por inducción sobre la derivación de $t : T$.

Teorema (Preservación) Si $t : T$ y $t \to t'$, entonces $t' : T$.

Prueba: Por inducción sobre la derivación de $t : T$.

Fragmento de la prueba de Progreso

Analicemos el caso del `if`. Las reglas relevantes son:

$$\frac{t_1 : \texttt{Bool} \quad t_2 : T \quad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \quad (\text{T-If})$$

$$\texttt{if true then } t_2 \texttt{ else } t_3 \to t_2 \quad (\text{E-IfTrue})$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \to t_3 \quad (\text{E-IfFalse})$$

$$\frac{t_1 \to t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \to \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \quad (\text{E-If})$$

Fragmento de la prueba de Progreso (cont.)

Prueba de Progreso por inducción en la derivación de $t : T$

Caso $\textsc{T-If}$: $t = \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$

$t_1 : \texttt{Bool},\ t_2 : T,\ t_3 : T$

- Por HI, t_1 es un valor, o bien existe $t_1{}'$ tal que $t_1 \rightarrow t_1{}'$.
- Si t_1 es un valor, entonces debe ser `true` o `false`, en cuyo caso puedo aplicar $\textsc{E-IfTrue}$ o $\textsc{E-IfFalse}$.
- Si existe $t_1{}'$ tal que $t_1 \rightarrow t_1{}'$, puedo aplicar $\textsc{E-If}$.

Fragmento de la prueba de Preservación

Prueba de Preservación por inducción en la derivación de $t : T$

Caso $\textsc{T-If}$: $t = \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$

$t_1 : \texttt{Bool},\ t_2 : T,\ t_3 : T$

- La evaluación de este término puede hacerse por $\textsc{E-IfTrue}$, $\textsc{E-IfFalse}$, o $\textsc{E-If}$.
- Si la evaluación es con $\textsc{E-IfTrue}$ o $\textsc{E-IfFalse}$, el resultado es t_2 o t_3.
  - Pero ambos tienen tipo T (al igual que t)
  - Por lo que el tipo se preserva.
- Si la evaluación es con $\textsc{E-If}$, sabemos que $t_1 \to {t_1}'$
  - Por HI, ${t_1}' : \texttt{Bool}$
  - Por $\textsc{T-If}$, podemos concluir $\texttt{if } {t_1}' \texttt{ then } t_2 \texttt{ else } t_3 : T$
  - Por lo tanto, el tipo se preserva.

Ejercicio: Progreso y Preservación

Probar que el lenguaje de expresiones aritméticas es seguro completando la prueba de progreso y preservación.

Extendiendo el lenguaje con `let`

- Extendemos el lenguaje con `lets`:

$$\begin{aligned} t ::= &\ \ldots \\ \mid &\ x \\ \mid &\ \texttt{let}\ x = t\ \texttt{in}\ t \end{aligned}$$

  donde x es la categoría sintáctica de los identificadores.

- Agregamos reglas de evaluación:

$$\texttt{let}\ x = v\ \texttt{in}\ t \rightarrow t\,[v\,/\,x] \qquad \text{(E-LetV)}$$

$$\frac{t_1 \rightarrow t_1{}'}{\texttt{let}\ x = t_1\ \texttt{in}\ t_2 \rightarrow \texttt{let}\ x = t_1{}'\ \texttt{in}\ t_2}\ \text{(E-Let)}$$

Tipando `let`

- Para poder escribir las reglas de tipado de `let`, necesitamos saber el tipo de los identificadores.
- Por lo tanto necesitamos un **contexto de tipado**.
- Un contexto de tipado es una asignación de tipos a identificadores.

$$\Gamma ::= x_1 : T_1, \quad \dots \quad , x_n : T_n$$

- La relación de tipado pasa a ser ternaria.

$$\Gamma \vdash t : T$$

- Significa "t tiene tipo T en el contexto Γ"

Notación en Contextos de Tipado

- El contexto vacío lo denotamos con $\emptyset$.
  - Pero escribimos $\vdash t : T$ en lugar de $\emptyset \vdash t : T$
- La extensión de un contexto:

  $$\Gamma, x : T$$

  asocia x al tipo T, y el resto de los identificadores al tipo asociado en Γ.
- Para decir que el tipo de una variable está en un contexto:

  $$x : T \in \Gamma \qquad \text{o también} \qquad \Gamma(x) = T$$

Reglas de Tipado (actualizadas)

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

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

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

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

$$\frac{\Gamma \vdash t_1 : \texttt{Nat}}{\Gamma \vdash \texttt{succ } t_1 : \texttt{Nat}} \qquad \text{(T-Succ)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Nat}}{\Gamma \vdash \texttt{pred } t_1 : \texttt{Nat}} \qquad \text{(T-Pred)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Nat}}{\Gamma \vdash \texttt{iszero } t_1 : \texttt{Bool}} \qquad \text{(T-IsZero)}$$

Reglas de Tipado (nuevas)

Agregamos las siguientes reglas de tipado

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-Var)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2} \qquad \text{(T-Let)}$$

Ejercicio: Probar que

$$\vdash \texttt{let}\ x = (\texttt{let}\ y = 0\ \texttt{in}\ y)\ \texttt{in}\ \texttt{succ}\ x : \texttt{Nat}$$

Extensión con Funciones

Extendemos el lenguaje con funciones

- Extendemos los términos

$$
\begin{aligned}
t ::=\ & \ldots \\
\mid\ & \lambda x \,.\, t \\
\mid\ & t\ t
\end{aligned}
$$

- Extendemos los valores

$$
\begin{aligned}
v ::=\ & \ldots \\
\mid\ & \lambda x \,.\, t
\end{aligned}
$$

- Extendemos los tipos

$$
\begin{aligned}
T ::=\ & \ldots \\
\mid\ & T \to T
\end{aligned}
$$

Reglas de evaluación

- Las nueva reglas de evaluación son:

$$\frac{t_1 \to t_1{}'}{t_1\ t_2 \to t_1{}'\ t_2} \qquad (\text{E-App1})$$

$$\frac{t_2 \to t_2{}'}{v\ t_2 \to v\ t_2{}'} \qquad (\text{E-App2})$$

$$(\lambda x\ .\ t)\ v \to t\ [v\ /\ x] \qquad (\text{E-AppAbs})$$

- La evaluación es de izquierda a derecha: 1ero la función (E-App1), luego el argumento (E-App2)
- La estrategia es **call-by-value**: el argumento es evaluado completamente antes de invocar la función (E-AppAbs)

Reglas de Tipado

Agregamos las siguientes reglas de tipado

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x\ .\ t_2 : T_1 \to T_2} \qquad (\textsc{T-Abs})$$

$$\frac{\Gamma \vdash t_2 : T_1 \to T_2 \quad \Gamma \vdash t_1 : T_1}{\Gamma \vdash t_2\ t_1 : T_2} \qquad (\textsc{T-App})$$

Ejercicio: Dado $\Gamma = \mathit{double} : \texttt{Nat} \to \texttt{Nat}$, derivar

$$\Gamma \vdash (\lambda f.\, f\ 0)\ \mathit{double} : \texttt{Nat}$$

Cálculo Lambda Simplemente Tipado

Si tomamos solamente el fragmento de funciones del lenguaje obtenemos el **cálculo lambda simplemente tipado** ($\lambda_\rightarrow$).

- Tipos. B es un conjunto de tipos base.

$$\begin{aligned} T ::= {} & B \\ \mid {} & T \rightarrow T \end{aligned}$$

- Términos. Los c son constantes.

$$\begin{aligned} t ::= {} & x \\ \mid {} & c \\ \mid {} & \lambda x.\, t \\ \mid {} & t\ t \end{aligned}$$

Reglas de Tipado

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x.\, t_2 : T_1 \rightarrow T_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_2 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_1 : T_1}{\Gamma \vdash t_2\ t_1 : T_2} \qquad \text{(T-APP)}$$

- Adicionalmente, se agregan reglas para las constantes.
- Notar que en realidad cuando uno habla de λ-cálculo simple tipado, en realidad está hablando de una **familia** de cálculos.

Comentarios

- Es necesario al menos **un** tipo base.
  - Si no, no sería posible construir tipos finitos.
- El cálculo lambda simplemente tipado es **fuertemente normalizante**: todos los términos bien tipados **siempre** reducen a un valor (para cualquier estrategia de reducción).
- La **auto aplicación** (Y o $(\lambda x.\, x\, x)$) no puede ser tipada.
- La recursión se podría recuperar agregando una constante $\mathit{fix} : (T \to T) \to T$ (ver el lenguaje PCF), tal que $\mathit{fix}\ t \to t\ (\mathit{fix}\ t)$.

Seguridad de $\lambda_{\rightarrow}$

- La seguridad del lambda cálculo se prueba en forma similar a lo ya vista.
- Es decir, mediante Progreso y Preservación.
- La mayor diferencia está en la prueba de Preservación.
- Se prueba con un **lema de substitución**.

$$\frac{\Gamma, x : S \vdash t : T \quad \Gamma \vdash s : S}{\Gamma \vdash t\,[s\,/\,x] : T}$$

- Estudiar las pruebas de progreso y preservación (sección 9.3 del Pierce).

Curry vs. Church

- Lo que vimos es el λ-cálculo simple tipado a la Curry.
- Un término $(\lambda x.\, x) : \alpha \to \alpha$ es en realidad un **esquema de términos**.
  - para cada tipo concreto T podemos instanciar α y obtener un término $(\lambda x.\, x) : T \to T$.
- Si definimos el cálculo a la Church, los términos tienen que tener un tipo definido.
- La abstracción es de la forma $\lambda x : T.\, t$.
- El λ-cálculo a la Church y a la Curry son equivalentes.
  - Hay una correspondencia uno a uno entre las derivaciones de tipo de los dos sistemas.

Sistema T de Gödel

- Extiende el λ-cálculo simple tipado con:
  - dos tipos base: `Bool` y `Nat`,
  - constantes `true`, `false`, `D`, 0, `succ`, y `R`.
- La constante `D` es esencialmente un `if then else`.
- La constante `R` es la **recursión primitiva** sobre los naturales.
- Sistema T puede representar todas las funciones cuya **totalidad** se puede probar usando lógica de primer orden.

Reglas adicionales de Sistema T

$$\Gamma \vdash \mathtt{true} : \mathtt{Bool} \qquad \text{(T-True)}$$

$$\Gamma \vdash \mathtt{false} : \mathtt{Bool} \qquad \text{(T-False)}$$

$$\frac{\Gamma \vdash t_1 : \mathtt{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \mathtt{D}\ t_1\ t_2\ t_3 : T} \qquad \text{(T-D)}$$

$$\Gamma \vdash 0 : \mathtt{Nat} \qquad \text{(T-Zero)}$$

$$\frac{\Gamma \vdash t_1 : \mathtt{Nat}}{\Gamma \vdash \mathtt{succ}\ t_1 : \mathtt{Nat}} \qquad \text{(T-Succ)}$$

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T \to \mathtt{Nat} \to T \quad \Gamma \vdash t_3 : \mathtt{Nat}}{\Gamma \vdash \mathtt{R}\ t_1\ t_2\ t_3 : T} \qquad \text{(T-Rec)}$$

Reglas adicionales de evaluación de Sistema T

$$\mathtt{D}\ \mathtt{true}\ t_2\ t_3 \to t_2 \qquad \text{(E-DTrue)}$$

$$\mathtt{D}\ \mathtt{false}\ t_2\ t_3 \to t_3 \qquad \text{(E-DFalse)}$$

$$\frac{t_1 \to t_1{}'}{\mathtt{D}\ t_1\ t_2\ t_3 \to \mathtt{D}\ t_1{}'\ t_2\ t_3} \qquad \text{(E-D)}$$

$$\mathtt{R}\ t_1\ t_2\ 0 \to t_1 \qquad \text{(E-RZero)}$$

$$\mathtt{R}\ t_1\ t_2\ (\mathtt{succ}\ t) \to t_2\ (\mathtt{R}\ t_1\ t_2\ t)\ t \qquad \text{(E-RSucc)}$$

$$\frac{t_3 \to t_3{}'}{\mathtt{R}\ t_1\ t_2\ t_3 \to \mathtt{R}\ t_1\ t_2\ t_3{}'} \qquad \text{(E-R)}$$

El operador $\mathtt{R}$

- $\mathtt{R}\ a\ b : \mathtt{Nat} \to T$ denota una función f definida por recursión primitiva tal que el caso base es a y el paso de recursión es b:

$$
\begin{aligned}
f\ 0 &= a \\
f\ (\mathtt{succ}\ n) &= b\ (f\ n)\ n
\end{aligned}
$$

- Notar la similaridad con un *fold*, excepto que en este caso el paso recursivo también tiene acceso a la entrada.
- Esto hace que sea más fácil definir operaciones como el predecesor.

Programando en Sistema T

- La función factorial

$$\begin{aligned} \texttt{fact}\ 0 &= 1 \\ \texttt{fact}\ (\texttt{succ}\ n) &= \texttt{mult}\ (\texttt{succ}\ n)\ (\texttt{fact}\ n) \end{aligned}$$

  se puede escribir en T, usando $\texttt{R}$:

$$\texttt{fact} = \texttt{R}\ 1\ (\lambda x\ y.\, \texttt{mult}\ (\texttt{succ}\ y)\ x)$$

- Ejercicio: Definir en T:
  1. $\texttt{pred} : \texttt{Nat} \to \texttt{Nat}$: predecesor,
  2. $\texttt{suma} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}$: suma de naturales,
  3. $\texttt{mult} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}$: multiplicación de naturales,
  4. $\mathit{is0} : \texttt{Nat} \to \texttt{Bool}$: comparación con 0,
  5. $\texttt{eq} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Bool}$: igualdad de naturales.

Resumen

- Los tipos son un método formal liviano automatizable.
- Los tipos tienen diversos usos.
  - Optimización, seguridad, verificación, abstracción, etc.
- Lenguajes seguros : progreso + preservación.
- Lambda cálculo simple tipado.
  - Plataforma para varios sistemas útiles, como Sistema T, PCF, etc.

Referencias

- *Types and Programming Languages*. B.C. Pierce (2002). Capítulos 1, 8, y 9.
- *Lambda-Calculus and Combinators*. J. R. Hindley and J. P. Seldin. Cambridge University Press (2008).
- *Theories of Programming Languages*. J. Reynolds (1998).
- *Proofs and Types*. Jean-Yves Girard, Yves Lafont and Paul Taylor (1987),