Análisis de Lenguajes de Programación

Sistemas de Tipos

30 de Septiembre de 2024

¿Cómo surgen?

- Los primeros sistemas (rudimentarios) aparecieron con los primeros lenguajes de programación, como Fortran y Cobol. Permitían detectar una variedad de errores en tiempo de compilación, pero tenían deficiencias.

- En 1970 surgen los algoritmos para inferencia de tipos para eliminar la información de tipos explícita. El algoritmo de **Hindley-Milner** fue usado por 1° vez para ML.

- Este algoritmo sigue siendo la base para los sistemas de tipos de lenguajes funcionales.

- Al agregar expresividad al sistema de tipos, la inferencia puede volverse indecidible.

¿Qué son?

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”.** (B. Pierce)

Además de detectar errores en una estapa temprana del desarrollo del software los sistemas de tipos ofrecen otras ventajas:

Sirven también para:

- Especificación rudimentaria o Documentación (siempre actualizada).
- Abstracción
- Optimización
- Lenguaje seguros

Chequeo estático

- Los sistemas de tipos proveen un **chequeo estático**, es decir en tiempo de compilación.
- Al ser estáticos, sólo se puede garantizar la ausencia de ciertos errores.
- Los sistemas de tipos son necesariamente **conservadores**, permiten probar la ausencia de determinados errores, pero no probar la presencia de errores. Por ej,

  ```
  if test then S else (error de tipo)
  ```

  es usualmente rechazado aunque test sea siempre verdadero.
- Los lenguajes con **tipos dependientes** permiten hacer algunas de estas verificaciones estáticamente.

Chequeo dinámico

- El chequeo del tipo de un programa es realizado en tiempo de **ejecución**.

  Por ejemplo, en una expresión de la forma $e_1 + e_2$, primero se evalúan las expresiones y dependiendo del tipo de éstas, se aplica la operación $+$.

- Algunos lenguajes incluyen un chequeo dinámico, incluso teniendo un sistema de tipos para el chequeo estático.

- Los lenguajes que incluyen un chequeo dinámico, pero no un sistema de tipos, se denominan **"Lenguajes con tipado dinámico"**.

¿Qué es un lenguaje seguro?

- Según Pierce: "Un lenguaje es seguro si protege sus abstracciones"
  Por ejemplo,
  - Si un lenguaje provee la abstracción de **arreglo**, para ser seguro debe garantizar que no hay manera de escribir fuera del arreglo.
  - Un lenguaje con variables locales, deben poder accerderse sólo bajo su alcance.
- Si un lenguaje no es seguro, es difícil entender qué hace un programa. Ejemplos: C, C++, donde el comportamiento de algunos programas que utilizan punteros no puede predecirse.

Seguridad

- ¡Lenguaje seguro no es igual a tipado estático!
- Formalizaremos una propiedad de seguridad básica: "Los términos tipados no causan errores".
- Las formas normales que no son valores se denominan **términos atascados**.
- El sistema de tipos debe garantizar que los términos bien tipados nunca se atasquen.

Seguridad= Progreso + Preservación

Progreso: Si $t : T$, entonces t es un valor o existe t' tal que $t \to t'$ (es decir, t no está atascado).

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

Ambas propiedades aseguran que ningún término bien tipado se atasca durante su evaluación.

Ejemplo

Sintaxis de lenguaje de expresiones aritméticas y booleanas:

$$
\begin{array}{lcl}
t & ::= & \mathit{true} \\
 & | & \mathit{false} \\
 & | & \mathit{if}\ t\ \mathit{then}\ t\ \mathit{else}\ t \\
 & | & 0 \\
 & | & \mathit{suc}\ t \\
 & | & \mathit{pred}\ t \\
 & | & \mathit{iszero}\ t
\end{array}
$$

$$v \quad ::= \mathit{true} \mid \mathit{false} \mid \mathit{nv}$$

$$\mathit{nv} \quad ::= 0 \mid \mathit{suc}\ \mathit{nv}$$

$$T \quad ::= \mathit{Bool} \mid \mathit{Nat}$$

Reglas de tipado

- Agregamos una relación de tipado $t : T$, se lee "t tiene tipo T".
- Decimos que t es tipado si existe T tal que $t : T$.
- La relación de tipado $t : T$ y se define como la menor relación que satisface las siguientes reglas:

$$\textit{true} : \textit{Bool}\ (\text{T-}\textsc{True}) \qquad \textit{false} : \textit{Bool}\ (\text{T-}\textsc{False})$$

Reglas de tipado

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

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

$$\frac{t : Nat}{pred\ t : Nat} \qquad (\text{T-PRED})$$

$$\frac{t : Nat}{suc\ t : Nat} \qquad (\text{T-SUC})$$

$$\frac{t : Nat}{iszero\ t : Bool} \qquad (\text{T-ISZERO})$$

Derivación de tipado

Mostraremos que un término tiene un tipo dado mediante una **derivación de tipado**.

Por ejemplo *if iszero* (*suc* 0) *then pred* (*suc* 0) *else* 0 : *Nat* dado que:

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

Reglas de semántica operacional

Definimos la semántica operacional del lenguaje mediante una relación $\rightarrow \in \mathcal{T} \times \mathcal{T}$, definida por las siguientes reglas:

$$\textit{if true then } t_2 \textit{ else } t_3 \rightarrow t_2 \qquad \text{(E-IFTRUE)}$$

$$\textit{if false then } t_2 \textit{ else } t_3 \rightarrow t_3 \qquad \text{(E-IFFALSE)}$$

$$\frac{t_1 \rightarrow t_1'}{\textit{if } t_1' \textit{ then } t_2 \textit{ else } t_3 \rightarrow t_3} \qquad \text{(E-IF)}$$

$$\frac{t \rightarrow t'}{\textit{suc } t \rightarrow \textit{suc } t'} \qquad \text{(E-SUC)}$$

Reglas de semántica operacional

$$\textit{pred } 0 \rightarrow 0 \qquad \text{(E-PREDZERO)}$$

$$\textit{pred } (\textit{suc nv}) \rightarrow \textit{nv} \qquad \text{(E-PREDSUC)}$$

$$\frac{t \rightarrow t'}{\textit{pred } t \rightarrow \textit{pred } t'} \qquad \text{(E-PRED)}$$

$$\textit{iszero } 0 \rightarrow \textit{true} \qquad \text{(E-ISZEROZERO)}$$

$$\textit{iszero } (\textit{suc nv}) \rightarrow \textit{false} \qquad \text{(E-ISZEROSUC)}$$

$$\frac{t \rightarrow t'}{\textit{iszero } t \rightarrow \textit{iszero } t'} \qquad \text{(E-ISZERO)}$$

Prueba de Progreso

Teorema (Progreso): Si $t : T$ entonces, t es un valor o existe t' tal que $t \to t'$.
La demostración es por inducción en la derivación $t : T$.
Caso T-IF Si la última regla aplicada en la derivación $t : T$ es T-IF entonces:

- $t = \textit{if } t_1 \textit{ then} t_2 \textit{ else } t_3$
- $t_1 : Bool$, $t2 : T$ y $t3 : T$

Dado que $t_1 : Bool$, por HI, t_1 es un valor o existe t_1' tal que $t_1 \to t_1'$. Si t_1 es un valor, éste es *true* o *false* y puedo aplicar E-IFTrue o E-IFFalse para encontar t'. Si existe t_1' tal que $t_1 \to t_1'$, puedo aplicar la regla E-If.

Cálculo lambda simplemente tipado ($\lambda_{\to}$)

Términos:

$t ::= x \mid c \mid t\,t \mid \lambda x\,.\,t$

Tipos:

$T ::= B \mid T \to T$

donde B es un conjunto de tipos bases y c un conjunto de constantes.

Definimos una familia de cálculos.

Reglas de tipado

- Al introducir variables al lenguaje, la relación de tipado deberá tener un elemento más: un contexto de tipado o entorno de tipo.
- Un entorno de tipos es una secuencia de variables con sus tipos.
- Algunas convenciones, sea Γ, un entorno de tipos:
  - El operador , extiende un entorno agregando una variable con su tipo. Por ej: $\mathbf{\Gamma, x : T_1}$
  - El entorno vacío se denota con $\emptyset$, o simplemente se omite.

Más convenciones

- Para evitar confusiones entre nombres de variables, se supone que al agregar una variable al entorno Γ, ésta no está en Γ, dado que los nombres de las variables ligadas pueden ser renombrados.
- Γ entonces puede pensarse también como una función de variables a tipos.
- Escribiremos $\mathbf{\Gamma(x) = T_1}$ o $\mathbf{x : T_1 \in \Gamma}$ para decir que la variable x tiene tipo T_1 en Γ.

Reglas de tipado

La relación de tipado es ternaria $\Gamma \vdash t : T$, significa que el término t tiene tipo T en el entorno Γ.

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

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

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

Faltan reglas para las constantes.

Ejercicios

Determinar el tipo de los siguientes términos dando el árbol de derivación siendo $\Gamma = f : (B \to B) \to B \to B, x : B$:

1. $(\lambda\, y\, .\, y)\, x$
2. $(\lambda\, g\, .\, f\, g)\, x$

Comentarios

- Algunos términos del λ-cálculo sin tipo no son válidos en $\lambda_{\rightarrow}$, ya que no pueden tiparse, por ejemplo Y.

- Es imposible encontrar un tipo para cualquier operador de punto fijo.

- La recursión se puede recuperar agregando un operador **fix**, se agrega como constante.

  $$fix : (T \rightarrow T) \rightarrow T$$
  $$fix\ t = t\ (fix\ t)$$

- A diferencia de λ-cálculo sin tipos, $\lambda_{\rightarrow}$ es fuertemente normalizante, dado que cualquier término es fuertemente normalizante.

Estrategias de reducción

- La estrategia de reducción call-by-value, es **estricta**, dado que el argumento de una función se evalúa independientemente si se usa en el cuerpo de la función.
- Mientras que las estrategias call-by-name y call-by-need son no estrictas o *lazy*.
- Notar que el operador de punto fijo Y, no sirve con una estrategia de reducción *call-by-value, Yg diverge para cualquier* g.

Evaluación

Daremos una estrategia *call-by-value para este cálculo:*

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

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

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

donde $v ::= c \mid \lambda \, x \, . \, t$

E-App1 y E-App2 son reglas de congruencia, mientras que E-AppAbs es de computación.

Seguridad de $\lambda_{\rightarrow}$

- $\lambda_{\rightarrow}$ satisface los teoremas de **Progreso** y **Preservación**.
- Para probar Preservación se utiliza el siguiente lema de sustitución:

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

Notación explícita de tipo

- El cálculo que vimos es el $\lambda_{\rightarrow}$ a la Currry, donde por ejemplo el término $\lambda\, x\, .\, x$ es un esquema de términos.
- La notación que agrega explícitamente el tipo a las variables se denomina a la Church.
  Las abstracciones son de la forma $\lambda\, x : T\ .\ t$
- Los cálculos $\lambda_{\rightarrow}$ a la Currry y la Church son equivalente.

Sistema T de Gödel

- El sistema T extiende $\lambda_{\rightarrow}$ con:

  1. Los tipos Bool y Nat.
  2. Constantes para naturales: **0** y **succ** (constructores), **R** (eliminador).
  3. Constantes para booleano: **true** y **falso** (constructores), **D** (eliminador).

- La constante **D** es la operación if-then-else.

- La constante **R** es la recursión primitiva para naturales.

Reglas de tipado adicionales

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

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

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

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

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

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

Reglas de evaluación adicionales

$$D\ \mathit{true}\ t_2\ t_3 \rightarrow t_2 \qquad \text{(E-DTrue)}$$

$$D\ \mathit{false}\ t_2\ t_3 \rightarrow t_3 \qquad \text{(E-DFalse)}$$

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

$$R\ t_1\ t_2\ 0 \rightarrow t_1 \qquad \text{(E-R0)}$$

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

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

Programando en Sistema T

- Factorial en Haskell:

```
fact 0 = 1
fact (n+1) = (n +1) * fact n
```

- Factorial en Sistema T

$$fact = R\ 1\ (\lambda\ r\ m\ .\ prod\ (succ\ m)\ r)$$

Ejercicios

Definir en Sistema T las siguientes funciones:

1. $pred \ : \ Nat \to Nat$
2. $suma \ : \ Nat \to Nat \to Nat$
3. $prod \ : \ Nat \to Nat \to Nat$
4. $isO \ : \ Nat \to Bool$
5. $eq \ : \ Nat \to Nat \to Bool$