Análisis de Lenguajes de Programación

Polimorfismo

30 de Septiembre de 2024

Repaso

- Vimos que $\lambda_{\to}$ es fuertemente normalizante. Es decir, que al agregar tipos al λ-cálculo obtenemos un cálculo donde todos los términos tienen reducción finita.

- Primero definimos una familia de cálculos cuyos términos son:

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

- Luego, extendimos este cálculo con tipos de datos recursivos. (Sistema T)

- Extenderemos $\lambda_{\to}$ con funciones polimórficas.

Polimorfismo

El polimorfismo permite que un programa pueda ser utilizado con diferentes tipos en contextos diferentes.

Existen distintas variantes:

- Polimorfismo ad-hoc: Una función polimórfica denota un conjunto de definiciones distintas, cada una soporta un tipo de datos diferente.
  Generalmente se implementa con la sobrecarga de operadores.
  Por ejemplo en Haskell:

```
(+) :: Num a => a -> a -> a
```

  Este tipo de polimorfismo es común en lenguajes orientados a objetos.

Variantes de polimorfismo

- Polimorfismo paramétrico: Permite que el tipo de un programa contenga variables, las cuales son instanciadas con tipos específicos cuando es necesario.
  Se tiene una sola definición de la función. Por ejemplo:

```
id :: a -> a
length :: [a] -> Int
fst :: (a, b) -> a
map :: (a -> b) -> [a] -> [b]
```

  En estos ejemplos los $\forall$ están implícitos.

  Existen variantes.

Tipos de polimorfismo paramétrico

- Polimorfismo de 1° clase: Las ligaduras de tipo pueden ocurrir en cualquier nivel. No admite inferencia de tipos.

- Polimorfismo Let o ML-style: Es más restrictivo, las ligaduras de tipo sólo pueden ocurrir en el nivel superior, como consecuencia no se pueden pasar funciones polimórficas como argumento. Admite inferencia de tipos.

  En Haskell, eliminamos esta restricción con la extensión:

  ```
  {-# LANGUAGE RankNTypes #-}
  ```

  Pero es necesario agregar anotaciones de tipo.

Cálculo Lambda Polimórfimo o Sistema F

- Extiende el *lambda*$_{\rightarrow}$ con polimorfismo paramétrico de 1° clase.
- El **sistema F** fue descubierto pot Girard (1972) e independientemente Reynolds (1974) definió un sistema con la misma expresividad y lo llamó **lambda cálculo polimórfico**.
- Agrega a los términos de $\lambda_{\rightarrow}$ abstracciones y aplicaciones de tipo.

Sintaxis

Se extienden los términos, valores y tipos de $\lambda_{\to}$:

$$t ::= \ldots \mid \Lambda\, X\, .\, t \mid t\, \langle X \rangle$$

$$v ::= \ldots \mid \Lambda\, X\, .\, t$$

$$T ::= \ldots \mid X \mid \forall\, X\, .\, T$$

Sistema de Tipos

Se agregan las siguientes reglas de tipado, donde el entorno Γ contiene variables de términos y de tipos:

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda\, X\, .\, t : \forall\, X\, .\, T} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash t_1 : \forall\, X\, .\, T}{\Gamma \vdash t_1\, \langle T_2 \rangle : T\,[T_2/X]} \qquad \text{(T-TApp)}$$

Semántica

Se agregan las siguientes reglas de evaluación:

$$\frac{t_1 \to t_1'}{t_1\langle T\rangle \to t_1'\langle T\rangle} \qquad \text{(E-TApp)}$$

$$(\Lambda X\,.\,t)\langle T\rangle \to t\,[T/X] \qquad \text{(E-TAppAbs)}$$

Ejemplos

$$id \ : \ \forall \, X \, . \, X \to X$$
$$id \equiv \Lambda \, X \, . \ \ \lambda \, x \ : \ X \, . \, x$$

$$double \ : \ \forall \, X \, . \, (X \to X) \to X \to X$$
$$double \equiv \Lambda \, X \, . \, \lambda \, f : X \to X \, . \, \lambda \, a : X \, . \, f \, (f \ a)$$

Evaluación:

$$id \ \langle Nat \rangle \ 0 \ \to \ (\lambda \, x \ : \ Nat \, . \, x) \ 0 \ \to \ 0$$

Representación de tipos de datos

- Con este cálculo podremos tipar las codificaciones de Church del λ-cálculo sin tipos para booleanos, números, listas, etc.

- Por lo tanto, no se necesitarán las constantes para tipos bases (por ej, Nat y Bool).

Representación de booleanos

1. Partimos de la representación de Church de los booleanos:

   $true \equiv \lambda\ t\ f\ .\ t$

   $false \equiv \lambda\ t\ f\ .\ f$

2. Definimos un tipo común para *true* y *false*:

   $CBool \equiv \forall\ X\ .\ X \to\to X \to X$

3. Definimos las constantes tipadas *true* y *false* en Sistema F:

   $true \equiv \Lambda\ X\ .\ \lambda\ t : X\ .\ \lambda\ f : X\ .\ t$

   $false \equiv \Lambda\ X\ .\ \lambda\ t : X\ .\ .\ \lambda\ f : X\ .\ f$

Funciones booleanas

$$not : CBool \to CBool$$
$$not \equiv \lambda\, b : CBool\ .\ \Lambda X\ .\ \lambda t : X\ .\ \lambda f : X\ .\ b\,\langle X\rangle\, f\ t$$

o también podemos definirla como:

$$not : CBool \to CBool$$
$$not \equiv \lambda\, b : CBool\ .\ b\,\langle CBool\rangle\ false\ true$$

Ejercicio: Dar una definición *and* en Sistema F.

Representación de naturales

1. Codificación de naturales de Church:

   $\underline{0} \equiv \lambda\, f\ x\ .\ x$
   $\underline{1} \equiv \lambda\, f\ x\ .\ f\ x$
   $\underline{2} \equiv \lambda\, f\ x\ .\ f\ (f\ x)$
   $\ldots$

2. Definimos un tipo para ellos:

   $CNat \equiv \forall\, X\ .\ (X \to X) \to X \to X$

3. Naturales en Sistema F:

   $0 : CNat$
   $0 \equiv \Lambda\, X\ .\ \lambda\, f : X\ .\ \lambda\, x : X\ .x$
   $suc : CNat \to CNat$
   $suc \equiv \lambda\, n : CNat\ .\ \Lambda\, X\ .\ \lambda f : X\ .\ \lambda\, x : X\ .\ s\ (n\ \langle X \rangle\ f\ x)$

Representación de listas

1. Codificación de listas de Church:

$$nil \equiv \lambda\ c\ .\lambda\ n\ .\ n$$
$$cons \equiv \lambda\ x\ xs\ .\ \lambda\ c\ n\ .\ c\ x\ (xs\ c\ n)$$

2. Tipo para listas:

$$CList\ X \equiv \forall\ R\ .\ (X \to R \to R) \to R \to R$$

3. Listas en Sistema F:

$$nil : \forall\ X\ .\ CList\ X$$
$$nil \equiv \Lambda\ X\ .\ \Lambda\ R\ .\ \lambda\ c : X \to R \to R\ .\ \lambda\ n : R\ .\ n$$
$$cons : \forall\ X\ .\ X \to CList\ X \to CList\ X$$
$$cons \equiv \Lambda\ X\ .\ \lambda\ x : X\ .\ \lambda\ xs : CList\ X\ .$$
$$\Lambda\ R\ .\ \lambda\ c : X \to R \to R\ .\ \lambda\ n : R\ .\ c\ x\ (xs\langle R\rangle c\ n)$$

Función length

En Haskell:

```
length : [a] -> Int
length = foldr (\x r -> r +1) 0
```

En Sistema F:

$$length : \forall\, X\ .\ CList\ X \to CNat$$
$$length = \Lambda\, X\ .\ \lambda xs : CListF\ X\ .$$
$$xs\ \langle CNat \rangle (\lambda\, x : X\ .\ \lambda\, r : CNat\ .\ suc\ r)\ 0$$

Ejercicio: Definir en sistema F *null*, que determina si una lista es vacía.

Probar las definiciones en Haskell

Para implementar las representaciones de Sistema F en Haskell usaremos algunas extensiones:

```
{-# LANGUAGE RankNTypes #-}
```

Permite cuantificaciones de tipo en cualquier nivel.

```
{-# LANGUAGE TypeApplications #-}
```

Permite aplicaciones de tipo **visibles**. Por ej: `read @Int "5"`

```
{-# LANGUAGE ImpredicativeTypes #-}
```

Sin esta extensión sólo poderemos instanciar una función polimórfica con tipos monomórficos.

```
{-# LANGUAGE ScopedTypeVariables #-}
```

Permite que las variables del tipo polimórfico estén al alcance de la función.

Propiedades de Sistema F

- Preservación: Si $\Gamma \vdash t : T$ y $t \to t'$, entonces $\Gamma \vdash t' : T$.
- Progreso: Si t es un término **cerrado** bien tipado, entonces t es un valor o existe t' tal que $t \to t'$.
- Es fuertemente normalizante.

Isomorfismo de Curry-Howard

Un poco de historia..

- En 1934 Curry observa que los axiomas del cálculo proposicional se correspondían con los tipos de los combinadores básicos: S, K e I.

- En 1965 Tait descubre una correspondencia entre *cut-elimination* y la β-reducción en el λ-cálculo.

- En 1969 Howard extiende las ideas de Curry a la lógica de predicados, concretamente hay una analogía entre la deducción natural y $\lambda_{\rightarrow}$.

Isomorfismo de Curry-Howard

Es una correspondencia entre la teoría de tipos y la lógica, que equipara:

- tipos con proposiciones
- términos con pruebas
- evaluación de términos con normalización de pruebas.

Deducción natural y $\lambda_{\to}$

Deducción natural

$$\Gamma_1, \alpha, \Gamma_2 \vdash \alpha$$

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta}$$

$$\frac{\Gamma \vdash \alpha \to \beta \quad \Gamma \vdash \alpha}{\Gamma \vdash \beta}$$

$\lambda_{\to}$

$$\Gamma_1, x : \alpha, \Gamma_2 \vdash x : \alpha$$

$$\frac{\Gamma, x : \alpha \vdash t : \beta}{\Gamma \vdash (\lambda x : \alpha . t) : \alpha \to \beta}$$

$$\frac{\Gamma \vdash t : \alpha \to \beta \quad \Gamma \vdash u : \alpha}{\Gamma \vdash t\ u : \beta}$$

Resultado del isomorfismo

- El isomorfismo permite interpretar el juicio de tipado $M : A$ de dos maneras:
  - M es un término (o programa) de tipo A o
  - M es la prueba (o derivación) de la fórmula A.

- Esta correspondencia nos permite probar que una fórmula de la lógica es un teorema encontrando un término cuyo tipo se corresponda con la fórmula asociada al tipo.

- Dió origen a muchos asistentes de prueba: Coq y Agda.