ECI 2014-T2
Demostración interactiva de teoremas: teoría y práctica

Guía de ejercicios

Beta Ziliani

August 1, 2014

1 El λ-cálculo polimórfico (System F de Reynolds)

La abstracción es uno de los principios básicos de la computación. Es lo que nos permite lidiar con un programa complejo, sin perder la cabeza entendiendolo. Ya vimos una forma de abstracción: la función anónima $\lambda x.\ t$. En esta sección nos detendremos a analizar los tipos *polimórficos*.

Para los que están ya familiarizados con leguajes de programación, los tipos polimórficos son como los template de C++, los generics de Java, o los tipos... ehm... ¡polimórficos! de Haskell (escritos en minúscula, como *a*).

Para motivarlo, tomemos la función identidad. Hasta ahora, estuvimos limitados a escribir el tipo de datos de la función, escribiendo por ejemplo

1. $\lambda x : \mathsf{nat}.x$
2. $\lambda x : \mathsf{bool}.x$
3. $\lambda x : \mathsf{nat} \to \mathsf{nat}.x$

Sin embargo, el tipo del argumento realmente no importa; la función es *paramétrica* en el tipo, puesto que el tipo no juega ningún rol. Queremos definir la función de forma tal que quede claro que la podemos aplicar a cualquier elemento de cualquier tipo.

Extendemos el λ-cálculo con dos nuevos tipos:

$$\sigma, \tau ::= \ldots \mid \alpha \mid \forall \alpha : \mathsf{Type},\ \tau$$

donde α es una *variable de tipo*. Necesitamos una nueva abstracción para tipos y una nueva aplicación:

$$M, N ::= \ldots \mid \lambda \alpha : \mathsf{Type}.\ M \mid M\ \tau$$

Volviendo a la función identidad, escribimos la versión polimórfica de la siguiente forma:

$$\mathsf{id} := \lambda \alpha : \mathsf{Type}.\ \lambda x : \alpha.x$$

Con esta definición podemos obtener la versiones que mencionamos antes:

1. id nat
2. id bool
3. id (nat → nat)

Cuál es el tipo de cada una de estos términos? El tipo varía de acuerdo al argumento que recibe la función, con lo que tenemos que definir la sustitución de tipos:

$$
\begin{array}{rll}
[\alpha := \tau]\alpha = \tau & & \\
[\alpha := \tau]\beta = \beta & & \alpha \neq \beta \\
[\alpha := \tau](\forall \beta : \mathsf{Type},\ \sigma) = \forall \beta' : \mathsf{Type},\ [\alpha := \tau][\beta := \beta']\sigma & & \beta' \text{ fresco} \\
[\alpha := \tau](\sigma \times \sigma') = ([\alpha := \tau]\sigma \times [\alpha := \tau]\sigma') & &
\end{array}
$$

(etc.)

Es fácil ver que para el juicio de tipado de la nueva aplicación tenemos que realizar una sustitución de tipos:

$$\frac{\Gamma \vdash \tau : \mathsf{Type} \qquad \Gamma \vdash M : \forall \alpha : \mathsf{Type},\ \sigma}{\Gamma \vdash M\ \tau : [\alpha := \tau]\sigma}\ \text{TTypeApp}$$

Note que necesitamos primero establecer que τ es un tipo válido ($\Gamma \vdash \tau : \mathsf{Type}$). Para entender el motivo de este requerimiento, tomemos por ejemplo el término $\lambda x : \alpha.x$ en el contexto vacío. De dónde sale α?

En el caso de variables de término, como x, nos preocupaba saber su tipo, y por ello recurrimos a utilizar un contexto. En el caso de las variables de término, como α, podemos decir que es irrelevante utilizar el contexto, puesto que no necesitamos saber su "tipo". Sin embargo, un buen motivo para introducir las variables (de tipo o de término) en el contexto es para poder distinguir entre aquellas variables que fueron declaradas y aquellas que no. Para utilizar una analogía con los lenguajes normales de programación, queremos evitar que un error de tipeo invalide nuestro programa, de la misma forma que el programa C

```
int main () {
  tipo_inexistente x;
  return x;
}
```

no es valido porque `tipo_inexistente` no fue declarado.

En la abstracción de tipo, entonces, empujamos la variable en el contexto:

$$\frac{\Gamma, \alpha : \mathsf{Type} \vdash M : \tau}{\Gamma \vdash \lambda\, \alpha : \mathsf{Type}.\ M : \forall \alpha : \mathsf{Type},\ \tau}\ \text{TTypeAbs}$$

Esto nos permite establecer cuando una variable de tipo es válida:

$$\frac{}{\Gamma_1, \alpha : \mathsf{Type}, \Gamma_2 \vdash \alpha : \mathsf{Type}}\ \text{TTypeVar}$$

Ahora, una variable de tipo también puede aparecer en otro tipo, como en el siguiente ejemplo:

$$\lambda\, \alpha : \mathsf{Type}.\ \lambda x : \alpha \to \alpha.x$$

con lo que tenemos que extender el juicio de tipado para todos los tipos, no solo las variables:

$$\frac{\Gamma \vdash \sigma : \mathsf{Type} \qquad \Gamma \vdash \tau : \mathsf{Type}}{\Gamma \vdash \sigma \to \tau : \mathsf{Type}}\ \text{TTYPEARR} \qquad \frac{\Gamma, \alpha \vdash \tau : \mathsf{Type}}{\Gamma \vdash \forall \alpha : \mathsf{Type},\ \tau : \mathsf{Type}}\ \text{TTYPEPROD}$$

(etc.)

Para concluir con el tipado, necesitamos modificar la regla para la abstracción de términos para asegurarnos que el tipo de le variable es válido:

$$\frac{\Gamma \vdash \sigma : \mathsf{Type} \qquad \Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma . M : \tau}\ \text{TLAM}$$

Extendemos el juicio de evaluación:

$$\begin{array}{rcll} M\ \tau & \rightsquigarrow & N\ \tau & \text{si } M \rightsquigarrow N \quad (\text{ETYPELEFT}) \\ \lambda \alpha : \mathsf{Type}.\ M\ \tau & \rightsquigarrow & [\alpha := \tau]M & (\beta\text{-TYPE}) \end{array}$$

Ejemplos:

- $(\lambda \alpha\ \beta : \mathsf{Type}.\ \lambda x : (\alpha \times \beta).\ \mathbf{fst}\ x)\ \mathsf{nat}\ \mathsf{unit}\ (\mathsf{true}, \mathsf{tt}) \rightsquigarrow$
  $\rightsquigarrow (\lambda \beta : \mathsf{Type}.\ \lambda x : (\mathsf{nat} \times \beta).\ \mathbf{fst}\ x)\ \mathsf{unit}\ (\mathsf{true}, \mathsf{tt}) \rightsquigarrow$
  $\rightsquigarrow (\lambda x : (\mathsf{nat} \times \mathsf{unit}).\ \mathbf{fst}\ x)\ (\mathsf{true}, \mathsf{tt}) \rightsquigarrow \mathbf{fst}\ (\mathsf{true}, \mathsf{tt}) \rightsquigarrow \mathsf{true}$

- Los tipos nat, $\times$, y $\times$ son redundantes en el λ-cálculo polimórfico, ya que se pueden emular con polimorfismo. Por ejemplo, los números naturales se pueden definir como:

$$\begin{aligned} \mathsf{nat} &\stackrel{\text{def}}{=} \forall \alpha : \mathsf{Type}, \alpha \to (\alpha \to \alpha) \to \alpha \\ 0 : \mathsf{nat} &\stackrel{\text{def}}{=} \lambda \alpha . \lambda x : \alpha . \lambda f : \alpha \to \alpha . x \\ \mathsf{S} : \mathsf{nat} \to \mathsf{nat} &\stackrel{\text{def}}{=} \lambda n : \mathsf{nat} . \lambda \alpha . \lambda x : \alpha . \lambda f : \alpha \to \alpha . f\ (n\ \alpha\ x\ f) \end{aligned}$$

  Al principio puede parecer raro, pero básicamente ésta definición hace que un número $n = \mathsf{S}\ (\mathsf{S}\ (\ldots (\mathsf{S}\ 0) \ldots))$ sea un *iterador* que dado un tipo τ, un término M de tipo τ, y una función f de tipo $\tau \to \tau$ retorne la aplicación n-aria de f a M, es decir $n\ \tau\ M\ f$ evalúa a $f\ (f\ (\ldots (f\ M) \ldots))$.

  La suma se define, por ejemplo, como

$$n + m \stackrel{\text{def}}{=} \forall \alpha\ x\ f . n\ X\ (m\ X\ x\ f)\ f$$

  La idea es que primero se itera m veces y luego n veces.

  (Ver mas ejemplos en numerals.v).

  Este lenguaje es realmente poderoso y permite construir todas las extensiones que mostramos en este curso. Pero esto es más un ejercicio intelectual que otra cosa, puesto que para un lenguaje de programación real necesitamos también eficiencia. Por ejemplo, la mayoría de los lenguajes funcionales compilan los números naturales en números de máquina. Este tipo de trato diferenciado según el tipo se pierde si codificamos todo en función de abstracciones de tipo, de valores, y de aplicaciones.

1.1 Ejercicios

Demuestre (mediante un árbol de derivación) cuáles de los siguientes términos están bien tipados, qué tipo tienen (o justifique porque no lo están). Evalúe aquellos términos bien tipados paso a paso (como en el primer ejemplo de arriba).

1. $(\lambda\alpha : \mathsf{Type}.\ \lambda x : \alpha.\lambda y : \mathsf{unit}.y)\ \mathsf{tt}\ \mathsf{unit}\ \mathsf{tt}$
2. $(\lambda\alpha : \mathsf{Type}.\ \lambda x : \alpha.\lambda\beta : \mathsf{Type}.\ \lambda y : \beta.y)\ \mathsf{unit}\ \mathsf{tt}\ \mathsf{unit}\ \mathsf{tt}$
3. $(\lambda x : \mathsf{unit}.\lambda\alpha : \mathsf{Type}.\ \lambda y : \alpha.x)\ \mathsf{tt}\ (\mathsf{unit} \to \mathsf{unit})\ (\lambda z : \mathsf{unit}.z)$

2 Ejercicios en Coq

Entregue los archivos con (al menos) los siguientes ejercicios:

1. **Induction:** double_plus, beq_nat_refl, plus_comm_informal.
2. **Lists:** list_funs, bag_functions, list_exercises, hd_opt, dictionary_invariant1, dictionary_invariant2.