Programación Funcional

Curso 2019-20

INFERENCIA DE TIPOS

Inferencia de tipos

No tenemos en cuenta clases de tipos

- Tipos simples

$$
\begin{array}{llll}
TS \ni \tau ::= & \alpha & & \textit{variables de tipo} \\
& | & b & \textit{tipos básicos Bool, Int,...} \\
& | & (\tau_1, \ldots, \tau_n) & \textit{tuplas} \\
& | & T\ \tau_1\ \ldots\ \tau_n & \textit{Tipo construido}^{(*)} \\
& | & \tau_1 \to \tau_2 & \textit{funciones}
\end{array}
$$

  (*): Incluye el caso de las listas: $[\tau]$

- Tipos genéricos (o 'esquemas de tipo')
  $TG \ni \sigma ::= \tau \mid \forall\alpha.\sigma$
  - Es decir: $\sigma \equiv \forall\alpha_1 \ldots \forall\alpha_n.\tau$
  - σ es *cerrado* si todas las variables de τ están en $\forall\alpha_1 \ldots \forall\alpha_n$
  - Si $\sigma \equiv \forall\alpha_1 \ldots \forall\alpha_n.\tau$, al tipo simple τ (posiblemente con un renombramiento de variables) le llamamos *instancia genérica* de σ.

Inferencia de tipos (II)

- Objetivo: dado un programa que define unas funciones $f_1, \ldots, f_n$, inferir los tipos genéricos más generales (*tipos principales*) para $f_1, \ldots, f_n$ que sean compatibles con sus definiciones (o descubrir que el programa está mal tipado).
- Suponemos ordenadas $f_1, \ldots, f_n$ de modo que la definición de cada f_i dependa solo de las anteriores $^{(*)}$. Inferimos los tipos en ese orden, de modo que al inferir el tipo de f_i ya disponemos del tipo de las funciones de las que depende.
  (*): Más en general, particionamos $f_1, \ldots, f_n$ en bloques de funciones $B_1, \ldots, B_k$ ordenados de modo que:
  - Las funciones de cada bloque B_i dependen mutuamente unas de otras (son mutuamente recursivas)
  - Cada bloque B_i depende solo de los bloques anteriores
  - Inferimos los tipos bloque a bloque

Inferencia de tipos (III)

Para cada función f (en general, para cada bloque), la inferencia de tipo se puede descomponer en las siguientes **fases**:

- **Decoración** con tipos de las reglas que definen f
- **Generación de restricciones** de tipo, que son ecuaciones entre tipos que definen lo que se llama un *problema de unificación*.
- **Resolución de las restricciones** de tipo (o sea, del problema de unificación).
- **Generalización** del tipo obtenido.

Decoración con tipos

Las constructoras de datos y algunos símbolos de función tienen ya tipos establecidos (o inferidos previamente), que denominamos *suposiciones de tipo (type assumptions)* para la inferencia en curso. Deben ser esquemas de tipo cerrados. Por ejemplo:

True:: Bool	*0::Int*
$[]::\forall\alpha.[\alpha]$	$(:)::\forall\alpha.\alpha \to [\alpha] \to [\alpha]$
$(\&\&)::Bool \to Bool \to Bool$	$(.)::\forall\alpha\beta\gamma.(\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta$

Decoración con tipos (II)

Todas las reglas $f\ t_1 \ldots t_n = e$ se decoran con tipos simples del siguiente modo:

- Cada símbolo se decora con:
  - una variable de tipo α *'fresca'* si el símbolo no tiene suposición previa.
  - una instancia genérica fresca de su tipo, en otro caso.
- Cada aplicación $(e\ e_1 \ldots e_n)$ se decora como $(e :: \tau\ e_1 :: \tau_1 \ldots e_n :: \tau_n) :: \alpha$, donde $\tau, \tau_1, \ldots, \tau_n$ son las decoraciones de $e, e_1, \ldots, e_n$.
- Cada tupla $(e1, \ldots, e_n)$ se decora como $(e_1 :: \tau_1, \ldots, e_n :: \tau_n) :: \alpha$, donde $\tau_1, \ldots, \tau_n$ son las decoraciones de $e_1, \ldots, e_n$
- Cada λ-abstracción $\lambda x.e$ se decora como $(\lambda x :: \alpha.e :: \tau) :: \beta$, donde τ es la decoración de e.

Las variables de tipo introducidas $\alpha, \beta, \ldots$ deben contemplarse como 'incógnitas' cuyo valor es descubierto en las siguientes fases.

Decoración con tipos: condiciones adicionales

- Al inferir el tipo de una función f todas las apariciones de f en las reglas que la definen han de decorarse con la misma variable de tipo. Esto se generaliza al caso de inferencia de bloques de funciones mutuamente recursivas.
- Todas las apariciones de las variables de un parámetro formal en su ámbito léxico (una misma regla, una misma λ-abstracción) han de decorarse con la misma variable de tipo. Pero si se repiten en otro ámbito (otra regla, otra λ-abstracción), se decoran con otra variable.
- Distintas apariciones de un símbolo con suposición de tipo (incluso en una misma regla) se decoran con instancias genéricas frescas.

Generación de restricciones de tipo (ecuaciones entre tipos)

A partir de las reglas (y sus subexpresiones) decoradas generamos una colección de ecuaciones entre tipos simples:

- Cada regla decorada $e :: \tau = e' :: \tau'$ genera la ecuación $\tau = \tau'$
- Cada aplicación $(e :: \tau \ e_1 :: \tau_1 \ldots e_n :: \tau_n) :: \tau'$ genera $\tau = \tau_1 \to \ldots \to \tau_n \to \tau'$
- Cada tupla $(e_1 :: \tau_1, \ldots, e_n :: \tau_n) :: \tau$ genera $\tau = (\tau_1, \ldots, \tau_n)$
- Cada λ-abstracción $(\lambda x :: \tau.e :: \tau') :: \tau''$ genera $\tau'' = \tau \to \tau'$

Resolución de las ecuaciones de tipos

- La colección de ecuaciones obtenida define lo que se denomina un *problema de unificación sintáctica*.

- Resolverlo consiste en hallar valores de las variables que conviertan a todas las ecuaciones en identidades sintácticas. Por ejemplo, la solución de $(\alpha, Bool) = (\beta, \alpha)$ vendrá dada por $\alpha = Bool, \beta = Bool$. A cada solución se le llama *unificador*.

- En general, puede haber más de un unificador. Por ejemplo, un unificador para $\alpha = \beta \to \gamma, [\beta] = [\gamma]$ viene dado por $\alpha = Int \to Int, \beta = Int, \gamma = Int$ y otro por $\alpha = Bool \to Bool, \beta = Bool, \gamma = Bool$. Nos interesa el *unificador más general (umg)*, que en el ejemplo es $\alpha = \beta \to \beta, \gamma = \beta$. Se demuestra que, si hay unificadores, hay con seguridad un umg.

- Hay muchos algoritmos para obtener umg's. Aquí presentamos el de *Martelli-Montanari*, que consiste en un proceso de transformación paso a paso del conjunto de ecuaciones hasta llegar a una *forma resuelta*.

Algoritmo de Martelli-Montanari

Dado un conjunto de ecuaciones $\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n$, aplicar reiteradamente y mientras se pueda las siguientes reglas (no importa el orden en el que se consideran las ecuaciones ni las reglas del algoritmo). La notación $E \vdash E'$ indica que el conjunto de ecuaciones E se transforma en E'. *Fallo* indica que no hay unificador.

- *(Eliminación)* $\tau = \tau, E \vdash E$
- *(Descomposición)*
  $\tau_1 \to \tau_2 = \tau'_1 \to \tau'_2, E \vdash \tau_1 = \tau'_1, \tau_2 = \tau'_2, E$
  $(\tau_1, \ldots, \tau_n) = (\tau'_1, \ldots, \tau'_n), E \vdash \tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n, E$
  $T\ \tau_1 \ldots \tau_n = T\ \tau'_1 \ldots \tau'_n, E \vdash \tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n, E$
- *(Ligadura)* $\alpha = \tau, E \vdash \alpha = \tau, E'$, siendo E' el resultado de sustituir α por τ en E, y suponiendo que α aparece en E, pero no en τ.
- *(Reorden)* $\tau = \alpha, E \vdash \alpha = \tau, E$, si τ no es una variable.
- *(Conflicto)* $\tau = \tau', E \vdash Fallo$, si ni τ ni τ' son variables y no son tipos con estructuras homólogas.
- *(Occur-check)* $\alpha = \tau, E \vdash Fallo$, si $\alpha \neq \tau$ pero α aparece en τ

Generalización del tipo

Esta fase consiste simplemente en hacer el cierre universal del tipo simple obtenido en la fase anterior. O sea, si el tipo simple para f dado por el unificador es τ y $\alpha_1, \ldots, \alpha_n$ son las variables de τ, entonces el esquema de tipo inferido finalmente para f es

$$\forall\alpha_1 \ldots \forall\alpha_n.\tau$$