Tipos (y Subtipado) para Lenguajes Orientados a Objetos

Introducción

- Desde mediados de los 80 han habido numerosos esfuerzos por realizar estudios rigurosos que analizan tipos para LOO
- Dos alternativas han sido exploradas
  - Codificar objetos en términos de lenguajes funcionales
    - Trabajo pionero de Cardelli en 1984
    - Utiliza funciones, registros, recursión y subtipado
  - Formulación de cálculos fundacionales (del estilo de Lambda Cálculo) para el paradigma orientado a objetos como por ejemplo
    - [Abadi y Cardelli, 1996] A Theory of Objects
    - [Castagna, 1997] Object-Oriented Programming: A Unified Foundation
    - [Bruce, 2002] Foundations of Object Oriented Languages

¿Qué es un error de tipos en un LOO?

- Además de los errores de tipos habituales como ser
  - métodos reciben número o tipo de parámetros incorrectos
  - asignaciones que no respetan el tipo declarado de las variables
- Hay un error de tipos característico que todo sistema de tipos para un LOO debe detectar
  - La invocación a métodos inexistentes
- Los sistemas de tipos actuales para LOO imponen restricciones severas para poder detectar estos errores de tipos

Tipos para LOO

- La noción de clase y de tipo se mantienen separadas
  - Clase: inherentemente de implementación (ej. variables de instancia privadas, código fuente de los métodos, etc.)
  - Tipo de un objeto: la interface pública del mismo
    - nombres de todos los métodos
    - tipo de los argumentos de cada método y tipo del resultado

  Especificación de un objeto (tipo) $\neq$ Implementación (clase)

- Esta separación beneficia el desarrollo modular de sistemas: varias clases pueden instanciar objetos con el mismo tipo
- El tipo de un objeto a veces se conoce como interface type

Ejemplo - Clase y su tipo

```
Object subclass: #Point
instanceVariableNames: 'x y'
xcord
 ...
ycord
 ...
dist: aPoint
 ...
```

Un objeto instancia de la clase `Point` tendría el siguiente tipo

```
PointType = {
  xcord: Unit -> Int;
  ycord: Unit -> Int;
  dist: PointType -> Int;
}
```

OBS: `xcord`,`ycord` y `dist` son las componentes del tipo

Tipos a partir de clases

- Como lo muestra el ejemplo podemos extraer de manera mecánica la información necesaria para construir los tipos de objetos a través de las declaraciones de clases
  - Esto aplica a lenguajes tipados estáticamente
- Notación: Si `C` es una clase usaremos `CType` para hacer referencia al tipo de los objetos extraídos de esa clase

Juicio de subtipado

$$\sigma <: \tau$$

- Lectura: "En todo contexto donde se espera una expresión de tipo τ, puede utilizarse una de tipo σ en su lugar sin que ello genere un error"
  - Ej. si `D` es subclase de `C`, entonces se espera que
    $$\texttt{DType} <: \texttt{CType}$$
- ¿Qué relación hay entre $\Gamma \triangleright M : \sigma$ y $\sigma <: \tau$?

Principio de sustitutividad

$$\sigma <: \tau$$

- Lectura: "En todo contexto donde se espera una expresión de tipo τ, puede utilizarse una de tipo σ en su lugar sin que ello genere un error"
- Lectura reflejada en la teoría como una nueva regla de tipado llamada Subsumption:

$$\frac{\Gamma \triangleright M : \sigma \quad \sigma <: \tau}{\Gamma \triangleright M : \tau} \text{(T-Subs)}$$

- Vamos a recordar el sistema de tipos para el lambda cálculo con registros

Tipado para LC con registros – Repaso

$$\frac{x : \sigma \in \Gamma}{\Gamma \triangleright x : \sigma} \text{(T-Var)}$$

$$\frac{\Gamma, x : \sigma \triangleright M : \tau}{\Gamma \triangleright \lambda x : \sigma . M : \sigma \to \tau} \text{(T-Abs)} \quad \frac{\Gamma \triangleright M : \sigma \to \tau \quad \Gamma \triangleright N : \sigma}{\Gamma \triangleright M\, N : \tau} \text{(T-App)}$$

$$\frac{\Gamma \triangleright M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \triangleright \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}} \text{(T-Rcd)}$$

$$\frac{\Gamma \triangleright M : \{l_i : \sigma_i \;^{i \in 1..n}\} \quad j \in 1..n}{\Gamma \triangleright M.l_j : \sigma_j} \text{(T-Proj)}$$

$$\frac{\Gamma \triangleright M : \sigma \quad \sigma <: \tau}{\Gamma \triangleright M : \tau} \text{(T-Subs)}$$

Subtipado de tipos base

- Para los tipos base asumimos que nos informan de qué manera están relacionados; por ejemplo

$$\begin{array}{rcl} Nat & <: & Float \\ Int & <: & Float \\ Bool & <: & Nat \end{array}$$

Subtipado como preorden

$$\frac{}{\sigma <: \sigma}\ \text{(S-Refl)} \qquad \frac{\sigma <: \tau \quad \tau <: \rho}{\sigma <: \rho}\ \text{(S-Trans)}$$

Nota:

- Sin antisimetría

Subtipado de registros a lo "ancho"

`{nombre: String, edad:Int} <: {nombre:String}`

La regla general es

$$\frac{}{\{l_i : \sigma_i | i \in 1..n+k\} <: \{l_i : \sigma_i | i \in 1..n\}} \text{ (S-RcdWidth)}$$

Nota:

- $\sigma <: \{\}$, para todo tipo registro σ
- ¿hay algún tipo registro τ tal que $\tau <: \sigma$, para todo tipo registro σ?

Otro ejemplo

```
Object subclass: #Point
instanceVariableNames: 'x y'
xcord
 ...
ycord
 ...
dist: aPoint
 ...
```

```
Point subclass: #ColorPoint
instanceVariableNames: 'color'
colorcord
 ...
```

Vemos que `ColorPointType<:PointType` donde

```
PointType = {
  xcord: Unit -> Int;
  ycord: Unit -> Int;
  dist: PointType -> Int;
}
```

```
ColorPointType = {
  xcord: Unit -> Int;
  ycord: Unit -> Int;
  dist: PointType -> Int;
  colorcord: Unit -> Int;
}
```

Digresión: Tipado Nominal à la Java

- Nuestro enfoque (subtipado estructural):
  - Asociar a cada clase C un registro CType
  - Determinar si CType<:DType en base a la estructura de los registros
- Enfoque de Java (subtipado nominal):
  - Asocia a cada clase C un símbolo #C
  - Declarar como nuevos axiomas de subtipado:

    $$\#C <: \#D$$

    siempre que `class C extends D` aparece en nuestro programa

Limitaciones de subtipado a lo ancho – Shallow clone

- Operación que permite hacer una copia o clon de un objeto
- Especialmente en lenguajes en los que todos los objetos se representan como referencias y la operación de asignación no hace más que copiar referencias
- Shallow cloning (la otra es deep cloning; no será usada en nuestro ejemplo):
  - Copiar los valores de las variables de instancia y tomar el mismo conjunto de métodos que el original
  - Si las variables de instancia tiene referencias a otros objetos sólo las referencias se copian (y no los objetos referenciados)
- Llamaremos `clone` (clase `Object`) a esta operación

Ejemplo de limitación - Shallow clone

```
Object
 clone
  ...
```

```
Cell subclass: #Object
 clone
  ...
```

- ¿Cuál es el tipo de `clone`?
  - En la clase `Object` debe retornar un valor de tipo `ObjectType`
  - Si `Cell` es una subclase de `Object`, uno querría que el método `clone` heredado por `Cell` sea `CellType`
  - ¡En sistemas de tipos invariantes, `clone` debe tener tipo `Object`, aún si el método retorna un valor de tipo `CellType`!

```
ObjectType = {
  clone: Unit -> ObjectType;
}
```

```
CellType = {
  clone: Unit -> ObjectType;
  ...
}
```

- El programador se verá forzado a hacer un typecast para permitir al sistema tratar el valor como si tuviera el tipo que debería tener

Ejemplo de limitación - Shallow clone

```
ObjectType = {
  clone: Unit -> ObjectType;
}
```

```
CellType = {
  clone: Unit -> ObjectType;
  m: Unit -> Int;
  ...
}
```

- Si `m` es un método de la clase `Cell` y `o` es una variable de tipo `CellType`, la expresión

  `(o clone()) m()`

  genera un error de tipos
- El programador debe insertar un typecast como en

  `[CellType](o clone()) m()`

  para que funcione como se espera

Typecasts

- Los typecasts son una manera de ayudar al sistema de tipos
- Hay dos tipos de typecast
  - "up cast": `[CType]e` donde `e` tiene tipo `DType` y `D` es una subclase de `C`
  - "down cast": `[DType]e` donde `e` tiene tipo `CType` y `D` es una subclase de `C`
- El up cast casi no se usa, mientras que el down cast se usa mucho (para permitir recuperar el tipo "real" de un objeto)

Nota: La necesidad de recurrir a typecasts es una señal de las limitaciones del sistema de tipos

¿Podemos evitar el cast? Subtipado en profundidad

```
Object
clone
 ...
```

```
Cell subclass: #Object
clone
 ...
```

Sería deseable que `CellType<:ObjectType` donde

```
ObjectType = {
  clone: Unit -> ObjectType;
}
```

```
CellType = {
  clone: Unit -> CellType;
}
```

Subtipado de registros en "profundidad"

Si fuese

```
Alumno <:  :Persona
```

Es de esperarse

```
{a: Alumno, b:Int} <: {a:Persona, b:Int}
```

La regla general es

$$\frac{\sigma_i <: \tau_i \quad i \in I = \{1..n\}}{\{l_i : \sigma_i\}_{i \in I} <: \{l_i : \tau_i\}_{i \in I}} \text{(S-RcdDepth)}$$

Ejemplos

$$\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}, y : \{\}\}$$

En efecto:

$$\dfrac{\dfrac{}{\{a : Nat, b : Nat\} <: \{a : Nat\}}\ \text{(S-RcdWidth)} \qquad \dfrac{}{\{m : Nat\} <: \{\}}\ \text{(S-RcdWidth)}}{\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}, y : \{\}\}}\ \text{(S-RcdDepth)}$$

Ejemplos

Utilizando (S-Refl) se puede subtipar sólo un campo:

$$\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}, y : \{m : Nat\}\}$$

En efecto:

$$\dfrac{\dfrac{}{a : Nat, b : Nat <: a : Nat}\,(\text{S-RcdWidth}) \qquad \dfrac{}{m : Nat <: \{m : Nat\}}\,(\text{S-Refl})}{\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}, y : \{m : Nat\}\}}\,(\text{S-RcdDepth})$$

Permutaciones de campos

- Los registros son descripciones de campos y no deberían depender del orden dado

$$\frac{\{k_j : \sigma_j | j \in 1..n\} \text{ es permutación de } \{l_i : \tau_i | i \in 1..n\}}{\{k_j : \sigma_j | j \in 1..n\} <: \{l_i : \tau_i | i \in 1..n\}} \text{ (S-RcdPerm)}$$

Nota:

- (S-RcdPerm) puede usarse en combinación con (S-RcdWidth) y (S-Trans) para eliminar campos en cualquier parte del registro

Combinando width, depth y permutation subtyping

$$\frac{\{l_i |\ i \in 1..n\} \subseteq \{k_j |\ j \in 1..m\} \qquad k_j = l_i \Rightarrow \sigma_j <: \tau_i}{\{k_j : \sigma_j |\ j \in 1..m\} <: \{l_i : \tau_i |\ i \in 1..n\}} \text{ (S-Rcd)}$$

Subtipado de tipos función

$$\frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \to \tau <: \sigma' \to \tau'} \text{ (S-Func)}$$

- Observar que el sentido de $<:$ se da "vuelta" para el tipo del argumento de la función pero no para el tipo del resultado
- Se dice que el constructor de tipos función es contravariante en su primer argumento y variante en el segundo.

Por ejemplo:

$$\mathit{Unit} \to \mathit{CellType} <: \mathit{Unit} \to \mathit{ObjectType}$$

Subtipado de tipos función

$$\frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \to \tau <: \sigma' \to \tau'} \text{ (S-Func)}$$

Si un contexto/programa P espera una expresión $f' : \sigma' \to \tau'$ puede recibir otra $f : \sigma \to \tau$ si se dan las condiciones indicadas. En efecto:

- Toda aplicación de f' se hace a argumentos de tipo σ'
- Los mismos se coercionan a argumentos de tipo σ para poder aplicar f
- Luego se aplica f, cuyo tipo real es $\sigma \to \tau$
- Finalmente se coerciona el resultado a τ', el tipo del resultado que P está esperando

Por ejemplo:

$$\mathit{Unit} \to \mathit{CellType} <: \mathit{Unit} \to \mathit{ObjectType}$$

El tipo *Top* – Tipo máximo

Puede verse como representando la clase Object en Smalltalk

$$\frac{}{\sigma <: Top} \text{(S-Top)}$$

- Notar que $Top \to Top <: Top$

Subtipando colecciones

List ¿es covariante? ¿Es contravariante?

$$\frac{\sigma <: \tau}{\mathit{List}\ \sigma <: \mathit{List}\ \tau}$$

Es covariante (en la mayoría de los lenguajes)

Subtipado de referencias

¿Covariante? Imaginemos esta regla:

$$\frac{\sigma <: \tau}{\mathit{Ref}\ \sigma <: \mathit{Ref}\ \tau}$$

¿Qué ocurre?

Ref no es covariante

```
letval r = ref 3 (*r:Ref int*)
 in
  r := 2.1; (*usando Ref int <: Ref float =>T-sub r: Ref float*)
  !r
end::int
```

¡Pero 2.1 no es int!

$$\frac{\sigma <: \tau}{\mathit{Ref}\,\sigma <: \mathit{Ref}\,\tau} \qquad \frac{\mathit{int} <: \mathit{float}}{\mathit{Ref}\ \mathit{int} <: \mathit{Ref}\ \mathit{float}}$$

¿Ref contravariante?

¿Contravariante? Imaginemos esta regla:

$$\frac{\sigma <: \tau}{\mathit{Ref}\,\tau <: \mathit{Ref}\,\sigma}$$

Otra vez, ¿qué ocurre?

Ref no es contravariante

```
letval r = ref 2.1 (*r:Ref float*)
 in
   !r (* por Ref float <: Ref int =>T-sub r: Ref int *)
 end :: int
```

pero 2.1 no es int!!!

$$\frac{\sigma <: \tau}{\mathit{Ref}\,\tau <: \mathit{Ref}\,\sigma} \qquad \frac{\mathit{int} <: \mathit{float}}{\mathit{Ref}\;\; \mathit{float} <: \mathit{Ref}\;\; \mathit{int}}$$

Ref es invariante

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\mathit{Ref}\,\sigma <: \mathit{Ref}\,\tau}$$

DEF. σ es equivalente a τ ssi $\sigma <: \tau$ y $\tau <: \sigma$
"Sólo se comparan referencias de tipos equivalentes."

Reglas de tipado como especificación de un algoritmo

- Las reglas de tipado sin subtipado son dirigidas por sintaxis.
- Ello hace que sea inmediato implementar un algoritmo de chequeo de tipos a partir de ellas.

$$\frac{x : \sigma \in \Gamma}{\Gamma \rhd x : \sigma}\ (\text{T-Var})$$

$$\frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd \lambda x : \sigma.M : \sigma \to \tau}\ (\text{T-Abs}) \quad \frac{\Gamma \rhd M : \sigma \to \tau \quad \Gamma \rhd N : \sigma}{\Gamma \rhd M\, N : \tau}\ (\text{T-App})$$

$$\frac{\Gamma \rhd M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \rhd \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}}\ (\text{T-Rcd})$$

$$\frac{\Gamma \rhd M : \{l_i : \sigma_i \ ^{i \in 1..n}\} \quad j \in 1..n}{\Gamma \rhd M.l_j : \sigma_j}\ (\text{T-Proj})$$

Agregando subsumption

- Con subsumption ya no son dirigidas por sintaxis.
- No es evidente cómo implementar un algoritmo de chequeo de tipos a partir de las reglas.

$$\frac{x : \sigma \in \Gamma}{\Gamma \rhd x : \sigma}\text{(T-Var)} \qquad \frac{\Gamma \rhd M : \sigma \quad \sigma <: \tau}{\Gamma \rhd M : \tau}\text{(T-Subs)}$$

$$\frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd \lambda x : \sigma . M : \sigma \to \tau}\text{(T-Abs)} \quad \frac{\Gamma \rhd M : \sigma \to \tau \quad \Gamma \rhd N : \sigma}{\Gamma \rhd M\, N : \tau}\text{(T-App)}$$

$$\frac{\Gamma \rhd M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \rhd \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}}\text{(T-Rcd)}$$

$$\frac{\Gamma \rhd M : \{l_i : \sigma_i{}^{\,i \in 1..n}\} \quad j \in 1..n}{\Gamma \rhd M.l_j : \sigma_j}\text{(T-Proj)}$$

“Cableando” subsumption dentro de las demás reglas

- Un análisis ciudadoso determina que el único lugar donde se precisa subtipar es al aplicar una función a un argumento
- Esto sugiere la siguiente formulación:

$$\frac{x : \sigma \in \Gamma}{\Gamma \mapsto x : \sigma}\,(\text{T-Var}) \qquad \frac{\Gamma, x : \sigma \mapsto M : \tau}{\Gamma \mapsto \lambda x : \sigma.M : \sigma \to \tau}\,(\text{T-Abs})$$

$$\frac{\Gamma \mapsto M : \sigma \to \tau \quad \Gamma \mapsto N : \rho \quad \rho <: \sigma}{\Gamma \mapsto M\, N : \tau}\,(\text{T-App})$$

$$\frac{\Gamma \mapsto M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \mapsto \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}}\,(\text{T-Rcd})$$

$$\frac{\Gamma \mapsto M : \{l_i : \sigma_i{}^{\,i \in 1..n}\} \quad j \in 1..n}{\Gamma \mapsto M.l_j : \sigma_j}\,(\text{T-Proj})$$

Variante dirigida por sintaxis

- Vamos a posponer la discusión de hasta qué punto es más fácil de implementar esta variante
- Antes: ¿Qué relación tiene con la formulación original?

Proposición:

1. $\Gamma \mapsto M : \sigma$ implica que $\Gamma \triangleright M : \sigma$
2. $\Gamma \triangleright M : \sigma$ implica que existe τ tal que $\Gamma \mapsto M : \tau$ con $\tau <: \sigma$

Hacia una implementación de chequeo de tipos

- Lo único que faltaría cubrir es de qué manera se implementa la relación $\sigma <: \tau$

$$\frac{x : \sigma \in \Gamma}{\Gamma \mapsto x : \sigma} \text{(T-Var)} \qquad \frac{\Gamma, x : \sigma \mapsto M : \tau}{\Gamma \mapsto \lambda x : \sigma . M : \sigma \to \tau} \text{(T-Abs)}$$

$$\frac{\Gamma \mapsto M : \sigma \to \tau \quad \Gamma \mapsto N : \rho \quad \rho <: \sigma}{\Gamma \mapsto M\, N : \tau} \text{(T-App)}$$

$$\frac{\Gamma \mapsto M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \mapsto \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}} \text{(T-Rcd)}$$

$$\frac{\Gamma \mapsto M : \{l_i : \sigma_i \ ^{i \in 1..n}\} \quad j \in 1..n}{\Gamma \mapsto M.l_j : \sigma_j} \text{(T-Proj)}$$

Reglas de subtipado – Recordatorio

$$\frac{}{\sigma <: \sigma}\text{ (S-Refl)} \qquad \frac{}{\sigma <: Top}\text{ (S-Top)}$$

$$\frac{}{Nat <: Float}\text{ (S-NatFloat)} \frac{}{Int <: Float}\text{ (S-IntFloat)} \frac{}{Bool <: Nat}\text{ (S-BoolNat)}$$

$$\frac{\sigma <: \tau \quad \tau <: \rho}{\sigma <: \rho}\text{ (S-Trans)} \qquad \frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \to \tau <: \sigma' \to \tau'}\text{ (S-Func)}$$

$$\frac{\{l_i \mid i \in 1..n\} \subseteq \{k_j \mid j \in 1..m\} \qquad k_j = l_i \Rightarrow \sigma_j <: \tau_i}{\{k_j : \sigma_j \mid j \in 1..m\} <: \{l_i : \tau_i \mid i \in 1..n\}}\text{ (S-Rcd)}$$

- No son dirigidas por sintaxis...
- El problema es (S-Refl) y (S-Trans)

Deshaciéndonos de (S-Refl) y (S-Trans)

- Observando que se puede probar $\sigma{<:}\sigma$ y la transitividad, siempre que se tenga reflexividad para los tipos escalares:
  - $Nat{<:}Nat$
  - $Int{<:}Int$
  - $Bool{<:}Bool$
  - $Float{<:}Float$
- Agregamos estas cuatro y no consideramos explícitamente a las reglas (S-Refl) y (S-Trans)

El algoritmo de chequeo de subtipos (obviando los axiomas de Nat, Int, Bool, Float)

subtype(S, T) =
  if T==Top
    then true
    else
     if S==$S1 \to S2$ and T==$T1 \to T2$
       then subtype($T1, S1$) and subtype($S2, T2$)
       else
        if S==$\{kj : Sj, j \in 1..m\}$ and T==$\{li : Ti, i \in 1..n\}$
          then $\{li, i \in 1..n\} \subseteq \{kj, j \in 1..m\}$ and
            $\forall\ i\ \exists\ j\ kj = li$ and subtype(Sj, Ti)
          else false

Lectura adicional

- A Theory of Objects, Martín Abadi, Luca Cardelli, Monographs in Computer Science, Springer-Verlag, 1996.
- Foundations of Object Oriented Languages, Kim Bruce, MIT Press, 2002.
- Some Challenging Typing Issues in Object-Oriented Languages, Kim Bruce. Electronic Notes in Theoretical Computer Science 82, no. 8 (2003). (disponible en su página web).
- On binary methods, Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. Theory and Practice of Object Systems, 1(1995).
- Types and Programming Languages, Benjamin C. Pierce, The MIT Press, 2002.