

Subtipado

15 febrero de 2018

Por qué subtipado

- ▶ El sistema de tipos que estudiamos descarta programas incorrectos.
- ▶ Pero también programas “buenos”.
 - ▶ $(\lambda x : \{a : \text{Nat}\}.x.a)\{a = 1, b = \text{true}\}$
- ▶ Queremos mayor flexibilidad y disminuir la cantidad de programas buenos que se descartan.

Principio de substitutividad

$$\sigma <: \tau$$

- ▶ 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”
- ▶ Esto se refleja con una nueva regla de tipado llamada **Subsumption**:

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

Subtipado de tipos base

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

Nat <: *Float*

Int <: *Float*

Bool <: *Nat*

Subtipado como preorden

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

Nota:

- Sin antisimetría, ni simetría

Tipado para LC con registros – Repaso

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

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

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

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

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

Subtipado de registros a lo “ancho”

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

La regla general es

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

Nota:

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

Subtipado de registros en “profundidad”

$\{a: \text{Nat}, b: \text{Int}\} <: \{a: \text{Float}\}$

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

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

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

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

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

Subtipado de tipos función

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

- 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.

Subtipado de tipos función

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

Si un contexto/programa P espera una expresión f de tipo $\sigma' \rightarrow \tau'$ puede recibir otra de tipo $\sigma \rightarrow \tau$ si dan las condiciones indicadas

- ▶ Toda aplicación de f se hace sobre un argumento de tipo σ'
- ▶ El argumento se coerciona al tipo σ
- ▶ Luego se aplica la función, cuyo tipo real $\sigma \rightarrow \tau$
- ▶ Finalmente se coerciona el resultado a τ' , el tipo del resultado que espera P

El tipo *Top* – Tipo máximo

- ▶ *Top* es el tipo universal (el tipo que tienen todos los elementos del dominio).
- ▶ En lenguajes orientados a objetos, es el Tipo asociado a la clase “Object”.
- ▶ Llamado supertipo universal, porque todo tipo es subtipo de *Top*.

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

Subtipado de referencias

¿Covariante? Imaginemos esta regla:

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

¿Qué ocurre?

Ref no es covariante

```
let r = ref 3
in
r := 2.1;
!r
```

```
let r = ref 3 (*r:Ref Int*)
in
r := 2.1;
!r
```

```
let r = ref 3 (*r:Ref Int*)
in
r := 2.1; (*usando Ref Int <: Ref Float => T-sub r:Ref Float*)
!r
::Int
```

¡Pero 2.1 no es int!

$$\frac{\sigma <: \tau}{\text{Ref } \sigma <: \text{Ref } \tau} \quad \frac{\text{Int} <: \text{Float}}{\text{Ref Int} <: \text{Ref Float}}$$

¿Ref contravariante?

¿Contravariante? Imaginemos esta regla:

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

Otra vez, ¿qué ocurre?

Ref no es contravariante

```
let r = ref 2.1 (*r:Ref Float*)  
in  
!r (* por Ref Float <: Ref Int =>T-sub r: ref int *)  
:: Int
```

pero 2.1 no es Int!!!

$$\frac{\sigma <: \tau}{\text{Ref } \tau <: \text{Ref } \sigma} \quad \frac{\text{Int} <: \text{Float}}{\text{Ref Float} <: \text{Ref Int}}$$

Ref es invariante

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\text{Ref } \sigma <: \text{Ref } \tau}$$

“Sólo se comparan referencias de tipos equivalentes.”

Refinando el tipo Ref

- ▶ Además de $Ref\ \sigma$ (que representa lecto-escritura), se separan las referencias en dos clases
- ▶ *Source* σ de lectura
- ▶ *Sink* σ de escritura
- ▶

$$\frac{\Gamma|\Sigma \triangleright M : Source\ \sigma}{\Gamma\Sigma \triangleright !M : \sigma}$$

$$\frac{\Gamma|\Sigma \triangleright M : Sink\ \sigma \quad \Gamma|\Sigma \triangleright N : \sigma}{\Gamma|\Sigma \triangleright M := N : Unit}$$

Ejemplo de uso de Source

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

$$\frac{\text{Int} <: \text{Float}}{\text{Source Int} <: \text{Source Float}}$$

!r puede verse como Float aunque r sea
Source Int.

```
let r = ref 3
in
  !r (* por Source Int <: Source Float *)
  :: Float
```

“Si espero leer de una ref a T, puedo esperar una ref a un tipo más bajo, más informativo”

Ejemplo de uso de Sink

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

$$\frac{\text{Int} <: \text{Float}}{\text{Sink Float} <: \text{Sink Int}}$$

```
let r = ref 2.1
in
  r := 3; (*usando Sink Float <: Sink Int*)
  !r
  ::Float
```

“Si espero escribir sobre una Ref a T, puedo esperar una Ref a un tipo más alto, menos informativo”

Relacionando con Ref

Toda vez que va Source (o Sink), puede ir Ref:

$$\frac{}{Ref\ \tau <: Source\ \tau} \text{ (S-RefSource)}$$

$$\frac{}{Ref\ \tau <: Sink\ \tau} \text{ (S-RefSink)}$$

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.

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

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

$$\frac{\Gamma \triangleright M : \sigma \rightarrow \tau \quad \Gamma \triangleright N : \sigma}{\Gamma \triangleright M N : \tau} \text{ (T-App)}$$

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

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

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.

$$\begin{array}{c} \frac{x : \sigma \in \Gamma}{\Gamma \triangleright x : \sigma} \text{ (T-Var)} \qquad \frac{\Gamma \triangleright M : \sigma \quad \sigma <: \tau}{\Gamma \triangleright M : \tau} \text{ (T-Subs)} \\[2ex] \frac{\Gamma, x : \sigma \triangleright M : \tau}{\Gamma \triangleright \lambda x : \sigma. M : \sigma \rightarrow \tau} \text{ (T-Abs)} \qquad \frac{\Gamma \triangleright M : \sigma \rightarrow \tau \quad \Gamma \triangleright N : \sigma}{\Gamma \triangleright M N : \tau} \text{ (T-App)} \\[2ex] \frac{\Gamma \triangleright M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \triangleright \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}} \text{ (T-Rcd)} \\[2ex] \frac{\Gamma \triangleright M : \{l_i : \sigma_i\}_{i \in 1..n} \quad j \in 1..n}{\Gamma \triangleright M.l_j : \sigma_j} \text{ (T-Proj)} \end{array}$$

“Cableando” subsumption dentro de las demás reglas

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

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

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

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

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

Variante dirigida por sintaxis

- ¿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 $\sigma <: \tau$

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

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

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

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

Reglas de subtipado – Recordatorio

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

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

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

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

- ▶ Son dirigidas por sintáxis? **No.**
- ▶ El problema está en (S-Refl) y (S-Trans)

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

- ▶ Se puede **probar** que $\sigma <: \sigma$ se puede derivar siempre que se tenga reflexividad para los tipos escalares:
 - ▶ $Nat <: Nat$
 - ▶ $Bool <: Bool$
 - ▶ $Float <: Float$
- ▶ Agregamos estos tres axiomas y no consideramos explícitamente a la regla. (S-Refl).

Deshaciéndonos de (S-Trans)

- ▶ Se puede **probar** la transitividad
- ▶ Es decir, no hace falta tenerla como una regla explícita

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

```
subtype( $S, T$ ) =  
  if  $T == \text{Top}$   
    then true  
  else  
    if  $S == S1 \rightarrow S2$  and  $T == T1 \rightarrow T2$   
      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
```