Verificación de Tipos

CI4721 – Lenguajes de Programación II

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Verificación con Polimorfismo Paramétrico

Aplicando las técnicas conocidas hasta ahora

Construiremos un Esquema Dirigido por Sintaxis para completar la verificación de tipos en presencia de polimorfismo paramétrico.

UNIVERSIDAD SIMÓN BOLÍVAR

Verificación con Polimorfismo Paramétrico

Aplicando las técnicas conocidas hasta ahora

Construiremos un Esquema Dirigido por Sintaxis para completar la verificación de tipos en presencia de polimorfismo paramétrico.

- Representaremos expresiones de tipos con grafos.
  - `mkleaf` – construye las hojas del grafo.
    - Comportamiento *singleton* **obligatorio** para variables de tipo.
    - Comportamiento *singleton* opcional para el resto de las hojas.
  - `mknode` – construye los nodos interiores.

UNIVERSIDAD SIMÓN BOLÍVAR

Verificación con Polimorfismo Paramétrico

Aplicando las técnicas conocidas hasta ahora

Construiremos un Esquema Dirigido por Sintaxis para completar la verificación de tipos en presencia de polimorfismo paramétrico.

- Representaremos expresiones de tipos con grafos.
  - `mkleaf` – construye las hojas del grafo.
    - Comportamiento *singleton* **obligatorio** para variables de tipo.
    - Comportamiento *singleton* opcional para el resto de las hojas.
  - `mknode` – construye los nodos interiores.
- `unify(m,n)` – unifica expresiones representadas por los nodos `m` y `n`.
  - Corresponde al algoritmo discutido en la clase anterior.
  - Tiene como efecto de borde recordar la sustitución – la calcula simultáneamente con la construcción del grafo.

UNIVERSIDAD SIMÓN BOLÍVAR

Verificación con Polimorfismo Paramétrico

Gestión de las variables de tipos

- Una variable de tipo puede aparecer en una definición de tipos – `mkleaf` se encarga de ese caso al procesar la definición.

UNIVERSIDAD SIMÓN BOLÍVAR

Verificación con Polimorfismo Paramétrico

Gestión de las variables de tipos

- Una variable de tipo puede aparecer en una definición de tipos – `mkleaf` se encarga de ese caso al procesar la definición.
- Cuando se procesa una *aplicación* funcional, es necesario crear variables de tipo para dominio y rango, y asociarla como tipo – `newtypevar()` es el "generador" de variables nuevas.

UNIVERSIDAD SIMÓN BOLÍVAR

Verificación con Polimorfismo Paramétrico

Gestión de las variables de tipos

- Una variable de tipo puede aparecer en una definición de tipos – `mkleaf` se encarga de ese caso al procesar la definición.
- Cuando se procesa una *aplicación* funcional, es necesario crear variables de tipo para dominio y rango, y asociarla como tipo – `newtypevar()` es el "generador" de variables nuevas.
- `fresh(t)` permitirá usar el tipo asociado a un identificador.
  - Construye una *copia* de la expresión `t`.
  - Si es una variable o expresión de tipos no cuantificada, simplemente retorna la copia sin cambios adicionales.
  - Si es una expresión de tipos cuantificada, retorna la copia después de sustituir todas las variables ligadas por nuevas.

UNIVERSIDAD SIMÓN BOLÍVAR

Esquema de Verificación Dirigido por Sintaxis

Solamente los casos interesantes

$E \rightarrow E_1\ (\ E_2\ )$

$E \rightarrow E_1\ ,\ E_2$
$E \rightarrow \textbf{id}$

- Las declaraciones usan `mkleaf` y `mknode` de forma apropiada para construir los grafos – ejercicio para el lector.

UNIVERSIDAD SIMÓN BOLÍVAR

Esquema de Verificación Dirigido por Sintaxis

Solamente los casos interesantes

$E \rightarrow E_1\ (\ E_2\ )$

$E \rightarrow E_1\ ,\ E_2$

$E \rightarrow \mathbf{id} \qquad \{\ E.type \leftarrow fresh(\mathbf{id}.type)\ \}$

- Las declaraciones usan `mkleaf` y `mknode` de forma apropiada para construir los grafos – ejercicio para el lector.
- **id**.*type* apunta a su expresión de tipos – copiarla y, de ser necesario, instanciarla antes de usarla.

UNIVERSIDAD SIMÓN BOLÍVAR

Esquema de Verificación Dirigido por Sintaxis

Solamente los casos interesantes

$$E \rightarrow E_1 \ (\ E_2\ )$$

$$E \rightarrow E_1\ ,\ E_2 \qquad \{\ E.type \leftarrow mknode('\times', E_1.type, E_2.type)\}$$

$$E \rightarrow \mathbf{id} \qquad \{\ E.type \leftarrow fresh(\mathbf{id}.type)\ \}$$

- Las declaraciones usan `mkleaf` y `mknode` de forma apropiada para construir los grafos – ejercicio para el lector.
- **id**.*type* apunta a su expresión de tipos – copiarla y, de ser necesario, instanciarla antes de usarla.
- Múltiples argumentos son manejados como tipos producto.

UNIVERSIDAD SIMÓN BOLÍVAR

Esquema de Verificación Dirigido por Sintaxis

Solamente los casos interesantes

$$
\begin{array}{ll}
E \rightarrow E_1\ (\ E_2\ ) & \left\{ \begin{array}{l} r \leftarrow mkleaf(newtypevar()) \\ unify(E_1.type, mknode('->', E_2.type, r) \\ E.type \leftarrow r \end{array} \right\} \\
E \rightarrow E_1\ ,\ E_2 & \{\ E.type \leftarrow mknode('\times', E_1.type, E_2.type)\} \\
E \rightarrow \mathbf{id} & \{\ E.type \leftarrow fresh(\mathbf{id}.type)\ \}
\end{array}
$$

- Las declaraciones usan `mkleaf` y `mknode` de forma apropiada para construir los grafos – ejercicio para el lector.
- **id**.*type* apunta a su expresión de tipos – copiarla y, de ser necesario, instanciarla antes de usarla.
- Múltiples argumentos son manejados como tipos producto.
- Unificaciones aparecen con cada aplicación funcional.

UNIVERSIDAD SIMÓN BOLÍVAR

Una gramática para tipos estructurados

$D \rightarrow T$ **id** ; D
$D \rightarrow \lambda$
$T \rightarrow B\ C$
$T \rightarrow$ **record** { D }
$B \rightarrow$ **int**
$B \rightarrow$ **float**
$C \rightarrow C$ [**num**]
$C \rightarrow \lambda$

- D genera listas definiciones – principal o dentro de un registro.
- C genera componentes de arreglos – base en cero.

UNIVERSIDAD SIMÓN BOLÍVAR

Espacio para identificadores locales

A procedimientos o bloques

- El tipo para un identificador permite al compilador:
  - Determinar cuánto espacio requiere a tiempo de ejecución.
  - Determinar su posición relativa (*offset*) en su bloque de alcance.
  - Incorporar esa información en la tabla de símbolos – será aprovechada por el generador de código.

UNIVERSIDAD SIMÓN BOLÍVAR

Espacio para identificadores locales

A procedimientos o bloques

- El tipo para un identificador permite al compilador:
  - Determinar cuánto espacio requiere a tiempo de ejecución.
  - Determinar su posición relativa (*offset*) en su bloque de alcance.
  - Incorporar esa información en la tabla de símbolos – será aprovechada por el generador de código.
- Para datos de longitud variable (cadenas, arreglos dinámicos)
  - La información definitiva solamente está disponible a tiempo de ejecución.
  - El compilador reserva espacio para un *apuntador* a un espacio de tamaño variable en el cual se almacenará:
    - La información de dimensión del objeto (*dope vector*) – longitud de la cadena o la forma del arreglo.
    - Los datos propiamente dichos – esto es lo que lo hace variable.

A tiempo de compilación *siempre* se puede calcular un espacio y desplazamiento para el dato o para la referencia al dato.

Espacio para nombres locales

Paramerización independiente de la plataforma

- La máquina destino condiciona el almacenamiento.
  - Algunas operaciones de máquina para tipos primitivos requieren una **alineación** particular – definirla como un atributo de los tipos.
  - Dejar espacio sin usar (*padding*) entre elementos de un tipo estructurado para aprovechar esas condiciones.

UNIVERSIDAD SIMÓN BOLÍVAR

Espacio para nombres locales

Paramerización independiente de la plataforma

- La máquina destino condiciona el almacenamiento.
  - Algunas operaciones de máquina para tipos primitivos requieren una **alineación** particular – definirla como un atributo de los tipos.
  - Dejar espacio sin usar (*padding*) entre elementos de un tipo estructurado para aprovechar esas condiciones.
- La **anchura** de un tipo es la cantidad de unidades de almacenamiento *contiguo* necesarias para contenerlo.
  - Tipos primitivos – balancear condiciones de plataforma y diseño.
  - Arreglos – espacio contiguo para el total de elementos.
  - Registros – espacio contiguo para el total de miembros, pero con *padding* para alinearlos maximizando eficiencia de acceso.
  - Arreglo de registros – balance de ambas técnicas.

UNIVERSIDAD SIMÓN BOLÍVAR

Cálculo de anchuras y espacio

Cálculo dirigido por sintaxis

$$
\begin{array}{lcl}
T & \rightarrow & B \\
 & & C \\
B & \rightarrow & \textbf{int} \\
B & \rightarrow & \textbf{float} \\
C & \rightarrow & C_1\ [\textbf{num}] \\
C & \rightarrow & \lambda
\end{array}
$$

UNIVERSIDAD SIMÓN BOLÍVAR

Cálculo de anchuras y espacio

Tipos primitivos – anchura establecida por plataforma

T	$\rightarrow$	B	
		C	
B	$\rightarrow$	**int**	$\{B.type \leftarrow \textbf{int};\ B.width \leftarrow 4\}$
B	$\rightarrow$	**float**	$\{B.type \leftarrow \textbf{float};\ B.width \leftarrow 8\}$
C	$\rightarrow$	C_1 [**num**]	
C	$\rightarrow$	λ	

UNIVERSIDAD SIMÓN BOLÍVAR

Cálculo de anchuras y espacio

La anchura del tipo primitivo es base de cálculo

$$T \rightarrow B \quad \left\{ \begin{array}{ll} t & \leftarrow B.type \\ w & \leftarrow B.width \end{array} \right\}$$

$$T \rightarrow C$$

$$B \rightarrow \textbf{int} \quad \left\{ \begin{array}{ll} B.type & \leftarrow \textbf{int} \\ B.width & \leftarrow 4 \end{array} \right\}$$

$$B \rightarrow \textbf{float} \quad \left\{ \begin{array}{ll} B.type & \leftarrow \textbf{float} \\ B.width & \leftarrow 8 \end{array} \right\}$$

$$C \rightarrow C_1\ [\textbf{num}]$$

$$C \rightarrow \lambda \quad \left\{ \begin{array}{ll} C.type & \leftarrow t \\ C.width & \leftarrow w \end{array} \right\}$$

UNIVERSIDAD SIMÓN BOLÍVAR

Cálculo de anchuras y espacio

Cada dimensión de arreglo aumenta la anchura proporcionalmente

T	$\rightarrow$	B	$\left\{\begin{array}{ll} t & \leftarrow B.type \\ w & \leftarrow B.width \end{array}\right\}$
		C	
B	$\rightarrow$	**int**	$\left\{\begin{array}{ll} B.type & \leftarrow \textbf{int} \\ B.width & \leftarrow 4 \end{array}\right\}$
B	$\rightarrow$	**float**	$\left\{\begin{array}{ll} B.type & \leftarrow \textbf{float} \\ B.width & \leftarrow 8 \end{array}\right\}$
C	$\rightarrow$	C_1 [**num**]	$\left\{\begin{array}{ll} C.type & \leftarrow array(\textbf{num}.val, C_1.type) \\ C.width & \leftarrow \textbf{num}.val * C_1.width \end{array}\right\}$
C	$\rightarrow$	λ	$\left\{\begin{array}{ll} C.type & \leftarrow t \\ C.width & \leftarrow w \end{array}\right\}$

UNIVERSIDAD SIMÓN BOLÍVAR

Cálculo de anchuras y espacio

Tipo y anchura total sintetizadas

T	$\rightarrow$	B	$\{t \leftarrow B.type;\ w \leftarrow B.width\}$
		C	$\{T.type \leftarrow C.type;\ T.width \leftarrow C.width\}$
B	$\rightarrow$	**int**	$\{B.type \leftarrow \textbf{int};\ B.width \leftarrow 4\}$
B	$\rightarrow$	**float**	$\{B.type \leftarrow \textbf{float};\ B.width \leftarrow 8\}$
C	$\rightarrow$	C_1 [**num**]	$\{C.type \leftarrow array(\textbf{num}.val, C_1.type);\ C.width \leftarrow \textbf{num}.val * C_1.width\}$
C	$\rightarrow$	λ	$\{C.type \leftarrow t;\ C.width \leftarrow w\}$

Funciona directo en LR.

Para LL basta mover C_1 al final y heredar t y w.

UNIVERSIDAD SIMÓN BOLÍVAR

Calculo de desplazamientos

- En lenguajes con bloques anidados, es posible declarar símbolos con alcance limitado a ese bloque.
  - Los símbolos tienen almacenamiento local – pila de ejecución.
  - Necesario determinar su desplazamiento en relación a la base conocida.

Calculo de desplazamientos

- En lenguajes con bloques anidados, es posible declarar símbolos con alcance limitado a ese bloque.
  - Los símbolos tienen almacenamiento local – pila de ejecución.
  - Necesario determinar su desplazamiento en relación a la base conocida.
- Un problema interesante...

```
{ int foo, bar;
  { int foo, bar;
  }
  { int qux, grok, meh, wtf;
  }
}
```

UNIVERSIDAD SIMÓN BOLÍVAR

Calculo de desplazamientos

- En lenguajes con bloques anidados, es posible declarar símbolos con alcance limitado a ese bloque.
  - Los símbolos tienen almacenamiento local – pila de ejecución.
  - Necesario determinar su desplazamiento en relación a la base conocida.
- Un problema interesante...

```
{ int foo, bar;
  { int foo, bar;
  }
  { int qux, grok, meh, wtf;
  }
}
```

  - Reservar espacio para *ocho* enteros – trivial.

UNIVERSIDAD SIMÓN BOLÍVAR

Calculo de desplazamientos

- En lenguajes con bloques anidados, es posible declarar símbolos con alcance limitado a ese bloque.
  - Los símbolos tienen almacenamiento local – pila de ejecución.
  - Necesario determinar su desplazamiento en relación a la base conocida.
- Un problema interesante...

```
{ int foo, bar;
  { int foo, bar;
  }
  { int qux, grok, meh, wtf;
  }
}
```

  - Reservar espacio para *ocho* enteros – trivial.
  - Reservar espacio para *seis* enteros – astuto.

UNIVERSIDAD SIMÓN BOLÍVAR

Calculo de desplazamientos

- En lenguajes con bloques anidados, es posible declarar símbolos con alcance limitado a ese bloque.
  - Los símbolos tienen almacenamiento local – pila de ejecución.
  - Necesario determinar su desplazamiento en relación a la base conocida.
- Un problema interesante...

```
{ int foo, bar;
  { int foo, bar;
  }
  { int qux, grok, meh, wtf;
  }
}
```

  - Reservar espacio para *ocho* enteros – trivial.
  - Reservar espacio para *seis* enteros – astuto.
  - Reservar espacio para *cuatro* enteros – *you sir are awesome!*

UNIVERSIDAD SIMÓN BOLÍVAR

Cálculo de desplazamientos

Cálculo dirigido por sintaxis – la técnica básica

$$
\begin{array}{lll}
P & \rightarrow & \{ offset \leftarrow 0 \} \\
 & & D \\
D & \rightarrow & T\ \mathbf{id}; \quad \left\{ \begin{array}{l} insert(top, \mathbf{id}.lexema, T.type, offset) \\ offset \leftarrow offset + T.width \end{array} \right\} \\
 & & D_1 \\
D & \rightarrow & \lambda
\end{array}
$$

- El *offset* se calcula desde el comienzo de cada lista de definiciones.
- Se inserta en la tabla de símbolos *top* – más anidada.

UNIVERSIDAD SIMÓN BOLÍVAR

Cálculo de desplazamientos

We heard you like blocks, so we got you blocks within blocks within blocks...

- Algunos lenguajes permiten procedimientos anidados
  - El *offset* debe calcularse desde el inicio de cada procedimiento.
  - El *offset* exterior debe retomarse al terminar el anidado.

Cálculo de desplazamientos

We heard you like blocks, so we got you blocks within blocks within blocks...

- Algunos lenguajes permiten procedimientos anidados
  - El *offset* debe calcularse desde el inicio de cada procedimiento.
  - El *offset* exterior debe retomarse al terminar el anidado.
- Campos de registros (¡atributos de clases!) se manejan parecido.

UNIVERSIDAD SIMÓN BOLÍVAR

Cálculo de desplazamientos

We heard you like blocks, so we got you blocks within blocks within blocks...

- Algunos lenguajes permiten procedimientos anidados
  - El *offset* debe calcularse desde el inicio de cada procedimiento.
  - El *offset* exterior debe retomarse al terminar el anidado.
- Campos de registros (¡atributos de clases!) se manejan parecido.
- Pila de *offsets* – mantiene activo el más interno y conserva exteriores.
- Una tabla de símbolos para cada tipo registro
  - Para verificar que no haya campos con nombres duplicados.
  - Para almacenar los tipos y *offsets* de cada campo.

UNIVERSIDAD SIMÓN BOLÍVAR

Cálculo de desplazamientos

Cálculo dirigido por sintaxis – pilas con las pilas

$$T \rightarrow \textbf{record}\ \{\ D\ \}$$

Cálculo de desplazamientos

Suspender el *offset* actual y crear nueva tabla

$$T \quad \rightarrow \quad \textbf{record}\ \{ \quad \left\{ \begin{array}{l} push(Tables, top) \\ top \leftarrow new\ \ Table() \\ push(Offsets, offset) \\ offset \leftarrow 0 \end{array} \right\}$$

$$D\ \}$$

- *Tables* es la pila de tablas de símbolos – *top* es la tabla actual.
- *Offsets* es la pila de *offsets* – *offset* es el actual.

Cálculo de desplazamientos

Recuperar el *offset* previo

$$
T \rightarrow \textbf{record}\ \{
\begin{cases}
push(Tables, top) \\
top \leftarrow new\ Table() \\
push(Offsets, offset) \\
offset \leftarrow 0
\end{cases}
D\ \}
\begin{cases}
T.type & \leftarrow \textbf{record}(top) \\
T.width & \leftarrow offset \\
top & \leftarrow pop(Tables) \\
offset & \leftarrow pop(Offsets)
\end{cases}
$$

- *Tables* es la pila de tablas de símbolos – *top* es la tabla actual.
- *Offsets* es la pila de *offsets* – *offset* es el actual.
- Las declaraciones en D usan *offset* para calcular.
- Las declaraciones en D usan *top* cuando verifican e insertan.

UNIVERSIDAD SIMÓN BOLÍVAR

Cálculo de desplazamientos

El tipo para el registro se sintetiza

$$T \rightarrow \textbf{record}\ \{\ \begin{Bmatrix} push(Tables, top) \\ top \leftarrow new\ Table() \\ push(Offsets, offset) \\ offset \leftarrow 0 \end{Bmatrix}\ D\ \}\ \begin{Bmatrix} T.type & \leftarrow \textbf{record}(top) \\ T.width & \leftarrow offset \\ top & \leftarrow pop(Tables) \\ offset & \leftarrow pop(Offsets) \end{Bmatrix}$$

- *Tables* es la pila de tablas de símbolos – *top* es la tabla actual.
- *Offsets* es la pila de *offsets* – *offset* es el actual.
- Las declaraciones en D usan *offset* para calcular.
- Las declaraciones en D usan *top* cuando verifican e insertan.
- El constructor de tipo **record** apunta a su tabla privada.

UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia de Tipos en Orientación a Objetos

You said POO...

- En términos de almacenamiento las clases son similares a los registros.
- En términos de tipos puede haber relaciones entre dos clases – **Polimorfismo por Subtipos**

UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia de Tipos en Orientación a Objetos

You said POO...

- En términos de almacenamiento las clases son similares a los registros.
- En términos de tipos puede haber relaciones entre dos clases – **Polimorfismo por Subtipos**
- Principio de Sustitución de Liskov

```
class Foo ...
class Bar extends Foo ...
```

  - Un objeto `Bar` es aceptable cuando se necesita un objeto `Foo`
  - Define la relación Foo $\lhd$ Bar – "`Foo` es *sustituible* por `Bar`".
  - Reflexiva y Transitiva

UNIVERSIDAD SIMÓN BOLÍVAR

Relación Estática y Dinámica

- Se usa la relación al verificar tipos a tiempo de compilación.

```
Foo f { baz(Bar b) };
Bar b0, b1 ;

f   := b0;     -- Ok
b1  := f;      -- BAD
f.baz(b0);     -- Ok
b1.baz(f);     -- BAD
```

  - El verificador de tipos comprueba que la sustitución **siempre** sea válida.
  - Tomar en cuenta el modelo de valor o de referencia del lenguaje.

- Se genera código considerando la relación, para verificar que se respete a tiempo de ejecución

UNIVERSIDAD SIMÓN BOLÍVAR

Terminología para la relación de subtipos

. . . que no sólo aplica a POO

- **Covariante** (*Covariant*) – de subclase a superclase.
  - Cuando la operación (asignación, pasaje de parámetros, etc.) preserva el orden de la relación $\lhd$.
  - Va de tipos más específicos hacia más generales.
- **Contravariante** (*Contravariant*) – de superclase a subclase.
  - Cuando la operación (asignación, pasaje de parámetros, etc.) invierte el orden de la relación $\lhd$.
  - Va de tipos más generales hacia más específicos.
- **Invariante** – cuando no se puede ninguna de las anteriores.

UNIVERSIDAD SIMÓN BOLÍVAR

Verificación de la asignación

No es tan complicado

Suponiendo `Foo` $\lhd$ `Bar`, consideramos

```
foo := bar
```

Verificación de la asignación

No es tan complicado

Suponiendo `Foo` $\lhd$ `Bar`, consideramos

```
foo := bar
```

- Modelo de valor
  - `bar` tiene al menos los atributos que requiere `foo`.
  - Se pueden copiar para asignación profunda, descartando el excedente.

UNIVERSIDAD SIMÓN BOLÍVAR

Verificación de la asignación

No es tan complicado

Suponiendo `Foo` $\lhd$ `Bar`, consideramos

```
foo := bar
```

- Modelo de valor
  - `bar` tiene al menos los atributos que requiere `foo`.
  - Se pueden copiar para asignación profunda, descartando el excedente.
- Modelo de referencia
  - Lo *apuntado* por `bar` tiene al menos los atributos que requiere `foo`.
  - Si se hace una copia superficial, cualquier instrucción futura que opere sobre `foo` siempre tendrá los atributos necesarios.
  - Si se hace una copia profunda, se reduce al Modelo de Valor.

UNIVERSIDAD SIMÓN BOLÍVAR

Verificación de la asignación

No es tan complicado

Suponiendo `Foo` $\lhd$ `Bar`, consideramos

```
foo := bar
```

- Modelo de valor
  - `bar` tiene al menos los atributos que requiere `foo`.
  - Se pueden copiar para asignación profunda, descartando el excedente.
- Modelo de referencia
  - Lo *apuntado* por `bar` tiene al menos los atributos que requiere `foo`.
  - Si se hace una copia superficial, cualquier instrucción futura que opere sobre `foo` siempre tendrá los atributos necesarios.
  - Si se hace una copia profunda, se reduce al Modelo de Valor.
- En ambos casos, el tipo de `foo` **siempre** es `Foo` o más específico.

UNIVERSIDAD SIMÓN BOLÍVAR

Verificación de la asignación

No es tan complicado

Suponiendo `Foo` $\lhd$ `Bar`, consideramos

```
foo := bar
```

- Modelo de valor
  - `bar` tiene al menos los atributos que requiere `foo`.
  - Se pueden copiar para asignación profunda, descartando el excedente.
- Modelo de referencia
  - Lo *apuntado* por `bar` tiene al menos los atributos que requiere `foo`.
  - Si se hace una copia superficial, cualquier instrucción futura que opere sobre `foo` siempre tendrá los atributos necesarios.
  - Si se hace una copia profunda, se reduce al Modelo de Valor.
- En ambos casos, el tipo de `foo` **siempre** es `Foo` o más específico.

La asignación es covariante

UNIVERSIDAD SIMÓN BOLÍVAR

Verificación de la asignación

¿Y la matemática?

$$\frac{\rho(v) = \tau_0 \qquad \tau_0 \lhd \tau_1 \qquad \rho \vdash e : \tau_1}{\rho \vdash v := e : \tau_0}$$

- La relación $\lhd$ **no** está en el ambiente ρ, pero forma parte del sistema formal de verificación de tipos.
- La asignación es covariante – tiene el tipo de la superclase.
- Puede extenderse de símbolos a expresiones *l-value* sin mayores complicaciones.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Y las llamadas a métodos?

El caso obvio y frecuente

- El valor de retorno debe ser verificado de forma **covariante**
  - El método produce un valor o referencia de algún tipo.
  - Sólo puede ser utilizado en una expresión de tipo igual o más general.
- Cada argumento debe ser verificado de forma **covariante**
  - El método espera argumentos de tipos particulares – posiblemente quiera usar *todos* sus atributos.
  - Sólo puede suministrarse una expresión de tipo igual o más específica.

Sigue siendo covariante.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Y las llamadas a métodos que llaman métodos?

Wait, what?

- Si nuestro lenguaje con subtipos (POO o no-POO) tiene *funciones de primera clase*...
- ...quiere decir que puede haber métodos (o funciones) cuya firma incluya cosas como

```
m :: (P -> Q) -> P -> Q -> R
```

donde `P`, `Q` y `R` son tipos cualesquiera.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Y las llamadas a métodos que llaman métodos?

Wait, what?

- Si nuestro lenguaje con subtipos (POO o no-POO) tiene *funciones de primera clase*...
- ...quiere decir que puede haber métodos (o funciones) cuya firma incluya cosas como

```
m :: (P -> Q) -> P -> Q -> R
```

donde `P`, `Q` y `R` son tipos cualesquiera.

¿Y si le pasamos `P' -> Q'`?

UNIVERSIDAD SIMÓN BOLÍVAR

Subtipos de funciones

- El método `m` espera una función `P -> Q`
  pero le suministramos una función `P' -> Q'`

Subtipos de funciones

- El método `m` espera una función `P -> Q`
  pero le suministramos una función `P' -> Q'`
- ¿Cuál es la relación "segura" entre `P` y `P'`?

UNIVERSIDAD SIMÓN BOLÍVAR

Subtipos de funciones

- El método `m` espera una función `P -> Q`
  pero le suministramos una función `P' -> Q'`
- ¿Cuál es la relación "segura" entre `P` y `P'`?
  - No tenemos idea de lo que `m` pretende hacer con una función `P -> Q`.

UNIVERSIDAD SIMÓN BOLÍVAR

Subtipos de funciones

- El método `m` espera una función `P -> Q`
  pero le suministramos una función `P' -> Q'`
- ¿Cuál es la relación "segura" entre `P` y `P'`?
  - No tenemos idea de lo que `m` pretende hacer con una función `P -> Q`.
  - Debemos *suponer* que le va a pasar un `P` o algo más específico.

UNIVERSIDAD SIMÓN BOLÍVAR

Subtipos de funciones

- El método `m` espera una función `P -> Q`
  pero le suministramos una función `P' -> Q'`
- ¿Cuál es la relación "segura" entre `P` y `P'`?
  - No tenemos idea de lo que `m` pretende hacer con una función `P -> Q`.
  - Debemos *suponer* que le va a pasar un `P` o algo más específico.
  - Entonces no hay ningún peligro si $P' \lhd P$ – es **contravariante**

UNIVERSIDAD SIMÓN BOLÍVAR

Subtipos de funciones

- El método `m` espera una función `P -> Q`
  pero le suministramos una función `P' -> Q'`
- ¿Cuál es la relación "segura" entre `P` y `P'`?
  - No tenemos idea de lo que `m` pretende hacer con una función `P -> Q`.
  - Debemos *suponer* que le va a pasar un `P` o algo más específico.
  - Entonces no hay ningún peligro si $P' \lhd P$ – es **contravariante**
- ¿Cuál es la relación "segura" entre `Q` y `Q'`?

UNIVERSIDAD SIMÓN BOLÍVAR

Subtipos de funciones

- El método `m` espera una función `P -> Q`
  pero le suministramos una función `P' -> Q'`
- ¿Cuál es la relación "segura" entre `P` y `P'`?
  - No tenemos idea de lo que `m` pretende hacer con una función `P -> Q`.
  - Debemos *suponer* que le va a pasar un `P` o algo más específico.
  - Entonces no hay ningún peligro si $P' \lhd P$ – es **contravariante**
- ¿Cuál es la relación "segura" entre `Q` y `Q'`?
  - No tenemos idea de lo que `m` pretende hacer con una función `P -> Q`.

UNIVERSIDAD SIMÓN BOLÍVAR

Subtipos de funciones

- El método `m` espera una función `P -> Q`
  pero le suministramos una función `P' -> Q'`
- ¿Cuál es la relación "segura" entre `P` y `P'`?
  - No tenemos idea de lo que `m` pretende hacer con una función `P -> Q`.
  - Debemos *suponer* que le va a pasar un `P` o algo más específico.
  - Entonces no hay ningún peligro si $P' \lhd P$ – es **contravariante**
- ¿Cuál es la relación "segura" entre `Q` y `Q'`?
  - No tenemos idea de lo que `m` pretende hacer con una función `P -> Q`.
  - Debemos *suponer* que usará el `Q` en un contexto igual o más general.

UNIVERSIDAD SIMÓN BOLÍVAR

Subtipos de funciones

- El método `m` espera una función `P -> Q`
  pero le suministramos una función `P' -> Q'`
- ¿Cuál es la relación "segura" entre `P` y `P'`?
  - No tenemos idea de lo que `m` pretende hacer con una función `P -> Q`.
  - Debemos *suponer* que le va a pasar un `P` o algo más específico.
  - Entonces no hay ningún peligro si $P' \lhd P$ – es **contravariante**
- ¿Cuál es la relación "segura" entre `Q` y `Q'`?
  - No tenemos idea de lo que `m` pretende hacer con una función `P -> Q`.
  - Debemos *suponer* que usará el `Q` en un contexto igual o más general.
  - Entonces no hay ningún peligro si $Q \lhd Q'$ – es **covariante**.

UNIVERSIDAD SIMÓN BOLÍVAR

Bibliografía

- [*Aho*] (Primera Edición)
  - Capítulo 6.6
  - Ejercicios 6.20 a 6.31
- [*Aho*] (Segunda Edición)
  - Secciones 6.3.3 a 6.3.6
  - Ejercicios 6.3.1 y 6.3.2
- Principio de Sustitución de Liskov
- Type Substitution for Object Oriented Programming
- Covariance and Contravariance
- Complete las reglas de Verificación Dirigida por Sintaxis incluyendo las necesarias para procesar las declaraciones.
- Extienda la gramática para incorporar registros variantes e incorpore las reglas para calcular anchuras y desplazamientos.

UNIVERSIDAD SIMÓN BOLÍVAR