Verificación de Tipos

CI4721 – Lenguajes de Programación II

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia de Tipos

- Cuando un lenguaje permite definir tipos y asociarle nombres, es necesario establecer Reglas de Equivalencia.
  - **Equivalencia por Nombre** – el nombre asignado al tipo lo identifica y *diferencia* del resto.
  - **Equivalencia por Estructura** – el nombre asignado al tipo es una *abreviatura* para la expresión de tipos completa.
- Desde el punto de vista de eficiencia en el compilador, el cálculo de equivalencia tiene que ser muy rápido.

UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia Estructural

- Es la noción natural de equivalencia – estructuras idénticas.
- Dos tipos T_1 y T_2 son estructuralmente equivalentes:
  - Cuando son el mismo tipo primitivo – **int** es equivalente a **int**.
  - Cuando son resultado de aplicar el mismo constructor a dos tipos estructuralmente equivalentes – *pointer*(**int**) es equivalente a *pointer*(**int**).
- La *forma* de las expresiones de tipos involucradas en la comparación debe ser idéntica.

Nos lleva a una implantación recursiva obvia...

UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia Estructural

El algoritmo recursivo inocente

function $sequiv(T_1, T_2)$: **bool**

if T_1 y T_2 son el mismo tipo base **then**
return **true**
else if $T_1 = array(I_A, T_A)$ **and** $T_2 = array(I_B, T_B)$ **then**
return $sequiv(I_A, I_B)$ **and** $sequiv(T_A, T_B)$
else if $T_1 = T_{1,A} \times T_{1,B}$ **and** $T_2 = T_{2,A} \times T_{2,B}$ **then**
return $sequiv(T_{1,A}, T_{2,A})$ **and** $sequiv(T_{1,B}, T_{2,B})$
else if $T_1 = pointer(T_A)$ **and** $T_2 = pointer(T_B)$ **then**
return $sequiv(T_A, T_B)$
else if $T_1 = D_1 \rightarrow R_1$ **and** $T_2 = D_2 \rightarrow R_2$ **then**
return $sequiv(D_1, D_2)$ **and** $sequiv(R_1, R_2)$
else
return **false**
end if

UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia Estructural

El algoritmo es útil, aunque inocente

- Sólo aplicable a expresiones de tipo que sean árboles o DAGs
  - Habría que detectar ciclos para manejar tipos recursivos.
  - Habría que recordar *caminos* para comprobar la equivalencia.

UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia Estructural

El algoritmo es útil, aunque inocente

- Sólo aplicable a expresiones de tipo que sean árboles o DAGs
  - Habría que detectar ciclos para manejar tipos recursivos.
  - Habría que recordar *caminos* para comprobar la equivalencia.
- La equivalencia de los tipos índice para un arreglo seguramente necesita manejar casos especiales.
  - ¿Rangos diferentes con misma cardinalidad son equivalentes?
  - ¿Rango más amplio es equivalente a rango menos amplio?

UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia Estructural

El algoritmo es útil, aunque inocente

- Sólo aplicable a expresiones de tipo que sean árboles o DAGs
  - Habría que detectar ciclos para manejar tipos recursivos.
  - Habría que recordar *caminos* para comprobar la equivalencia.
- La equivalencia de los tipos índice para un arreglo seguramente necesita manejar casos especiales.
  - ¿Rangos diferentes con misma cardinalidad son equivalentes?
  - ¿Rango más amplio es equivalente a rango menos amplio?
- No contempla la posible simetría entre dos árboles o DAGs.
  - ¿Registros con misma cantidad y tipo de campos pero en órdenes diferentes, son equivalentes?

UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia Estructural

El algoritmo es útil, aunque inocente

- Sólo aplicable a expresiones de tipo que sean árboles o DAGs
  - Habría que detectar ciclos para manejar tipos recursivos.
  - Habría que recordar *caminos* para comprobar la equivalencia.
- La equivalencia de los tipos índice para un arreglo seguramente necesita manejar casos especiales.
  - ¿Rangos diferentes con misma cardinalidad son equivalentes?
  - ¿Rango más amplio es equivalente a rango menos amplio?
- No contempla la posible simetría entre dos árboles o DAGs.
  - ¿Registros con misma cantidad y tipo de campos pero en órdenes diferentes, son equivalentes?

Esos agregados lo hacen cada vez menos práctico.

UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia Estructural

Alternativa General pero no es para todo el mundo

- Algoritmo de Unificación
  - Método general que decide si dos grafos representan expresiones estructuralmente idénticas.
  - Viene en versiones costosa y muy costosa.
- Particularmente útil en lenguajes dinámicos con tipos anidados definidos por el usuario – ¡Prolog representando!

Lo estudiaremos en detalle al final del tema.

UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia por Nombres

- Noción práctica de equivalencia – nombres distintos, tipos distintos.

```
type link = *cell;
var  foo  : link;
     bar  : link;
     qux  : *cell;
     grok : *cell;
```

- Los tipos `link` y `*cell` son diferentes – tienen nombres diferentes.
- `foo` y `bar` tienen el mismo tipo.
- `qux` y `grok` tienen el mismo tipo.
- `foo` y `grok` no tienen el mismo tipo.

UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia por Nombres

Implantación

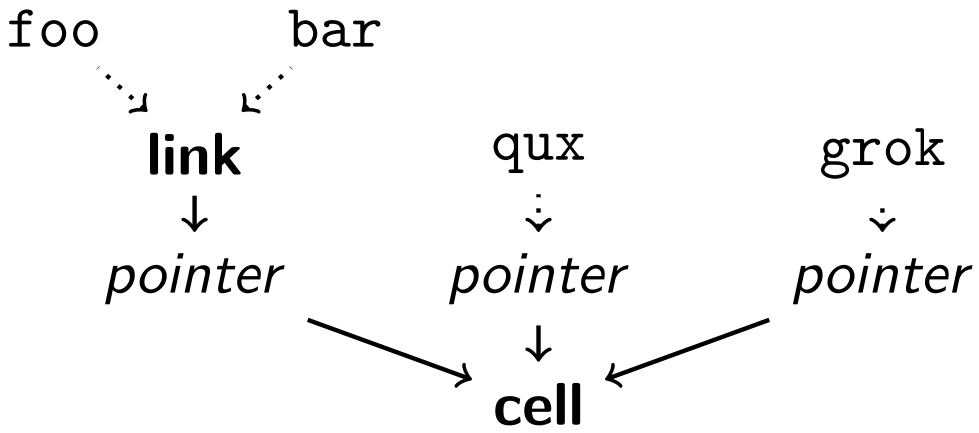
- Considerar nombres de tipos como expresiones de tipos.
- Construir los grafos de expresiones de tipos con el método habitual.
- Los nombres de tipos forman parte de la estructura
  - Apuntan a la estructura de su expresión de tipos.
  - Son apuntados por cualquier expresión que haga referencia al nombre.
- Si dos nombres apuntan al mismo nodo, son equivalentes.

UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia por Nombres

En nuestro ejemplo, si fuese Pascal

```
type  link = *cell;
var   foo  : link;
      bar  : link;
      qux  : *cell;
      grok : *cell;
```

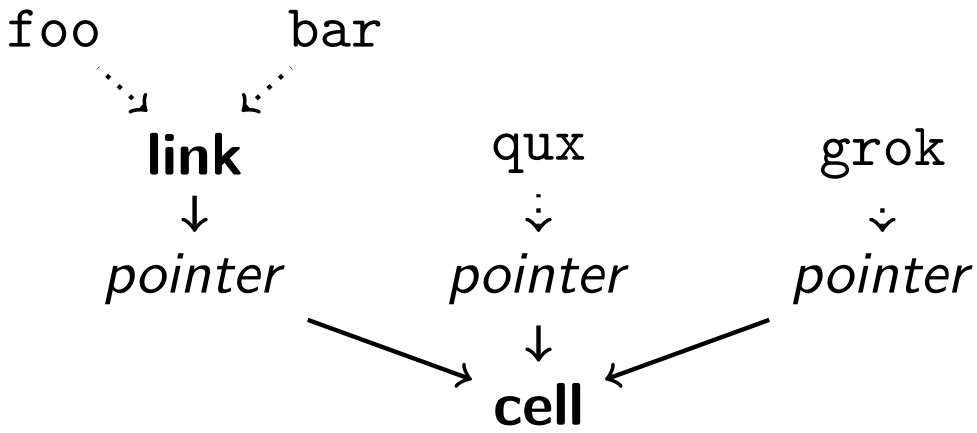


UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia por Nombres

En nuestro ejemplo, si fuese Pascal

```
type  link = *cell;
var   foo  : link;
      bar  : link;
      qux  : *cell;
      grok : *cell;
```



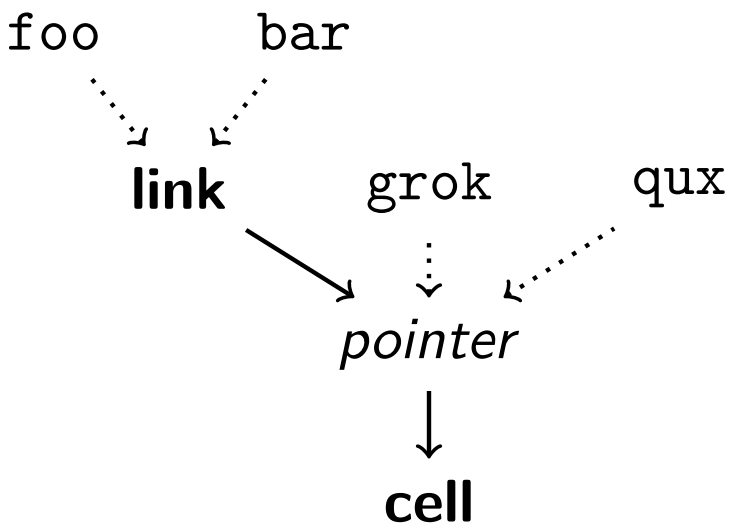
¡qux y grok **no** son equivalentes!
Declaraciones diferentes, nombres diferentes.

UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia por Nombres

En nuestro ejemplo, si usamos el cerebro

```
type  link = *cell;
var   foo  : link;
      bar  : link;
      qux  : *cell;
      grok : *cell;
```



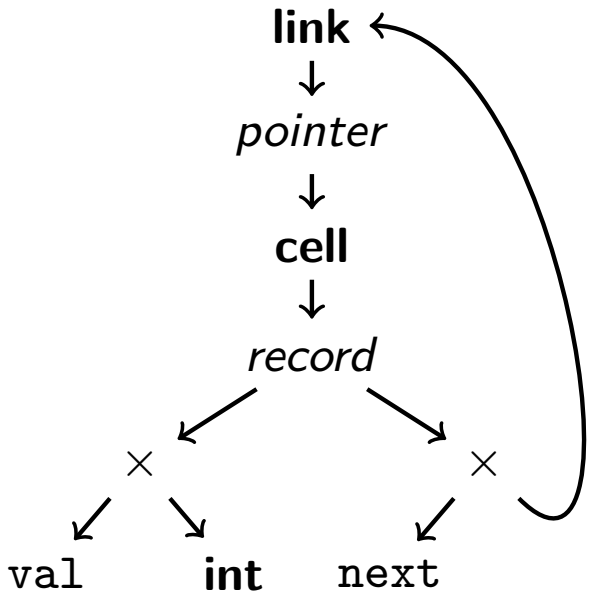
¡Compartir! – Constructores *singleton*

UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia por Nombres

La cosa funciona con tipos recursivos

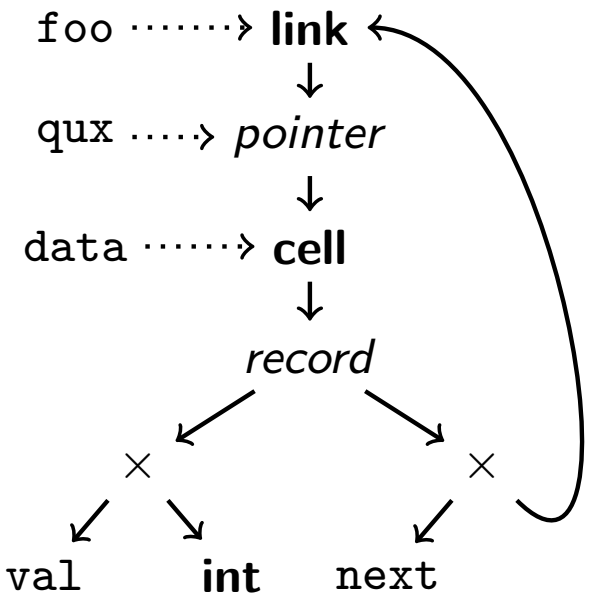
```
type link = *cell;
     cell = record
              val  : int;
              next : link;
            end;
```



UNIVERSIDAD SIMÓN BOLÍVAR

Equivalencia por Nombres

No es la flecha, sino el indio




```
data : cell;
qux = &data;       -- Ok
foo = *qux.next;   -- Ok
foo = qux;         -- Error
```

¿Equivalencia por Nombres o Estructural?

- La combinación de...
  - Permitir ciclos en las expresiones de tipos.
  - Utilizar constructores *singleton* – compartir expresiones.
  - Hacer que los nombres sean sustituidos por su estructura.
- ...facilita la "ilusión" de equivalencia estructural
  - Nombres diferentes, tipos diferentes – rápido y barato.
  - Apuntadores diferentes, estructuras diferentes casi siempre.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Equivalencia por Nombres o Estructural?

- La combinación de...
  - Permitir ciclos en las expresiones de tipos.
  - Utilizar constructores *singleton* – compartir expresiones.
  - Hacer que los nombres sean sustituidos por su estructura.
- ...facilita la "ilusión" de equivalencia estructural
  - Nombres diferentes, tipos diferentes – rápido y barato.
  - Apuntadores diferentes, estructuras diferentes casi siempre.
- Para la mayoría de los lenguajes, "casi siempre" es suficiente
  - Las excepciones o adaptaciones vienen de los tipos recursivos.
  - El orden o condiciones de declaración suelen influir – en el caso de C que se puede apuntar a un `struct` que aún no está declarado del todo.

Esta técnica mixta se usa con más frecuencia –
equivalencia por nombres siempre que se pueda,
simular estructural cuando no queda más remedio.

UNIVERSIDAD SIMÓN BOLÍVAR

Conversión de Tipos

```
int i;
float x;

i := x + i;
x := x + i;
```

- Las representaciones de bajo nivel para **int** y **float** son diferentes.
- Las operaciones de bajo nivel para sumar también son diferentes.
- El compilador se ve obligado a convertir entre representaciones.

UNIVERSIDAD SIMÓN BOLÍVAR

Conversión Explícita de Tipos

Responsabilidad del programador

- Si el sistema de tipos requiere conversión explícita, el fragmento anterior produciría errores de tipo.
- El programador *debe* indicar las conversiones necesarias.
  - Con funciones
    - `fromInteger :: Integer -> Float` de Haskell
    - `chr` y `ord` en Pascal, ...
  - Denotando el tipo deseado (*casting*) – el compilador generará el código adecuado para el cambio de representación.

UNIVERSIDAD SIMÓN BOLÍVAR

Conversión Explícita de Tipos

Responsabilidad del programador

- Si el sistema de tipos requiere conversión explícita, el fragmento anterior produciría errores de tipo.
- El programador *debe* indicar las conversiones necesarias.
  - Con funciones
    - `fromInteger :: Integer -> Float` de Haskell
    - `chr` y `ord` en Pascal, ...
  - Denotando el tipo deseado (*casting*) – el compilador generará el código adecuado para el cambio de representación.
- Implantación simple
  - Son funciones o parecen funciones.
  - Verificar los tipos de la aplicación funcional.

Es el sistema preferido en lenguajes con tipos fuertes

UNIVERSIDAD SIMÓN BOLÍVAR

Conversión Implícita de Tipos

Responsabilidad del compilador

- Cuando el sistema de tipos es capaz de hacer la conversión implícitamente, el fragmento anterior no produciría errores.
- Se aplicarían conversiones automáticas (*Coercion*) definidas por el lenguaje – en algunos casos con pérdida de información.

UNIVERSIDAD SIMÓN BOLÍVAR

Conversión Implícita de Tipos

Responsabilidad del compilador

- Cuando el sistema de tipos es capaz de hacer la conversión implícitamente, el fragmento anterior no produciría errores.
- Se aplicarían conversiones automáticas (*Coercion*) definidas por el lenguaje – en algunos casos con pérdida de información.
- Implantación compleja
  - El componente verificador de tipos debe contemplar todas las combinaciones posibles según las reglas del lenguaje.
  - El componente generador de código debe insertar el código de conversión de manera automática.

UNIVERSIDAD SIMÓN BOLÍVAR

Conversión Implícita de Tipos

Responsabilidad del compilador

- Cuando el sistema de tipos es capaz de hacer la conversión implícitamente, el fragmento anterior no produciría errores.
- Se aplicarían conversiones automáticas (*Coercion*) definidas por el lenguaje – en algunos casos con pérdida de información.
- Implantación compleja
  - El componente verificador de tipos debe contemplar todas las combinaciones posibles según las reglas del lenguaje.
  - El componente generador de código debe insertar el código de conversión de manera automática.
- En asignaciones – convertir hacia el tipo del *l-value*.
- En expresiones – convertir hacia el tipo más "amplio".

Puede ser confuso para el programador
pues nunca se indica un error.

UNIVERSIDAD SIMÓN BOLÍVAR

Esquema de Traducción con Conversión Implícita

$E \rightarrow$ **num**
$E \rightarrow$ **num.num**
$E \rightarrow$ **id**

$E \rightarrow E_1 + E_2$

Esquema de Traducción con Conversión Implícita

$E \rightarrow$ **num** $\quad \{E.type \leftarrow$ **int**$\}$

$E \rightarrow$ **num.num** $\quad \{E.type \leftarrow$ **float**$\}$

$E \rightarrow$ **id** $\quad \{E.type \leftarrow lookup($**id**$.lexema)\}$

$E \rightarrow E_1 + E_2$

UNIVERSIDAD SIMÓN BOLÍVAR

Esquema de Traducción con Conversión Implícita

$$
\begin{array}{ll}
E \rightarrow \textbf{num} & \{E.type \leftarrow \textbf{int}\} \\
E \rightarrow \textbf{num.num} & \{E.type \leftarrow \textbf{float}\} \\
E \rightarrow \textbf{id} & \{E.type \leftarrow lookup(\textbf{id}.lexema)\} \\
E \rightarrow E_1 + E_2 & \left\{ \begin{array}{ll} E.type \leftarrow & \textbf{if } (E_1.type = \textbf{int} \wedge E_2.type = \textbf{int}) \\ & \textbf{then int} \\ & \textbf{elsif } (E_1.type = \textbf{float} \wedge E_2.type = \textbf{float}) \\ & \textbf{then float} \\ & \textbf{elsif } (E_1.type = \textbf{int} \wedge E_2.type = \textbf{float}) \\ & \textbf{then float} \\ & \textbf{elsif } (E_1.type = \textbf{float} \wedge E_2.type = \textbf{int}) \\ & \textbf{then float} \end{array} \right\}
\end{array}
$$

- Todas las combinaciones necesarias –
  el generador de código debe convertir el lado preciso.

UNIVERSIDAD SIMÓN BOLÍVAR

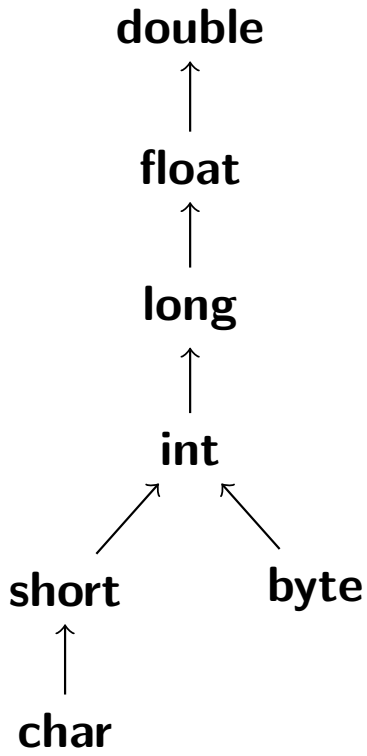
¿Cómo representar las conversiones implícitas?

- Si el lenguaje define muchas conversiones implícitas la cantidad de combinaciones aumenta.
- Las conversiones pueden ser por
  - **Ampliación** (*Widening*) cuando preservan información – de **int** a **float**, de **char** a **int**.
  - **Contracción** (*Narrowing*) cuando es *posible* perder información – de **float** a **int**, de **int** a **char**.
- Conversiones implícitas solamente cuando sean por ampliación, para no "sorprender" al programador.
- Dejar de considerar todas las combinaciones – establecer un orden parcial entre los tipos.

UNIVERSIDAD SIMÓN BOLÍVAR

Jerarquía de ampliación

Solamente para tipos primitivos



- $max(T_1, T_2)$ – máximo (o mínima cota superior) entre T_1 y T_2
  - $max(\textbf{float}, \textbf{long}) = \textbf{float}$
  - $max(\textbf{byte}, \textbf{char}) = \textbf{int}$
- Retorna **type_error** si T_1 o T_2 no está en la jerarquía.

UNIVERSIDAD SIMÓN BOLÍVAR

Esquema de Traducción con Conversión Implícita

Ahora aprovechando la jerarquía

$$E \rightarrow E_1 + E_2$$

Esquema de Traducción con Conversión Implícita

Ahora aprovechando la jerarquía

$$E \rightarrow E_1 + E_2 \quad \left\{ \begin{array}{ll} E.type & \leftarrow max(E_1.type, E_2.type) \\ a_1 & \leftarrow widen(E_1.addr, E_1.type, E.type) \\ a_2 & \leftarrow widen(E_2.addr, E_2.type, E.type) \\ \ldots & \end{array} \right\}$$

- $widen(A, T, W)$ genera el *código* necesario para ampliar los *contenidos* de A, de tipo T, para que sea un valor de tipo W.
  - Cuando $T = W$, simplemente devuelve A sin modificar.
  - En caso contrario, implanta *todas* las alternativas de conversión implícita, retornando la *dirección* donde quedará el valor ampliado.
- Los puntos suspensivos indican la ubicación del generador de código que usa a_1 y a_2 para implantar la suma de tipo $E.type$.

UNIVERSIDAD SIMÓN BOLÍVAR

Sobrecarga de Símbolos

- Cuando el contexto determina el significado particular de un símbolo, se dice que está **Sobrecargado**.
- Múltiples comportamientos para un símbolo único
  - Suma de dos enteros – `+(int,int)`
  - Suma de dos reales – `+(float,float)`
  - Suma de dos caracteres.– `+(char,char)`
  - Suma de dos ...– `+(T,T)`
- El verificador de tipos debe *resolver* la sobrecarga encontrando una interpretación *única* en el contexto.

Se reduce a mejorar el análisis de la aplicación funcional.

UNIVERSIDAD SIMÓN BOLÍVAR

Tipo de la Aplicación Funcional

Generalización del método

$$E \rightarrow \textbf{id} \qquad \{E.types \leftarrow lookup(\textbf{id}.lexema)\}$$
$$E \rightarrow E_1\ (\ E_2\ ) \qquad \{E.types \leftarrow \{t \mid \exists s \in E_2.types : s \rightarrow t \in E_1.types\}\}$$

- Las expresiones tienen un *conjunto* de tipos posibles.
- Los identificadores sobrecargados tendrían *múltiples* tipos asociados en la tabla de símbolos.
- El conjunto vacío permite detectar un error de tipos eventualmente.

UNIVERSIDAD SIMÓN BOLÍVAR

No obstante ...

- Supongamos las definiciones sobrecargadas

```
function +(i, j : integer) : integer;
function +(i, j : integer) : float;
function +(x, y : float)   : float;
```

- El conjunto de tipos posibles para + contiene

$$\textbf{int} \times \textbf{int} \to \textbf{int}$$
$$\textbf{int} \times \textbf{int} \to \textbf{float}$$
$$\textbf{float} \times \textbf{float} \to \textbf{float}$$

- Supongamos que 6, 5 y 12 *solamente* tienen tipo **int**.

UNIVERSIDAD SIMÓN BOLÍVAR

No obstante ...

- Supongamos las definiciones sobrecargadas

```
function +(i, j : integer) : integer;
function +(i, j : integer) : float;
function +(x, y : float)   : float;
```

- El conjunto de tipos posibles para + contiene

$$\textbf{int} \times \textbf{int} \to \textbf{int}$$
$$\textbf{int} \times \textbf{int} \to \textbf{float}$$
$$\textbf{float} \times \textbf{float} \to \textbf{float}$$

- Supongamos que 6, 5 y 12 *solamente* tienen tipo **int**.
  - ¿Qué tipo tiene 6+5 en 12+(6+5)?

UNIVERSIDAD SIMÓN BOLÍVAR

No obstante ...

- Supongamos las definiciones sobrecargadas

```
function +(i, j : integer) : integer;
function +(i, j : integer) : float;
function +(x, y : float)   : float;
```

- El conjunto de tipos posibles para + contiene

$$\textbf{int} \times \textbf{int} \to \textbf{int}$$

$$\textbf{int} \times \textbf{int} \to \textbf{float}$$

$$\textbf{float} \times \textbf{float} \to \textbf{float}$$

- Supongamos que 6, 5 y 12 *solamente* tienen tipo **int**.
  - ¿Qué tipo tiene 6+5 en 12+(6+5)?
  - ¿Qué tipo tiene 6+5 en x+(6+5)?

UNIVERSIDAD SIMÓN BOLÍVAR

No obstante ...

- Supongamos las definiciones sobrecargadas

```
function +(i, j : integer) : integer;
function +(i, j : integer) : float;
function +(x, y : float)   : float;
```

- El conjunto de tipos posibles para + contiene

$$\textbf{int} \times \textbf{int} \to \textbf{int}$$

$$\textbf{int} \times \textbf{int} \to \textbf{float}$$

$$\textbf{float} \times \textbf{float} \to \textbf{float}$$

- Supongamos que 6, 5 y 12 *solamente* tienen tipo **int**.
  - ¿Qué tipo tiene 6+5 en 12+(6+5)?
  - ¿Qué tipo tiene 6+5 en x+(6+5)?
  - ¿Qué tipo tiene 6+5?

UNIVERSIDAD SIMÓN BOLÍVAR

No obstante ...

- Supongamos las definiciones sobrecargadas

```
function +(i, j : integer) : integer;
function +(i, j : integer) : float;
function +(x, y : float)   : float;
```

- El conjunto de tipos posibles para + contiene

$$\textbf{int} \times \textbf{int} \to \textbf{int}$$
$$\textbf{int} \times \textbf{int} \to \textbf{float}$$
$$\textbf{float} \times \textbf{float} \to \textbf{float}$$

- Supongamos que 6, 5 y 12 *solamente* tienen tipo **int**.
  - ¿Qué tipo tiene 6+5 en 12+(6+5)?
  - ¿Qué tipo tiene 6+5 en x+(6+5)?
  - ¿Qué tipo tiene 6+5?

El contexto es crucial para determinar el tipo

UNIVERSIDAD SIMÓN BOLÍVAR

Usando el contexto para refinar los tipos

- Queremos que cada expresión tenga exactamente un tipo.

Usando el contexto para refinar los tipos

- Queremos que cada expresión tenga exactamente un tipo.
- El atributo $E.types$ sintetiza los tipos *factibles* para la expresión.
  - El esquema de traducción refina el conjunto al subir por el árbol.
  - Se usa el contexto para reducir el conjunto de tipos factibles o detectar un error de tipos por infactibilidad.

UNIVERSIDAD SIMÓN BOLÍVAR

Usando el contexto para refinar los tipos

- Queremos que cada expresión tenga exactamente un tipo.
- El atributo $E.types$ sintetiza los tipos *factibles* para la expresión.
  - El esquema de traducción refina el conjunto al subir por el árbol.
  - Se usa el contexto para reducir el conjunto de tipos factibles o detectar un error de tipos por infactibilidad.
- Al concluir el ascenso del atributo $E.types$, el conjunto debe tener exactamente un tipo, de lo contrario hay un error de tipos.

UNIVERSIDAD SIMÓN BOLÍVAR

Usando el contexto para refinar los tipos

- Queremos que cada expresión tenga exactamente un tipo.
- El atributo $E.types$ sintetiza los tipos *factibles* para la expresión.
  - El esquema de traducción refina el conjunto al subir por el árbol.
  - Se usa el contexto para reducir el conjunto de tipos factibles o detectar un error de tipos por infactibilidad.
- Al concluir el ascenso del atributo $E.types$, el conjunto debe tener exactamente un tipo, de lo contrario hay un error de tipos.
- Una vez establecido el tipo de la expresión, debe refinarse el tipo de todas las subexpresiones
  - Podría haber sub-expresiones con múltiples tipos factibles y debemos forzarlas al tipo conveniente.
  - Se usa un atributo heredado para bajar la conclusión de la refinación y aplicarla en las sub-expresiones.

Esto requiere al menos dos pasadas por el árbol.

UNIVERSIDAD SIMÓN BOLÍVAR

Bibliografía

- [*Aho*] (Primera Edición)
  - Capítulo 6
  - Ejercicios 6.11 a 6.19
- [*Aho*] (Segunda Edición)
  - Secciones 6.3.1, 6.3.2, 6.5.1, 6.5.2 y 6.5.3
  - Ejercicios 6.5.1 y 6.5.2
- Complete la generalización de sobrecarga para los operadores binarios y lógicos.

UNIVERSIDAD SIMÓN BOLÍVAR