

Procesadores de lenguaje

→ Tema 5 – Comprobación de tipos



Salvador Sánchez, Daniel Rodríguez
Departamento de Ciencias de la Computación
Universidad de Alcalá

→ Resumen

- Sistemas de tipos.
- Expresiones de tipo.
- Equivalencia de tipos.
- Sobrecarga, polimorfismo y conversiones implícitas.



→ Sistema de tipos

- **Sistema de tipos:** reglas de un lenguaje que permiten asignar tipos a las distintas partes de un programa y verificar su corrección.
- Formado por las definiciones y reglas que permiten comprobar el dominio de un identificador, y en qué contextos puede ser usado.
- Cada lenguaje tiene un sistema de tipos propio, aunque puede variar de una a otra implementación.
- La comprobación de tipos es parte del análisis semántico.



→ Sistema de tipos

- Funciones principales:
 - Inferencia de tipos: calcular y mantener la información sobre los tipos de datos.
 - Verificación de tipo: asegurar que las partes de un programa tienen sentido según las reglas de tipo del lenguaje.
- La información de tipos puede ser estática o dinámica:
 - LISP, CAML o Smalltalk utilizan información de tipos dinámica.
 - En ADA, Pascal o C la información de tipos es estática.
 - También puede ser una combinación de ambas formas.
- Cuantas más comprobaciones puedan realizarse en la fase de compilación, menos tendrán que realizarse durante la ejecución
 - Mayor eficiencia del programa objeto.



→ Sistema de tipos

- Es parte de la comprobación de tipos:
 - Conversión de tipos explícita: transformación del tipo de una expresión con un propósito determinado.
 - Coerción: conversión de tipos que realiza de forma implícita el compilador.
- Conversión de tipos **explícita**: el programador indica el tipo destino.
 - Funciona como una llamada a función: recibe un tipo y devuelve otro.
- Conversión de tipos **implícita**: el compilador convierte automáticamente elementos de un tipo en elementos de otro.
 - La conversión se lleva a cabo en la acción semántica de la regla donde se realiza.



→ Sistema de tipos

- Comprobador de tipos seguro: Durante la compilación (comprobación estática) detecta todos los posibles errores de tipo.
- Lenguaje *fuertemente tipado*: Si un fragmento de código compila es que no se van a producir errores de tipo.
- En la práctica, ningún lenguaje es tan fuertemente tipado que permita una completa comprobación estática.



→ Sistema de tipos

- Información de tipos **dinámica**: El compilador debe generar código que realice la inferencia y verificación de tipos durante la ejecución del programa que se está compilando.

```
#let primero (a,b) = a;;
primero : 'a * 'b -> 'a = <fun>
```

- Información de tipos **estática**:
 - Se utiliza para verificar la exactitud del programa antes de la ejecución.
 - Permite determinar la asignación de memoria necesaria para cada variable.

```
function Primero(a,b:integer):integer;
begin
  Primero:= a
end;
```



→ Sistema de tipos

- Tipo de datos = conjunto de valores + operaciones aplicables
- En el ámbito de los compiladores, un tipo se define mediante una **expresión de tipo** (información de tipos explícita)
 - Nombre de tipo: **float**
 - Expresión estructurada explícita: **set of integer**
 - Estas expresiones se utilizan en la construcción de otros tipos o para declarar variables.
- También es posible incluir información de tipos implícita:

```
const MAX = 10;
```



→ Sistema de tipos

- La información de tipos, implícita o explícita, se mantiene en la tabla de símbolos.
- Esta información se recupera de la tabla de símbolos mediante el verificador de tipo cuando se hace referencia al nombre asociado.
- Ejemplo:

```
A[i] → si A es de tipo array [1..10] of real  
      → si i tiene tipo integer  
      → entonces A[i] es correcto y su tipo es real
```

Saber si $i \in 1..10$, no es una verificación de tipo,
sino de si el rango es o no correcto



→ Expresiones de tipo

- Un lenguaje de programación contiene un conjunto de tipos predefinido denominados **tipos simples**.
 - Algunos lenguajes permiten definir nuevos tipos simples: enumerado, subrango.
- Todos los lenguajes permiten crear nuevos tipos complejos a partir de otros más simples mediante **constructores de tipos**:
 - Matrices, productos, registros, punteros, funciones, ...
 - En Pascal: **array, set, record**, ...
 - En C++: **struct, class, union**, ...



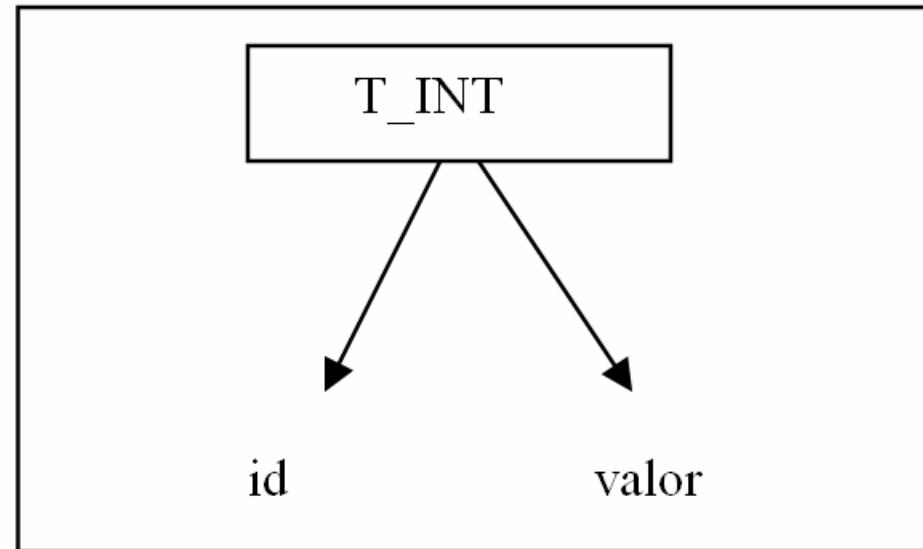
→ Expresiones de tipo

- Para analizar los diferentes tipos que intervienen dentro de un programa, el compilador debe contar con una estructura interna que le permita manejar cómodamente las expresiones de tipos.
- Esta estructura interna:
 - Debe ser fácilmente manipulable, pues su creación se realizará conforme se hace la lectura del programa fuente.
 - Debe permitir comparar fácilmente las expresiones asignadas a distintos trozos de código, especialmente a los identificadores de variables.
- La forma más habitual de representación son los grafos acíclicos dirigidos (GADs).
 - La ventaja de estas representaciones es que ocupan poca memoria y por tanto la comprobación de equivalencia se efectúa con rapidez.



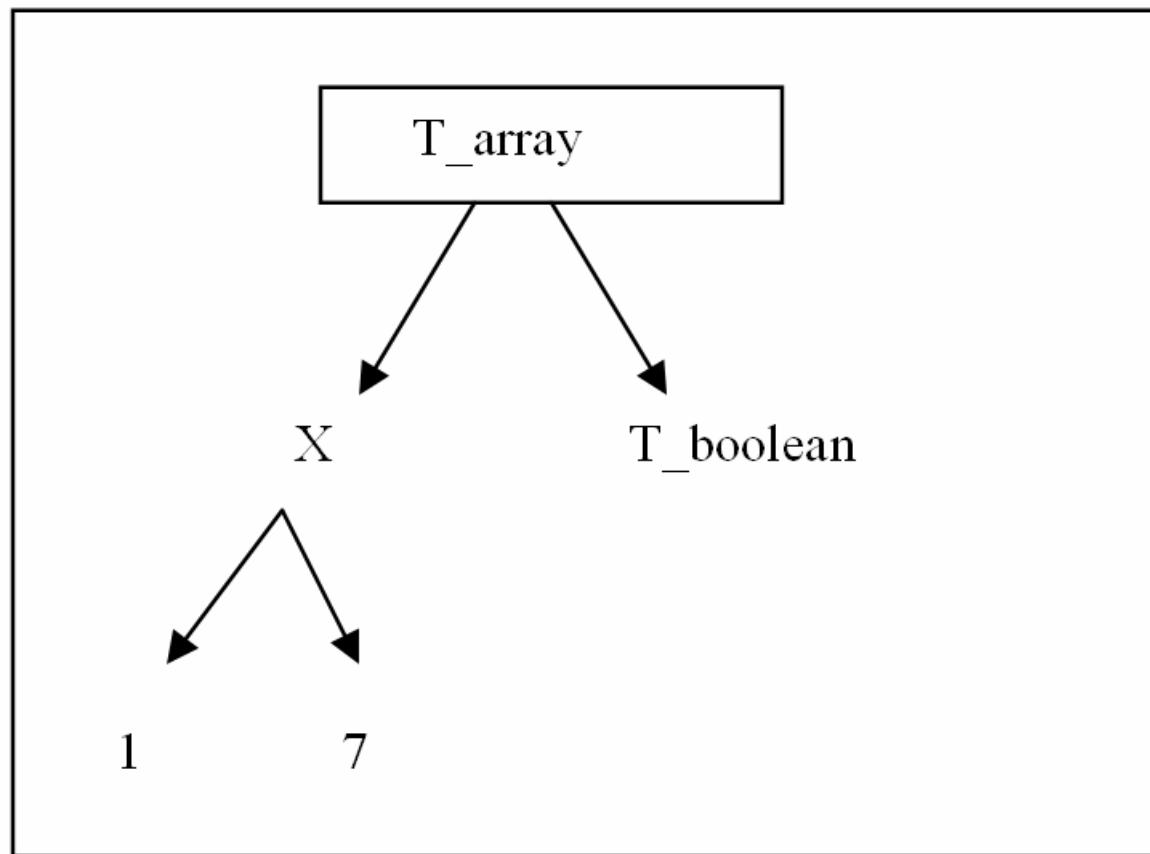
→ Expresiones de tipo: ejemplos

Tipo Simple : X: Integer



→ Expresiones de tipo: ejemplos

Array (1..7) of boolean



→ Expresiones de tipo: ejemplos

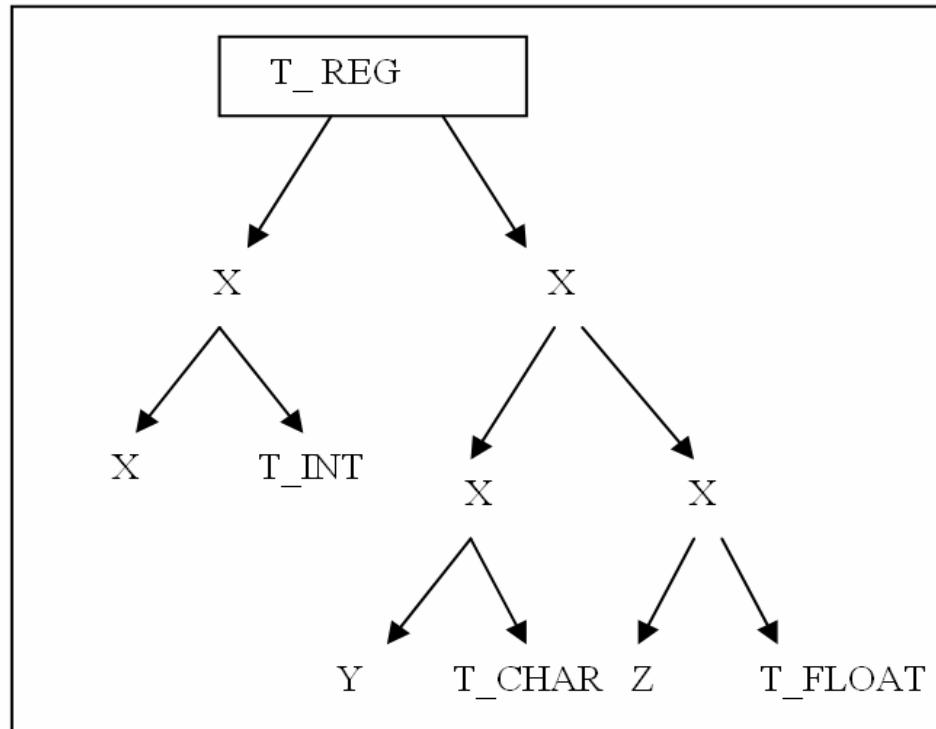
record

x : int

y : char

z : float

end record



→ Equivalencia de tipos

- Una característica importante del sistema de tipos de un lenguaje de programación es el conjunto de reglas que permiten decidir cuándo dos expresiones de tipo representan al mismo tipo.
- En muchos casos estas reglas no se definen como parte de las especificaciones del lenguaje.
 1. Diferentes interpretaciones de los creadores de compiladores.
 2. Estas diferencias afectan a la compatibilidad entre diferentes dialectos de un mismo lenguaje.
- Tres formas de equivalencia: estructural, nominal y funcional.



→ Equivalencia de tipos

- **Equivalencia estructural:** cuando se trata del mismo tipo básico, o se forman por aplicación de un mismo constructor a tipos estructuralmente equivalentes.
- Ejemplo:

```
type
    tipovector = array [0..5] of real;
    tipomatriz = array [0..5] of array [0..5] of real;
var
    vector : array [0..5] of real;
    array1 : array [0..5,0..5] of real;
    matriz : tipomatriz;
    vecvec1 : array [0..5] of tipovector;
    vecvec2 : array [0..5] of tipovector;
```

- Los tipos de las variables *array1*, *matriz*, *vecvec1* y *vecvec2* son **estructuralmente equivalentes**, pues su expresión de tipos es **array(0-5,array(0-5,real))**
- La variable *vector* no lo es. Expresión de tipos: **array(0-5,real)**



→ Equivalencia de tipos

- **Equivalencia nominal:** dos tipos son nominalmente equivalentes cuando son estructuralmente equivalentes considerando como tipos básicos e indivisibles a los identificadores de tipos.
- Ejemplo:

```
type      tipovector = array [0..5] of real;
var
          array1 : array [0..5,0..5] of real;
          vector1 : array [0..5] of tipovector;
          vector2 : array [0..5] of tipovector;
```

- Las variables *vector1* y *vector2* son nominalmente equivalentes, porque en ambos casos su expresión de tipos es: **array(0-5,tipovector)**
- La expresión de tipos de la variable *array1* no puede considerarse nominalmente equivalente a las anteriores: **array(0-5,array(0-5,real))**



→ Equivalencia de tipos

- **Equivalencia funcional:** Dos tipos se consideran equivalentes funcionalmente cuando pueden emplearse indistintamente en un mismo contexto.
- Ejemplo: *los tipos de A y de B son funcionalmente equivalentes:*

```
void Ordenar (int mat[], int n){ /* ... */ }
int A[10], B[20];
Ordenar(A,10);
Ordenar(B,20);
```

- Diferencia entre tipos compatibles y tipos funcionalmente equivalentes:
 - En el primer caso, el compilador realiza sobre uno de ellos una transformación interna para que ambos se transformen en tipos equivalentes.
 - En el segundo caso no se realiza ninguna modificación interna, sino que se relaja el criterio de equivalencia.



→ Aspectos avanzados: sobrecarga

- **Sobrecarga:** el mismo nombre de operador se refiere a diferentes funciones.
- Resolución de la **sobrecarga:** determinación del tipo de cada expresión interviniente para identificar la función a aplicar.
- La siguiente expresión es ilegal en C pero legal en C++:

```
int Primero(a,b:int) { return a; }
float Primero(a,b:float) { return a; }
```

- El compilador debe eliminar la ambigüedad:
 - Aumentando las búsquedas en la tabla de símbolos con información de tipos, pues no es suficiente con el identificador.
 - Manteniendo conjuntos de tipos para cada identificador y devolviendo el tipo adecuado en cada utilización del identificador.



→ Aspectos avanzados: polimorfismo

- Polimorfismo: una interfaz, métodos múltiples.
- Un lenguaje es polimórfico si una entidad puede tomar más de un tipo.
- La siguiente expresión es polimórfica:

```
#let Intercambiar(a,b) = (b,a);;
Intercambiar : 'a * 'b -> 'b * 'a = <fun>
```

- El compilador debe determinar en tiempo de ejecución a qué tipo corresponden los patrones de tipo de una expresión.
- Existen algoritmos de verificación para tipos polimórficos, fundamentalmente en los lenguajes de programación funcional, que implican técnicas sofisticadas de coincidencia de patrones.



→ Bibliografía

- *Básica:*

- *Compiladores: principios, técnicas y herramientas.* A.V. Aho, R. Sethi, J.D. Ullman. Addison-Wesley Iberoamerica. 1990.
- *Construcción de compiladores. Principios y práctica.* Kenneth C. Louden. Thomson-Paraninfo. 2004.

