Apuntes de Programación y estructuras de datos. Tipos de datos

Nikos Mylonakis, Fernando Orejas y Ana Cristina Zoltan

nicos@lsi.upc.edu

Dept. Llenguatges i Sistemes Informátics Universitat Politécnica de Catalunya

Barcelona

Introducción a la comprobación de tipos

- Un tipo se puede definir como una propiedad de un conjunto de valores
- Ejemplos: booleanos, enteros, entero_par, entero_impar, tablas, tuplas, etc
- Los tipos también se pueden asociar a todas las construcciones del lenguaje, por ejemplo las operaciones, las instrucciones y los subprogramas
- Los tipos se introducen en los LPs por motivos de seguridad.

- Los tipos determinan cómo se pueden utilizar las entidades del LP
- Ejemplo 1: Si x es entero lo podemos sumar a un entero pero no lo podemos utilizar en una operación lógica o como una acción
- Ejemplo 2: Si A:**tabla** [1..100] **de entero** la expresión A[50] es correcta pero A[120] no
- Para detectar los errores de uso de las diferentes entidades de un LP los LP realizan una comprobación de tipos

- Ejemplo 1: Para que $x + y$ sea correcta x e y han de ser bien enteros bien reales
- Ejemplo 2: Para que $A[50]$ sea correcta A ha de ser de tipo tabla y 50 ha de estar entre el subrango de la tabla.
- La comprobación de tipos puede ser estática (en tiempo de compilación) o dinámica (en tiempo de ejecución)
- Hay comprobaciones de tipos que sólo se pueden realizar en tiempo de ejecución.

- La comprobación de tipos dinámica hace que los programas se ejecuten más lentamente
- En algunos LP las comprobaciones dinámicas son opcionales. Java las realiza siempre
- Las ventajas de la comprobación de tipos son fiabilidad (si hay errores se detectan lo más pronto posible) y eficiencia (la memoria asignada a las variables se puede gestionar con una pila si no se utilizan punteros)
- Las desventajas son que a veces se imponen restricciones innecesarias y que es pesado tener que declarar todo

Clases de sistemas de tipos

- Los sistemas de tipos en los LP pueden ser monomórficos (toda entidad sólo tiene un único tipo) o polimórficos (una entidad puede tener más de un tipo).
- La notación algorítmica que utilizamos es monomórfica aunque puede haber sobrecarga de operadores y Java es polímorfico (por la relación clase-subclase)
- Hay dos clases de polimorfismo: polimorfismo de subclase y polimormismo parámetrico
- Ejemplo de polimorfismo parámetrico: listas de cualquier tipo ([α]) en Haskell donde α denota cualquier tipo

- Otra decisión de diseño de los sistemas de tipo LP es la equivalencia de tipos
- Ejemplo : x:=y requiere la comprobación de que ambos tipos son equivalentes
- Hay dos clases de equivalencia: por nombre y por estructura
- La equivalencia por nombre requiere que los dos tipos tengan el mismo nombre
- La equivalencia por estructura requieren que los dos tipos tengan la misma estructura

Ejemplo:

$$
\begin{array}{l}
\textbf{tipo} \\
T = \textbf{tabla}\,[1..100]\,\textbf{de}\,\textbf{entero} \\
T' = \textbf{tabla}\,[1..100]\,\textbf{de}\,\textbf{entero} \\
\textbf{ftipo} \\
\textbf{var} \\
t1, t2 : T; \\
t3 : T'; \\
t4 : \textbf{tabla}\,[1..100]\,\textbf{de}\,\textbf{entero} \\
\textbf{fvar}
\end{array}
$$

- Si hay equivalencia por nombre $t1 := t2$ es correcto pero $t1 := t3$ y $t1 := t4$ no.
- Si hay equivalencia estructural $t1 := t2$, $t1 := t3$ y $t1 := t4$ son correctas
- En Java la equivalencia de tipos es por nombre
- La equivalencia estructural varía según el LP

- Otro tema relacionado con la equivalencia de tipos son las coerciones
- Las coerciones son conversiones ímplicitas de tipos
- Ejemplo: En C se realiza una coerción al realizar $x = y$ si x es real y y entero
- En Java no existen coerciones
- Siempre se permite realizar conversiones explícitas

Concepto de subclase

- Se dice que existe una relación de subclase entre SC y C (SC es subclase de C) si siempre que podemos utilizar elementos de la clase C también podemos utilizar elementos de la subclase SC
- La relación de subclase no es lo mismo que la relación de subconjunto en teoría de conjuntos

tipo

$t =$ **tupla**

$A : entero;$

$B : booleano;$

ftupla

$st =$ **tupla**

$A : entero;$

$B : booleano;$

$C : caracter;$

ftupla

ftipo

- Los valores de st no están incluidos en t pero en cambio sí que puede existir una relación de subclase
- El motivo es porque las operación de acceso a un campo de las tuplas hace que siempre que se utilice un elemento de t se puede utilizar un elemento de st pues los campos de t están incluidos en st.
- En algunos LP como en Java para que puede existir relación de subclase se tiene que declarar explícitamente

Reglas de inferencia en sistema de tipos

- Si los tipos son propiedades de conjuntos de valores, la lógica es una buena herramienta para tratarlas
- Un sistema de tipos quedará definido por un conjunto de reglas para inferir si una cierta construcción del LP tiene su tipo correcto
- Las reglas de inferencia tendrán la forma

$$\frac{premisa1 \quad \ldots \quad premisaN}{conclusion}$$

- Ejemplo de reglas lógicas

$$\frac{A \quad B}{A \wedge B} \qquad \frac{A \Rightarrow B \quad A}{B}$$

- Las premisas y conclusiones de las reglas de un sistema de tipos para determinar el tipo de una expresión podrían ser de la forma $Expr : T$ ($Expr$ tiene el tipo T)
- Ejemplo de estas reglas serían

$$\frac{E1 : entero \quad E2 : entero}{E1 + E2 : entero} \qquad \frac{E1 : entero \quad E2 : entero}{E1 \leq E2 : booleno}$$

- El problema está en que la comprobación de tipos requiere un conjunto de declaraciones de constantes y variables ademas de la aridad de las operaciones predefinidas
- Por tanto las reglas requieren de entorno Γ que contiene una lista con todas las declaraciones definidas y predefinidas
- Nosotros no veremos las reglas de generación del entorno pero si veremos el formato general de las reglas para comprobar el tipo de las expresiones

$$\frac{}{\Gamma \cup \{x : T\} \vdash x : T}$$

$$\frac{}{\Gamma \cup \{f : T1 \times \ldots \times TN \rightarrow T\} \vdash f : T1 \times \ldots \times TN \rightarrow T}$$

$$\frac{\Gamma \vdash f : T1 \times \ldots \times TN \rightarrow T \quad \Gamma \vdash E1 : T1 \quad \ldots \quad \Gamma \vdash EN : TN}{\Gamma \vdash f(E1, \ldots, EN) : T}$$

- Los operadores unarios y binarios tendrían reglas específicas por su posición infija y prefija

Las reglas para las instrucciones serían:

$$\frac{\Gamma \vdash x : T \quad \Gamma \vdash E : T}{\Gamma \vdash x := E : Instr}$$

$$\frac{\Gamma \vdash I1 : Instr \quad \Gamma \vdash I2 : Instr}{\Gamma \vdash I1; I2 : Instr}$$

$$\frac{\Gamma \vdash E : booleano \quad \Gamma \vdash I1 : Instr \quad \Gamma \vdash I2 : Instr}{\Gamma \vdash \mathbf{Si}\, E\, \mathbf{entonces}\, I1\, \mathbf{sino} I2 : Instr}$$

$$\frac{\Gamma \vdash E : booleano \quad \Gamma \vdash I : Instr}{\Gamma \vdash \mathbf{mientras}\, E\, \mathbf{hacer}\, I\, \mathbf{fmientras} : Instr}$$

- La comprobación de tipos de los subprogramas requeriría un nuevo tipo (por ejemplo $Subpr$)
- El entorno se tendría que actualizar con la lista de parámetros y variables locales
- A continuación se pasaría a comprobar que el tipo del cuerpo del subrprograma es una instrucción

- Finalizada la comprobación el entorno se tiene que actualizar con el nombre del subprograma y el tipo de sus parámetros para comprobar que las llamadas a los subprogramas son correctas
- Debido a que pueden haber llamadas recursivas esta actualización se ha de hacer antes de comprobar el tipo del cuerpo del subprograma

Ejercicio: Comprobar el tipo de la siguiente función

$$
\begin{array}{l}
\textbf{funcion}\, f(x, y : entero)\, \textbf{retorna}\, b : booleano \\
\quad x := x + 1; \\
\quad y ::= y + 2; \\
\quad b := x \leq y \\
\quad \textbf{retorna}\, b \\
\textbf{ffuncion}
\end{array}
$$

Algoritmos de comprobación de tipos

- Los compiladores de los LP no implementan sistemas deductivos de comprobación de tipos por ser ineficientes
- Los compiladores utilizan una tabla de símbolos para guardar la información que tenemos en el entorno de las reglas
- La representación del programa es mediante un árbol sintáctico
- La comprobación de tipos requiere realizar un recorrido recursivo del árbol sintáctico desde las hojas hasta la raíz asociando un tipo a cada nodo

Reglas de inferencia con subtipo

- Recordemos que un tipo st es subtipo de un tipo t si cualquier elemento de st puede ser utilizado en un contexto en el que se utiliza t
- Las reglas de inferencia con subtipos tienen reglas adicionales como la reflexividad y la transitividad de la relación de subtipos y la regla de subsumción.
- La regla para subsumir nos dice que cuando un elemento es de un tipo st, ese elemento también tiene los tipos de los supertipos de st, esto es, de todos los tipos t de los cuales st es subtipo

La formalización de las reglas es la siguiente

$$\Gamma \vdash T < T$$

$$\frac{\Gamma \vdash T1 < T2 \quad \Gamma \vdash T2 < T3}{\Gamma \vdash T1 < T3}$$

$$\Gamma \cup \{ST < T\} \vdash ST < T$$

$$\frac{\Gamma \vdash expr : ST \quad \Gamma \vdash ST < T}{\Gamma \vdash expr : T}$$

- Veamos cómo se implementa la comprobación de tipos con subtipos
- El funcionamiento general es el mismo teniendo una tabla de símbolos y un árbol sintáctico al que hay que comprobar su tipo
- La tabla de símbolos contiene el menor subtipo asociado a las variables, constantes y parámetros de las operaciones y las relaciones de subtipos entre los tipos
- El proceso recursivo es el mismo de las hojas a la raíz pero con comprobaciones adicionales

- Si tenemos que $f : T1 \times \ldots \times TN \rightarrow$ y al realizar la comprobación de $f(E1, \ldots, EN)$ tenemos que $E1 : T1', \ldots, EN : TN'$ si $TJ' < TJ$ para todo J entre 1 y N el tipo de $f(E1, \ldots, EN)$ será correcto y de tipo T.
- Si hay sobrecarga se tienen que tratar todas las alternativas posibles

El sistema de tipos de Java

- El sistema de tipos de Java es polimórfico (por relación de subclase/subtipo)
- No realiza coerciones implícitas
- Permite sobrecarga si no hay ambiguedades
- Ejemplo $g : t1 \times t2 \rightarrow ts, g : t1' \times t2' \rightarrow ts'$, $t1 < t1', t2' < t2$ y $a : t1, b : t2'$, la expresión $g(a, b)$ da error cuando es ambigua. Si hiciesemos $x = g(a, b)$ y $x : ts$ podríamos desambiguar pero en Java da error

- Recordamos que en Java la asociación de tipos es estática pero la asociación de métodos es dinámica con ciertas restricciones
- En Java se puede asignar a una variable x de clase C un valor de una subclase de C SC pero x continua siendo de la clase C y x sólo puede acceder a los atributos declarados en C
- Con respecto de los métodos, por asociación dinámica si hay una redefinición de un método de C en SC una llamada a ese método con la variable x utilizaría el de SC

- La relación de subclase en Java se define mediante la cláusula **extends** entre clases como ya vimos y sólo se puede heredar de una superclase (herencia simple)
- En Java también se permite dejar sin definir ciertos métodos de una clase. En estos casos la clase se denomina abstracta

- Mediante la herencia podremos completar la clase definiendo los métodos no definidos
- Así podríamos implementar en Java el ejemplo del programa del cálculo de los salarios de los trabajadores de una empresa
- En Java también existen interfaces y son similares a las clases abstractas con la particularidad que permiten la herencia múltiple

- En Java tenemos dos clases de tipos: los primitivos y los referencia subclase de la clase general Object
- Los primitivos denotan un conjunto de valores y son por ejemplo int, boolean, byte, float, long, etc
- Los tipos referencia incluyen los strings, arrays y las clases definidas por el usuario
- Los tipos referencia denotan una posición de memoria que contiene un valor del tipo referenciado

- Una decisión de diseño de Java que se le critica de insegura es que si B es subclase de A entonces los arrays de B (B[]) son subclase de los arrays de A (A[])
- Si generamos una tabla de 10 posiciones de clases de B mediante la inicialización $B[]\, b = new B[10]$ y declaramos a como un array de A $A[]\, a$
- Como los arrays de B son subclase de los arrays de A podemos realizar la asignación a=b,

- Ahora podríamos hacer la asignación $a[4] = new\ A()$ y por tanto el array b contendría un valor de A
- La implementación de Java no permite ejecutarlo
- Por este y otros motivos esta relación de subclase en arrays no se considera una buena decisión de diseño

El sistema de tipos de C++

- El sistema de tipos de C++ también es polimórfico (por relación de subclase/subtipo)
- C++ sí que realiza coerciones implícitas
- También permite sobrecarga si no hay ambiguedades

- En C++ la asociación de tipos es estática y la asociación de métodos redefinidos también es dinámica
- Para ello se tienen que definir los subprogramas como virtuales y los objetos como referencias a sus clases.
- Si no se definen como virtuales se considera el subprograma sobrecargado

- La relación de subclase en C++ se define mediante el símbolo : y una lista de superclases separadas por comas. Por tanto se permite herencia multiple.
- Como ya vimos en el anterior capítulo se puede poner operadores de acceso private, public o protected.
- En C++ también se permite dejar sin definir los métodos virtuales de una clase.