Lenguajes de Programación 2022-1
Nota de clase 14: Subtipado
Orientación a Objetos

Javier Enríquez Mendoza

14 de enero de 2022

1 Orientación a Objetos

Hasta este punto en el curso hemos estudiado y formalizado las principales características de los lenguajes de programación funcionales y procedimentales, como última parte estudiaremos el paradigma de Orientación a Objetos. Podemos pensar en la Orientación a Objetos como el paradigma procedimental agregando el concepto de objeto. El uso de lenguajes orientados a objetos afecta en distintos niveles en el proceso de desarrollo:

- El análisis y diseño de los programas.
- El sistema de tipos.
- La filosofía de programación.

Los conceptos clave cuando se habla de Orientación a Objetos son las clases y los objetos, que se definen como sigue.

Definición 1.1 (Clase)**.** Una clase es una colección de atributos y métodos que definen las características y comportamientos de los objetos. Son plantillas para la creación de objetos. Se puede pensar en una clase como el tipo de un objeto.

Definición 1.2 (Objeto)**.** Un objeto es una instancia de una clase.

No existe un estándar de como se debe diseñar un lenguaje orientado a objetos, pues existen muchas formas de modelar objetos y dependen de la persona que lo diseña. Sin embargo, hay ciertas características importantes del paradigma que deben incluirse en todas las implementaciones, resumidas a continuación.

Representaciones múltiples Son los objetos los que deciden su comportamiento según la definición que tengan. Dos objetos de la misma clase puede presentar comportamiento distinto, pues cada uno tiene su propio estado e incluso con el uso de interfaces las implementaciones de los mismos métodos podrían funcionar de forma distinta.

Encapsulamiento de datos La información de un objeto solo es manipulable por el mismo. Sólo los métodos del objeto pueden manipular sus atributos. Esto implica que los cambios a la representación interna de un objeto afectan sólo a una región pequeña y fácilmente identificable del programa, lo que ayuda en el mantenimiento de sistemas muy complejos.

Herencia Permite factorizar la implementación de objetos distintos de tal forma que el comportamiento común se implemente una única vez. La mayoría de lenguajes orientados a objetos permiten este reuso de comportamientos mediante el uso de subclases que permite derivar nuevas clases a partir de otras agregando implementaciones para nuevos métodos.

Recursión Abierta Un método de un objeto puede llamar a otro dentro del mismo objeto con ayuda de las primitivas `self` ó `this`. Las cuales se ligan dinámicamente de manera que se puedan llamar métodos definidos en subclases.

Polimorfismo La capacidad de representar un objeto como perteneciente a una clase más general que con la que fue instanciado. Este tipo de polimorfismo es conocido como polimorfismo de inclusión.

En esta nota de clase se estudiara el concepto de subtipado que nos permite definir y formalizar los conceptos de herencia y polimorfismo. Mas adelante en el curso se presentará un lenguaje orientado a objetos de estudio llamado Featherweight Java con el cual se terminarán de formalizar las principales características de los lenguajes de este paradigma.

2 Subtipos

La subtipificación nos permite eliminar las conversiones explicitas de tipos en los programas haciéndolos más compactos y legibles.

Por ejemplo los tipos Int y Float tienen definiciones muy diferentes y modelan dos conjuntos de datos con propiedades distintas, sin embargo, ambos son tipos numéricos y comparten algunas operaciones aritméticas como lo es el caso de la suma. Los sistemas de tipos que hemos definido hasta ahora son muy rígidos y no nos permitirían sumar un Float con un Int pues la operación de suma verifica que ambos sumandos tengan el mismo tipo. Pero esto es algo que en la práctica se hace constantemente y sería muy tedioso tener que hacer una conversión de Int a Float solo para poder sumarlos.

Siguiendo con el ejemplo anterior, la expresión:

$$4{,}5 + 2$$

Sería rechazada por nuestro sistema de tipos, pero se trata de una expresión que desde el punto de vista de la persona que desarrolla en el lenguaje se comporta adecuadamente.

De forma general se busca que las reglas de tipado cuenten con cierta flexibilidad en los casos en los que tenga sentido recibir argumentos de otro tipo que puede usarse de forma segura como el tipo que se espera.

Para lograr esto introducimos la relación de subtipos $\leq$, que nos permite usar valores de un tipo en lugar del otro siempre y cuando éstos estén relacionados.

Definición 2.1 (Relación de subtipado)**.** El principio básico de la relación de subtipado es la propiedad de sustitución: si S es subtipo de T, es decir $\mathsf{S} \leq \mathsf{T}$ entonces cualquier expresión de tipo S puede emplearse sin causar errores en cualquier contexto que espere una expresión de tipo T.
Esta relación tiene las siguientes propiedades:

Reflexividad

$$\frac{}{\mathsf{T} \leq \mathsf{T}}$$

Transitividad

$$\frac{\mathsf{R} \leq \mathsf{S} \quad \mathsf{S} \leq \mathsf{T}}{\mathsf{R} \leq \mathsf{T}}$$

Subsunción

$$\frac{\Gamma \vdash e : \mathsf{S} \quad \mathsf{S} \leq \mathsf{T}}{\Gamma \vdash e : \mathsf{T}}$$

Esta propiedad expresa que si una expresión tiene tipo S y S es subtipo de T entonces puede usarse en cualquier contexto en donde sea necesaria una expresión de tipo T.

Formalicemos ahora el ejemplo visto antes para entender mejor el funcionamiento del subtipado.

Ejemplo 2.1 (Suma con subtipado)**.** Para este ejemplo consideremos la expresión anterior:

$$4{,}5 + 2$$

en donde $4{,}5 : \mathsf{Float}$ mientras que $2 : \mathsf{Int}$, suponemos entonces que los tipos Float e Int están relacionados bajo la relación de subtipado como

$$\mathsf{Int} \leq \mathsf{Float}$$

por lo que la derivación de tipos de la expresión anterior queda como sigue:

$$\frac{\dfrac{}{\varnothing \vdash 4{,}5 : \mathsf{Float}} \qquad \dfrac{\dfrac{}{\varnothing \vdash 2 : \mathsf{Int}} \quad \mathsf{Int} \leq \mathsf{Float}}{\varnothing \vdash 2 : \mathsf{Float}}}{\varnothing \vdash 4{,}5 + 2 : \mathsf{Float}}$$

Gracias a la propiedad de subsunción podemos construir una derivación de tipos, lo que hace que la expresión sea válida.

Existen dos estilos de manejo de subtipos:

Interpretación por conjuntos Si $\mathsf{S} \leq \mathsf{T}$ entonces toda expresión de tipo S es también una expresión de tipo T.

Interpretación por coerción La coreción se define como *la presión ejercida sobre algo para forzar su conducta*. Por lo que si $\mathsf{S} \leq \mathsf{T}$ entonces cualquier expresión de tipo S se puede convertir de forma única en una expresión de tipo T. Es decir hay una coerción (forzamiento) del tipo S al tipo T mediante una conversión implícita.
En el ejemplo anterior, por coerción el valor entero 2 se convierte en 2,0 que es un valor de tipo Float para resolver la operación.

A continuación se presentan las reglas de subtipado para los distintos constructores de tipos que hemos estudiado durante el curso.

2.1 Subtipado para tipos primitivos

En general las reglas de subtipado de los tipos primitivos de un lenguaje representan los axiomas del sistema de subtipado y el resto de las reglas se derivan con base en éstas. Estas reglas deben definirse desde el diseño del lenguaje dejando claro qué tipos son subtipos de cuáles.

Durante el curso hemos trabajado únicamente con tres tipos primitivos Nat ,Int y Bool . Para trabajar con un sistema ligeramente mas interesante agregaremos también el tipo Float con el que hemos trabajado en los ejemplos de esta nota.

Para estos tipos primitivos se define las siguientes reglas de subtipado:

$$\frac{}{\mathsf{Nat} \leq \mathsf{Int}} \qquad \frac{}{\mathsf{Int} \leq \mathsf{Float}}$$

El resto de las reglas de subtipado para tipos numéricos se pueden generar a partir de las propiedades de reflexividad y transitividad. Para el caso del tipo Bool no hay ningún subtipado definido.

2.2 Subtipado para tipos función

Para entender como funciona el subtipado con los tipos función veamos un ejemplo.

Sea $f : \mathsf{Int} \to \mathsf{Int}$, entonces se tiene que $f\, n : \mathsf{Int}$ para cualquier expresión $n : \mathsf{Int}$, así que podemos concluir que $f\, n : \mathsf{Float}$ por la regla de subsunción. De esto se sigue que:

$$\mathsf{Int} \to \mathsf{Int} \leq \mathsf{Int} \to \mathsf{Float}$$

Es decir la relación de subtipos original se preserva en el segundo argumento (el codominio) del tipo función. En tal caso se dice que esta posición es covariante.

Por el otro lado si $f : \mathsf{Float} \to \mathsf{Int}$, y como todas las expresiones $e : \mathsf{Int}$ son también de tipo Float por subsunción, entonces podemos restringir el dominio de f y de esta forma obtener una función con dominio solo en Int . Por lo que se cumple la siguiente relación de subtipado:

$$\mathsf{Float} \to \mathsf{Int} \leq \mathsf{Int} \to \mathsf{Int}$$

Es decir, la relación de subtipos original $\mathsf{Int} \leq \mathsf{Float}$ usada en el primer argumento del tipo función se invierte en el tipo función, en tal caso decimos que este argumento es contravariante.

Finalmente, usando la propiedad de transitividad tenemos el siguiente subtipado

$$\mathsf{Float} \to \mathsf{Int} \leq \mathsf{Int} \to \mathsf{Float}$$

De esta forma podemos definir la regla de subtipado para el caso general de los tipos función como sigue:

$$\frac{\mathsf{T}_1 \leq \mathsf{S}_1 \quad \mathsf{S}_2 \leq \mathsf{T}_2}{\mathsf{S}_1 \to \mathsf{S}_2 \leq \mathsf{T}_1 \to \mathsf{T}_2}$$

2.3 Subtipado para tipos suma y producto

Para el subtipado de los tipos suma y producto ambos argumentos son covariantes, es decir, se respeta el orden de la relación $\leq$. Lo que se modela con las reglas siguientes:

$$\frac{\mathsf{S}_1 \leq \mathsf{T}_1 \quad \mathsf{S}_2 \leq \mathsf{T}_2}{\mathsf{S}_1 + \mathsf{S}_2 \leq \mathsf{T}_1 + \mathsf{T}_2} \qquad \frac{\mathsf{S}_1 \leq \mathsf{T}_1 \quad \mathsf{S}_2 \leq \mathsf{T}_2}{\mathsf{S}_1 \times \mathsf{S}_2 \leq \mathsf{T}_1 \times \mathsf{T}_2}$$

Los tipos variante generalizan la regla para los tipos suma.

2.4 Subtipado para tipos registro

Los tipos registro son de suma importancia en el paradigma de Orientación a Objetos pues nos ayudan a modelar las clases y los tipos de los objetos. Para estos tipos tenemos tres reglas que modelan el subtipado entre éstos.

Amplitud Un tipo registro con campos adicionales a otro tipo registro dado es subtipo de este. Mientras más campos tenga un tipo registro, más restricciones impone en su uso y por lo tanto describe a menos valores posibles.

$$\frac{}{(l_1 : \mathsf{T}_1, \ldots, l_{n+k} : \mathsf{T}_{n+k}) \leq (l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n)}$$

Profundidad Los tipos de cada campo pueden variar, siempre y cuando la relación de subtipos se mantenga en cada uno de los campos del registro de forma covariante.

$$\frac{\mathsf{T}_1 \leq \mathsf{S}_1 \quad \cdots \quad \mathsf{T}_n \leq \mathsf{S}_n}{(l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n) \leq (l_1 : \mathsf{S}_1, \ldots, l_n : \mathsf{S}_n)}$$

Permutación El orden de los campos de un valor de tipo registro no importa.

$$\frac{(s_1 : \mathsf{S}_1, \ldots, s_n : \mathsf{S}_n) \text{ permutación de } (l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n)}{(s_1 : \mathsf{S}_1, \ldots, s_n : \mathsf{S}_n) \leq (l_1 : \mathsf{T}_1, \ldots, l_n : \mathsf{T}_n)}$$

2.5 Subtipado para Continuaciones

El subtipado de continuaciones es contravariante, pues cualquier continuación que espera un valor de tipo T puede considerarse una continuación que espera un valor de tipo S siempre y cuando el valor de tipo S pueda considerarse un valor de tipo T. Lo que se modela con la siguiente regla

$$\frac{\mathsf{S} \leq \mathsf{T}}{\mathsf{Cont}(\mathsf{T}) \leq \mathsf{Cont}(\mathsf{S})}$$

2.6 Los tipos Topy Bot

El tipo Top es el elemento máximo de la relación de subtipado $\leq$ de tal forma que cualquier tipo T cumple que

$$\frac{}{\mathsf{T} \leq \mathsf{Top}}$$

Intuitivamente Top corresponde al tipo de todos los programas correctamente tipados. Este tipo puede ser eliminado de cualquier lenguaje sin dañar sus propiedades ni seguridad. Sin embargo puede ser de utilidad de manera que no es conveniente removerlo de un sistema con subtipos. En especial en lenguajes orientados a objetos Top es de suma importancia pues corresponde al tipo Object.

Por otro lado es natural preguntarse si es de utilidad contar con un elemento mínimo para la relación de subtipado, este tipo es Bot y cualquier tipo Tcumple que

$$\frac{}{\mathsf{Bot} \leq \mathsf{T}}$$

El tipo Bot debe ser inhabitado, es decir no debe existir ninguna expresión de dicho tipo. Podría parecer que no tiene sentido sin embargo Bot proporciona una manera conveniente de expresar el hecho de que algunas operaciones no deben devolver un valor, operaciones como la creación de una excepción o la llamada a una continuación. De forma similar a como usamos el tipo Void en TinyC .

Al igual que con el tipo Top, Bot puede eliminarse sin afectar las propiedades del lenguaje y en este caso es recomendable hacerlo pues el contar con este tipo dificulta la implementación de los algoritmos de inferencia y verificación de tipos.

3 *Casting*

Si bien las transformaciones implicitas entre tipos que nos ofrece el la regla de subsunción del subtipado es una herramienta de gran utilidad, en muchos casos nos gustaría que la persona que esta desarrollando en el lenguaje tuviera un mecanismo de conversión explícita de tipos, tal mecanismo es conocido como *casting*. Y existen dos formas de emplearlo:

Upcasting En el *casting* hacia arriba a un término dado se le atribuye un supertipo del tipo esperado. De esta forma en presencia de subtipos si se tiene $\mathsf{T} \leq \mathsf{R}$ entonces podemos concluir, usando subsunción que $\Gamma \vdash \langle \mathsf{R} \rangle\, e : \mathsf{R}$, en donde $\langle \mathsf{R} \rangle$ es la notación para el *casting* explícito. Se modela con la regla de tipado:

$$\frac{\Gamma \vdash e : \mathsf{T} \qquad \mathsf{T} \leq \mathsf{R}}{\Gamma \vdash \langle \mathsf{R} \rangle\, e : \mathsf{R}}$$

Downcasting El *casting* hacia abajo asigna a una expresión un tipo arbitrario que probablemente el verificador de tipos no asignaría. La regla de atribución se modifica de la siguiente manera:

$$\frac{\Gamma \vdash e : \mathsf{T}}{\Gamma \vdash \langle \mathsf{R} \rangle\, e : \mathsf{R}}$$

Primero se verifica que e esté correctamente tipada y luego se le atribuye un tipo Rsin ninguna restricción en la relación entre Ty R.
Este tipo de *casting* pueden ser desastroso en la seguridad del lenguaje, pero sigue la filosofía "confía pero verifica". El verificador de tipos acepta el tipo dado por el *downcasting* en tiempo

de compilación. Sin embargo, en tiempo de ejecución, causará que se verifique que el valor en cuestión corresponda realmente con el tipo que se afirma.

Referencias

[1] Miranda Perea F., González Huesca L., Nota de Clase del curso de Lenguajes de Programación, Facultad de Ciencias UNAM, Semestre 2021-1.

[2] Harper R., Practical Foundations for Programming Languages. Working draft, 2010.

[3] Mitchell J., Foundations for Programming Languages. MIT Press 1996.

[4] Krishnamurthi S., Programming Languages Application and Interpretation; Version 26.04.2007.