2: Sistemas de tipos

2.1. Repaso sobre el concepto de tipos

En general, muchas personas comprenden el concepto de “tipo de datos” en un lenguaje de programación, de manera, al menos, intuitiva. Sin embargo, cuando se les pregunta qué es un tipo, suelen quedarse sin palabras para responder. Comencemos entonces por partir de esa idea intuitiva para poder comprender mejor qué son, y luego proceder con entenderlos en profundidad en lenguajes orientados a objetos. Sin embargo, cabe destacar que gran parte de lo que trataremos en este capítulo aplica a los lenguajes en general, y no solo a los lenguajes orientados a objetos, aunque eventualmente veremos detalles específicos de estos.

En primer lugar, entonces, analizaremos el siguiente pseudo-código.

Ejemplo 2.1. Variables y tipos en pseudo-código I

```
var x := 5
var y := “Hola”
```

Sí ahora consultara acerca del tipo de las variables “x” e “y”, probablemente todos coincidieron en que “x” es de tipo número, mientras que “y” es una cadena de texto (string). Pero la gran pregunta es, ¿Qué me indica esa información?

En primer lugar, me dice qué cosas puedo hacer y qué cosas no con cada una de esas variables. Se que a “x” se lo puede sumar con otro número, restarlo, multiplicarlo, etc. Pero esas operaciones no son algo que pueda hacer con “y”. Con “y” en cambio puedo hacer otras operaciones, como pasar el texto a mayúscula o minúscula, pedir la primera letra, etc.

Por otro lado, se que sí quiero volver a asignar “x” con otro valor, un dato válido sería “9”, o “-3”, incluso tal vez “3,1416”, pero no el string “Hola”. Algunos lenguajes van a permitir la asignación igual, pero sí puedo guardar cualquier cosa en cualquier lado, en cualquier momento, ¿Cómo distingo el tipo? más aún ¿Cómo se qué operaciones puedo hacer con ese dato?. Algunos lenguajes incluso proveen construcciones y herramientas que impiden tal asignación, lo que en general implica declarar los tipos de las variables, como se muestra en el siguiente ejemplo:

Ejemplo 2.2. Variables y tipos en pseudo-código II

```
var x : Número := 5
var y : String := “Hola”
```

Sin embargo, el tipo existe independientemente de su declaración o no en el código, y todos comprendemos eso, y sabemos que romper las reglas, incluso sí el lenguaje me lo permite, es peligroso.

Entonces podemos concluir que el tipo me dice dos cosas sobre una variable:

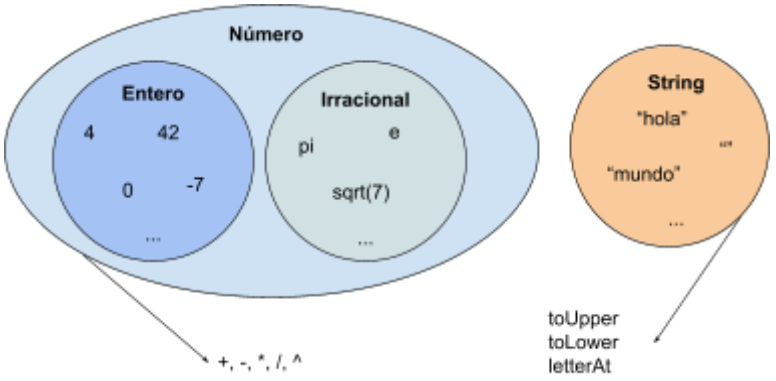
- Qué valores puede tomar esa variable
- Qué cosas puedo hacer con la variable

Esas de hecho son las dos formas de visualizar un tipo. **La primera es como un conjunto de valores (elementos) qué un dato de dicho tipo puede tomar. La segunda como un conjunto de operaciones que se puede realizar sobre un dato de ese tipo.**

Definición 2.1

Un tipo de datos es una agrupación sobre todos los datos posibles que determina qué operaciones se pueden realizar sobre un dato en particular.

Podemos entonces pensar a los tipos como conjuntos, que agrupan todos los datos posibles del lenguaje, y operaciones, asociadas a dichos conjuntos, como se muestra en el siguiente gráfico de ejemplo:



Notar como en el ejemplo, el tipo “Número” engloba a los tipos “Entero” e “Irracional”, algo que será importante más adelante cuando hablemos de subtipado.

Ahora que ya sabemos qué son los tipos con una idea más clara, podemos empezar a analizar otras cuestiones importantes, ya que distintos lenguajes trabajan los tipos de diferentes formas. Vamos a poder describir un sistema de tipos en base a tres preguntas que uno debe realizar sobre el sistema de tipos de un lenguaje:

- **¿Dónde y cuándo se realiza la verificación de tipos (sí es que se realiza)?** Lo que llamamos “**momento de la verificación**”
- **¿Qué tanta información sobre los tipos debo escribir?** Lo que llamamos “**declaratividad de tipos**”
- **¿Cómo se puede hablar de un tipo?** Lo que llamamos “**identificación de tipos**”

2.1.1. Momento de la verificación de tipos

Una de las cosas interesantes a determinar es cómo y cuándo se verifican los tipos en un lenguaje, sí es que efectivamente se verifican.

Un lenguaje puede permitir que a “x” se le asigne “3”, y luego “hola”, cambiando la variable de número a string, sin problemas, aunque tal vez, cuando se intente hacer una operación matemática con “x”, el código falle de forma irremediable. Otros lenguajes, directamente se rehusaran a ejecutar si se intenta realizar algo como eso, pues es algo sumamente desaconsejable. Otros podrían directamente ignorar el problema, y al momento de la operación matemática, hacer cualquier cosa, dando resultados inconsistentes al usuario.

En base a dónde y cuándo se verifican los tipos en un lenguaje podemos distinguir entonces 3 formas bien claras: sin tipado, tipado estático y tipado dinámico.

Sin tipado

Los tipos no se verifican en absoluto. El concepto de tipos en realidad no existe en el lenguaje de ninguna forma y solo está presente en la mente del programador.

Un claro ejemplo son los lenguajes ensambladores, en donde todo es tratado como bits, y la idea de que una serie de bits sean tratados como un número, como una letra o como cualquier otra cosa, depende del programador.

Ejemplo 2.3. Pseudo-código de lenguaje ensamblador sin tipado

```
MOV r0 101001001 # El caracter “A” se mueve a r0
MOV r1 101001010 # El caracter “B” se mueve a r1
ADD r2 r0 r1 # Se suman los contenidos de r0 y r1
             # y el resultado se guarda en r2
PRINT r2     # Se imprime lo que hay en r2
# Código válido, pero, ¿qué imprime? ¿Cuánto da “A” + “B”?
```

Tipado estático

Los tipos se verifican antes de arrancar el programa, en lo que se conoce como tiempo de compilación (ya que en general esto se encontraba en lenguajes que usan compiladores, pero hoy en día no necesariamente es así).

Este tipo de lenguajes suelen requerir al usuario declarar los tipos de los elementos (variables, parámetros, funciones, etc.) para contar con la información que le permita realizar la verificación de que los tipos coinciden, previo a ejecutar (Aunque nuevamente, no siempre es así).

Un lenguaje con tipado estático garantiza (en la mayoría de los casos, pero no siempre) que las operaciones que se realizan tienen sentido, es decir, que si se realiza una operación con un dato, esa operación, es una de las operaciones asociadas para el tipo de ese dato.

Ejemplo 2.4. Pseudo-código de lenguaje con tipado estático

```
var x : Número := 42
var y : Número := -4
var z : String := “Hola”

print (x + y)   // Funciona porque tanto x como y son números
                // y la suma es una operación que se puede hacer
                // con datos de tipo Número.

print (x + z)   // Error, z no es un número y por tanto
                // no se lo puede sumar con un número.
```

La característica más interesante del tipado estático consiste en que el código no llega siquiera a ejecutar antes de indicarse que se está incurriendo en un error. Es decir, un código con errores de tipos jamás podría llegar a una etapa productiva, pre-productiva o de testing, porque el desarrollador no puede entregar un producto ejecutable.

¿Quién realiza entonces esta verificación? Una herramienta que se conoce como **verificador de tipos**, o más comúnmente con su nombre en inglés, **typechecker**. El proceso se denomina entonces **typechecking**. Esta herramienta corre antes de ejecutar el programa o de empaquetar un producto ejecutable.

Cuándo un error de tipos en estos lenguajes se suele decir que el programa “no compila” (nuevamente, no necesariamente el lenguaje debe ser compilado, pero el término suele usarse de forma indistinta). Sin embargo, el typechecker suele estar acoplado al compilador, y es una de las primeras tareas que se realizan en el proceso de compilación, previo a la generación de código

ejecutable, motivo por el cual, al fallar el typechecking, falla la totalidad de la compilación, y ergo el nombre informal del error.

Algunos lenguajes que conocemos que hacen este tipo de verificación son C, C++, Java, entre otros muchos.

Tipado dinámico

Los tipos se verifican durante la ejecución del programa, en lo que se conoce como tiempo de ejecución.

La mayoría de estos lenguajes no poseen anotaciones de tipos en su sintaxis (aunque no siempre es así, a veces hay anotaciones, pero el lenguaje las ignora).

Ejemplo 2.5. Pseudo-código de lenguaje con tipado dinámico

```
var x := 42
var y := -4
var z := “Hola”

print (x + y)   // Funciona porque tanto x como y son números
                // aunque no se declaren los tipos.
if (False) {
  print (x + z) // Funciona, porque esta linea nunca se ejecuta,
                // y por tanto no hay conflicto.
}
print (x + z)   // Error, luego de haber ejecutado el
                // código anterior, al llegar a esta línea,
                // e intentar ejecutar, el programa falla,
                // ya que z no es un número.
```

El tipado dinámico tiene como ventajas gozar en general de código más compacto y menos verboso, además de permitir prototipar ideas rápidamente. En general también es un código que es más adaptable a grandes modificaciones, si se usa con propiedad.

Por supuesto, requiere disciplina por parte del equipo de programación. Ya que para evitar errores de tipos no se cuenta con ninguna herramienta más que la mente del desarrollador, este debe ser consciente de los tipos que maneja de forma constante.

Muchos de estos lenguajes suelen incluir constructos como “**typeof**” que permiten determinar el tipo de un dato particular, algo (en general) innecesario en lenguajes con tipado estático.

Algunos lenguajes con tipado dinámico que tal vez te resulten conocidos son Python, PHP o JavaScript.

¿Qué es mejor, estático o dinámico?

Sí comparamos ambos tipos de verificación, podemos decir que estático nos da cierto nivel de confiabilidad en nuestra aplicación, sabiendo que al menos, cierto tipos de errores (errores de tipo) no van a ocurrir en el mismo, aunque no al 100%. Siempre se pueden hacer “cosas raras” en el código, “engañando” al typechecker, o “confundiéndolo”. Un caso clásico se da cuando hacemos uso de la metaprogramación, algo que veremos en otro capítulo.

Así, también queremos contar con tipado dinámico en algunas ocasiones, y que sí hacemos una operación particular en tiempo de ejecución, esta falla con error de tipos, incluso sí el typechecker no la detectó en tiempo de compilación.

Para cerrar esta sección, vale la pena mencionar que no todo es blanco o negro en la vida, y por tanto no siempre hay solo estático o solo dinámico, sino que hay lenguajes que realizan ambos tipos de verificaciones, algunos de forma más fuerte que otros.

2.1.2. Declaratividad de los tipos

Otro de los ejes que debe ser tenido en cuenta al momento de hablar de tipos tiene que ver con cuánta información de los tipos debe escribir el programador en su código. Nuevamente, esto es algo que no es “todo” o “nada” sino que hay matices que van desde una punta a la otra.

Tipado explícito

Por ejemplo, del lado del “todo” tenemos lenguajes como C o Java. Estos lenguajes exigen que cada elemento que uno declara tenga su tipo declarado. Por ejemplo, sí se declara una variable, hay que definir en primer lugar de qué tipo va a ser dicha variable (si no se menciona el tipo, simplemente no es posible definirla). Pero además de variables, esto aplica para parámetros, tipos de retorno de funciones y procedimientos, atributos de datos con estructuras compuestas, etc.

El siguiente es un ejemplo con código C en donde se declara una función que suma dos números y una función que la utiliza para luego imprimir el resultado.

Ejemplo 2.6. Código C ejemplificando tipado explícito

```
   int sumaDeDosNumeros(int a, int b) {
//  ^                   ^      ^
//  |                   |      |
// se debe declarar     |______|
// el tipo de retorno       |
// que tiene la función   y también se debe declarar
//                        el tipo de los parámetros
     return (a + b);
   }

   void sumarEImprimir() {
//  ^
//  |
// void indica que no devuelve nada, es decir
// indica que se trata de un procedimiento
     int x = 5;
//    ^
//    |
// el tipo de las variables debe declararse, incluso cuando
// es evidente que x se trata de un entero.
     int y = 42;
     int z = sumaDeDosNumeros(x, y);
     printf(z);
   }
```

Decimos que estos lenguajes tienen **tipado explícito**. Cada elemento debe tener su tipo declarado.

Tipado implícito

En la otra punta, del lado de “nada”, tenemos lenguajes como JavaScript o Python. En estos lenguajes no se brinda información de tipos en absoluto, al menos, no como parte de la sintaxis del lenguaje. Para compensar y brindar información a los programadores se pueden usar

convenciones de nombres en las variables o parámetros, o comentarios que ayuden a documentar los tipos esperados, pero nada que sea parte explícita del lenguaje.

Decimos que estos últimos tienen **tipado implícito**. No es que los elementos no tengan un tipo. Si se le asigna 5 a una variable, va a ser un número, y no un string, pero no hay que decirlo, es el lenguaje el que se percata de que esto es así.

Ejemplo 2.7. Código Python ejemplificando tipado implícito

```python
def sumaDeDosNumeros(a, b):
  # Como se aprecia, no se definen los tipos de los
  # parámetros, son números porque los voy a usar como tales.
  # Tampoco se define lo que devuelve la función, ya que si
  # el resultado es una suma de números, obviamente
  # el resultado será un número.
  return (a + b)

def sumarEImprimir():
  x = 5
  y = 42
  # tampoco se definen tipos para las variables
  # son números porque así las declaré
  z = sumaDeDosNumeros(x, y)
  printf(z)
```

Correlación entre declaratividad y verificación

Ahora bien, hay **un error común que tiene que ver con asociar tipado explícito con tipado estático, y a su vez, tipado implícito con tipado dinámico**. Esto tiene que ver con que muchos de los lenguajes más populares suelen tener esta concordancia, pero no es del todo cierto. La declaratividad de los tipos es ortogonal al momento donde se realiza la verificación.

Un ejemplo claro en donde se puede apreciar que esta correlación no es real es el lenguaje Haskell. Haskell posee una característica conocida como **inferencia de tipos**, es decir, el lenguaje es capaz de inferir de qué tipo es un dato en base a cómo se usa, por ejemplo, sí se usa para hacer operaciones matemáticas, entonces es un número. No solo eso, sino que es capaz de determinar entonces que ese dato, siempre debe ser un número, incluso cuando se llama a otra función, etc. Haskell permite declarar los tipos de forma explícita, pero es una cuestión optativa. En el 99% de las situaciones, pueden no declararse (Hay unos pocos escenarios en donde la inferencia de tipos no alcanza y hay que declarar explícitamente los tipos de los elementos), y sin embargo, hay un proceso de compilación en donde se realiza un typechecking, y el proceso falla sí los tipos de algo no coinciden.

Así como Haskell, otros lenguajes están sumando inferencia de tipos a sus sistemas de verificación, algunos más potentes, otros menos. Java puede hacer inferencia en algunos casos muy simples, en donde se inicializa una variable con un dato simple, o en casos en donde se trabaja con generics. C# puede hacerlo con algunos elementos ligeramente más complejos. Scala tiene un nivel de inferencia bastante grande, aunque un poco menor que el de Haskell, en donde solo algunos datos deben ser declarados, tal como parámetros y funciones recursivas.

¿Qué es mejor, implícito o explícito?

La respuesta, como en tantas cosas de programación, es que depende de la necesidad. Muchos desarrolladores y equipos prefieren que el tipado sea explícito, pues entienden que el código

queda auto-documentado al declarar los tipos. Eso permite a los nuevos programadores, en particular a los más inexpertos, entender más fácilmente el código fuente. Esta es la filosofía de Java y de quienes desarrollan el lenguaje. Es decir, Java no cuenta con mayor nivel de inferencia, no porque no sea posible realizarlo, sino porque no es una característica deseable según estos desarrolladores.

Por otro lado, explicitar cada tipo puede ser visto como un boilerplate, que hace que el código sea más largo y difícil de leer, pero por sobre todo, de escribir. Entonces un ideal sería no tener que declarar nada, o al menos, declarar lo menos posible. A fin de cuentas, “yo sé lo que hago”, y “el lenguaje sabe lo que hago”, ¿Por qué sería necesario explicitar la situación aún más?. En la mayor parte del código, incluso programadores inexpertos pueden comprender lo que sucede, a veces, incluso más que en código con tipado explícito, ya que suele ser más fácil de leer.

Para comunicar de forma clara los tipos, en lenguajes implicitos, se recurre a nombres claros en variables y parámetros (Por ej. sí una variable se llama “cantidadDePersonas” nadie esperaría que fuera otra cosa más que un número), incluso a veces con convenciones bien definidas (Por ej. toda variable que comience con “n” es un número, toda variable que comience con “s” es un string, etc.). Esto hace que el código se auto-documente, al igual que en los lenguajes con tipado explícito, pero además fuerza la práctica de elegir buenos nombres, algo que no es necesario en los otros, y donde “cantidadDePersonas” suele adquirir nombres poco claros como “cp”, en particular en grupos de trabajo desprolijos o con programadores con poca formación o interés. Quienes defienden el tipado implícito ven esta característica como una ventaja.

Entonces a fin de cuentas, no hay una respuesta, sino que depende mucho de lo que uno esté buscando.

2.1.3 Identificación de tipos

Por último, un eje que es importante poder analizar tiene que ver con la forma en la que definimos un tipo. En ese sentido, los tipos que ya vienen con el lenguaje suelen tener un nombre que los identifica, como “int” o “string” (o “Int” y “String”, ya que en general los nombres son case-sensitive). Pero ¿Qué pasa con nuevos tipos? ¿Y con tipos estructurados?

Primero, recordemos las dos formas que tenemos de ver a un dato en base a un tipo. Puedo verlo como un elemento de un conjunto en particular, o puedo verlo como un elemento sobre el cuál puedo hacer ciertas operaciones.

Tipado nominal

Muchos lenguajes tienen lo que se conoce como **tipado nominal**, es decir, **para poder hablar de un tipo, debo darle un nombre a dicho tipo**. Esto ocurre en lenguajes como C o Java. **Esta visión implica definir los elementos del conjunto** de alguna forma. Veamos un ejemplo en Haskell, que hace a la declaración de tipos bastante simple:

Ejemplo 2.8. Código parcial en Haskell mostrando la definición de un nuevo tipo enumerativo

```
data DiaDeLaSemana = Lunes  | Martes  | Miercoles
                   | Jueves | Viernes | Sabado
                   | Domingo
# Se define un tipo algebráico (enumerativo) bajo el nombre
# DiaDeLaSemana, en donde los posibles valores son aquellos
# que están a la derecha, separados por pipe.
# Es decir, se define el conjunto de todos los valores posibles
# de un dato cuyo tipo es DiaDeLaSemana.
```

```
esFinde :: DiaDeLaSemana -> Bool
# Se declara el tipo de la función esFinde, como una función
# que toma un dato de tipo DiaDeLaSemana como parámetro y
# describe un Bool como resultado.
# La forma en la que se define la función a continuación
# no es relevante y por eso se omite de este ejemplo.
...
```

En ese sentido, se puede apreciar como se definió un tipo mediante la enumeración de todos los posibles valores que un dato de dicho tipo puede tomar, es decir, se definieron los elementos del **conjunto por enumeración**. Más importante aún, se definió el nombre de dicho conjunto de valores, es decir, el nombre del tipo, como “DiaDeLaSemana”.

Por supuesto, no siempre es necesario definir cada uno de los posibles valores para definir los valores del conjunto, a veces se puede definir el **conjunto por comprensión**, y en ocasiones, incluso, el conjunto es infinito. Un ejemplo claro de este tipo de definición es en los registros, es decir, datos con partes.

Ejemplo 2.8. Código parcial en Haskell mostrando la definición de un nuevo tipo enumerativo

```
data Persona = ConstructorPersona {
    nombre    :: String,
    apellido  :: String,
    edad      :: Int,
    esDonante :: Bool
}
# Se define un tipo algebráico (registro) bajo el nombre
# Persona, en donde los posibles valores son todos los que
# pueden crearse a partir del ConstructorPersona, asignando
# valores a cada uno de los campos definidos.
# Claramente la cantidad de valores posibles es infinita, pero
# el conjunto de valores igual está bien delimitado.

esMayorDeEdad :: Persona -> Bool
...
```

Nuevamente, vemos que definimos un conjunto con nombre, y por tanto, podemos decir que Haskell tiene **tipado nominal** únicamente (al menos en principio, el sistema de tipos de Haskell es sumamente complejo y no viene al caso, pero su sintaxis para definirlos es cómoda para mostrar el ejemplo en cuestión).

Ahora bien, veamos una problemática bien clara mediante el siguiente ejemplo:

Ejemplo 2.9. Ejemplo en Haskell de diferentes tipos que no pueden usarse de forma polimórfica

```
data Estudiante = CtorEstudiante {
    nombre :: String, apellido :: String, edad :: Int
}
data Empleado = CtorEmpleado {
    nombre :: String, apellido :: String, legajo :: Int
}
```

La pregunta que surge es ¿Son el mismo tipo Estudiante y Empleado? En principio, uno podría argumentar que claramente no lo son, ya que tienen nombres distintos. Sin embargo, tienen

exactamente la misma estructura, y por tanto las operaciones que puedo hacer sobre ellos (las partes del dato a las que puedo acceder) son idénticas. Entonces, sí quisiéramos crear una función que dado un Estudiante describa su nombre y apellidos completos, y una que haga lo propio para el Empleado, la realidad es que van a ser idénticas en términos de código. Esto tiene que ver con que **me interesa pensar el tipo en términos de las operaciones que puedo hacer con el dato**, y no a los posibles valores que puede tomar.

Estamos acostumbrados a este problema y a cómo se soluciona en los lenguajes orientados a objetos, usando **polimorfismo basado en subtipos**, cuya construcción depende del lenguaje. Las construcciones que permiten polimorfismo de subtipos de forma simple son comunes en los lenguajes orientados a objetos, ya que el polimorfismo es una característica fundamental del paradigma, como ya discutimos en el capítulo anterior. Veremos además los detalles de qué significa un subtipo en la siguiente sección. Pero no todos los lenguajes tienen una forma sencilla de definir este tipo de polimorfismo con subtipos. Haskell, en particular, no la tiene (por supuesto que se puede solucionar de forma sencilla mediante otras construcciones, pero para ello hay que comprender conceptos que no son relevantes en este momento). Para evitar meternos en los sistemas de tipos de los lenguajes orientados a objetos por ahora, y para evitar cambiar demasiado la sintaxis y el ejemplo, vamos a hacer trampa, e inventar una nueva sintaxis para Haskell para soportar el polimorfismo ad-hoc como se hace en los lenguajes orientados a objetos, es decir, vamos a escribir un pseudo-código que se parece a Haskell, aunque no es Haskell.

Ejemplo 2.9. Pseudo-código simil Haskell que muestra polimorfismo ad-hoc para tipos con estructura similar.

```
data Estudiante = CtorEstudiante {
    nombre :: String, apellido :: String, edad :: Int
}
data Empleado = CtorEmpleado {
    nombre :: String, apellido :: String, legajo :: Int
}
type Persona is supertype of Empleado
type Persona is supertype of Estudiante

nombreCompleto :: Persona -> String
...
```

La sintaxis que inventamos, “type ... is supertype of ...” permitiría hipotéticamente definir un nuevo tipo bajo el nombre “Persona”, que actúa de supertipo del tipo indicado a la derecha. Entonces podemos crear nuestra función “nombreCompleto” una sola vez, para “Persona” y que la podamos invocar tanto con un “Empleado” como con un “Estudiante”. Obviamente, los campos a usar en la función nombre completo solo serán “nombre” y “apellido”, lo cual hace posible ese polimorfismo, ya que la parte que es diferente, puede ignorarse.

Sin embargo, lo importante a destacar es que **se sigue trabajando con un sistema de tipado nominal**. Es decir, tuve que declarar un tipo nuevo, “Persona”, y declarar las relaciones entre los tipos. Es decir, tengo que declarar el conjunto de valores posibles para el tipo Persona, y pensar en los valores del conjunto, incluso sí lo que me interesa son las operaciones.

Tipado estructurado

El tipado estructurado es una forma de salir de ese problema. El **tipado estructurado tiene un enfoque completamente centrado en las operaciones que puedo hacer sobre un dato** (por ejemplo, qué partes tiene y puedo pedirle al dato).

Imaginemos el caso anterior, en donde la única función para la que necesito la idea de “Persona”, es en para la función “nombreCompleto”. Tener que declarar el tipo “Persona” es un poco

engorroso, y uno quisiera evitarlo. Más aún, si lo pensamos, para obtener el nombre completo, solo nos interesa que el dato posea dos campos, el nombre y el apellido, la edad, no es relevante en absoluto en esa porción de código. Sería bueno entonces poder decir que precisamente eso es lo que necesitamos.

Así, la idea de tipos estructurados se centra en declarar cuál es la estructura requerida, de forma similar a la siguiente:

Ejemplo 2.10. Pseudo-código simil Haskell que muestra tipos estructurados en la función

```
data Estudiante = CtorEstudiante {
    nombre :: String, apellido :: String, edad :: Int
}
data Empleado = CtorEmpleado {
    nombre :: String, apellido :: String, edad :: Int
}

nombreCompleto :: {nombre :: String, apellido :: String} -> String
...
```

Básicamente, lo que dice la función “nombreCompleto” es que, el parámetro, es algo que tiene “nombre” y “apellido” como campos (entendiendo esta sintaxis como, algo que tiene los campos “nombre” y “apellido”, y tal vez otros campos adicionales). Esto quiere decir que cualquier dato que cumpla con esa estructura es válido, tanto “Estudiante”, como “Empleado”, pero lo más interesante es que no son solo esos dos tipos los válidos, sino que cualquier otro dato que pueda definirse a futuro que tenga “nombre” y “apellido” como campos puede ser utilizado como argumento para esa función.

Este tipo de definición no solo evita definir un tipo para casos particulares, sino que adicionalmente provee una forma de hacer el código mucho más extensible ante cambios futuros.

El tipado estructurado puede ser más simple, pudiendo decir cosas como “un tipo que tenga al menos estos campos”, como en el ejemplo anterior, o cosas más complejas como “un tipo que tenga exactamente estos campos y nada más” o “un tipo que tenga cualquier cosa, menos estos campos”, entre otras variantes.

Algunos lenguajes que utilizan tipado estructurado son Go o TypeScript, donde este convive por supuesto con tipado nominal. Por su lado, los lenguajes con tipado implícito tienen un tipado naturalmente estructural, ya que los tipos nunca se mencionan, y el enfoque siempre está puesto en las operaciones que se pueden realizar sobre el dato.

¿Qué es mejor, nominal o estructural?

Otra vez, la comparación es relativa. No hay lenguajes que tengan tipado estructural únicamente, en general, siempre proveen tipado estructural junto con tipado nominal. Luego hay lenguajes que solo proveen tipado nominal. Entonces, bajo esta condición, prefiero entonces los primeros, porque me dan la opción de elegir, en dónde quiero tipado nominal y en dónde quiero tipado estructurado, mientras que los segundos no me dan opción alguna.

Ahora bien, cuándo quiero utilizar tipado nominal y cuándo estructurado es otra cuestión. Ponerle nombre a los tipos provee seguridad en los tipos que se van a usar (sabemos que es específicamente un tipo y no otro) y hace que nuestras funciones sean más específicas y seguras. Por otro lado, usar tipos estructurados hace que la función sea mucho más genérica, y por tanto sea utilizable en más escenarios posibles. Cuándo usar uno u otro, depende entonces de qué se está programando, y cuál es la intención de uso de este código.

2.2. Tipos en los lenguajes orientados a objetos

Vamos a comenzar por analizar un ejemplo de código simple:

Ejemplo 2.10. Pseudo-código en objetos para entender tipos.

```
class Perro {
    ladrar() { ... }
    gruñir() { ... }
}

var firulais := new Perro()
```

La pregunta obvia que nos hacemos es ¿De qué tipo es el objeto referenciado por “firulais”? La no sorprendente respuesta que la mayoría detecta inmediatamente es que “firulais” es de tipo “Perro”, pues se está haciendo “new Perro” en el código.

Sin embargo, la realidad es un poco más compleja. **En la programación orientada a objetos, la visión que debe realizarse sobre los tipos es aquella que se centra en las operaciones**, y no directamente en los nombres.

Así, **los tipos en un objeto están dados por las interfaces que dicho objeto posee**. Para recapitular, una interfaz es un subconjunto de la totalidad de mensajes que el objeto sabe responder.

Así, podemos pensar a “firulais” como un objeto que tiene los tipos, “{ ladrar() }”, “{ gruñir() }” y “{ ladrar(), gruñir() }”. Pero claro, esos tipos requieren pensar en tipado estructural para tener sentido, algo a lo que no estamos acostumbrados en principio. Entonces cuando pensamos el tipo, buscamos algo con nombre, y por supuesto, lo primero que nos surge es “Perro”, que podemos pensarlo como equivalente de “{ ladrar(), gruñir() }”. Por supuesto, si el lenguaje en el que se está trabajando no soporta tipado estructural, entonces pensar otro tipo distinto a “Perro” no tiene tampoco mucho sentido.

La realidad es que en la programación orientada a objetos, **un objeto siempre tiene uno o más tipos**. En este caso, podemos detectar 3 distintos (ignoramos el tipo con estructura vacía). Pero más aún, podemos decir que **en un lenguaje basado en clases, un objeto siempre tiene, al menos, el tipo de la clase de la cual es instancia**.

Sin embargo, hay una complejidad un poco mayor. Pensemos en el siguiente ejemplo:

Ejemplo 2.10. Pseudo-código en objetos para entender tipos.

```
class Animal {
    comer()  { ... }
}

class Perro extends Animal {
    ladrar() { ... }
    gruñir() { ... }
}

var firulais := new Perro()
```

Sí ahora se piensa en los tipos de “firulais” enfocándonos en las operaciones, vamos a ver que se suman tipos como “{ comer() }”. Pero sí pensamos en tipado nominal, vamos a encontrar que ese

tipo es lo mismo que decir “Animal”. Así, podemos asegurar ahora que “firulais” tiene no solo el tipo “Perro”, sino también el tipo “Animal”.

Esto es así porque, **cuando hay una relación de herencia, el objeto tiene tanto el tipo de la clase de la cual es instancia, como de cada una de las superclases en su cadena de herencia**. Si pensamos además que el lenguaje podría tener alguna superclase implícita en el caso de no declarar una, como “Object”, entonces esta también sería un tipo del objeto en cuestión.

El concepto que se aplica en estos escenarios es el de subtipado.

2.2.1 Subtipado

Un **subtipo** es precisamente un tipo que es englobado por otro tipo. Para que “B” sea subtipo de “A” se deben cumplir 3 propiedades que describen a un subtipo según distintos aspectos:

- **Desde las operaciones**: Todas las operaciones que puedo hacer con un elemento de tipo “A”, seguro también las puedo hacer con un elemento de tipo “B”.
- **Desde los usos**: En todo lugar que se espera algo de tipo “A”, entonces también puedo proveer algo de tipo “B”.
- **Desde la concepción**: Se debe poder decir a nivel conceptual que un “B” es un “A”.

A su vez, **si B es subtipo de A, podemos decir que A es supertipo de B**. A nivel formal, la regla de los sistemas de tipos que permiten el subtipado se conoce como **subsumption**, y permite determinar sí un tipo B es subtipo de A o no.

Si pensamos en el ejemplo anterior, “Perro” es un subtipo de “Animal”, ya que:

- Todas las operaciones que puedo hacer con un “Animal”, seguro también las puedo hacer con un “Perro”.
- En todo lugar que se espera algo de tipo “Animal”, entonces también se puede proveer algo de tipo “Perro”.
- Un “Perro” es un “Animal”.

Es decir, **en los lenguajes orientados a objetos, la relación de herencia, define una relación de subtipado**. Cada vez que decimos que “B” es subclase de “A”, también estamos diciendo que “B” es un subtipo de “A”.

Los subtipos en los lenguajes orientados a objetos son lo que **posibilita el polimorfismo con subtipos**, que es el que simplemente llamamos polimorfismo (a secas) en este paradigma, aunque existen otros tipos de polimorfismo.

En un lenguaje basado exclusivamente en clases surge una gran problemática con los subtipos. ¿Qué sucede sí quiero polimorfismo entre dos objetos que no pertenecen a la misma jerarquía de clases? Pues entonces, no hay un supertipo que subsuma a ambos, y no puedo obtener polimorfismo. Para esto, muchos lenguajes crean una construcción especial, que permite definir tipos nominales y determinar una relación de subtipado entre este y otros tipos. En la mayoría de los lenguajes esto se conoce como interfaz (el “interface” de Java y otros lenguajes), algo que llamaremos interfaces para subtipado, para distinguirlo del concepto puro de interfaz.

2.2.2. Interfaces para subtipado

Las interfaces para subtipado existen en la mayoría de los lenguajes de programación orientada a objetos modernos que tengan tipado explícito. Básicamente **proveen una forma de declarar un tipo nominal y declarar las operaciones sobre los datos de ese tipo, para luego asociar al mismo como supertipo de otros**. Veamos un ejemplo en código para que sea más claro:

Ejemplo 2.11. Código Java con interfaces para subtipado

```
interface Volador {
    volar(Int kms)
}

class Golondrina extends Animal implements Volador {
    volar(Int kms) { ... }
    ...
}

class Superhéroe implements Volador {
    volar(Int kms) { ... }
    ...
}

...
function volarDiezKilometros(Volador unObjetoVolador) {
    unObjetoVolador.volar(10)
}
...
var pepita   := new Golondrina()
var superman := new Superhéroe()
...
volarDiezKilometros(pepita)   // Funciona porque Golondrina
                              // es subtipo de volador
volarDiezKilometros(superman) // También Superheroe lo es
                              // aunque no compartan jerarquía
```

En este caso, la interfaz para subtipado declarada es “Volador”, y la misma define que las operaciones (mensajes) posibles sobre un dato (objeto) de dicho tipo son “volar(Int kms)” y nada más. Luego, en la definición de las clases “Golondrina” y “Superhéroe” se declara que estas son subtipos de “Volador” mediante el uso de la palabra clave “implements”. Como se puede observar, “Golondrina” tiene una jerarquía de clases diferente a la que tiene “Superhéroe”, heredando el primero de “Animal” y el segundo no teniendo una superclase definida (Hereda entonces de “Object”). Para una función “volarDiezKilometros” que espera un objeto que sea capaz de entender el método “volar(Int kms)” es indistinta la clase, ya que solo interesa el supertipo en cuestión, “Volador” y ya declaramos entonces que hay dos subtipos en este código. Así, tanto “pepita” como “superman” son argumentos válidos.

En general se dice que **una interfaz para subtipado determina el contrato de un objeto**. Se habla de contrato porque, en la mayoría de las implementaciones, la interfaz solo provee la signatura de los métodos a implementar, pero nada más. El comportamiento en cuestión depende pura y exclusivamente de las clases, que deberán implementar su propia versión de los métodos definidos por la interfaz. Así, en el ejemplo anterior, tanto “Golondrina” como “Superhéroe” deben implementar un método que determine el comportamiento a accionar al recibir el mensaje “volar” con una cantidad de kilómetros.

¿Es necesario el concepto de interfaz para subtipado? Depende del lenguaje. Como dijimos, sí hay tipado explícito y no existe tipado estructurado, cuando necesite poder mencionar el tipo del objeto que se recibe en la función “volarDiezKilometros”, no podría hacerlo. Ahora bien, sí existiera tipado estructurado, se podría mencionar el tipo como “{ volar(Int) }” o algo similar, pero, sigue existiendo el problema de declarar a “Golondrina” y a “Superhéroe” como subtipos de “{ volar(Int) }”. En general, lenguajes más modernos que proveen tipado estructurado son capaces

de percatarse de forma automática de esto, y entonces el usuario no tiene que realizar ninguna acción.

Sin embargo, sí bien no siempre es necesario, seguro es deseable. El tipo “{ volar(Int) }” es simple de mencionar, pero que pasaría sí el objeto que necesito debe tener una estructura mucho más compleja, con decenas de métodos. Mencionar un tipo de forma estructurada en esos casos no suele ser cómodo ni claro, y es conveniente usar tipado nominal. Entonces ahí es donde uno quiere poder crear una interfaz para subtipado y poder usar el tipo mediante su nombre.

Lo que no es necesario claro, es tener que mencionar que “Golondrina” o “Superhéroe” son subtipos de “Volador”. Un lenguaje con un analizador de tipos más avanzados podría percatarse automáticamente que tanto “Golondrina” como “Superhéroe” implementan el método “volar(Int)” y entonces automáticamente considerarlos subtipos válidos. Go es un lenguaje que tiene esta característica, entonces el desarrollador sólo tiene que escribir la interfaz y los métodos que esta tiene. Java es un lenguaje que no la tiene, y por tanto fuerza a declarar el subtipado de forma explícita.

2.2.3. Subtipado y duck typing

La idea de interfaz para subtipado consiste básicamente en nombrar una interfaz a nivel conceptual. Esto es necesario porque los tipos son explícitos y además, probablemente se quiera realizar alguna verificación de los tipos en tiempo de compilación. Pero ¿Qué sucede sí los tipos nos se deben nombrar? Pues que el concepto es completamente innecesario.

Cuando el lenguaje tiene tipado implícito (no se escriben los tipos) y además es dinámico, se tiene naturalmente tipado estructural, y por tanto el tipo “{ volar(Int) }” es un tipo perfectamente válido, incluso sí nunca lo escribimos de forma explícita. La interfaz para subtipado entonces carece de sentido, porque se usa directamente la idea de interfaz pura como tipo. Podemos concluir entonces que, **en los lenguajes con tipado dinámico e implícito, las interfaces para subtipado son innecesarias**.

Y esto tiene que ver con un concepto conocido como “***duck-typing***” (tipado de pato). El concepto implica que el foco está puesto en el comportamiento del objeto, y no en el objeto en sí. Lo relevante no es de quien es instancia el objeto, sino que sepa responder a los mensajes apropiados, es decir, que posea una interfaz en particular.

El nombre viene de la frase “*If it walks like a duck, swims like a duck, and quacks like a duck, then it might as well be a duck*” (Sí camina como pato, nada como pato, y habla como pato, entonces bien podría ser un pato). Dicho de otra forma, puedo tratar como pato cualquier cosa que tenga el comportamiento de un pato, incluso sí en lugar de un pato es un ganso o un flamenco, no es relevante, solo importa que tenga el comportamiento esperado.

El ***duck-typing*** **no es un sistema de tipos**, sino una forma de conceptualizar los tipos, que implica pensar en las interfaces del objeto, más allá de las clases definidas en el sistema.

2.24. Tipo más específico

Cuándo nos preguntamos cuál es el tipo de un objeto, como vemos, la respuesta puede ser muy grande, ya que sí pensamos en las interfaces, pueden haber decenas o centenas de interfaces a las que el objeto en cuestión obedece. Sin embargo, sí preguntamos a alguien de forma coloquial cuál es el tipo de “pepita”, esperamos que nos diga “Golondrina”, y no otra cosa. ¿Por qué?

Bueno, lo que buscamos es el tipo más específico. **El tipo más específico de un dato es aquel tipo que tiene el objeto que no es supertipo de ningún otro tipo, y a su vez, es subtipo de todos los otros posibles tipos del objeto**.

Es decir, buscamos en general decir el tipo de la clase del cual el objeto es instancia, ya que ese tipo va a responder a la relación “es un” con cualquier otro tipo que podamos mencionar del objeto.

2.3. Tipos de objetos y tipos de referencias

Comenzaremos por un código de ejemplo y unas preguntas, ¿De qué tipo es “firulais” en el siguiente ejemplo? ¿Y “lula”?

Ejemplo 2.12. Pseudo-código para mostrar tipos de objetos y de referencias

```
class Animal {
    ...
}

class Perro extends Animal {
    ...
}

...
var firulais : Perro  := new Perro()
var lula     : Animal := new Perro()
```

Por supuesto, la pregunta tiene truco, ya que lo primero que hay que responder es “¿A qué te referís cuando decís el tipo de firulais?”. Para poder responder esto, es necesario repasar los conceptos fundamentales. “firulais” es una variable, y por tanto, es una referencia que apunta a un objeto. El objeto en cuestión, no tiene nombre alguno, es la referencia la que tiene nombre. El objeto entonces, es una cosa, y la referencia, es otra. Además cuándo decimos coloquialmente “el tipo de firulais” podemos entender que seguramente se refiere al tipo más específico de “firulais”, pero aún así, podemos estar preguntando dos cosas completamente distintas. La primera, es “cuál es el tipo del objeto referenciado por la variable firulais”, y la segunda es “cuál es el tipo de la referencia firulais”.

Esto nos trae a un concepto que existe en muchos lenguajes que tienen tipado explícito, y es que **las referencias tienen un tipo que no siempre coincide con el tipo del objeto al que apuntan**. Si analizamos a firulais, vemos que el objeto es de tipo “Perro”, y la referencia (variable) está declarada como una variable del mismo tipo. Pero “lula” no cumple el mismo criterio. El objeto tiene tipo “Perro”, porque es creado con “new Perro”, pero la referencia tiene tipo “Animal”, pues así se declara.

Pero, ¿Por qué se permite esa diferencia?, y más aún ¿Qué implicancias tiene?. Como el objeto referenciado por “firulais” es de tipo “Perro”, se entiende que dicho objeto responde los mensajes “ladrar”, “gruñir” y “comer” (Este último heredado de “Animal”). A su vez la referencia “firulais” también indica ser de ese tipo, por lo que puedo hacer uso de todos esos mensajes.

Ahora bien, con “lula”, la situación es distinta, ya que la referencia dice ser un “Animal”, aunque el objeto que referencia se trate de un “Perro”. Esto trae aparejado que, sí bien el objeto referenciado entendería “ladrar” y “gruñir”, la referencia dice solo entender “comer”. Por tanto, no voy a poder mandar a “lula” otro mensaje que no sea “comer” (bueno, algo así, hay detalles que explicaremos más adelante).

Esto es algo deseable en los lenguajes con tipado estático, en donde se va a realizar una verificación de que los mensajes que se le envían a un objeto tengan sentido en tiempo de compilación. Al declarar a “lula” como un “Animal”, le estamos pidiendo al typechecker que ignore el tipo real del objeto referenciado, y que se centre en el tipo de la referencia para realizar las

verificaciones. Así, incluso sí el mensaje “ladrar” puede ser entendido por el objeto, el typechecker lo tomará como un error, pues algo de tipo “Animal” no entiende ese mensaje.

Ejemplo 2.13. Pseudo-código mostrando un tipo

```
class Entrenador {
    ...
    asignarAnimalAEntrenar(Animal animal) { ... }
    animalQueEntrena() : Animal { ... }
    ...
}
...
var firulais : Perro  := new Perro()
var pepe : Entrenador := new Entrenador()
pepe.asignarAnimalAEntrenar(firulais)
...
pepe.animalQueEntrena().comer()   // Funciona bien, porque el animal
                                  // que entrena pepe siempre sabe
                                  // comer.
...
pepe.animalQueEntrena().ladrar() // ERROR. Los animales no
                                 // saben ladrar, solo los
                                 // perros ladran.
```

En el ejemplo de arriba vemos como es útil a veces tratar a un objeto de un tipo con una referencia de otro tipo. En este caso, sí bien “pepe” está entrenando a “firulais”, pero si pido el animal que está entrenando, la referencia que se devuelve es a un “Animal” y no a un “Perro”, por lo que no puedo pedirle que ladre. Sin embargo, esto es algo deseable, pues permite que “pepe” entrene cualquier tipo de animales, ya sean perros, gatos o cualquier otro, tratando a los mismos de forma genérica.

Además, el tipo de la referencia puede tener serias implicancias en qué código se ejecutará eventualmente al recibir un mensaje, algo que veremos más en profundidad cuando analicemos binding estático y dinámico en futuros capítulos.

2.3.1 Casteo

El casteo consiste en asignar un objeto de un tipo a una referencia de otro tipo, siempre y cuando los tipos sean compatibles. Existen dos formas de castear, “hacia arriba”, lo que se conoce como **upcasting**, y que se realiza de forma implícita, y “hacia abajo”, conocido como **downcasting**, que debe hacerse de forma explícita. Decimos que son casteos “hacia arriba”, pues sube en la cadena de subtipado, y “hacia abajo” pues baja en la cadena de subtipado, entendiendo a la cadena como un gráfico vertical donde los subtipos siempre están abajo de los supertipos.

El **upcasting** es lo que ocurre en el ejemplo de “lula”, es decir, **el tipo del objeto es un subtipo de la referencia** a la que se está asignando.

Ejemplo 2.14. Pseudo-código mostrando upcasting

```
var lula : Animal := new Perro()
```

En este caso, “Perro” es subtipo de “Animal”, y por tanto, sabemos que se puede utilizar un perro en cualquier lugar en donde vaya un animal. Entonces, el typechecker hace el trabajo de casteo

por nosotros de forma implícita, porque está seguro de que la conversión de tipos va a funcionar, un “Perro” siempre es un “Animal”.

Lo importante a destacar es que **no hay ningún tipo de conversión de tipos en este proceso**. El objeto al que apunta “lula” sigue siendo de tipo “Perro”, es solo la referencia la que es tratada como un “Animal”.

El upcasting no solo se da al asignar una variable, sino al retornar un dato en una función o método, al asignar un atributo, etc. En todos lados en donde un objeto de un tipo pase a ser tratado como uno de su supertipo de forma implícita y no pueda volver a usarse como su tipo original sin un proceso explícito, hay upcasting.

Por otro lado, imaginemos el escenario en donde “pepe” es un “Entrenador”, pero estamos seguros de que “pepe” solo entrena perros, y nada más que perros. Podríamos entonces pedir el animal que está entrenando, y pedirle que ladre, pues sabemos que se trata de un perro. Pero como el tipo “Entrenador” está pensado no solo para “pepe” sino para cualquier entrenador, lo que obtenemos es un animal, y por tanto, no podemos pedirle ladrar. Esto puede resolverse mediante **downcasting**.

El downcasting sucede cuando se desea asignar un objeto declarado como de un tipo, a una referencia que es subtipo de ese tipo. El ejemplo siguiente ejemplifica este caso:

Ejemplo 2.14. Pseudo-código mostrando upcasting

```
var animalEntrenadoPorPepe : Animal := pepe.animalQueEntrena()
var firulais : Perro := (Perro) animalEntrenadoPorPepe
```

Como vemos, cuando le pedimos a “pepe” el animal que entrena, obtenemos algo de tipo “Animal”, pero sí luego queremos tratar a ese animal como un perro, deberemos entonces hacer un downcasting. El **downcasting es explícito, ya que es una operación que puede fallar, a diferencia del upcasting que nunca falla**. La sintaxis varía de lenguaje a lenguaje, pero en general implica mencionar que se desea tratar a la referencia como de un tipo en particular, como en este caso, en donde los paréntesis que antecede a la referencia “animalEntrenadoPorPepe” indican que se desea tratar dicha referencia como un “Perro”.

Por supuesto, sí “pepe” estuviera entrenando un gato y no un perro, este proceso debería fallar. Pero **el fallo en el downcasting es algo que no puede detectarse en tiempo de compilación**, por lo que el typechecker dirá que el programa cumple con los requisitos de tipado, y **será al momento de ejecutar el programa que falle**. Por eso, muchos lenguajes incluyen algún tipo mínimo de verificación de tipos en tiempo de ejecución, incluso sí su principal verificación la hacen en tiempo de compilación, para poder lidiar con este tipo de situaciones. En Java por ejemplo, este tipo de errores lanza una excepción conocida como “ClassCastException”, que ocurre en tiempo de ejecución.

2.3.1 Coerción de tipos

La coerción de tipos (type coerción) es un concepto que, en apariencia, se ve similar al concepto de casteo, pero que funciona de forma sumamente diferente. La coerción de tipos se da en lenguajes que tienen tipos primitivos y variables que guardan valores en lugar de referencias.

La coerción de tipos implica una transformación de un tipo a otro, a nivel de su representación interna. Por ejemplo, transformar un número entero en un número racional, en donde ambos tipos usan representaciones subyacentes diferentes, con diferente cantidad de bits u organización de los mismos, implica coerción.

Ejemplo 2.14. Pseudo-código mostrando coerción con widening y narrowing

```
var i1 : Int   := 7
var f1 : Float := 3.14

var f2 : Float := i1   // Coerción implícita (widening)
var i2 : Int   := (Int) f1   // Coerción explícita (narrowing)
```

Para los tipos primitivos, existe una idea similar al subtipado, que tiene que ver con la cantidad de información que puede representar. En un tipo “Float” en general se pueden representar todos los números enteros (e.g. 1 como 1.0, 2 como 2.0, etc.), pero esto no ocurre a la inversa, (no puedo representar 3,14 en un entero, pues pierdo la parte decimal). Así podemos decir que un “Int” entra en un “Float”, pero no al revés.

Entonces, tenemos dos tipos de casos de coerción, y en general se habla de datos “más grandes” y de datos “más chicos”, donde la idea no necesariamente coincide con el tamaño en bits de la representación del dato, sino que tiene que ver con la información que puede representar uno u otro. Un tipo “A” es “más grande” que otro “B”, si puedo representar la información de “B” con un dato de tipo “A”, por otro lado, “B” es “más chico” que “A” en esa situación.

Así, el primero de los casos se conoce como **ensanchamiento** (**widening**), y **ocurre cuando la asignación se realiza de un dato más chico a uno más grande**. En estos casos, **no hay pérdida de información**, y **la coerción se realiza de forma implícita**.

El segundo de los casos se conoce como **estrechamiento** (**narrowing**), y **ocurre cuando la asignación se realiza de un dato más grande a uno más chico**. En estos casos, **puede haber pérdida de información** (aunque no necesariamente, por ejemplo, 7.0 puede ser transformado a 7 sin perder información, aunque 7.5 no). **La coerción debe realizarse de forma explícita**.

El motivo por el que muchas veces los conceptos de coerción y de casteo se mezclan, es porque muchos lenguajes usan la misma sintaxis para ambos conceptos, tal como ocurre en Java o en C#. Sin embargo la diferencia es sustancial. Veamos una comparación:

Ejemplo 2.14. Pseudo-código mostrando coerción vs casteo

```
var f1 : Float   := 7.5
var i1 : Int     := f1
var f2 : Float   := i1
// f2 es 7.0, se perdió la información de los decimales al
// realizar el narrowing.

var p1 : Perro   := new Perro()
var a1 : Animal  := p1
var p2 : Perro   := a1
// p2 es exactamente igual a p1, y de hecho también lo es a1, porque
// son todas referencias, y el objeto nunca cambió realmente.
```

En el primero de los escenarios, “f2” termina con menos información, ya que al convertir “f1” en un entero, la parte decimal se pierde, y se obtiene un entero igual a 7. Al volver a pasar ese entero a flotante, se termina con “7.0”. En cambio, cuando se trabaja con objetos, y se “castea”, solo es el tipo de la referencia lo que se modifica, lo que solo sirve para dar información al typechecker. En el objeto subyacente, todo sigue igual.

La distinción de este tipo de conceptos es sumamente importante en lenguajes orientados a objetos no puros, en donde, además de objetos, existen valores que no son considerados objetos, como los números, o los booleanos. Tal es el caso de lenguajes como Java o C#, entre otros.

2.4. Tipos compuestos

Hasta ahora vimos conceptos que aparecen en la mayoría de los lenguajes de programación de una forma u otra. Sin embargo, podemos mencionar algunos conceptos más avanzados en cuanto a los sistemas de tipos, que no todos los lenguajes poseen, pero que aparecen con suficiente frecuencia para merecer ser analizados en mayor profundidad. En particular, existen casos de tipos compuestos a partir de otros (lo cual técnicamente aplica para cualquier tipo estructurado) y que suelen cobrar relevancia principalmente al momento de poder realizar análisis estático de los tipos, ya que para la mayoría de los casos de lenguajes dinámicos e implícitos, estos conceptos aparecen de forma natural.

- Tipos paramétricos
- Tipos unión
- Tipos intersección
- Tipos nullificables
- Tipos dinámicos o mixtos

Iremos viendo qué significa cada uno en las siguientes sub-secciones, pero vale aclarar que estamos hablando de conceptos que permiten definir tipos de alguna forma especial, sin que sean conceptos esotéricos o experimentales, sino sumamente probados y usados en la industria en lenguajes que van desde Java hasta TypeScript. Es decir, incluso sí nunca te topaste con un lenguaje que tenga estas características hasta ahora, son conceptos que se usan mucho en lenguajes industriales.

2.4.1 Tipos paramétricos

Los **tipos paramétricos** son uno de los casos más conocidos, ya que es una característica que aparece en varios lenguajes, desde Haskell (e.g. Tree a) hasta Java, en donde recibe el nombre de "***generics***".

Los tipos paramétricos vienen a solucionar una necesidad bien clara, definir tipos estructurados genéricos, y a la vez mantener la posibilidad de realizar análisis estáticos de tipos que sea adecuado y solucione necesidad de casteo. Un ejemplo claro son las estructuras de datos como listas, pilas y colas, donde el comportamiento de estas estructuras no depende del tipo del elemento interno, y funciona independientemente de sí se trata de pilas de enteros, de pilas de strings o de pilas de booleanos, por poner algunos ejemplos concretos.

Cabe resaltar que esta característica tiene sentido en lenguajes con tipado explícito y estático, pues de lo contrario, pensar mediante duck typing soluciona el problema, al tener tipado estructuralmente natural que se verifica solo en tiempo de ejecución.

Un tipo paramétrico puede pensarse como un tipo incompleto, que, para completarse, debe instanciarse con uno o más tipos como argumento. En la definición interna del tipo que queda incompleta y que debe ser instanciada, se realizan menciones al tipo que será dado como argumento, mediante un identificador, en lo que se conoce como **parámetro de tipo** o **variable de tipo** (Un nombre poco adecuado, ya que no actúa como variable, sino como parámetro). **Esta variable de tipo tendrá valor concreto al momento de "instanciar" el tipo incompleto**, momento en el cual se debe mencionar el valor que tendrá la variable, mencionando a un tipo concreto. A continuación hay una definición de una pila en pseudo-código que muestra la idea.

Ejemplo 2.15. Pseudo-código mostrando tipos paramétricos

```
class Pila<TElem> {
    push(TElem e) { ... }
    pop() { ... } : TElem
}

var p1 : Pila<Int> := new Pila<Int>()
var p2 : Pila<String> := new Pila<String>()
```

En el ejemplo se observa que la clase define un parámetro bajo el nombre “TElem”, mediante una sintaxis similar a la de lenguajes como Java o C# (definiendo la misma en el nombre de la clase, entre “<” y “>”). Se puede observar que, dentro de la definición de la clase, se utilizan menciones a “TElem”, tanto para “push” como para “pop”. Luego observamos dos variables, “p1” que es una instancia de “Pila” donde “TElem” es instanciada con “Int” como valor, y “p2” donde se la instancia con “String”. Ya en este escenario, podemos evaluar qué casos de operaciones funcionan y cuáles no.

Ejemplo 2.16. Pseudo-código mostrando el uso de tipos paramétricos

```
p1.push(3)      // Funciona, p1 fue instanciada con TElem como Int
p2.push(“Hola”) // Funciona, p2 fue instanciada con TElem como String
p1.push(“Hola”) // No funciona, p1 fue instanciada con TElem
                   como Int, pero se está intentando realizar
                   una operación que espera un TElem con un
                   dato de tipo String como argumento, lo cual no
                   coincide con el tipo con el que fue instanciado p1
```

Como vemos, una vez instanciado, el tipo “TElem” pasa a ser un tipo concreto, y por tanto debe respetarse.

Los tipos paramétricos permiten polimorfismo paramétrico. Este tipo de polimorfismo se suma a los conceptos de polimorfismo mediante subtipos, permitiendo entonces más y mejor reutilización.

Cotas de tipos

Pensemos nuevamente en el ejemplo del entrenador de animales, y pensemos ahora en usar parámetros de tipos. Esto es sumamente útil ya que al colocar tipos paramétricos en el entrenador de animales, podemos decir que tipo de animal es el que entrena, pudiendo tener entrenadores de perros, entrenadores de gatos, etc.

Ejemplo 2.17. Pseudo-código mostrando tipos paramétricos con subclases, con problemas

```
class Entrenador<T> {
    ...
    asignarAnimalAEntrenar(T animal) { ... }
    animalQueEntrena() : T { ... }
    ...
}
...
var firulais : Perro  := new Perro()
var garfield : Gato   := new Gato()
var pepe : Entrenador<Perro> := new Entrenador<Perro>()
```

```
pepe.asignarAnimalAEntrenar(firulais) // Bien porque firulais es un
                                      // perro, y pepe entrena perros
var unPerro : Perro  := pepe.animalQueEntrena()
                     // Efectivamente el animal entrenado es un perro
pepe.asignarAnimalAEntrenar(garfield) // Mal, porque garfield es un
                                   // gato, y pepe solo entrena perros
```

Podemos observar el beneficio, ya que ahora es seguro pedirle a pepe el animal que entrena, y cuando pedimos su animal, obtendremos un perro. Además vemos que es imposible decirle a pepe que entrene un gato.

Sin embargo, esta definición puede dar lugar a cosas sin sentido, como en la siguiente definición:

Ejemplo 2.18. Pseudo-código mostrando definición errónea de un entrenador

```
var juan : Entrenador<String> := new Entrenador<String>()
```

Qué sentido tiene decir que juan entrena “strings”, o lo mismo, que sentido tendría con otros tipos, como int o bool. Claramente eso es incorrecto.

Lo que queremos entonces es poner algún tipo de restricción sobre los posibles valores que puede tomar el tipo paramétrico “T”. Este tipo de restricciones se conocen como “**cotas de tipos**” o “***type bounding***”. En estos casos, hay dos tipos de cotas, la superior (***upper-bound***) y la inferior (***lower-bound***).

Upper-bound

Por ejemplo, una de las cosas que quisiéramos poder indicar es que “T” tiene que ser un subtipo de Animal, y no cualquier tipo. Es decir, queremos establecer su cota superior. Esta cota nos dice que los tipos que podemos utilizar deben ser subtipos del tipo establecido como cota. Veamos el ejemplo:

Ejemplo 2.19. Pseudo-código mostrando tipos paramétricos con cota superior

```
class Entrenador<T <: Animal> {
    ...
    asignarAnimalAEntrenar(T animal) { ... }
    animalQueEntrena() : T { ... }
    ...
}
var pepe : Entrenador<Perro> := new Entrenador<Perro>()
                        // Bien porque Perro es subtipo de Animal
var pepe : Entrenador<String> := new Entrenador<String>()
                        // Mal porque String no es subtipo de Animal
```

En este caso, la restricción me dice que “T” debe ser un subtipo de Animal (o el mismo tipo Animal), para lo cual elegimos la sintaxis “T <: Animal”. Esto implica que “String” deja de ser un tipo válido como argumento para el parámetro de tipos, ya que no cumple con la restricción impuesta de ser un subtipo de “Animal”.

Para un parámetro de tipo T, se puede definir un tipo concreto S que actúa como cota superior, lo que indica que se debe instanciar a T con el tipo o S o un subtipo de este.

Este tipo de restricción es tal vez de los más comúnmente utilizados, y se expresa de distintas formas en distintos lenguajes. En Scala se usa la sintaxis “T <: S”, mientras que en Java adquiere la forma “T extends S” y en C# “T : S”.

Lower-bound

La cota inferior o lower-bound, aplicada sobre un parámetro de tipo T, con un tipo concreto S, indica que el parámetro de tipo T debe instanciarse con el tipo S o un supertipo del mismo.

Un ejemplo podría ser el siguiente:

Ejemplo 2.20. Pseudo-código mostrando tipos paramétricos con cota superior

```
class Vegano { ... }
class Vegetariano extends Vegano { ... }
class Omnivoro extends Vegetariano { ... }

class Orden<T >: Vegetariano> { ... }

var orden1 : Orden<Vegetariano> := new Orden<Vegetariano>()
        // Funciona porque Vegetariano es exactamente Vegetariano
var orden2 : Orden<Vegano> := new Orden<Vegano>()
        // Funciona porque Vegano es supertipo de Vegetariano
var orden3 : Orden<Omnivoro> := new Orden<Omnivoro>()
        // No funciona porque Omnivoro no es supertipo de Vegetariano
```

En este caso, el tipo “T” solo puede ser instanciado como “Vegetariano” o un supertipo de “Vegetariano”, como “Vegano”. “Omnivoro” no es un tipo adecuado, ya que es subtipo de “Vegetariano”, pero no supertipo.

La realidad es que los casos de uso en el mundo real del uso de cotas inferiores es prácticamente inexistente, salvo por ejemplos forzados o muy específicos.

Sumado a las restricciones impuestas sobre los posibles valores que puede tomar un argumento de tipo, se puede considerar el concepto de subtipos para los tipos paramétricos, ya que aparecen algunos conceptos adicionales a tener en cuenta. Estos conceptos se conocen como **varianza**, e implican tres casos particulares, **invarianza**, **covarianza** y **contravarianza**.

Invarianza

Veamos ejemplos para entender estos conceptos.

Ejemplo 2.16. Pseudo-código mostrando invarianza de tipos paramétricos

```
class Caja<T> {
    // Una caja permite almacenar un único dato para
    // recuperarlo más adelante.
    almacenar(T e) { ... }
    recuperar() { ... } : T
}
var cajaDePerros   : Caja<Perro>  := new Caja<Perro>()
var cajaDeAnimales : Caja<Animal> := cajaDePerros
                                  // No funciona en algunos lenguajes
```

Para entender por qué no funciona el código del ejemplo anterior, pensemos en el tipo “Perro”, y en el tipo “Animal”, y su relación en términos de subtipos. Imaginemos que tenemos una única

caja, creada como caja de perros, pero referenciada de dos formas distintas, como caja de perros, y como caja de animales, ¿Qué podemos hacer con esta última? Bueno, podríamos tomar el contenido de la caja, lo cuál no sería un problema, ya que obtendremos un perro, y un perro es un animal. Entonces, ¿Dónde está el problema? El problema es que también podemos guardar cosas en la caja. Si consideramos la caja de animales, entonces deberíamos poder guardar cualquier dato que sea un animal, por ejemplo, un gato o un conejo. Sin embargo, el objeto en realidad era una caja de perros, no de animales. La primera referencia que teníamos, del objeto como caja de perros, entonces dejaría de tener sentido, porque ya no habría un perro, sino cualquier cosa. Es por eso que este código no funciona en la mayoría de los lenguajes.

Decimos entonces que los parámetros de tipo son **invariantes**. Este concepto de **invarianza** en los tipos implica que la relación de subtipado entre los argumentos de tipo no implica un subtipado en términos de los datos que utilizan dicho tipo como argumento. En el ejemplo anterior, la relación de subtipado en donde “Perro” es subtipo de “Animal”, no hace que haya una relación de subtipado entre “Caja<Perro>” con “Caja<Animal>”.

Covarianza

Pero esto no tiene que ser necesariamente así, pensemos en el siguiente ejemplo:

Ejemplo 2.17. Pseudo-código mostrando covarianza de tipos paramétricos

```
class CajaImutable<+T> {
    // Una caja inmutable es creada con un valor, y este
    // nunca puede ser modificado en el futuro
    constructor(T e) { ... }
    recuperar() { ... } : T
}
var cajaDePerros : CajaInmutable<Perro> := new CajaInmutable<Perro>()
var cajaDeAnimales : CajaInmutable<Animal> := cajaDePerros
                                    // En este caso, sí funciona
```

En esta situación, como la caja es inmutable, la operación que implica modificar el dato, no existe, y por tanto, no hay peligro en tratar una caja de perros como una caja de animales, ya que cada vez que recuperemos el dato, lo que obtenemos si teníamos una caja de perros, será un animal, y por tanto, los tipos coinciden. En este caso el símbolo “+” que utilizamos al momento de declarar el parámetro de tipo, es una forma de indicar que queremos tratar a la caja como **covariante** en T.

Dicho de otra forma, la relación de **covarianza** en un tipo “Cov<+T>” sobre el argumento “A” dice que sí “B” es subtipo de “A”, entonces “Cov<B>” es subtipo de “Cov<A>”. En el ejemplo anterior, “Caja<Perro>” es subtipo de “Caja<Animal>” ya que “Perro” es subtipo de “Animal”.

Pero pensemos en en el caso contrario, sí tengo una caja de animales, ¿Puedo tratarla como caja de perros? La respuesta es que claramente no, ya que sí quiero extraer un animal, no puedo estar seguro de que lo que extraigo es un perro.

Contravarianza

Para solucionar este último tipo de relaciones, en donde quiero poder tratar una caja de animales como caja de perros, pensemos en el caso en donde podemos solamente guardar cosas en la caja, pero nunca recuperarlas.

Ejemplo 2.18. Pseudo-código mostrando contravarianza de tipos paramétricos

```
class CajaEscritura<-T> {
```

```
    // Una caja de solo escritura sirve para escribir, pero
    // el dato nunca puede ser recuperado.
    almacenar(T e) { ... }
}
var cajaDeAnimales : CajaEscritura<Animal> :=
                                        new CajaEscritura<Animal>()
var cajaDePerros : CajaEscritura<Perro> := cajaDeAnimales
```

En este ejemplo, cuando tenemos una caja de animales, podemos tratarla como caja de perros, ya que la única operación permitida es almacenar datos, y cuando almacene un perro, un perro es un animal, y por tanto puede almacenarse.

Decimos entonces que el tipo es **contravariante** en T, expresado por "-" en la declaración del tipo paramétrico. La relación de un tipo "Contra<-T>" para un argumento "A", implica que si "B" es subtipo de "A", entonces "Contra<A>" es subtipo de "Contra<B>". En el ejemplo anterior, como "Perro" es subtipo de "Animal", entonces "CajaEscritura<Animal>" es subtipo de "CajaEscritura<Perro>".

En principio el ejemplo anterior de la caja puede parecer muy poco práctico, pero cuando contamos con cosas como buffers de escritura, o serializadores, aplicar la contravarianza tiene mucho más sentido.

Mutabilidad, inmutabilidad y varianza

Como vemos, las relaciones de varianza tienen mucho que ver con las operaciones finales que pueden realizarse sobre el tipo en cuestión. Cuando los tipos son tanto de lectura y escritura (mutables) en términos de algún dato interno, entonces suele haber invarianza, mientras que los tipos inmutables (sólo lectura) suelen ser covariantes. La contravarianza es un poco más infrecuente, pero igualmente existente en los casos donde solo hay escritura.

Un tipo que suele ser invariante son las listas, porque la lista soporta tanto operaciones para agregar elementos como para recuperar los elementos. Otro ejemplo claro suele ser el concepto de tipo enumerable (IEnumerable en C# o Traversable en Java por ejemplo) en donde solo se leen los elementos de una colección, pero no se los pueden modificar, donde se ve claramente la covarianza. La contravarianza como dijimos, se ve en tipos que representan datos serializables o acciones, como (Action en C#).

Una cosa más a tener en cuenta es que u**n tipo puede ser covariante y contravariante al mismo tiempo**, en cuyo caso decimos que es **bivariante**.

Por otro lado, la notación "+", y "-" que usamos en nuestro pseudo-código es la misma que se utiliza por ejemplo en Scala, pero otros lenguajes tienen otras formas de expresar esta idea. C# por ejemplo, utiliza las palabras claves "out" (covarianza) e "in" (contravarianza). Este tipo de declaración de varianza se realiza al momento de declarar el tipo en cuestión, y se conoce como ***"declaration-site variance***".

En Java en cambio, la varianza no se representa al momento de la declaración del tipo, sino al momento de utilizar el mismo, es decir, cuando se instancia el dato en cuestión, en lo que se conoce como "**use-site variance**". Esto se logra mediante el uso de un "wildcard" y la declaración de un límite de tipos al momento de la declaración de la referencia. Por ejemplo una declaración "Caja<? extends Animal>" en donde se declara una covarianza para un tipo desconocido que contendrá la caja, pero que debe ser subtipo de Animal. Por otro lado, una declaración de "Caja<? super Animal"> define una relación de contravarianza para dicho tipo.

2.4.4. Tipos unión

Pensemos en el siguiente código en JavaScript, cuyo patrón es muy común en lenguajes dinámicamente tipados:

Ejemplo 2.18. Pseudo-código mostrando contravarianza de tipos paramétricos

```
function elemAsString(elem) {
    if (typeof elem == “number”) {
        return (elem.toString());
    }
    else if (typeof elem == “boolean”) {
        return (elem ? “T” : “F”);
    }
    else {
        throw “error de tipos”;
    }
}
```

Podemos observar que tanto los valores que números, como booleanos son aceptados en la función, pero cualquier otro valor no lo es, resultando en un error en tiempo de ejecución con el mensaje “error de tipos”. Si quisiéramos documentar el tipo de “elem”, o pasar este mismo código a un lenguaje con tipado estático y explícito, ¿Qué tipo podríamos decir que tiene?

El problema surge porque “elem” puede ser de dos tipos, que no están vinculados en absoluto entre sí. En algunos lenguajes, uno podría pensar en un tipo “Object” que actúa como supertipo de todos los otros, pero decir que “elem” es de tipo “Object”, implicaría que elementos de otros tipos, como arrays, listas, perros o gatos, son perfectamente válidos, algo que no es cierto.

Se requiere entonces **un tipo unión (union type), que consiste en un tipo compuesto de otros tipos, cuyos valores posibles corresponden a la unión (pensada en términos de conjuntos) de los valores de los tipos de los que está compuesto**. Se define entonces como una “disyunción de tipos”, motivo por el cual en oportunidades se los denomina como **tipos disjuntos** o “***disjoint types***”.

Sí evaluamos el ejemplo anterior, el tipo de “elem” se corresponde a una union entre los tipos “number” y “boolean”. Es decir, “elem” puede tomar cualquier valor, pero dicho valor será o bien de tipo “number” o bien de tipo “boolean”, motivo por el cual se entiende como una disyunción.

Estos tipos en general se definen para casos en donde los valores de los tipos que lo componen son disjuntos, es decir, no hay un valor que pertenezca tanto al tipo “number” como al tipo “boolean” al mismo tiempo. Así, cuando miramos el valor concreto que tiene “elem” en tiempo de ejecución, será o bien “number” o bien “boolean”, pero no ambos. Con tipos más complejos podría darse el caso de que un tipo unión involucre a dos tipos “T” y “S”, y que existan valores que pertenezcan a ambos conjuntos, en cuyo caso pertenecen al tipo unión de ambos, que denominamos comúnmente “T | S”. El código anterior en TypeScript, una versión tipada de JavaScript que admite tipos unión, sería:

Ejemplo 2.18. Pseudo-código mostrando contravarianza de tipos paramétricos

```
function elemAsString(elem : number | boolean) : string {
    if (typeof elem == “number”) {
        return (elem.toString());
    }
    else if (typeof elem == “boolean”) {
```

```
        return (elem ? “T” : “F”);
    }
    // Por tipado estático, “elem” no puede ser otra cosa.
}
```

Se dice que los union types requieren entonces “uno de entrada, pero todos de salida”, es decir, el dato va a ser de un único tipo, pero necesitamos manejar todos los posibles casos de tipos que el dato pueda tener para no tener errores de tipos.

En principio uno podría pensar que son pocos los casos en donde se necesita una característica como esta, pero la realidad es que es mucho más común de lo esperado. Por otro lado, los casos interesantes surgen cuando se utilizan tipos estructurados o con comportamiento como componentes de tipos unión. Para entender estos escenarios analizaremos otro ejemplo en TypeScript.

Ejemplo 2.18. Pseudo-código mostrando contravarianza de tipos paramétricos

```
type NetworkLoadingState = { state: "loading" }
type NetworkFailedState  = { state: "failed", code: number }
type NetworkSuccessState = { state: "success", response: { ... } }
type NetworkState =
    NetworkLoadingState | NetworkFailedState | NetworkSuccessState
```

Imaginemos entonces ahora que tenemos un dato cuyo tipo es “NetworkState”. Por tratarse de un tipo unión, sabemos que el valor concreto del dato puede pertenecer o bien a “NetworkLoadingState”, o bien a “NetworkFailedState” o bien a “NetworkSuccess”. Pero cuando analizamos, independientemente del valor concreto que tome el dato, podemos observar que todos comparten algo en común, el atributo “state”.

Por tanto, podemos asegurar que solicitar a un dato de tipo “NetworkState” su atributo “state” es una operación segura, pues sin importar cual sea el objeto, sabe responder a dicho mensaje. Por otro lado, los atributos “code” y “response” sólo pueden ser utilizados una vez que se está seguro el tipo concreto que tiene el dato.

Los lenguajes que proveen union types suelen proveer alguna forma de discriminar el tipo del elemento en cuestión, por ejemplo, a través de **pattern matching** (Haskell, Scala), o mediante la discriminación mediante una función que describa el tipo, como “typeof” (JavaScript) o algun campo compartido con valores fijos (JavaScript, TypeScript). Esto permite entonces castear de forma segura los valores a su tipo concreto y poder usar las operaciones correspondientes.

Volviendo a las operaciones que son posibles utilizar sobre un tipo unión, podemos decir entonces que se corresponden a aquellas operaciones que son posibles realizar sobre cualquiera de los tipos que componen al tipo unión. Sí pensamos en objetos, podemos decir que la interfaz del tipo unión se corresponde a la intersección de las interfaces más específica de cada uno de los tipos que componen la unión.

Los lenguajes con tipado en runtime tienen naturalmente union types, ya que al no declarar los tipos, el tipo de un elemento queda a discreción del programador, en base al uso, aunque se debe poder discriminar el tipo concreto para poder operar adecuadamente.

2.4.5. Tipos intersección

Los **tipos intersección**, o **intersection types** son el opuesto a los tipos unión. Consisten en **declarar que un elemento es de todos los tipos que lo componen al mismo tiempo**.

Esto tiene sentido solo en los casos donde los tipos son tipos con estructura (como en el caso de los objetos) y no con tipos simples, ya que no tiene sentido decir que algo es un string y un número al mismo tiempo. Podemos analizar un ejemplo en TypeScript:

Ejemplo 2.18. Pseudo-código mostrando contravarianza de tipos paramétricos

```
type Mamifero = { mamar: () -> () }
type AnimalMarino = { nadar: (kms: number) -> () }
type Ballena = Mamifero & AnimalMarino
```

En este caso, el tipo “Ballena” es un tipo intersección, es decir, los objetos de ese tipo pueden tanto “mamar” como “nadar”. Esto es algo viable en TypeScript donde la declaración de tipos no se produce únicamente a través de clases, ni los objetos pueden ser creados únicamente mediante estas. Es decir, definir un objeto mamífero y marino es perfectamente realizable y simple en dicho lenguaje, teniendo entonces tipo “Ballena”.

Desde un punto de vista de los valores, un dato del tipo intersección compuesto por dos tipos “T” y “S”, debe ser un elemento que pertenezca tanto al conjunto de los valores de “T” como al de los de “S”. Por lo tanto, los conjuntos no pueden ser disjuntos, ya que no habría valores que satisfagan la condición.

Ahora bien, desde un punto de vista de las operaciones, puedo realizar todas las operaciones que podría realizar en cada uno de los tipos. Es decir que en términos de operaciones, la interfaz de un tipo intersección consiste en la conjunción de las interfaces de cada uno de los tipos. Por este motivo muchas veces se los llama también “**tipos de conjunción**” o “***conjunction types***”.

Los casos de uso reales de los tipos conjunción son escasos y poco frecuentes, pero sin embargo existen casos puntuales en donde son necesarios o donde su existencia facilita el código resultante. Así, hay lenguajes que implementan este tipo de declaración.

2.4.2. Tipos nullificables

Uno de los conceptos que aparece en muchos lenguajes de programación es el concepto de “null” o “nil”, llamado **referencia nula**, y utilizado para describir la idea de que una referencia aún no apunta a un objeto. Este concepto resulta útil para expresar la idea de “no valor” de forma relativamente simple y sin complejizar los tipos manipulados con ideas de tipos envoltorios, como “Maybe” en Haskell, u “Option” en Scala.

La implementación concreta de la referencia nula suele estar aplicada de dos formas diferentes. **En la primera, “null” es un concepto separado del universo de los valores, y no consiste en un valor primitivo o un objeto**. En este caso, es un concepto que aplica a las referencias, de forma pura y exclusiva, y por tanto podemos distinguir aquellas referencias que no están inicializadas (referencias nulas) a aquellas que sí están inicializadas (referencias a objetos). **En el segundo caso, “null” es un valor único en el sistema**, que es equivalente a otros valores, como “42”, “true” o “3.14”. Es decir, “null” es un valor que puede asignarse a una referencia, así como se asigna cualquier valor a una referencia.

Comenzaremos por analizar el concepto de “null” como un concepto que solo aplica a las referencias. Esta visión suele predominar principalmente en lenguajes que permiten desacoplar la declaración de una variable de su inicialización (asignación del valor inicial). Un ejemplo sería:

Ejemplo 2.18. Pseudo-código mostrando “null” como la idea de referencia no inicializada

```
var x : String
... // Durante el código entre la declaración y
... // la primer asignación, “x” no está inicializada,
... // por lo que se considera “null”.
x := “Hola mundo”
...
if (defined x) {
    ...
}
```

En estos casos, “null” no aparece casi nunca escrito de forma literal. Como no es un valor, no se puede comparar una variable con “null”, ni tampoco puede usarse “null” como argumento en una invocación. En estos casos se requiere algún mecanismo para determinar si una variable fue o no inicializada, como es el caso de “defined” elegida en nuestro pseudocódigo. Lenguajes como PHP hacen algo similar. Otros, conceptualizan a “null” de esta forma, pero aún así permiten escribir “null” en algunos lugares especiales, como comparaciones o argumentos.

Otros lenguajes, piensan a “null” como valor, que puede escribirse y pasarse como argumento,. Pero si lo pensamos como valor, nos puede surgir entonces una pregunta clave, ¿De qué tipo es ese valor? Y allí aparece la idea de **tipo nulo**. El tipo nulo es aquel tipo que tiene el valor “null” o equivalente, según el lenguaje, y puede tener a su vez, dos interpretaciones distintas. En la primera, muy común, **el tipo nulo es un subtipo de cualquier otro tipo que exista en el sistema**. En ese sentido, el siguiente código es perfectamente aceptable:

Ejemplo 2.18. Pseudo-código mostrando asignación de “null” por ser subtipo

```
var x : String := NULL
var y : Perro := NULL
var z : Pila<Animal> := NULL
```

En todos los casos, “null” puede ser asignado a la variable, mediante “**upcasting**”, ya que es un subtipo del tipo de la referencia. Este código también podría aplicar cuando se piensa en “null” como referencias, como una forma de expresar que la referencia no está inicializada, aunque generalmente en esta visión, sería raro escribir literalmente el valor “null”.

Cuando se conceptualiza el tipo nulo como un subtipo de cualquier otro, surge el problema de que todo dato puede ser entonces “null”. Es decir, no hay una forma de indicar que algunos datos deben forzosamente estar inicializados en un punto del programa. Otra visión entonces, que soluciona dicho problema, es la de pensar el tipo nulo como un tipo independiente completamente separado del resto. En esta visión asignar “null” a una variable que declara ser un “string” no es correcto, ya que no es un valor de un subtipo de “string”. Entonces se debe definir la variable como perteneciente a un tipo unión.

Ejemplo 2.18. Pseudo-código mostrando asignación de null mediante tipos unión.

```
var x : String := NULL // Error: NULL no es subtipo de String.
var y : String | NullType := NULL
      // Bien, porque al ser un tipo unión entre String o
      // NullType, puede elegir cualquier valor de uno
      // u otro conjunto, y NULL pertenece al tipo nulo.
var z : String? := NULL
```

```
      // ? suele ser una sintaxis para representar
      // lo mismo que el tipo unión anterior.
```

Sin embargo, como es un caso tan común, muchos lenguajes ofrecen alguna sintaxis simple para el caso de la unión entre un tipo y el tipo nulo (el cual a veces no tiene siquiera un nombre formal en el lenguaje). Dependiendo del lenguaje, por supuesto, la sintaxis puede variar, pero es común encontrar la sintaxis del nombre del tipo seguido de un signo de pregunta, como el usado en el último caso.

Se habla entonces de **tipos "nullificables**", es decir, "String" y "String?" son dos "versiones" del mismo tipo, la primera "no nullificable" y la segunda "nullificable". En ese sentido, "String" solo puede tomar valores del tipo "String", y "null" no es un valor que esté incluido. Por otro lado "String?" puede tomar cualquier valor de "String" y también el valor "null".

No existe entonces un "tipo nullificable", sino que un tipo ya existente cualquiera puede "volverse nullificable". Por ejemplo, sí existe un tipo "T" que es "no nullificable", (no incluye al valor "null" entre todos los posibles valores que puede tomar) decimos que existe entonces otro tipo, "T?" que es "nulificable" y por tanto puede tomar cualquier valor del tipo "T", y adicionalmente el valor "null". Sin embargo, vale aclarar que en la mayoría de los lenguajes, nada impide declarar tipos nominales "nullificables", es decir, un tipo "S" que ya incluya dentro de otro valores al valor "null", a pesar de que su nombre no finalice con un signo de pregunta.

Otra cosa interesante a tener en cuenta tiene que ver con el subtipado en el caso de un tipo "T", y su versión nullificable "T?", donde esta última es entonces supertipo de la primera. Así, podemos usar un valor de tipo "T" en donde se espera un valor de tipo "T?", pero no a la inversa.

Ejemplo 2.18. Pseudo-código mostrando asignación de null mediante tipos unión.

```
var noNulo     : String  := "Hola mundo"
var capazNulo  : String? := noNulo
      // Funciona, String es subtipo de String?
var otroNoNulo : String  := capazNulo
      // No funciona, String? no es subtipo de String.
...
if (capazNulo /= NULL) {
    var otroNoNulo : String  := capazNulo
    // Funciona, pues en este punto del programa podemos
    // estar seguros de que capazNulo no es NULL, y por
    // tanto se puede castear automáticamente a String.
}
...
```

Para pasar entonces de un tipo nullificable a su versión no nullificable, debe existir un proceso de casteo. Esto ocurre de forma manual, aunque muchos lenguajes que tienen tipos nullificables son capaces además de detectar sí en un punto de programa particular, una referencia nullificable ya fue analizada previamente por su valor, en una alternativa condicional o similar, dando lugar a un proceso de casteo automático en esos casos.

Los tipos nulificables permiten entonces un mejor control de errores, al permitir detectar mediante el typechecker usos de valores no inicializados o nulos. Por supuesto que no todos los casos pueden ser detectados, pero en los lugares en donde se desea asegurarse que una referencia seguro tiene un valor no nulo, pueden utilizarse tipos no nulificables.

2.4.3. Tipos dinámicos o mixtos

Algunos lenguajes tienen soporte para lo que se llaman **tipos dinámicos** ("**any types**"), o también **tipos mixeados** ("***mixed types***"). En estos casos se menciona a una referencia como de un tipo bajo el nombre de "any" o "mixed", que **resulta equivalente a decir que dicha referencia puede ser de cualquier tipo**.

Uno podría pensar en que cuando decimos que algo es un "Object", en lenguajes como Java, estamos diciendo que es de "cualquier tipo", en el mismo sentido, pero la realidad es que los tipos "any" representan otra cosa, por lo que no son equivalentes. Cuando decimos que un dato es de tipo "Object", estamos diciendo que puede ser de cualquier tipo, gracias al subtipado, sí, pero también estamos diciéndole al typechecker que las únicas operaciones que espero hacer sobre ese dato son aquellas que corresponden a "Object". De querer realizar otras operaciones, es necesario realizar un proceso de casteo, verificando previamente el tipo de dato, ya que desconocemos qué otras operaciones podría tener el objeto a priori.

En ese sentido, **un tipo "any" actúa como una forma de indicarle al typechecker que no queremos que realice validaciones de tipo en tiempo de compilación, sino, en tiempo de ejecución únicamente**. A un objeto de tipo "any" le podemos mandar entonces cualquier mensaje, y el typechecker no va a dar error. Por supuesto, nada impide que el mensaje falle en tiempo de ejecución.

Los tipos any permiten tener el dinamismo de un lenguaje con chequeo de tipos dinámico, en uno que realiza verificaciones estáticas. Más aún, este concepto permite realizar lo que se conoce como **tipado incremental** (***gradual-typing***). Con gradual-typing algunas definiciones tendrán información de tipos, la cual se usará para realizar una verificación en tiempo de compilación, mientras que otras no tendrán información, recayendo la verificación al proceso de ejecución. La mayoría de los lenguajes que permiten gradual-typing asignan automáticamente el tipo "any" a toda definición que no declare explícitamente los tipos, o sí tienen un sistema con tipado explícito, proveen el tipo "any" para que pueda ser usado en cualquier definición.

Un ejemplo de lenguaje que soporta el tipado incremental es TypeScript, que actúa como un superset de JavaScript (es decir, todo programa JavaScript es un programa TypeScript). En ese sentido, cualquier programa de JavaScript puede ser ejecutado en TypeScript sin modificaciones, ya que toda definición se percibe como "any". Así, portar una aplicación de un lenguaje a otro es trivial. Luego, se puede ir agregando la información explícita de tipos específicos para que el compilador haga una verificación estática de los mismos, y detectar posibles errores, lo cual puede realizarse en sucesivas etapas y lentamente.

2.3. Análisis simples de sistemas de tipos

Muchas veces escuchamos catalogaciones de los sistemas de tipos de un lenguaje, como decir "el lenguaje X es un lenguaje con tipado estático", pero esto suele ser solo una catalogación parcial, o mezclar conceptos como "tipos explícitos" con "tipado estático". Lo que quisiéramos es una forma de poder catalogar los sistemas de tipos que sea más precisa, y que deje claras todas las implicancias que ese sistema de tipos tiene. Para ello, vamos a hacer uso de los tres ejes que ya mencionamos antes:

- Momento de la verificación
- Declaratividad de los tipos
- Identificación de los tipos

En base a estos criterios, vamos a poder expresar qué características tiene el sistema de tipos de un lenguaje particular. Pero cuidado, no hay que pensar que podemos poner una escala numérica en cada uno de esos elementos, por ejemplo, decir que solo se verifica al momento de la

compilación o solo en la ejecución, pues hay casos en donde ocurre en ambas, lo mismo vale para la identificación, o para la declarativid.

Vamos a ver un par de ejemplos

2.3.1. Análisis del sistema de tipos de Java

Java es un lenguaje cuya sintaxis demanda escribir los tipos en prácticamente todos lados, y la inferencia disponible es escasa o nula (dependiendo de la versión del lenguaje). Estas anotaciones sirven para realizar un análisis estático de los tipos, que ocurre en la etapa de compilación. Sin embargo, en el caso de los “casteos”, o cuando se usan caracteristicas avanzadas como “reflection”, también se realiza una verificación en tiempo de ejecución. La única forma en Java para mencionar un tipo, es mediante nombre, no soportando tipados estructurados.

Bibliografía y enlaces útiles

- Benjamin C. Pierce. **Types and programming languages**. MIT Press, 2002.
- K.B. Bruce. **Foundations of Object-Oriented Languages: Types and Semantics**. MIT Press, 2002.
- TypeScript Handbook, especialmente leer secciones “Everyday Types”, “Object Types” y “Type Manipulation”, https://www.typescriptlang.org/docs/handbook/intro.html