Eva López Puente
Jaime Urquiza Fuentes

Ayuda al Aprendizaje de la Comprobación de Tipos mediante una API de Desarrollo

Número 2021-01

Serie de Informes Técnicos DLSI1-URJC
ISSN 1988-8074
Departamento de Lenguajes y Sistemas Informáticos I
Universidad Rey Juan Carlos

Índice

Ayuda al Aprendizaje de la Comprobación de Tipos mediante una API de Desarrollo

Eva López Puente[1][0000−0002−9494−5851] and Jaime Urquiza Fuentes[2][0000−0002−4101−7023]

Universidad Rey Juan Carlos, Madrid

Abstract. La enseñanza de compiladores en la universidad debe afrontar diferentes dificultades, entre ellas el nivel de abstracción de muchos de los conceptos que se manejan y el esfuerzo de desarrollo por parte de los estudiantes a la hora de trabajar en los proyectos propuestos en este tipo de asignaturas. En este trabajo se presenta una API dedicada al desarrollo de sistemas de tipos dentro del marco de una asignatura de compiladores. Por un lado, la API se centra en facilitar la manipulación de conceptos abstractos como las expresiones de tipo y la comprobación de su equivalencia. Por otro, se ha diseñado teniendo en cuenta diferentes paradigmas de programación, de forma que se puedan desarrollar sistemas de tipo para programación imperativa, lógica, funcional u orientada a objetos. Con esta API se pretende facilitar por parte de los estudiantes el desarrollo de proyectos de compiladores, manteniendo la necesidad de manipular conceptos como las expresiones de tipo, pero facilitando su desarrollo al proporcionar la infraestructura necesaria para dicha manipulación.

Keywords: API · Expresiones · Tipo.

1 Paradigmas de programación

Un paradigma de programación [1] nos indica un método para realizar cómputos y la manera en que se deben estructurar y organizar las tareas que debe llevar a cabo un programa, afectando a las construcciones más básicas de los mismos. Así nos encontramos con diferentes enfoques:

Imperativo Nos describe a través de una secuencia de acciones (sentencias de secuencia) realizadas a través de subprogramas y ejecutadas según un control de flujo, lo que modificará el estado del programa.

Declarativo Nos indica la lógica de computación necesaria para resolver un problema, pero no indica el flujo de control que se debe seguir, es decir, no es necesario definir algoritmos previos, ya que no existe una sentencia de asignación y el control suele estar asociado a la composición funcional, recursividad y/o técnicas de reescritura y unificación. En este paradigma podemos diferenciar, a su vez, entre programas funcionales o lógicos.

Orientado a objetos Describe la secuencia que debe seguir un programa para resolver un problema dado, en este caso, la programación orientada a objetos hace uso de estructuras de datos que contienen propiedades y métodos que permiten interactuar entre ellas.

2 Sistemas de tipos

Un sistema de tipos [2] en un lenguaje indica cómo este clasifica los valores y las expresiones y cómo se pueden manipular e interactuar con ellos.

2.1 Tipos de datos

Un tipo de dato es el conjunto de términos de un lenguaje que tiene unas características comunes que les permite interactuar entre sí o ser objeto de transformaciones sólo aplicables a ellos mismos. Los tipos en cuya expresión no aparecen variables se conocen como monotipo, por el contrario, si las variables sí aparecen tendríamos los tipos polimórficos.

Imperativo En este paradigma, nos encontramos por ejemplo, el lenguaje Pascal [3]. Donde hace uso de variables con valores modificables (lógicos, numeros, caracteres...), tipos valor o tipos de referencia, y tiene procedimientos y funciones que nos permite dividir el programa en bloques sencillos consiguiendo que sea más legible y comprensible por todos, tanto el programador inicial como aquellos que tengan que realizar modificaciones de este.

Declarativo Funcional En la programación funcional, como el lenguaje Haskell [4], tenemos variables que pueden cambiar de valor (valores lógicos, números, cadenas...), tipos algebraicos donde el usuario necesita una estructura no predefinida y la crea a partir de constructores de datos, así como listas y árboles.

Declarativo Lógico En el paradigma declarativo lógico, tenemos un lenguaje como Pascal [5], donde es un conjunto finito de fórmulas lógicas que reflejan el conocimiento del que se dispone para resolver un problema. Así, constaría de variables, constantes, listas, hechos y reglas para hallar la solución a la pregunta.

Orientado a objetos En este paradigma, un lenguaje como Java [6], donde tendríamos tipos primitivos, que no necesitan invocación para ser creados; tipos objetos, proporcionados por las bibliotecas, arrays o los equivalentes a los primitivos pero en objeto; y, tipos abstractos de datos, es decir, creados por el usuario.

3 Implementación

En cuanto a la implementación de dicha API, nos enfrentamos a una clase común de tipos en la que se podrá almacenar el nombre y el resultado booleano de la comparación entre dos tipos. A través de esta clase se realizará la división de tipos en básicos y complejos (ver Fig. 1).

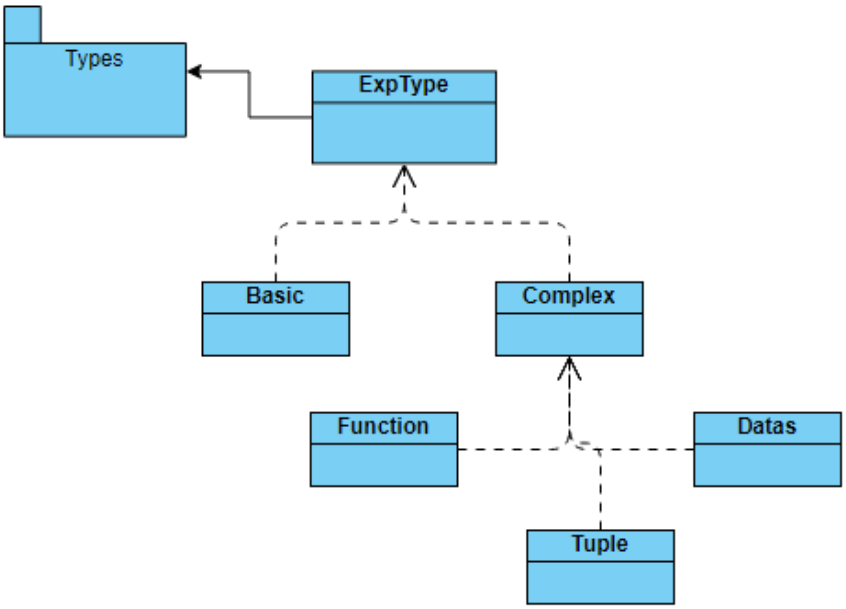

Fig. 1. Diagrama de clases del paquete tipos

3.1 Tipos básicos

En los tipos básicos se creará un tipo por su nombre *"ExpType createBasic (String name)"* y se comparará con otro tipo a través del nombre y el contenido almacenado, devolviendo valor true o false, según sean comparables o no, *"boolean compareType (ExpType e1, ExpType e2)"*.

3.2 Tipos complejos

En los tipos complejos, el contenido almacenado de una expresión de tipo se coloca en un array, puesto que serán funciones, pares de elementos o tuplas y estructuras de datos, y no se pueden almacenar como tipos básicos, asimismo, tendrá el método de comparación para conocer si dos expresiones son comparables o no, en este caso, el método sí es el mismo que en el apartado para crear tipos básicos. Una función está formada por los elementos que recibe como argumentos y por aquello que debe devolver, por tanto, necesitamos una estructura que nos permita guardar dos expresiones de tipo, en este caso, hemos optado por un array de dos elementos, siendo el primero los argumentos y el segundo aquello que devuelve la función, *"ExpType createFunction (ExpType e1, ExpType e2)"*. Para el caso de las tuplas, se almacenarán, de nuevo en un array de dos elementos, las expresiones de tipo. Si fuera necesario almacenar más de dos expresiones, éstas se concatenan con la anterior, es decir, en el primer hueco del

array irían las dos 'antiguas' mientras que en el segundo aparecería la nueva, *ExpType createTuple (ExpType e1, ExpType e2)"*. Por último, las estructuras de datos, donde no se conoce de antemano qué es lo que el usuario va a almacenar, por tanto, en vez de almacenar las expresiones en un array de tamaño definido, esta vez será una lista dinámica de expresiones de tipo y se añadirán a través del método *add* de las listas, *"Datas createDatas (String name)"*, se proporciona un nombre a la estructura y dentro tendrá la lista de expresiones.

3.3 Programación lógica

La programación lógica es un poco especial puesto que no se pueden comparar expresiones, es por ello, que tenemos unas nuevas clases donde se almacenarán las expresiones de los hechos y las reglas. Para crear un hehco, primero se crearán los objetos *"ExpType createObject (String name)"* que forman parte de ese hecho y después se les asignará una relación en el hecho, *ExpType createFact (String name, ExpType e1)"*. En el caso de las reglas, nuestro array será de dos elementos, donde el primero corresponde al hecho de la parte izquierda de una regla, la cabeza, mientras que el segundo serán todos los hechos que indican que esa regla es cierta, la parte derecha de la misma o cuerpo, *ExpType createRule (ExpType e1, ExpType e2)"*.

4 Conclusiones y métricas

El diseño de una API abarca decisiones que van desde su arquitectura y funcionalidad hasta el nombre específico de clases, funciones o métodos. A través del artículo de Myers y Stylos [7], indican que las dos cualidades básicas que debe tener una API son usabilidad (qué tan fácil es aprenderla y usarla) y potencia (extensibilidad a criterios específicos de los usuarios o capacidad de evolución para crear nuevas versiones por otros desarrolladores). Asimismo, proponen una manera sencilla de evaluar el diseño a través de diez pautas de evaluación heurística de Nielsen. Nuestra API cumple siete de estas diez pautas, descartando visibilidad e información del sistema ya que no podemos aportar en qué estado de ejecución o comprobación se encuentra; el control y libertad del usuario, dado que, en principio, el usuario no tiene acceso al código; y por último, la recuperación de errores. Sin embargo, de las pautas restantes podemos decir que están cumplidas porque: los nombres de los métodos y su organización en clases coinciden con las expectativas de los usuarios, indicando nombres genéricos y conocidos; todas las partes de la API son coherentes a través de los nombres identificativos y semejanza en el uso; se ayuda al usuario haciendo que la API funcione correctamente, por ejemplo, con valores predeterminados o errores de escritura; es indispensable que los nombres sean claros y comprensibles, no demasiado largos y que no exista un mismo método repetido varias veces con el único cambio en el número de argumentos, así se puede utilizar la ventana emergente de autocompletar en los entornos de programación y se escribe poco para que lo reconozca; las tareas se realizan de manera eficiente consultando al mínimo

el manual de ayuda, debido a que el usuario apenas tiene funciones que recordar; por último, es posible utilizar la API a través de un manual que se proporciona junto con el ejecutable de la misma.

4.1 Métricas

Finalmente, como conclusión a este trabajo se expone que se ha utilizado un ejercicio con y sin la ayuda de esta API. Si comparamos las líneas de código del archivo principal del ejercicio (main.java) este tiene 786 líneas mientras que, si utilizamos la aplicación desarrollada, las líneas de código disminuyen a 778. Es una reducción mínima de ocho líneas, lo que a simple vista puede ser insignificante, pero, si nos basamos en la funcionalidad, se ha dado al usuario la posibilidad de crear punteros, arrays y mapas. Por tanto, son solo ocho líneas las que nos permiten aumentar la capacidad de la práctica inicial. Otro enfoque, podrían ser las mejoras que se podrían aplicar a esta API, donde se podría dar al usuario la capacidad de mostrar el código según el programador que lo vaya a desarrollar, es decir, poder cambiar la salida final de los métodos que vaya a utilizar y sus estructuras, y poder ampliarla desarrollando otros conceptos relacionados con compiladores, como la generación y representación de código intermedio.

References

1. Gabrielli Maurizio, Martini Simone: Programming Languages: Principles and paradigms. Springer (2010)
2. Aho, Lam, Sethi, Ullman.: Compiladores: Principios, técnicas y herramientas. Pearson (2008)
3. Joyanes Aguilar, Luis: Programación en Pascal. McGraw Hill (2006)
4. Ruiz Jiménez, Blas Carlos: Razonando con Haskell: Un curso sobre programación funcional. Thompson (2004)
5. Sterling, Leon: The Art of Prolog. MIT Press (1999)
6. Arnold, Ken: El lenguaje de programación en Java. Addison-Wesley (1997)
7. Myers, Brad A., Stylos, Jeffrey: Improving API Usability. Communications of the ACM 59(6):62-69 (2016)