Actas de las Jenui, vol. 6. 2021.
Páginas: 279-282

Ayuda al aprendizaje de la comprobación de tipos mediante una API de desarrollo

Eva López Puente
Departamento de Informática y Estadística
Universidad Rey Juan Carlos
Móstoles
e.lopezpu@alumnos.urjc.es

Jaime Urquiza Fuentes
Departamento de Informática y Estadística
Universidad Rey Juan Carlos
Móstoles
jaime.urquiza@urjc.es

Resumen

La enseñanza de compiladores debe afrontar diferentes dificultades, entre ellas el nivel de abstracción 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. Este trabajo presenta una API dedicada al desarrollo de sistemas de tipos. No evita que los estudiantes afronten la complejidad del desarrollo de un sistema de tipos, pero facilita la tarea ofreciendo una infraestructura básica para la manipulación de expresiones de tipo contemplando diversos paradigmas de programación. Aunque este trabajo sigue en progreso, se ha realizado una evaluación preliminar con un enfoque analítico que ha dado resultados prometedores.

Abstract

The teaching of compilers must face different difficulties, among them the level of abstraction of the concepts that are handled and the development effort on the part of the students when working on the projects proposed in this type of subjects. This paper presents an API dedicated to the development of type systems. It does not prevent students from facing the complexity of developing a type system, but it does facilitate the task by offering a basic infrastructure for manipulating type expressions by contemplating various programming paradigms. Although this work is still in progress, a preliminary evaluation with an analytical approach has been carried out and has given promising results.

Palabras clave

Enseñanza de compiladores, Prácticas, Sistemas de tipo, API de desarrollo.

1. Introducción

Compiladores es una asignatura compleja debido fundamentalmente al grado de abstracción de los conceptos que maneja [3]. Este trabajo se enmarca en una asignatura de máster de 6 ECTS (4 horas semanales durante un cuatrimestre) donde la primera mitad se dedica a compiladores y la segunda mitad a intérpretes y máquinas virtuales. El temario dedicado a compiladores supone conocimientos previos de procesadores de lenguajes hasta los fundamentos de traducción dirigida por la sintaxis (comúnmente impartidos en los grados de informática) y trata los temas de tabla de símbolos, comprobación de tipos, generación de código intermedio, optimización de código y generación de código final. Asociado a los temas de tabla de símbolos y comprobación de tipos existe una práctica sobre sistemas de tipos cuyo desarrollo lleva toda la mitad del cuatrimestre.

La complejidad de esta asignatura ha motivado el desarrollo de un buen número de herramientas para la docencia. Los ejemplos más significativos son las herrameinatas generadoras de parsers, que a pesar de tener una clara utilidad en el mundo profesional también lo son en la enseñanza. Así, se pueden encontrar generadores de analizadores sintácticos ascendentes como CUP[1], descendentes como Coco[2] o generadores de analizadores léxicos como Flex[3]. Pero también hay otro tipò de herramientas, algunas más centradas en los fundamentos teóricos como JFLAP [5], que permite experimentar de forma visual e interactiva conceptos de autómatas y lenguajes formales o algo más prácticos como VAST[2], que está centrada la visualización de árboles sintácticos y su proceso de construcción.

Aun así, no se ha podido encontrar nada desarrollado más relacionado con el objeto de este trabajo, centrado en facilitar el aprendizaje de la parte de comprobación de tipos mediante un API específica que propor-

[1] http://www2.cs.tum.edu/projects/cup, 2021
[2] http://ssw.jku.at/coco, 2021
[3] https://github.com/westes/flex, 2021

cione a los estudiantes una infraestrcutura que ahorre tiempo de trabajo sin que dejen de encarar todos los retos que supone aprender a manejar y construir sistemas de tipos para diversos paradigmas.

2. Sobre los sistemas de tipos

El sistema de tipos [1] de un lenguaje comprueba el correcto uso, desde el punto de vista semántico, de las construcciones del lenguaje. Así, indica cómo se clasifican los valores, las expresiones y cómo se pueden manipular e interactuar con ellos. Un tipo de dato es el conjunto de términos del lenguaje que posee unas características comunes que les permite interactuar entre sí o ser objeto de transformaciones solo aplicables a ellos mismos. Así, podemos diferenciar los tipos de datos en cada uno de los siguientes cuatro paradigmas: imperativo, declarativo funcional, declarativo lógico y orientado a objetos.

El paradigma imperativo hace uso de variables con valores modificables (lógicos, números, 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. Un lenguaje ejemplo sería *Pascal*.

El paradigma funcional tiene tipos que pueden cambiar de valor (valores lógicos, números, cadenas...), tipos algebraicos donde el usuario necesita una estructura no definida y la crea a partir de constructores de datos, así como listas y árboles. Un lenguaje ejemplo sería *Haskell*.

En el paradigma lógico se define 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. Un lenguaje ejemplo sería *Prolog*.

El paradigma orientado a objetos tiene tipos primitivos, que no necesitan invocación para ser creados y tipos objetos, proporcionados por la bibliotecas. También se tienen arrays y tipos abstractos de datos, estos últimos creados por el usuario. Un lenguaje ejemplo sería *Java*.

3. Descripción de la API

El objetivo de este trabajo es desarrollar una API Java –lenguaje utilizado en la asignatura– que ayude a los estudiantes en el desarrollo de sistemas de tipo. Dicha API se basa en una jerarquía de clases (véase la figura 1) que representa los diferentes tipos que podrían ser necesarios manejar. Comienza con una clase común de

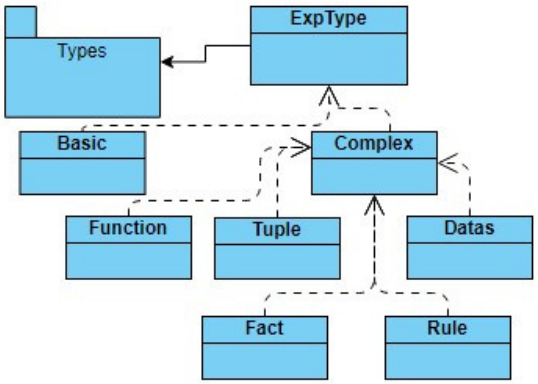

Figura 1: Diagrama de clases del paquete *Types* dedicado a los tipos.

tipos (*ExpType*) 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 en tipos básicos (*Basic*) y complejos (*Complex*).

Los tipos básicos se crearán fácilmente utilizando su nombre *ExpType createBasic (String name)* y se comparará con otro tipo a través del nombre y el contenido almacenado, devolviendo *true* o *false*, según sean comparables o no, *boolean compareType (ExpType e1, ExpType e2)*.

En los tipos complejos, el contenido almacenado de una expresión de tipo se coloca en un array, puesto que serán funciones (clase *Function*), pares de elementos o tuplas (clase *Tuple*) y estructuras de datos (clase *Datas*), 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 el tipo devuelto, 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.

Con todo lo aportado anteriormente se cubren los paradigmas imperativo, declarativo funcional y orientado a objetos. Sin embargo, la programación lógica es un poco especial y se han creado nuevas clases que permitan manipular sus elemtos fundamentales: los hechos y las reglas. Para crear un hecho, 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. Ejemplos de uso

En este apartado vamos a dar unos ejemplos de cómo se utilizaría esta API y las salidas que nos proporcionaría. En todos los paradigmas, lo primero que hay que hacer es crear un objeto de aquel que se va a utilizar, *XParadigm pX = new XParadigm();*, donde 'X' se sustituye por el nombre del paradigma. Una vez tenemos el paradigma, se procede a crear tipos básicos o complejos, según se necesiten. Además la API proporciona una salida textual a modo de log para depuración informando sobre las acciones realizadas. En la figura 2 se puede ver un fragmento de la especificación de un parser utilizando la API para realizar labores de comprobación de tipos. Usaremos esta figura para los ejemplos descritos a continuación.

4.1. Crear un tipo básico

Para crear un tipo básico basta con darle un nombre y asignarlo a una expresión de tipo para que después se pueda usar, por ejemplo, si queremos crear un tipo entero o un tipo en coma flotante, sería *ExpType t0 = pI.createBasic('int');', 'ExpType t1 = pI.createBasic('float');*, respectivamente, donde 'pI', indica el paradigma que se está utilizando, en este caso, el paradigma imperativo. La salida textual mostraría *BasicType : int, BasicType : float*. Su uso se puede ver en las líneas 1-3 de la figura 2.

4.2. Crear un tipo complejo

Un tipo complejo, como se ha dicho anteriormente, sería una función. Supongamos que tenemos una función que recibe un valor entero y nos devuelve un valor en coma flotante, primero tendríamos que crear

```
1  value returns [ExpType t] :
2      NUM_INT_CONST  {$t = pI.createBasic('int');}
3      | NUM_REAL_CONST {$t = pI.createBasic('float');};
4
5  dclfun returns [ExpType t] :
6      type IDENT '(' formalparams ')' ';' {
7          $t = pI.createFunction($type.t, $formalparams.t);
8          // ST es el objeto de la tabla de símbolos
9          ST.insertType($IDENT.text, $t);
10     };
11
12 dclstruct returns [ExpType t] :
13     'struct' IDENT {
14         Datas tipo = pI.createDatas($IDENT.text);
15     }
16     '{' dclvarlist[tipo] '}' ';' {
17         $t = tipo;
18     };
19
20 dclvarlist[ExpType t] :
21     type IDENT ';' {
22         (Datas)$t.addElement($type.t);
23     } dclvarlist[$t]
24     |
25     ;
26
27 sent :
28     IDENT '=' exp {
29         ExpType t1 = ST.getType($IDENT.text);
30         if(!pI.compareTypes(t1, $exp.t)){
31             // Tratar error de tipos ...
32         }
33     };
```

Figura 2: Ejemplo especificación de parser con llamadas a la API para comprobación de tipos

los tipos básicos para el entero (t0) y la coma flotante (t1) y después crear la función, la cual se crearía como *ExpType t2 = pI.createFunction(t0, t1);*. Su salida se mostraría como *ComplexType : Function : (int) ->(float)*. La utilización de estos tipos se puede ver en las líneas 5-10 de la figura 2 donde se declara una función.

4.3. Crear una estructura

Al igual que ocurre con las funciones o las tuplas, primero es necesario crear los objetos básicos que forman la estructura, para después poder añadirlos. Así, si queremos una estructura formada por tres colores (rojo, verde y azul), primero crearemos los básicos asociados. A continuación, la estructura que los almacena, *Datas color = pF.createDatas('dataColor');*, y después, añadirle a esa estructura o lista los tipos básicos, *color.addElement(rojo)*, así con cada uno de ellos. Una vez está creado, se mostraría al usuario como *ComplexType : dataColor (Rojo, Verde, Azul)*. El uso de estas estructuras se puede ver en las líneas 12-25 de la figura 2 donde se reconoce una sentencia '*struct*'.

4.4. Comparación de tipos

En la comparación de tipos, el programa muestra aquello que se está comparando y el resultado obtenido de la misma. Igualmente, es necesario acceder al método de comparación a través del programa seleccionado. Si comparamos un valor entero (t0) con un valor en coma flotante (t1), *pI.compareTypes(t0, t1)*, el

programa nos devuelve *Comparación int con float: false*. Se puede ver su uso en las líneas 27-33 de la figura 2 donde se comprueban los tipos de una sentencia típica de asignación.

5. Evaluación preliminar

Debido a las restricciones actuales impuestas por la pandemia, no ha sido posible evaluar esta API en contexto educativo controlado, por ello hemos utilizado un enfoque más analítico. El diseño de una API abarca decisiones que van desde su arquitectura y funcionalidad hasta el nombre específico de clases, funciones o métodos. Myers y Stylos [4] proponen una manera sencilla de evaluar el diseño APIs adaptando las diez reglas heurísticas de Nielsen. Nuestra API cumple siete de estas diez pautas. Descartamos la *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* (que no está relacionado con los errores detectados por el sistema de tipos en desarrollo).

Según nuestro análisis, nuestra API cumple el resto de reglas. 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 necesita escribir poco para que los 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.

6. Conclusiones y trabajos futuros

Este trabajo presenta una API cuyo objetivo es ayudar a los estudiantes a la hora de desarrollar las prácticas de la asignatura de Compiladores relacionadas con la fase de comprobación de tipos. La API pretender abarcar diferentes paradimas, ofreciendo una estructura que permite a los estudiantes manejar fácilmente las expresiones de tipos sin evitar la complejidad que supone el diseño de parsers que implementan sistemas de tipos. Hemos analizado la API según las heurísticas de Myers y Stylos [4] obteniendo un buen resultado.

No olvidamos que este es un trabajo en curso y se pueden realizar mejoras, por ejemplo comparando la complejidad de las soluciones con más parámetros aparte del número de líneas de código. También se pretende aumentar el juego de prueba de prácticas para realizar las anteriores comparativas con una población mayor. Finalmente, si el contexto actual lo permitiera se planea realizar un experimento controlado con estudiantes para hacer un estudio más detallado de la aportación de la API al proceso de aprendizaje.

Para terminar, la API podría ampliar su funcionalidad, tanto a nivel detallado permitiendo que los programadores cambien la salida de log de depuración según sus necesidades como a nivel conceptual cubriendo otros aspectos como la generación y representación de código intermedio.

Agradecimientos

Este trabajo ha sido financiado por el Ministerio de Economía y Competitividad (ref. TIN2015-66731-C2-1-R) y la Comunidad de Madrid, con el proyecto e-Madrid-CM (P2018/TCS-4307), también cofinanciado por los fondos estructurales (FSE y FEDER).

Referencias

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi y Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools (2nd Edition) Addison-Wesley Longman Publishing Co., Inc. 2006.

[2] Francisco J. Almeida-Martínez, Jaime Urquiza-Fuentes y J. Ángel Velázquez- Iturbide. Visualization of Syntax Trees for Language Processing Courses J. of Universal Computer Science 15(7):1546–1561, apr 2009.

[3] Michael Hewner. Undergraduate Conceptions of the Field of Computer Science. In ICER 2013, pages 107–114, New York, NY, USA, 2013. ACM.

[4] Brad A. Myers y Jeffrey Stylos. Improving API Usability. Association for Computing Machinery: 62-–69, Communications of the ACM, 2016.

[5] Susan H. Rodger, Eric Wiebe, Kyung Min Lee, Chris Morgan, Kareem Omar y Jonathan Su. Increasing Engagement in Automata Theory with JFLAP. ACM Press: 403–407. Proceedings of the 40th ACM Technical Symposium on Computer Science Education 2009.