Compilador de Cool

Ariel Bazán Rey Greidy Valdés Vivanco Ernesto Quevedo Caballero

17 de junio de 2019

Índice

1. Arquitectura

Figura 1: Arquitectura del Compilador

La arquitectura del compilador puede observarse en la figura 1. Primero el Analizador Léxico o *Lexer* recibe el código fuente, que no es más que cadenas de caracteres, devolviendo una cadena de *tokens* cada uno con los valores de sus atributos que apuntan a la tabla de símbolos de dicho *token*. La segunda fase es el Análisis Sintáctico o *Parsing* que utiliza los *tokens* para crear y devolver el Árbol de Sintaxis Abstracta (AST) que describe la estructura gramatical de los *tokens*. La tercera fase es el Análisis Semántico que utiliza el AST y la información en la tabla de símbolos para chequear la consistencia semántica con respecto a las definiciones del lenguaje. La cuarta fase es la Generación de Código Intermedio, en este caso *Class Intermediate Language* (CIL) donde se traduce el AST de COOL a un AST de CIL y este último es la salida de esta fase. Finalmente la fase de Generación de Código MIPS, traduce la representación del lenguaje intermedio, en este caso su AST, en lenguaje MIPS.

2. Analizador Léxico y Analizador Sintáctico

Para el análisis léxico se diseña una clase *Lexer* que se implementa utilizando las funcionalidades de la librería **ply**(https://github.com/dabeaz/ply). Particularmente mencionar la utilización de los *states* que pro-

porciona **ply**, pues para *tokens* como *strings* o *comments* se desea un análisis y construcción léxica distinta.

Para el análisis sintáctico se diseña una clase *Parser* que recibe una instancia de la clase *Lexer*. A partir de la utilización de la librería **ply**, los *tokens* construidos en la clase *Lexer* y la gramática del lenguaje construye el AST.

La gramática del lenguaje no es LL(1) pues posee varios conflictos, como por ejemplo el que ocurre en la producción *feature_list* $\Longrightarrow$ ID COLON TYPE | ID COLON TYPE ASSIGN expr, donde el **FFIRST{ID COLON TYPE, feature_list}** $\cap$ **FFIRST{ID COLON TYPE ASSIGN expr, feature_list}** $\neq \varnothing$. Tampoco es LR(1) pues posee varios conflictos **Shift-Reduce**, como por ejemplo con las producciones:

1. *expr* $\Longrightarrow$ *expr* + *expr*•, DOT
2. *expr* $\Longrightarrow$ *expr*•DOT ID (params_expr), }

Para resolver los conflictos durante el *parsing* LR(1), la biblioteca **ply** posee un mecanismo llamado precedencias. Pues en **ply** ante un conflicto **Shift-Reduce** siempre se decide hacer **Shift** en vez de **Reduce**, aunque esta estrategia funciona en muchos casos, en otros no es suficiente, como es el caso de las expresiones aritméticas. Para resolver la ambiguedad, se permite a *tokens* individuales asignarles un nivel de precedencia y una asociatividad. Por ejemplo precedence = ((left, PLUS, MINUS),(left, MUL, DIVIDE)). Este ejemplo representa que la multiplicación, división, suma y resta tienen asociatividad a la derecha y que la multiplicación y división tienen mayor presedencia que la suma y resta.

3. Analizador Semántico

Para el análisis semántico, donde se validan todas las reglas del lenguaje, es necesario tener la representación del contexto y cada nodo del AST tiene acceso a dicho contexto. Para eso se diseña una clase que representa esta estructura de datos, llamada *Context*, donde se tiene acceso a los entornos de objeto y método respectivamente. El entorno de objeto permite obtener para un identificador v cuál es su tipo. El entorno de método permite obtener para un tipo C y el nombre de una función f de C lista de tipos de los parámetros de f y además el tipo del valor de retorno.

Para la construcción del contexto es necesario hacer un primer recorrido por el AST recolectando todos los tipos definidos y almacenándolos en el contexto con la información de sus respectivos atributos y métodos. También en este recorrido se aprovecha y ya se hacen algunos chequeos semánticos como que no existan dos atributos o métodos iguales dentro de un mismo tipo. Todos los recorridos sobre el AST se hacen utilizando el Patrón Visitor.

Para poder representar la jerarquía de clases en el contexto y hacer chequeos semánticos sobre la misma, se diseña una clase llamada *LCA* como estructura de datos para trabajar con el árbol de la jerarquía. A dicha estructura se le puede preguntar si un tipo existe, si un tipo conforma a otro, también se puede obtener el preorden del árbol y además el *lca* de dos tipos en el árbol de jerarquía.

También se diseña una clase *Scope* para poder hacer los chequeos semánticos con respecto a los ámbitos de cada clase y método. De esta forma se puede comprobar si una variable está definida en el ámbito actual o en algún ancestro del mismo.

Finalmente se hace un último recorrido por el AST para el resto del chequeo semántico utilizando toda la información guardada en el contexto.

4. Generación de Código Intermedio

El proceso de traducir un programa en código MIPS es complicado hacerlo de forma directa, por lo que se utiliza un lenguaje intermedio llamado *Class Intermediate Language* (CIL), el cual es un lenguaje de

máquina con instrucciones de tres direcciones, todas las variables son enteros de 32 bits y tiene soporte para operaciones orientadas a objetos. Este lenguaje cumple con las dos propiedades esenciales de un lenguaje intermedio, es fácil de producir y es fácil de traducir a MIPS.

Se utiliza el lenguaje visto en Compilación 1, pero se hacen algunas modificaciones. Por ejemplo, se añade un nodo case, pues esta expresión se maneja directamente en MIPS. Además se añaden nodos para las funciones de los tipos *built-in* que son implementadas en MIPS también, esto se debe a que muchas de ellas pueden resultar difícil de representar en CIL. Ejemplo de esto es la función copy, type_name y abort.

5. Generación de Código MIPS

La generación de código MIPS es la última fase del compilador. En esta etapa es importante destacar algunos aspectos y convenios que se tomaron en cuenta.

5.1. Representación de Tipos

Una instancia es representada mediante el tipo al que corresponde y una serie de atributos. A su vez, el tipo es representado por el espacio en memoria que ocupa una instancia, las funciones que posee y las etiquetas en las que se encuentran esas funciones. Esta representación se ilustra en la figura 2. Para invocar

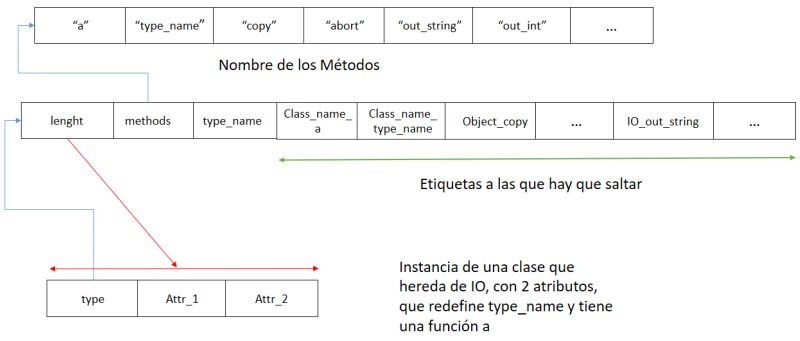

Figura 2: Representación de Tipos

una función, se busca en la tabla de nombre de los métodos y se ejecuta la etiqueta correspondiente.

5.2. Boxing y Unboxing

En COOL, la mayoría de los tipos tienen un comportamiento por referencias. Sin embargo, existen tipos *built-in* que se comportan por valor. Este es el caso de Int, String y Bool. Para el correcto funcionamiento de estos tipos, no se representan como se ilustra en la figura 2, sino que la variable debe contener el valor y no una referencia. Sin embargo, hay ocasiones en que, para obtener el resultado esperado es necesario 'envolver' el valor de las variables en una estructura similar a la anteriormente descrita. Ejemplo de estos casos es cuando se almacena una variable de tipo por valor en un Object o cuando una rama de If tiene tipo por valor y el tipo estático es object, o al invocar funciones.

De igual manera hay casos en que una expresión tratada como referencia debe comenzar a tratarse por valor. El principal ejemplo de esto es el case.

Como se puede apreciar en la figura 3, si el tipo dínamico de $expr_0$ es Int, en un primer momento se le debe hacer boxing. Esto se debe a que para conocer el tipo dinámico se utiliza la función type_name. Sin embargo, si se ejecuta la primera rama entonces en $id1$ debe quedar una variable entera por valor. En este caso se debería hacer unboxing.

```
case 5 off
    id1: Int => id1 + 1
    id2: Object => new Object
esac
```

Figura 3: Unboxing

5.3. Estado de la pila

Para la ejecución de funciones se utiliza el convenio de que una subrutina debe dejar la pila en el mismo estado que antes de ser añadidos sus parámetros. Además, en la pila se guardan las variables locales y $\$ra$, con el objetivo de que funcione bien la recursividad y llamado a otras subrutinas.

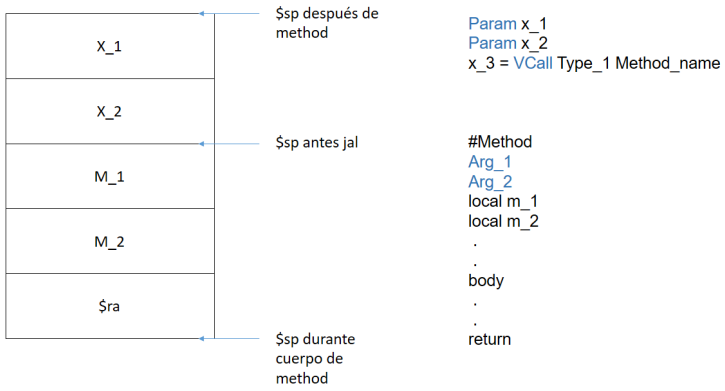

Figura 4: Comportamiento de la pila

6. Pendiente

Existen una serie de funcionalidades que no funcionan como es debido. La mayoría de ellas tienen fácil solución.

6.1. SELF_TYPE

Este aspecto del compilador era opcional y no se realizó. Sin embargo, se efectuaron algunos ajustes para que las funciones de los tipos *built-in* que tienen tipo **SELF_TYPE** funcionen correctamente. Estas son copy, out_string y out_int.

6.2. Palabras reservadas en mayúscula

Aunque en COOL las palabras reservadas pueden encontrarse tanto en mayúscul como en minúscula, el lexer solamente identifica aquellas que comienzan con minúscula.