Compilador de Cool

Alejandro Rodríguez Pérez
Jorge Ignacio Rodríguez de la Vega Castro
Rodrigo Sua García Eternod

25 de junio de 2019

Índice

1. Arquitectura

Figura 1: Arquitectura del Compilador

La arquitectura del compilador consiste primeramente en un *lexer*, el cual nos transforma una entrada de código COOL en una serie de tokens que lo representan y detecta errores respecto a la escritura incorrecta de símbolos. La segunda fase del compilador es el *parser*, el cual nos transforma la serie de tokens devueltos por el *lexer* en un Árbol de Sintaxis Abstracta (AST) que nos representa el código del programa. La siguiente fase es el análisis semántico en donde se verifica el cumplimiento de las reglas semánticas del lenguaje. La penúltima fase es la generación del lenguaje intermedio CIL y mediante la transformación del AST del código en COOL, en un AST de CIL. La última fase es la generación de código MIPS, apoyándose en el AST anterior.

2. Lexer y Parser

Las implementaciones tanto del *lexer* como del *parser* fueron realizadas utilizando la biblioteca de python **ply**. La gramática base con la cual se reconoce el lenguaje es la especificada en el manual de COOL, mientras que problemas como el de la precedencia entre los operadores fue resuelta no a nivel de gramática, sino mediante las facilidades que ofrece **ply**.

Uno de los problemas más interesantes de resolver a la hora de *parsear* correctamente código COOL es el hecho de que, hablando en términos informales, una expresión que aparece en el cuerpo de una expresión **let** tiene que abarcar tantos *tokens* como permita la gramática, lo cual implica que nunca debe aparecer un **let** (no parentizado) como parte izquierda de una expresión binaria. Este problema causa que, al usar la gramática especificada en el manual de COOL, genere un *parser* con conflictos *shift-reduce* en los estados

donde se pueda aplicar. En lugar de resolver este problema a nivel de gramática, se le especificó al *parser* generado por **ply** que desambiguara haciendo *shift* ante cualquiera de estos conflictos. Esta propuesta trae consigo el comportamiento deseado.

Una consecuencia directa de utilizar estas facilidades de **ply** es que la gramática que se le especifica inicialmente no es libre del contexto.

3. Análisis Semántico

En el análisis semántico se realizan 3 pasadas (mediante el patrón *visitor*) por el AST, 2 para construir los contextos de objetos y de métodos respectivamente y una para realizar el chequeo de tipos. Durante la construcción de los contextos se detectan tempranamente algunos errores, y todos aquellos errores semánticos que no aparecen como parte de expresiones. Un problema interesante en esta etapa radica en la captura de la mayor cantidad de errores posible, tal que unos no dependan de errores previos. Para atacar este problema el chequeo de cada expresión retorna si fue correcto o no, y en cualquier expresión donde la correctitud de una de sus subexpresiones no exté correlacionada con la correctitud de otras, se pueden detectar errores válidos en todas ellas. Los errores que se detectan en alguna de los primeras etapas son críticos, lo que hace no se avance a la siguiente etapa si alguna de ellas falla.

Para el manejo de información relevante relacionada con variables, métodos y clases se hace uso de los objectos *VariableInfo*, *MethodInfo*, *ClassInfo* respectivamente, los cuales son almacenados en los nodos que vinculen a una de estas tres entidades correspondientemente, de tal forma que dos usos en el AST de la misma entidad, por ejemplo una variable, en lugares diferentes del AST, están asociados a su *Info* correspondiente.

4. Generación de Código Intermedio

La importancia de esta fase está en que permite mediante un lenguaje orientado a objetos, alcanzar una fase intermedia (como su nombre lo indica) entre COOL y MIPS.
Para esta parte se utilizó un código intermedio de tres direcciones, con el que resolvimos varios problemas, por ejemplo el desenrollamiento de la herencia; un tipo esta conformado por sus atributos y después por sus métodos, por lo que nos apoyamos en lo anterior para la representación de un tipo, pues posee primeramente los atributos de su padre y después los suyos y se procede de manera análoga con los métodos. Otro problema es el *boxing* y el *unboxing*, cuyo uso se detecta de manera estática y para su solución se crearon operaciones box y unbox.
También se realizó un optimización respecto a la cantidad de variables temporales. Para esto en una lista se almacenan las variables temporales usadas y se lleva un puntero del índice de la ultima variable usada, es decir cuando se requiere hacer uso de una variable temporal, se utiliza la señalada por el puntero y este se corre y en caso de que el puntero este ya al final de la lista, se añade una nueva variable. Para este procedimiento fue útil el uso del *VariabelInfo*, ya que este va a tener asociada su dirección en memoria, por lo que si en algún momento se necesita una variable temporal, lo único que se tiene q hacer es reutilizar el *VariabelInfo* previamente creado y si no hay ninguno libre, crear uno nuevo.

5. Generación de Código MIPS

En esta última fase se pasa de código CIL a código MIPS, mediante un recorrido del AST de CIL. La representación de los tipos en memoria se divide en dos partes. Una va a contener la información común para todas las instancias del tipo, dígase nombre de la clase, la serie de métodos virtuales, puntero al padre y tamaño de la clase para poderla instanciar dinámicamente; y la segunda contiene el puntero a esta primera representación y todos los atributos.
La representación de un método esta dada primeramente por sus parametros, el *frame pointer* anterior,*return address* y sus locales.
El *void* es representado mediante una dirección estática en memoria.
La representación de *String* se hizo, primeramente reservando estáticamente los que se podían reservar de esta forma y para el resto se utilizó un buffer de tamaño 1024 bytes donde se almacenaba el string, ya

que no se sabe de antemano el tamaño del mismo, después se recorre el buffer y se guarda el string en un espacio dinámico con tamaño length del string(Nota:el buffer se sigue reutilizando durante el resto del código).