Documentacion del Proyecto final de Complementos de Compilación

Adrian Gonzalez Sánchez- C412 - [@adriangs1996]
Eliane Puerta Cabrera- C412 - [@NaniPuerta]
Liset Alfaro Gonzalez- C411 - [@LisetAlfaro]

Curso 2019-2020

1 Introducción

COOL (Classroom Object-Oriented Language) es un lenguaje pequeño, prácticamente de expresiones; pero que aun así, mantiene muchas de las características de los lenguajes de programación modernos, incluyendo orientación a objetos, tipado estático y manejo automático de memoria. La gramatica de Cool utilizada es libre del contexto y es definida en nuestro proyecto como una gramática s-atributada, una definición informal de la misma puede encontrarse en el manual de Cool que aparece en la carpeta doc, precisamente donde se encuentra esta documentación.
En este documento quedan reflejadas las principales ideas a seguir por el equipo para la implementación de un compilador funcional de este lenguaje, pasando por un lenguaje intermedio CIL que facilita su transición a Mips.

2 Requerimientos para ejecutar el compilador

Lo primero que se debe hacer ejecutar el archivo Makefile que se encuentra en la carpeta src del proyecto utilizando el siguiente comando:

```
$ make
```

una vez termine de ejecutarse, habrá ocurrido lo siguiente

- Se tendrá instalada la librería cloudpickle, requerida para cargar el Parser que posteriormente mencionaremos.
- Se creará la carpeta build donde habrá un módulo que contiene el binario de nuestro Lexer y nuestro Parser.

3 Arquitectura:

Módulos:

- **Tokenizer**: *src/tknizer.py*
- **Parser**: El parser no es un módulo, en vez de eso utilizamos nuestro propio generador de parsers, el cual se encuentra en *src/parserr/slr.py*. La tabla de parsing devuelta es lo que se compila con cloudpickle.

- **COOL AST**: *src/abstract/tree.py*
- **Visitors**: *src/travels/*.py*
- **CIL AST**: *src/cil/nodes.py*
- **MIPS Instruction Set**: *src/mips/*.py*

Fases en las que se divide el compilador:

Para una mejor organización se divide el compilador en varias fases

- **Lexer**: En esta fase se realiza un preprocesamiento de la entrada. Este preprocesamiento consiste en sustituir los comentarios ,que pueden ser anidados y por tanto no admiten una expresión regular para detectarlos, por el caracter *espacio*, manteniendo los caracteres fin de línea en el archivo fuente (Esto es necesario para detectar un errores en la línea y la columna exacta donde ocurren).
  El tokenizer que recibe este archivo fuente ya procesado tiene una tabla de expresiones regulares donde se encuentran clasificados los tokens por prioridad, esto nos permite, por ejemplo, no confundir palabras claves(keywords) con identificadores. Este devuelve una lista de *tokens* y pasamos a la segunda fase.

- **Parser:** El generador del parser fue hecho por el equipo y la implementación puede encontrarse en los archivos cuyo nombre corresponden con esta fase. Este devuelve un parser *LALR*, que produce una derivación extrema derecha sobre una cadena de tokens y luego esta se evalúa en una función llamada *evaluate* que devuelve al *AST de Cool*.

- **Chequeo Semántico:** En esta fase implementan varios recorridos sobre el AST de COOL, siguiendo el patrón *visitor*. Cada uno recolecta errores de índole semántica y si uno de ellos encuentra un error lanza una excepción, siendo este el comportamiento deseado:
  - Recolector de Tipos(TypeCollector): Este recolecta los tipos y los define en el contexto. En este recorrido se detecta si se redefinen las clases built-in, o si las clases son redefinidas.
  - Constructor de Tipos(TypeBuilder): Este Visitor define en cada tipo las funciones y atributos, y maneja la herencia y redefinición de funciones. En este se detectan errores como dependencias circulares, los atributos definidos con tipos inexistentes, se chequean la correctitud de las definiciones de los parámetros de los métodos que se redefinen, etc..
  - Inferencia de tipos(Inferer): En este recorrido se realiza el chequeo de Tipos y además incorporamos inferencia de tipos al lenguage, feature que no está contemplado en la definición formal. Este recorrido visualiza el codigo de COOL como una gran expresión y realiza un análisi Bottom-Up, partiendo de tipos bien definidos de las expresiones en las hojas como enteros, strings, etc. Como paso extra, para inferir los tipos nos basamos en las mismas reglas semánticas de tipos, pero agregamos recorridos para resolver los problemas de dependencia que puedan surgir entre los elementos a inferir. Tomemos como ejemplo el siguiente fragmento de código, tomado en parte del test **Complex.cl**:

    ```
    class Main inherits IO {
        main() : AUTO_TYPE {
    (let c : AUTO_TYPE <- (new Complex).init(4, 5) in
        out_int(c.sum())
    ```

```
)
    };
};

class Complex inherits IO {
    x : AUTO_TYPE;
    y : AUTO_TYPE;

    init(a : AUTO_TYPE, b : AUTO_TYPE) : Complex {
{
    x <- a;
    y <- b;
    self;
}
    };

   sum() : AUTO_TYPE {
      x + y
   };
};
```

```
$ python testing.py > testing.mips
$ spim -file testing.mips
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
9%
```

El compilador es perfectamente capaz de inferir los tipos de cada argumento y de cada atributo en este caso, pero, cómo ???

Es fácil resolver los tipos en este ejemplo a vista, ya que miramos primeramente el método **sum** en el cual los atributos son usados en una suma, y como en *COOL* solo se pueden sumar enteros, entonces los atributos deben ser enteros. Por otro lado, luego verificamos los argumentos de la función **init** y vemos que están siendo utilizados como asignación para los atributos, por lo que deben ser enteros también. La inferencia de los tipos de retorno de los métodos **sum y main** es trivial, como lo es inferir el tipo de la variable c. Pero para el compilador, esta forma de "elegir" por donde comenzar a mirar, no es para nada trivial, ya que las dependencias de los tipos de retorno forman un grafo que debe ser recorrido en orden topológico (de no existir dicho orden, o sea, no ser un DAG, evidentemente existiría una dependencia cíclica que no es posible resolver), pero definir este recorrido se ve complicado por el scoping de los métodos, y nuestra forma de tratar la inicialización de atributos, que terminan siendo métodos "anónimos". Dicho esto, nuestro enfoque es definir una relación de dependencia en profundidad, o sea, $d(x) = p$ si para poder inferir el tipo de "x", es necesario inferir, como máximo, p elementos antes. Luego se hacen p recorridos, sobre el AST de COOL, en cada paso actualizando el contexto en el que existe cada elemento para poder utilizarlo en pasadas posteriores. Así pudiéramos inferir en una primera pasada los tipos de retornos de los atributos, y luego en una segunda pasada, los argumentos. En la práctica, el **inferer**

revisa primero los métodos de inicialización de cada atributo, y luego los métodos en orden de definición en el código fuente, por lo que en este caso, se infieren primero los argumentos del método, pues son usados constantes enteras en su llamada a **init**, y de ahí se deduce el tipo de los atributos al ser asignados en la función. Es fácil entonces ver que esta forma de inferencia está limitada a la profundidad configurada por el compilador, la cual, por defecto es 5, ya que en la práctica, un nivel de anidación de dependencias mayor es dificil de conseguir, aunque se puede modificar pasándole como argumento la profundidad al compilador, de la siguiente manera:

```
$ python pycoolc.py --deep=10 <INPUT> [<OUTPUT>]
```

- **Generación de código:** En esta fase se pasa de Cool a CIL y luego de CIL a MIPS. El conjunto de instrucciones de MIPS que se utiliza son del emulador de SPIM, o sea, utilizamos el conjunto de instrucciones extendidas de MIPS, donde existen instrucciones como LW y MOVE que no son parte de la Arquitectura original y que se traducen en dos o más instrucciones en modo *bare*. En esta fase primeramente traducimos el AST de Cool a un AST de CIL (lenguaje intermedio), con el objetivo de facilitar el paso hacia MIPS. Este es un código de tres direcciones, compuesto por tres secciones principales **.DATA .CODE .TYPES**. En la sección .DATA almacenamos nuestros strings literales y lo que nosotros llamamos *TDT*. En la sección de tipos va una representación de los tipos que luego se puedan convertir fácilmente a MIPS, aquí ocurre el mangling de los nombres de las funciones de modo que se puedan referenciar distintamente luego. Por último en la sección .CODE van todas las funciones que nuestro programa necesite, incluidas las built-in que son bootstrapeadas al inicio de este recorrido. Es interesante resaltar en CIL como manejamos los métodos y atributos heredados, ya que en los records que representan los tipos, cada método y atributo heredado es agregado primero antes que los definidos por el tipo en particular; lo que garantiza que para cualquier método "x", el índice de x en la lista de métodos de cualquier tipo que lo herede, es el mismo. Una vez que se termina el recorrido por el AST de COOL, hemos conformado las tres secciones de CIL, y estas son pasadas al Generador de MIPS, que no es más que otro recorrido sobre este nuevo AST que hemos obtenido.
- **Generación de código MIPS**: El primer reto para generar código ensamblador es representar el conjunto de instrucciones de la arquitectura. Para lograr esto definimos varias clases bases (branches, loads, stores, comparisson, etc) que redefinen el método **__str__**, de modo que utilizan el nombre de la clase para representarse, y agregan el símbolo $ en dependencia de si ponemos una constante o un registro. De esta forma la generación de código referente a un nodo de CIL se realiza de manera natural, casi como si programáramos en ensamblador, como por ejemplo, en este caso:

```
@visit.register
    def _(self, node: cil.ArgNode):
        # Pasar los argumentos en la pila.
        self.add_source_line_comment(node)
        src = self.visit(node.name)
        reg = self.get_available_register()
        assert src is not None and reg is not None

        if isinstance(src, int):
            self.register_instruction(LI(reg, src))
        else:
            self.register_instruction(LW(reg, src))
```

```
self.comment("Push arg into stack")
self.register_instruction(SUBU(sp, sp, 4, True))
self.register_instruction(SW(reg, "0($sp)"))

self.used_registers[reg] = False
```

Como se puede ver, registramos una instruccion, casi como si programáramos assembly.

Otra aspecto significativo es la representación de los tipos en runtime. Al final, todo son solo números, por lo que tenemos que darles un formato que sea adecuado para poderlos referenciar y, que, además, permita acceder a sus propiedades de manera eficiente. En este aspecto, usamos la siguiente estructura para los tipos:

```
#################################  address
#          TYPE_POINTER         #
#################################  address + 4
#          VTABLE_POINTER       #
#################################  address  + 8
#           ATTRIBUTE_1         #
#################################  address + 12
#           ATTRIBUTE_2         #
#################################
#               ...             #
#               ...             #
#               ...             #
#################################
```

Esta estructura permite varias cosas:

- Acceso a atributos en $O(1)$.
- Acceso al nombre del tipo que corresponda a una instancia en $O(1)$.
- Dispatch dinámico en $O(1)$.

El mencionado puntero a la VTABLE, no es más que el inicio de un array (Idea sacada de *C++*) que en cada tipo contiene los nombres manglados de sus funciones, y recordemos que anteriormente dijimos que en CIL, garantizábamos que el indíce de cada método es el mismo en cada tipo que lo herede, de ahí que acceder a una función determinada sobre una instancia es tan fácil como cargar el puntero a la VTABLE de la instancia, calcular el índice del método que se quiere en tiempo de compilación y indexar en la VTABLE en ese indice en runtime, para un dispatch en $O(1)$.

Otro aspecto que le prestamos interés en esta fase es el tema de los *case*. Las expresiones "case" requieren que la jerarquía de tipos sea mantenida de alguna forma en runtime. En vez de esto, calculamos lo que llamamos una TDT (Type Distance Table), que no es más que una tabla donde para cada par de tipos A y B se le asigna la distancia a la que se encuentran en la jerarquía, o -1 en caso de que no pertenezcan a la misma rama. Para conformar esta tabla hacemos un preprocesamiento en $O(n^2)$ basado en los resultados de los tiempos de descubrimiento de un recorrido DFS sobre la jerarquía de tipos, partiendo desde la raíz, Object.

Importante también son las convenciones que adoptamos al generar código. Por ejemplo, cada función de MIPS tiene un marco de pila que es creado con un mínimo de 32 bytes,

porque es convenio entre programadores de MIPS. Los parámetros a funciones se pasan en la pila, ya que aunque MIPS cuenta con 32 registros, no todos son de propósito general y complica mucho la implementación el hecho de llevar una cuenta de los registros utilizados.