Compilación
COOL Compiler

Karl Lewis Sosa Justiz
Juan David Menéndez del Cueto
C-412

November, 2020

Uso del compilador

Para usar el compilador es necesario tener instalado Python 3.7 o superior. Todos los requerimientos pueden ser instalados ejecutando `pip install -r requeriments.txt` desde la raíz del proyecto.

El módulo que contiene toda la lógica del compilador es `exec_cool.py`. Para utilizarlo solo es necesario pasarle como argumento un fichero con código fuente de COOL, por ejemplo, ejecutar `python exec_cool.py <path>`. Un archivo en el mismo path del código fuente será creado, con el mismo nombre, pero con extensión .mips. Este fichero contendrá cdigo MIPS con pseudo instrucciones y se puede probar en cualquier implementación del simulador SPIM.

Arquitectura del compilador

La estructura del compilador se puede separar en módulos, que pertenecen a las fases del desarrollo del mismo. A continuación se explicarán las fases antes mencionadas y las ideas principales seguidas durante su desarrollo.

Análisis lexicográfico y sintáctico

El análisis lexicográfico y sintáctico trata de la comprobación del programa fuente hasta su representación en un árbol de derivación, incluye desde la definición de la gramática hasta la construcción del lexer y el parser.

En estas fases se emplearon las herramientas de construcción de compiladores lex y yacc a través del paquete de *Python* `ply`.

El lexer y el parser del proyecto se encuentran implementados en los módulos `lexer.py` y `Parser.py` respectivamente, además `AstNodes.py` ofrece la jerarquía del AST de COOL propuesta.

Análisis semántico

El análisis semántico consiste en la revisión de las sentencias aceptadas en la fase anterior mediante la validación de que los predicados semánticos se cumplan.

El árbol de derivación es una estructura conveniente para ser explorada. Por lo tanto, el procedimiento para validar las reglas semánticas es recorrer cada nodo. El patrón visitor fue el utilizado para esta tarea.

En el primer recorrido se recogen todos los tipos definidos; en el segundo se construye el contexto de métodos y atributos; y en la tercera pasada se procede a la construcción de los escopes recolectando las variables definidas en el código, teniendo en cuenta su visibilidad. Por último, se verifica la consistencia de tipos en los nodos del AST.

Generación de código

Esta fase consiste en la generación de código de máquina a partir del código ya procesado y validado por las fases anteriores. Debido a que la dificultad de

llevar directamente el código de COOL a MIPS es demasiado alta, se genera un código intermedio para facilitar esta tarea.

CIL

Como lenguaje intermedio para este proceso, se utilizó el lenguaje CIL, un lenguaje parecido al dado en clases pero al que le fueron añadidas algunas instrucciones para facilitar el proceso de generación de código MIPS.

En particular unas de las estructuras más interesantes fue el caso del case.

```
case <expr0> of
<id1> : <type1> => <expr1>;
. . .
<idn> : <typen> => <exprn>;
esac
```

La instrucción case es utilizada para conocer el tipo que más se ajusta al de la expresión en tiempo de ejecución pero en la representación de objetos escogida no se tiene información relativa al padre del tipo por lo tanto cambiamos la representción de la instrucción. Primeramente ordenamos los typen con un orden topológico de manera tal que los tipos hijos queden antes de los padres, luego por cada uno de estos tipos hacemos un recorrido preorden por sus descendientes y a cada uno de estos le asociamos la expresión relacionada al typen, en caso de repetirse tipos siempre se escoge la asociación previa, luego solo preguntamos por el tipo de la expresión en el orden antes explicado y escogemos esa rama. Nótese que gracias a que ordenamos los typen si typei es el que más se ajusta a la expresión, el tipo de la expresión no puede ser subtipo de los typen anteriores, y al mismo tiempo debe ser subtipo o ser el propio typei.

También es importante señalar la creación de 2 nuevas funciones por tipo que son el constructor y el inicializador, el primero es llamado para llenar cada instancia al ser creada y el segundo es llamado desde el constructor para rellenar los atributos heredados del padre y los propios.

Para la generación de código CIL se usó el patrón visitor, generando para cada nodo del AST de COOL el conjunto de instrucciones equivalentes en este lenguaje. Además, a cada variable, atributo y función se le asigna un identificador único para facilitar la generación de código MIPS.

MIPS

Se recorren los nodos creados de CIL usando, nuevamente, el patrón visitor, creando las operaciones de MIPS equivalentes para cada instrucción.

Registro de activación (Activation Record AR)

El código para llamar a las funciones y crearlas depende directamente de como fue diseñado el AR.

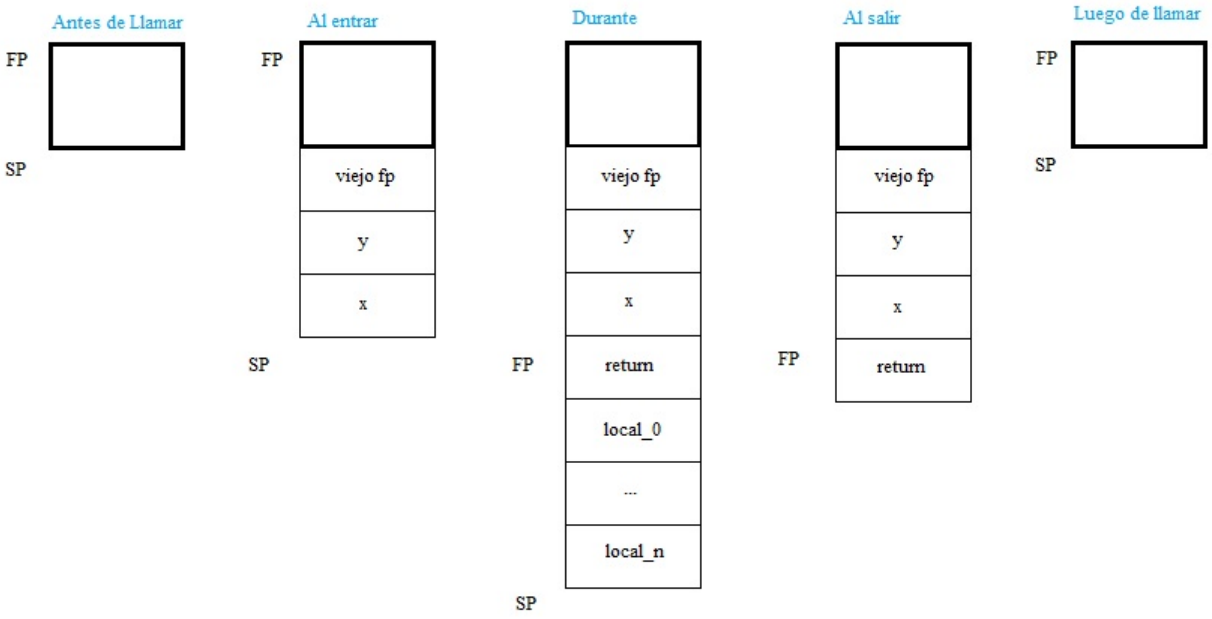

Figure 1: Llamado para f(x,y)

Las principales ideas del AR usado son:

1. El resultado siempre va a ser guardado en el acumulador. (Por lo tanto no hay necesidad de guardar el resultado en el AR)
2. EL $sp es el mismo a la entrada y salida de una llamada.
3. Los parámetros son guardados en el AR y son introducidos antes de llamar en orden inverso.
4. Es necesario guardar la dirección de retorno.
5. Se va a utilizar al $fp como guía para obtener tanto los parámetros como las variables locales. Por lo tanto es conveniente guardarlo antes de llamar y recuperarlo justo al volver
6. Las variables locales son introducidas por en el cuerpo
7. El iésimo parámetro está $fp+ 4(i+1) , análogamente la iésima variable local está en $fp - 4(i+1)

Con este modelo siempre se asume que los registros están sucios por lo tanto no hay necesidad de guardarlos y recuperarlos antes y después de los llamados, lo cual es de gran ayuda porque mantiene simple el código e incluso puede ser más eficiente porque si bien el uso de los registros es recomendado, al guardar y salvar de la pila se pierde tiempo considerable además en muchos métodos no es necesario usar gran cantidad de registros y muchas otras implemetaciones los guardan y salvan todos de igual manera.

Representación de los objetos

La representación escogida siguió las siguientes ideas.

1. Los objetos están guardados en memoria contigua.
2. Cada atributo está ubicado en el mismo corrimiento del objeto
3. Cuando un método es llamado el objeto es self.
4. El ID de clase es la dirección asociada al string que contiene su nombre.
5. Tamaño es entero que representa el cuantas word componen al objecto.
6. El Dispatch Ptr es un puntero a una tabla de métodos (Explicado luego).
7. Notar que si B hereda de A la representación de B es igual a la de A solo adicionando nuevos espacios para otros atributos.
8. Los atributos están de manera sucesiva.
9. Cada atributo está ubicado en el mismo corrimiento para todas las subclases.

Table 1: Representación en memoria.

	Offset
ID de clase	0
Tamaño	4
Dispatch Ptr	8
Atributo_0	12
Atributo_1	16
...	...
Atributo_n	n*4+ 12

Table 2: Representación en memoria de las clases del ejemplo.

Clase / Offset	0	4	8	12	16	20
A	Atag	5	*	a	d	
B	Btag	6	*	a	d	b
C	Ctag	6	*	a	d	c

Table 3: Representación en memoria de las Virtual Table del ejemplo.

Clase / OFFset	0	4
A	fA	
B	fB	g
C	fA	h

```
Class A{
a:Int<-0;
d:Int<-1;
f():Int{a<-a+d};
};

Class B inherits A{
b:Int<-2;
f():Int{a};
g():Int{a<-a-d};
}

Class C inherits A{
c:Int<-2;
h():Int{a<-a*c};
};
```

Listing 1: Ejemplo

Virtual Table

Cada clase posee un puntero a una dirección de memoria que indexa todos sus métodos (Dispatch Ptr) incluyendo lo heredados y de manera análoga a los atributos cada método se encuentra en el mismo corrimiento para una clase A y todas sus subclases.

Nótese como:

1. La tabla de A solo tiene un método.

2. La tabla de B y C extienden la tabla de A a la derecha.

3. Y como los métodos pueden ser sobrescritos el método f no es el mismo para cada clase pero siempre se halla en el mismo corrimiento.

Problemas técnicos

Análisis lexicográfico y sintáctico

En esta fase el mayor problema enfrentado fue la tokenización de strings y comentarios de varias líneas. El mismo fue resuelto mediante el uso de estados que brinda `ply`.

Análisis semántico

El mayor problema enfrentado fue el trabajo con los scopes, principalmente en las instrucciones `case` y `letIn`.

Generación de código

En esta sección, los problemas más interesantes fueron: la necesidad de hacer `Boxing` cuando era necesario castear los tipos `String`, `Bool` o `Int` a `Object`; y la necesidad del valor `void`, el cual fue manejado como una dirección de memoria.