coolcmp

Nuestro compilador *coolcmp* está finalmente terminado. En este reporte exponemos el proceso de construcción del compilador, las decisiones de diseño y el uso del mismo.

Requisitos

- pytest
- pytest-ordering
- ply
- print-tree2

También estos se encuentran en el fichero `/requirements.txt`.

Usando coolcmp

Para usar el compilador, nos movemos a `/src` y allí ejecutamos `python -m coolcmp -h` para mostrar la ayuda, esto da la siguiente salida:

```
usage: python -m coolcmp [-h] [--ast] [--cil_ast] [--tab_size TAB_SIZE] [--no_mips] file_path

Cool compiler programmed in Python.

positional arguments:
  file_path            Path to cool file to compile

optional arguments:
  -h, --help           show this help message and exit
  --ast                Print AST
  --cil_ast            Print CIL AST
  --tab_size TAB_SIZE  Tab size to convert tabs to spaces, default is 4
  --no_mips            Dont generate mips file
```

Por tanto, para compilar un fichero `code.cl` solo debemos hacer `python -m coolcmp code.cl`. Esto genera un fichero `code.mips`, el cual podemos ejecutar utilizando el simulador SPIM haciendo `spim -f code.mips`.

Estructura del proyecto

El proyecto se divide en varios módulos:

En la carpeta `/src/coolcmp` tenemos:

- `main.py` es el punto de entrada del proyecto.
- `cmp/ast_cls.py` contiene la definición de los nodos de cada Abstract Syntax Tree (*AST*) creado (*Parser* y *CIL*).

- `cmp/constants.py` contiene constantes usadas por la fase de generación de código principalmente.
- `cmp/environment.py` representa un entorno, esto es básicamente un mapa que hace corresponder cada nombre de variable con información sobre su definición.
- `cmp/errors.py` contiene los posibles errores que puede dar el compilador (a excepción de los runtime errors).
- `cmp/gen_cil.py` se encarga de generar código *CIL*.
- `cmp/gen_mips.py` se encarga de generar código *MIPS*.
- `cmp/lexer.py` contiene la definición y métodos del lexer.
- `cmp/parser.py` contiene la definición y métodos del parser.
- `cmp/parse_tab.py` archivo generado automáticamente por `ply` .
- `cmp/print_ast.py` se encarga de imprimir un *AST* por consola.
- `cmp/semantics.py` se encarga del análisis semántico.
- `cmp/source_code.py` representa un código fuente de *Cool*.
- `cmp/type_checker.py` se encarga del chequeo de tipos.
- `cmp/utils.py` define un logger para debuggear.

En la carpeta `/src/unit_tests` tenemos varias pruebas unitarias con las que probamos nuestro compilador (leer el fichero `/src/unit_tests/README.md` para más información).

Fases de construcción de coolcmp

El desarrollo del compilador se realizó en varias fases, cada una planteando una serie de problemas interesantes a resolver:

Análisis Léxico

La fase de análisis léxico constituye la primera por la que atraviesa el compilador. Toma como entrada código *Cool* y devuelve una lista de tokens en caso de éxito, en caso contrario reporta los errores lexicográficos detectados en el código.

Para la generación del lexer de *Cool* con `ply` identificamos tres tareas fundamentales:

1. la definición de los tokens de nuestro lenguaje.
2. la definición mediante expresiones regulares de las reglas que nos permiten identificar los tokens.
3. la lógica encargada de crear los tokens.

Entre las reglas más interesantes a analizar resaltan las de los comentarios y los strings. Al detectar un inicio de cadena o comentario necesitamos indicarle al lexer que aplique las reglas válidas para ese tipo de expresiones y deseche todas las demás, pues dejan de cobrar sentido cuando estamos analizando un string o un comentario. Para esto `ply` provee un mecanismo que llamaremos pila de estados y que funciona como se explica a continuación:

- Al detectar un string o comentario, se le indica al lexer que pase al estado referente a ese tipo de expresiones. Se le hace “push” a la pila.
- Mientras el lexer se mantenga en el nuevo estado aplicará solamente las reglas definidas para ese estado.
- Una vez se detecte el fin del string o comentario se pasa al estado anterior que registraba el tope de la pila de estados del lexer. Se le hace “pop” a la pila.

Análisis Sintáctico

En primer lugar convertimos la gramática de *Cool*, que se encuentra en la forma extendida de la notación de Backus-Naur (tiene expresiones regulares, por ejemplo), a la forma estándar.

A continuación creamos un pequeño script que genera un "esqueleto" del fichero `parser.py`, el cual contiene cada una de las reglas de la gramática como un método de la clase `Parser`, lo cual es requerido por el parser la herramienta `ply`.

Procedemos a completar cada uno de los métodos con la parte derecha de cada regla. Para esto creamos todas las clases necesarias para modelar cada símbolo de la gramática, estas clases serían los nodos del *AST*. A estas clases añadimos la información de línea y columna en la que se encuentra su símbolo correspondiente.

Análisis Semántico

Verificamos las reglas semánticas de *Cool* especificadas en el manual.

Árbol de Herencia

Procedemos a la creación del árbol de herencia, esto es, un árbol donde cada nodo representa una clase, en el que el nodo u tiene como hijo a v si la clase v hereda de u. Notemos además, que este árbol tiene como raíz a la clase *Object*.

Al tener el grafo creado, chequeamos que sea un árbol. Esto lo hacemos con el clásico algoritmo de detección de ciclos en grafos dirigidos, esto es, hacemos un Depth First Search (*DFS*) que va visitando los nodos, si estamos en el nodo u, vamos al nodo v y detectamos que v es ancestro de u en el *DFS-tree*[1] podemos afirmar que existe un ciclo. El siguiente código realiza este *DFS*:

```
def check_cycles(self):
    seen = {}
    up = {}

    for cls in self.ast_root.cls_list:
        if cls.type.value not in seen:
            self._dfs(cls, seen, up)

def _dfs(self, u, seen, up):
    seen[u.type.value] = up[u.type.value] = True

    for v in u.children:
        if v.type.value not in seen:
            self._dfs(v, seen, up)

        elif up[v.type.value]:
            raise SemanticError(v.type.line, v.type.col, f'Inheritance cycle detected')

    up[u.type.value] = False
```

Podemos notar que este árbol de herencia representa perfectamente la relación de *Conformance* ($\leq$) definida en el manual de *Cool*. Más aún, desde el punto de vista de nuestro árbol podemos afirmar que $A \leq B \iff B \text{ es ancestro de } A$.

Sobre *SELF_TYPE*

SELF_TYPE cumple lo siguiente con respecto a *Conformance*:

1. $\text{SELF\_TYPE}_X \leq \text{SELF\_TYPE}_X$
2. $\text{SELF\_TYPE}_C \leq P$ si $C \leq P$

Nuestro árbol de herencia actual no maneja *SELF_TYPE*, pero podemos manejarlo fácilmente si decimos que cada nodo C tiene un hijo SELF_TYPE_C. De esta forma, el punto 1 obviamente se cumple, y el punto 2 también dado que si P es ancestro de C, también lo es de SELF_TYPE_C, porque SELF_TYPE_C es hijo de C.

Chequeo de Tipos

Realizamos el chequeo de tipos usando visitor pattern; cumpliendo con las especificaciones en el manual de *Cool*. En esta fase surgen algunos problemas interesantes:

Conformance Test

Nos hace falta poder responder rápido si un nodo u conforma con v o no. Esto es posible hacerlo en $O(1)$ por cada pregunta. Para esto hacemos un *DFS* por nuestro árbol de herencia calculando dos valores para cada nodo x:

- $\text{td}(x) =$ tiempo de descubrimiento del nodo x, esto es, el primer momento en el que el *DFS* llega a x.
- $\text{tf}(x) =$ tiempo de finalización del nodo x, esto es, el último momento en el que el *DFS* está en x (cuando la recursión va a "salir" de x).

Sería algo como:

```
self._t = 0

def _dfs(self, u):
    self._t += 1
    u.td = self._t

    for v in u.children:
        self._dfs(v)

    u.tf = self._t
```

Luego u conforma con v si $\text{td}(v) \leq \text{td}(u) \leq \text{tf}(v)$. Esta es una de las tantas propiedades del *DFS-tree*[2].

Lowest Common Ancestor

Necesitamos poder contestar preguntas de Lowest Common Ancestor (*LCA*) para realizar la operación "join" descrita en el manual. Para esto hay muchos algoritmos que van desde $O(n)$ por pregunta hasta $O(1)$ con $O(n \log n)$ de pre-procesamiento[3]. Decidimos no complicarnos e implementamos uno de complejidad lineal por pregunta, este algoritmo es simple:

- supongamos que u está mas lejos de la raíz que v (sino, intercambiamos u con v), entonces $lca(u, v) = lca(p(u), v)$, donde $p(u)$ es el padre de u. Seguimos haciendo esto mientras que $u \neq v$.
- cuando $u = v$ el *LCA* es u.

Generación de código CIL

En esta fase tomamos varias decisiones de diseño sobre la representación de las clases y funciones de cara a su implementación en MIPS.

Representación de clases

Cada clase se transforma a una función de inicialización (le llamamos `FuncInit`), es decir, en código MIPS un objecto de una clase se crea llamando a una función de este tipo.

Cada clase tiene una serie de atributos, que se dividen en dos tipos:

- atributos reservados, entre ellos tenemos:
  - `_type_info` contiene una referencia a la dirección de memoria en MIPS de los datos de su tipo.
  - `_size_info` contiene el número de bytes que la instancia va a ocupar.
  - `_int_literal`, este atributo solo lo tiene la clase `Int`, representa el entero que tiene la instancia.
  - `_string_length`, este atributo solo lo tiene la clase `String`, representa la cantidad de caracteres de la instancia.
  - `_string_literal`, este atributo solo lo tiene la clase `String`, representa el string de la instancia.
  - `_bool_literal`, este atributo solo lo tiene la clase `Bool`, contiene 1 (`true`) o 0 (`false`).
- atributos añadidos por el programador en la definición de la clase.

Para cada clase C guardamos también su correspondiente tiempo de descubrimiento ($\mathrm{td}(C)$) y su tiempo de finalización ($\mathrm{tf}(C)$), esto nos va a ser útil para resolver los dispatches y para la resolución de las expresiones `Case`.

Representación de métodos

Cada método f se transforma en una "función" (le llamamos `Function`).

Digamos que C es la clase donde está definido el método f, entonces guardamos los siguientes datos:

- $\mathrm{td}(C)$
- $\mathrm{tf}(C)$
- $\mathrm{level}(C)$ (distancia a la raíz del nodo C en el árbol de herencia)

Estos datos lo usamos para los dispatches.

Sobre las variables y su entorno

En *Cool* cada variable está dentro de alguna clase. Diferenciamos dos tipos de variables:

- variables de clase (o atributos).
- variables locales, estas serían parámetros formales de un método o variables definidas en una expresión `Let` o `Case`.

Cada una de estas variables tienen un entorno determinado en el cual viven. Las variables de atributo viven en toda la clase, las variables locales solo viven en los bloques que fueron definidas. Además de esto, notemos que una variable local puede ofuscar a una variable, con el mismo nombre, definida en un entorno "superior" al de esta.

Las variables contienen la dirección de memoria de objetos, los cuales se guardan en el *Heap* (aunque hay ciertos objetos que guardamos en el *Data Segment*, por ejemplo: los literales y los objetos *Bool* `true` y `false`).

Las variables necesitan estar accesibles a la hora de acceder a un atributo, o realizar un cálculo, por lo que ellas se guardan en el stack de *MIPS*.

Un paso importante que tomamos es el de asignar de antemano a cada variable la posición que va a ocupar en el stack. Para esto llevamos un entorno que contiene para cada nombre de variable x un número que indica donde se va a encontrar x en el stack.

Un entorno E además tiene un padre $p(E)$, que es el bloque en el cual E está contenido. Un nuevo entorno se crea al llegar a un `Let`, `CaseBranch` (la rama de algún `Case`), `FuncInit` o `Function`.

Si el entorno actual es E y la última variable definida está en la posición k:

- si nos encontramos la definición de una variable x, hacemos $E(x) = k + 1$ y aumentamos k en 1.
- si nos encontramos con una variable que hace referencia a x, guardamos para esa variable su posición en ese momento en la pila (esto es, $E(x)$). Además guardamos si esta referencia fue a un atributo o a una variable local, cosa que podemos saber fácilmente.

Cuando salimos de E, reseteamos el entorno actual a $p(E)$ y ajustamos k correspondientemente.

Generación de código MIPS

En esta última fase procedemos a generar código *MIPS*. Implementamos cuidadosamente toda la funcionalidad explicada en el manual.

Sobre los registros

Se usó un esquema simple para la selección de registros a usar:

- tenemos un registro que solamente va contener en todo momento la dirección de memoria del objeto `self` actual.
- los registros temporales se usan para cualquier operación temporal necesaria.
- como internamente se necesita "pasar" referencias de algún objeto, destinamos un registro de argumento para esto.
- igualmente tenemos un registro dedicado a "regresar" una referencia de algún objeto.

Notar que de ser necesario "pasar" más de un argumento o "regresar" más de un valor, usamos el stack de *MIPS* para guardarlos.

Resolución de Dispatches

Digamos que tenemos un dispatch $x.f(...)$ y x es instancia de la clase C (si es un dispatch estático en la clase T, hacemos $C := T$). Para resolver el dispatch hay que buscar el ancestro más profundo de C, en el árbol de herencia, que contenga una función con nombre f.

Para esto vamos a guardar el td, tf y level de cada función con nombre f, como 3 números en el *Data Segment*, ordenados por el mayor level.

Ahora resolver el dispatch se reduce a buscar el primer X tal que $C \leq X$[4]. Esto lo podemos hacer iterando por los números guardados de las funciones con nombre f.

Notemos que la complejidad por dispatch es de $O(\#\text{funciones distintas})$. Otra alternativa pudo haber sido tener una tabla de punteros a funciones, pero crearla requería de un pre-procesamiento más lento: $O((\#\text{funciones distintas})^3)$[5]; aunque tendría un tiempo de respuesta por dispatch de $O(1)$. Pensamos que es un trade-off entre ambas opciones y escogimos la primera por considerarla un poco mejor.

Resolución de expresiones Case

Las expresiones `Case` se resuelven de forma similar a los dispatches.

Para esto reordenamos las ramas (notemos que la rama i es de clase T_i y por lo tanto tiene td, tf y level) de la misma forma que hicimos en el punto anterior. Ahora nos queda un problema similar: si la expresión que se le hace el `Case` es instancia de C, entonces es buscar el primer X tal que $C \leq X$.

Sobre Strings y Data Segment

En el *Data Segment* guardamos también los literales de `Int`, `String` y `Bool`. En particular, a los literales de `String` se le añaden ceros de forma tal que su tamaño sea un múltiplo de 4, para cumplir con las cuestiones de alineamiento de MIPS.

---

1. El *DFS-tree* de un grafo es un Spanning Tree obtenido por una pasada de *DFS*. ↩
2. Para más información, incluida demostración, revisar el CLRS, epígrafe 22.3 sobre *Depth First Trees*. ↩
3. Incluso existe un algoritmo offline (todas las preguntas se saben de antemano) que funciona en $O(1)$ por pregunta con $O(n)$ de pre-procesamiento. ↩
4. Porque ordenamos por mayor nivel esto garantiza que el primero que se encuentre es el correcto. ↩
5. Podemos alcanzar está complejidad si tenemos una línea de herencia: $C_1 \leftarrow C_2 \leftarrow \ldots \leftarrow C_n$ con cada C_i teniendo n funciones distintas. ↩