Complementos de Compilación: Implementación de un compilador de COOL

Laura Tamayo LAURA.TAMAYO@ESTUDIANTES.MATCOM.UH.CU
Grupo C411
Alejandro Klever A.KLEVER@ESTUDIANTES.MATCOM.UH.CU
Grupo C411
Miguel González MIGUEL.GONZALEZ@ESTUDIANTES.MATCOM.UH.CU
Grupo C411

Uso del compilador

El compilador ha sido realizado en el lenguaje de programación Python y posee las siguientes dependencias:

- pytest
- pytest-ordering
- pyjapt
- typer

Detalles sobre las versiones necesarias por cada biblioteca en el requirements.txt del proyecto.

Para ejecutar el compilador es necesario ir al directorio `src` y una vez ahí ejecutar el comando `python coolc.py <fichero de cool>` para ejecutar el compilador.

Arquitectura del compilador

Lexer

Para este proceso se utiliza la biblioteca de python PyJapt la cual ha sido desarrollada por este equipo y publicada bajo licencia MIT en GitHub y Pypi. El enlace a la documentación de PyJapt se encuentra en la sección de Enlaces Útiles y es rica en ejemplos y casos de uso.

La gramática del lenguaje se construyó para ser una gramática LALR(1), consecuentemente tablas action y goto bastante pequeñas en comparación con las de una parser LR(1), y por lo tanto mucho mas rápido de construir y menos costoso espacialmente.

Semántica

Para esta parte se utiliza el patrón visitor aprendido durante el curso de 3er año. Por lo cual se recibe el ast obtenido producto del proceso de parsing y pasará por diferentes clases que irán construyendo:

1. El contexto: Donde de guardará la información de las clases del programa en COOL.
2. El scope: Donde se almacena la información de las atributos, variables y parámetros de cada clase y método.

En el proceso de análisis semántico se utiliza un pipeline con las siguientes clases que usan el patrón visitor para acceder a cada nodo del ast y obtener nueva información en distintos recorridos:

1. TypeCollector: Encargada de recolectar los tipos de todas las clases del programa de COOL.
2. TypeBuilder: Encargado de recolectar la información de los atributos y métodos de cada clase
3. OverriddenMethodChecker: Encargado comprobar que la sobre escritura de métodos es consistente.
4. TypeChecker: Encargado del cumplimiento de todas las reglas entre los tipos.

Otro aspecto importante dentro del análisis semántico es la inferencia de tipos. El problema consiste en detectar para cada atributo, variable, parámetro de función o retorno de función el primer tipo que le puede ser asignado, modificando en el árbol de sintaxis abstracta el string AUTO_TYPE por el nombre del tipo correspondiente y asignando los tipos correspondientes en el contexto y el ámbito en que seo declarados, para resolverlo se utiliza el patrón visitor y teoría de grafos. A continuación se presentan detalles sobre el algoritmo a utilizar

Entrada : Un árbol de sintaxis abstracta, un contexto con todos los tipos declarados en el programa de COOL.

Salida : Un árbol de Sintaxis Abstracta, Un Contexto y un Scope con tipos bien etiquetados.

Algoritmo : Durante el recorrido del AST será construido un grafo dirigido cuyos nodos encerrarán el concepto de las expresiones marcadas como AUTO_TYPE y las aristas representan las dependencias entre las expresiones de estos nodos para inferir su tipo. Sea E1 una expresión cuyo tipo estático es marcado como AUTO_TYPE, y sea E2 una expresión a partir de a cual se puede inferir el tipo de estático de E1 entonces en el grafo existirá la arista <E2, E1>. Una vez construido el árbol se comenzará una cadena de expansión de tipos estáticos de la forma E1, E2, ..., En donde Ej se infiere de Ei con $1 < j = i + 1 \leq n$ y E1 es una expresión con tipo estático definido, al cual llamaremos átomo. Cuando todos los átomos se hayan propagado a través del grafo los nodos que no hayan podido ser resueltos serán marcados como tipos Object al ser esta la clase mas general del lenguaje.

Implementación : Para utilizar el algoritmo se utiliza una estructura llamada DependencyGraph donde es posible crear nodos como una estructura llamada DependencyNode y arcos entre ellos. La estructura DependencyGraph consiste en un OrderedDict de nodos contra lista de adyacencia. Esta lista de adyacencia contiene los nodos a los cuales la llave propagar su tipo, estos nodos de la lista tienen un orden y esto es fundamental para el algoritmo de solución de inferencia. Si se tiene un caso $\{x : [y, z]\}$ donde x, y, z son nodos, entonces el algoritmo determinará el tipo de y y z antes de continuar con todas sus cadenas de expansión, por lo que si z forma parte de una cadena de expansión de y entonces y no propagará su tipo a z ya que x lo hizo antes. Como se puede ver el algoritmo es un BFS simple donde en la cola, al inicio, serán incluido los nodos del grafo que tengan su tipo definido, es decir que no sea AUTO_TYPE.

Cada nodo del grafo será una abstracción de un concepto en el que se use un tagueo explícito de AUTO_TYPE y tendrá las referencias a las partes del proceso de semántica del programa, además de que cada nodo contará con un método update(type) el cual actualiza el tipo estático de estos conceptos.

El algoritmo funciona de manera análoga para atributos, variables, parámetros y retorno de funciones. Explicado de forma recursiva puede ser visto como:

- Un AUTO_TYPE será sustituido por su tipo correspondiente si este forma parte de una operación que permita saber su tipo, o es usado en una expresión de la cuál es posible determinar su tipo.

- Es importante señalar en que contexto estas dependencias son tomadas en cuenta:
  - Para los atributos marcados como AUTO_TYPE su tipo podrá ser determinado dentro del cuerpo de cualquiera de las funciones de la clase, o si es detectable el tipo de la expresión de inicialización.
  - Para las variables su tipo será determinado dentro del scope donde estas son válidas.
- Para los parámetros su tipo será determinado dentro del cuerpo de la función o cuando esta función sea llamada a través de una operacion de dispatch.
- Para los retornos de funciones, su tipo será determinado con su expresión y los llamados a dicha función a través de una operacion de dispatch.
- En las expresiones if-then-else o case-of asignan automáticamente el tipo Object si al menos una de sus ramificaciones devuelve una expresión que no tiene tipo definido (AUTO_TYPE), en caso contrario asignará el join de las ramificaciones.

Generación de código

En la última parte del proyecto se utiliza un lenguaje intermedio creado con el objetivo de facilitar la generación del código MIPS pues un salto directo desde COOL sería demasiado complejo. En este lenguaje intermedio se resuelven expresiones como `case` y se simplifica el código original.

Los tipos básicos del lenguaje tienen su código definido en MIPS para facilitar la generación de código.

Enlaces útiles

Proyecto en GitHub:
https://github.com/StrangerBugs/cool-compiler-2021 PyJapt:
https://github.com/alejandroklever/PyJapt