Informe de Complementos de Compilación

Datos Generales

Autores

- Miguel Tenorio Potrony
- Mauricio Lázaro Perdomo Cortés
- Lázaro Raúl Iglesias Vera

Sobre el proyecto

Para la implementación de este proyecto se tomaron como base, los proyectos realizados durante 3er año, donde se desarrollaron las fases de chequeo e inferencia de tipos, además de parsing. El código de dichos proyectos conserva su estructura pero estuvo sujeto a cambios y mejoras.

La mayoría de nuestras implementaciones siguen las ideas y utilizan las herramientas dadas en clase durante 3er año.

Todas las fases del proceso de compilación y ejecución serán explicadas a continuación.

Pipeline

Como se puede apreciar en main.py el pipeline de nuestro proceso de compilación es:

1. Lexer
2. Parsing
3. Recolección de tipos
4. Construcción de tipos
5. Chequeo/Inferencia de tipos
6. Verificación de tipos
7. Traducción de Cool a CIL
8. Traducción de CIL a MIPS

Cada parte del proceso será discutida en detalle durante las siguientes secciones.

Como se puede apreciar en la etapa #5 del proceso, el chequeo e inferencia de tipos se realizan al unísono, sin embargo cada parte se explicará en secciones separadas y se hará notar por qué se decidió realizarlas al mismo tiempo.

Lexer

Para el proceso de lexer y tokenización se utilizó el paquete PLY. Se creó un un lexer que consta de tres estados:

- INITIAL
- comments
- strings

Para cada uno de estos estados se definieron las expresiones regulares que representan cada uno de los tokens posibles, y se manejan otras variables que conforman el estado del lexer, como la línea actual.

Parsing

Para el proceso de parsing se utilizó el parser LR1 y la gramática de Cool que fueron implementados para el proyecto de 3er año sobre chequeo de tipos.

Fue necesario modificar la salida del Parser para poder devolver la información referente al token de error en caso de que alguna falla fuera detectada.

Dado que los proyectos llevados a cabo previamente fueron desarrollados para mini-Cool, se hizo necesario modificar la gramática, y se obtuvo como resultado:

Gramática de Cool

La grámatica implementada es S-Atributada. Una descripción de los símbolos y producciones de la grámatica, se puede ver en grammar

Recolección de tipos

Durante la recolección de tipos se visitan todas las declaraciones de clases, se crean los tipos asociados a ellas y se valida la correctitud de las mismas.

Errores detectados:

- Herencia cíclica
- Redefinición de clases
- Nombres de clase no válidos

Construcción de tipos

A los tipos creados en la fase anterior se le añaden todos sus atributos y métodos. Además se verifica que se cumplan los requerimientos de un programa válido de Cool q son tener una clase `Main` con su método `main`.

Errores detectados:

- Problemas de nombrado de atributos y métodos
- Redefinición de atributos
- Redefinición incorrecta de métodos
- Uso de tipos no definidos
- No definición de la clase `Main` o su método `main`
- Incorrecta definición del método `main`
- Mal uso de herencia

Chequeo de tipos

En esta fase se evalúa la correctitud de todas las expresiones del lenguaje y se decide el tipo estático de cada una de ellas según lo establecido en el manual de Cool.

Errores detectados:

- Incompatibilidad de tipos
- Uso de tipos no definidos
- Uso de variables, tipos y métodos no definidos
- mal usos de `self` y `SELF_TYPE`
- mal usos del `case`

Inferencia de tipos

Para la implementación de esta fase se expandió el comportamiento del visitor encargado del chequeo de tipos, razón por la cual ambos procesos se realizan en la misma fase.

Para lograr la inferencia de tipos, se realizó un algoritmo de punto fijo en el cual mediante repeticiones sucesivas del proceso de inferencia se van definiendo los tipos de aquellas variables declaradas como `AUTO_TYPE`.

Idea

Una variable en Cool dada su utilización puede definir dos conjuntos

1. Tipos a los que se conforma (**Ancestros**)
2. Tipos que se conforman a ella (**Descendientes**)

Dados los dos conjuntos anteriores se puede decidir si una variable `AUTO_TYPE` puede ser inferida correctamente o no.

Ambos conjuntos recibieron un nombre intuitivo mencionado anteriormente en **negrita** para hacer referencia a su contenido.

El tipo que se decida otorgar(inferir) a la variable en cuestión, llamémosle ***T***, deberá conformarse a todos los tipos del conjunto 1. Al mismo tiempo todos los tipos del conjunto 2 deberán conformarse a él.

Dicho lo anterior y dado el hecho de que un tipo *A* se conforma a un tipo *B* solamente si *B* es ancestro de *A*, podemos notar que:

1. El tipo a seleccionar debe ser un ancestro del **Menor Ancestro Común** (**LCA** por sus siglas en inglés) a todos los nodos del conjunto 2, llamémosle *N*. En otras palabras el primer tipo que es ancestro de todos los tipos en el conjunto 2.
2. Como todos los tipos del conjunto 1 necesitan ser ancestros de ***T***, todos pertenecerán al camino que se forma desde ***T*** hasta *Object* en el árbol de tipos, por tanto ***T*** necesita ser descendiente del primero que aparezca en el camino mencionado y pertenezca al conjunto 1, llamémosle *M*.
3. Tomando el operador **<=** para referirnos a la relación *ser ancestro de*, se puede afirmar que ***T*** es de la forma ***N <= T <= M***, o lo que es lo mismo ***T*** podría ser cualquier tipo en el camino de *N* a *M*.

> El nodo que representa el **LCA** siempre existe dado que el árbol de tipos es único, por tanto en caso extremo *Object* siempre será válido como ancestro a todos los tipos.

El algortimo implementado tras cada recorrido del **AST**(Árbol de sintaxis abstracta) infiere el tipo de todas aquellas variables de las cuales se tenga información, seleccionando como tipo inferido siempre el que representa a *N*.

Al ser este algoritmo una extensión del chequeo de tipos, mientras se van infiriendo los tipos se valida que los mismos no ocasionen error.

> En todo lo anterior se asume que todo tipo es ancestro y descendiente de sí mismo.

Errores detectados:

- Mal usos de `AUTO_TYPE` en casos donde no se cumpla que ***N <= M*** o todos los tipos en el conjunto 1 no se encuentren en un camino del árbol de tipos
- Todos los errores de chequeo semántico que existan en el código o surgan tras la inferencia de una o varias variables.

Verificación de tipos

Esta fase surge dado que tras el proceso de inferencia puede haber ocurrido un error que durante el chequeo semántico no se valida. Dado que permitimos *AUTO_TYPE* en los parametros de las funciones, al terminar la inferencia pueden generarse conflictos de mala redefinición de métodos, los cuales son chequeados en la fase de Construcción de los tipos (etapa #4). Por tanto la única función de esta fase es verificar la correctitud de los tipos.

Errores detectados:

- Mala redefinición de métodos ocacionada por la inferencia de tipos

Traducción a CIL

En esta etapa del proceso de compilación, requirió especial atención la generación de las expresiones *case*. Para ello se requiere ordenar las instrucciones de tal modo que se asegure el emparejamiento del tipo de la expresión principal con el tipo más específico declarado en las ramas del *case*.

Primero por cada rama **b** se cuentan cuántos tipos declarados en las demás ramas se conforman a **b**, creando de este modo una tupla `(cantidad, tipo declarado en b)`. Luego se ordenan todas estas tuplas por su primer elemento, obteniendo así una secuencia ordenada donde el primero elemento representa la rama cuyo tipo declarado se encuentra en el nivel más bajo en la jerarquía de tipos del programa.

Luego por cada rama **b** de esta secuencia, se obtienen todos los tipos del programa que conforman a **b**, y por cada uno de estos que no haya sido tomado en cuenta en el procesamiento de ramas anteriores, se generan las instrucciones necesarias para comprobar si el tipo de la expresión principal del *case* coincide con él. En caso de coincidencia, se salta al bloque de las instrucciones generadas por el cuerpo de **b**; si no entonces se procede a comprobar con el tipo siguiente. Nótese que no se repiten comprobaciones.

Errores detectados:

- Dispatch estático o dinámico desde un objeto void
- Expresión principal de un *case* tiene valor `void`
- Ejecución de un *case* sin que ocurra algún emparejamiento con alguna rama.
- División por cero
- Substring fuera de rango

> Aunque estos errores realmente se detectan en ejecución, es en esta fase que se genera el código que permite detectarlos.

Traducción a MIPS

En la fase de generación de código `MIPS` se enfrentaron tres problemas fundamentales:

- Estructura de los objetos en memoria.
- Definición de tipos en memoria.
- Elección de registros.

Estructura de los objetos en memoria.

Determinar el modelo que seguirían los objetos en la memoria fue un paso fundamental para la toma de múltiples decisiones tanto en la generación de código `CIL` como `MIPS`. Los objetos en memoria siguen el siguiente modelo:

```
| Tipo | Tamaño | Tabla de dispatch | -- Atributos -- | Marca de objeto |
```

- Tipo: Esta sección tiene tamaño 1 `palabra`, el valor aquí encontrado se interpreta como un entero e indica el tipo del objeto.
- Tamaño: Esta sección tiene tamaño 1 `palabra`, el valor aquí encontrado se interpreta como un entero e indica el tamaño en `palabras` del objeto.
- Tabla de dispatch: Esta sección tiene tamaño 1 `palabra`, el valor aquí encontrado se interpreta como una dirección de memoria e indica el inicio de la tabla de dispatch del objeto. La tabla de dispatch del objeto es un segmento de la memoria donde interpretamos cada `palabra` como la dirección a uno de los métodos del objeto.
- Atributos: Esta sección tiene tamaño **N** `palabras` donde **N** es la cantidad de atributos que conforman el objeto, cada una de las `palabras` que conforman esta sección representa el valor de un atributo del objeto.
- Marca de objeto: Esta sección tiene tamaño 1 `palabra`, es un valor usado para marcar que esta zona de la memoria corresponde a un objeto, se añadió con el objetivo de hacer menos propenso a fallos la tarea de identificar objetos en memoria en el `Garbage Collector`.

Definición de tipos en memoria.

Un tipo está representado por tres estructuras en la memoria:

- Una dirección a una cadena alfanumérica que representa el nombre del tipo.
- Un prototipo que es una especie de plantilla que se utiliza en la creación de los objetos. Cuando se crea un objeto este prototipo es copiado al segmento de memoria asignado al objeto. Un prototipo es un objeto válido por lo que tiene exactamente la misma estructura explicada anteriormente. El prototipo es también la solución escogida para el problema de los valores por defecto de los objetos.
- Una tabla de dispatch que como se explicó anteriormente contiene las direcciones de los métodos del objeto. Existe una tabla de prototipos (nombres) donde se puede encontrar el prototipo (nombre) de un tipo específico, utilizando como índice el valor que representa al tipo.

Elección de registros.

La elección de registros fue un proceso que se decidió optimizar para disminuir la utilización de las operaciones `lw` y `sw` en `MIPS` que como se sabe, añaden una demora considerable a nuestros programas por el tiempo que tarda en realizarse un operación de escritura o lectura en la memoria. El proceso de elección de registros se realiza para cada función y consta de los siguientes pasos:

- Separación del código en bloques básicos:

  Para obtener los bloques básicos primero se hace un recorrido por las instrucciones de la función marcando los líderes. Son considerados líderes las instrucciones de tipo `Label` y las instrucciones que tengan como predecesor un instrucción de tipo `Goto` o `Goto if`. Luego de tener marcados los líderes, se obtienen los bloques que serán los conjuntos de instrucciones consecutivas que comienzan con un líder y terminan con la primera instrucción que sea predecesor de un líder (notar que un bloque puede estar formado por una sola instrucción).

- Creación del grafo de flujo:

  Este es un grafo dirigido que indica los caminos posibles entre los bloques básicos su elaboración es bastante sencilla: si la última instrucción de un bloque es un `Goto`, entonces se añadirá una arista desde este bloque hacia el bloque iniciado por la instrucción `Label` a la que hace referencia el `Goto`; si la última instrucción es de tipo `Goto if`, entonces se añadirán dos aristas una hacia el bloque que comienza con la instrucción `Label` a la que se hace referencia, y otra hacia el bloque que comienza con la instrucción siguiente en la función; en el caso de que la última instrucción sea de cualquier otro tipo, se colocará una sola arista desde el bloque actual hacia el bloque que comienza con la instrucción siguiente en la función.

- Análisis de vida de las variables:

  En este procedimiento se computan cinco conjuntos para cada instrucción **I**: `succ`, `gen`, `kill`, `in` y `out`. `succ` contiene las instrucciones que se pueden ejecutar inmediatamente después de la instrucción **I**; `gen` contiene las variables de las que se necesita el valor en la instrucción **I**; `kill` contiene las variables a las que se les asigna un valor en la instrucción **I**; `in` contiene las variables que pueden estar vivas al llegar a la instrucción **I**, y `out` contiene las variables que pueden estar vivas luego de ejecutada la instrucción **I**.

- Creación del grafo de interferencia:

  Los vértices de este grafo serán las variables que se utilizan en la función y existirá una arista entre los vértices **x** y **y**, si las variables que representan esos nodos interfieren. Dos variables interfieren si existe alguna instrucción **I** tal que **x** pertenezca al `kill` de **I** y **y** pertenezca al `out` de **I**.

- Asignación de registros:

  Contando con el grafo de interferencia, se asignan registros a las variables de forma tal que dos variables que interfieran no se les asigne el mismo registro, esto puede verse como el problema de colorear un grafo con **N** colores siendo **N** la cantidad de registros que

se tienen. Es conocido que este problema es *NP* por lo que para asignar los registros se usa una heurística muy sencilla que consiste en lo siguiente:

```
  Primero se va eliminando del grafo y colocando en una pila cada nodo que tenga menos de N
vecinos, se nota que todos estos elementos pueden ser coloreados sin problemas. Si en algún
momento no existe algún nodo con menos de N vecinos, se tomará un nodo al azar; este proceso
terminará cuando no queden nodos en el grafo. Luego se va sacando cada nodo de la pila y se le
asigna un registro que no esté usado por alguno de los nodos que eran vecinos de este en el
momento en que se eliminó del grafo, en el caso de que existan más de un nodo posible, se le
asigna el menor, en caso de que no exista nodo posible la variable no tendrá registro y su
valor permanecerá en la memoria.
```

Errores detectados:

- Heap overflow

Ejecución

Para ejecutar el proyecto se necesita tener instalado `Python` y el conjunto de dependencias listado en requirements.txt.

Para instalar las dependencias puede utilizar:

```
make install
```

Una vez estén instaladas las dependencias, puede compilar y ejecutar cualquier archivo de código cool utilizando el comando:

```
make main CODE=<path-to-your-code-file>.cl
```

> Para usar `make` necesita estar en la dirección `<project-dir>/src`

Estructura

Los archivos del proyecto se encuentran modularizados de la siguiente manera:

1. **core**
    1. **cmp**
        1. **cool**
        2. **parser**
    2. **lexer**
    3. **visitors**
        1. **type_check**
        2. **cil**
        3. **mips**

cmp contiene todos los archivos heredados de las clases de 3er año y proyectos anteriores.

cool contiene el *AST*, Gramática y Parser de Cool

parser contiene la implementación parser LR1 utilizada

lexer todo lo referente a lexer y tokenización

visitor contiene la implementación del patrón visitor

type_checking fases de la #3 a la #6

cil traducción a cil

mips traducción a mips