COOL – Proyecto de Compilación

Andy Sánchez Sierra C-411

Eduardo Moreira González C-411

Yadiel Felipe Medina C-411

1. Arquitectura del Compilador

Para el desarrollo de este proyecto, se utilizaron los proyectos realizados durante 3er año donde se implementaron las fases de parsing, chequeo e inferencia de tipos.

El proceso de compilación es el siguiente:

1. Lexer
2. Parser
3. Collector
4. Builder
5. Checker/Inferencer
6. Verifier
7. COOL to CIL
8. CIL to MIPS

1.1. Lexer

Para el proceso de lexer y tokenización se utilizó el paquete PLY. 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.

1.2. Parser

Para el proceso de parsing, se implementaron todos los parsers dados en clases (LL1, LR1 SLR1, LALR1), y se utilizó el parser LR1 para realizar dicho proceso.

La gramática implementada es S-Atributada. La descripción de símbolos y producciones es la siguiente:

Terminals: class, type, inherits, id, let, in, isvoid, not, new, case, of, esac, if, then, else, fi, while, loop, pool

program	→ class_list
class_list	→ class_def
	→ class_def class_list
class_def	→ class type { feature_list } ;
	→ class type inherits type { feature_list } ;
feature_list	→ feature feature_list
	→ ϵ
feature	→ param ;
	→ value_param ;
	→ id ( ) : type { expression } ;
	→ id (param_list) : type { expression } ;
param_list	→ param
	→ param , param_list
param	→ id : type
value_param	→ param ← expression
block	→ expression ;
	→ expression ; block
let_list	→ param
	→ param , let_list
	→ value param
	→ value param , let list
case_list	→ param ⇒ expression ;
	→ param ⇒ expression ; case_list
func_call	→ . id ( )
	→ @ type . id ( )
	→ . id (arg list)
	→ @ type . id (arg_list)
arg_list	→ expression

	→ expression , arg_list
member_call	→ id (arg_list)
	→ id ( )
expression	→ special
	→ comparison_expr
special	→ arith ≤ special_arith
	→ arith < special_arith
	→ arith = special_arith
	→ special_arith
special_arith	→ arith + special_term
	→ arith – special_term
	→ special_term
special_term	→ term ∗ special_unary
	→ term / special_unary
	→ special_unary
special_unary	→ isvoid special_unary
	→ ~ special_unary
	→ final_expr
final_expr	→ let let_list in expression
	→ id ← expression
	→ not expression
comparison_expr	→ arith ≤ arith
	→ arith < arith
	→ arith = arith
	→ arith
arith	→ arith + term
	→ arith − term
	→ term
term	→ term ∗ unary
	→ term / unary

	→ unary
unary	→ isvoid unary
	→ ~ unary
	→ func_expr
func_expr	→ func_expr func_call
	→ atom
atom	→ id
	→ bool
	→ string
	→ interger
	→ new type
	→ member call
	→ (expression)
	→ { block }
	→ if expression then expression else expression fi
	→ while expression loop expression pool
	→ case expression of case_list esac

1.3. Collector

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:

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

1.4. Builder

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 que son tener una clase Main con su método main.

Errores:

- 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

1.5. Checker/Inferencer

Para la fase de chequeo de tipos, 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:

- 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

Para la fase de inferencia de tipos 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.

Una variable en Cool dada su utilización se puede definir dos conjuntos

- Tipos a los que se conforma (Ancestros).
- 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.

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.

Dado el hecho de que un tipo A se conforma a un tipo B solamente si B es ancestro de A, se puede notar que:

- El tipo a seleccionar debe ser un ancestro del Menor Ancestro Común (LCA) a todos los nodos del conjunto 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.
- 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 algortimo implementado tras cada recorrido del AST, 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 engriendo los tipos se valida que los mismos no ocasionen error.

Errores:

- 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 surjan tras la inferencia de una o varias variables.

1.6. Verifier

La fase de verificación de tipos 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 parámetros 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. Por tanto, la única función de esta fase es verificar la correctitud de los tipos.

Errores:

- Mala redefinición de métodos ocasionada por la inferencia de tipos

1.7. COOL to CIL

En la etapa de traducción a CIL, se requiere especial atención a 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:

- 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

1.8. CIL to MIPS

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

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

1.8.1. Estructura de los objetos en memoria.

Determinar el modelo que seguirían los objetos en la memoria es 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:

- tamaño: 1 palabra
- valor: interpreta como entero
- indica: tipo del objeto.

Tamaño:

- tamaño: 1 palabra
- valor: interpreta como un entero
- indica: tamaño en palabras del objeto.

Tabla de dispatch:

- tamaño: 1 palabra
- valor: se interpreta como una dirección de memoria
- indica: inicio de la tabla de dispatch del objeto.

La tabla de dispatch del objeto es un segmento de la memoria donde se interpreta cada palabra como la dirección a uno de los métodos del objeto.

Atributos:

- tamaño: N palabras (N es la cantidad de atributos que conforman el objeto)
- valor: cada una de las palabras representa el valor de un atributo del objeto.

Marca de objeto:

- tamaño: 1 palabra
- valor: usado para marcar que esta zona de la memoria corresponde a un objeto

Se añade con el objetivo de hacer menos propenso a fallos la tarea de identificar objetos en memoria en el Garbage Collector.

1.8.2. 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.
- Una tabla de dispatch que como se explica anteriormente contiene las direcciones de los métodos del objeto

1.8.3. Elección de registros.

La elección de registros fue un proceso que se decide optimizar para disminuir la utilización de las operaciones lw y sw en MIPS que como se sabe, añaden una demora considerable al programa por el tiempo que tarda en realizarse una 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 a las instrucciones de tipo Label y las instrucciones que tengan como predecesor una instrucción de tipo Goto o Goto if. Luego, se obtienen los bloques que serán los conjuntos de instrucciones consecutivas que comienzan con dicha instrucción y terminan con la primera instrucción que sea predecesor de esta misma.

- 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: succ, gen, kill, in y out.

  . succ contiene las instrucciones que se pueden ejecutar inmediatamente después de la instrucción.

  . gen contiene las variables de las que se necesita el valor en la instrucción.

  . kill contiene las variables a las que se les asigna un valor en la instrucción.

  . in contiene las variables que pueden estar vivas al llegar a la instrucción.

  . out contiene las variables que pueden estar vivas luego de ejecutada la instrucción.

- 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 elimina del grafo y coloca 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 saca 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:

- Heap overflow