A Compiler for C.O.O.L.

Michael Chao

Introduction

This text will describe the components of constructing a compiler for the Classroom Object Oriented Language or COOL. To accomplish this it will also cover some of the basic elements of COOL as well as the concepts behind their functional implementation.

COOL

Is an acronym for Classroom Object Oriented Language, it was designed as a basic language that was appropriate for teaching compiler construction principles. A complete description can be obtained at http://www.cs.berkeley.edu/~aiken/cool/.

Cool is an expression language where all expressions have values and types. It is a type safe language where runtime type checking is unnecessary. Cool programs are a set of classes which encapsulate their data and function definitions. Each class is its own scope that is declared in the global scope. Methods are sub-scopes of their respective classes class and 'let' statements declare further scopes within functions.

Cool has two identifier classes, types and variables. Types represent the table mapping of a class, and variables represent instances of those types.

Cool has four basic classes String, Int, Bool, and IO. These base classes are predefined because they are core constructs that cannot be defined with the Cool syntax.

Implementation

Parse Tree

The first step in implementing a compiler is to parse its input. This was done by creating a parse tree that would accept any valid input while rejecting syntactically incorrect input. This concrete tree was copied from the Cool specification. Next it is necessary to create an abstract tree for performing type and name analysis. The difference between the two is that the concrete tree is concerned with the textual structure, whereas the abstract syntax tree is concerned with textual meaning. These were both described in an Eli LIDO specification.

Analysis

The second task is to perform name and type analysis on the input file. Name analysis is done to check that the input text corresponds to the scoping rules defined by the Cool specification. In this case variables in the current or upper scope can be accessed but not adjacent scopes. Next type analysis must be performed, which checks that the proper values are returned and passed across calls and in expressions. Since Cool is a type safe language this will ensure that run time errors do not occur.

These tasks also include error reporting for occurrences such as multiply defined identifiers and incorrect types being exchanged. Also inheritance is defined in this module to avoid cyclic inheritance. These tasks were all accomplished using computational roles, provided by Eli, which encompass the common dependency computations.

Translation

Finally this brings us to the tree transformation and the core topic of this paper. To put it simply the tree transformation takes the nodes from the parsed tree and performs structured output depending on the order and type of the nodes. In this case SPIM was chosen to be the target assembly language for translation. Because of this each computation in the tree must be structured in a way that it can translate its particular part without needing external source text. In Eli this is done with dependencies, properties, and chains.

- Chains define a left to right computation order within a tree.
- Properties are values shared among common nodes such as defining and applied occurrences.
- Dependencies specify that a computation is to take place before another computation.

In this case these were combed with predefined SPIM macros as well as the PTG output of assembly code that was more difficult to define. The combination of the two was used because the macros made the translation definable at a higher level, but simultaneously did not provide some features necessary to execute certain operations associated with object oriented languages.

Translation Methodology

The methodology for attacking this problem was to produce the output as it would be executed by a SPIM processor. In this case that started with setting up the class structure. Because Cool is an object oriented language all variables are pointers to objects that hold its internal data. The definitions for these objects are class definitions, which makes the defining, referencing, and allocating objects a reasonable place to start.

As an extension of class definition it was necessary to do some setup for features, or functions as they are more commonly called. This involved placing stack and frame pointers on the stack, along with the arguments the function was passed.

Next certain basic arithmetic operations were defined. The reason being that they were the simplest expressions that could be defined.

Next the basic identifier nodes such as string and integer would be defined. These computations are necessary to set up and initialize the objects that would represent the variables within the tree.

At this point it was necessary to include code that defined the built in basic objects. These objects are needed for built in classes that would be impossible to define with the Cool language. These would handle tasks like object copying and output.

The last step but most involved would be to define the more advanced operators, such as let, if and while statements, that require more computations. These could then be debugged using the existing functionality of the compiler, and errors quickly diagnosed.

Note that the PTG output was used for object initialization and class definitions, more abstract operations that the existing macros did not support. The translation stage was done in the order of personal whim, and should be changed to whatever the programmer feels is the simplest.

Detailed Analysis

Name Analysis

Name analysis consists of inheriting Eli roles onto tree nodes. These then perform computations necessary to assure the definitions and uses are correct with respect to scope. There are six basic names that need roles, these are attribute declarations and uses, function definitions and uses, and type declarations and uses. There is also two more roles that are special cases of type declarations and uses. One is for inheritance, because roles must be associated with type definitions that will inherit from another type, and in Cool not all types must explicitly inherit. The second is a special case of type use, that is the ‘self’ identifier. This requires some special setup since it is distinct from a new instance of the current class.

Scopes are defined using computational roles that are associated with certain nodes in the tree. In this case Class, feature, and let have scopes. When variables are multiply defined within a scope, or referenced in an incorrect scope, appropriate errors are reported.

Type Analysis

Type analysis consists of ensuring that the input file deals with checking the input file to make sure the correct types are being declared and passed.

Base Type Analysis

This involves propagating types through the tree, and having Primary contexts when types are set at leaf nodes of the tree. This does the task of checking that variables are assigned appropriate types, as well as assigning types to the basic terminals of the tree i.e. Int, Bool, and String. This prints out appropriate errors when the types do not match.

Parameter Type Analysis

Since it is necessary to check that the arguments of a function call are correct and there is no direct relation between the formals of the function call and the arguments of the definition in the tree, this form of type analysis is distinct and necessary. This is conceptually done by creating a map of the function formals and then relating that to the function arguments. This is physically done with computational roles and the procedure ProcOperN, where CallContext does the actual type checking at the function call. This also checks the number of arguments, and prints out appropriate errors.

Detailed Translation

Classes

With classes it is necessary to construct a valid class tag with the structure of.

offset -4	Garbage Collector Tag
offset 0	Class tag
offset 4	Object size
offset 8	Dispatch pointer
offset 12 …	Attributes

Figure 1

Where a good deal of the information must be colleted from other nodes in the tree. This is done with chains computing offsets in left to right recursive order. After the chains are run it is possible to know the object size, once this has been set and the arbitrary class and garbage collector tags formulated other chains are sent down the tree to fill out the list referenced by the dispatch pointer, and to store references to the attributes default values. Note that this is only the stored template of a class, and that in run time this template must be copied and initialized in order to be viable for computation. While the classes themselves are referenced with labels the function definitions have a property that is the index of the function. This is used to reference the function during runtime relative to the dispatch pointer. PTG functions are used for this portion of the translation as there are no macros that contain the appropriate commands.

Function Call

Functions require that they be insulated from the code that called them, while their arguments are made locally available. Several tasks must be performed to do this. First it is necessary to store the registers that the function may alter in order to isolate it from the rest of the runtime code. Second it is necessary to set certain registers necessary for function execution. The first is accomplished by storing the necessary registers on the stack. The second involves setting the frame pointer to a relevant location on the stack and putting references to all the arguments that are passed to the function. The arguments are again done with a chain, in left to right order, where offsets are again computed for their retrieval from the stack.

Function Definition

PTG functions are used here to set up the function reference table. This associates the offset as a property of the function definition, which is then accessible in the function use.

Conditional

The 'if else' requires conditional jumps based on the value of a Bool that is the first expression. This is a relatively simple task, where the value of the Bool object must first be collected and then a branch on its value (1 or 0) to the labels associated with the following two expressions.

Loop

Looping takes the form of two expressions that must be evaluated at each iteration, the first of which determines weather the iteration continues. This is done by executing each expression starting with the first, and jumping on the value of the first. This is a lot like the conditional statement with the exception that instead of exiting after the second expression is evaluated you return to the beginning.

Blocks

Blocks are expression constructs that allow statement style computations within Eli. Instead of the value of all the statements within a block being propagated up the tree only the last expressions value is saved. This is done by executing the expressions in order and sending the result of the last one as the result.

New

New initializes a new object and then equals its reference. This is used to initialize objects for storage. To do this built in Cool functions are called to copy the objects template and initialize the object. Once this is done all attributes of the 'New' function will themselves reference new objects. The object will then be ready to store and execute its features.

Isvoid

This is a construct that checks to see if an object is initialized. This is as simple as following the variable reference and checking that the garbage collector tag has been set. Therefore this expression branches on the value of the garbage collector tag, to either create a true Bool object or a false Bool object.

Parsed Object Values

Instead of simply storing input values from Cool code as expressions, it is necessary to create a new instance of its associated type, and set its value to the leaf node of the tree. This takes the form of creating a new object of the parsed type and then setting its internal value equal to the leaf nodes value. This will make a valid expression that can be used in other expression constructs.

Conclusion

The aforementioned are the basic tasks to construct modern compiler. While the source or destination language could change this process would not change given an object oriented language.