Final Report: GoLite and LLVM

Xavier Denis

April 14, 2015

1 Introduction

While many languages sport a simple design which helps simply the task for the compiler designers, Go attemps to make as many exceptions. To combat the insanity I chose to implement everything Haskell compiling to LLVM bytecode. The type safety and pattern matching support in Haskell allowed for rapid and correct development of the compiler over the course of this semester.

On the output side of the equation, support for LLVM allows for an easy interaction between the Haskell AST and the final compiled LLVM code. This project allowed me to explore the many problems faced by compilers especially within the context of strictly typed functional languages. I was forced to submerge myself in monads and learn to properly structure Haskell projects.

2 Key Concepts

While compilers written in Haskell share many common points to compilers written in other languages, there is some Haskell terminology which can mask those similarities.

2.1 Monads

Monads, the primary cause of strife between functional languages and the rest are ubiquitous in Haskell. While it is true that they are simply "monoids

in the category of endo-functors", it is also true that such a definition helps no one. The key idea behind monads is to seperate the environment in which computation happens from the actual computation. Many smarter people have written explanations of the laws of monads and how they work so I will leave it to them. Instead, I will cover a few of the important monads for this project

2.1.1 IO

The 'IO' monad is most likely the most widespread monad, it seperates the outside world from the code inside. This can cause issues not common in imperative or OOP languages where printing is allowed anywhere. While any value can be wrapped in the IO monad to allow for printing inside, doing so generally increases the complexity of the code unnecessarily. In practice it becomes simpler to compute the value to be printed within non-IO code and pass it back out to be printed at the very end in an IO monad. Within the compiler, the IO monad occurs in the Main module and the code emission module.

2.1.2 Either

Many programmers have heard about so caled 'option types' or the Maybe monad which allow for a value to either be there or not. In many cases though, it isnt enough simply know that the value is missing, instead often times, the programmer would like to return some sort of error message or data back out. The 'Either' monad is an extension of the 'Maybe' monad which represents abortable computation. An Either value can be a Right, which represents a correct value or a Left, which represents an error of some kind. The key advantage of Either is that if at any point of the monadic computations a Left is returned, the rest of the computation is aborted. This dramatically simplifies the handling of errors since only the caller needs to check for error values. These types are found all over the code most notable during parsing, weeding and typechecking.

2.1.3 State

Finally, many computations require state to complete successfully, this is where the state monad comes in. The state monad provides access to an

arbitrary type which is used to encode the relevant state. This is especially useful when traversing the AST in a stateful manner for typechecking or code generation. Knowing which loop a 'Continue' is in allows to the correct jump locations to be set.

2.1.4 Other Monads

While there are many other monads strewn about the code, they are generally all variations on the above monads with some use case specific bits thrown in.

2.2 Type classes

Type classes are the method used by Haskell to support ad hoc polymorphism. Typically, functions are unique in name up to module scope. However, type classes allow for function overloading. Typically, these are used to describe a behaviour shared by distinct types. In the scope of this project, it mainly applies to various passes such as Weeding, Pretty Printing and Typechecking. Each of these is implemented as a typeclass allowing other functions to operate solely on the class specific information. Addtionally, providing instances of these classes for commonly occuring monads such as the Maybe and List monads means that they can be traversed transparently.

3 AST Types

For a language like Haskell, the choice of types matters a lot when designing programs. Properly designing the data types allows for a more natural deifinition of methods to emerge. The AST types represent the overall structure of a go program, divided into 5 groups:

1. Package: A simple wrapper over the list of declarations with a name.
2. TopLevelDeclaration: These encompass the global delcarations. However, there is a special construct 'Decl' which takes a Statement as a parameter. The purpose of this constructor is to allow global type and variable definitions.

3. Statement: Statements are mainly flow control related but they also include type and variable declarations.
4. Expression: Go expressions including both L and R values.
5. Type: The type of types, represents GoLang types in an abstract manner used both for typechecking and codegen.

There are several supporting type constructors that are used to abstract some other differences such as anonymous fields and arithmetic operation types.

3.1 AST Typing

While this set of types captures the structure and content of Go programs, there is a problem posed by typechecking. After verification of proper typing, we'd like to encode that information somehow into the actual AST. This is achieved through higher kinded types, that is, by changing the AST types to take a second type as a parameter. At the same time there is still a need to encode the lack of types before checking. To solve this the parsers return types parameterized with the unit type, '()', the typecheckers produce a new ast typed using the 'Ann' type which holds the various typechecking results. This also means that functions can be written to only process various stages of the compilation. As an example all the code generating functions expect an AST parameterized by 'Ann', that is one that has been typechecked.

4 Parser

Unlike the approaches to parsing and lexing covered in class, functional languages tend to dislike meta-programming and generated code. This means that parser generators are generally not the recommended tools to implement parsers within those languages. Instead, parser *combinators* allow the programmer to easily compose and interact with the parsers. These combinators allow for the lexing and parsing to happen at once, producing the full AST as a result.

4.1 Parser Combinators

The Parsec library is by far the most widely used library for parser combinators. It provides access to these combinators through the 'Parser' monad, which allows for the composition of parser actions.

Below is the parser for simple statements in go, a restricted set of statements which can occur in 'if', 'switch', and 'for' statements as part of the clauses.

```
simpleStatement :: Parser (Statement ())
simpleStatement = try $ incDec <|> assign <|> shortDec
  <|> opAssign <|>  exprStmt
```

The parser is itself the combination of several other parsers. By using Haskell's '< | >' operation, we can perform a choice between two different choices. Combining this with the 'try' function we have backtracking which allows for many different choices to be tried without failure. The base elements of the parsers are various character parsers, from these, progressively more complex parsers are built up allowing for a vocabulary describing go to emerge.

4.2 Weeding

While parser combinators could allow for weeding to happen in the same step, it is much simpler to simply write a recursive descent on the outputted AST to do the final weeding. This is implemented as the 'Weed' typeclass and uses the State monad discussed above, it checks for things such as 'Continue' and 'Break' being in correct locations.

5 Typechecking

Typechecking for golite is a relatively simple task, the most complex part being the rules for various binary operators. Typechecking again exploits typeclasses to encompass the desired behaviour. It also makes very heavy use of the State monad to handle the symbol table issues, implementing a form of scoped state.

The minimal definition of the 'Check' typeclass is given as:

```
class Checkable a where
    check :: (a ()) -> Check (a Ann)
```

The behaviour of this function is also clear from the type, it takes an untyped AST and produces a monadic, typed AST. A second function, 'typecheck' unwraps the monad to give us the final AST or the appropriate error information.

For actual typechecking there are a few important types and functions. The current stack of scopes is represented in the 'Env' datatype which is a list of maps from string to Type. The 'getvar' method performs a recursive lookup on the list, traversing it until the value is found, if nothing is found a type error is propagated back. The 'pushScope' and 'popScope' methods are used to modify the environment during checking.

5.1 Type Annotation

As part of the output, we rewrite the type information in the AST to carry meaningful values. We record two values, the user type, which is the type given in code and the raw type which is the type after being broken down to primitives.

5.2 Issues: Binary Operations

This less of an issue with the specific choice of language or implementation and more with the actual language design of go. The section on operator compability is fairly compelx and full of exceptions. The result is that operator compatibility is checked using several truthtables and pattern matching. Since operators can transform the type of their output, another truthtable is used to determine the correct output type. This sollution is tedious to maintain and implement, though luckily the language is not likely to change.

6 LLVM Code generation

The target language for this project is the LLVM intermediate representation. LLVM provides a large and mature set of tools for manipulating,

optimizing and working with this IR. Targeting this platform allows for a highly performant result to be generated.

6.1 LLVM

The LLVM IR is a fairly simple language which remains flexible enough to express most of the source constructs. There only a few major differences. For example, only struct types are valid as declarations, no aliasing of primitive types is supported. Since the AST annotations carry both the aliased name and the raw type, this does not matter much for us. Secondly, IO must be provided through externs, this means that some helpers needed to be written directly in the IR to ensure the link between the GoLite print functions and the actual OS level calls. Additionally, all memory must be managed manually through use of 'alloca' and 'malloc'. Finally, the biggest issue is that LLVM IR must be in SSA form, which renders mutability challenging. To get around this, an optimization pass called 'mem2reg' is used which promotes stack allocations to registers.

6.2 Code generation

To interact with LLVM FFI bindings are provided through the Haskell library LLVM General. To use this library the AST is transformed into an LLVM AST which is then run through optimization passes and outputted to LLVM byte and bitcode.

LLVM differs from Go in a few major ways which require processing to resolve. The first is that global (top-level) and local values are quite different and don't collide. All local values are scope to the function, meaning that if a function has multiple variables with the same name then they must be renamed. Functions also have a different structure, with the body specified by a list of basic blocks.

On the Haskell side of things, this gives rise to two seperate, nested state monads which are used handle the code generation. The outer monad, called LLVM uses an LLVM.General Module as it's state. This means that subsequent operations with that monad modify the same code module for LLVM. This monad is used at the top level to encompass all the function declarations.

The second, inner monad is the Codegen monad. This monad holds the state relevant to function declaration. Since the vast majority of the complexity of GoLite is held by functions, the state record is significantly more complex. A few of the important operations performed by this monad include:

1. Tracking Blocks: A list of blocks and which block is current being modified
2. Variables: A list of references to current variables. This is used to access variables as they are required.
3. Bubbling Type Declarations: Since LLVM can only declare types at the top level, Codegen saves a list of types which need to be declared at an outer scope.
4. Continue, Break, Fallthrough: Each of these statements needs a label to jump to, so Codegen tracks that.
5. Name Supply: Since all variables need to be unique in a function, a supply of names tracks how many times we've used a specific string as a variable name.

The Codegen module provides many helper functions to produce the correct LLVM IR. These functions lift the IR instructions into the Codegen monad and simplify the overall work dramatically. They are used by the Emit module which actually generates code.

The Emit module implements yet another recursion through the tree to output the various code templates that represent the mapping of Go constructs to LLVM IR. The combination of the Codegen helpers and Haskell's support for monads means that the templates take a form very similar to the final IR.

6.2.1 Difficult Constructs

While LLVM's instruction set is vast, there are still some constructs in Go which require special handling to correctly capture.

While LLVM has a native 'switch' instruction, it can only switch on constant values. This means that Expression Switches required special handling. Instead, it is evaluated as a linked series of conditional branches. If one test fails it passes onto the next branch but if it succeeds it jumps to the body of

the case. Similarily a fallthrough statement will perform an unconditional jump to the next case statement.

The GoLite for loops also pose a problem. Because there are two major types of loops, which differ vastly in their structure and meaning, they must actually be implemented as two seperate cases. The typical ForClause loops needs to perform pre-initialization a test and a post increment during the loop. On the other hand the Conditional loop or "while" loop instead only performs the precheck with none of the post conditions.

6.3 Performance

Suprisingly the outputted code acheived performances consistently higher than the official go compiler. In cases where there was no output, the generated code would eliminate all computuations since the optimizer recognized that none of the calculations matter. On more realistic programs it wasn't unusual to see a 10% increase in speed over mainline go. The optimizations passed used represent a very small subset of the overall optimizations available in LLVM project, using the entire set, even better performance is certainly in reach.

7 Conclusion

Implementing a compiler in Haskell proved to be a challenging and rewarding experience. The power and expressivity of the language allow for me to have huge power and expressiveness reducing the overall workload. Learning to develop and structure fairly large applications in Haskell was an interesting challenge as was dealing with deploying and packaging the build environment. Developing the correct workflow and development practices significantly improved the speed of development and code correctness as the project moved on. This project also provided me with an opportunity to work with LLVM a huge and very interesting project. I developed a much greater understand of the process that code takes on its path to execution, and the interaction with OS level libraries and support functions. An unexpected effect of this project was the amoujnt of hate that I developed for Go and its toolchain. As I was working on this project I occasionally attempted to consult the LLGO project for tips on how to translate go features to LLVM bytecode. The project, itself written in Go is an incomprehensible

mess involving structs of functions, builtings written in 3 languages and poorly labelled source files. If I learned any lesson over the course of this project it was that having generics is critical to maintainable code in any serious language, without generics there is no way to easily express structures like slices or maps.

This project was easily the most enjoyable one I had in my time at McGill, made especially so with the help of Vincent. Yet, if I could have changed anything it would have been to cover liveness earlier so that deallocations would have been simpler in LLVM.

```
func main() int {
  var x = 2
  for y := 0; y < x; y += 2 {
    x++
  }
  println(x)
  return x
}
```

This input function is transformed to the code below

```
; Function Attrs: nounwind
define i64 @main() #0 {
for.body.lr.ph:
  br label %for.body

for.body:
; preds = %for.body.lr.ph, %for.body
  %0 = phi i64 [ 0, %for.body.lr.ph ], [ %3, %for.body ]
  %1 = phi i64 [ 2, %for.body.lr.ph ], [ %2, %for.body ]
  %2 = add i64 %1, 1
  %3 = add i64 %0, 2
  %4 = icmp slt i64 %3, %2
  br i1 %4, label %for.body, label %for.end

for.end:
; preds = %for.body
  tail call void @print.tinteger(i64 %2)
  tail call void @print.trune(i8 10)
  ret i64 %2
}
```