ECE491 Advanced compilers final report
Implementing a lazy functional language

Jonathan Lam

2022/05/12

Contents

	Purely functional	Non-purely functional
Untyped	Core	Scheme[1]
Typed	Hazel[2]	C[3]

Table 1: Summary of programming language implementation projects

1 Project overview

1.1 Motivation and overview

This project was the semester-long project for the ECE491 independent study on advanced compilers. It is the implementation of a simple lazy (normal-order) evaluation functional language similar to Haskell. The work follows the tutorial, "Implementing functional languages: a tutorial" by Simon Peyton Jones at Microsoft Research [3]. This was published two years after Haskell 1.0 was defined in 1990 [1], to which SPJ was a major contributor.

This project is the culmination of my studies in programming languages and functional programming over the past two years. These studies include writing a C compiler in C (Spring 2021), a Scheme (LISP) interpreter in Scheme (Summer 2021), and this project, a Core interpreter and compiler in Haskell (Spring 2022). Additionally, my Master's thesis (2021-2022 school year) was about updating the evaluation model of Hazel, a programming language implemented in OCaml. We summarize these languages in Table 1.

1.2 Commentary on the tutorial

The tutorial used for this project assumes a basic understanding of functional programming and a non-strict language such as Miranda or Haskell. For this report, we do not assume this and give a brief introduction to these ideas.

The text is largely in tutorial format, in that it provides much of the code for the reader to follow along with, but it is also in large part similar to a textbook. While much of the base code is given, much of the implementation is left in the form of non-trivial exercises to be completed using the provided theory. As the book progresses, less code is provided directly; rather, the semantics are given using the state transition notation, and the reader is asked to implement this in code.

The structure of the tutorial goes as follows: Chapter 1 introduces the Core language. Chapter 2 provides an evaluator implementation called the Template Instantiation (TI) evaluator, which we will describe in Section 4. Chapter 3 provides a compiled version of the TI evaluator, called the G-Machine[4] (GM), which we will describe in Section 5. We note that Haskell officially uses a Spineless Tagless G-Machine (STG) [2].

[1] Mostly functional
[2] Gradually-typed
[3] Weakly-typed
[4] "G" for "graph," most likely.

Chapter 4 describes the Three-Address Machine compiled implementation, and Chapter 5 describes the Parallel G-Machine. The G-Machine (a stack-based machine) may be translated into a TAM representation. Chapter 6 introduces the λ-abstraction syntax using the λ-lifting program transformation. Chapters 4-6 were not covered for this independent study.

Rather than developing each implementation vertically (e.g., finishing the lexer, then the parser, then the intermediate representation, then exporting opcodes), the tutorial uses a horizontally incremental development method. In this style of development, we first complete a base implementation of only the core language features, and then incrementally add support for new language features. The tutorial calls these horizontal versions "marks," i.e., "TI Mark 3" or "GM Mark 7."

I greatly enjoyed the tutorial and found the exercises delightfully challenging and enlightening. My only complaints with the tutorial are that sometimes the exercises seemed to require knowledge from future marks, and that the names of certain concepts seem to be arbitrarily named[5].

1.3 Implementation setup

My implementation is called `flc`, short for "fun(ctional) lazy compiler." `flc` is written in Haskell, similar to the tutorial[6]. Instructions for use may be found on the GitHub's README.

The implementation may be found at jlam55555/fun-lazy-compiler. A transpiler for the GM Mark 1 to x86-64 assembly code may be found at jlam55555/flc-transpiler.

[5] For example, the compilation schemes are named with arbitrary letters, and the "G" in "G-Machine" and the "I" in "`Iseq`" are never explained.

[6] The tutorial says it was written in Miranda, but I am not sure if this is correct. All of the code samples run successfully in Haskell, and I believe that Miranda's syntax is somewhat different.

$$e ::= \lambda x.e \mid x \mid e\ e$$

Figure 1: Grammar of Λ

2 Background

This section provides brief background on programming language theory from a functional perspective.

2.1 Definition of a programming language

Interfaces are necessary for efficient and effective communication. A *programming language* serves as an interface between humans and computers. To rigorously work with a programming language, we define its *syntax* and *semantics*.

The syntax of a programming language describes the way valid expressions and programs are formed. It is specified using a *grammar*. The grammar for the untyped λ-calculus Λ is shown in Figure 1.

The *semantics* of a programming language describes its behavior. The *static semantics* denotes the behavior of processes that happen prior to evaluation, such as type checking. The *dynamic semantics* describes the behavior of *evaluation*. Evaluation is the process of reducing an expression down to a *value*, or irreducible expression.

For this project, we actually define two languages: the Core language, and the abstract

2.2 Implementations of programming language

TODO: compilers, interpreters

2.3 The untyped λ-calculus

2.4 Functional programming

TODO: notation with spaces for function application and curried-by-default application

TODO: strict languages like the ML family

TODO: haskell and miranda

TODO: pure functional programming and lack of side effects

TODO: algebraic datatypes

3 The Core language

3.1 Terminology

TODO: lazy evaluation (call-by-value, call-by-name, call-by-need, applicative-order, normal-order), supercombinators, currying, lambda-abstraction, function application, ADTs (product and sum types) and structured data, whnf, standard representations (booleans, list constructors), standard definitions/prelude (e.g., if, simple combinators, standard representations), scrutinee, definiens, opcodes/instructions, runtime

3.2 Syntax

3.3 Dynamic semantics

TODO: function application and basic forms have the same dynamic semantics as the lambda calculus, except using an environment to store variable bindings rather than using substitution

3.4 Sample programs

TODO: show basic forms
- **TODO: twice twice twice**
- **TODO: show algebraic datatypes**

3.5 Lexer

TODO: very simple tokenization

3.6 Parser

TODO: build up LL(1) parser from useful subparsers

3.7 Pretty-print utility

TODO: constant-time append with indentation, only linear-time rendering

4 Template instantiation (TI) evaluator

TODO: only cover this briefly, relation to the GM machine

4.1 The TI abstract machine

TODO: process: instantiate and unwind

4.1.1 Instantiation

TODO: conditionals

4.1.2 Unwinding

4.1.3 Memory model

4.2 Sample evaluations

TODO: sample updating
TODO: sample lazy evaluation
TODO: tco
TODO: arithmetic

5 G-Machine (GM) implementation

The G-Machine is an abstract machine with a very similar idea to the TI machine with graph reduction and unwinding. The difference is that the graph creation, which is a complicated process performed by the instantiation step of the TI machine, is compiled into a series of opcodes on a stack-based machine. The unwinding step remains largely the same. The state of the machine is largely the same, with a stack, dump, and heap. Notably, this also includes an instruction queue (which replaces instantiation), and supercombinators are compiled to a sequence of instructions.

This stack-based machine is desirable for several efficiency reasons: it requires a simpler (smaller) runtime; it does not require the re-traversal of a supercombinator's body expression on each expression; and the runtime does not require an expression-level representation of the language.

5.1 List of opcodes

The list of opcodes of the abstract stack machine that the G-Machine compiles to is given below. For the sake of this document, the dynamic semantics of each opcode is only provided informally[7]. A precise definition of most of the opcodes are given in the tutorial[8].

`Pushglobal` f Push the address of the global (supercombinator) f onto the stack[9].

`Pushint` n Push the address of an integer node representing the integer n onto the stack. If such an integer node representing the integer n already exists, this may use that existing address; otherwise, this will allocate a new integer node n.

`Push` n Push the n-th address of the stack onto the stack.

`Update` n Set the node pointed to by the n-th address of the stack to an indirection node pointing to the top element of the stack, then pop one element from the stack.

`Pop` n Pop n elements from the stack.

`Alloc` n Push n invalid addresses onto the stack, or decrease the stack pointer by n [10].

[7] The syntax of this assembly-like language is trivial and need not be stated.

[8] Most of the time, the state transition(s) of an opcode are provided, and the implementation of the opcode in Haskell is left as an exercise.

[9] This is overloaded in Mark 6 for a technicality regarding unsaturated data constructors. This is Exercise 3.38.

[10] This is used for `letrec` expressions. These addresses initially point to an invalid node, and will be updated by the evaluation of the definiens.

`Slide` n "Slide" the top element of the stack up n slots. In other words, the $(n+1)$-th element of the stack will be replaced with the top of the stack, and then n elements will be popped.

`Unwind` Same as in the TI implementation Section 4.1.2.

`Mkap` Pop two elements from the stack, allocate a new application node (with the first popped element in function position, and the second popped element in argument position) and push the application node onto the stack.

`Eval` Strictly evaluate the element on the top of the stack to WHNF on a new stack (frame)[11].

`Add`, `Sub`, `Mul`, `Div`, `Neg` Arithmetic binary and unary operators. Assumes the two elements on the top of the stack are evaluated to integers (otherwise will crash the program). Pops the top two elements and leaves a number node on the top of the stack representing the arithmetic result.

`Eq`, `Ne`, `Lt`, `Le`, `Gt`, `Ge` Relational binary operators. Assumes the two elements on the top of the stack are evaluated to booleans in the standard representation (otherwise will crash the program). Pops the top two elements and leaves a node on the top of the stack representing the boolean result in the standard representation.

`Pack` t n Pops n addresses from the stack. Allocate and push a structured data node representing structured data with tag t and the n popped arguments.

`Casejump` *rules* *rules* is of the form $\{t_j \to [i_1, i_2, \dots]\}$: a mapping of tags t_j to code sequences $[i_1, i_2, \dots]$. Assumes the top of the stack is the scrutinee of a `case` expression, and thus a data node with tag t. Appends the instruction list of the appropriate rule onto the current instruction queue. Throws an error if the scrutinee is not a structured data node or if there is no match.

`Split` n Assumes the top of the stack is a structured data node. Pops it from the stack, and then pushes its n arguments onto the stack.

`Print` Output the top element on the stack. Assumes that it is a data node (integer or structured data node). Recursively generates additional `Print` and `Eval` opcodes for structured data.

5.2 Compilation schemes

TODO: list each compilation scheme, describe it, show its behavior on expression types

[11] This pushes a new stack on the dump and evaluates the node on the new stack. This is more or less equivalent to a function call in the conventional procedural ABIs.

5.2.1 Non-strict vs. strict compilation context

TODO: fit this in nicely with the previous section???

5.3 Evaluator

The evaluator implements the abstract stack machine. Each of the opcodes described in Section 5.1 is implemented as a separate function in the G-Machines's `Evaluator` module.

5.3.1 Transpilation to x86-64 assembly code

To illustrate the simplicity of the G-Machine's runtime, the stack machine of the Mark 1 of the G-Machine was implemented in NASM x86-64 assembly code. Each stack machine opcode was implemented using an assembly macro. Supercombinator node definitions are also implemented with a macro. This can be found on GitHub at jlam55555/flc-transpiler.

I did not have time to implement later Marks of the G-Machine in assembly code, and I believe that it is a reasonable project for future work. However, the increased complexity of later Marks will likely be convenienced by small helper functions written in C.

5.4 Sample compilations and evaluations

TODO: this section

6 Future work

6.1 Garbage collection

For a functional language, garbage collection in the runtime is very important, since nodes are automatically allocated without control of the user. Garbage collection is covered in the last part of the TI implementation after Mark 6. It was not implemented for sake of time.

6.2 Three-address machine and parallel G-machine

These are two additional implementations of the compiler that are covered in future chapters of the tutorial. Both are very intriguing: the three-address machine is closer to the architecture of many modern CPU architectures, and inherently parallelized reduction is also an interesting idea.

6.3 Particulars of the STG and Haskell's implementation

It will be interesting to study the intricacies of Haskell's implementation. Haskell, despite being a high-level, lazy functional language, is known to be very fast to the point of bafflement[12], with optimized GHC-compiled programs usually only 2 to 5 times slower than gcc-compiled programs[13].

We've seen that the graph-reduction mechanism in the TI machine is relatively intuitive, but there is significant complexity in the construction (instantiation) of graphs. The G-machine achieves some efficiency by compiling the construction of a graph to code. It will be interesting to see additional optimizations that allow GHC to achieve its famed performance. While this tutorial is meant to be an introductory tutorial on such machines, it should be formative to read documents specifically about the design of Haskell, such as [2].

6.4 Compilation to native binaries

We achieved compilation of the Mark 1 G-machine to native binaries using assembly macros and the NASM assembler, as described in Section 5.3.1. We leave for future work the same work for the Mark 6 G-machine, which is significantly more complicated.

However, it may be more fruitful to attempt this after reading the chapter on the three-address machine, since this chapter should describe the translation from the stack-based machine to a three-address machine. Transpiling the three-address machine to x86-64 opcodes should also be a simpler process.

[12] `https://stackoverflow.com/q/35027952`

[13] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/ghc-gcc.html`

7 Conclusions

This tutorial teaches how to efficiently evaluate a pure lazy functional language. First, an intuitive method involving lazy graph reduction called the Template Instantiation (TI) evaluator is introduced. The runtime of the TI comprises alternating graph creation (instantiation) and reduction (unwinding) processes. The efficiency of the runtime is improved by compiling the graph creation process into a series of opcodes in an stack machine, called the G-machine.

For this project, I was able to complete both base implementations and achieve the desired results. There is still much of the tutorial that has not been implemented, such as garbage collection, the λ-lifting program transformation, and other implementations such as the three-address machine and the parallel G-machine.

8 References

[1] Paul Hudak et al. "A history of Haskell: being lazy with class". In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. 2007, pp. 12–1.

[2] Simon L Peyton Jones. "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine". In: *Journal of functional programming* 2.2 (1992), pp. 127–202.

[3] Simon L Peyton Jones and David R Lester. "Implementing functional languages: a tutorial". In: *Department of Computer Science, University of Glasgow* (2000).