ECE491 Advanced compilers final report
Implementing a lazy functional language

Jonathan Lam

2022/05/12

Contents

1 Project overview **3**
- 1.1 Motivation and overview . . . 3
- 1.2 Commentary on the tutorial . . . 3
- 1.3 Implementation setup . . . 4

2 Background **5**
- 2.1 Definition of a programming language . . . 5
- 2.2 Implementations of programming language . . . 5
- 2.3 The untyped λ-calculus . . . 5
- 2.4 Functional programming . . . 5

3 The Core language **6**
- 3.1 Terminology . . . 6
- 3.2 Syntax . . . 6
- 3.3 Dynamic semantics . . . 6
- 3.4 Sample programs . . . 6
- 3.5 Lexer . . . 6
- 3.6 Parser . . . 6
- 3.7 Pretty-print utility . . . 6

4 Template instantiation (TI) evaluator **7**
- 4.1 The TI abstract machine . . . 7
  - 4.1.1 Instantiation . . . 7
  - 4.1.2 Unwinding . . . 7
  - 4.1.3 Memory model . . . 7
- 4.2 Sample evaluations . . . 7

5 G-Machine (GM) compiler **8**
- 5.1 The GM abstract machine . . . 8
  - 5.1.1 List of opcodes . . . 8
- 5.2 Compilation schemes . . . 8

5.3 Evaluator 8
5.4 Sample compilations and evaluations 8

6 Future work **9**

7 Conclusions **10**

8 References **11**

	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)

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) compiler

5.1 The GM abstract machine

5.1.1 List of opcodes

TODO: list each opcode, describe it, show its behavior using the state machine

5.2 Compilation schemes

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

5.3 Evaluator

TODO: the evaluator performs the opcode's descriptions; maybe show its behavior using state machine here

5.4 Sample compilations and evaluations

6 Future work

TODO: garbage collection

TODO: last marks of TI and GM

TODO: 3-address machine and parallel g-machine

TODO: lambda lifting

TODO: study implementation of STG

7 Conclusions

TODO: got things to work as expected

TODO: learned a lot about lazy evaluation

TODO: lots of future work to do still

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).