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 [5]. This was published two years after Haskell 1.0 was defined in 1990 [2], 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) [4].

[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.

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*. For example, 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 define two languages: the Core language, and the abstract stack-based language of the G-machine. The semantics of the Core language is, like many functional languages, highly based on the λ-calculus. The semantics of the stack-based G-machine language is similar to other stack languages such as Forth.

2.2 Implementation(s) of a programming language

A programming language is a specification. There may be multiple *implementations* of a language. The specification and implementations are orthogonal. A single language may have multiple implementations with different performance characteristics and language support. For example, the Core language has four implementations proposed in the tutorial (TI, GM, TAM, PGM); other major languages such as the JVM, Python, or Haskell also have multiple implementations.

We typically classify an implementation into one of two broad classes: *intepreters* (a.k.a. *evaluators*) or *compilers*. The difference is that interpreters take a program's source code and runs it "directly." On the other hand, a compiler performs a two-stage process: compiling to a low-level representation, to be evaluated at some later time. Compiled representations tend to perform better and require simpler *runtimes*. The distinction between interpreters and compilers may not always be clear; in some cases, they form a spectrum delineated by the complexity of the runtime necessary for program evaluation.

In this project, we will examine two separate implementations of the Core language. The first implementation directly encodes the idea of graph template instantiation and reduction. Since all of the graph operations occur within Haskell code (a Haskell runtime), this is an interpreter. We improve this with the G-machine by compiling graph construction to a series of opcodes that may

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

Figure 1: Grammar of Λ

run on a simpler stack-based machine, which may be implemented on many computer architectures with a simpler runtime[7].

2.3 The untyped λ-calculus

The untyped λ-calculus Λ is a very basic model of universal computation[8] proposed by Alonzo Church in the 1930's. The grammar of Λ is incredibly simple: it is given by Figure 1. We only have three expression forms in Λ: variables (bound by functions), λ-abstractions (functions of one variable x and return the expression e); and function application $e_1\ e_2$, in which e_1 is in function position and e_2 is in argument position. Note that function application is represented using spaces, and parentheses are only necessary to specify order of operations.

The dynamic semantics are also very simple, illustrated by the rules shown in Figure 2 using a big-step operational semantics. Without diving deep into the syntax, this means that λ-abstractions evaluate to themselves, and the application of e_1 to e_2 involves (recursively) evaluating e_1 to a λ-abstraction $\lambda x.e_1'$; substituting e_2 for every instance of x in e_1'; and then recursively evaluating that result. This is a very simple way to think about function application, and it can be shown that any data or computation can be abstractly represented using this minimalistic representation[9].

The λ-calculus serves as the logical and notational basis for much of functional programming, including Haskell and Core. For example, using space as the function application operator originates from the λ-calculus. Also, lazy evaluation and pure functions mesh very nicely with the more mathematical framework that the λ-calculus provides, as opposed to the statuful model of imperative programming languages.

However, since Λ is so minimalistic, it is not practical nor efficient. Usually we extend Λ with static typing, resulting in the simply-typed λ-calculus (STLC). In our case, we add a base type (integers) and structured data, and extend the grammar with `let` and `case` expressions, but do not implement static typing (which runs us the possibility of run-time type errors).

[7] Chapter 4 of the tutorial, which is not implemented for this project, uses a three-address machine, which may require an even simpler runtime on a three-address architecture such as x86-64.

[8] I.e., Λ can simulate a Turing machine.

[9] See Church notation.

$$\frac{}{\lambda x.e \Downarrow \lambda x.e} \qquad \frac{e_1 \Downarrow \lambda x.e_1' \qquad [e_2/x]e_1' \Downarrow e}{e_1\ e_2 \Downarrow e}$$

Figure 2: Dynamic semantics of Λ

2.4 Functional programming

Functional programming is a programming paradigm highly involved with function application, function composition, and first-class functions. It is a subclass of the declarative programming paradigm, which is concerned with pure expression-based computation. Declarative programming is often considered the complement of imperative programming, which may be characterized as programming with mutable state, side effects, or statements. Purely functional programming is a subset of functional programming that deals solely with mathematically-pure functions; non-pure languages may allow varying degrees of mutable state but typically encourage the use of pure functions. Miranda, Haskell, and Core are examples of non-strict (lazy) pure functional languages.

There are a number of memes[10] among functional languages that may not be familiar to the imperative programmer. Some of these memes include (but are not limited to):

Spaces for function application This is a notational meme from Λ.

Immutability (i.e., lack of state) is an integral part of functional programming that allows for many optimizations such as structural sharing[11] or fast structural equality checking.

No side-effects Pure functional languages additionally disallow side-effects (stateful actions) such as printing or array-based operations, unless specially encapsulated. This may make debugging via printing difficult, so we need a special `Print` opcode in Core.

Curry-by-default functions and partial application In some functional languages, functions of multiple variables may be *partially applied*. For example, consider the following Haskell program.

```
sum x y = x + y
add1 x = sum 1
main = add1 3
```

This may seem to be an error, since `sum` is only applied to one argument in the definition of `add1`. However, this is allowable because `sum` will not be invoked until it is applied to the right number of arguments, which will happen after applying `add1` to 3. If we treat function application

[10] In the original sense of the word, not in the Internet sense of the word.
[11] We implement structural sharing for integer nodes.

as substitution, this may be more clear: `add1 3` may be substituted to `(sum 1) 3` ≡ `sum 1 3`[12]. Alternatively, we may think of any multi-arity function as being a series of nested λ-abstractions of one argument. The following is an equivalent definition of `sum`.

```
sum = \x . \y . x + y
```

Algebraic datatypes (ADTs) In Haskell, you can create type definitions such as the following:

```
data ListNode a = Nil | Cons a ListNode
data TreeNode = Leaf | Node Int TreeNode TreeNode
```

In this example, we see that we can very naturally describe linked lists using a recursive data structure with one of two forms: the base case `Nil`, or a `Cons` cell comprising a value and another (recursive) `ListNode`. This `ListNode` datatype is also parametrically polymorphic with type parameter `a`. We see a similar case for the (non-parametric, recursive) definition of `TreeNode`. ADTs allow us to naturally and safely express composite data and variant types, and language constructs such as pattern-matching and `case` expressions allow for safe deconstruction of structured data. In Core, we will implement a simpler-to-implement system of data constructors than user-defined data constructors as shown in the example above, but it is equally as powerful.

2.5 Terminology

The tutorial assumes a background in (lazy) functional programming. Instead, I assume a background in imperative programming and a basic background in programming language compilers. Below are some terms that may be useful for understanding Haskell, Core, and this project in general. These terms are fairly standard and can be easily looked up on Wikipedia or Stack Overflow, but the following are my own definitions. I thought I'd include them here to avoid confusion, since some of them have somewhat unassuming names.

Lazy evaluation A general term that describes evaluation that only occurs when the expression is used. For the purposes of this paper, it refers to *normal-order* evaluation, and more particularly *call-by-need* evaluation. Note that laziness (and eagerness) can refer to

Eager evaluation The opposite of lazy evaluation: evaluation that occurs when an expression is encountered. The eager analogue to normal-order evaluation is *applicative-order* evaluation.

Normal-order evaluation Lazy function application. In other words, the arguments to a function are not evaluated before the function body is

[12] These are equivalent due to the operator precedence and associativity of the infix function application operator (space).

evaluated. This is usually a feature only of pure functional languages such as Haskell or Core.

Applicative-order evaluation Eager function application. In other words, the arguments to a function are evaluated before the function body is evaluated. This is the way that most languages are implemented, such as C or OCaml.

Strict/non-strict More or less synonyms for eager/lazy. These terms are usually used when referring to Haskell's normal-order evaluation. Haskell allows you to control strictness using special operators[13]. We explore controlling strictness with strict/non-strict compilation contexts in Section 5.4.1.

Call-by-value More or less synonymous with applicative-order evaluation.

Call-by-name A naïve form of normal-order evaluation. The unevaluated expression assigned to the function argument is lazily substituted into each instance of the argument, and evaluated when encountered.

Call-by-need A better form of normal-order evaluation, in which evaluation of arguments are memoized. Haskell and Core are call-by-need. This requires node updates and indirections as described in Section 4.2.5.

Supercombinator The top-level expression construct in Haskell and Core. Roughly equivalent to a function definition. All supercombinators are always in scope, including from the body of the supercombinator or before it is declared, allowing for easy simple and mutual recursion.

λ-abstraction Also known as λ-function, anonymous function, arrow function, function object, or simply function, in a language with first-class functions. In the λ-calculus, these typically take exactly one argument; however, in the case of Core or Haskell supercombinators or anonymous functions, these may take multiple arguments.

Function application Another name for function invocation, usually used in the context of functional programming. Note that functions may be *partially applied* in languages that support *currying-by-default*.

Currying A notational convenience in which a function of n arguments may be *partially-applied* by only calling it with $m < n$ arguments. This partially-applied expression itself is a (curried) function of $n - m$ arguments. When the function is applied to all n arguments, then the function is applied normally. This notation fits nicely into the interpretation of nested one-argument λ-abstractions, and is nicely handled by the TI and GM machines.

[13] `https://wiki.haskell.org/Performance/Strictness#Explicit_strictness`

Currying-by-default A property of a programming language in which functions of multiple arguments may be curried (partially applied). Haskell, OCaml, and Core are examples of such languages. In languages without currying-by-default, currying may be achieved using nested one-argument λ-abstractions.

Product types Composite datatypes, e.g., structs (a.k.a., objects, `records`, or `named tuples`) or tuples.

Sum types Disjunctive datatypes, notably *tagged unions*.

Algebraic datatypes (ADTs) Composite datatypes allow for arbitrary compositions of sum (tagged union) and product (tuple) types. These are a common construct available in many strongly-typed functional languages, such as Haskell, OCaml, and Rust. These are safely deconstructed using `case` expressions and pattern-matching. Data that is formed from ADTs are known as *structured data*.

Weak head normal form (WHNF) An expression is in WHNF if the outermost expression is a data constructor or a λ-abstraction. In other words, it defines what a fully-evaluated expression is in Core. Alternatively, we can think of it as the canonical representation of a datum in Core. (There are also normal and head normal forms, which are not relevant to Core.)

Scrutinee The expression to be matched in a `case` expression. In the case of the Core language, this must be structured data, but it usually is any pattern to match against.

Definiens The expression that a variable is bound to.

Opcode A.k.a. instruction. Simple statement in an imperative assembly-like language.

Runtime For the purposes of this project, usually refers to the run-time evaluator for a programming language implementation. As opposed to "runtime," which is an adjective meaning "occurring at evaluation time."

Precedence	Associativity	Operator
6	left	application
5	right	`*`
5	none	`/`
4	right	`+`
4	none	`-`
3	none	`==`, `~=`, `>`, `>=`, `<`, `<=`
2	right	`&`
1	right	`|`

Table 2: Core infix operator precedence and associativity

3 The Core language

3.1 Syntax

The grammar of Core is given in Figure 3. This is reproduced from Figure 1.1 in the tutorial. λ-abstractions are omitted from the syntax, since they are only introduced in Chapter 6 with the introduction of the λ-lifting program transformation.

The precedence of infix operators is provided in Table 2, reproduced from Table 2 in the tutorial. Note that unary negation is not provided as a primitive operator; instead, the primitive unary function `negate` is provided, and it behaves like a normal function.

Core may be thought of as Λ extended with number types, arithmetic and boolean operators, structured data, `let(rec)` expressions, and `case` expressions. The top-level of a program is simply a sequence of semicolon-delimited supercombinator definitions.

3.2 Dynamic semantics

The dynamic semantics of basic function application in Core is the same as Λ, taking into account currying. The rest of the expression forms should be fairly clear, and are familiar for those acquainted with Haskell or other functional languages with local definitions, algebraic datatypes, and pattern matching.

The expression forms to note are `let(rec)` expressions, `case` expressions, and data constructors using the `Pack{t,a}` expression form. `let` expressions allow for a series of local variable bindings. `letrec` expressions allow for a series of local variable bindings, but all of the variables being defined are in scope of the other variables being defined (e.g., allowing for mutually-recursive definitions)[14]. `Pack` and `case` are used to construct and destruct structured data, respectively, in a simple form without introducing user-defined data con-

[14]Neither `let` nor `letrec` expressions are too important for a language like Core. The same functionality may be implemented with supercombinator definitions (including mutual recursion), but local definitions afford greater convenience.

$$
\begin{array}{rll}
\mathit{program} & ::= & \mathit{sc}_1 \ ; \ \cdots \ ; \ \mathit{sc}_n & \text{(top-level program)} \\
\mathit{sc} & ::= & \mathit{fn} \ \mathit{var}_1 \ \cdots \ \mathit{var}_n = \mathit{expr} & \text{(supercombinator)} \\
\mathit{expr} & ::= & \mathit{expr} \ \mathit{aexpr} & \text{(application)} \\
 & | & \mathit{expr}_1 \ \mathit{binop} \ \mathit{expr}_2 & \text{(infix binary application)} \\
 & | & \texttt{let} \ \mathit{defns} \ \texttt{in} \ \mathit{expr} & \text{(local definition(s))} \\
 & | & \texttt{letrec} \ \mathit{defns} \ \texttt{in} \ \mathit{expr} & \text{(local recursive definition(s))} \\
 & | & \texttt{case} \ \mathit{expr} \ \texttt{of} \ \mathit{alts} & (\texttt{case} \text{ expression}) \\
 & | & \mathit{aexpr} & \text{(atomic expression)} \\
\mathit{aexpr} & ::= & \mathit{var} & \text{(variable)} \\
 & | & \mathit{num} & \text{(number (integer))} \\
 & | & \texttt{Pack}\{\mathit{num}, \mathit{num}\} & \text{(data constructor)} \\
 & | & (\ \mathit{expr}\ ) & \text{(parenthesized expression)} \\
\mathit{defns} & ::= & \mathit{defn}_1 \ ; \ \cdots \ ; \ \mathit{defn}_n & \text{(definitions)} \\
\mathit{defn} & ::= & \mathit{var} = \mathit{expr} & \text{(definition)} \\
\mathit{alts} & ::= & \mathit{alt}_1 \ ; \ \cdots \ ; \ \mathit{alt}_n & (\texttt{case} \text{ alternatives (rules)}) \\
\mathit{alt} & ::= & <\mathit{num}> \ \mathit{var}_1 \ \cdots \ \mathit{var}_n \ -> \mathit{expr} & (\texttt{case} \text{ alternative}) \\
\mathit{binop} & ::= & \mathit{arithop} \mid \mathit{relop} \mid \mathit{boolop} & \text{(binary operators)} \\
\mathit{arithop} & ::= & + \mid - \mid * \mid / & \text{(arithmetic ops)} \\
\mathit{relop} & ::= & < \mid <= \mid == \mid \sim= \mid >= \mid > & \text{(comparison ops)} \\
\mathit{boolop} & ::= & \& \mid \mid & \text{(boolean ops)} \\
\mathit{var} & ::= & \mathit{alpha} \ \mathit{varch}_1 \ \cdots \ \mathit{varch}_n & \text{(variables)} \\
\mathit{alpha} & ::= & \text{an alphabetic character} & \text{(identifiers 1)} \\
\mathit{varch} & ::= & \mathit{alpha} \mid \mathit{digit} \mid \_ & \text{(identifiers 2)} \\
\mathit{num} & ::= & \mathit{digit}_1 \ \cdots \ \mathit{digit}_n & \text{(numbers)}
\end{array}
$$

Figure 3: BNF syntax for the Core language

```haskell
data ListNode = Nil | Cons Int ListNode

length xs = case xs of
  Nil -> 0
  Cons _ xs' -> 1 + length xs'

main =
  let lst = Cons 2 (Cons 3 (Cons 4 Nil))
  in length lst -- returns ``3''
```

(a) Structured data in Haskell

```
-- Nil is represented with Pack{3, 0}
-- Cons(x, y) is represented with Pack{4, 2}
length xs = case xs of
  <3> -> 0 ;
  <4> x xs -> 1 + length xs

main =
  let lst = Pack{4,2} 2 (Pack{4,2} 3 (Pack{4,2} 4 Pack{3,0}))
  in length lst -- returns ``3''
```

(b) Structured data in Core

Figure 4: Structured data usage

structors such as in Haskell. Figure 4 demonstrates the use of both of these forms with an implementation of `length`, a function to compute the length of a linked-list ADT.

3.3 Base definitions

We adopt the the conventions shown in Figure 5. These represent canonical/standard representations of booleans, lists, and the `if` expression using previously-defined expressions[15]. Note that different structured data do not necessarily need to have globally distinct tags; only different variants of the same ADT need be unique. For simplicity, we make all of the ADTs have unique tags, and reserve tags 1 through 4 for these[16].

This forms part of our *prelude* (builtin definitions) for Core, and these definitions may be overridden at any time. Additionally, we provide a sample standard library of useful definitions, such as common combinators, arithmetic aggregation operators, list/stream operations, etc. in the file `examples/gmprelude.core`.

[15] My conventions are slightly different from the textbook's ones, but this is only really a matter of style.

[16] This is a soft restriction.

```
False = Pack{1, 0} ;
True = Pack{2, 0} ;
Nil = Pack{3, 0} ;
Cons = Pack{4, 2} ;
if scrut t f =
  case scrut of
    <1> -> f ;
    <2> -> t
```

Figure 5: Standard representations of structured data and `if` expressions

We call this the standard library or extended prelude. These may be found in Appendix A.1.

3.4 Sample programs

Sample Core code may be found in Appendix A.

3.5 Lexer

Tokenization (lexing) of the Core language is fairly simple. There are almost no reserved keywords, except `Pack`, `let`, and `case`, and the number of expression types is small. Identifiers and numeric tokens are easy to identify using the grammar rules. Other symbols (except whitespace, which is discarded) are counted as one- or two-letter operators. Lines beginning with `--` are treated as comments and discarded[17].

3.6 Parser

The parser described in the tutorial is a fairly simple LL(1) recursive-descent parser, built from scratch. Following the functional way™, this is built up from a set of primitive subparsers in a very composable way. Each (sub)parser is a function that takes a set of tokens as input, and returns a set of transformed matches and remaining tokens. We first define the primitive subparsers, which may recognize forms such as numeric literals, identifiers, or a specified string. Then, we develop two compositional parsers, `pAlt` and `pThen`. `pAlt` takes two subparsers and returns a parser that will match either subparser (and thus may return multiple matches). `pThen` takes two subparsers and matches only if the first subparser matches, and then the second subparser matches on the remaining tokens. By composing these generic subparsers, we build up a parser for Core. The top-level parser parses the whole program and returns the first successful parse. There may be multiple parses due to the *dangling-else problem*.

[17] This is the Haskell comment syntax.

The tutorial goes into significant detail about practical considerations when implementing a parser, such as *left recursion* (to prevent infinite recursion), implementing infix operator precedence and associativity (by splitting the *expr* production into separate productions for each precedence level), and the *dangling-else problem* (which is a parsing ambiguity that arises in *case* expressions). For brevity, I omit a thorough discussion of these topics.

3.7 Pretty-print utility

The tutorial describes a pretty-print utility that is useful for performance and presentation purposes. This is not only useful for printing expressions, but also for any other debugging output, such as printing the program state.

A naïve approach to printing information would be simply performing string concatenation. However, this is a $O(n)$ process on each string concatenation (and thus roughly $O(mn)$ with a large number of concatenations m), and not efficient for very large outputs. Additionally, we would like the ability to easily indent chunks of the formatted output. This is where the `ISeq` data structure comes in[18]. Stringbuilders objects (e.g., Java's `java.lang.StringBuilder`) are able to efficiently concatenate strings, but do not have the ability to provide the indentation layout that we desire.

`ISeq` works by storing the built string as a tree of appended strings. The string concatenation process then becomes the $O(n)$ process of creating an `IAppend seq1 seq2` node with the two subsequences. Additionally, indented blocks may be created using the `IIndent seq` node, which takes a subsequence to indent at the current column number. As a result, building a formatted string is very efficient, since each of these operations is constant time.

The `ISeq` is then *rendered* to a string using the `iDisplay` method. This performs the single string concatenation. This method implements indentation by always keeping track of column count. Rendering the `ISeq` is a $O(n)$ operation as desired.

The `Iseq` data structure is very general and may be used for pretty-printing purposes outside of this project.

[18] I believe the "I" stands for "indentation," but I do not know. I have asked on Stack Overflow but have not received a response.

4 Template instantiation (TI) evaluator

Template instantiation is only briefly reviewed in the tutorial. A more full description is given in Chapters 11 and 12 of "The implementation of functional programming languages" [6]. I will do my best to provide an intuitive high-level overview.

4.1 Intuitive walkthrough of a TI evaluation

The principal idea behind lazy evaluation is that an expression is only evaluated when necessary. We represent an expression using a graph, and evaluate the program by repeatedly reducing the graph until we are left with a data value (a final program state).

For now, let us consider two sample programs, shown below.

```
-- Program A
f x = x + 1
main = f 2

-- Program B
square x = x * x ;
main = square (square 3)
```

We may represent any supercombinator using a graph. For example, the graphs for `square` and `main` from Program B are shown below. The `@` symbol is used to represent function application. Due to automatic function currying, the evaluation of a multi-arity function application `g x y z` actually is implemented as a series of multiple function applications of one variable, i.e., `((g x) y) z`. Infix binary operators appear as functions of two variables in the call graph. Also note that if the same variable is referenced multiple times in the graph, the nodes will point to the same instance – this is important for call-by-need evaluation.

```
-- square
    @
   / \
  @   \
 / \---x
*

-- main
-- Program B state 0
     @
    / \
   /   @
  /   / \
square   3
```

There are two types of reductions. We may either reduce a supercombinator (function) application, or a primitive operation. The reduction of a supercombinator means replacing the graph with the supercombinator and its arguments with the supercombinator's graph. The reduction of a primitive (e.g., arithmetic, relational, or boolean operators) means performing the primitive operation.

Consider Program A for an example of a supercombinator reduction. The graphs for `main` and `f` are shown below.

```
-- f
    @
   / \
  @   1
 / \
+   x

-- main
-- Program A state 0 (initial program state)
  @
 / \
f   2
```

The program begins with the graph of `main`. Clearly, we have to reduce `f` next. We do this by *instantiating* the graph of `f`, and substituting the arguments (in this case, $x \mapsto 2$) in the graph. Thus the next evaluation state would be:

```
-- Program A state 1
    @
   / \
  @   1
 / \
+   2
```

At this point, we see that the next reduction would be the application of `+` to arguments `2` and `1`, i.e., `(+ 2) 1` or `2 + 1`. We use the underlying addition operator in Haskell, and replace the addition operation with the result. At this point, there are no more redexes in the program, so we are done and the program terminates.

```
-- Program A state 2 (final program state)
3
```

The TI always chooses the outermost reducible expression (redex) to reduce; this evaluation order is called *normal-order evaluation*. In other words, if we consider Program B again, we see that there are two redexes that we can reduce from the initial program state: the outer or the inner `square`. If we reduce the inner `square` first, then this is *applicative-order* (strict) evaluation. However,

we wish for *normal-order* (lazy) evaluation, so we reduce the outer expression first.

The way we find the outermost supercombinator or primitive is by walking the left-branch of application nodes; this list of nodes is called the *spine* of the graph. The spine will be stored on the stack; we start from the root, and push the application nodes of the spine onto the stack until we reach a supercombinator or primitive. The spine (and thus the stack) also contains the list of arguments passed to a supercombinator or primitive. With this in mind, the next program state would be:

```
-- Program B state 1
    @
   / \
  @   \
 / \--!@
*     / \
square   3
```

At this point, we would like to reduce the `*`, but there is a problem. The arguments have not been evaluated yet. Thus a primitive operator is only a redex if all of its arguments have been evaluated (to WHNF). This is not the case for this program: the root of our next redex is the node marked `!@`. Note that this isn't on the spine, so we create a new temporary graph with which to evaluate the arguments before returning to the evaluation of the `*` primitive. The node marked `!@` becomes the root of our new graph, and the old graph is saved.

```
-- Program B state 2
      @
     / \
square  3

-- Program B state 3
    @
   / \
  @   \
 / \---3
*

-- Program B state 4
9
```

The multiplication in state 3 is a redex since both arguments are fully evaluated to WHNF. Thus, this reduces down to a number. Now, we are not done, since we still have to finish the initial graph. Thus, we "pop" the old graph from state 1 off of the "stack of old graphs," and replace the unevaluated argument at the node marked `!@` with the evaluated result, 9.

```
-- Program B state 5
    @
   / \
  @   \
 / \---9
*

-- Program B state 6
81
```

At this point, there are no more "old graphs," so evaluation terminates and we have the result.

This section is intended to give a high-level overview of the TI evaluator. The following sections will provide more clarity and be more precise.

4.2 The TI abstract machine

The TI machine is represented using an *abstract state machine*, and evaluation is described using *state transitions*. The state comprises a *stack*, a *heap*, and a *dump*.

The stack holds the spine of the (current) graph. We use this to find the next redex during the unwinding process, and to retrieve the list of arguments for a primitive or supercombinator. This will be discussed in Sections 4.2.1 and 4.2.2. The heap stores all of the (automatically-allocated) nodes used throughout. This will be discussed in Sections 4.2.4 and 4.2.5. The dump is a stack of stacks, and stores the "old graphs" or "old spines" when we need to strictly evaluate an argument, as shown in the Program B example in Section 4.1.

The TI evaluation process alternates between unwinding and instantiation. We will describe these in Sections 4.2.1 and 4.2.2.

4.2.1 Unwinding

Unwinding is the main evaluation semantics of the TI abstract machine. It dispatches an action based on the element on the top of the stack, and thus is easy to model using a state transition machine[19].

It is the process of finding the next redex and choosing which action to perform. In the simplest form, it works by traversing the spine until we find a supercombinator. The state transition rules are very simple. If we encounter a number on the spine and the stack is empty, then evaluation is complete and the program terminates. If we encounter a function application node, we push the application node onto the stack and continue unwinding the left node (continue unwinding the spine). Lastly, if we encounter a supercombinator node, then we instantiate its body using the arguments that are currently on the stack.

[19] We will see that this is not the case of instantiation, which puts it at odds with the state transition formalization. However, the G-machine will compile the instantiation step into a series of small, relatively-atomic opcodes.

This gets more complicated when primitive operators are introduced. If we encounter a primitive operator, we first check if the argument node(s) are evaluated to WHNF. If they are already evaluated, then the primitive operator is a redex and we perform the operation. If an argument is not a redex, then we push the current stack onto the dump, and set the argument as the singleton element of the new stack, and use this to evaluate the argument. Once the argument is fully-evaluated, then we pop the old stack from the dump, and resume evaluation of the primitive operator.

This may remind you of a call stack. In fact, the dump is very similar to using a call stack, and this similarity will be accentuated when we encounter the `Eval` opcode in the G-machine, which performs the analogous operation. Moreover, the dump may be implemented like the call stack by pushing and popping *stack frames*; we do not need a separate stack data structure. The difference is that stack frames on the dump are only created when we need to strictly evaluate an argument, rather than every time a function is invoked as is the case for a call stack; this naturally allows for tail-call optimization (TCO)[20].

4.2.2 Instantiation

Instantiation is the process of creating a supercombinator graph when a supercombinator is applied. The process is fairly intuitive from a high-level perspective (e.g., it is visually clear from the examples in Section 4.1 how instantiation works). We may say that instantiation involves copying the graph structure of a supercombinator, and falling in any instances of its argument variables using the parameters used to invoke the supercombinator.

Instantiation is also fairly easy to implement in native Haskell code. It involves recursing through the Core language expression (represented as an AST), and building the corresponding call graph out of variable, numeric literal, and application nodes in the heap.

As mentioned in an earlier footnote, instantiation is at odds with the state machine representation because it is largely an atomic action and cannot be easily decomposed into simpler steps to run on a simpler runtime. The main difference between the TI and the G-machine implementations is that the instantiation step (the graph creation) is compiled to a series of opcodes in the G-Machine. The G-machine also allows us to discard the AST representation after compilation, since we don't need the instantiation function which operates on Core language expressions.

4.2.3 Advanced instantiations

TODO: let, letrec expressions

TODO: structured data

[20] TCO allows for arbitrary recursion depth if a function is tail-recursive. This is not true of regular call-stack-based languages, since each recursive call will generate a new stack frame and quickly overflow the stack. However, it can be implemented on a call stack using techniques such as trampolines, but this is much messier and less elegant than our natural TI interpretation of graph reduction.

4.2.4 Memory model

The memory model of the TI is very simple. The heap is simply an abstract data structure mapping addresses to nodes. The interface for this data structure has methods to allocate, lookup, and free nodes. Nodes are automatically allocated on demand. Nodes are never freed in my implementation, since I did not implement garbage collection; however, the *mark-scan* and *two-space* garbage collection methods are developed at the end of the chapter. This heap data structure will remain the same for the G-machine implementation.

4.2.5 Updates and indirections for call-by-need evaluation

Updates and indirections are two topics that are necessary to implement efficient call-by-need normal-order evaluation. Without updating nodes after they have been evaluated, repeated usages of a node will result in repeated evaluations, resulting in an inefficient call-by-name normal-order evaluation.

TODO: working here – show sample programs and why these are necessary

4.3 Walkthrough revisited

TODO: revisit the walkthrough example, now using the full knowledge from other sections

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 Sample compilation

We first try to develop an intuition of the compilation process. The example below is given in section 3.1.1 of the tutorial.

```
f g x = K (g x)
```

This compiles down to the following opcode sequence.

```
Push 1
Push 1
Mkap
Pushglobal K
Mkap
Update 3
Pop 3
Unwind
```

These instructions should be indicative of a stack-based machine. We can envision the state of the program after each instruction. Assume that the supercombinator was invoked like so: `f I 2`. Thus we expected `I` and `2` to be already pushed onto the stack.

Note that this uses the updated addressing mode described in Section 5.3. Thus the stack does not initially contain the application spine, but rather only the root of the supercombinator application and the arguments, pushed in reverse order.

We omit the dump from this evaluation trace; it is not relevant here, since there are no primitive operators.

```
-- State 0: initial program state (upon applying f I 2)

Instruction queue: Push 1; Push 1; Mkap; Pushglobal K; Mkap; Update 3; Pop 3; Unwind
```

```
Stack:
0. #0
1. #2
2. #4

Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2
#3 NSupercomb f
#4 NSupercomb I
#5 NSupercomb K
#6 NSupercomb main
```

Node 0 represents the root of the supercombinator application, which is represented by the following graph:

```
Graph(s):
         #0(@)
        /     \
     #3(@)    #2(2)
    /   \
#3(f)   #4(I)
```

We will now walk through the instruction sequence. The effect of each instruction should be relatively straightforward, since they are relatively simple stack instructions.

```
-- State 1: after (first) Push 1
Instruction queue: Push 1; Mkap; Pushglobal K; Mkap; Update 3; Pop 3; Unwind

Stack:
0. #0
1. #2
2. #4
3. #2

Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2
#3 NSupercomb f
#4 NSupercomb I
#5 NSupercomb K
#6 NSupercomb main
```

```
Graph(s):
         #0(@)
        /     \
     #3(@)    #2(2)
    /   \
#3(f)   #4(I)

-- State 2: after (second) Push 1
Instruction queue: Mkap; Pushglobal K; Mkap; Update 3; Pop 3; Unwind

Stack:
0. #0
1. #2
2. #4
3. #2
4. #4

Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2
#3 NSupercomb f
#4 NSupercomb I
#5 NSupercomb K
#6 NSupercomb main

Graph(s):
         #0(@)
        /     \
     #3(@)    #2(2)
    /   \
#3(f)   #4(I)

-- State 3: after (first) Mkap
Instruction queue: Pushglobal K; Mkap; Update 3; Pop 3; Unwind

Stack:
0. #0
1. #2
2. #4
3. #7

Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2
```

```
#3 NSupercomb f
#4 NSupercomb I
#5 NSupercomb K
#6 NSupercomb main
#7 NAp #4 #2

Graph(s):
         #0(@)
        /     \
     #3(@)    #2(2)
    /   \
#3(f)   #4(I)

   #7(@)
   /   \
#4(I)  #2(2)

-- State 4: after Pushglobal K
Instruction queue: Mkap; Update 3; Pop 3; Unwind

Stack:
0. #0
1. #2
2. #4
3. #7
4. #5

Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2
#3 NSupercomb f
#4 NSupercomb I
#5 NSupercomb K
#6 NSupercomb main
#7 NAp #4 #2

Graph(s):
         #0(@)
        /     \
     #3(@)    #2(2)
    /   \
#3(f)   #4(I)

   #7(@)
   /   \
```

```
#4(I)  #2(2)

-- State 5: after (second) Mkap

Instruction queue: Update 3; Pop 3; Unwind

Stack:
0. #0
1. #2
2. #4
3. #8

Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2
#3 NSupercomb f
#4 NSupercomb I
#5 NSupercomb K
#6 NSupercomb main
#7 NAp #4 #2
#8 NAp #5 #7

Graph(s):
         #0(@)
        /     \
     #3(@)    #2(2)
    /   \
#3(f)   #4(I)

   #8(@)
   /   \
#5(K)  #7(@)
       /   \
    #4(I)  #2(2)

-- State 6: after Update 3

Instruction queue: Pop 3; Unwind

Stack:
0. #8
1. #2
2. #4
3. #8
```

```
Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2
#3 NSupercomb f
#4 NSupercomb I
#5 NSupercomb K
#6 NSupercomb main
#7 NAp #4 #2
#8 NAp #5 #7

Graph(s):
         #0(@)
        /     \
     #3(@)    #2(2)
    /   \
#3(f)   #4(I)

   #8(@)
   /   \
#5(K)  #7(@)
       /   \
    #4(I)  #2(2)

-- State 7: after Pop 3
Instruction queue: Unwind

Stack:
0. #8

Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2
#3 NSupercomb f
#4 NSupercomb I
#5 NSupercomb K
#6 NSupercomb main
#7 NAp #4 #2
#8 NAp #5 #7

Graph(s):
   #8(@)
   /   \
#5(K)  #7(@)
       /   \
```

```
#4(I)  #2(2)
```

As we can see, the graph constructed by the compiled opcodes produces the result as instantiation.

The last instruction is the most complicated. This is the `Unwind` opcode, which corresponds to the unwinding action discussed in the TI evaluator (setting up the instructions for the next instantiation). Up until now, all of the instructions have been used to create the graph that would have been created through instantiation. After the `Unwind` opcode, we expect to have a setup similar to state 0 for evaluation to continue again.

5.2 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[21]. A precise definition of most of the opcodes are given in the tutorial[22].

`Pushglobal` f Push the address of the global (supercombinator) f onto the stack[23].

`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 [24].

`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.

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

[22] 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.

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

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

`Unwind` Similar in function to unwinding in the TI implementation Section 4.2.1. Finds the next redex along the spine. Also performs the stack restructuring described in Section 5.3. When the supercombinator or primitive is found, adds the compiled opcodes to the instruction queue.

`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)[25].

`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* The set *rules* is of the form $\{t_j \rightarrow [i_1, i_2, \ldots]\}$: a mapping of tags t_j to code sequences $[i_1, i_2, \ldots]$. 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.

5.3 A simpler stack addressing mode

In the template instantiation, the spine was pushed onto the stack. This means that for a supercombinator with n arguments, the spine has n `NAp` nodes, followed by the supercombinator node at the top of the stack. In order to get the m-th argument, we traverse up $(m + 1)$ nodes, and have to perform an extra lookup on the `NAp` to get the address of the argument.

[25]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.

Mark 3 of the G-machine proposes a simpler stack addressing scheme. To prevent extra indirections, we restructure the stack during an `Unwind` instruction so that only the arguments for a supercombinator are left on the stack, rather than the `NAp` nodes. The exception is that the root of the supercombinator application redex is left on the stack, so that it may be updated with the evaluation result of the redex. This method extends naturally to `let(rec)` expressions.

This aligns with the x86 ABI, which involves pushing arguments onto the stack in reverse order. Of course, there is no notion of application nodes in a low-level language. The only difference is the root of the redex remaining under the arguments on the stack.

5.4 Compilation schemes

Compilation is divided into several `compilation schemes`. A high-level overview of the compilation schemes is shown below[26]. A list of rules for these compilation schemes may be found in [3].

$\mathcal{SC}$ **compilation scheme** Compile a supercombinator. Wrapper around $\mathcal{R}$

$\mathcal{R}$ **compilation scheme** Strictly compile a supercombinator body using the $\mathcal{E}$ compilation scheme, and then unwind the stack.

$\mathcal{C}$ **compilation scheme** Compile an expression in a lazy compilation context.

$\mathcal{E}$ **compilation scheme** Compile an expression in a strict compilation context.

$\mathcal{D}$ **compilation scheme** Compile a `case` expression.

$\mathcal{A}$ **compilation scheme** Compile an alternative in a `case` expression.

For sake brevity, I will not describe all of the compilation schemes in detail. Instead, I will only discuss the compilation schemes for the Mark 1 G-machine, which should give a good idea of how the compilation schemes work. These are shown in Figure 6, which reproduces Figure 3.3 from the tutorial.

The rules for $\mathcal{SC}$ and $\mathcal{R}$ are simple; these are drivers for code generation for top-level supercombinators. The $\mathcal{C}$ compilation scheme is the most interesting. We see how functions, variables, and integers are compiled. ρ is an environment mapping variable names to offsets on the stack [27]. Global variables and numbers that appear in the expression are simply pushed onto the stack using the `Pushglobal` and `Pushint` opcodes. Local variables are looked up in the environment, and a `Push` opcode is generated. Lastly, function application is generated postfix-style. This compiles the argument, then the function. After these two are compiled, a `Mkap` instruction is generated to create the application

[26] N.B. I have no idea why these evaluation schemes are named the way they are. There is no indication in the textbook, or anywhere online, as far as I can tell.

[27] This is similar to `%rbp`-relative addressing for local variables.

$$
\begin{aligned}
\mathcal{SC}[\![f\ x_1\ \cdots\ x_n = e]\!] &= \mathcal{R}[\![e]\!]\ [x_1 \mapsto 0, \cdots, x_n \mapsto n-1]\ n \\
\mathcal{R}[\![e]\!]\ \rho\ d &= \mathcal{C}[\![e]\!]\ \rho\ {+\!\!+}\ [\texttt{Slide}\ d+1, \texttt{Unwind}] \\
\mathcal{C}[\![f]\!]\ \rho &= [\texttt{Pushglobal}\ f] \\
\mathcal{C}[\![x]\!]\ \rho &= [\texttt{Push}\ (\rho x)] \\
\mathcal{C}[\![i]\!]\ \rho &= [\texttt{Pushint}\ i] \\
\mathcal{C}[\![e_0\ e_1]\!]\ \rho &= \mathcal{C}[\![e_1]\!]\ \rho\ {+\!\!+}\ \mathcal{C}[\![e_0]\!]\ \rho^{+1}\ {+\!\!+}\ [\texttt{Mkap}]
\end{aligned}
$$

Figure 6: G-machine Mark 1 compilation schemes

node from the top two elements on the stack. This covers the basic operations for the λ-calculus logic.

A full description of the full compilation schemes for the Mark 6 G-machine would be too long to include here. Mark 3 implements `let(rec)` expressions, which are straightforward but tedious. Mark 4 implements arithmetic (which introduces the `Eval` opcode to strictly evaluate an element on the top of the stack by using the dump). Mark 5 implements the strict $\mathcal{E}$ compilation scheme described in Section 5.4.1. Mark 6 implements structured data, which is fairly complicated and has some nuances regarding strict compilation and unsaturated constructors that require program transformations or other tricks[28].

5.4.1 Non-strict vs. strict compilation context

Mark 4 of the G-machine introduces primitives into the language, and is the first iteration of the G-machine in which strictness becomes relevant. The ordinary compilation of an primitive operation such as addition produces the following opcode sequence.

```
Push 1
Eval
Push 1
Eval
Add
Update 2
Pop 2
Unwind
```

However, this seems like an awful lot of work to perform a simple addition operation. The book motivates this with the simple program:

```
main = 3+4*5
```

[28] See section 3.8.7 or exercise 3.38 in the tutorial.

Ordinarily, this program will compile to the following program. In addition to this opcode sequence, each of the function applications will spawn the eight opcodes shown above.

```
Pushint 5
Pushint 4
Pushglobal "*"
Mkap
Mkap
Pushint 3
Pushglobal "+"
Mkap
Mkap
Eval
```

However, we can compile this to the following set of opcodes by inspection, which is much shorter, and doesn't include any function applications or `Eval` opcodes:

```
Pushint 5
Pushint 4
Mul
Pushint 3
Add
```

It should not be surprising that strict code may be more efficient than non-strict code. This gives rise to the $\mathcal{E}$ evaluation scheme described in Section 5.4. However, we cannot always generate strict code, or else defeat the purpose of a lazy language. In particular, we cannot generate strict code for function arguments or definiens in a `let(rec)` expression, since these may never be evaluated and evaluating them strictly would make evaluation not normal-order.

The intuition behind the $\mathcal{E}$ compilation scheme is that any expression that lies directly in a supercombinator body (i.e., not in a `let(rec) definiens or in a function argument`) must be evaluated when the supercombinator is evoked. We compiled such expressions in the `strict evaluation context` $\mathcal{E}$. Other expressions are compiled in the `lazy evaluation context` $\mathcal{C}$. Additionally, we do not need `Eval` opcodes when in the strict compilation context, since we can recursively deduce that all subexpressions are strictly evaluated.

Strict and lazy evaluation is purely for improvements in performance, and do not (should not) change the behavior of the program. The current mechanism to determine when a strict evaluation context is acceptable is very simple; the tutorial cites other research on the material such as [1]. Haskell also allows users to explicitly specify strict evaluation of an expression.

5.5 Evaluator

The evaluator implements the abstract stack machine. Each of the opcodes described in Section 5.2 is implemented as a separate function in the G-Machines's

`Evaluator` module.

5.5.1 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.

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[29], with optimized GHC-compiled programs usually only 2 to 5 times slower than gcc-compiled programs[30].

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 [4].

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 5.5.1. 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.

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

[30] `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] Geoffrey Burn. *Lazy functional languages: abstract interpretation and compilation*. MIT Press, 1991.

[2] 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.

[3] Thomas Johnsson. "Efficient compilation of lazy evaluation". In: *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*. 1984, pp. 58–69.

[4] 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.

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

[6] Simon L Peyton Jones. *The implementation of functional programming languages*. Prentice-Hall, Inc., 1987.

A Code samples

A.1 Core standard library

These are a set of standard definitions that I found useful for exercises. They are not part of the "prelude," i.e., the list of standard definitions builtin to the language as described in Section 3.3, so these definitions need to be loaded from a file.

```
-- Logical operators
and x y = if x y False ;
or x y = if x True y ;
xor x y = if x (not y) y ;
not x = if x False True ;

-- Useful simple combinators
id x = x ;
double x = x + x ;

-- Useful simple arithmetic operators and predicates
inc x = x + 1 ;
zero x = x == 0 ;
even x = x == ((x / 2) * 2) ;
odd = compose not even ;
mod m n = m - (m / n) * n ;

add acc val = acc + val ;
mul acc val = acc * val ;

-- Sample math functions
fac n = if (zero n) 1 (n * fac (n - 1)) ;
fib n = if (n < 2) n (fib (n - 1) + fib (n - 2)) ;

-- List operations, LISP-style
car l = case l of
  <4> x y -> x ;
cdr l = case l of
  <4> x y -> y ;
length l = case l of
  <3> -> 0 ;
  <4> x y -> 1 + length y ;
nth n l = case l of
  <3> -> 0 ;
  <4> x xs -> if (zero n) x (nth (n - 1) xs) ;

caar = compose car car ;
cadr = compose car cdr ;
```

```
cdar = compose cdr car ;
cddr = compose cdr cdr ;
caadr = compose car cadr ;
caaar = compose car caar ;
cdadr = compose cdr cadr ;
cdaar = compose cdr caar ;
caddr = compose car cddr ;
cadar = compose car cdar ;
cdddr = compose cdr cddr ;
cddar = compose cdr cdar ;

-- Sequences/streams
numsFrom x = Cons x (numsFrom (x + 1)) ;
nats = numsFrom 0 ;
ones = Cons 1 ones ;

-- List operations
map f lst = case lst of
  <3> -> Nil ;
  <4> x xs -> Cons (f x) (map f xs) ;
filter p lst =
  case lst of
  <3> -> Nil ;
  <4> x xs ->
    let rest = filter p xs in
    if (p x) (Cons x rest) rest ;
foldl f acc lst = case lst of
  <3> -> acc ;
  <4> x xs -> foldl f (f acc x) xs ;
take n lst = if (n <= 0) Nil (take2 n lst) ;
take2 n lst = case lst of
  <3> -> Nil ;
  <4> x xs -> Cons x (take (n - 1) xs) ;
takeWhile p lst = case lst of
  <3> -> Nil ;
  <4> x xs -> if (p x) (Cons x (takeWhile p xs)) Nil ;
drop n lst = if (n <= 0) lst (drop2 n lst) ;
drop2 n lst = case lst of
  <3> -> Nil ;
  <4> x xs -> drop (n - 1) xs ;

-- Aggregate operations
sum = foldl add 0 ;
prod = foldl mul 1 ;

-- Useful function to generate a range
```

```
range a b = map (add a) (take ((b - a) + 1) nats)
```

A.2 Project Euler 1

See Project Euler 1. This requires the standard library from Appendix A.1.

```
N = 1000 ;

ltN x = x < N ;

sumMultsOfMUnderN m =
  let multsOfM       = map (mul m) nats in
  let multsOfMUnderN = takeWhile ltN multsOfM in
  sum multsOfMUnderN ;

main =
  let f = sumMultsOfMUnderN
  in f 3 + f 5 - f 15
```

TODO: additional examples, such as lazy infinite streams, twice twice twice