6CCS3PRJ Final Year

Exploring LLVM IR as a Target for Functional Language Compilation

Final Project Report

Author: Talha Abdulkuddus

Supervisor: Dr. Christian Urban

Student ID: 20076327

4th April 2024

Abstract

Functional programming languages are characterised by their focus on functions as first-class objects, with programs being constructed by composing functions together. Typical targets for compilers, such as the LLVM, are designed with imperative languages in mind, and as such, require frontends to convert from a functional paradigm to an imperative one.

This project aims to explore the effectiveness of implementing functional language features that target the LLVM, and how well the LLVM toolchain can optimise and compile these features to machine code.

Results show that, after applying the LLVM optimiser to the generated IR, the performance of the compiled code is comparable to that of other modern compilers such as Clang. The LLVM toolchain provides a powerful backend for functional language compilers, with the ability to optimise the generated IR for a variety of target architectures.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Talha Abdulkuddus

4th April 2024

Acknowledgements

I would like to thank my supervisor, Dr. Christian Urban, for his guidance and support throughout the project. His module on compilers sparked an interest in a topic that I had previously found daunting. I would also like to thank my friends, Saif Latifi and Yusuf Yacoobali, for listening to my ramblings when they probably had more important things to do. Finally, I would like to thank my family for their support and encouragement throughout my studies.

Contents

Chapter 1

Introduction

Functional languages are a class of programming languages that stem from the lambda calculus, treating computation as the evaluation of functions, avoiding side effects and mutable state. Benefits of this paradigm include the ability to apply formal mathematical reasoning about programs, while encouraging modularity and code reuse. Functional languages are often easier to reason about, as the lack of side effects makes programs more predictable and easier to test. This predictability also makes functional languages more amenable to automatic parallelisation, as the lack of side effects means that functions can be executed in any order without affecting the result.

Advantages of this paradigm have even led to the adoption of some functional language features in other popular languages, including Java, C#, and Rust. During the development of this project, new languages have been released that take heavy inspiration from the functional paradigm, such as the Erlang runtime-based language Gleam.

To facilitate programming in this paradigm, modern functional languages typically provide a number of features, such as higher-order functions, pattern matching, and algebraic data types [1]. Typical imperative constructs such as loops and mutable variables are also discouraged, instead favouring recursion and immutable data structures. These characteristics make compilers for functional languages an interesting case study, as more work is offloaded to the compiler for transforming programs into efficient imperative machine code – code that is in a different paradigm to that of a functional language.

One method of compiling functional languages is via the LLVM toolchain, which provides a collection of utilities for compiling, optimising, and linking programs. For language designers, one can create a frontend that compiles a language into the LLVM Intermediate Representation

(IR), which can then be compiled into machine code for a variety of architectures by leveraging the LLVM backend. This approach can be preferable in comparison to:

- **Compiling to native machine code:** While theoretically the most efficient method of compiling a program, this approach requires the language designer to implement a backend for each target architecture. This can be time-consuming and error-prone, as the language designer must be familiar with the intricacies of each architecture.
- **Compiling to a virtual machine (VM):** This approach involves compiling the program to a VM runtime, which can then be executed on any machine that has the VM installed. While this approach is portable, it can be less performant than compiling to native machine code, as the program must be interpreted by the VM at runtime. Additionally, the language designer must either implement a VM runtime for each target architecture (providing similar challenges to compiling to native machine code), or rely on an existing VM runtime (which may not consider the intricacies of a functional language when executing the program).

The LLVM IR also provides a number of optimisations that can be applied to the program before it is compiled to machine code, such as dead code elimination, loop unrolling, and inlining. These optimisations can be applied to the program without the need for the language designer to implement them, and can be tailored to the target architecture.

1.1 Aims

This project aims to implement a compiler for a small functional language, targeting the LLVM IR. This language (inspired by the language presented in the KCL module 6CCS3CFL) will support a number of features, including higher order functions and pattern matching. The project seeks to evaluate the effectiveness of the LLVM toolchain as a backend for a functional language compiler, and to explore the challenges of compiling functional languages to imperative machine code.

1.2 Report Structure

Contemporary compiler design will be explored within Chapter 2, with a focus on translation between functional languages and the LLVM IR. As functional languages rely on the compiler to perform more work than imperative languages, analysing how modern compilers are designed

to facilitate this is of significance. The full scope of the compiler and language will be defined in Chapter 3, with the design of the compiler architecture presented in Chapter 4. The implementation of the compiler, including how specific language features are translated to the LLVM IR, will be discussed in Chapter 5. Evaluation of the compiler, including performance in comparison to other modern compilers, will be presented in Chapter 6. An analysis of legal, social, ethical and professional concerns related to this project is discussed in Chapter 7, with conclusions drawn in Chapter 8.

Chapter 2

Background

2.1 Functional Programming

Functional programming languages are characterised by the use of first-class functions, higher-order functions, and the use of recursion instead of loops. These languages are also typically *referentially transparent*, meaning that the same function call with the same arguments will always return the same result, regardless of the context in which it is called. This category of languages can be seen as the application of mathematical functions to programming, with the language's syntax and semantics being inspired by lambda calculus.

2.1.1 Lambda Calculus

Lambda calculus is a computational model based on the abstraction and application of functions. The model was introduced by Alonzo Church in the 1930s as a way to formalise the notion of computability [2], and has since been used as a foundation for the design of functional programming languages. The understanding of procedural languages could also be seen in terms of lambda calculus [3]. The core of lambda calculus consists of three elements:

- **Variables**: Representations of values that can be passed to functions.
- **Abstraction**: The process of defining a function by specifying its arguments and body.
- **Application**: The process of applying a function to an argument.

For instance, the lambda calculus expression $\lambda x.x + 1$ represents a function that takes an argument x and returns $x + 1$. By nesting lambda abstractions, more complex functions can be defined. For example, the expression $\lambda x.\lambda y.x + y$ represents a function that takes two arguments and returns their sum.

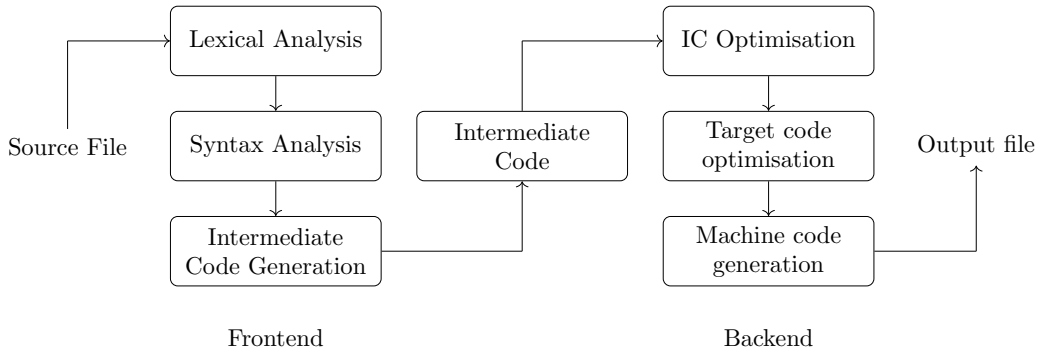

Figure 2.1: Simplified view of the phases of a compiler (adapted from [4]).

The lambda calculus has the notion of *free* and *bound* variables, where a variable is bound if it is within the scope of a lambda abstraction, and free otherwise. A function that contains no free variables are known to be *closed*.

Expressions that are open cannot be evaluated, as they are not fully defined. For instance, the expression $(\lambda y.x + y)$ is not closed, as the variable x is free. However, if we define x as a constant, or supplied the value of x in it's enclosing environment, the expression would be closed and therefore evaluable. A similar concept is seen in programming languages, where functions that reference variables outside their scope are known as *closures*. These functions are able to capture the environment in which they were defined and can be passed around as first-class values.

2.2 The Compiler Pipeline

Compilers play the role of converting a program written in a given programming language into a corresponding program in a defined target language. While this conversion can theoretically be done in one step, it is common for compilers to employ multiple phases responsible for each major transformation of the source code [4].

As seen in Figure 2.1, these phases are split into a frontend responsible for converting the source language into some intermediate representation, and a backend responsible for converting the given intermediate representation into executable code for the target platform. This modular approach of dividing the compilation process into phases provides the ability to reuse certain phases when targeting different platforms, as well as to allow for semantically correct program optimisations within appropriate phases.

As an example, the Glasgow Haskell Compiler (GHC) makes use of ten mandatory compiler

phases [5]. However, most phases within the GHC are intentionally simple transformations, leaving the majority of the optimisations to a specific single phase between the frontend and code generation. Additionally, the GHC is able to select multiple backend targets, including a C subset, LLVM, and native code generation [6].

2.2.1 Lexical Analysis

During the lexical analysis phase, the source input is converted into a stream of tokens by identifying meaningful character sequences that can be grouped together. These character sequences, known as *lexemes*, are associated with a token type indicating whether it is an operator, identifier, keyword or another feature of the source language. This phase is also responsible for filtering characters inconsequential to the semantics of a program, such as whitespace and comments.

For instance, the following:

```
// A simple multiplication
val x = 4 * 2
```

could be converted into the following stream of tokens:

```
KEYWORD(val) IDENTIFIER(x) EQUALS INTEGER(4) OPERATOR(*) INTEGER(2)
```

where `KEYWORD` represents a keyword token, `EQUALS` represents an equals token, and so on.

A natural mechanism for identifying tokens from a stream of characters is to use regular expressions [7]. For instance, the regular expression `[a-zA-Z]+` could be used to identify identifiers, while `[0-9]+` could be used to identify integers.

2.2.2 Syntax Analysis

The syntax analysis phase, also known as the parsing phase, is responsible for converting the stream of tokens produced by the lexical analysis phase into a parse tree. This tree represents the syntactic structure of the lexed tokens, which are derived from a context-free grammar (CFG). The parse tree is then converted into an abstract syntax tree (AST) by removing nodes that do not contribute to the semantics of the program, such as those representing parentheses.

While there are a plethora of algorithms and techniques for parsing, the most common method used for parsing contemporary programming languages as surveyed in 2021 is via a

handwritten *recursive descent parser* [8]. This parsing method is based on the idea of a top-down parser, where the parser starts at the root of the syntax tree and recursively works its way down in a depth-first, pre-order manner to the leaves of the tree.

Another top-down parsing method are *parser combinators*, which are a form of higher-order functions that can be used to construct and combine parsers. These combinators provide greater modularity than recursive descent parsers, as well as the ability to construct parsers that are more expressive than those that can be constructed by recursive descent.

2.2.3 Intermediate Code Generation

The intermediate code generation phase is responsible for converting the AST produced by the syntax analysis phase into an intermediate representation (IR) that is closer to the target language. This IR is typically a low-level representation that is easier to optimise and convert into machine code than the original source code. The IR is also used to decouple the frontend from the backend of the compiler, allowing for platform-generic optimisations to be implemented, as well as facilitating the reuse of the frontend when targeting different platforms.

2.3 LLVM IR

The LLVM project is an open-source collection of tools and libraries for the construction of compilers and related programming tools, the core of which revolves around the LLVM intermediate representation (LLVM IR). This IR is low-level enough to be used as a target for compilers from which an LLVM backend can generate machine code, while also being high-level enough to be used as a portable assembly language targeting a variety of architectures. While

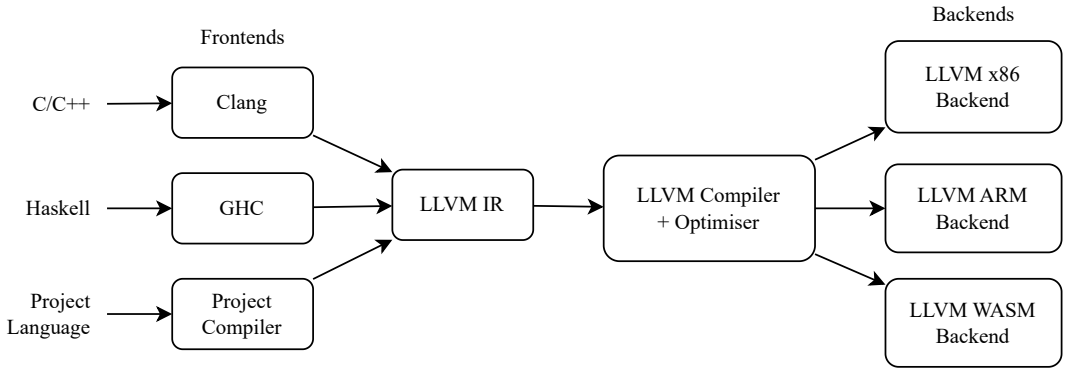

Figure 2.2: Integrating the LLVM into a compiler pipeline.

originally designed as a target for compiling C and C++ programs, the LLVM has since been used as a backend for a variety of languages, including Rust, Swift, and Haskell.

As seen in Figure 2.2, targeting the LLVM IR in a compiler pipeline allows for the compiler to take advantage of the LLVM optimiser, which is capable of performing a variety of optimisations such as constant propagation, dead code elimination, and loop optimisations. No extra work is required to target different architectures, as the LLVM backend is capable of generating machine code for a variety of platforms, including x86, ARM, and even the web via WebAssembly.

From its inception, the IR was designed to be represented in Static Single Assignment (SSA) form [9], restricting the IR such that each variable must be assigned to exactly once before usage, and cannot be reassigned. This restriction alone would imply that purely functional languages could be represented in SSA form, as these languages enforce immutability. However, the SSA used in LLVM IR also enforces that expressions be atomic, with functions being strictly applied to only variables or constants, akin to assembly instructions. The effect of this is that the evaluation order of expressions and program flow is made explicit, simplifying optimisations such as dead code elimination and constant propagation.

For instance, the following Scala code:

```
def foo(x: Int, y: Int): Int = x + y * x
```

could be converted into the following LLVM IR:

```
define i32 @foo(i32 %x, i32 %y) {
    %mul = mul i32 %y, %x
    %add = add i32 %x, %mul
    ret i32 %add
}
```

Observe that the arithmetic order of operations is made explicit in the LLVM IR, with the multiplication of `%y` and `%x` being computed before the addition of `%x` and `%mul`. This explicitness allows for the compiler to reorder instructions and perform optimisations that would not be possible in the original source code.

Chapter 3

Requirements

The aim of this project is to implement a compiler for a small functional language, targeting the LLVM IR. The language syntax is largely inspired by the language presented in the KCL module 6CCS3CFL (which in turn was inspired by Scala), with additional features.

The functional language should be able to express and support the following concepts (with examples):

- **Arithmetic** and **Boolean** expressions

  (Operator precedence and associativity must be respected.)

  ```
  1 + 2 * 3 < 4 && 5 >= 6
  ```

- **Variable Bindings**

  ```
  val x = 1; val y = 2; val z = x + y
  ```

- **If** expressions with optional else clauses

  (Care must be taken to avoid the dangling else problem.)

  ```
  if (x < 0) 100 else if (x > 0) -100 else 50
  ```

- **Printing** to the console

  ```
  print(9 * (5 + 3))
  ```

- **Functions** and **Function Application**

```
def foo(x: Int, y: Int): Int = x + y;
foo(1, 2)
```

- **Higher-Order Functions** (i.e. functions that take other functions as arguments)

```
def apply(f: Int => Int, x: Int): Int = f(x);
def increment(x: Int): Int = x + 1;
apply(increment, 1)
```

- **Closures** (i.e. functions that capture variables from their surrounding scope)

```
def makeIncrementer(x: Int): Int => Int = {
    def incrementer(y) = x + y;
    incrementer
}
val incrementBy5 = makeIncrementer(5);
incrementBy5(10)
```

- Basic atomic **Types** (e.g. `Int`, `Bool`, `Float`, ...)

```
val x: Int = 1; val y: Bool = true
```

- User-defined **Structure Types**

```
type Point = { x: Int, y: Int };
val p: Point = { x = 1, y = 2 }
```

- A basic form of **Pattern Matching** on **Enumerated Types**

```
enum Priority = Low | Medium | High;

def foo(p: Priority): Int = p match {
    case Low => 1
    case Medium => 2
    case High => 3
}
```

The compiler should be able to take a program written in this language and produce the semantically equivalent LLVM IR.

The emitted LLVM IR should be compiled and executed using the LLVM toolchain. If possible, this should result in the final compiler being able to compile to multiple target architectures, such as x86 and ARM. Access to the LLVM toolchain should also provide the

ability to apply various optimisations to the generated LLVM IR without the need to implement them manually.

Chapter 4

Specification & Design

Mirroring the convention for modern compilers, the compiler will be implemented with distinct phases for each major transformation step outlined in the requirements. This clear separation of concerns allows for each phase to be developed and tested independently, and for the compiler to be easily extended or modified in the future. The phases are as follows:

1. **Parsing**: Responsible for transforming the source code into an Abstract Syntax Tree (AST).
2. **IR Generation**: Transforms the AST into an intermediate representation (IR) based on Administrative Normal Form (ANF).
3. **Closure Conversion**: Converts the IR into a form that supports higher-order functions by introducing closures.
4. **Hoisting**: Hoists nested functions and other declarations to the top-level global scope.
5. **LLVM Generation**: Generates the LLVM IR from the ANF IR, and any additional runtime functions required.

This is illustrated in Figure 4.1.

4.1 Technologies

The language selected for this compiler is *Scala*, a JVM-based functional programming language. This choice of language was primarily motivated by the concise and expressive syntax Scala provides, such as operator overloading and extensive support for pattern matching. However,

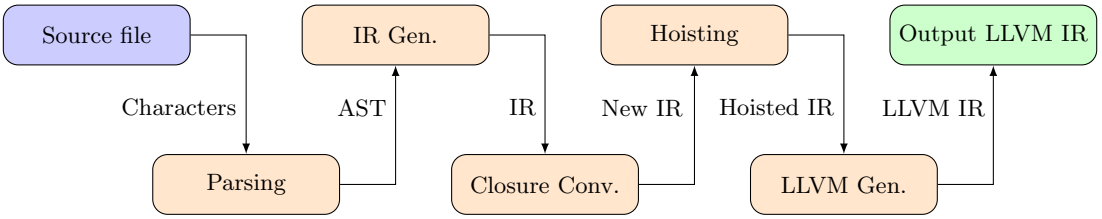

Figure 4.1: A diagram of each phase of the proposed compiler.

the choice of language is not a major factor in dictating the design of the compiler, as many of the implementation and design decisions could easily be ported to other languages.

The *ScalaTest* library will be used for testing, ensuring that the compiler is semantically correct, as well as for identifying any regressions in the compiler's behaviour.

The LLVM static compiler `llc` will be used to generate the object file required for linking the final executable with `gcc`. Using `llc` provides access to the same compiler optimisations as seen in `clang` (i.e., `-O1, -O2, -O3`) and other compilers that utilise the LLVM optimiser, negating the need for implementing certain optimisations within the compiler itself.

4.2 Compiler Phases

4.2.1 Parsing

This phase is responsible for transforming the stream of characters from a source file into an AST. The implementation will be carried out using *FastParse*, a parser combinator library for Scala. This decision was motivated by the library's parsing performance in comparison to other parser libraries, extensive tooling for debugging and testing, in addition to its judicious use of Scala's language features. This results in a syntax for defining parsers that is modular and simple to reason about. An advantage of using a parser combinator library is that it allows for the implementation to closely resemble the grammar of the language being parsed, making it easier to maintain and extend the parser.

The trade-off when using *FastParse* is that it operates directly with the stream of characters, rather than a stream of tokens (known as *scannerless parsing*). Therefore, the lexical analysis phase will be omitted, with the parser responsible for identifying tokens.

The context-free grammar used by the parser must be unambiguous, and (particularly for arithmetic and boolean expressions) correctly reflect the precedence and associativity of operators. While technically possible for parser combinators to handle ambiguous grammars

[10], the *FastParse* library does not.

4.2.2 IR Generation

As the LLVM IR is specified to be in SSA form, compilers targeting the LLVM IR must convert their programs into SSA form before generating LLVM IR. For functional languages, it is useful to convert to another intermediate representation that is close to SSA form before converting to LLVM IR.

Two such intermediate representations commonly used for compiling functional languages are:

- **Continuation-Passing Style**
- **A-Normal Form**

These specific representations were chosen as they both have been demonstrated to map closely to lambda calculus [11, 12], making them ideal for representing functional languages.

Continuation-Passing Style

Continuation-Passing Style (CPS) refers to a programming style where, instead of function invocations returning a value, functions instead invoke a continuation function with the value as an argument.

For instance, the following Scala expression:

```
foo(1, 2) + bar(3, 4)
```

could be converted into the following CPS:

```
// foo and bar accept and return additional continuation function arg
foo(1, 2,
    (a) => bar(3, 4,
        (b) => a + b
    )
)
```

where `k` represents the continuation.

While not identical to SSA, it has been proven that a subset of CPS is equivalent to SSA, and this subset is sufficient for use as an intermediate representation [13].

A-Normal Form

A simpler, more compact IR is Administrative Normal Form (ANF), referring to a programming style where all expressions are either variables or function calls with atomic arguments.

For instance, the same Scala code from Section 4.2.2 could be converted into the following ANF code:

```
val a = foo(1, 2)
val b = bar(3, 4)
a + b
```

Both of these IR forms have historically been implemented in the Glasgow Haskell Compiler's (GHC). The initial implementation of an intermediate representation made use of CPS conversion [14], but later switched to a direct, functional IR, akin to ANF [15]. The motivation for migrating IR forms was in part due the ease of reasoning about programs and compiler optimisations [16].

For the project, the resulting AST from the parser will be transformed into an intermediate representation (IR), based on ANF. An ANF IR was selected over a CPS IR due to its ease in converting to the eventual LLVM IR.

Within this phase, enumerated types are also de-sugared into integer constants, and pattern matching is de-sugared into a series of `if` statements.

4.2.3 Closure Conversion

Supporting higher-order functions requires the introduction of closures, due to the need to capture free variables (variables that are not defined within the function) present inside a function. This phase is done in multiple steps:

1. **Identify free variables**: Free variables are identified by analysing variables used but not declared within a function.

2. **Create closure**: A closure is created for each function that contains free variables. This closure is a structure that contains a pointer to the function, and the values of the free variables.

3. **Rewrite body**: The body of the function is rewritten to reference the free variables from the closure, rather than the original free variables.

4. **Rewrite calls**: Calls to the function are rewritten to pass the closure as an additional parameter. The function pointer is extracted from the closure, and the function is called with the environment as an additional parameter.

4.2.4 Hoisting

Hoisting is the process of moving nested functions and other declarations to the top-level global scope. This ensures that all declarations are visible to the entire program, and that functions can be called from anywhere in the program. This is necessary for the LLVM IR, as functions must be declared before they are called.

4.2.5 LLVM Generation

This final phase is responsible for traversing the ANF IR and generating the corresponding LLVM IR. Any additional runtime functions required for supporting the language features are also generated within this phase, such as functions to `print()`. It is imperative that the type information from the source language is preserved up to this point, as the LLVM IR is strongly typed. The generated LLVM IR is then written to a file, ready for compilation with `llc`.

Chapter 5

Implementation

This chapter details the implementation of the compiler, with the first few sections covering all major phases from parsing to LLVM IR generation. Where appropriate, examples of how the language is represented at each stage of the compilation process will be provided. The final sections of this chapter will specifically cover the implementation and handling of structs, closures and higher-order functions, as these language features first require the full understanding of the implementation in order to be explained effectively.

5.1 Parsing

As the library used for parsing, *FastParse*, is a parser combinator library, the parser is able to closely reflect the formal grammar designed, with each non-terminal symbol being implemented as a function that returns a parser. The parser output is mapped to an abstract syntax tree (AST), represented by Scala case classes.

A condensed example can be seen as follows:

```
sealed trait Exp
case class Call(name: String, args: Seq[Exp]) extends Exp
case class Num(i: Int) extends Exp
case class Bool(b: Boolean) extends Exp
case class Var(s: String) extends Exp

def NumParser[$: P] =
    P(CharsWhileIn("0-9", 1)).!.map(_.toInt)

def BoolParser[$: P] =
    P("true").map(_ => true) |
    P("false").map(_ => false)
```

```
def IdParser[$: P] = P(
    !StringIn("if", "then", "else", "print", "def", "val", "enum", "struct",
    ↪  "true", "false")
    ~ CharIn("A-Za-z_") ~~ CharsWhileIn("A-Za-z0-9_", 0)
).!

def Primary[$: P]: P[Exp] = P(
    (IdParser ~ "(" ~ Exp.rep(0, ",") ~ ")").map(Call) |
    NumParser.map(Num) |
    BoolParser.map(Bool) |
    IdParser.map(Var)
)
```

where `IdParser`, `NumParser` and `BoolParser` define the lexical structure of identifiers, numbers and booleans respectively. This parser allows for the following transformation between the input and the AST:

```
// input
foo(6 + bar(4), 2)
-------------------------------------------------------------------------------
// AST
Call("foo", Seq(
    Op(
        Num(6),
        "+",
        Call("bar", Seq(Num(4)))
    ),
    Num(2)
))
```

A particular implementation detail of the parser is in handling operator precedence. A common grammar for parsing with operator precedence, notated in Extended Backus-Naur Form (EBNF), is as follows:

$$
\begin{aligned}
\langle expr \rangle \quad &::= \langle term \rangle \\
&\mid \langle expr \rangle \text{ `+'} \langle term \rangle \\
&\mid \langle expr \rangle \text{ `-'} \langle term \rangle \\
\\
\langle term \rangle \quad &::= \langle factor \rangle \\
&\mid \langle term \rangle \text{ `*'} \langle factor \rangle \\
&\mid \langle term \rangle \text{ `/'} \langle factor \rangle \\
\\
\langle factor \rangle \quad &::= \langle number \rangle \\
&\mid \text{`('} \langle expr \rangle \text{ `)'}
\end{aligned}
$$

While this correctly handles operator precedence, it does not handle left-associativity of operators if implemented naively. For example, the grammar above would parse $1 - 2 - 3$ $[= -4]$

as $1 - (2 - 3)$ [$= 2$], which is not the desired behaviour. To handle this, the grammar must be modified to include left-associative rules:

$$
\begin{aligned}
\langle expr \rangle &::= \langle term \rangle \ \{ \ (\text{`+'} \mid \text{`-'}) \ \langle term \rangle \ \} \\
\langle term \rangle &::= \langle factor \rangle \ \{ \ (\text{`*'} \mid \text{`/'}) \ \langle factor \rangle \ \} \\
\langle factor \rangle &::= \langle number \rangle \\
&\quad \mid \ \text{`('} \ \langle expr \rangle \ \text{`)'}
\end{aligned}
$$

Using the semantic actions of parser combinators, a left-associative transformation can be applied to the parsed output. This is achieved by defining a recursive function that takes the parsed output and applies the left-associative transformation. The implementation of this using *FastParse* is as follows:

```
def lft(a: Exp, b: Seq[(String, Exp)]): Exp = (a, b) match {
    case (`a`, (b, c) :: next) => lft(Op(a, b, c), next)
    case _ => a
}

def Expr[$: P]: P[Exp] =
    (Term ~ (CharIn("+\\-").! ~ Term).rep).map(lft(_,_))

def Term[$: P]: P[Exp] =
    (Factor ~ (CharIn("/*%").! ~ Factor).rep).map(lft(_,_))

def Factor[$: P]: P[Exp] =
    NumParser.map(Num) |
    "(" ~ Expr ~ ")"
...
```

To parse the source file as a whole, the file is converted into a String, and passed to the parser via the `fastparse.parse(string, parser)` function. If the parser is successful, the AST is returned, otherwise an error message is printed. As mentioned in Chapter 4, care was taken to ensure the grammar created was unambiguous and not left-recursive. The final language grammar specification is available in Appendix A.

5.2 IR Generation

After generating the AST from the parser, it is converted to the ANF IR as a series of let-expressions, defining the scope of variables and their values. This can be seen as equivalent to lambda abstractions in the lambda calculus, with the variable bindings representing the arguments to the lambda, and the body of the let-expression representing the body of the

lambda. The conversion is performed via a Continuation-Passing Style function. Consider the following program and it's ANF-style intermediate data representation:

```
// Program
foo(6 + bar(4), 2)
```

```
// ANF IR
Let("tmp0", Call("bar", [Num(4)]),
    Let("tmp1", Op(Num(6), "+", Var("tmp0")),
        Let("tmp2", Call("foo", [Var("tmp1"), Num(2)])
            Return(Var("tmp2"))
        )
    )
)
```

Observe that the deeper expressions appear before shallower ones in the ANF representation. A simple recursive solution for converting the program to ANF would not work as this structure cannot be created bottom-up. It can be seen that the later expressions of the IR depend on the expressions constructed earlier on, such as `tmp2` depending on `tmp1`, which in turn depends on `tmp0`.

To solve this 'inversion' of calculating values, CPS allows for passing a continuation function representing the context in which a value is used. This allows for the expression to remain incomplete until the value it depends on is assigned to a variable.

The IR generation function as implemented takes an AST expression and a continuation function, that takes a value and returns an ANF expression. The CPS function itself then returns the full program under an ANF expression. A simplified version of the CPS function is as follows:

```
def CPS(e: Exp)(k: KVal => KAnf) : KAnf = e match {
    // Values are passed to the continuation function
    case Var(s) => k(KVar(s))
    case Num(i) => k(KNum(i))
    case Bool(b) => k(KBool(b))
    case Flt(f) => k(KFloat(f))
    ...
    // Other expressions are handled by creating a new continuation function
    case Op(e1, o, e2) => {
        val z = counter.Fresh("tmp")
        CPS(e1)((y1) =>
            CPS(e2)((y2) =>
                KLet(z, Kop(o, y1, y2), k(KVar(z)))
            )
        )
    }
```

```
    case Func(name, args, body) =>
        KFun(name, args, CPS(body)((y) => KReturn(y)), k(KVar(name)))
    case StructDef(name, items) =>
        KStructDef(Struct(name, items), k(KVar(name)))
    ...
}
```

The `Fresh` function is used to generate a globally unique name for each temporary variable introduced, in order to fulfil the SSA requirements that LLVM IR has.

For a cleaner function interface (and to hide the implementation detail of the continuation), the CPS function is wrapped in a higher-order function that takes the AST expression, which calls the CPS function with the AST and a continuation that binds the final result to a return statement.

5.2.1 Handling typing

As the LLVM IR is statically typed, the ANF IR to be generated must also be typed. While more complex type systems exist, such as Hindley-Milner type inference, the type system used in this project heavily leans on the existing type system of LLVM IR, with the programmer being required to declare the types of variables alongside their definitions. This is done by adding type hints to variable definitions, function arguments and return types.

If the type hint specified by the programmer is outside the set of pre-defined types in the language (e.g. `Int`, `Float`, etc.), the type is marked in the parser as a user-defined type until after the ANF IR generation process, where the type is resolved to a concrete type (e.g. a struct).

These types are then propagated through the ANF IR generation process via a type environment, which is a map from variable names to their types. The type environment is updated as new variables are bound to typed values, and the types of expressions are checked against the type environment. As the type environment is recursively passed through the function, the scope of the type environment will naturally reflect available types at any given point in the program. Using the example from earlier, an updated version of the CPS function that includes type propagation is as follows:

```
type TypeEnv = Map[String, Type]

def CPS(e: Exp, ty: TypeEnv)(k: (KVal, TypeEnv) => KAnf) : KAnf = e match {
    // Values and their types are passed to the continuation function
    case Var(s) => k(KVar(s, ty(s)), ty)
```

```
    case Num(i) => k(KNum(i), ty)
    case Bool(b) => k(KBool(b), ty)
    case Flt(f) => k(KFloat(f), ty)
    ...
    // Other expressions are handled by creating a new continuation function
    case Op(e1, o, e2) => {
        val z = counter.Fresh("tmp")
        CPS(e1, ty)((y1, t1) =>
            CPS(e2, t1)((y2, t2) =>
                KLet(z, Kop(o, y1, y2), k(KVar(z, y1.get_type), t2))
            )
        )
    }
    case Func(name, args, ret, body) => {
        // update the type environment for the function body
        val body_ty = ty
            ++ args.map{case (x, t) => (x, t)}
            + (name -> FnType(args.map(_._2).toList, ret))
        // update the type environment for after the function itself
        val return_ty = ty
            + (name -> FnType(args.map(_._2).toList, ret))
        KFun(name, args, ret, CPS(body, body_ty)((y, _) => KReturn(y)),
        ↪  k(KVar(name), return_ty))
    }
    case StructDef(name, items) =>
        // update the ty with struct type, as well as items inside the struct
        val updated_ty = ty
            + (name -> UserType(name))
            ++ items.map{case (x, t) => (s"$name.$x", t)}
        KStructDef(Struct(name, items), k(KVar(name), updated_ty))
    ...
}
```

5.3 Enumerated Types and Pattern Matching

The implementation of enums and pattern matching requires the consideration of two main aspects:

1. The definition of enums and their values.
2. Referencing enums in comparisons and pattern matching expressions.

Consider the following program containing an enum definition and a pattern matching expression:

```
enum Days = Spring | Summer | Autumn | Winter;

def main() = {
    val contrivedExample: Season = Season::Spring;
    print(contrivedExample match {
        case Season::Spring => true
        case Season::Autumn => true
        case _ => false
    });
    0
}
// Output: true
```

An example program that demonstrates the use of enums and pattern matching. This example will be used throughout this section to illustrate the changes required.

The LLVM IR does not have a native representation of enums, nor does it have native control flow instructions for pattern matching. To accommodate these language features, a pre- and post-processing step is introduced to the ANF IR generation.

The pre-processing step is responsible for extracting all enum definitions from the AST, collecting them into a map from enum names to their possible values, and transforming match expressions into a series of if-else expressions. This step is done prior to the ANF IR generation as it eliminates the need to handle the pattern matching transformation within the ANF IR generation itself. Using the example above, the pre-processing step would transform the program into:

```
def main() = {
    val contrivedExample: Season = Season::Spring;
    print(
        if (contrivedExample == Season::Spring) {
            true
        } else if (contrivedExample == Season::Autumn) {
            true
        } else {
            false
        }
    );
    0
}
```

For illustration purposes, this code is a representation of the changes that would be made to the AST.

This step is implemented as a series of recursive functions that traverse the AST and transform the program accordingly. Where a default catch-all case is provided in the match

expression, the transformation is straightforward, as the default case can be directly translated to an else statement, with the remaining cases following it ignored. Note that for the match expression, a 'no-op' instruction is added to the end of the if-else chain to ensure that the match expression returns a value. This is necessary in the event where no default (else) case is provided. The no-op instruction is eliminated by the LLVM IR optimiser during compilation, leaving the semantics of the program intact.

```
// Removes Enum definitions from the AST and returns them separately
def extract_enums(e: Exp) : (Map[String, Type], Exp) = e match {
    case Sequence(e1: EnumDef, e2) =>
        val (enums, e_) = extract_enums(e2)
        (enums + (e1.name -> EnumType(e1.vals)), e_)
    // ... Other recursive cases ...
    case e: EnumDef => (Map(e.name -> EnumType(e.vals)), e)
    case e => (Map(), e)
}

// Convert match statements to if-else statements
def transform_match_to_if(e: Exp) : Exp = e match {
    case Match(a, cases) =>
        // Fold start value is a no-op (0 + 0) if no default case is given
        cases.foldRight[Exp](Op(Num(0), "+", Num(0)))((mcase, acc) => {
            val MCase(root, item, exp) = mcase
            if root == "" && item == "_" then
                transform_match_to_if(exp)    // default
            else
                If(Op(Var(a), "==", EnumRef(root, item)),
                ↪  transform_match_to_if(exp), Some(acc))
        })
    // ... Other recursive cases ...
    case e => e
}
```

The post-processing step is responsible for replacing enum references with their corresponding integer values, using the map generated in the pre-processing step. The integer value assigned to each enum value is determined by the index in which they are defined in the enum definition, identical to the behaviour of C and C++ enums. This step is done after the ANF IR generation to allow for the specific enum types to be properly propagated through the ANF IR. The post-processing step would transform the program into:

```
def main() = {
    val contrivedExample: Int = 0;
    print(
        if (contrivedExample == 0) {
            true
        } else if (contrivedExample == 2) {
            true
        } else {
            false
        }
    );
    0
}
```

For illustration purposes, this code is a representation of the changes that would be made to the ANF IR.

This step is also implemented as a series of recursive functions that traverse the ANF IR, with the enum replacement carried out as a simple map lookup.

```
// ...
case v: KEnum =>
    en_map.get(v.root) match {
        // check if enum is in map
        case Some(EnumType(items)) =>
            val idx = items.indexOf(v.item)
            // check if valid enum value
            if idx == -1 then
                throw new Exception("Undefined enum value")
            KNum(idx)
        case _ => throw new Exception("Undefined enum type")
    }
// ...
```

At this point, all references to enums have been removed, and the program is in a form that can almost directly be translated to LLVM IR.

5.4 LLVM IR Generation

After both the ANF IR generation and post-processing steps, the state of the program is such that the chain of let-expressions encompasses the entire program, with the final continuation function being the return statement. This includes the definitions of functions, structs and closures, as well as the main function itself. To extract certain constructs like function definitions from these let-expressions, a recursive function is used to hoist them out and into the top-level of the program. This is done to ensure that definitions are available globally, as they are in the

source language.

```
def hoist(e: KAnf): (List[CFunc], KAnf, List[Env], List[Struct]) = e match {
    case KFun(fnName, args, ret, body, next) => {
        val (fns, e, envs, structs) = hoist(body)
        val (fns2, e2, envs2, structs2) = hoist(next)
        val entry = counter.Fresh("entry")
        val fn = CFunc(fnName, args, ret, e)
        (fn :: fns ::: fns2, e2, envs ::: envs2, structs ::: structs2)
    }
    case KIf(x1, e1, e2) => {
        val (fns, t, envs, structs) = hoist(e1)
        val (fns2, f, envs2, structs2) = hoist(e2)
        val thn = counter.Fresh("then")
        val els = counter.Fresh("else")
        (fns ::: fns2, KIf(x1, t, f), envs ::: envs2, structs ::: structs2)
    }
    case KLetEnv(x, env: Env, next) => {
        val (fns, e1, envs, structs) = hoist(next)
        (fns, KLetEnv(x, env, e1), env :: envs, structs)
    }
    case KLet(x, v, next) => {
        val (fns, e1, envs, structs) = hoist(next)
        (fns, KLet(x, v, e1), envs, structs)
    }
    case KConst(x, v, e) => {
        val (fns, e1, envs, structs) = hoist(e)
        (fns, KConst(x, v, e1), envs, structs)
    }
    case KStructDef(struct, e) => {
        val (fns, e1, envs, structs) = hoist(e)
        (fns, KStructDef(struct, e1), envs, struct :: structs)
    }
    case _ => (Nil, e, Nil, Nil)
}
```

The final step of the compilation process is the generation of LLVM IR from the ANF IR. A prelude is added to the top of all programs, containing the necessary LLVM IR code to define runtime library code, such as `print_*` functions. While limited in the scope of this project, the prelude can be expanded to include more complex library functions, such as the implementation of language features like strings. This could be achieved by writing the implementation in a higher-level language like C, and then appending the emitted LLVM IR to the final executable.

The LLVM IR generation is done by recursively traversing the ANF IR, emitting the corresponding LLVM IR for each construct. Certain constructs, such as arithmetic operations, are conditional on the type of the values involved. For example, the LLVM IR for adding two integers is different from adding two floating-point numbers.

Other constructs, such as floating point literals themselves, require special handling as they cannot be emitted directly. The LLVM IR require floating point literals to be exact

representations of the floating point number, using the 64-bit IEEE-754 standard. This requires all (32-bit) floats to first be extended to (64-bit) doubles, before being converted to their hexadecimal representation.

The LLVM IR is then written to a file, which can be in turn be compiled using the LLVM toolchain, or ran using an LLVM interpreter.

5.5 Composite Types

The implementation of composite types (structs) requires the handling of three main concerns:

1. The definition of structs and their members.
2. The allocation of memory for a new instance of a struct.
3. The referencing of struct members.

Despite structs sharing the same two concerns as enums, the implementation detail is quite different from that of enums, as the former requires the generation of a new type and the allocation of memory for the struct in the LLVM IR. The following program demonstrates the use of structs:

```
struct Point = {
    x: Int,
    y: Int
};

def main() = {
    val p: Point = Point(1, 2);
    print(p.x + p.y);
    0
}
// Output: 3
```

An example program that demonstrates the use of structs. This example will be used throughout this section to illustrate the changes required.

After the ANF IR generation, the struct definition is enclosed within the let-expression encompassing the entire program. As with function definitions, the struct definition is hoisted out into the top-level of the program.

Structs in the LLVM IR are represented as a series of elements, with each element being a member of the struct. These elements do not have names as in the source language, but are instead accessed by their index. It is therefore important to keep track of the order of elements

in the struct definition throughout generation. The struct definition is then emitted as a new type in the LLVM IR, like so:

```
// Program
struct Point = {x: Int, y: Int};
```

```
; LLVM IR
%Point = type { i32, i32 }
```

When creating a new instance of a struct, memory must be allocated for it. Memory allocation in the LLVM IR can either be done on the stack or the heap. Allocating to the stack provides a simpler and efficient solution, as the memory is automatically deallocated when a function returns. However, this raises implications for the lifetime of the struct, as it is only valid within a narrow scope as determined by the stack. Allocating to the heap (e.g. `malloc` in C) provides a more flexible solution, as the memory is valid for the lifetime of the program, but requires either explicit manual deallocation or the creation of a garbage collector.

For the purposes of this project, memory allocation is done on the stack using the `alloca` instruction in the LLVM IR, returning a pointer to the allocated memory. For each element in the struct to be allocated, a `getelementptr` instruction is used to calculate the address of the element within the struct via its index, and a `store` instruction is used to store the value of the element. The following is the LLVM IR for the creation of a new instance of the `Point` struct:

```
// Program
val p: Point = Point(1, 2);
```

```
; LLVM IR
%p = alloca %Point
%tmp0 = getelementptr %Point, %Point* %p, i32 0, i32 0
store i32 1, i32* %tmp0
%tmp1 = getelementptr %Point, %Point* %p, i32 0, i32 1
store i32 2, i32* %tmp1
```

The referencing of struct members is done by using the `getelementptr` instruction to calculate the address of the struct member via its index, and the `load` instruction to load the value of the member into a local variable. To access the `x` and `y` members of the `Point` struct, the following LLVM IR is generated:

```
// Program
p.x + p.y
```

```
; LLVM IR
%tmp2 = getelementptr %Point, %Point* %p, i32 0, i32 0
%tmp3 = load i32, i32* %tmp2
%tmp4 = getelementptr %Point, %Point* %p, i32 0, i32 1
%tmp5 = load i32, i32* %tmp4
%tmp6 = add i32 %tmp3, %tmp5
```

5.6 Closures and Higher-order functions

As mentioned in Chapter 4, there are three steps to implementing closures and higher-order functions:

1. Identifying free variables.
2. Performing closure conversion.
3. Hoisting the closure to the top-level.

The following program demonstrates the use of closures and higher-order functions:

```
def foo(x: Int): (Int) => Int = {
    def bar(y: Int): Int = x + y;
    bar
};

def main() = {
  val add: (Int) => Int = foo(9);
  print(add(10));
  0
}
// Output: 19
```

An example program that demonstrates the use of closures and higher-order functions. This example will be used throughout this section to illustrate the changes required.

The general idea is to convert the function into a closure, which will be a LLVM structure type containing the free variables of the closure. The closure will then be rewritten to access the environment for these variables. For example, the function definition `bar(y: Int): Int` in the example above will convert to `bar(env: BarEnv, y: Int): Int`.

For identifying free variables within a function the ANF IR is traversed, while collecting all variables that are not defined within the function itself. This is done by maintaining a set of variables that are defined within the function, and adding any variable that is referenced but

not defined to the set of free variables. For instance, the function `bar` in the example above contains free variable `x`. A series of recursive functions are used to achieve this:

```
def free_anf(e: KAnf): Set[KVar] = e match {
    case KFun(fnName, args, _, body, in) =>
        (free_anf(body) ++ free_anf(in))
            .filterNot(x => args.map(_._1).contains(x.s) || x.s == fnName)
    case KLet(x, e1, e2) => (free_exp(e1) ++ free_anf(e2)).filterNot(_.s == x)
    case KConst(x, v, e) => free_val(v) ++ free_anf(e)
    case KIf(x, e1, e2) => free_anf(e1) ++ free_anf(e2)
    // ... Other recurisve cases ...
}

def free_exp(e: KExp): Set[KVar] = e match {
    case Kop(o, v1, v2) => free_val(v1) ++ free_val(v2)
    case KStructDec(struct, vals) => vals.flatMap(free_val).toSet
    case KCall(o, vrs) => vrs.flatMap(free_val).toSet
}

def free_val(v: KVal): Set[KVar] = v match {
    case k: KVar => Set(k)
    case _ => Set()
}
```

Once the free variables have been identified, the environment is created to store these variables. A fresh global name is allocated to it to avoid any naming collisions. Included in the environment is a pointer to the function, allowing the function to be passed around as a first-class object: `BarEnv = [Ptr, Int]`. This environment is effectively the closure, as it contains all the information needed for the function to be 'closed' over. Note that for each new closure, a new environment type is needed, as the free variables may differ between closures.

The function body is then rewritten to access the environment for these variables. By passing the environment in as the first argument, the function can index into the corresponding field where a free variable would be referenced. The final return statement of the function is also modified to return the closure instead of the function itself. As an example, The function `bar` in the example above is rewritten to access the environment for the variable `x`:

```
def foo(x: Int): (Int) => Int = {
    def bar(env: BarEnv, y: Int): Int = env[1] + y;
    bar // This is now type of BarEnv, not a function.
};
```

For illustration purposes, this code is a representation of the changes that would be made to the ANF IR. This is not valid code in the project's current implementation of this language.

Note that the return type of `foo` is now incorrect! It should be returning the closure `BarEnv`

instead of a function `(Int) => Int`. This is resolved by traversing the ANF IR again and rectifying all references to the function to instead reference the closure.

```
def convert(e: KAnf): (KAnf, Option[Env]) = e match {
    case KFun(fnName, args, ret, body, in) => {
        // Collect free variables
        val fvs = free_anf(body)
            .filterNot(x => args.map(_._1).contains(x.s)).toList
        ...
        // Create environment
        val envId = counter.Fresh("env")
        val newArgs = (envId, EnvType(envId)) +: args
        val newRet = if next_env.isEmpty then ret else
        ↪  EnvType(next_env.get.name)
        val newType = FnType(newArgs.map(_._2), newRet)
        val env = Env(envId, KVar(fnName, newType) :: fvs)

        // For each free variable, reference it from environment
        val (body2, _) = fvs.zipWithIndex.foldLeft(converted_body, 1) {
                // LET x = env[i] IN anf
                case ((next, i), (ssaVar, _)) =>
                    (KLetEnvRef(ssaVar.s, Ref(KVar(envId, EnvType(envId)), i),
                    ↪  next), i + 1)
            }

        val (in2, _) = convert(in)
        // LET fn = (@fn, fvs...) IN anf
        val let_env = KLetEnv(fnName, env, in2)

        // Propagate the new type through the program
        val updated_body = update_types(body2, Map(fnName -> newType))
        val updated_in = update_types(let_env, Map(fnName -> newType))
        (KFun(fnName, newArgs, newRet, updated_body, updated_in), Some(env))
    }
// ... Other cases ...
}
```

For function calls that are now calling a closure, the function pointer within the environment must be dereferenced before the function can be called. As the environment is constructed by placing the function pointer as the first element, the function pointer can be accessed by indexing into the environment with the integer 0. The function call is then rewritten to use the function pointer. For example, the call to `add` in the example above is rewritten to access the function pointer from the closure:

```
def main() = {
    val add: BarEnv = foo(9);
    print(add[0](10));
    0
};
```

For illustration purposes, this code is a representation of the changes that would be made to the ANF IR. This is not valid code in the project's current implementation of this language.

After closure conversion is complete and all relevant functions are closed, nested functions can be safely hoisted to the top-level of the program. This is done similarly to the hoisting of function and struct definitions. At this point, the closures are in a form that can be directly translated to LLVM IR.

For lowering closures to LLVM IR, a structure type is created for each environment hoisted to the top-level. In the case of the `BarEnv` environment, the LLVM IR would be:

```
; LLVM IR
%env_bar = type { i32 (%env_bar*, i32)*, i32 }
```

The first type, `i32 (%env_bar*, i32)*`, is a function pointer that takes a pointer to the environment and a 32-bit integer (`i32`) as arguments, returning an `i32` (i.e., the function `bar`). The second type, `i32`, is the free variable `x` in the closure.

When creating a new instance of a closure, memory must be allocated for it. This is performed similarly to the allocation of structs, where stack memory is allocated using the `alloca` instruction in the LLVM IR, returning a pointer to the allocated memory. For each element in the closure to be allocated, a `getelementptr` instruction is used to calculate the address of the element within the closure via its index, and a `store` instruction is used to store the value of the element. As the function was hoisted to the top-level, it is globally accessible in the LLVM IR, and so can be added into the closure by name directly.

When calling a closure, `getelementptr` is used to locate the function pointer within the closure, and the function is then called using the `call` instruction after dereferencing the function pointer with `load`.

Finally, when accessing the free variables within the closure, `getelementptr` is used to locate the appropriate field within the closure via its index, and `load` is used to load the value of the free variable into a local variable.

The program example provided at the beginning of this section would be translated to the following LLVM IR, after closure conversion:

```
%env_bar = type { i32 (%env_bar*, i32)*, i32 }

define %env_bar* @foo (i32 %x) {
    %bar = alloca %env_bar
    %bar0 = getelementptr %env_bar, %env_bar* %bar, i32 0, i32 0
    store i32 (%env_bar*, i32)* @bar, i32 (%env_bar*, i32)** %bar0
    %bar1 = getelementptr %env_bar, %env_bar* %bar, i32 0, i32 1
    store i32 %x, i32* %bar1
    ret %env_bar* %bar
}

define i32 @bar (%env_bar* %env, i32 %y) {
    %tmp_3 = getelementptr %env_bar, %env_bar* %env, i32 0, i32 1
    %x = load i32, i32* %tmp_3
    %tmp_4 = add i32 %x, %y
    ret i32 %tmp_4
}

define i32 @main () {
    %add = call %env_bar* @foo(i32 9)
    %tmp_0 = getelementptr %env_bar, %env_bar* %add, i32 0, i32 0
    %tmp_1 = load i32 (%env_bar*, i32)*, i32 (%env_bar*, i32)** %tmp_0
    %tmp_2 = call i32 %tmp_1(%env_bar* %add, i32 10)
    call void @print_i32(i32 %tmp_2)
    ret i32 0
}
```

Chapter 6

Evaluation

This chapter will evaluate the effectiveness of the compiler developed in this project, by comparing the generated LLVM IR code and the performance of the compiled code against other compilers utilising the LLVM as a backend. A quantitative analysis will be discussed first, followed by an analysis on the limitations of the compiler as implemented, and potential improvements that could be made.

For the purpose of transparency, the CPS implementation in this project made use of (and appropriately transformed) approximately 70 lines of code found in the KCL module 6CCS3CFL.

6.1 Software Testing

To ensure the correctness of the compiler, as well as ensuring the requirements set out in Chapter 3 were met, a suite of unit tests were written to test the various phases of the compiler. The unit tests were written using the ScalaTest framework, covering the parser, enum refactor, IR generator, and closure conversion steps. The tests were written to cover a range of scenarios, including edge cases. Where appropriate, mock objects were used to isolate the unit under test from its dependencies (e.g. the generation of unique variable names).

Provided are example unit tests for the enum refactor stage:

```
class EnumSpec extends AnyFlatSpec {
    // ... other tests ...

    behavior of "Match to If transformation"

    it should "correctly transform a match statement with a default case" in {
```

```
        val m = Match("a", List(MCase("A", "B", Num(6)), MCase("", "_",
        ↪  Num(5))))
        val expected = If(Op(Var("a"), "==", EnumRef("A", "B")), Num(6),
        ↪  Some(Num(5)))
        assert(transform_match_to_if(m) == expected)
    }

    it should "correctly transform a match statement with multiple cases" in {
        val m = Match("a", List(MCase("A", "B", Num(6)), MCase("C", "D",
        ↪  Num(5))))
        val expected = If(Op(Var("a"), "==", EnumRef("A", "B")), Num(6),
        ↪  Some(If(Op(Var("a"), "==", EnumRef("C", "D")), Num(5),
        ↪  Some(Op(Num(0), "+", Num(0))))))
        assert(transform_match_to_if(m) == expected)
    }

    it should "correctly transform a match statement with multiple cases and a
    ↪  default case" in {
        val m = Match("a", List(MCase("A", "B", Num(6)), MCase("C", "D", Num(5)),
        ↪  MCase("", "_", Num(4))))
        val expected = If(Op(Var("a"), "==", EnumRef("A", "B")), Num(6),
        ↪  Some(If(Op(Var("a"), "==", EnumRef("C", "D")), Num(5),
        ↪  Some(Num(4)))))
        assert(transform_match_to_if(m) == expected)
    }

    it should "correctly skip other cases after a default case" in {
        val m = Match("a", List(MCase("A", "B", Num(6)), MCase("", "_", Num(5)),
        ↪  MCase("C", "D", Num(4))))
        val expected = If(Op(Var("a"), "==", EnumRef("A", "B")), Num(6),
        ↪  Some(Num(5)))
        assert(transform_match_to_if(m) == expected)
    }
}
```

As seen, the tests are written in a BDD (Behaviour-driven development) style, with each test case describing the expected behaviour of the function under test. In total, 50 unit tests were written to test the various phases of the compiler.

End-to-end testing was used to evaluate the LLVM IR output. A suite of test programs were written to test the various features of the language, including functions, structs, enums, and closures. The test programs were compiled using the project compiler, and the output LLVM IR code was manually inspected and executed to ensure that the programs produced the correct output.

6.2 Performance Benchmarking

A key language feature ubiquitous in functional programming languages is the ability to define recursive functions. A pure functional language paradigm will utilise recursion as a primary

method of iteration, as opposed to the traditional imperative loop constructs. Therefore, the ability for a compiler to optimise recursive functions is crucial for the performance of the generated code.

To evaluate the performance of the project compiler, two recursive functions were chosen for benchmarking: the Ackermann function and the Fibonacci function.

6.2.1 Ackermann Function

The Ackermann function is a recursive computable function that grows rapidly with respect to its inputs, and is a classic example used to benchmark the ability of a compiler to optimise recursive calls.

The Ackermann function is defined as follows:

$$ack(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ ack(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ ack(m-1, ack(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Despite the only arithmetic operation being an addition or subtraction of one, even small inputs such as $ack(4,2)$ result in a value of 2×10^{19728} due to the nested recursion, implying that the output value of the function is indicative of the running time of the function itself. Without proper optimisation, the Ackermann function can quickly exhaust the call stack, and result in a stack overflow error. For evaluation purposes, the Ackermann function will be tested with $m = 3$ to restrict function growth to (at most) exponential.

Below is the implementation of the Ackermann function in all three languages:

```
def ack(m:Int, n:Int): Int =
    if m == 0 then
        n+1
    else if n == 0 then
        ack(m-1, 1)
    else
        ack(m-1, ack(m,
        ↪  n-1));

def main() = {
    print(ack(3,10))
    0
}
```

$ack(m,n)$ in the project.

```
ack :: Int -> Int -> Int
ack 0 n = n+1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m
↪  (n-1))

main = print (ack 3 10)
```

$ack(m,n)$ in Haskell.

```
#include <stdio.h>

int ack(int m, int n) {
    if (m == 0) {
        return n+1;
    } else if (n == 0) {
        return ack(m-1, 1);
    } else {
        return ack(m-1, ack(m,
        ↪  n-1));
    }
}

int main() {
    printf("%d\n", ack(3, 10));
    return 0;
}
```

$ack(m,n)$ in C.

6.2.2 Fibonacci Function

Another recursive function that will be tested is the Fibonacci function, defined as follows:

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n > 1 \end{cases}$$

The Fibonacci function is another example of a recursive function that, while not as computationally intensive as the Ackermann function, is still a good benchmark for evaluating the performance of a compiler when generating code for recursive functions.

Below is the implementation of the Fibonacci function in all three languages:

```
def fib(n: Int): Int =
    if n == 0 then
        0
    else if n == 1 then
        1
    else
        fib(n-1) + fib(n-2);

def main() = {
    print(fib(40))
    0
}
```

fib(*n*) in the project.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib
↪  (n-2)

main = print (fib 40)
```

fib(*n*) in Haskell.

```
#include <stdio.h>

int fib(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) +
        ↪  fib(n-2);
    }
}

int main() {
    printf("%d\n", fib(40));
    return 0;
}
```

fib(*n*) in C.

6.2.3 Benchmarking Methodology

To properly evaluate the performance of the project compiler, the generated LLVM IR code was compiled to a binary using the LLVM compiler (`llc`), and then linked against the C standard library `libc` using the GNU Compiler Collection (`gcc`). Linking against `libc` is necessary to provide the required `printf` function for output, as it not defined in the LLVM. The resulting binary was then executed to measure the time taken to compute both the Ackermann and Fibonacci functions for various values of n.

The project compiler was benchmarked against Haskell (compiled with GHC), and C (compiled with Clang). Haskell was chosen as a comparison due to its functional programming nature that also has a compiler compatible with the LLVM backend. C was chosen due to its high performance, and high compatibility with the LLVM.

To ensure a fair comparison, and to evaluate the effect of the LLVM toolchain on compiler frontends, all compilers were compiled at two optimisation levels: `-O0` and `-O3`. The `-O0` flag disables all optimisations, while the `-O3` flag enables all optimisations[1].

[1]The GHC compiler actually needs two flags to enable all optimisations; one for GHC (`-O2`), and one for the LLVM optimiser (`-optlo-O3`).

Table 6.1: Results for the Ackermann (left) and Fibonacci (right) functions (in seconds)

Lang	Project		C		Haskell		Lang	Project		C		Haskell	
	`-O0`	`-O3`	`-O0`	`-O3`	`-O0`	`-O3`		`-O0`	`-O3`	`-O0`	`-O3`	`-O0`	`-O3`
$ack(3,10)$	0.080	**0.024**	0.120	**0.024**	1.334	0.101	$fib(40)$	0.713	0.207	0.428	**0.206**	3.724	0.367
$ack(3,11)$	0.312	0.093	0.490	**0.092**	5.449	0.387	$fib(41)$	1.152	**0.333**	0.705	0.335	15.18	0.593
$ack(3,12)$	1.267	0.370	1.976	**0.366**	22.63	1.578	$fib(42)$	1.867	0.535	1.112	**0.532**	24.55	0.953
$ack(3,13)$	5.071	1.502	8.088	**1.495**	99.05	6.540	$fib(43)$	3.019	**0.860**	1.804	0.874	39.58	1.534
$ack(3,14)$	20.75	6.193	32.90	**6.130**	475.3	28.08	$fib(44)$	4.870	1.389	2.925	**1.388**	64.31	2.486
$ack(3,15)$	–	24.74	–	**23.26**	2237	127.1	$fib(45)$	7.878	**2.250**	4.767	2.262	104.2	4.044

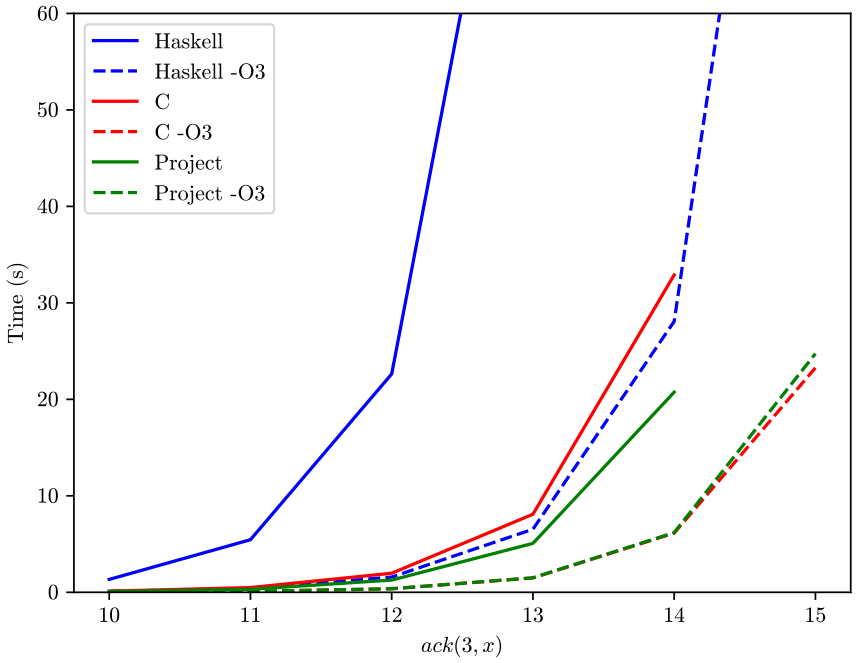

Figure 6.1: A graph reflecting the results of the Ackermann function benchmark.

Each compiler was benchmarked across both optimisation levels (`-O0` and `-O3`). The Ackermann function was tested as $ack(3, n)$ with n ranging from 10 to 15 inclusive, while the Fibonacci function was tested as $fib(n)$ with n ranging from 40 to 45 inclusive. Each function was run ten times for each input value of n, and the average time taken was recorded.

Results were obtained on a machine with an Intel Core i7-13620H CPU and 32 GB of RAM, running Ubuntu 22.04.4 LTS under WSL2.

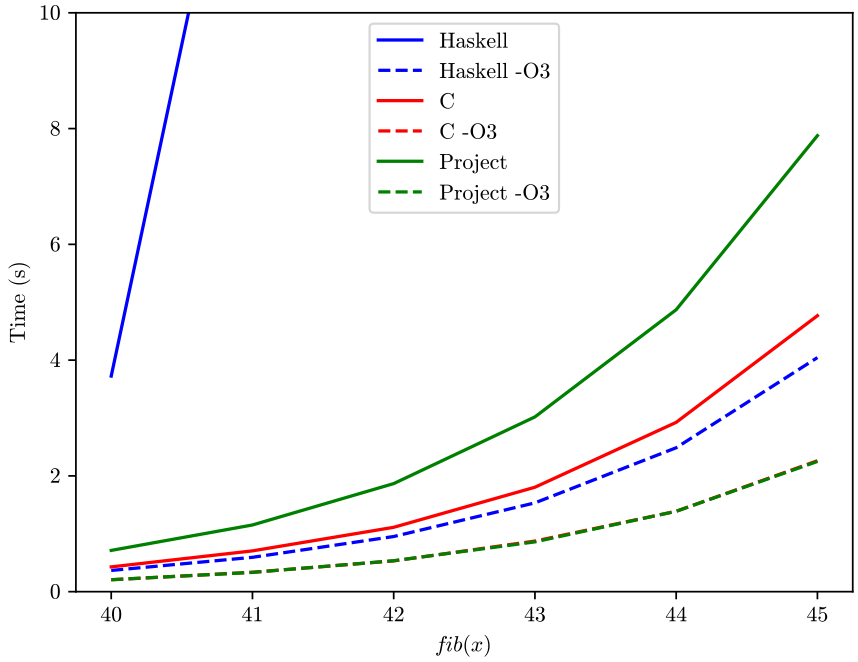

Figure 6.2: A graph reflecting the results of the Fibonacci function benchmark. Note the C and Project compilers are identical at `-O3`.

6.2.4 Results

The results of the benchmarks are shown in Table 6.1, and graphed in Figures 6.1 and 6.2 for the Ackermann and Fibonacci functions respectively.

The results show that, at optimisation level `-O0`, the project compiler performs significantly better than Haskell, but is inconclusive when compared to C. For the Ackermann function, the project compiler consistently performs better than C across all inputs. However, the opposite is true for the Fibonacci function, where the project compiler consistently performs worse than C for all inputs.

On analysing the generated LLVM IR code, Clang (with all optimisations disabled using `-O0`) appears to copy all function arguments to the stack as local variables before using them, occurring for each function invocation. This results in many `alloca`, `store` and `load` instructions being called per function call. The project compiler, on the other hand, utilises the LLVM's SSA form to eliminate unnecessary memory operations, and instead uses the function arguments directly. This however does not explain the performance difference between the project compiler and C for the Fibonacci function and remains an area for further investigation.

A common behaviour observed in both the project compiler and Clang is the failure to compute the Ackermann function for $ack(3, 15)$, terminating with a segmentation fault. This is

due to the Ackermann function rapidly exhausting the call stack from the nested recursion. The project compiler does not have an implementation for tail call optimisation, and Clang does not enable it by default.

On the other hand, the Haskell compiler is able to compute the function for $ack(3, 15)$ without any optimisations. This is likely due to Haskell's lazy evaluation strategy, as it does not create stack frames on each function call, but when an unevaluated value (a 'thunk') is evaluated. Due to this lack of a traditional call stack, Haskell is able to compute the Ackermann function without running out of stack space.

At optimisation level `-O3`, all compilers expectedly perform better than at `-O0`, but at different rates. Haskell sees the most significant improvement, with both the Ackermann and Fibonacci functions being computed in a fraction of the time compared to `-O0`. The project compiler and Clang also see modest improvements, where they now both perform almost identically for all inputs. Haskell, however, still performs significantly worse than the other two compilers, as the GHC compiler appears to particularly struggle optimising for the Ackermann function.

The generated LLVM IR code for the project compiler and Clang are now very similar, with both utilising tail call optimisation to reduce the number of recursive calls inside the Ackermann function down to one. This results in a significant reduction in the number of stack frames required, and allows the function to be computed for larger inputs, such as $ack(3, 15)$. This also explains the very similar performance of the two compilers at `-O3`, as both compilers are able to leverage the LLVM's optimisation passes to generate efficient code.

Despite primitive nature of the project compiler compared to the other well-established compilers, the results of the benchmark show that by leaning on the LLVM toolchain to perform optimisations, the project compiler is able to generate efficient code in-line with other compilers utilising the LLVM as a backend. Certain optimisations such as tail call optimisation can be leveraged 'for free' without the need to implement them in the compiler itself.

6.3 Limitations

It is near impossible to claim any compiler is perfectly implemented against the specification and semantics of a language. This project is no exception, and due to time constraints has a number of limitations that prevent it from being considered a complete compiler. There are also additional functional programming features that could be implemented to improve the experience of the end-user. The following sections will discuss some specific limitations of the

project compiler, and potential improvements that could be made.

6.3.1 Sum Types

While the compiler supports enumerated types, the more general concept of sum types (also known as tagged unions) are not supported. Sum types are a common feature of functional programming languages – particularly when paired with pattern matching – and are used to represent a value that can take on one of several different forms.

For example, a sum type in Rust can be defined as follows:

```
enum Foo {
    Bar(bool),
    Baz(i32)
}
```

Note that the type `Foo` can take on one of two forms: `Bar` with a boolean value, or `Baz` with an integer value. This differs from the enumerated types supported by the project compiler, which can only take on one of a fixed set of integer values.

Support for sum types cannot be implemented using the same method as enums, as the value of a sum type is not known at compile time. Rather, multiple LLVM structures would need to be generated:

```
%Foo = type { i8, [4 x i8] } ; tag, data
%Foo_Bar = type { i8, i1 }   ; tag (0), boolean
%Foo_Baz = type { i8, i32 }  ; tag (1), integer
```

The base type, `Foo`, would contain an integer 'tag' to indicate which variant of the sum type it is, and an array of bytes to store the data associated with the variant. The `Foo_Bar` and `Foo_Baz` types represent the exact layout of the data for each variant. Creating a variant of type `Foo` would require allocating memory for the base type, setting the tag to the appropriate value for the variant, bit casting the memory address to the appropriate variant type, and then storing the data in the appropriate field. Likewise, extracting the data from a value of type `Foo` would require branching on the tag, bit casting the value to the appropriate variant type depending on the tag, and then extracting the data from the appropriate fields.

Pattern matching on sum types would also require an implementation of a more complex algorithm that binds variables to the data associated with each variant. An ML style pattern matching implementation as seen in [17] would be suitable, as it utilises a heuristic to avoid

generating redundant branches.

As can be seen, implementing sum types is not trivial, and would not only require a refactor of the code generation phase, but also a redesign of the type system to support the additional complexity of sum types.

6.3.2 Type System

The project compiler has a very basic type system, and does not support many of the features found in more advanced type systems. For example, the type system does not support type inference or polymorphic types. The type system also does not support type classes, which are used to define a set of functions that can be implemented for a type, and are used to provide ad-hoc polymorphism. A more robust type system would allow for more primitives to be defined, and would allow for more expressive programs to be written.

Implementing a more advanced type system would require a significant refactor of the type checker phase, and would require a redesign of the type checker to support the additional complexity of the type system. It would also require the implementation of additional type checking rules to support the new features of the type system. As mentioned earlier, the Hindley-Milner type inference algorithm could be implemented to provide type inference for the project compiler. This would allow the compiler to infer the types of variables and expressions without requiring the user to provide explicit type annotations.

6.3.3 Heap-allocated variables

The project compiler does not support heap-allocated variables. All variables are stack-allocated, and are deallocated when they go out of scope. This can be problematic for programs that require variables to persist beyond the scope in which they were defined, such as returning a struct from a function. Additionally, the implementation of data structures based on lists is limited without heap-allocated variables, as the size of the list must be known at compile time.

```
def make_pair(a: Int, b: Int): Pair = Pair(a, b)

def main() = {
    val p = make_pair(1, 2)
    print(p.b)
    0
}
```

The output for this program might not be `2` as expected. This is undefined behaviour, as the pair returned in `make_pair` is deallocated when it returns and falls out of scope.

A method to bypass this limitation is to allocate stack memory for the variable in the parent scope, and add a parameter passing the pointer to the child scope. However, adding a parameter for every variable that needs to persist beyond the scope in which it was defined can quickly become cumbersome, and can lead to code that is difficult to read and maintain.

Adding support for heap-allocated variables would require a refactor of the code generation phase. The phase would need to be modified to allocate and deallocate memory on the heap, ensuring that memory is not leaked or accessed after it has been deallocated (dangling pointers). Without providing explicit functions for memory management like `malloc` and `free` to the end-user, this would need be implemented in the form of a garbage collector.

Chapter 7

Legal, Social, Ethical and Professional Issues

This chapter discusses the implications of the British Computing Society (BCS) Code of Conduct on the project, as well as other legal, social, ethical and professional considerations that may affect the project. While a project on a compiler may not necessarily have as many legal and ethical considerations in comparison to other projects, it is still important to consider these issues in order to ensure that the project is conducted professionally.

7.1 The BCS Code of Conduct

The BCS define a code of conduct that 'set rules and professional standards to direct the behaviour of its members in professional matters.' [18]. The code of conduct highlights four key principles that must be adhered to:

1. **Public Interest:** Due regard must be given to the safety and interests of others. The relevance of this principle to the development of a compiler is limited, as the effect of a compiler on the public is minimal.

   Theoretically, if the compiler were to be distributed as a binary, a malicious actor could inject malicious code into the compiler, infecting all programs compiled with it. A similar vulnerability is a compiler backdoor, where bootstrapped compilers (compilers compiled with themselves) could contain a malicious payload that replicates itself in all future versions of the compiler. [19]

However, the source code of this project is open-source, and not bootstrapped (i.e. it is not compiled with itself, but with Scala), and is therefore not a realistic concern.

2. **Professional Competence and Integrity:** The work carried out must be done to the best of the one's ability, without misrepresenting one's skills or knowledge. Within the remit of this project, justification was provided for the decisions that were made, and where possible were built upon reputable sources.

3. **Duty to Relevant Authority:** Care must be taken to ensure that work completed under the Relevant Authority (in this case, King's College London) meets their requirements, and that any conflicts of interest are declared. Throughout the project, the Academic Regulations of the College, the requirements of module 6CCS3PRJ, as well as the Academic Honesty and Integrity Policy, were adhered to. No conflicts of interest were present.

4. **Duty to the Profession:** The reputation of the profession must be upheld as a personal duty, acting in a way that promotes the profession. This project is presented as a case study for the implementation of functional language constructs in LLVM IR, where others are able to learn and improve upon the work done here.

7.2 Other implications

The software licence chosen for a compiler is important, as it determines how the software can be used, modified and distributed. Not only does the licence affect the software itself, but it may also potentially effect the software that is compiled with it.

For instance, if the runtime library of a compiler is licensed under the 'GNU General Public License' (GPL), then any software compiled with the compiler must also be licensed under the GPL. This is because the library is linked with the compiled software, and the GPL requires that any software linked with GPL software must also be licensed under the GPL.

In fact, the GCC compiler *does* licence its runtime libraries under the GPL, which is why the authors of GCC include a GCC Runtime Library Exception, allowing the compilation of non-GPL software with certain GPL-licensed library files [20]. Similarly, the LLVM project is licenced under the Apache 2.0 licence 'with LLVM Exceptions'. This allows for certain licence conditions to be ignored when redistributing embedded LLVM code in a compiled binary [21].

An appropriate licence for this project is the MIT Licence, which allows for the software to be used, modified and distributed freely, as long as the original licence and copyright notice is

included. This licence is permissive, and allows for the software to be used in both open-source and proprietary projects.

Chapter 8

Conclusion

Targeting the LLVM as a backend for a functional language compiler has proven to be an effective method for generating efficient machine code. The LLVM IR provides a low-level representation of programs that can be compiled to a variety of architectures, without the need for the compiler to target each architecture individually. Despite the complex transformations required to compile functional language paradigms into imperative machine code, the SSA form of LLVM IR provides a suitable target for this transformation.

Performance of the generated code was on par with that of the Clang compiler, with the LLVM toolchain providing a number of optimisations that could be applied to the generated IR, such as dead code elimination, loop unrolling, and inlining. These optimisations came at no additional cost to the complexity of the compiler, with the LLVM toolchain providing these features out of the box. Any future improvements to the LLVM optimiser would be inherited by the compiler, providing a significant advantage over other compilation methods. Key features of functional languages, such as recursion, can be efficiently compiled to LLVM IR, with the LLVM backend providing the tail call optimisation necessary to prevent stack overflows.

The LLVM IR provides a portable target for functional language compilers, with the ability to target a variety of architectures and operating systems. The LLVM project is actively maintained and developed, with a large community of contributors and users. This ensures that the LLVM IR will remain a viable target for compilers in the future, with new features and optimisations being added regularly.

Areas of improvement for this compiler lie in an improved type system, with the ability to handle more complex types such as algebraic data types and type classes. As discussed in Chapter 6, the current type system is limited in its expressiveness. Improvements to the type

system would allow for more complex programs to be compiled, with the ability to handle more advanced features of functional languages.

Bibliography

[1] P. Hudak, "Conception, Evolution, and Application of Functional Programming Languages," *ACM Comput. Surv.*, pp. 359–411, Sep. 1989. DOI: 10.1145/72551.72554.

[2] A. Church, "An unsolvable problem of elementary number theory," *American Journal of Mathematics*, vol. 58, no. 2, pp. 345–363, 1936. DOI: 10.2307/2371045.

[3] P. J. Landin, "Correspondence between algol 60 and church's lambda-notation: Part i," *Commun. ACM*, vol. 8, no. 2, pp. 89–101, Feb. 1965. DOI: 10.1145/363744.363749.

[4] D. Grune, K. van Reeuwijk, H. E. Bal, C. J. H. Jacobs and K. Langendoen, *Modern Compiler Design*. New York, NY: Springer New York, 2012, ISBN: 978-1-4614-4699-6. DOI: 10.1007/978-1-4614-4699-6.

[5] *The Glasgow Haskell Compiler Wiki - Compiler*. [Online]. Available: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/hsc-main (visited on 03/12/2023).

[6] S. Peyton Jones and A. Santos, "A transformation-based optimiser for Haskell," *Science of Computer Programming*, vol. 32, no. 1, Oct. 1997. [Online]. Available: https://www.microsoft.com/en-us/research/publication/a-transformation-based-optimiser-for-haskell/.

[7] A. V. Aho, M. S. Lam, R. Sethi and J. Ullman, *Compilers: principles, techniques, & tools*, 2nd ed. Pearson/Addison Wesley, 2007, ISBN: 9780321486813.

[8] P. Eaton. "Parser generators vs. handwritten parsers: surveying major language implementations in 2021." (21st Aug. 2021), [Online]. Available: https://notes.eatonphil.com/parser-generators-vs-handwritten-parsers-survey-2021.html (visited on 06/12/2023).

[9] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on*

Code Generation and Optimization (CGO'04), Mar. 2004. [Online]. Available: https://llvm.org/pubs/2004-01-30-CGO-LLVM.pdf.

[10] R. A. Frost, R. Hafiz and P. Callaghan, "Parser combinators for ambiguous left-recursive grammars," in *Practical Aspects of Declarative Languages*, P. Hudak and D. S. Warren, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 167–181, ISBN: 978-3-540-77442-6. DOI: 10.1145/1149982.1149988.

[11] G. Morrisett, D. Walker, K. Crary and N. Glew, "From System F to Typed Assembly Language," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 3, pp. 527–568, May 1999. DOI: 10.1145/319301.319345.

[12] C. Flanagan, A. Sabry, B. F. Duba and M. Felleisen, "The Essence of Compiling with Continuations," *SIGPLAN Not.*, vol. 28, no. 6, pp. 237–247, Jun. 1993. DOI: 10.1145/173262.155113.

[13] R. A. Kelsey, "A Correspondence between Continuation Passing Style and Static Single Assignment Form," *SIGPLAN Not.*, vol. 30, no. 3, pp. 13–22, Mar. 1995. DOI: 10.1145/202530.202532.

[14] *The Glasgow Haskell Compiler Wiki - CPS*. [Online]. Available: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/cps (visited on 04/04/2024).

[15] *The Glasgow Haskell Compiler Wiki - Generated Code*. [Online]. Available: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/generated-code (visited on 03/12/2023).

[16] L. Maurer, P. Downen, Z. M. Ariola and S. Peyton Jones, "Compiling without continuations," ser. PLDI 2017, Association for Computing Machinery, 2017-06-14, pp. 482–494. DOI: 10.1145/3062341.3062380.

[17] J. Jacobs, "How to compile pattern matching," 2nd May 2021. [Online]. Available: https://julesjacobs.com/notes/patternmatching/patternmatching.pdf (visited on 22/03/2024).

[18] BCS The Chartered Institute for IT, "Code of Conduct for BCS Members," 8th Jun. 2022. [Online]. Available: https://www.bcs.org/media/2211/bcs-code-of-conduct.pdf (visited on 22/03/2024).

[19] K. Thompson, "Reflections on trusting trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984. DOI: 10.1145/358198.358210.

[20] Free Software Foundation. “GCC Runtime Library Exception.” (31st Mar. 2009), [Online]. Available: https://www.gnu.org/licenses/gcc-exception-3.1.en.html (visited on 22/03/2024).

[21] LLVM Project. “LLVM Developer Policy.” (7th Aug. 2017), [Online]. Available: https://llvm.org/docs/DeveloperPolicy.html#open-source-licensing-terms (visited on 22/03/2024).