An LLVM Backend of the COGENT Compiler

by
Zhanlin Shang

Supervisor: Christine Rizkallah

University of New South Wales
May, 2020

ABSTRACT

COGENT is a purely functional programming language for implementing easily verified system code with a certifying compiler developed in Haskell. The current COGENT compiler generates C code and hence many optimizations rely on the C compiler. The task of this project is to implement a compiler backend of COGENT that produces LLVM Intermediate Representation (LLVM IR) instead of C in order to enable low level optimizations on the generated code, as well as to explore the possibility of using Vellvm, a Coq formalization of a core subset of the LLVM IR, to avoid the limitations in the C-based proof tools.

TABLE OF CONTENTS

1 Introduction

COGENT [9] is a purely functional programming language that is being compiled to C, a higher order logic embedding and a proof relating the two in the Isabelle/HOL theorem prover. COGENT can assist programmers in writing system programs that are easy to verify[1].

Since COGENT is a system programming language, it is necessary for it to be as efficient as possible. The current approach involves some C verification tools which are limited to a subset of C. This limits the efficiency of the C code that can be produced by the COGENT compiler.

LLVM [6] is a high performance modern (compared to GCC) compiler framework that has been developed over 17 years. Many low level optimizations can be done by LLVM, such as loop unrolling, partial redundancy elimination and tail call optimisation. As LLVM was designed to be a compiler framework, it is easy to add or change the passes of the compilation process.

The goal of this project is to implement a compiler backend of COGENT that targets the LLVM Intermediate Representation (LLVM IR) to avoid the limitations caused by the C verification tools.

To preserve the verifiability of COGENT, we plan to use a verification framework that targets LLVM called Vellvm [15] in the future. Since Vellvm has only formalized a core subset of the LLVM IR, our backend only uses the subset of LLVM IR that is formalized by Vellvm.

This report will introduce COGENT and LLVM first and then demonstrate the implementation of an LLVM backend of a core subset of COGENT.

2 Background

2.1 Cogent

COGENT [9] is designed to be a high level, purely functional language for implementing systems software. The COGENT compiler guarantees the memory safety and totality of COGENT programs.

There are several phases in the compilation of a COGENT program: parsing, type checking, desugaring, normalization, monomorphisation, and then code generation to the target language (C code in the existing compiler and LLVM IR in this project).

At the code generation phase, there is a small set of primitive types. The primitive types that are covered by this project are:

Booleans	:	`Bool`
Unsigned integers	:	`U8, U16, U32, U64`
Strings	:	`String`
Pairs	:	`(A, B)`
Records	:	`{x :A , y :  b}`

primops	o	$\in$	$\{$`+,*,/,>,<,>=,<=,==,!=`$\}$
literals	l	$\in$	$\{$`123,True,'a',...`$\}$
expressions	e	::=	$l \mid x \mid o\,(\bar{e})$
		$\mid$	**let** $x = e_1$ **in** e_2
		$\mid$	**let!** $(\bar{y})\,x = e_1$ **in** e_2
		$\mid$	**if** e_1 **then** e_2 **else** e_3
		$\mid$	**promote** $\langle \overline{\mathsf{C}\ \tau} \rangle\, e$
		$\mid$	$\{\overline{\mathrm{f}\ = e}\} \mid e.\mathrm{f}$
		$\mid$	**take** $x\ \{\mathrm{f} = y\} = e_1$ **in** e_2
		$\mid$	**put** $e_1.f := e_2$
function def.	d	::=	$\langle f :: \pi, f\ x = e \rangle \mid \langle f :: \pi, \blacksquare \rangle$
programs	P	::=	$\overline{d}$
function names		∋	f, g
variable names		∋	x, y
constructors		∋	$\mathsf{A}, \mathsf{B}, \mathsf{C}$
record fields		∋	f, g

Figure 1: The Syntax of COGENT Core Language Subset Covered

primops	o	$\in$	$\{$`||,<<,...`$\}$
expressions	e	::=	$() \mid f\,[\overline{\tau}]$
		$\mid$	**cast** $t\ e$
		$\mid$	**case** e_1 **of** $\mathsf{C}x \to e_2$ **else** $y \to e_3$
		$\mid$	**esac** $e \mid \mathsf{C}\ e$

Figure 2: The Syntax of COGENT Core Language Subset Not Covered

The syntax of COGENT [9] covered by this project is shown in Figure 1:

For primitive operations, they are defined on `int`s.

For literal values, types `int`, `boolean` and `string` are defined.

For expressions, literal value, variable, binary operations, let binding (and let bang), if branching, integer type promoting, record definition and record field accessing are implemented.

For functions, both function definitions and abstract function declarations are supported. Abstract function declaration will be treated as a `extern` declaration.

Some parts of the core syntax is left for future work due to the time limitations, as shown in Figure 2.

2.2 LLVM

LLVM [6] is a modern compiler framework that provides a language independent code representation (LLVM IR) which can be used for multiple purposes, including analysis and transformation. There are many analyses and transformation passes included by the LLVM framework, such as printing function call graphs, printing control flow graphs, dead code elimination, function inlining and loop unrolling.

In addition to assembly, LLVM includes a simple type system which is primitive in mainstream languages, such as integers of arbitrary length, floating point numbers, structures, arrays, functions and pointers, and operations on these types. LLVM also hides machine-specific details such as physical registers and calling conventions [6].

LLVM provides an infinite set of virtual registers in the Static Single Assignment (SSA) form [6], i.e. each virtual register can be written by only one instruction and an assignment of a new value to a variable will be represented as initialization of a new variable based on the old value, the instructions used in this report are explained in Figure 3. For example, the following C code:

```
1 int x =1;
2 x = 2;
3 int y = x;
```

can be represented in LLVM IR as:

```
1 %x.1 = alloca i32 ; allocate a 32 bits int and assign the address to %x.1
2 %x.2 = alloca i32 ; allocate a 32 bits int and assign the address to %x.2
```

```
%y = alloca i32 ; allocate a 32 bits int and assign the address to %y
store i32 1, i32* %x.1 ; store a 32 bits int value 1 to address %x.1
store i32 2, i32* %x.2 ; store a 32 bits int value 2 to address %x.2
%tmp = load i32, i32* %x.2 ; load the value from address %x.2 into %tmp
store i32 %tmp, i32* %y ; store the value of %tmp to address %y
```

- `alloca` *type*:
  Allocate memory for *type*
- `store` *type value*, *type * place*:
  Put *value* into the address pointed by *place*
- `load` *type*, *type * place*:
  Load value of type *type* from the address pointed by *place*
- `ret` *type operand*:
  Return the value of *operand* from the current block, operand can be either literal value or register
- `getelementptr` *type*, *type * place*, `i32` *idx* {, `i32` *idx*}*:
  Get from the aggregate type pointed by *place* the pointer to the element determined by *idx*s, for example, given `int * a`, `a[1]` will be `getelementptr i32, i32 * %a, i32 1`
- `insertvalue` *agg_type agg*, *elm_type elm*, *idx*:
  Insert *elm* into the *idx* place of the aggregate type value *agg*
- `call` *ret_type name* (*arg_type arg*, ...):
  Call the function *name* with argument *arg* and get the returned value as type *ret_type*

Figure 3: Subset of LLVM Instructions covered in this report

SSA can be seen as equivalent to a well-behaved subset of Continuation-Passing Style (CPS) [5]. In CPS [12], a procedure does not return, instead it is passed an additional "call back" argument, which is applied to the procedure's return value. There are many studies on optimising CPS and the optimisations for CPS can all be applied for SSA.

In LLVM IR, the minimal unit is an **Instruction**. LLVM uses RISC-like instructions [6], plus higher level information such as types. An instruction has the form:

```
instruction-name arg_1, arg_2, ... , arg_n
```

where `args` can be one of the followings:

- operand: can be literal or register

  e.g. in `add i32 1 %x`

  both `1` and `%x` are operands

- type: can be type or type name

  e.g. in `load i32, i32* %x`

  `i32` and `i32*` are types

- attribute: different instructions have different attributes

  e.g. in `alloca i32, align 4`

  and `add nuw nsw i32 1, 2`

  `align 4`, `nuw` and `nsw` are all attributes

The LLVM instruction set consists of several kinds of instructions: terminator instructions, binary instructions, bitwise binary instructions, memory instructions and other instructions [7]. Most of LLVM instructions are in three-address form: they take one or two operands and produce a single result [6].

Instructions are contained by **Blocks**, which form the body of a program, e.g. function body, conditional branches, etc. A block contains one and only one terminator. Here is a sample program in LLVM IR:

```
1  define i8 @foo({ i8, i8 }* %0) {
2  entry: ; this is a label, LLVM has first class labels that can be used for jumping
3    %1 = getelementptr inbounds { i8, i8 }, { i8, i8 }* %0, i32 0, i32 0
4    ; get pointer to the first field of %0
5    %2 = load i8, i8* %1 , align 1
6    ; load the value from %1 to %2
7    %3 = getelementptr inbounds { i8, i8 }, { i8, i8 }* %0, i32 0, i32 1
8    ; get pointer to the second field of %0
9    %4 = load i8, i8* %3 , align 1
10   ; load the value from %3 to %4
11   %5 = add i8 %2, %4
```

```
12    ; add %2 and %4, store result in %5
13    ret i8 %5
14    ; return %5
15  }
16
17  define i8 @main() {
18    %1 = alloca { i8, i8 }, align 1
19    ; allocate memory for a struct { i8, i8 }
20    %2 = insertvalue { i8, i8 } undef, i8 1, 0
21    ; insert the value 1 to the first field of { i8, i8 } (starts from undef)
22    ; and store the struct in %2
23    ; %2 is { 1, undef }
24    %3 = insertvalue { i8, i8 } %2, i8 2, 1
25    ; insert the value 2 to the second field of %2
26    ; %3 is { 1, 2 }
27    store { i8, i8 } %3, { i8, i8 }* %1
28    ; store the value of %3 to address %1
29    %4 = call i8 @foo({ i8, i8 }* %1)
30    ; call function @foo with %1 as argument and store the returned value in %4
31    ret i8 %4
32    ; return %4
33  }
```

This sample program defines two functions, one named `foo` and the other named `main`. Similar to C, an LLVM program starts with the function `main`.

In this `main` function, firstly we allocate memory for a structure that contains two 8-bit integers, then we assign the two fields in the allocated structure to `1` and `2` respectively, next we call the function `foo` with the allocated structure as argument and return the result from `main`.

In this `foo` function, firstly we take values from both fields in the structure given as argument, then we compute the sum of them and return the result from `foo`.

3 Implementation

The LLVM backend of COGENT compiler is a drop-in replacement of the existing C code generation pass. The following two components are of the most importance in the code generation process:

- `data Type t`
- `data Expr t v a e`

They correspond to COGENT's types and expressions respectively and the mapping between them to LLVM IR will be explained in the following sections.

In this project, we use a LLVM's Haskell binding named "llvm-hs" [2], which is an one-to-one binding from the LLVM IR to Haskell, with abstraction for generating conditional branching.

In the LLVM backend, two data types are responsible for generating instructions and blocks:

- `data BlockState`
- `data CodegenState`

`BlockState` is used to track the states of a basic block, such as instructions in a block and the terminator of a block. `CodegenState` is used to track the general states of the compilation, such as blocks contained in this program, the block we are currently working with, variables indexing, virtual registers usages, etc.

The following utility functions for the code generation will be used in the later sections:

- `fresh :: Codegen Word`

  This generates a new virtual register with a fresh name.

- `instr :: Type → Instruction → Codegen Operand`

  This function takes an instruction and stores the result into a newly assigned register by `fresh`, then returns the register.

- `addBlock :: ShortByteString → Codegen Name`

  This function takes a name and creates a new block in the current `CodegenState`.

- `setBlock :: Name → Codegen Name`

  This function takes a name and sets the block bound to the name as the working block. This is essential when generating codes for branching.

3.1 Mapping Cogent Core Language to LLVM

In the typechecked core language, expressions are of type `TypedExpr t v a` which states the type of the expression. Therefore, mapping the COGENT types to LLVM types should occur before compiling the COGENT expressions.

3.1.1 Cogent Types to LLVM Types

The function `toLLVMType` maps the following COGENT types to LLVM types:

• `Unit`:

Since the type `Unit` is similar to `void` in C, we generate `VoidType` of LLVM for the unit type in COGENT.

• `PrimInt`:

In COGENT, there are 5 primitive unsigned integer types: `Bool`, `U8`, `U16`, `U32`, `U64`. Since LLVM doesn't differentiate signed and unsigned value at the type level and arbitrary length integer value are allowed, denoted in LLVM as `i`n where n can range from 1 to $2^{23} - 1$, we can generate integer types with corresponding length in LLVM.

In `llvm-hs`, the constructor `IntegerType` takes an integer n and produces the corresponding LLVM integer type `i`n , we have `IntegerType 1` to `IntegerType 64` for the COGENT primitive integer types respectively.

• `Record and Pair`:

In LLVM, a structure type is defined as:

```
1 type { type_1, type_2, ..., type_n }
```

In `llvm-hs` the constructor `StructureType` takes a list of LLVM types and produces the required type.

Since a COGENT record is a structure and a product is essentially a record without names, therefore, for a record R with field types $f_1 \dots f_n$, we can have:

$$\texttt{toLLVMType}\ R = \texttt{StructureType}\{\texttt{toLLVMType}\ f_i \mid i \in [1, n]\}$$

• `String`:

Since strings are char pointers (i.e. `char *`), they are mapped to `PointerType{IntegerType 8}`.

3.1.2 Cogent Expressions to LLVM IR

COGENT expressions are compiled by the function:

```
exprToLLVM ::  TypedExpr t v a → Codegen (Either Operand (Named Terminator))
```

The `Either` in the type signature determines whether the returned value is a terminator or not. When a value needs to be returned (e.g. the last expression in a let block), a terminator `Ret` will be constructed from the value and then returned as `Right`, otherwise we just return the value as `Left`.

The following expressions are currently covered:

• `ILit` *val bits*:

Since arbitrary length integer values in the LLVM IR are primitive, for a COGENT integer of length *bit* and value *val*, we generate:

$$\texttt{ConstantOperand Int}\ \{bits, \ val\}$$

• `Op` *op* `[`$operand_1$`,` $operand_2$`]`:

First we get both operands by recursively generating instructions from $operand_1$ and $operand_2$ and we have to ensure that both operands produce terminator values, since it doesn't make sense to have code: `1 + (return 2)` in C.

Then we generate an instruction according to *op* and construct the instruction through `instr` to ensure that the returned value can be used as an operand by its upper level.

• `Variable` *idx*, *name*:

In core COGENT, de Brujin indices are used to denote the variables used, but LLVM uses an incremental indexing that starts from 0 in each function. For example, the COGENT program:

```
foo : (U8, U8) -> U8
foo (a, b) = a + b
```

will be A-normalised into (with irrelevant parts removed):

```
foo = take (a, x_1) = x_0<_v0> { .0 }
  in take (b, x_2) = x_1<_v1> { .1 }
     in a<_v2> + b<_v0>
```

The equivalent LLVM IR is as follows:

```
define i8 @foo({ i8, i8 }* %0) {
entry:
  %1 = getelementptr inbounds { i8, i8 }, { i8, i8 }* %0, i32 0, i32 0
  %2 = load i8, i8* %1, align 1   ; a
  %3 = getelementptr inbounds { i8, i8 }, { i8, i8 }* %0, i32 0, i32 1
  %4 = load i8, i8* %3, align 1   ; b
  %5 = add nuw i8 %2, %4
  ret i8 %5
}
```

To resolve this problem, list named `indexing` in `CodegenState` is introduced. Whenever a new value is introduced (e.g. entering a new scope), the new value will be added to the front of `indexing`, then the order will correspond to de Brujin index.

• `Struct` [*fields*]:

In generating instructions for a structure, memory is firstly allocated for the struct, then values of each field in the struct is set.

Therefore we have the following algorithm (pseudo code) to generate structures:

```
struct ← Alloca { type = recordType }
flds ← [ GetElementPtr from struct then Store exprToLLVM(f) | f ∈ fields]
```

```
return fold (>>) struct flds
-- >> ensures the sequencing of the generation of each fields in the struct
```

• `Let` *name val body* / `Let!` *var name val body*:

The intuitive translation from a `let` is (in de Brujin's notation):

$$(\lambda.body)(val)$$

As literally translation of anonymous functions to LLVM is not efficient, we instead use the `indexing` method as introduced in the `Var` section above.

• `Take` (*name*$_a$`,` *name*$_b$`)` *record field body*:

This expression is similar to `let`. At the stage of code generation, the linearity is already checked in previous passes, we can take this equivalence as an intuition:

$$\texttt{take}\ (a, b)\ r\ f\ body \Leftrightarrow \texttt{let}\ a = r.f\ \texttt{in let}\ b = r\ \texttt{in}\ body$$

Hence, following the same idea as generating `let`, we load the field value into a fresh register with the `load` instruction, and push the value and the record to the front of `indexing`. Then, with the extended `indexing`, we generate instructions for *body*.

• `Put` *record field val*:

As opposed to `take`, `put` stores the value into a structure at a certain field. We need to generate the pointer to the field and construct the `store` instruction of LLVM to store the value to the address of the target field.

• `If` *cond then else*:

In SSA, conditional branching is achieved with the help of the Φ function [10] . The idea of the Φ function can be demonstrated with the following program:

```
x = 5
x = x - 3
if x > 3:
  y = 1
```

```
else:
  y = 2
w = y + x
```

A new definition of `y` will be generated and initialized to "choose" the correct branch:

```
x_1 = 5
x_2 = x_1 - 3
if x_2 > 3:
    y_1 = 1
else:
    y_2 = 2
 y_3 = Φ(y_1, y_2)
w = y_3 + x_2
```

As mentioned above, `llvm-hs` provides an abstraction for conditional branching, a special terminator named `CondBr`. Therefore we can write:

```
currentBlk <- gets currentBlock
blkTrue <- addBlock "brTrue"
blkFalse <- addBlock "brFalse"
setBlock blkTrue
exprToLLVM tb
setBlock blkFalse
exprToLLVM fb
setBlock currentBlk
CondBr { condition = cond
       , trueDest = blkTrue
       , falseDest = blkFalse
       }
```

This produces the following LLVM program:

```
entry:
%1 = ;condition
br i1 %1, label %brTrue, label %brFalse
brTrue:  ; preds = %entry
...
brFalse: ; preds = %entry
...
```

3.2 Interfacing with C

With the support of the LLVM infrastructure, inter-operation between COGENT and C can be simple:

- To use COGENT function in C, the COGENT function need to be marked as `extern`
- To use C function in COGENT, an abstract function declaration will be compiled into external linkage by default.

Then by compiling and assembling each components to LLVM bitcodes and linking them using 'llvm-link', we will have an executable as desired. An example is included in Appendix A.

4 Future Work

There is still a considerable number of features not implemented yet in this backend and the most important features still lacking are:

- Sum type (i.e. tagged union)
- Case / Esac (`case` and `esac` takes an value of sum type and branches for each case)

Due to the time limitations, these have been left for future development.

Our proposed approach on them is: we can append a "tag" to the value of sum type, e.g. for a sum type $R =< E \mid S >$, we generate:

$$\%R = \{\texttt{i8}, \max(\texttt{sizeof}(E), \texttt{sizeof}(S))\}$$

In which the `i8` will be the tag and this should be used in C / C++ in the same way as:

```
1 enum tag : uint_8t { ... };
2 struct s {
3   tag t;
4   union {
5     E ...
```

```
6       S ...
7     } v;
8   };
```

Once we have the sum type implemented in this way, we can compile `case` and `esac` as `switch` statements in C.

5 Related Work

There are many functional programming languages that have LLVM backend, such as Haskell (GHC's LLVM backend) [14] and many Lisp dialects (e.g. Clasp [11], Open Dylan [4]). Modern languages with functional features such as Rust and Swift also target LLVM.

Because of historical reasons, the LLVM backend of GHC compiles the old intermediate language C minus minus (Cmm) to LLVM. Therefore, our work with the LLVM backend of COGENT compiler is more related to the Cmm generation.

Clasp [11] is a complier for Common Lisp which aims to enable the interfacing between Common Lisp and C++. Their implementations could act as a reference when extending our LLVM backend to support interfacing with C, especially on the handling polymorphic types.

Swift and Rust are both languages with functional features such as first class closure without the use of runtimes or virtual machines. Swift has an intermediate language, Swift Intermediate Language (SIL) [3], which provides native support for closure capture. Rust also adopts a similar design by having an intermediate language named *mid-level* IR (MIR) [8]. The closure expansion of Rust is done on the MIR level [13]. As we are planning to add the support of closures in COGENT, the implementations of SIL and MIR could be good references for the future development.

A common feature among these works are, they all have intermediate languages between their core languages and LLVM IR. We may also want to adopt this approach by having an IR before LLVM IR so that we can bridge up the abstraction gap between core COGENT and LLVM, for example, when we are adding closure supports to COGENT in the future, having a mid level IR could help to resolve the captures.

6 Conclusions

We have presented a working prototype of LLVM backend of the COGENT compiler. As there is no longer a reliance on the C compilers, we can produce outputs that better suit our needs from the same COGENT input, such as real 1 bit `boolean`, and produce smaller binary (see Appendix B). This even allows us to combine COGENT program with not only C, but any language that is capable of being compiled into LLVM bitcode.

A Calling Cogent from C

A simple example for calling COGENT from C:

```c
#include <stdlib.h>

typedef struct {
  char a;
  char b;
} arg;

extern char foo( arg * );


int main(){
  arg * a = malloc(sizeof(arg));
  a->a = 1;
  a->b = 2;
  char res = foo(a);
  free(a);
  return res;
}
```

Figure 4: main.c

```
foo : (U8, U8) -> U8
foo (a,b) = a + b
```

Figure 5: adder.cogent

```
CLANG=clang
AS=llvm-as
LINK=llvm-link
COGENT=cogent

all: main.bc adder.bc
    $(LINK) main.bc adder.bc -o main
    chmod +x main

main.bc: main.ll
    $(AS) main.ll -o main.bc

main.ll: main.c
    $(CLANG) -S -emit-llvm main.c -o main.ll

adder.bc: adder.ll
    $(AS) adder.ll -o adder.bc

adder.ll: adder.cogent
    $(COGENT) --llvm adder.cogent
```

Figure 6: Makefile

B Comparing Outputs from Different Backends

For the input file:

```
1 foo : (U8, U8) -> U8
2 foo (a,b) = a + b
```

Figure 7: simple-arith.cogent

different backends generate different outputs (some metadata were removed):

```
%struct.t1 = type { i8, i8 }

; Function Attrs: noinline nounwind optnone uwtable
define dso_local signext i8 @foo(i16 %0) #0 {
  %2 = alloca %struct.t1, align 1
  %3 = alloca i8, align 1
  %4 = alloca i8, align 1
  %5 = alloca i8, align 1
  %6 = bitcast %struct.t1* %2 to i16*
  store i16 %0, i16* %6, align 1
  %7 = getelementptr inbounds %struct.t1, %struct.t1* %2, i32 0, i32 0
  %8 = load i8, i8* %7, align 1
  store i8 %8, i8* %3, align 1
  %9 = getelementptr inbounds %struct.t1, %struct.t1* %2, i32 0, i32 1
  %10 = load i8, i8* %9, align 1
  store i8 %10, i8* %4, align 1
  %11 = load i8, i8* %3, align 1
  %12 = sext i8 %11 to i32
  %13 = load i8, i8* %4, align 1
  %14 = sext i8 %13 to i32
  %15 = add nsw i32 %12, %14
  %16 = trunc i32 %15 to i8
  store i8 %16, i8* %5, align 1
  %17 = load i8, i8* %5, align 1
  ret i8 %17
}
```

Figure 8: simple-arith.ll by compiling the generated C code

```
define i8 @foo({ i8, i8 }* %0) {
entry:
  %1 = getelementptr inbounds { i8, i8 }, { i8, i8 }* %0, i32 0, i32 0
  %2 = load i8, i8* %1, align 1
  %3 = getelementptr inbounds { i8, i8 }, { i8, i8 }* %0, i32 0, i32 1
  %4 = load i8, i8* %3, align 1
  %5 = add nuw i8 %2, %4
  ret i8 %5
}
```

Figure 9: simple-arith.ll by using the LLVM backend

Bibliography

[1] Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., Tuong, J., Keller, G., Murray, T., Klein, G., and Heiser, G. Cogent: Verifying high-assurance file system implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, GA, USA, Apr. 2016), pp. 175–188.

[2] Cowley, A., Diehl, S., Kiefer, M., and Scarlet, B. S. *llvm-hs: General purpose LLVM bindings*, 2020 (accessed Apr. 2020). `https://hackage.haskell.org/package/llvm-hs`.

[3] Groff, J., and Lattner, C. *Swift Intermediate Language: A high level IR to complement LLVM* , 2015 (accessed Apr. 2020). `https://llvm.org/devmtg/2015-10/slides/GroffLattner-SILHighLevelIR.pdf`.

[4] Housel, P. S. LLVM Code Generation for Open Dylan. In *Proceedings of the 13th European Lisp Symposium on European Lisp Symposium* (2020), ELS2020, European Lisp Scientific Activities Association.

[5] Kelsey, R. A. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations* (New York, NY, USA, 1995), IR '95, Association for Computing Machinery, p. 13–22.

[6] Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, Mar 2004).

[7] LLVM Developers. *LLVM Langauge Reference Manual*, 2020 (accessed Apr. 2020). `https://llvm.org/docs/LangRef.html`.

[8] Matsakis, N. *Introducing MIR*, 2016 (accessed Apr. 2020). `https://blog.rust-lang.org/2016/04/19/MIR.html`.

[9] O'Connor, L., Chen, Z., Rizkallah, C., Amani, S., Lim, J., Murray, T., Nagashima, Y., Sewell, T., and Klein, G. Refinement through restraint: Bringing down the cost of verification. In *International Conference on Functional Programming* (Nara, Japan, Sept. 2016).

[10] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1988), POPL '88, Association for Computing Machinery, p. 12–27.

[11] Schafmeister, C. A., and Wood, A. Clasp common lisp implementation and optimization. In *Proceedings of the 11th European Lisp Symposium on European Lisp Symposium* (2018), ELS2018, European Lisp Scientific Activities Association.

[12] Sussam, G. J., and Steele Jr., G. L. SCHEME: An Interpreter For Extended Lambda Calculus. *AI Memo*, 349 (1975).

[13] Team, R. C. *Closure Expansion in Rustc*, (accessed May. 2020). `https://rustc-dev-guide.rust-lang.org/closure.html`.

[14] Terei, D. A., and Chakravarty, M. M. An llvm backend for ghc. In *Proceedings of the Third ACM Haskell Symposium on Haskell* (New York, NY, USA, 2010), Haskell '10, Association for Computing Machinery, p. 109–120.

[15] Zhao, J., Nagarakatte, S., Martin, M. M., and Zdancewic, S. Formalizing the llvm intermediate representation for verified program transformations. *SIGPLAN Not. 47*, 1 (Jan. 2012), 427–440.