CONSTRUCTOR UNIVERSITY

Compiling Lama to LLVM bytecode

by

Fedor Kudriavtsev

Bachelor Thesis in Computer Science

Submission: May 14, 2023 Supervisor: Daniil Berezun

Constructor University | School of Computer Science and Engineering

Statutory Declaration

Family Name, Given/First Name	Kudriavtsev, Fedor
Matriculation number	30006567
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

....

Date, Signature

Abstract

Constructor university, Bremen offers a compiler course that utilizes a programming language called Lama. Lama serves a dual purpose: it serves as the language in which students implement their compilers and also as the language that is being compiled. It is common practice to write a compiler in the same language it is designed for, as it showcases the language's capabilities. Lama is an easy-to-use, functional programming language, which makes it an excellent choice for the course. However, there are some challenges associated with Lama. Currently, only an X86 version of the compiler exists, limiting the usage of Lama on Mac systems to Docker. Additionally, there are issues such as a lack of debugging capabilities and uninformative error messages.

To address these problems, an attempt was made last year to develop an LLVM front-end, which is a compiler that takes Lama code as input and produces LLVM Intermediate Representation (IR). LLVM is a programming infrastructure for building compilers. By utilizing this tool, it would be possible to leverage the various optimizers and backends provided by LLVM, which can compile LLVM IR into machine code or other programming languages. Moreover, it would allow Lama to interact with other programming languages that have an LLVM front-end. However, the previous attempt faced a challenge where LLVM IR was generated directly from the Lama Abstract Syntax Tree (AST), which represents the second level of the compiling system. As a result, the optimizations performed during the compilation from the AST to the stack machine (representing the third level of the compiling system) were not reflected in the LLVM IR, resulting in significant differences in the generated machine code. Consequently, the project was abandoned.

The goal of my project is to develop an LLVM front-end that compiles from the Lama stack machine. This approach provides all the opportunities that were envisioned by the previous attempt while eliminating the aforementioned issues.

Contents

1 Introduction

This project is built using the Ocaml LLVM library. LLVM is a compiler infrastructure that comprises three types of blocks: front-end, middle-end, and back-end. Front-end blocks compile source code into LLVM intermediate representation. Middle-end blocks apply optimizations to the intermediate representation. Back-end blocks compile the optimized intermediate representation into machine code or another programming language. Blocks of different types can be combined as needed. Therefore, to add LLVM support for a particular language, one only needs to create an LLVM front-end for that language.

Typically, compilers consist of three main parts: lexer, parser, and code generation. Code generation, which can be further divided into implementation-dependent parts, is the largest component. The lexer and parser prepare the input for the final part. The lexer takes the input and breaks it down into tokens based on its predefined set. It will encounter an error only if the input contains a word that is not recognized as a valid token. The parser then takes the stream of tokens and creates an abstract syntax tree (AST), which represents the structure of the input code as a tree. Each node in the AST represents a semantic concept. At this level, more complex errors, such as incorrect token sequences, can be detected.

Lama is a small programming language developed by JetBrains specifically for a compiler course at Constructor University. In this project, we have implemented an LLVM front-end for Lama. This front-end provides Lama with enhanced portability, improves its execution speed through LLVM optimizers, enables integration with LLVM tools such as debuggers and profilers, and facilitates integration with other programming languages.

In most modern compilers, during the code generation step, the AST is first transformed into a linear, low-level, platform-independent code. Subsequently, this code is transformed into platform-dependent code. In Lama, the intermediate code is written using a stack machine, which means that the machine primarily uses a stack for storing and manipulating data. There are also register-based, memory-based, and hybrid-based machines that use registers, direct memory, and registers combined with a stack, respectively. The Lama stack machine includes the instructions listed in Table 1.

Where Value.designation is any of this

During project we compile Lama to LLVM intermediate representation. LLVM IR is platform-independent single static assignment language. It means that every variable exists and assigned only once in whole code. It makes it easy to optimize this code. Using IR representation in compiler makes it possible to use different back-end and make the whole project retargetable. The goals that i pursue:

- Make LLVM front-end from Lama stack-machine.
- Cover it with tests
- Add runtime support

Instruction	Content	Description
BINOP	string	binary operator
CONST	int	put a constant on the stack
STRING	string	put a string on the stack
SEXP	string * int	create an S-expression
LD	Value.designation	load a variable to the stack
LDA	Value.designation	load a variable address to the stack
ST	Value.designation	store a value into a variable
STI		store a value into a reference
STA		store a value into array/sexp/string
ELEM		takes an element of array/string/sexp
LABEL	string	a label
FLABEL	string	a forwarded label
SLABEL	string	a scope label
JMP	string	unconditional jump
CJMP	string * string	conditional jump
BEGIN	string * int * int * Value.designation list * string list * scope list	begins procedure definition
END		end procedure definition
CLOSURE	string * Value.designation list	create a closure
PROTO	string * string	proto closure
PPROTO	string * string	proto closure to a possible constant
PCALLC	int * bool	proto call
CALLC	int * bool	calls a closure
CALL	string * int * bool	calls a function/procedure
RET		returns from a function
DROP		drops the top element off
DUP		duplicates the top element
SWAP		swaps two top elements
TAG	string * int	checks the tag and arity of S-expression
ARRAY	int	checks the tag and size of array
PATT	patt	checks various patterns
FAIL	Loc.t * bool	match failure (location, leave a value)
EXTERN	string	external definition
PUBLIC	string	public definition
IMPORT	string	import clause
LINE	int	line info

Table 1: All Lama stack machine instructions

Instruction	Content	Description
Global	string	
Local	int	
Arg	int	
Access	int	
Fun	string	

Table 2: All Lama stack machine instructions

2 Statement and Motivation of Research

Lama is a small programming language that was made by JetBrains specifically for a compiler course held at Constructor University. The main problem with this language is its lack of portability. The only available compiler for Lama is the x86 compiler, which compiles from the Lama stack machine. Consequently, Mac users are limited to using Docker, which is quite inconvenient. Additionally, Lama lacks support for debuggers/profilers and has uninformative error messages. All of these issues could be resolved by creating an LLVM front-end for Lama. That is why an attempt was made to develop it a year ago. However, the problem with that attempt was that it tried to generate LLVM IR directly from the AST tree, resulting in machine code that differed significantly from the machine code generated by the x86 compiler. This was due to a considerable number of optimizations made during the transformation of the AST tree to the Lama stack machine. As a result, the project was abandoned. Now, the idea is to create an LLVM front-end from the Lama stack machine, which would preserve the optimizations and maintain code consistency. However, this approach also poses challenges since the stack machine is a lower-level language compared to LLVM IR, and some instructions in the stack machine do not have a direct representation in LLVM IR. Furthermore, the entire philosophy behind the languages differs: the stack machine utilizes a stack to operate on memory, whereas LLVM IR adheres to the SSA ideology. This raises the question: How different will the machine code generated by LLVM from the stack machine be compared to the machine code produced by the x86 compiler?

3 Description of the Investigation

Firstly, let's assume that all arguments in all functions are of type int32, and all return values also have the type int32. With these assumptions, how can we create LLVM IR?

Since we are compiling from a stack machine, we need to emulate the stack in some way. To accomplish this, we have created a class called Function_c that handles all stack operations and labels. Labels are also an essential part of managing the stack, as they contain information about the stack depth.

Listing 1: label example

```
BEGIN ("Lf", 1, 0, [], ["x"], [{ blab="L4"; elab="L5"; names=[];
subs=[...]; }])
    SLABEL ("L4")
    SLABEL ("L7")
    CONST (1)
    CJMP ("z", "L10")
    SLABEL ("L11")
    LINE (1)
    LD (Arg (0))
    CONST (1)
    BINOP ("+")
    SLABEL ("L12")
    JMP ("L6")
    LABEL ("L10")
    SLABEL ("L15")
    CONST (3)
    SLABEL ("L16")
    JMP ("L6")
    SLABEL ("L8")
    LABEL ("L6")
    SLABEL ("L5")
END
```

Here, you can find a list of stack machine instructions that are used to compile this simple function in Lama:

Listing 2: lama function

```
fun f (x) { if 1 then x + 1 else 3 fi }
```

If we remove all code except for jumps and labels, we will observe the following:

Listing 3: label example

```
BEGIN (...)
    ...
    CJMP ("z", "L10")
    ...
    JMP ("L6") [1]
    LABEL ("L10")
    ...
    JMP ("L6") [2]
    SLABEL ("L8")
    LABEL ("L6")
    ...
END
```

This is how an if statement is compiled. As you can see, there are two unconditional jumps and one conditional jump. After an unconditional jump, the only valid possibilities are LABEL or SLABEL. Anything else would be considered dead code that will never be executed. Labels should contain information about the stack depth, which we load if the label follows a jump. Furthermore, there are two ways to reach label l6: through jump [1] and jump [2]. However, this is not a problem because the stack machine adheres to an invariant: regardless of how we arrive at a code block, the stack depth remains the same.

In LLVM, there are three types of variables: regular variables, pointers to allocated memory on the stack, and pointers to allocated memory on the heap. Using pointers to the heap doesn't make sense as it would slow down the program. Therefore, the only option is to use pointers to the stack, where we will store all the values. The LLVM library in OCaml allows us to insert code at any point. Whenever an instruction requires loading something onto the stack and there are no available pointers, the program simply allocates one more pointer at the beginning of the LLVM function.

The Function_c class provides these functions:

Listing 4: label example

```
method get_name // return name of the function
method get_function // returns llvm function
method set_depth (n:int)
method get_free_ptr // returns ptr[depth + 1], depth increases
method get_taken_ptr // returns ptr[depth], depth decreases
method get_front_ptr // returns ptr[depth], depth stays the same
method drop
method load (value : Llvm.llvalue)
method load_from_ptr (value : Llvm.llvalue)
method store
method dup // copies ptr[depth] to ptr[depth+1]
method swap // swaps
method get_depth
method update_label_depth (name:string) (depth: int)
```

Set of instructions that can be implemented by just using function_c class:

BINOP	CONST	STRING
LD	FLABEL	SLABEL
JMP	CJMP	END
RET	DROP	DUP
SWAP		

One of the challenges in emulating the stack of a Stack Machine (SM) with i32 pointers to the stack is that it can store not only i32 types. However, this can be addressed by casting all types that occupy less than 32 bits to i32, and for types that occupy more than 32 bits, obtaining i8 pointers to the stack memory and then casting the i8 pointer to i32. Nevertheless, this approach would prevent us from determining the specific type stored in the stack. The question arises: Do we need to know the types of variables stored in the stack?

Here are the instructions that utilize values from the stack:

BINOP	SEXP	ST
STI	STA	ELEM
CJMP	CLOSURE	PROTO
PPROTO	PCALLC	CALLC
CALL	ARRAY	PATT
TAG		

BINOP: During the execution of this instruction, we can assume that the last two values on the stack are integers.

SEXP: We cannot assume the type of values on the stack, but it is not necessary to create an S-expression.

ST, STA: We cannot assume the type of the value, but we can store it as an i32 and then cast it to its actual type.

ELEM: We can assume that the last variable on the stack is an array, string or sexp. We can store all of them the same way and do not even distinguish them. In can be achieved by storing

sexp like [i8 pointer to string, i32 size of sexp, i8*, i8*, ...],
array like [i8 pointer to string ”array”, i32 size of array, i8*, i8*, ...],
string like [i8 pointer to string ”string”, i32 size of array, i8*, i8*, ...],

If all types in the array/sexp/string are stored in the same format (e.g., as i8 pointers), we do not depend on their specific type and can also store it as an i32 or i8 pointer.

CJMP: We can assume that the last value on the stack is an integer.

CLOSURE: We cannot assume any specific types on the stack, which creates a limitation that all functions should only receive i32 values.

PCALLC, CALLC : with the limitation that all functions get ints, all values from stack are casting to i32.

CALL : This function is the most difficult. Because if for all inner functions we can assume that they are getting i32, we have outer functions from stdlib in c language. And their arguments should be declared in llvm the same way it is declared in object file stdlib.o. And this is hard because EXTERN instruction only gives us a name of a function but not its arguments. So we have 2 solutions. Either to somehow store values so we will be able to identify its type. For example make every type an array of 2 values: int bound to a type and pointer to real type.

MAKE PICTURE.

Because of this we would have to store all values in heap, because we would not be able to convey pointers to local stack between functions. And it would make all operations much more longer.

Another option is to save all types at the same time with stack. So To handle this, we would need to maintain another data structure that keeps track of all the types present on the stack at any given moment during compilation. Because of this i had to rewrite store and load methods. New functions signatures are

Listing 5: function signatures

```
    method store : Llvm.llvalue * variable
    method load : Llvm.llvalue -> variable -> Llvm.llvalue
```

Where variable type is:

Listing 6: variable type

```
type variable =
    | Ptr of variable
    | Int
    | String of Llvm.lltype
    | Sexp of string * variable list * int
    | Array of variable list * int
    | List of variable list * int
    | Closure
```

Now we are able to keep track of types of stack variables in main function. To keep this in other functions and closures we need to know type of arguments and locals that are taken in closure. It leads to another problem: functions can be called with different types because Lama is untyped. It can be solved by generating function when function is called. And every call with different arguments types leads to generating new function with new body. So comparison of this 2 methods looks like this:

first solution	second solution
occupies a lot of space on heap every variable is stored on heap	Only arrays and s-expressions are stored on heap
We do not need to maintain any structures for understanding type in compile time	We need to keep track of types on stack, convey it to labels that should upload this types when label is reached, convey it to functions
We do not regenerate code in compile time, s-expressions, strings and arrays can be treated the same way	We can regenerate code in compile time to make functions type independent
Functions are generated in runtime, because we do not know function signature in compile time. All generation can be made only in runtime.	Functions are not generated in runtime.
All function signatures for different types are the same: i8* casted to i32. Limited generation of code.	Functions signatures represent real types. Except of S-expressions and arrays
Before every call, we have to collect all the argument types from heap and save a function to the map, where the key is its signature and the value is llvm function.	Easier to get types of function arguments. Also need to maintain map between signature and llvm function.
Functions that are not called can be omitted.	All existing functions are generated

First type is very memory expensive and every operation costs time in runtime: we are trying to understand type of variable in runtime and building our behaving depending on it.
Second type is to try understand type in compiling time and it can be broken by functions, whose return type can vary.

3.1 Functions, whose return type can vary

Functions like this are not supported by LLVM: llvm ir is strictly typed language. Additionally, functions with varying return types can disrupt the entire system of tracking stack variable types. If we are unaware of the return type of a function, we cannot determine the type of the variables on the stack after the function call. To address this, we can either restrict the number of function return types to just one, such as i32, or limit the number of outer functions to a reasonable number, which would require special handling during the call process. Lama's compiler for SM was designed with support for the first approach. Nonetheless, our LLVM compiler can also accommodate such a feature by introducing an additional pass through the function. In the first pass, we simulate the stack and determine the function's return type. In the second pass, we create the LLVM function.

3.2 remaining instructions

The remaining instructions were implemented by comparing the output of clang with the input consisting of similar expressions in C. My script compiles the LLVM IR representation emitted by LLVM, links it with stdlib.o, which includes the functions used by Lama from C. Here is an example of the script:

Listing 7: label example

```
#!/bin/sh
llvm-as $1.ll
llc $1.bc
gcc -c $1.s -o $1.o
gcc stdlib.o $1.o
./a.out
```

3.3 optimizations

In this implementation, every instruction that uses a variable firstly creates it from a pointer on the stack. However, this is only necessary if there are conditional jumps that disrupt the code structure. In the Lama stack machine, there is an abstraction known as a Scope. Scopes consist of a begin label (blab), an end label (elab), local variables, and subscopes. An interesting aspect of this is that if a scope contains a conditional jump (cjmp), both of its branches will be in subscopes. As a result, variables created within the same scope can be used without loading them onto the stack and then retrieving them.

Here's an example:

Listing 8: stack machine instructions

```
    Const(0)
    Const(1)
    Binop (+)
```

Right now is compiling to

Listing 9: llvm code

```
    store i32 3, i32* %"8", align 4
    store i32 4, i32* %"12", align 4
    %"13" = load i32, i32* %"12", align 4
    %"14" = load i32, i32* %"8", align 4
    %"15" = add i32 %"13", %"14"
    store i32 %"15", i32* %"8", align 4
```

while it can be compiled to

Listing 10: shorted llvm code

```
    %"15" = add i32 3, 4
    store i32 %"15", i32* %"8", align 4
```

Instead of zero lines, constants now require two lines, and any operation that results in a value used within the same scope requires three lines instead of one. To implement this, I created a class called Scope_c with the following methods:

Listing 11: Scope_c

```
  method add_to_stack (value : variable_c)
  method front
  method get_from_stack
  method get_from_stack_without_cast =
  method is_empty : bool =
  method get_elab = s.elab
  method get_blab = s.blab
```

And variable_c with this methods:

Listing 12: Variable_c

```
  method get_value -> variable_type
  method get_parent -> parent_c

type variable_type =
  | Ptr of Llvm.llvalue
  | Value of Llvm.llvalue
```

Now, we need to determine the parent of a variable in order to load it onto the stack when exiting a parent scope or function. This adds complexity because the scope stack contains both variables and pointers to stack-allocated memory. Whenever the program encounters an 'SLABEL' instruction that marks the end of a scope, it calls the 'go_up' function to transfer the scope stack to its parent.

An alternative approach is to assign stack ownership to the function class. This eliminates the need for many copies between scope stacks during the compilation to LLVM IR process and simplifies the retrieval of values from the stack. In the older version, if a scope has an empty stack, it would need to traverse to its parent and retrieve the value from there, which could be a lengthy process.

3.4 Pattern matching.

LLVM IR and LAMA code are at a higher level than LAMA stack machine instructions. Some language constructions can be translated to similar constructions in LLVM. If we are able to recognize them, it would allow us to decrease the number of LLVM instructions. Many load-to-stack and store-from-stack instructions could be replaced with a single assignment. To implement this idea, we need to analyze the code not sequentially as it is done currently, but rather by considering the entire code at all times. Some constructions can be part of larger constructions, and if we recognize only the smaller ones and compile them to similar constructions in LLVM, we may miss the bigger ones. For example, an if-else construction can be part of a larger if-else if-else construction. Also, an if-else construction can be misunderstood as a case construction with several branches. The scope construct makes this work easier because it divides the code into smaller parts, giving us an understanding of how this code looked in Lama. Let's see some examples:

Listing 13: Variable_c

```
If statement is translated to:  fun f (x){
    if 1 then {x + 2 } else { x + 3 } fi
  }
```

Listing 14: Variable_c

```
BEGIN ("Lf", 1, 0, [], ["x"], [{ blab="L4"; elab="L5";
names=[]; subs=[{ blab="L7"; elab="L8"; names=[]; subs=[{ blab="L17";
elab="L18"; names=[]; subs=[]; }; { blab="L11"; elab="L12";
names=[]; subs=[]; }]; }]; }])
SLABEL ("L4")
SLABEL ("L7")
CONST (1)
CJMP ("z", "L10")
SLABEL ("L11")
LD (Arg (0))
CONST (2)
BINOP ("+")
CONST (0)
SEXP ("cons", 2)
SLABEL ("L12")
JMP ("L6")
LABEL ("L10")
SLABEL ("L17")
LD (Arg (0))
CONST (3)
BINOP ("+")
SLABEL ("L18")
JMP ("L6")
SLABEL ("L8")
LABEL ("L6")
SLABEL ("L5")
END
```

You can see a pattern here: the if statement consists of a conditional jump, scope, unconditional jump, and label connected with a conditional jump, scope, and again an unconditional jump to the same label.

Without pattern matching, these stack machine instructions compile to:

Listing 15: Variable_c

```
define i32 @Lf(i32 %0) {
  entry:
    %"7" = alloca i32, align 4
    %"4" = alloca i32, align 4
    store i32 1, i32* %"4", align 4
    %"5" = load i32, i32* %"4", align 4
    %"6" = icmp ne i32 %"5", 0
    br i1 %"6", label %"3", label %L10

  "3":                                              ; preds = %entry
  store i32 %0, i32* %"4", align 4
  store i32 2, i32* %"7", align 4
  %"8" = load i32, i32* %"7", align 4
  %"9" = load i32, i32* %"4", align 4
  %"10" = add i32 %"8", %"9"
  store i32 %"10", i32* %"4", align 4
  br label %L6

L10:                                                    ; preds = %entry
  store i32 %0, i32* %"4", align 4
  store i32 3, i32* %"7", align 4
  %"11" = load i32, i32* %"7", align 4
  %"12" = load i32, i32* %"4", align 4
  %"13" = add i32 %"11", %"12"
  store i32 %"13", i32* %"4", align 4
  br label %L6

L6:                                                     ; preds = %L10, %"3"
  %"14" = load i32, i32* %"4", align 4
  ret i32 %"14"
}
```

As you can see, the returning type from branch blocks must be stored in a pointer to the stack and then stored in a value for further purposes.

Alternatively, it can be compiled using the phi function: The phi function examines the last block from which the flow came to the current block. If the flow came from the "3" label, the phi function would assign the value of

What else can benefit from pattern matching Loops: Loops, such as for loops and while loops, often have repetitive patterns in their control flow. By identifying these patterns, we can transform them into more efficient LLVM code. For instance, we can use phi functions to track loop variables and eliminate unnecessary load and store instructions.

Switch statements: Switch statements with multiple branches can be optimized by recognizing patterns in the control flow. Similar to the if-else construct, we can use phi functions to handle the different cases and eliminate redundant instructions.

Function calls: Function calls can also benefit from pattern matching and optimization techniques. By analyzing the arguments, return values, and control flow within functions, we can eliminate unnecessary temporary variables and optimize the function call sequences.

Error handling: Error handling constructs, such as try-catch blocks, can be optimized by identifying patterns in exception handling and control flow. By recognizing common exception scenarios, we can generate more efficient LLVM code that handles exceptions more effectively.

Memory management: Memory allocation and deallocation patterns, such as dynamic memory allocations and deallocations, can be optimized. By analyzing the memory usage within a function or a scope, we can eliminate redundant allocations or deallocations and optimize memory management operations.

The main problem with pattern matching is that it requires a thorough understanding of the possible control flow. To make assumptions and apply pattern matching, we rely on scopes, where all jumps can only be performed to higher-level scopes. However, there are limitations to using pattern matching in cases where we lack information about the starting and ending labels of scopes. Unfortunately, the Lama compiler does not provide information about scopes in the main function.

3.5 Parsing Stack Machine Instructions

The Lama infrastructure provides two compilers: lamac and Lama-devel. While the Lama-devel compiler is configured to use Dune as a building system and the LLVM library for OCaml, it is unable to compile a significant portion of Lama constructions. On the other hand, lamac is a compiler for x86 systems. Fortunately, lamac can generate stack machine instructions as output. Therefore, we have decided to enhance the capabilities of compiling not only Lama code but also Lama stack machine instructions to LLVM. To achieve this, we have created a module that parses the file and emits a list of instructions. This module does not utilize any lexers or parsers; it simply splits the file into lines and separates the lines using the separators: [" (),] regular expression. Due to the simplicity of the instructions, this approach does not affect the parsing capabilities. However, since the current version of the LLVM compiler does not rely on scopes, the parser excludes them, resulting in empty arrays for the 'begin' instructions.

3.6 Function generation

In the previous chapter, we made the decision to generate functions at the time of their calls during compile time. But how can we accomplish this? To achieve this, we need to save the list of instructions for each function during the first pass. Then, we call a function that generates functions within the main function. When this function encounters a function call, it checks if a function with the specific signature has been generated before. If not, it generates the function. Additionally, while checking the existence of a function, it needs to verify if it is an external function and declare it accordingly.

3.7 Closures

4 Evaluation of the Investigation

This section discusses criteria that are used to evaluate the research results. Make sure your results can be used to published research results, i.e., to the already known state-of-the-art.

(target size: 5-10 pages)

5 Conclusions

Summarize the main aspects and results of the research project. Provide an answer to the research questions stated earlier.

(target size: 1/2 page)