A System for Bounding the Execution Cost of WebAssembly Functions

by

John Shortt

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Carleton University
Ottawa, Ontario
January 2023

© Copyright by John Shortt, 2023

I dedicate this work to my family whose support and encouragement sustained me but most especially to my wife Donna for helping me every day.

Table of Contents

List of Tables . . . . **vii**

List of Figures . . . . **ix**

Abstract . . . . **x**

Acknowledgements . . . . **xi**

Chapter 1 Introduction **1**

1.1 What Do We Mean by "Cost" 2
1.2 Motivation 3
1.3 Research Questions 5
1.4 Contributions 5
1.5 Outline 6
1.6 Dissemination 6

Chapter 2 Background and Related Work **7**

2.1 WebAssembly 7
2.1.1 Timeline 8
2.1.2 Technical Overview 9
2.1.3 Formal Specification and Type Safety 11
2.1.4 An Example WebAssembly Module 15
2.1.5 Current and Future Features 17
2.2 Program Analysis Techniques 18
2.2.1 Basic Blocks 19
2.2.2 Extended Basic Blocks 20

2.2.3 Static Single Assignment Form . . . 20
2.2.4 Loop Analysis . . . 21
2.2.5 Worst Case Runtime Analysis . . . 23
2.2.6 Static Analysis for WebAssembly . . . 23
2.3 Mobile Agent Security, Proof Carrying Code . . . 25
2.4 Example Application Domains . . . 26
2.4.1 eBPF . . . 26
2.4.2 Ethereum . . . 27
2.5 OCaml . . . 28
2.6 Bubble Sort . . . 29
2.6.1 C Implementation of Bubble Sort . . . 29
2.6.2 WebAssembly Code for Bubble Sort . . . 30

Chapter 3 Approach . . . 35

3.1 Current Needs and Methods . . . 35
3.2 Opportunities for Improvement . . . 36
3.3 A Path to a Solution . . . 37
3.4 Defining Cost . . . 38
3.5 Solution Overview . . . 39

Chapter 4 Analyzing WebAssembly Functions . . . 41

4.1 Background . . . 42
4.2 Basic Blocks . . . 45
4.2.1 Definition and Background . . . 45
4.2.2 Bubble Sort Example . . . 48
4.3 Extended Basic Blocks . . . 49
4.3.1 Definition and Background . . . 51
4.3.2 Bubble Sort Example . . . 53
4.4 Code Path Generation . . . 55
4.4.1 A Simple Example . . . 55
4.4.2 Bubble Sort Example . . . 61
4.4.3 Summary . . . 61
4.5 Analysis of Execution . . . 62
4.5.1 Symbolic Execution . . . 64
4.5.2 Static Single-Assignment Form . . . 66
4.5.3 Summary . . . 69

4.6 Loop Execution Cost 70
4.6.1 Approach 70
4.6.2 Bubble Sort Example 72
4.7 Cost Analysis 77
4.7.1 Notation and Cost Equations 77
4.7.2 Bubble Sort Example 79
4.8 Summary 82

Chapter 5 WANALYZE Implementation 84

5.1 Internal Representation 84
5.2 WebAssembly Module Reader 87
5.3 Basic Blocks 90
5.4 Extended Basic Blocks 93
5.5 Control Flow Graphs 95
5.6 Non-looping Extended Basic Block Algorithms 99
5.7 Symbolic Execution 102
5.8 Static Single-Assignment Form 104
5.8.1 Generation 105
5.8.2 OCaml Types 107
5.8.3 Simplification 109
5.9 Loop Analysis 110
5.9.1 L1 Loop Exit Conditions 111
5.9.2 L2 Loop Condition Variables 112
5.10 Cost Analysis 113

Chapter 6 Test Results and Evaluation 116

6.1 Bubble Sort 116
6.1.1 Glue Layer Code 117
6.1.2 Results 121
6.2 Dhrystone Benchmark 121
6.3 Sample Programs 122
6.4 Observations and Discussion of Results 123

Chapter 7 Conclusions and Future Work 125

7.1 Research Questions 125

7.2 Future Work 126
7.3 Conclusion 127

Bibliography **128**

List of Tables

Table 4.1 WebAssembly structure blocks 42

Table 4.2 Applying the definition to determine extended basic blocks for bubble sort 53

Table 4.3 Basic block costs for the compute function 59

Table 4.4 Path costs for the compute function 60

Table 4.5 Basic block costs for the bubble-sort inner loop 61

Table 4.6 Non-looping path costs for the bubble sort inner loop 61

Table 4.7 Array names for variables and memory in WebAssembly 63

Table 4.8 Symbolic execution of bubble-sort inner loop 65

Table 4.9 SSA example 68

Table 4.10 SSA condition simplification 68

Table 4.11 SSA condition variable updates 69

Table 4.12 Determining initial values for loop variables 74

Table 4.13 SSA simplification - first path 75

Table 4.14 SSA simplification - second path 75

Table 5.1 WebAssembly module sections 85

Table 5.2 Basic block successors 96

Table 5.3 Instruction groups and sub-groups 105

Table 5.4 SSA generation by instruction groups and sub-groups 106

Table 6.1 Bubble sort bound vs. actual - **wasmtime** fuel 118

Table 6.2 Dhrystone predicted / actual 122

Table 6.3 Sample programs (AMD Ryzen 7 3800XT, 3900 MHz) 123

List of Figures

Figure 2.1 Validation rule for load instruction [75] 13

Figure 2.2 Execution rule for load instruction [75] 14

Figure 2.3 Code for a sample WebAssembly module 16

Figure 2.4 C code to implement bubble sort 30

Figure 3.1 WANALYZE high-level architecture 40

Figure 4.1 Basic block flow control diagram for bubble sort 50

Figure 4.2 Extended basic block flow control diagram for bubble sort . . 54

Figure 4.3 Basic block flow control diagram for compute function 58

Figure 4.4 Loop structures 71

Figure 5.1 OCaml type definitions for the WebAssembly module type section 91

Figure 5.2 OCaml type definitions for the WebAssembly module 92

Figure 5.3 OCaml type definitions WebAssembly basic blocks 98

Figure 5.4 OCaml type definitions WebAssembly extended basic blocks . 99

Figure 5.5 OCaml type definitions for SSA 108

Abstract

WebAssembly is a programming language and virtual machine architecture that allows code to be executed in any environment that implements a WebAssembly runtime. WebAssembly has been formally specified using an abstract syntax, and a soundness proof of this specification has been written and mechanized. We build on this to create a system that determines a bound on the runtime cost of a WebAssembly function. We show that for a broad class of real-world programs this cost can be computed efficiently and we develop a software tool called WANALYZE that does so. The software tool is comprised of a set of algorithms that perform a series of transformations on the raw WebAssembly bytecode into forms that are more suitable for analysis. We test WANALYZE against a suite of programs of varying size and complexity and find that WANALYZE is able to successfully analyze over 99.9% of the functions in these programs.

Acknowledgements

Most importantly I would like to extend deepest thanks to my thesis supervisors Dr. Amy P. Felty and Dr. Anil Somayaji. Our regular discussions both challenged and inspired me. Your support, encouragement and feedback helped make the result of this work more satisfying. I really could not have done this without your guidance.

I'm very grateful to Dr. Guy-Vincent Jourdan and Dr. Vio Onut who strongly supported my idea of returning to school after 40 years in business. Their positive reaction and unreserved encouragement convinced me that this was an attainable dream. I'd also like to thank my thesis committee members, Dr. Jourdan and Dr. Lianying (Viau) Zhao for reviewing this work and providing valuable feedback.

I also extend thanks to my friend Ken Leese who first suggested the idea of the analysis of WebAssembly as a research project. The things that I've learned from Ken over the past 30 years, and longer, have contributed greatly to any success I've achieved in that time.

I'd also like to thank the professors and staff at Carleton University and Ottawa University who made great efforts and sacrifices to make sure that the graduate school experience was excellent during COVID times.

Finally, and without parallel, I'm grateful for the support that my wife Donna and the rest of my family provided to make sure that I had the freedom to pursue this idea of returning to school.

Chapter 1

Introduction

Significant computing and software engineering resources are expended to determine the expected or actual computational cost of executing a function, program, application or system. To mention a few examples: real-time systems undergo extensive analysis before deployment to ensure that latency requirements are met; environments that host arbitrary code limit CPU utilization by analyzing code when loaded or by running instrumented code and trapping when a predetermined limit is exceeded; and, cloud computing environments make load balancing decisions based on expected computational resource requirements. In other words, being able to determine the execution cost of an arbitrary piece of code has the benefit of allowing systems built with that code to be more reliable, efficient, secure and scalable. As we know, this problem cannot be solved in the general case (otherwise we would have a solution to the halting problem [51]) but we also know that formally defined systems can be reasoned about with sophisticated tools that allow us to make interesting conclusions about the behaviour of those systems. In this thesis we explore the analysis of code written in a formally defined language, namely WebAssembly, with the goal of statically determining the cost of executing a function in WebAssembly given the range

of inputs to that function. We imagine a system that takes the code of a function as an input and produces one of two possible outputs: i) a mathematical expression, in variables that represent the inputs to that function, for the cost of function; or ii) a signal that the analysis cannot be done. We design, develop and test a system, WANALYZE, to perform this analysis with the objective of maximizing the proportion of functions for which we successfully produce such an expression. We pay particular attention to results obtained from analyzing real applications written in high level languages such as C, C++, Rust and Go and compiled to WebAssembly binaries.

As a motivating example consider the Linux kernel's Extended Berkely Packet (eBPF) support. eBPF allows a user written, dynamically loaded, operating system performance measurement library to run in the OS kernel. Care must be taken to ensure that it will not destabilize the kernel and methods have been developed, including limiting the size and execution time of eBPF functions, to ensure this. We'll see that the fact that WebAssembly has a very simple and restricted set of flow control primitives (when compared to a high level language) that allow us to enforce these kinds of restrictions more precisely. There's a clear benefit in being able to assess the execution cost of a function: the environment that this function is to run in can be provided this information and use it to decide whether the cost is acceptable or not. This can improve stability, performance and reliability in environments that are CPU resource constrained or sensitive to CPU resource utilization.

1.1 What Do We Mean by "Cost"

We make a precise definition of cost in Chapter 3 where we define cost as the number of WebAssembly instructions executed by a function. In general this cost will be dependant on the inputs to the function and so we expect that it will be formulated as an expression that's dependant on those inputs. In determining a bound on cost

we seek to determine the worst case cost of executing a function.

More fine-grained cost functions that differentiate between the cost of different WebAssembly instructions (e.g. a division operation is more costly than an add operation) or that use other cost metrics (e.g. memory operations) could be considered and our work provides the basis for this, but we choose to treat all instructions as having the same cost.

1.2 Motivation

A solution, even a partial solution, to the problem of determining the execution cost of a software component using static analysis methods will allow us to make progress on both practical and theoretical issues that are broadly relevant in computer science today.

In practical terms, knowing the execution cost of a function can allow a software system that uses or contains that function to determine, a priori, if this cost will have a negative effect on the working of that software system. For example, systems that perform control functions in real time often need to meet latency guarantees for certain functions. A response to an input must be made within a restricted time window. Knowing the worst-case execution time of a function that responds to such an input allows us to determine whether the latency requirement can be met or not. Currently, significant software engineering testing and analysis resources are devoted to making sure that these requirements can be met. As another example, consider a client device such as a web browser or mobile device that loads code to be executed from a remote web site. The experience of the user of the web site will potentially be improved if the client device can offer a guarantee that the remotely obtained code will not use an inordinate amount of client device computing resources. More generally, any system that provides the ability to execute arbitrary code written by a third

party can benefit from this kind of guarantee. The system has the freedom to choose how to execute the third party code in this and may make a wide range of decisions such as letting it execute, denying it from executing, delaying its execution to a more suitable time, or allocating additional resources to ensure it executes satisfactorily.

The theoretical aspects of this problem are in once sense quite cut and dry and in another sense a challenging area of research investigation. It's clear that this problem admits no general solution: otherwise we would have a solution to the halting problem. Lucas [51] provides a history of the halting problem. For our purposes the main result is that it's been proven that, in languages that are Turing complete, it's not possible to write a program that takes an arbitrary program as input and determines whether that input program halts or not. All general purpose computing languages, like WebAssembly, are known to be Turing complete. If we were able to compute a strict execution bound for any WebAssembly function then we could use this information to state whether it halts or not thus contradicting the result of the halting problem.

We see, however, that many real world software components are amenable to the kinds of static analysis that will allow us to provide a solution. Furthermore, the language that we've chosen to use as the basis for this work, WebAssembly, has three characteristics that particularly make it suitable: it is formally specified and has no undefined semantics, being generally low level means it's conceptually simpler than a high level language and therefore, one expects, is simpler to reason about, and finally, its program flow control mechanisms use the ideas of structured programming in a way that also simplifies reasoning about them.

This research will result in methods that have a broader applicability than the problem at hand. We expect that we'll arrive at a deeper understanding of the practical limits of the approach we take, especially as applied to WebAssembly. We also think that the methods we develop will have uses for or extensions into solutions for a broader set of static analysis problems related to, for example, other security

and testing requirements.

1.3 Research Questions

In this thesis we focus on the following research questions:

- **RQ1** What minimum set of properties or preconditions must be satisfied by a WebAssembly function in order to allow us to successfully bound its execution cost? Or, conversely, when will such an analysis fail?
- **RQ2** For a given real-world application what proportion of functions that make up that application should we expect to be able to successfully analyse?
- **RQ3** Can we implement a system that performs this analysis in a timely enough manner that would allow it to be readily integrated into a WebAssembly runtime environment with minimal performance overhead?

1.4 Contributions

In this thesis we make the following contributions to computer science, static analysis and WebAssembly function analysis.

- We define a set of algorithms and data structures that can be applied to a WebAssembly function to facilitate the analysis of that function.
- We design a system that uses these algorithms and data structures to compute the cost of an arbitrary WebAssembly function.
- We build and test an implementation of that system, namely WANALYZE.
- We demonstrate that this system can successfully measure the cost of a functions in real-world applications in time that is linear in the size of the function.

1.5 Outline

The remainder of this thesis is organized as follows. Chapter 2 contains detailed background on WebAssembly, program analysis techniques, mobile agent security and proof-carrying code, OCaml, and the bubble sort algorithm (which we use as motivating example throughout the thesis).

Chapter 3 contains an overview of our approach to solving the problem. Chapter 4 contains detailed explanations of the analysis techniques that we use. In Chapter 5 we describe the implementation of WANALYZE with particular focus on the data structures and algorithms.

In Chapter 6 we describe the experimental tests that have been performed and their results. In Chapter 7 we discuss our conclusions and future work.

1.6 Dissemination

A preliminary version of this research along with preliminary results was presented in June, 2022 at the Program Analysis for WebAssembly (PAW) workshop during the ECOOP 2002 Conference in Berlin. Information about this workshop can be found here: https://2022.ecoop.org/home/paw-2022.

Chapter 2

Background and Related Work

In this Chapter we review background technical information from industry and research that's used in this thesis. We begin in Section 2.1 with an overview of WebAssembly including technical details, background and future developments. In Section 2.2 we describe program analysis methods, from the literature, that have direct applicability to our work. Section 2.3 describes the related fields of Mobile Agent Security and Proof Carrying Code and the relevance of our work to them. Section 2.4 describes the application domains eBPF and Ethereum. In Section 2.5 we describe aspects of the OCaml language that are used in our code examples. Finally, Section 2.6 contains the source code in both C and WebAssembly for the bubble sort example that's used throughout this thesis.

2.1 WebAssembly

WebAssembly [73] is an emergent systems technology that allows software components to be written in a range of programming languages to be compiled to a specified virtual machine target and to be executed in any environment that supports that virtual

machine target. It is supported in modern web browsers as a means of allowing applications and libraries written in a wider variety of languages than had been previously been possible. For example a video codec written in C [69] can now be compiled to WebAssembly and exposed as a library to be used by a JavaScript-based web application. Or a large, complex application written for a desktop environment (such as a CAD program [5] or a game rendering engine [68]) can be compiled to WebAssembly to, again, run in a web browser. It's not limited, however, to only run in a web browser: there are efforts under way in industry and standards groups to apply the concepts of WebAssembly in operating systems [10], Internet of Things devices [47], and network edge computing [80], for example.

2.1.1 Timeline

WebAssembly began in 2015 as a project undertaken by a community group whose members came from organizations (Apple, Firefox, Google, Microsoft) responsible for development of the most widely-used web browsers of the time. This group defined WebAssembly as follows:

> WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. WebAssembly is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications. [72]

One motivator for this group was the observation that JavaScript was increasingly becoming a compilation target for high level languages. Aspects of the language such as dynamic typing mean that JavaScript is not well-suited for this task.

The main goal of WebAssembly evolved as the project unfolded. By the time the first implementations of WebAssembly became available in 2017 [74] this goal had been re-stated as::

> WebAssembly is a safe, portable, low-level code format designed for efficient execution and compact representation. Its main goal is to enable high performance applications on the Web, but it does not make any Web-specific assumptions or provide Web-specific features, so it can be employed in other environments as well. [73]

There are some important ideas introduced by this restated goal: "low-level", "safe", "compact" and "[not] Web-specific". We have an explicit statement that WebAssembly provides a minimal set of operations ("low-level"). The emphasis in safety manifested as a decision to formulate the syntax and semantics of WebAssembly in a way that allows formal safety properties to be proven and those proofs to be mechanized. The original emphasis on WebAssembly as a compilation target has not diminished and, in fact, is currently supported by many high level languages (including C and C++ [23], Go [27] and Rust [60]) and is the primary means by which applications use WebAssembly.

By 2021 the work of this group resulted in browser-based implementations of WebAssembly becoming broadly available, and being widely used in a variety of applications including some novel examples like the Unity Games Engine [68], Google Earth [28] and AutoCAD [5]. Additionally, the use of WebAssembly in non-Web contexts was evolving quickly [10, 47, 80].

2.1.2 Technical Overview

In practical terms, the core components of WebAssembly can be considered to be

- A low-level language with a formal specification
- A virtual machine specification consisting of memory and instruction set architecture definitions that comprise that virtual machine,

- A translator that converts a module written in the portable WebAssembly instruction set to a hardware-specific instruction set,
- A runtime environment (typically a JavaScript-based code fragment hosted by a web browser but not restricted to that) in which functions in the translated module can be invoked.

A WebAssembly module is stored and transmitted in a binary format. It consists of multiple variable-length sections. The virtual machine has a stack architecture. Instructions perform low-level operations (hence the “assembly” in WebAssembly) and use the stack to store local variables, function parameters and intermediate results. Memory is modeled as a linear array of integer values. Data types are limited to 32, 64 bit integers and IEEE 754 floating point numbers

A typical usage scenario is that a developer compiles an application or library written in a high level language such as C, Rust or Go to WebAssembly. This produces a binary file called a module that contains the code that was compiled and other information that will allow the code to run in the runtime environment. The runtime environment, hosted in a browser or other hosting system, loads the binary and translates the code to the native machine language. The rules for this translation are given by the virtual machine and language specification. The module also defines which of its functions may be called by the host and their function signature. This allows, for example, a JavaScript program that loads a WebAssembly module to call a function in that module.

The main aspects of the low-level language and virtual machine that are important to our work are the following:

- It’s stack based. A value stack is used to hold the arguments to and the results from computational operations.

- There are only 4 native data types: integers and floating point numbers, 32 and 64 bit versions of each. These types are used for function parameters, function results, local variables, and global variables.

- The arithmetic and other data manipulation operations in the instruction set are low level. They typically operate on a small number of arguments on the stack.

- Flow control within a WebAssembly program is structured with no ability to branch to an arbitrary instruction.

- Memory is modeled as a linear array of bytes.

2.1.3 Formal Specification and Type Safety

The ideas behind WebAssembly were first introduced in the paper "Bringing the Web up to Speed" [32] published in 2017. The design goals were defined as fast, safe, and portable with semantics that are fast, safe, well-defined, independent across hardware, language, and, platform and open. The core semantic phases were defined as:

- Decoding: converting the binary representation of a module to a usable form e.g., compiling to machine code, de-compiling to a textual representation

- Validation: checking that a decode module is well-formed, including function type checks and operand stack state consistency

- Execution: Instantiation of a module (i.e. initialization) followed by invocation via an exported function

In addition to describing the motivation and goals of WebAssembly, this paper introduced the rules for the language's abstract syntax, small-step reduction and

validation. The abstract syntax is defined by a set of production rules for the grammar of the language. For example the rule

$$binop_{fN} ::= \textbf{add}|\textbf{sub}|\textbf{mul}|\textbf{div}|\textbf{min}|\textbf{max}|\textbf{copysign}$$

states that a floating point binary operator can be one of 7 possibilities and this operator is valid for both 32 and 64 bit floating point numbers.

The small-step reduction rules specify the type reduction that occurs as a result of executing an instruction. For example, the rule

$$(t.\textbf{const}\ c)\ t.unop \hookrightarrow t.\textbf{const}\ unop_t(c)$$

states that a unary operator reduces a constant of type t to another constant of the same type that's determined by applying the unary operator to the first constant.

Version 1.0 of this specification [75] was published in 2019. The rules evolved as the language specification was reviewed and revised. This is particularly true of the validation rules where a prose description was added to each rule. These rules are in a commonly used form that's described in Pierce's "Types and Programming Languages" [57]. Figure 2.1 is the validation rule for the load instruction that has been copied verbatim from the specification. The first two bullet points state the preconditions that must be met in order for the instruction to be valid. They correspond to the two predicates above the horizontal line in the rule. The third bullet point states the type of a valid load instruction. This corresponds to portion of the rule below the horizontal line.

A proof of the soundness of WebAssembly was given in the "Bringing the Web up to Speed" paper ([32] section 4.2). Some of the implications of this soundness are that valid programs have no undefined behaviour, no illegal memory accesses, no

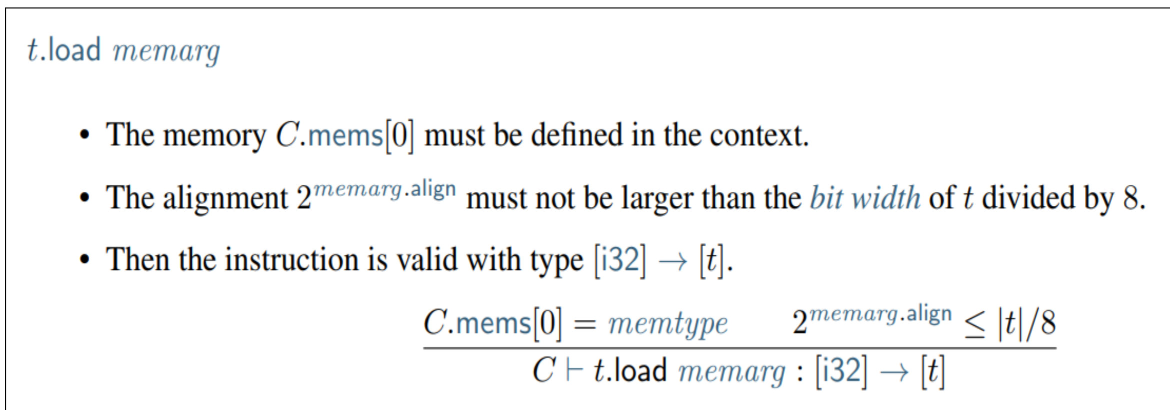

Figure 2.1: Validation rule for load instruction [75]

function calls or instructions executed with illegally typed arguments. As a result of the formalization in the language specification a sequence of papers was published [35, 70, 71] that mechanised this proof of soundness using the theorem proving tools Coq [7] and Isabelle [55].

In the specification document an additional type of rule was added: the execution rule. These rules specify the effect that an instruction has on the state of the virtual machine. They essentially define the semantics of each instruction. Figure 2.2 shows an example of an execution rule. The rule is for both the *load* and *loadN_sx* instructions. Note that the validation of the WebAssembly module allows some useful assertions to be made. In the figure these are items 2, 4, and 6 which respectively assert that (2) the module definition includes a memory object, (4) the module has been instantiated with a memory object, and (6) there is a valid memory index on the top of the stack. Item 10 verifies that the address being stored to is valid and traps if not. A trap causes execution of the WebAssembly code to be halted, the module's instantiation rendered invalid, and control returned to the host environment.

The benefit to our research of this formalization and proof of soundness is that we're able to apply the tools of static analysis to a WebAssembly binary without having to be concerned with the module's validity. We can assume that a WebAssembly

t.load $memarg$ **and** t.load$N$_sx $memarg$

1. Let F be the *current frame*.
2. Assert: due to *validation*, F.module.memaddrs[0] exists.
3. Let a be the *memory address* F.module.memaddrs[0].
4. Assert: due to *validation*, S.mems[a] exists.
5. Let mem be the *memory instance* S.mems[a].
6. Assert: due to *validation*, a value of *value type* i32 is on the top of the stack.
7. Pop the value i32.const i from the stack.
8. Let ea be the integer $i + memarg$.offset.
9. If N is not part of the instruction, then:
   a. Let N be the *bit width* $|t|$ of *value type* t.
10. If $ea + N/8$ is larger than the length of mem.data, then:
    a. Trap.
11. Let b^* be the byte sequence mem.data[$ea : N/8$].
12. If N and sx are part of the instruction, then:
    a. Let n be the integer for which $\mathrm{bytes}_{iN}(n) = b^*$.
    b. Let c be the result of computing $\mathrm{extend}^{sx}_{N,|t|}(n)$.
13. Else:
    a. Let c be the constant for which $\mathrm{bytes}_t(c) = b^*$.
14. Push the value t.const c to the stack.

$$
\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{load}\ memarg) \quad \hookrightarrow \quad S; F; (t.\mathsf{const}\ c) \\
\quad (\text{if } ea = i + memarg.\mathsf{offset} \\
\quad\ \wedge\ ea + |t|/8 \leq |S.\mathsf{mems}[F.\mathsf{module.memaddrs}[0]].\mathsf{data}| \\
\quad\ \wedge\ \mathrm{bytes}_t(c) = S.\mathsf{mems}[F.\mathsf{module.memaddrs}[0]].\mathsf{data}[ea : |t|/8]) \\
S; F; (\mathsf{i32.const}\ i)\ (t.\mathsf{load}N_sx\ memarg) \quad \hookrightarrow \quad S; F; (t.\mathsf{const}\ \mathrm{extend}^{sx}_{N,|t|}(n)) \\
\quad (\text{if } ea = i + memarg.\mathsf{offset} \\
\quad\ \wedge\ ea + N/8 \leq |S.\mathsf{mems}[F.\mathsf{module.memaddrs}[0]].\mathsf{data}| \\
\quad\ \wedge\ \mathrm{bytes}_{iN}(n) = S.\mathsf{mems}[F.\mathsf{module.memaddrs}[0]].\mathsf{data}[ea : N/8]) \\
S; F; (\mathsf{i32.const}\ k)\ (t.\mathsf{load}(N_sx)^?\ memarg) \quad \hookrightarrow \quad S; F; \mathsf{trap} \\
\quad (\text{otherwise})
\end{array}
$$

Figure 2.2: Execution rule for load instruction [75]

module that can be executed is valid and, therefore, we don't need to be concerned with edge cases like an invalid operand for an instruction or difficult cases like code that performs a branch into the middle of loop.

2.1.4 An Example WebAssembly Module

To bring the material in this section into perspective we provide a small example of a WebAssembly module that exports a function definition, imports another function and uses a third function internally. Figure 2.3 contains the text format of the WebAssembly module. This text format is formally defined in chapter 6 of the WebAssembly specification and is supported by the tools that have been built for WebAssembly development. The text format has an S-expression structure with the syntactic entity being the operator and the definition the operands.

Lines 1 and 26 open and close the module. They are required for every module. Lines 2 and 3 define the type signatures of the functions of the module. These type signatures are used by functions defined in the module and functions imported into the module from the environment. They specify the types of the inputs to the function and the output of the function. Line 4 defines a function called "sqrt" that's imported by the module. It's the responsibility of the host environment to provide the implementation of this function. Lines 7 through 18 are the implementation of a function that, as we see on line 25, is exported by the module and is called "distance". This function takes the x and y coordinates of 2 points in the plane and returns the distance between them. The calculations are done with 32 bit floating point numbers. The function on lines 20 through 24 is used as a helper function to square a number. The `local.get` instructions put the value of a local variable on the stack. The parameter of the instruction specifies which local variable. Parameters to the function are treated as local variables.

Each function has an index that uniquely identifies it. For clarity it's standard

practice to show that index as a comment in the function's definition. We can see this in lines 4 (;0;), 7 (;1;), and 20 (;2;). These indices are used when a function is invoked in a `call` instruction. It's also possible to use symbolic names for functions instead of indices. A function that's exported must be given a name as we see on line 25.

```
1  (module
2    (type (;0;) (func (param f32 f32 f32 f32) (result f32)))
3    (type (;1;) (func (param f32) (result f32)))
4    (import "env" "sqrt" (func (;0;) (type 1)))
5    ;; A function that takes 2 points in a plane and returns
6    ;; the distance between them.
7    (func (;1;) (type 0) (param f32 f32 f32 f32) (result f32)
8      local.get 0
9      local.get 2
10     f32.sub
11     call 2
12     local.get 1
13     local.get 3
14     f32.sub
15     call 2
16     f32.add
17     call 0          ;; sqrt function
18   )
19   ;; A function that takes a float and returns its square
20   (func (;2;) (type 1) (param f32) (result f32)
21     local.get 0
22     local.get 0
23     f32.mul
24   )
25   (export "distance" (func 1))
26 )
```

Figure 2.3: Code for a sample WebAssembly module

2.1.5 Current and Future Features

Development on WebAssembly is on-going with new features being planned for addition to the language. The current status of feature development is available at https://webassembly.org/roadmap/.

Features of note that are listed on this page as currently being available in all browsers are "Multi-value" which allows a function to return more than one value, "Reference Types" which allows a handle to a host object to be made opaque to a WebAssembly function thus limiting the risk of inadvertent information disclosure, and "Fixed-Width SIMD" which adds vector and matrix instructions to the WebAssembly instruction set.

We present the list of in-development or future features so that the reader may gain an appreciation of what's not in WebAssembly now and what is planned for:

- Exception handling - The proposal is to add `throw, try, catch` and `catch_all` instructions to the WebAssembly instruction set, giving it the ability to natively handle and generate exceptions. The challenge with adding this feature is the plethora of exception implementations in hosting environments that must be supported.

- Extended constant expressions - This feature allows simple arithmetic operations to be performed when creating a constant value.

- Memory 64 - Currently WebAssembly memory is indexed by a 32 bit integer and so is limited in size to 2^{32} bytes or 4 gigabytes. This proposal allows for memory that's indexed by a 64 bit integer which increases the maximum possible memory size substantially.

- Multiple memories - Only one linear array memory is allowed today. This proposal is to allow for multiple distinct linear memory arrays.

- Relaxed SIMD - This adds to the existing SIMD support by adding instructions that are similar to those found in some hardware SIMD implementations. It's not expected that these instructions would be available on hardware that doesn't support them. Additionally the results produced by these instructions vary depending on the hardware implementation and so there the possibility of nondeterminism that must be handled in the WebAssembly specification.

- Tail calls - Tail calls are important for compilers to be able to generate efficient WebAssembly code for some types of recursive functions.

- Threads and atomics - No specific thread support is planned to be added to WebAssembly. Instead it's expected that the hosting environment will provide this. There are, however, a set of atomic memory operations that have been proposed to support thread safe memory access and synchronization primitives in general.

- Type reflection - This feature is not about the WebAssembly language itself but about adding to the JavaScript API for WebAssembly. This API provides the mechanisms that allow JavaScript to host WebAssembly and this feature proposes to add API methods that allow it to query some aspects of the state of the WebAssembly virtual machine at runtime.

The Module Linking feature which appears on this web page is currently inactive.

2.2 Program Analysis Techniques

We use a number of techniques drawn from program analysis with an emphasis on those used in compiler and code optimization. Our problem is different from code optimization but many of the tools that compiler designers and developers use to analyze code, and make decisions about how to optimize it, require an understanding

of the cost of executing that code. And so, from this perspective, the tools that they use are also suitable for addressing our problem. These tools, however, aren't algorithms that produce a solution to a problem given a set of inputs. They are truly tools that need to be correctly used to solve the problem at hand. And so, like the compiler designer, we make choices about how we use these tools. Sections 4.2, 4.3 and 4.5.2 contain the details of how we do this. The remainder of this section is a brief introduction to the main ideas behind each of these tools.

2.2.1 Basic Blocks

The concept of basic blocks appeared in the code optimization literature as early as 1970. Cocke [13] defined basic blocks as "sections of a program which have the property that if any instruction in the block is executed all must be". Allen [2] refined this definition to "is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed)". We'll see in Section 4.2 that this second definition is similar to ours with the additional requirement that the instructions are executed sequentially. Any branching done in a basic block must be done by the last instruction of that basic block.

Contrast the idea of basic blocks with that of structure blocks (in a block structured program). Basic blocks aren't part of the definition of a programming language. They are constructs created by a compiler or some other tool to support the process of optimizing or analyzing generated code. Structure blocks are part of a language's definition and do carry semantics in that language, such as scope, which basic blocks don't.

The primary usage of basic blocks is the construction of control flow graphs where the nodes are basic blocks and the edges represent all valid execution paths. These graphs are analyzed to find ways to simplify expression calculation and eliminate common sub-expressions.

Basic blocks continue to be a tool that is used in the solution of code optimization problems. See, for example, Abel [1], a recent paper which describes models to predict basic block execution throughput on different microarchitectures.

2.2.2 Extended Basic Blocks

Extended basic blocks ([15] section 4.3.2) are a relaxed form of basic blocks where the restriction that branching is not allowed being removed. And so, for example, a loop can be wholly contained within an extended basic block. In fact this is the basis for how we can perform our analysis on loops as we'll see in Section 4.3. We do keep the requirement that execution of the extended basic block must begin with its first instruction.

2.2.3 Static Single Assignment Form

Static single assignment (SSA) form [19, 59] is a core analysis tool used in optimizing compilers. Its purpose is to allow common sub-expressions to be found in a sequence of assignment statements, the re-use without re-calculation of those common sub-expressions, and the re-ordering of calculations without changing the results. All of this is done with the objective of making the calculations more efficient.

SSA form achieves this by labeling each occurrence of a variable that's assigned to, as a new version of that variable. It then labels each of the variables in the computed expression to reference the version of the variable preceding the assignment. Since labeled variables are only unassigned to once in this scheme, optimization becomes simpler.

An additional aspect of SSA form deals with the problem of how to determine which version of a variable to use in an expression when there are multiple execution paths to the evaluation of the expression. This is achieved by introducing the function

Φ which takes as input the possible versions of the variable and returns the correct one based on the actual execution path. Program optimization routines are aware of the meaning of this Φ function and incorporate it in their algorithms.

Our use of SSA form is aimed at allowing us to simplify the expressions that result from calculations that are performed using WebAssembly's value stack. We create new temporary variables to which we assign the result of a single stack-based calculation. We use earlier created temporary variables in the new SSA form that we create. We'll see in sections 4.5.2 and 5.8 how this simplifies the calculations that we do. Because of the way we use SSA form we don't create cases where the Φ function is required. We always know which version of a variable is required.

2.2.4 Loop Analysis

Loop analysis or loop invariant analysis is an aspect of program analysis that focuses on finding program invariants that allow us to reason about the run time behaviour of a loop. In practice this is used in a variety of ways: loop optimization, dead code elimination, execution time prediction, and, software verification for example. As we'll see in Chapter, 4 a solution to the problem of determining a loop's run time behaviour is central to the objective of bounding the cost of a function.

Researchers have taken a variety of approaches including methods based on linear algebra [17, 30, 39], static analysis [33], reasoning methods based on theorem proving, SAT solvers, or algebraic method [20, 34, 44, 43, 46], learning algorithms [61], and a combination of these methods, for example [48].

Linear algebraic methods work by discovering linear relationships between the variables of a function at a specific point in the function and encoding these linear relationships in a matrix. This matrix effectively defines a linear subspace of all possible values that the variables can take at that point. Using standard matrix operations

such as row reduction the set of relationships can be simplified and any linear dependence between them removed. This can produce a matrix that defines a linear subspace of the variables that's of lower dimension, i.e. stronger set of invariants. The challenge is to effectively discover the pertinent linear relationships in programs with loops or recursive function calls. Wegbreit's algorithm [77] is commonly used, it uses symbolic execution of loops and recursive functions, in a way that's guaranteed to terminate, to generate invariants in these cases.

In [33] the authors describe static analysis methods that can be applied to analyzing the code of a loop in order to determine how many iterations of the loop will be executed. The method they describe allows for the use of a variable (rather than a constant) for loop increment and loop bound values. In this case a symbolic expression is produced for the number of loop iterations. We'll see in Chapters 4 and 5 that we provide a similar result. This paper has no discussion of nested loops.

Linear algebra and static analysis methods may, in general, suffer from imprecision and over-estimation of the number of iterations a loop will execute. This is because both methods are limited by the invariants that they discover and their ability to determine those invariants over all possible execution paths in a loop. More recent research has focused on methods to alleviate these problems by producing more precise invariants [20] and invariants that allow the number of execution paths to be reduced [46]. Removing the linearity restriction on invariants can also improve precision and so researchers [43] have focused on this. Automated theorem proving tools have been incorporated in some research work [34, 44]. This has the benefit of verifying the correctness of the analysis that's done.

In our work we found that we were able to produce an expression for most loops in the real-world WebAssembly applications through static analysis methods. We also were able to apply these techniques to nested loops and produce symbolic expressions in these cases. Chapters 4 and 5 describe this in detail.

2.2.5 Worst Case Runtime Analysis

Real-time systems can be defined as those that must respond to an input or stimulus within a defined time interval, usually on the order of milliseconds or less. Systems designers and testers primarily use a method called schedulability analysis [78] where, given the worst case execution time (WCET) of tasks that the system performs, an analysis is done to determine if a set of tasks can be feasibly completed within the required time. Researchers have investigated methods for schedulability analysis such as Earliest Deadline First (EDF) scheduling [81], and solutions for specific environments such as systems with a fixed set of priorities [8], multiprocessor systems [6], and distributed systems [56]. Of note is the fact that all of these results assume the existence of a WCET model of the system.

Knowledge of a bound for the execution cost of code within the system can greatly assigned the system designer and tester in ensuring that the response time requirement is satisfied. This information could be used in a schedulability analysis system to determine whether the response time requirements of the system can be met or not.

2.2.6 Static Analysis for WebAssembly

Researchers have published methods to perform general static analysis on WebAssembly code. This section describes in overview form some of these research results.

The basis for the static analysis of WebAssembly code was established in the 2019 publication of the WebAssembly 1.0 specification document [75] which include both a prose and formal specification of the language.

Several authors have described methods that use static analysis [49, 64, 65] to the problem of detecting security vulnerabilities in WebAssembly functions. These are summarized in a 2022 paper by by Kim, Jang, and Shin [42]. The paper by Stiévenart and De Roover [64] applied static analysis methods to perform data flow analysis of

WebAssembly functions. Their objective was to produce data flow summaries for those functions that could then be used to detect security vulnerabilities. Notably, they observed that the fact that WebAssembly only has structured flow control and doesn't have arbitrary jump instructions alleviates problems generally found in the static analysis of binaries.

These authors followed up in 2021 with a paper [62] introducing WASSAIL, a static analysis tool for WebAssembly. Their stated objective is to support a broad range of static analysis methods. The main features of WASSAIL are the ability to produce call graphs (showing which functions a given function calls), control flow graphs, and, to perform some data flow analysis functions.

In 2022, Stiévenart, Binkley and De Roover [63] described work to further WASSAIL's capabilities to support program slicing of WebAssembly. Program slicing is a general technique that, given a set of criteria, reduces a program to a minimal program that still meets those criteria. In their tests they showed that program slicing reduced binary sizes to 52% of their original size. Program slicing techniques might usefully be applied to loop analysis.

Goltzsche, Nieke, Knauth, and Kapritza describe [49] an efficient means of instrumenting a WebAssembly function for the purpose of counting, at runtime, the number of instructions executed. They use a method that's similar to the one that we've described in Section 4.4 to aggregate basic blocks into extended basic blocks (in our terminology). It's different from our approach in that it achieves this by modifying the code and collects data at runtime, rather than statically analysing the code.

Generally, the static analysis methods from these researchers are focused on analyzing data flow and don't use the basic block structures that we've defined. For this reason they're not completely applicable in our solution.

2.3 Mobile Agent Security, Proof Carrying Code

As web and mobile applications began to proliferate in the mid to late 1990s, the realization emerged that issues related to the security of these applications required attention. Particularly, researchers turned their attention to mobile agent security proposing platforms and models to address the issues [29, 66]. Mobile agents are applications that operate, in a distributed system, on behalf of third party. The third party does not know, or is necessarily concerned with, how or on what device(s) the mobile agent runs. The distributed system is free to assign computing resources to the mobile agent in a way that balances overall throughput with the need to perform the mobile agent's task. In some cases the mobile agent may need the credentials of the the third party in order to perform its task. Because a mobile agent is usually required to run in a heterogeneous environment they are often written in interpreted, machine independent programming languages [29].

A variety of security techniques have been used by mobile agents and their host environments to protect against security vulnerabilities. For host protection these include authenticating credentials, access-level monitoring and control, code verification, limitation techniques, and, audit logging. For agent protection fault tolerance, and, encryption methods can be used [9, 29]. Of particular interest to us are code verification, which includes scanning the agent code for validity and vulnerabilities, and, limitation techniques, which limit the computing resources made available to the mobile agent. Host-based limitation techniques also need to be sensitive to the resource requirements of a given agent in order to ensure it's allowed to run in an environment where it can complete successfully [66].

The ideas of proof carrying code emerged in the context of mobile agent security [9, 36, 45]. The main idea of proof carrying code in this context is that a mobile agent's code carries with it a ”safety proof” [53] that verifies that the agent adheres

to a given security policy. Importantly, this safety proof is constructed in a way that both it and the agent code cannot be tampered with, and, it can be quickly validated by the host. Proof carrying code also has broader applicability than mobile agents. For example, operating system kernel extensions in the form of proof carrying code can provide formal verification of the adherence of those extensions to the operating system's security policy [54]. Work has also been done on the problems of "verifying the verifier" and of efficiently producing proofs that can be verified [3].

In the context of our research, these developments are important. Supporting a security policy that allows the specification of limits on the execution time of an agent uses is a useful capability. Providing a way to provide this assurance automatically by statically analyzing the code of the mobile agent to produce a bound on its execution time could allow such a bound to be included in the proof of adherence to the policy that a proof carrying code system produces. This would have a positive effect on the security of the system that the mobile agent runs in.

2.4 Example Application Domains

In this section we provide background on two application domains that we'll show in Chapter 3 can benefit from the capability of bounding a function's execution cost.

2.4.1 eBPF

The extended Berkely Packet Filter (eBPF) is a facility within the Linux operating system that allows user-written functions to be added to the kernel without compromising its integrity [21]. Applications of eBPF include monitoring, data collection, security, and networking functionality.

The safety of eBPF is primarily achieved by using a special purpose virtual machine to sandbox the user-written functions. At the time the user function is loaded,

the kernel validates that it is well-formed and safe. One issue that this validation must address is that is that since the user function runs in the kernel its execution must be restricted. We'll come back to this issue in Section 3.1. eBPF functions are typically, but not necessarily, written in C.

Researcher and developers have proposed and produced a wide range of innovative solutions based on eBPF. Examples include a sandboxing facility for containers [26], network security [12], and load balancing [40].

2.4.2 Ethereum

Ethereum is a technology platform that supports a broad range of financial applications and functions based on its digital currency Ether (ETH) [24]. A key aspect of Ethereum is that it's a platform with a set of services and programming interfaces that allow it to be used by third parties to host their services within the Ethereum network.

Ethereum supports a capability termed Smart Contracts which allows the conditions and outcomes of a contract on the Ethereum block chain to be implemented as an executable program provided by the parties to the contract (or a third party). The principal behind this is that it provides an unambiguous method for contract settlement.

The computing costs for operations, including smart contracts, on the Ethereum network are paid for by the parties that perform those operations. The currency for this is termed gas and is equivalent to the Ethereum currency. One unit of gas is is equivalent to 10^{-9} ETH.

Controlling the cost of operations is an important part of using Ethereum effectively. If the cost of a smart contract cannot be limited then the parties risk an undesirable loss of currency. To this end Ethereum provides a way to provide a budget for the cost of an operation and to fail the operation if the budget is exceeded.

2.5 OCaml

The primary implementation language for our solution is OCaml [52]. In Chapter 5 the description of our methods requires an understanding of type definitions in OCaml. We present here a bit of background on types in OCaml that is needed in Chapter 5.

OCaml has a rich set of built-in types, including *list* and *option*, which our implementation makes extensive use of. Both of these types are used as type modifiers on other types to create a new type.

The *list* type has the semantics of a singly-linked list. Lists are immutable and so operations that modify a list create a new list as a result.

The *option* type supports the case where a value can be either a concrete value or *Nothing*. Loosely speaking, it can be seen as a generalization of the idea of *null* in Java, for example, by adding an underlying type to a *null* value. This improves the ability of the OCaml compiler to do type checking.

We use OCaml types in two different ways. The first is an enumerated type that specifies the finite set of values that a type can take. For example consider this line of OCaml code:

```
1 type primaryColor = RED | BLUE | GREEN
```

It creates a new type called *primaryColor* whose values can be one of *RED, BLUE,* or *GREEN*. These values are symbolic and have no other meaning unless we create one with further type definitions or functions.

We use OCaml types to create records as well. A record type consists of a set of type definitions for the attributes of that record. Here's an example of an OCaml record type definition:

```
type student =
{  name:     string;
   major:    string;
   gradYear: int option;
}
```

This defines the type *student* which has 3 attributes: a name and a major, both of which are a string (an OCaml built-in type) and optionally a graduation year which is an integer.

2.6 Bubble Sort

To make the ideas that we present in this thesis clearer we will use an implementation of bubble sort as an example throughout. Since a large part of the description of our methods requires examining, in detail, the intermediate results that we produce, we need an example that won't be overwhelmingly large and yet still illustrate the salient features of our analysis. Bubble sort meets this need: it's a small number of lines of C code but it contains 2 loops including a nested loop and a conditional execution block (the code that swaps out-of-order items). We'll reference both the C and WebAssembly versions of this code in the following sections so we provide an overview of it here.

2.6.1 C Implementation of Bubble Sort

Figure 2.4 contains the C code for bubble sort. It takes two parameters, the number of items to be sorted and a memory pointer to the items. The items to be sorted are signed integers and they are sorted, in place, in increasing order. The algorithm

is straightforward: it makes increasingly smaller passes through the list of items swapping items that are out of order. At the end of a pass the smallest integer found is guaranteed to be in the correct position.

```
1  void bubble(int n, int* data) {
2    int i, j, temp;
3
4    for(i = 0; i < n - 1; i++) {
5      for(j = 0; j < n - i - 1; j++) {
6        if(data[j] > data[j + 1]) {
7            temp = data[j];
8            data[j] = data[j + 1];
9            data[j + 1] = temp;
10       }
11     }
12   }
13 }
```

Figure 2.4: C code to implement bubble sort

2.6.2 WebAssembly Code for Bubble Sort

This end of this section contains the code that's produced by the **emscripten** [23] tool chain when the C code in Figure 2.4 is compiled to WebAssembly. Some aspects of this code are worth noting:

- Line 1: The parameters are of type signed integer. As with the C code the first parameter is the number of elements to be sorted. The second parameter is the index in WebAssembly's linear memory array at which the items to be sorted are located.
- Line 2: The function has eight local variables of type signed integer.
- Lines 3, 5, 10, etc: The code is annotated with comments of the form ";; BB

N". These comments were added after compilation. BB stands for basic block and the significance of their positioninf in the code will become clear in Section 4.2.

- Lines 4, 9, 18, 65 (and others): This code contains many new instructions compared to our previous example. The instructions `block`, `br_if`, `loop` and `end` are of particular interest and we'll see them in Chapters 4 and 5.

The length of this code testifies to the low-level nature of WebAssembly. A C function with less than 20 lines of code compiles to a WebAssembly function of close to 100 lines of code.

```
1   (func (;6;) (type 6) (param i32 i32)
2     (local i32 i32 i32 i32 i32 i32 i32 i32)
3     ;; BB 0
4     block  ;; label = @1
5     ;; BB 1
6       local.get 0
7       i32.const 2
8       i32.lt_s
9       br_if 0 (;@1;)
10      ;; BB 2
11      i32.const 0
12      local.set 2
13      local.get 0
14      i32.const -1
15      i32.add
16      local.tee 3
17      local.set 4
```

```
18        loop  ;; label = @2
19          ;; BB 3
20          i32.const 0
21          local.set 5
22          block  ;; label = @3
23            ;; BB 4
24            local.get 2
25            i32.const -1
26            i32.xor
27            local.get 0
28            i32.add
29            i32.const 1
30            i32.lt_s
31            br_if 0 (;@3;)
32            ;; BB 5
33            loop  ;; label = @4
34              ;; BB 6
35              block  ;; label = @5
36              ;; BB 7
37                local.get 1
38                local.get 5
39                i32.const 2
40                i32.shl
41                i32.add
42                local.tee 6
43                i32.load
44                local.tee 7
```

```
45            local.get 1
46            local.get 5
47            i32.const 1
48            i32.add
49            local.tee 5
50            i32.const 2
51            i32.shl
52            i32.add
53            local.tee 8
54            i32.load
55            local.tee 9
56            i32.le_s
57            br_if 0 (;@5;)
58            ;; BB 8
59            local.get 6
60            local.get 9
61            i32.store
62            local.get 8
63            local.get 7
64            i32.store
65          end
66          ;; BB 9
67          local.get 5
68          local.get 4
69          i32.ne
70          br_if 0 (;@4;)
71          ;; BB 10
```

```
72              end
73              ;; BB 11
74            end
75            ;; BB 12
76            local.get 4
77            i32.const -1
78            i32.add
79            local.set 4
80            local.get 2
81            i32.const 1
82            i32.add
83            local.tee 2
84            local.get 3
85            i32.ne
86            br_if 0 (;@2;)
87            ;; BB 13
88          end
89        ;; BB 14
90        end
91      ;; BB 15
92    )
```

Chapter 3

Approach

In the Background chapter (specifically, Section 2.4) we identified some example application domains that would benefit from a capability of determining a bound on a function's execution cost. We begin this chapter by elaborating on how this benefit could be achieved. We provide support for our perspective that methods that are currently used in these domains are primitive or inefficient.

3.1 Current Needs and Methods

We consider two application domains of interest, the extended Berkely Packet Filter facility for the Linux operating system (eBPF) and the Ethereum block chain network. Background on both of these domains is provided in Section 2.4.

For eBPF, the Linux kernel enforces restrictions that limit what an eBPF kernel extension is permitted to do: the virtual machine underlying eBPF is restricted in what it can do and the number of machine instructions that a kernel extension is allowed to execute in one function call is limited [14]. This impacts the functionality that can be effectively implemented in an eBPF kernel extension and represents a

roadblock to innovation. See, for example, this thread [67] on Stack Overflow.

Operations on the Ethereum network require payment (in the Ethereum digital currency, ether) for the computing resources they consume. Smart contracts are one example of this [58]. Since execution of a smart contract can involve a software function written by the author or a party to a contract, the cost of this contract depends on the computing resources consumed by this software function. To control the cost of an operation, a limit can be specified on its cost. If in the course of performing the the operation the limit is exceeded, the operation is failed having consumed the computing resources that allowed it to get to this failure point[25]. This is an inefficient use of resource but if it were possible to determine, a priori, what resources would be consumed then this inefficiency could be reduced or eliminated.

3.2 Opportunities for Improvement

Since a general solution to the problem of bounding a function execution cost needs to work with any function, a formal basis for ensuring this is required. As has been described in Section 1.2 the halting problem implies that there is no general solution; however, it may be the case that a solution for a broad subset of functions can be arrived at. And so we will want a solution that is able to identify the cases where we cannot produce a bound. All of this implies that a strong formal basis with existing proofs of correctness will be required.

If we can design a solution with a suitable technology that meets the characteristics of having a strong formal foundation then we will have made improvements to the application domains that we identified as examples and provided a general solution to a pervasive problem.

In recent history new programming languages such as Python, Rust, and Go

have become popularly used. The formal foundations of these languages aren't contained in the original language specification and efforts to address this issue have taken the form of follow-on research projects after the fact[4, 31, 38, 50]. Often the research projects reduce the scope of the language definition to make the effort possible. These languages have comprehensive specifications but without a formal proof of their soundness the possibility remains that these specifications contain contradictions or are incomplete. Other languages, such as C and C++ allow for programs with unspecified and undefined behaviour in their specifications [18]. The manner in which these specifications are written, as well as some of the language features (like dynamic typing), make this kind of formal proof impractical with the tools available today.

3.3 A Path to a Solution

As we described in Section 2.1.3 WebAssembly meets the need for a language with a strong formal specification that has been proven to be sound. Additionally the low level nature of WebAssembly, coupled with the fact that its flow control mechanisms are structured, means it's suitable for the kind of analysis that a solution to the function cost bounding requires.

The fact that the principal use of WebAssembly is as a compilation target means that code written in high level languages and compiled to WebAssembly inherits these formalization benefits of WebAssembly. It also means it's possible to use WebAssembly in the example application domains described above as long as they are capable of hosting a WebAssembly runtime.

The main approach that we've taken to solve this problem is to take a static analysis approach that uses methods from compiler construction and code optimization to develop a model of the flow of control within a WebAssembly function and to analyze

that flow of control to produce a bound on execution. The formal specification of WebAssembly means that we are proceeding on a solid foundation and the low level nature combined with structured flow control means the analysis is possible.

The community of researchers doing static analysis and language design work related to WebAssembly have generally used OCaml for their implementation work [62, 63, 64]. And so it is clearly an appropriate choice for the work being done here.

The result of this work is a tool called WANALYZE that takes as input a binary WebAssembly module and, for each function in the module, produces an bound on the cost of executing the function (based on its inputs) or fails if the bound can't be determined.

3.4 Defining Cost

To clarify our objective we provide a definition for the cost of a function, given the input values to the function, as the number of WebAssembly instructions executed by the function when called with those input values. Our goal is to produce an expression for this cost that is dependent on the inputs.

Definition 1. (Cost of a function) Given a WebAssembly function f that's invoked with the parameter values $x_1, ...x_n$, the cost of the function, denoted $|f|$, is the number of instructions executed by the WebAssembly virtual machine to complete the function invocation. Note that generally $|f|$ will depend on $x_1, ...x_n$.

The soundness [32] of WebAssembly ensures that this definition is meaningful. Any run-time implementation of WebAssembly that adheres to the specification has a well-defined behaviour that is the same for all such implementations.

We forego a more intricate cost metric (for example, one that accounts for differences in instruction costs) in order to focus on the problem of determining the cost for all possible paths through a function.

3.5 Solution Overview

Figure 3.1 depicts the high-level architecture of WANALYZE. There are 3 main categories of components that make up WANALYZE.

Internal Representation

The main ideas behind components to create WANALYZE's internal representation of a WebAssembly binary module are described in detail in sections 4.2 and 4.3 and the implementation in WANALYZE is described in 5.1 through 5.4. They consist of components to read a WebAssembly module and create OCaml record structures that mirror the module structure. Based on these record structures we build additional data structures (basic blocks and extended basic blocks) that are useful for analyzing the flow of execution in a WebAssembly function.

Execution Analysis

Execution Analysis components are based on basic blocks and extended basic blocks. They are described in detail in Chapters 4 and 5. Their main purpose is to determine cost bounds for certain cases, notably fragments of a function comprised of basic blocks or extended basic blocks with or without loops.

Cost Analysis

These components, that are also described in Chapters 4 and 5, take the intermediate results from Execution Analysis and combine them to create a cost bound for the function. This is done based on a set of inequalities that can be found in Section 4.7.1.

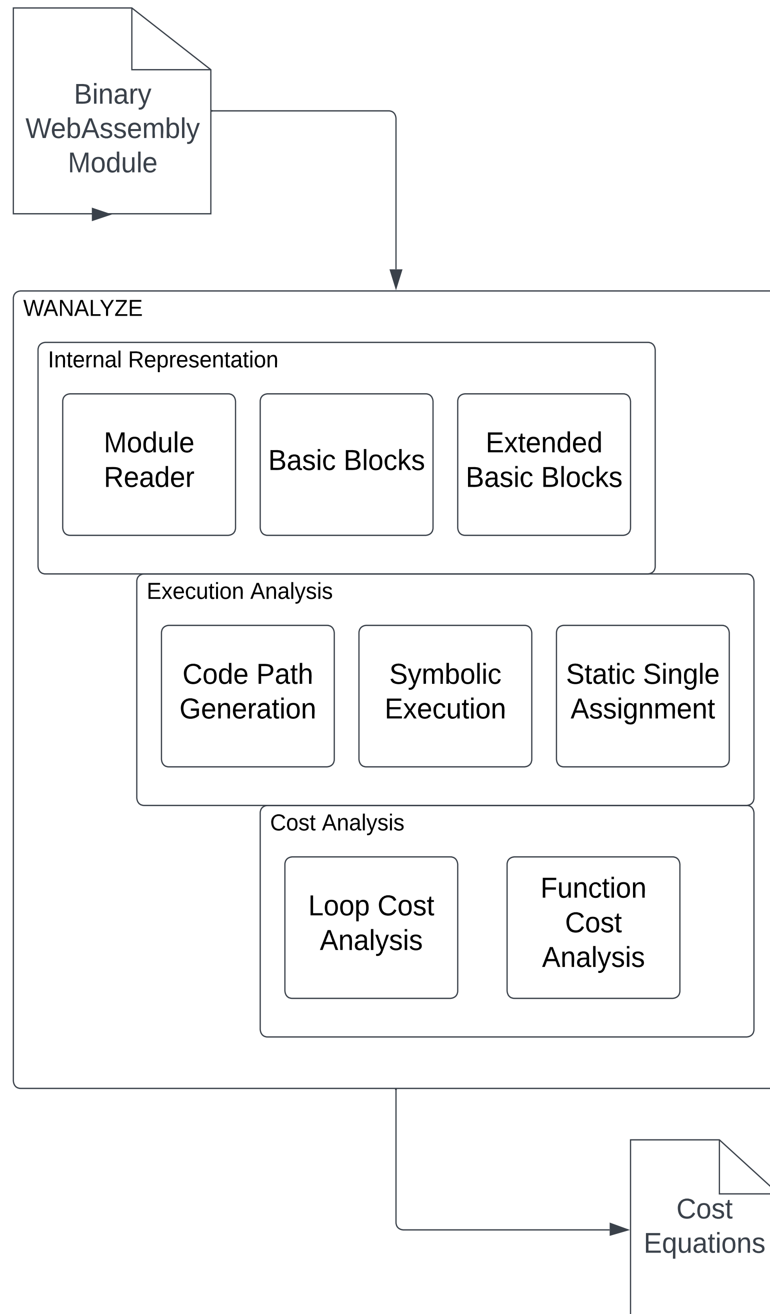

Figure 3.1: WANALYZE high-level architecture

Chapter 4

Analyzing WebAssembly Functions

In this chapter we describe, in detail, the ways in which we analyze the execution characteristics of a WebAssembly function. We define the main structures in a function that we analyze: the basic block and the extended basic block. We describe analysis tools that allow us to determine the cost of the parts of a function that meet certain assumptions: symbolic execution and the static single-assignment form. We define building block algorithms that put these structures and analysis tools together to allow a cost expression for the function to be determined.

Throughout this chapter we use examples to motivate and make clear how our techniques work. The primary example that we use is the bubble sort function but we supplement this with additional examples where beneficial. We show how what we learn from the examples can strengthened and applied in general to a broad class of functions.

Table 4.1: WebAssembly structure blocks

Block type	Label after	Notes
`if/else/end`	`end`	`else` is optional
`block/end`	`end`	
`loop/end`	`loop`	

4.1 Background

Before beginning to describe our methods for analyzing a WebAssembly function we provide some details that are important to understand about the WebAssembly instructions that create block structures in WebAssembly functions and how these structure blocks define statement labels which are used to define the targets of branch instructions that can alter the flow of control in a function. These *structure blocks* are different from *basic blocks and extended basic blocks* that we'll define in the following sections. Structure blocks are inherent in the WebAssembly language definition and not something that we define.

Table 4.1 contains a summary of the 3 types of structure block that can be found in WebAssembly functions. We use the names of the instructions that make up the structure block to distinguish these 3 types. The slash characters between the names of the instruction denote arbitrary instructions that make up the body of the block. Each instance of a structure block type in a function implicitly creates a label (depending on the structure block type). These labels are placed on the instruction immediately after the instruction specified in the "Label after" column in the table.

For example, consider a function that takes a signed integer n as input and returns the sum of the first n integers. It returns 0 if n is less than or equal to zero. (The code is written this way to illustrate the example. There are more efficient ways to do this. Integer overflows are ignored since they complicate the example unnecessarily).

The C code for this function might be written as follows:

```
1  int sum(int n)
2  {
3      int sum = 0;
4      if (n > 0) {
5          int current = n;
6          while (current > 0) {
7              sum += current;
8              current = current - 1
9          }
10     }
11     return sum;
12 }
```

A WebAssembly function, where a `block/end` structure block is used for the C `if` block and a `loop/end` structure block is used for a C `while` loop, is the following:

```
1  (func (;0;) (param i32) (result i32) (local i32 i32)
2      i32.const 0
3      local.set 1
4      block                    ;; create label 0 on line 27
5          i32.const 0
6          local.get 0
7          i32.le_s
8          br_if 0              ;; conditional branch to line 27
9          local.get 0
10         local.set 2
```

```
11          loop                    ;; create label 1 on line 12
12              i32.const 0         ;; label 1
13              local.get 2
14              i32.le_s
15              br_if 1             ;; conditional branch to line 27
16              local.get 2
17              local.get 1
18              i32.add
19              local.set 1
20              i32.const 1
21              local.get 2
22              i32.sub
23              local.set 2
24              br 1                ;; unconditional branch to line 12
25          end
26      end
27      local.get 1                  ;; label 0
28  )
```

In this function we have 2 structure blocks: the `block/end` structure block that's made up of instructions on line 4 through 26 and the `loop/end` structure block that contains the instructions on lines 11 through 25. This example shows that structure blocks can be nested. It also shows that label indexes being created when the first instruction either a `block, if` or `loop`, of a structure block is encountered.

4.2 Basic Blocks

4.2.1 Definition and Background

Reviewing the literature we see that analytical methods and their associated data structures that are used in computer language compilers have applicability to our problem. Especially relevant are the definitions found in Cooper and Torczon [15] for basic blocks (section 4.3.2), extended basic blocks (section 8.2.3), and the static single-assignment form (section 9.3). In the following sections we review the relevant aspects of these.

Cooper and Torczon define basic blocks defined are as follows:

Definition 2. (Basic Block) In a computer program a Basic Block (BB) is a contiguous sequence of instructions or statements that has the following properties:

- instructions in the BB are always executed sequentially (i.e. there are no changes in flow control within the basic block),
- if one instruction in the BB is executed then they all are

To apply the idea of basic blocks to WebAssembly we must understand how it allows for non-sequential execution. Unlike most machine instruction sets, WebAssembly does not include instructions that provide the means to branch to an arbitrary instruction. Rather, these 5 instructions: `block, loop, if, else,` and `end`, implicitly define labels that can be branched to, and a scope in which the labels can be referenced. The location of these labels is determined by the semantics of the label-defining instruction:

- the `block, if,` and `else` instructions each define a label on the instruction immediately following their corresponding `end` instruction. This provides a way to branch out of an `if/else/end` structure block.

- the `loop` instruction defines two labels: one on the next sequential instruction and one on the instruction immediately following its corresponding `end` instruction. These provide a way to branch to the "top" of the loop and a way to exit the loop. When discussing loops we'll use the terms *continue* and *break* for these two actions respectively.

A label is only visible within the scope of the structure block (i.e `block/end,` `if/else/end`, or `loop/end`) in which it's defined. For example a branch from the body of one loop to that of another non-nested loop is not possible.

WebAssembly has 6 instructions that can cause a non-sequential flow control within a function. They are `if, br, br_if, br_table, return,` and `unreachable`. The semantics of these instructions is what one would expect from their names:

- `if` removes the top value from the value stack and continues execution sequentially if the value is non-zero, otherwise it branches to the instruction immediately after the corresponding `else` if there is one, and to the corresponding `end` if there isn't.

- `br` unconditionally branches to the label provided as the operand to this instruction.

- `br_if` removes the first value from the value stack and branches to the specified label if it's non-zero

- `br_table` removes the first value from the value stack and uses it as an index into a table of labels that's provided as an instruction operand. The label at this index is branched to.

- `return` exits the current function.

- `unreachable` causes a runtime error that makes the current function terminate and the runtime environment to stop functioning.

It's the responsibility of the WebAssembly compiler to track the definitions of these labels and to generate code in the target machine language that implements the required semantics. The language semantics allow this to be done statically, once, when the module is loaded. There is no facility in the language to generate a label dynamically.

With this understanding of the mechanics of flow control in a WebAssembly function in hand we can now define basic blocks for WebAssembly.

Definition 3. (Basic Blocks of a WebAssembly function) The (set of) Basic Blocks of a WebAssembly function are those basic blocks with these properties:

- each instruction in the body of the function is contained in exactly one of the basic blocks,
- each basic block ends with either a `block, loop, if, else, end, br, br_if, br_table, return, unreachable` instruction and contains no other occurrences of any of these instructions.

Defining basic blocks in this way benefits our analysis in two ways:

- the set of basic blocks is uniquely determined and contains the minimum number of basic blocks possible for the function,
- no basic block can be made larger by including more instructions in it.

In this sense the basic block structure is optimal given the basic block constraints.

The WebAssembly syntax rules require that the last instruction of a function must be an `end` instruction. So each function has at least one basic block. The cost of a function with exactly one basic block is the number of instructions in that basic block which is the same as the number of instructions in the function. This leads us to the following definition.

Definition 4. (Cost of a Basic Block) The cost of a basic block of a WebAssembly function is defined as the number of instructions in that basic block.

As a final note in this section observe that we don't regard the WebAssembly `call` and `call_indirect` instructions to be terminators of a basic block. This is because, at this point, we are only interested in flow of control within a function and these instructions don't alter that.

4.2.2 Bubble Sort Example

The WebAssembly source code for the implementation of bubble sort can be found in Chapter 2. This source code is annotated with comments that delimit the basic blocks contains the source code. The basic blocks are number sequentially from zero and each of them terminates in one of the instructions that we've identified in the definition. From this basic block structure we are also able to produce a control flow diagram that shows how execution can proceed from basic block to basic block. This diagram (actually, the underlying data structure) will be useful when we begin to compute code paths in the function. Figure 4.1 depicts this.

Note that labels are written in the source code at the beginning of a `block/end` structured block, and that branches to that label go to the `end` instruction in the structured block. For example, consider the `block/end` structured block that includes source code lines 4 through 90. The source code comment on line 4 references "label @1" which is the label implied by the `block/end` structured block. The conditional branch on line 9, which uses this label, takes control to line 91 if the condition is true. In the control flow diagram, this is illustrated by the arrow from BB 1 to BB 15. On the other hand, in a structured block of the form `loop/end` the label is defined to be at the beginning of that structured block. This allows a loop to be continued by branching to the label. For example, consider the `loop/end` structured block on lines

33 through 72. Label "@4" is on line 33, and the branch on line 70 takes control to line 34. This corresponds to the dashed (red) arrow from BB 9 to BB 6.

It's worth noting a few other points about this diagram and how it relates to the bubble sort WebAssembly source code:

- Basic blocks are indexed sequentially from the beginning of the function. A basic block with a lower index occurs closer to the beginning of the function than one with a higher index.

- Transitions shown with dashed (red) lines (i.e. from basic blocks 9 to 6 and from 12 to 3) represent those where execution flows backwards from a later basic block to an earlier one. This occur in loops.

- the nested structure of the bubble sort loops is apparent in the diagram. It's shown by the nested dashed (red) transition from basic blocks 9 to 6.

- A conditional expression (i.e. `if` or `br_if`) is apparent in the diagram at basic blocks 1, 4, 7, 9, and 12. In those cases there is more than 1 successor to the basic block.

4.3 Extended Basic Blocks

Basic blocks provide a higher level abstraction of a function than that of a list of instructions. They provide us a way to create diagrams where we see the overall flow of control within a function. They also, importantly, provide a way to measure execution cost at this higher level. We see the a potential benefit in defining an even higher level of abstraction and applying these ideas of a higher level flow control diagram, and a way of measuring execution cost to it.

We observe that a function has a coarser grained structure than that represented by basic blocks. Specifically, sets of basic blocks within the function can be grouped

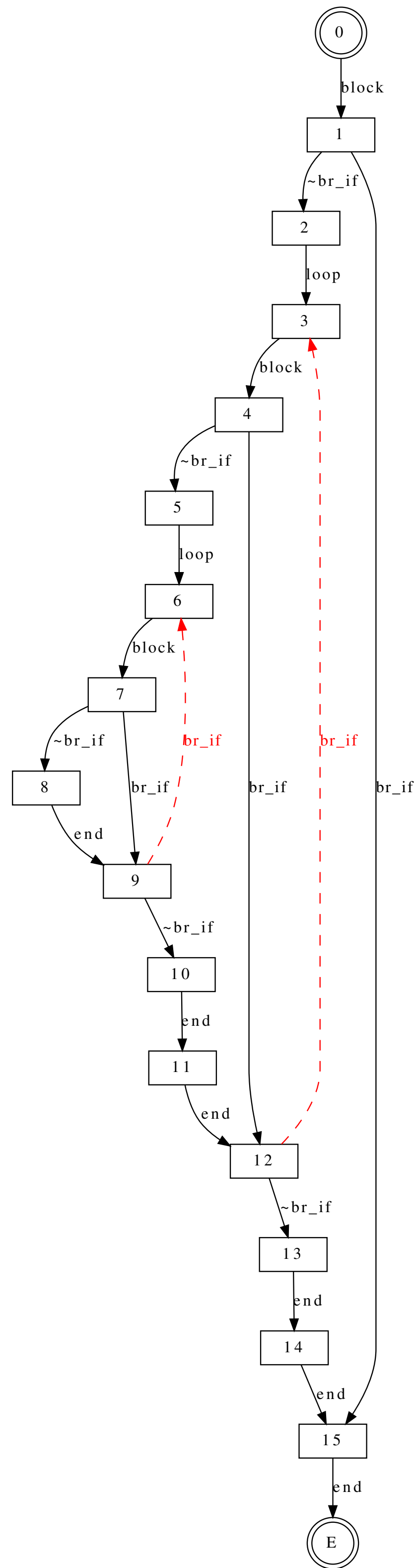

Figure 4.1: Basic block flow control diagram for bubble sort

to make up the structure blocks such as `loop/end`, and `if/else/end` code blocks. It seems that it would be useful to take advantage of this structure and to apply analysis to these groupings of basic blocks.

To represent this higher level structure we define Extended Basic Blocks. In our definition, an Extended Basic Block is composed of basic blocks, and, in the case of nested loops, other Extended Basic Blocks. We want to diagram and analyze the flow of control at the level of Extended Basic Blocks. In order to ensure this, we require that the basic blocks within an Extended Basic Block are consecutive and that execution flow control can only enter an Extended Basic Block at its first basic block.

We also require that a loop structure be contained entirely in a single Extended Basic Block and that the non-loop basic blocks of the function be aggregated into as large as possible an Extended Basic Block, subject to the other constraints. These provisos give us a coarser grained structure without obscuring any of the important structural aspects of the function.

Since an Extended Basic Block may contain a loop or other conditional structure we see that we won't be able to compute a strict cost for it but we want to be able to compute a bound on the cost of execution of an Extended Basic Block. The important structural components of a loop that we'll need to understand for the purpose of computing a cost bound are the initial state of the virtual machine when we enter the loop, its body, which can be made up of many basic blocks, the conditions under which the loop is exited, and its nesting structure.

4.3.1 Definition and Background

Synthesizing these ideas into a data structure definition we arrive at the concept of Extended Basic Blocks.

Definition 5. (Extended Basic Block of a WebAssembly function) An Extended Basic Block (EBB) of a WebAssembly function contains both Basic Blocks and possibly other Extended Basic Blocks. The conditions that determine how Basic Blocks are assigned to Extended Basic Blocks are as follows:

- C1 An Extended Basic Block is created for each `loop/end` structure block in the function. It contains the basic blocks of that code fragment beginning with the first basic block after the `loop` instruction and ending with, and including, the basic block containing the associated `end` instruction.

- C2 The Extended Basic Block of a nested loop is contained in the Extended Basic Block of the loop in which it's immediately nested.

- C3 A function's flow of execution can only enter an Extended Basic Block at the first Basic Block in the Extended Basic Block.

- C4 Consecutive Basic Blocks with no loops are placed in the same Extended Basic Block subject to C3.

Some clarifying remarks about this definition are in order. We ensure that all the basic blocks in a loop structured block are contained in a distinct extended basic block, except the basic block containing the loop instruction. It isn't part of the loop EBB because the instructions in that basic block are not inside the loop. The loop instruction is the last one in that basic block. Also, the loop structured block is the only structured block that's treated in this way. The others (if and block) are treated as prescribed in the other conditions in the definition.

With this definition we see that the function can be decomposed into Extended Basic Blocks of two types: those with loops and those without. Extended Basic Blocks can be named unambiguously by the index of the first Basic Block they contain. When

Table 4.2: Applying the definition to determine extended basic blocks for bubble sort

BBs of EBB	Which rule(s)?	Contained in
3 ... 13	C1	
6 ... 10	C1, C2	3 ... 13
0 ... 2	C4	
15	C3, C4	
14	C3, C4	
3 ... 5	C4	3 ... 13
11	C3, C4	3 ... 13
12 .. 13	C3, C4	3 ... 13

a function has nested loops, the Extended Basic Block that contains the outer loop will contain the Extended Basic Block that contains the inner loop.

4.3.2 Bubble Sort Example

Table 4.2 summarizes how we derive the set of Extended Basic Blocks for our bubble sort example. First we create EBBs for each of the loops in the function by applying condition 1 from the definition. This give us an EBB containing the BBs 3 to 13 inclusive and then an EBB containing the BBs 6 through 10 which, since it's a nested loop, is contained in the first EBB. Next we apply C4 to get the EBB containing BBs 0 through 2. At this point the remaining BBs that haven't been assigned to an EBB are 14 and 15. Since BB 15 can be entered from either BB 14 or BB 1, C3 tells us that it can't be in the same EBB as BB 14. So we create an EBB for each of BB 14 and BB 15.

The diagram in Figure 4.2 shows the resulting extended basic block structure for the bubble sort example. Each rectangle represents an EBB. A dashed (red) rectangle is used for an EBB that contains other EBBs. Lines with arrows are used to show flow control between the EBBs with a dashed (red) line used when the flow control is a loop being continued.

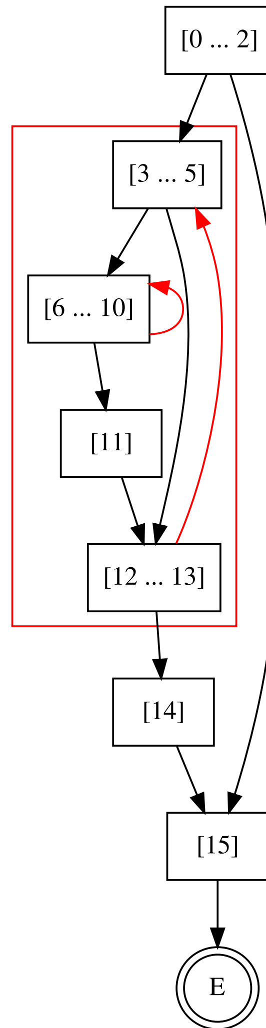

Figure 4.2: Extended basic block flow control diagram for bubble sort

4.4 Code Path Generation

We continue by using the definitions of basic blocks and extended basic blocks to show how they help us with solving the problem of determining the cost of a function with no loops.

4.4.1 A Simple Example

We start with a simple C function that's compiled to WebAssembly to illustrate the issues we encounter. Our example function takes 5 parameters and evaluates a polynomial whose definition is based on the parameters. The first parameter is the degree of the polynomial, the second is the value of the variable in the polynomial. The remaining 3 parameters are the polynomial coefficients. If the degree of the polynomial is less than 4 it's evaluated, otherwise the value of variable is returned.

```
1  int compute(int n, int x, int a, int b, int c) {
2    if (n == 0)
3      return a;
4    else if (n == 1)
5      return a*x + b;
6    else if (n == 2)
7      return a*x*x + b*x + c;
8    else
9      return x;
10 }
```

The C compiler in the **emscripten** [23] toolchain is able to optimize the WebAssembly code that's generated for this function to use the `br_table` instruction.

This instruction is similar to the `switch/case/break` construct in C. The resulting WebAssembly code has eleven basic blocks. Three of these basic blocks (5, 7 and 9) consist of single `end` instruction that serves only to delimit a `block/end` structure block. In all three cases the `end` instruction is preceded by an instruction that causes an unconditional branch in the flow of control: a `br_table` or `return` instruction. As a result control flow can never enter these basic blocks.

```
1  (func (;0;) (type 0) (param i32 i32 i32 i32 i32) (result i32)
2    ;; BB 0
3    block  ;; label = @1
4      ;; BB 1
5      block  ;; label = @2
6      ;; BB 2
7        block  ;; label = @3
8          ;; BB 3
9          block  ;; label = @4
10           ;; BB 4
11           local.get 0
12           br_table 3 (;@1;) 0 (;@4;) 1 (;@3;) 2 (;@2;)
13         ;; BB 5
14         end
15         ;; BB 6
16         local.get 2
17         local.get 1
18         i32.mul
19         local.get 3
20         i32.add
```

```
21          return
22        ;; BB 7
23        end
24        ;; BB 8
25        local.get 2
26        local.get 1
27        i32.mul
28        local.get 3
29        i32.add
30        local.get 1
31        i32.mul
32        local.get 4
33        i32.add
34        return
35      ;; BB 9
36      end
37      ;; BB 10
38      local.get 1
39      local.set 2
40    end
41    ;; BB 11
42    local.get 2
43  )
```

The control flow graph for this function is shown in Figure 4.3. Table 4.3 summarizes the cost of each basic block in the function.

Two facts allow us to determine a first bound on the cost of such a function:

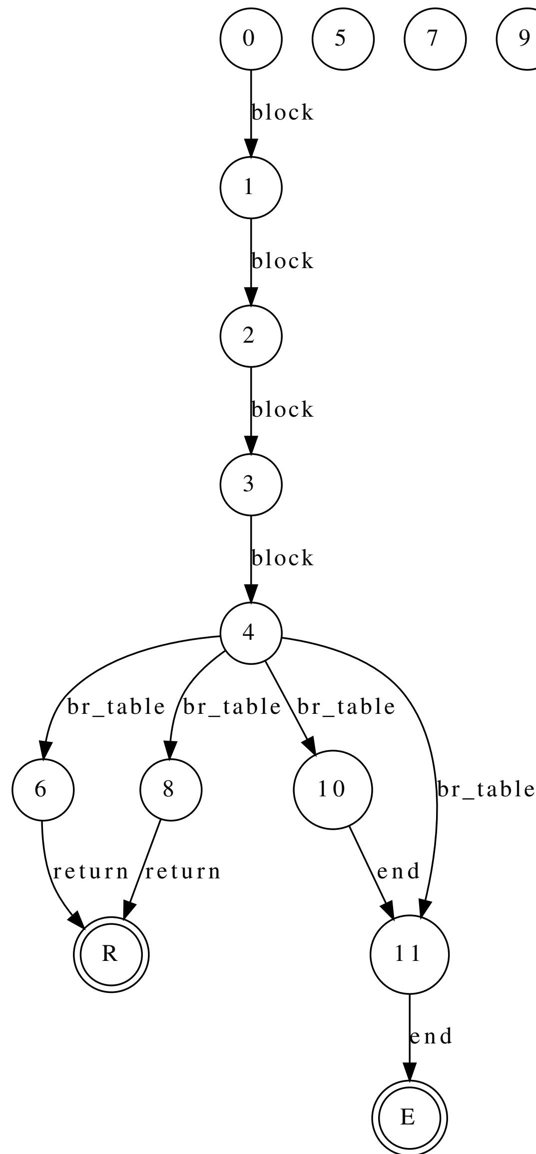

Figure 4.3: Basic block flow control diagram for compute function

Table 4.3: Basic block costs for the compute function

Basic block	Cost
0	1
1	1
2	1
3	1
4	2
5	1
6	6
7	1
8	10
9	1
10	3
11	1
Total	29

(i) the cost of executing a basic block is constant; (ii) in a function with no loops each basic block is executed at most once. This allows us to formulate the following bound on the execution cost: *the sum of the costs of all basic blocks*. For the `compute` function this is 29.

We can improve on this by making a further observation: the Control Flow Graph (CFG) for such a function is a directed graph with no cycles. And so we can formulate a tighter bound on the cost: *the maximum cost over all paths from the root of the tree to a leaf node* where we consider the cost of a path to be the sum of the cost of the basic blocks on that path. That is, we consider all possible execution paths through the function and take the maximum cost of those execution paths as our upper bound on the function cost. Since we may exclude the cost of some basic blocks in this bound it's clearly a tighter cost bound than the first.

We summarize the paths from the function entry to a leaf node for the `compute` function in Table 4.4. Taking the maximum path cost as the cost of the function gives us a bound of 16 for the function cost.

The challenge that we have in determining this second bound is that it requires

Table 4.4: Path costs for the compute function

Basic Blocks	Cost
0 1 2 3 4 6	12
0 1 2 3 4 8	16
0 1 2 3 4 11	11
0 1 2 3 4 10 11	8
Maximum	16

that we consider all paths through the function. In examining real-world applications it's not uncommon to encounter functions with hundreds of `if/end` structure blocks. The number of paths through the function is exponential in the number of such blocks and so can exceed 10^{10}. A brute force approach of enumerating all paths will consume gigabytes of memory on functions like this. While testing against functions found in real applications, we find functions that practically can't be costed because we run out of memory using a complete enumeration approach.

Instead of enumerating all possible paths we can iterate through the list of basic blocks and track the greatest cost, partial path to that basic block. After processing all basic blocks we will have the cost of traversal to each leaf basic block. This give us the maximum cost path. The runtime of this algorithm is shown in Cormen, Leiserson, Rivest and Stein [16] to be $O(V + E)$ where V is the number of vertices (i.e. basic blocks in our case) and E is the number of edges (i.e. execution paths between pairs of basic blocks) in the CFG. In the worst case E could be as large as V^2 (there's an execution path between every pair of basic blocks) and so the algorithm would have a time complexity of $O(n^2)$ where n is the number of basic blocks. This is a great improvement on the approach of enumerating all possible paths. Furthermore, as we'll see in the next section this analysis can also be applied to an extended basic block that have no loops to give us a cost bound in this case.

Table 4.5: Basic block costs for the bubble-sort inner loop

Basic block	Cost
6	1
7	21
8	7
9	4
10	1
Total	34

Table 4.6: Non-looping path costs for the bubble sort inner loop

Basic Blocks	Cost
6 7 8 9 10	34
6 7 9 10	27
Maximum	34

4.4.2 Bubble Sort Example

This example is based on the extended basic block that contains the inner loop of our bubble-sort program. From Figure 4.2 we can see that this inner loop is contained in the extended basic block made up of the basic blocks numbered 6 through 10. The cost of these basic blocks is summarized in Table 4.5. Looking at those basic blocks in Figure 4.1 we see that there are two non-looping paths through this extended basic block. Their composition and cost is summarized in Table 4.6. This gives us 34 instructions as the worst-case cost for executing the body of the inner loop of our bubble sort implementation.

4.4.3 Summary

To this point in this Chapter we've analyzed the cost of a function or extended basic block with no loops by examining the static structure of the function at the level of the basic block. We haven't had the need to know what the inputs to the function are. We could potentially make improvements to our bound by examining the code within each basic block and considering the possible execution paths within a

function. For example we might be able to determine that certain execution paths are not possible. We don't do this and instead make the more conservative assumption that any execution path is possible. We then use that maximum cost bound to determine a bound on the cost of the function or non-looping extended basic block.

From this analysis we arrive at the following result: *The cost of executing a non-looping extended basic block is bounded by the maximum cost of all basic block paths in the extended basic block.* In the next Chapter we explore ways to analyze the execution of a function in order to bound the cost of executing a loop.

4.5 Analysis of Execution

We now begin to turn our attention to the problem of bounding the cost of functions with loops. We start with a simple example, a non-nested loop whose body is contained in a single extended basic block. Recall from the previous chapter that we have a method for bounding the cost of such a body: it's the maximum cost of all paths through the extended basic block. If we can further bound the number of iterations of the loop then we can bound the cost of the loop over all iterations. Generally we expect the bound on the number of iterations to be dependent on the inputs to the function and so this bound will be parameterized by those inputs.

With this approach to bounding the cost of a loop as our motivation, we explore ways to further our understanding of the execution characteristics of WebAssembly code. Our goal is to understand the way in which branching decisions are made within a loop.

Given that the decisions made within a loop are based on the state of the virtual machine, we need a way to express the state of the virtual machine at any point during execution. For this purpose the machine state consists of three distinct types of information:

Table 4.7: Array names for variables and memory in WebAssembly

	i32	i64	f32	f64
Parameters and locals	n	N	f	F
Globals	gn	gN	gf	gF
Memory	m	m	m	m

- The value stack: we use a list for this,
- Memory: this is expressed as a single array called m,
- Local and global variables and function parameters: we use a small number of arrays that follow a naming convention to distinguish the data type (integer or float), the bitness (32 or 64), and the scope (global or local) of the variable values they store. This naming convention is shown in Table 4.7.

For example, the 6th local variable of a function that's a 32 bit integer is expressed as `n[6]`. The memory pointed to by this variable is `m[n[6]]` and a value stack containing this memory value followed by its location is `[m[n[6]]; n[6]]`. Observe that we use square brackets as a delimiter for the value stack and array indices and that we use the semi-colon to separate value stack items.

With this view of the virtual machine state in mind, and our focus on bounding the execution cost for loops, we can see that the body of a loop actually operates on this heterogeneous set of variables. We can observe that the looping decision (i.e. the decision to break a loop or continue it) will be made based on the content of those variables. This leads us to two analysis tools that can be applied to our objective of bounding loop execution cost: symbolic execution and the static single-assignment form.

The bulk of this section is a description of these two approaches and how they apply to our problem.

4.5.1 Symbolic Execution

The first technique that we explore is symbolic execution. Symbolic execution is a method of recording the effects on the machine state symbolically, instruction by instruction, when executing a code fragment. It gives us the ability to inspect the state of the virtual machine at any point in a functions execution. For example, we can look at a basic block that ends with a conditional branch instruction and determine symbolically what expression the conditional branch is based on.

By way of example, we examine the WebAssembly code for the inner loop of our bubble-sort implementation. From Figure 4.2 we can see that this code is contained in the single extended basic block comprised of the basic blocks with index 6 through 10. Looking at the WebAssembly bubble sort code listing in Chapter 2 we can see that this corresponds to lines 34 through 72 of the bubble sort code.

Table 4.8 contains this code. It has three columns: the first is the line number from the original WebAssembly module, the second is the instruction at that line number and the third is the stack contents after the instruction is executed and any variable or memory changes that the instruction causes. We use the convention that the left-most item in the stack is the one most recently added. Lines with comments are not shown in the table. To improve the readability of this we also convert the label indices in branch instructions to line numbers.

At a higher level this code behaves as follows: in lines 37 through 49 the 2 values to be compared are loaded into temporary variables, and the loop counter is updated (line 49), in lines 50 through 65 the two values are swapped if they are not in the correct order, and in lines 67 through 70 the loop counter is compared to the loop bound to determine if another loop pass is required. The variable `n[5]` is used for the loop counter and is modified only on line 49. The variable `n[4]` is used as the loop bound and is not modified in the loop body. Note that the code fragment uses

Table 4.8: Symbolic execution of bubble-sort inner loop

Line	Instruction	Updated stack contents / Instruction side effects
37	local.get 1	[n[1]]
38	local.get 5	[n[5]; n[1]]
39	i32.const 2	[2; n[5]; n[1]]
40	i32.shl	[(n[5] shl 2); n[1]]
41	i32.add	[n[1] + (n[5] shl 2)]]
42	local.tee 6	[n[1] + (n[5] shl 2)]
		n[6] ← n[1] + (n[5] shl 2)
43	i32.load	[m[n[1] + (n[5] shl 2)]]
44	local.tee 7	[m[n[1] + (n[5] shl 2)]]
		n[7] ← m[n[1] + (n[5] shl 2)]
45	local.get 1	[n[1]; m[n[1] + (n[5] shl 2)]]
46	local.get 5	[n[5]; n[1]; m[n[1] + (n[5] shl 2)]]
47	i32.const 1	[1; n[5]; n[1]; m[n[1] + (n[5] shl 2)]]
48	i32.add	[n[5] + 1; n[1]; m[n[1] + (n[5] shl 2)]]
49	local.tee 5	[n[5] + 1; n[1]; m[n[1] + (n[5] shl 2)]]
		n[5] ← n[5] + 1
50	i32.const 2	[2; n[5] + 1; n[1]; m[n[1] + (n[5] shl 2)]]
51	i32.shl	[((n[5] + 1) shl 2); n[1]; m[n[1] + (n[5] shl 2)]]
52	i32.add	[n[1] + ((n[5] + 1) shl 2); m[n[1] + (n[5] shl 2)]]
53	local.tee 8	[n[1] + ((n[5] + 1) shl 2); m[n[1] + (n[5] shl 2)]]
		n[8] ← n[1] + ((n[5] + 1) shl 2)
54	i32.load	[m[n[1] + ((n[5] + 1) shl 2)]; m[n[1] + (n[5] shl 2)]]
55	local.tee 9	[m[n[1] + ((n[5] + 1) shl 2)]; m[n[1] + (n[5] shl 2)]]
		n[9] ← m[((n[5] + 1) shl 2) + n[1]]
56	i32.le_s	[m[n[1] + (n[5] shl 2)] ≤ m[n[1] + ((n[5] + 1) shl 2)]]
57	br_if 67	[]
59	local.get 6	[n[6]]
60	local.get 9	[n[9]; n[6]]
61	i32.store	[]
		m[n[9]] ← n[6]
62	local.get 8	[n[8]]
63	local.get 7	[n[7]; n[8]]
64	i32.store	[]
		m[n[7]] ← n[8]
65	end	[]
67	local.get 5	[n[5]]
68	local.get 4	[n[4]; n[5]]
69	i32.ne	[n[5] ≠ n[4]]
70	br_if 37	[]

variables like `n[4]` that have been previously initialized. We'll need a way to account for that.

In summary, at this point we can bound the cost of a function with no loops or an extended basic block with no loops and we can find we have a symbolic representation of the virtual machine state at any point of execution.

Our goal is to use our ability to determine the loop condition to find a bound on the number of loop iterations. We don't, however, have a way to actually evaluate the loop condition. We need another technique to give us the ability to do that.

4.5.2 Static Single-Assignment Form

Although we know the condition under which a loop terminates we don't yet have a means of determining the values of the variables that are used in the loop condition expression. In addition, looking at Table 4.8 we can observe that the majority of instructions in the loop body don't actually affect the loop condition, the expression that's used to determine if the loop continues or not. If we can distill the code of the loop down to a minimal set of statements or instructions that allow us to determine how the variables involved in the loop condition are updated in the loop then perhaps this will simplify the analysis required to reach our goal of determining an expression for the number of iterations that a loop will execute. This is an example of a technique called "program slicing" [79].

A static single-assignment form (SSA) consists of a sequence of assignment statements using temporary variables and virtual machine state variables with the following properties:

- each temporary variable used in the sequence is assigned to exactly once,
- each temporary variable contains a value before it's used.

We don't impose restrictions on how the machine state variables are used.

Compilers generate SSA as an intermediate representation of a computation and then use this SSA to optimize the code generated for the computation. We can create this SSA representation for our loop body example. See Table 4.9 for the result of this. In this table we reproduce the line number and instruction from the symbolic execution table but we replace the machine state column with the SSA that's equivalent to the instruction. We use the identifier t with a subscript for the temporary variables that we use when an instruction produces a value on the stack.

We can use SSA when analyzing loops to determine loop conditions and how variables used in the loop condition are updated within the loop. In both cases we do this by first generating the SSA for the loop body and then iteratively expanding an initial SSA by replacing temporary variables with the right-hand side of their corresponding SSA. For the loop condition we start with the SSA of the loop condition. For loop variables we start with the set of variables in the loop condition.

This iterative expansion of an SSA is accomplished by SSA traversing the list of SSAs in reverse order, replacing occurrences of a temporary variable in the SSA we're expanding with the right hand side of the its corresponding SSA.

Table 4.10 shows how we do this for the loop condition. We start with the loop expression condition from the instruction on line 70 as our expression to expand: t_{22}. Stepping backwards we see that t_{22} is initialized on line 69 and so we update our expression to be expanded to be $t_{21} \neq t_{20}$. Continuing we expand t_{21} with its definition from line 68 and t_{20} with its definition from line 67. At this point we no longer have any temporary variables in our expression to expand and so we've identified the loop condition.

Next we scan the loop condition to determine the set of variables that are referenced in it. In this case it's $n[4]$ and $n[5]$. To determine how these loop variables are updated in the loop we follow a similar process to that described in the previous paragraph. We start with each SSA that updates the loop variable and expand it so

Table 4.9: SSA example

Line	Instruction	Equivalent SSA
37	`local.get 1`	$t_1 \leftarrow n[1]$
38	`local.get 5`	$t_2 \leftarrow n[5]$
39	`i32.const 2`	$t_3 \leftarrow 2$
40	`i32.shl`	$t_4 \leftarrow t_2$ shl t_3
41	`i32.add`	$t_5 \leftarrow t_4 + t_1$
42	`local.tee 6`	$n[6] \leftarrow t_5$
43	`i32.load`	$t_6 \leftarrow m[t_5]$
44	`local.tee 7`	$n[7] \leftarrow t_6$
45	`local.get 1`	$t_7 \leftarrow n[1]$
46	`local.get 5`	$t_8 \leftarrow n[5]$
47	`i32.const 1`	$t_9 \leftarrow 1$
48	`i32.add`	$t_{10} \leftarrow t_8 + t_9$
49	`local.tee 5`	$n[5] \leftarrow t_{10}$
50	`i32.const 2`	$t_{11} \leftarrow 2$
51	`i32.shl`	$t_{12} \leftarrow t_{10}$ shl t_{11}
52	`i32.add`	$t_{13} \leftarrow t_{11} + t_{12}$
53	`local.tee 8`	$n[8] \leftarrow t_{13}$
54	`i32.load`	$t_{14} \leftarrow m[t_{13}]$
55	`local.tee 9`	$n[9] \leftarrow t_{14}$
56	`i32.le_s`	$t_{15} \leftarrow t_{14} \leq t_6$
57	`br_if 67`	t_{15}
59	`local.get 6`	$t_{16} \leftarrow n[6]$
60	`local.get 9`	$t_{17} \leftarrow n[9]$
61	`i32.store`	$m[t_{17}] \leftarrow t_{16}$
62	`local.get 8`	$t_{18} \leftarrow n[8]$
63	`local.get 7`	$t_{19} \leftarrow n[7]$
64	`i32.store`	$m[t_{19}] \leftarrow t_{18}$
65	`end`	
67	`local.get 5`	$t_{20} \leftarrow n[5]$
68	`local.get 4`	$t_{21} \leftarrow n[4]$
69	`i32.ne`	$t_{22} \leftarrow t_{21} \neq t_{20}$
70	`br_if 37`	t_{22}

Table 4.10: SSA condition simplification

Instruction	SSA
70	t_{22}
69	$t_{21} \neq t_{20}$
68	$n[4] \neq t_{20}$
67	$n[4] \neq n[5]$

Table 4.11: SSA condition variable updates

Instruction	$n[4]$	$[n5]$
49		$n[5] \leftarrow t_{10}$
48		$n[5] \leftarrow t_8 + t_9$
47		$n[5] \leftarrow t_8 + 1$
46		$n[5] \leftarrow n[5] + 1$

that it has no temporary variables. In Table 4.11 we have the results of this process for each of the variables found in the loop condition. For $n[4]$ there are no SSA that assign to it and so there's no expansion required. For $n[5]$ we find the assignment on line 49 and then we use the SSA on lines 48, 47, and 46 to expand this assignment to remove the temporary variables.

With this approach we've reduced the condition of the loop to a single expression: $n[4] \neq n[5]$ and we've determined that the variables in that expression are updated in one way: $n[5] \leftarrow n[5] + 1$.

4.5.3 Summary

In earlier sections we've described ways that we can bound the cost of a function with no loops or an extended basic block with no loops. We used a basic block and extended basic block representation of the function together with a maximum cost graph algorithm for directed graphs with no cycles to achieve this. In this section we address the problem of determining the conditions under which a loop continues and how variables used in these loop conditions are updated. We show how to solve this problem for loops that meet certain restrictions: they have no nested loops, the body is a single extended basic block and there's a single condition that's used for determining loop condition.

Next we'll turn our attention to the problem of bounding the number of loop iterations. We'll use the results so far to show how we can do this.

4.6 Loop Execution Cost

The goal of loop analysis in WANALYZE is to find an expression that bounds the cost of execution of a loop. We observe that if we can bound on the cost of executing the loop body once and we can bound the number of times a loop body will be executed then we can combine these two bound to arrive at an expression for a bound on the loop execution cost. It's the product of the two bounds. This expression will generally include the parameters to the function.

4.6.1 Approach

We already have a bound on the execution cost of a loop body. It's an extended basic block and we have already shown that its execution cost can be bounded by the cost of the maximum cost path in the EBB. And so we focus on the problem of determining a bound on the number of iterations a loop will execute.

In order to understand the execution time characteristics of the loop it seems that we need to know the following:

- L1 Conditions that cause the loop to exit.
- L2 Variables that are used in evaluating these conditions.
- L3 Initial values of those variables when the loop is entered.
- L4 How the variables are updated in the loop.

We can determine the expressions that cause the loop to exit by noticing that there are essentially two ways to terminate a loop. Figure 4.4 shows an example of each of these. The diagram on the left shows a loop that has a conditional branch that continues the loop when the condition of a `br_if` instruction is true and stops

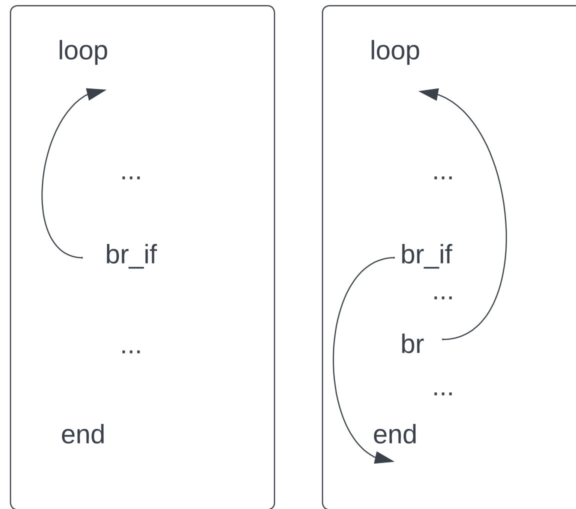

Figure 4.4: Loop structures

looping otherwise. The diagram on the right illustrates an alternative loop structure: a conditional branch that breaks the loop followed by an unconditional branch that continues the loop. In both cases the condition of a `br_if` instruction will determine if the loop is exited. In the first form the condition under which the loop continues is the condition evaluated in the `br_if` instruction. In the second form it's the logical negation of the condition of the `br_if` instruction.

To complete the set of possible ways to determine the conditions that cause a loop to exit we need to allow for cases where an `if` instruction or a `br_table` instruction are used. The structure will be the same as the `br_if` examples: there will be a `br` or `br_table` instruction that either continues or exits the loop and the corresponding condition of the `if` instruction will need to be reversed in logical sense if the loop is being continued.

And so the procedure to find the conditions that cause a loop is as follows:

1. Find all basic blocks in the extended basic block of the loop that are terminated by a `br, br_if,` or `br_table` instruction that either continues or exits the loop.

2. For the cases where a `br` instruction is used find the corresponding `if` basic block.

3. Enumerate all code paths from the first basic block to any of the basic blocks terminated by these `br_if`, `br_table`, or `if` instructions.

4. Symbolically execute each of these code paths to determine the loop exit condition. Either it or its logical negation will be the first item on the stack before the `br_if`, `br_table`, or `if` instruction is executed.

This procedure gives L1, the list of conditions under which the loop will be exited. The list will have more than one entry if there's more than one way to exit the loop.

We can now parse those conditions to determine what variables they use, i.e. L2.

Then, for each variable we can then create the static single-assessment form for all function execution paths to the loop EBB. We can then simplify these SSA as we did in the bubble sort example (Table 4.11) to get the possible values of the variables at loop entry. This gives us L3.

Finally, we can create a static single-assessment form for each of the variables but this time simplify the SSA over all execution paths from the first BB of the loop to the BB of their associated condition. This will give us L4.

This is then used to apply a set of heuristics depending on the condition and the manner in which the condition variables are update to determine the number of loop iterations.

4.6.2 Bubble Sort Example

To illustrate this we can use the example of the bubble sort inner loop, which is exactly the code covered by the SSA analysis in Table 4.8. Examining Table 4.8 we can see there are 2 `br_if` instructions in the body of the loop. The first one, on line 57, is a branch forward to line 67. Importantly, it doesn't exit the loop and so

we don't need to consider it's loop condition. (In fact it's checking whether a swap is needed or not. This has no bearing on whether the loop continues or not). The second `br_if` occurs on line 70, and it branches to the first instruction of the loop if it's condition is met. And so if the condition is not met the loop is exited. From Table 4.8 we can see that the condition is `n[5]` $\neq$ `n[4]` and the loop continues if this condition is true. Since there are no other conditionals in the loop we only have one item in the set of of loop exit conditions and it's the logical negation of this condition, namely `n[5] = n[4]`. This is our L1 in the terminology of the previous section

We can parse this expression to get L2 = {`n[4], n[5]`}. That is, our loop variables are `n[4]` and `n[5]`.

To determine L3, the initial value of each of these variables when we enter the loop we look at each code path to the loop. Examination of Figure 4.2 shows that there are two paths to the extended basic block of the loop, [6 ... 10] :

- the path that uses the extended basic blocks [0 ... 2] and [3 ... 5]
- and the path made up of the 6 extended basic blocks [0 ... 2], [3 ... 5], [6 ... 10], [11], [12 ... 13], and [3, ... 5]

The second one is the longest non-looping path from the start of the function to the extended basic block of the inner loop. By non-looping, in this case, we mean that it doesn't loop within the extended basic block of the loop. It does, however, loop outside that extended basic block. We must examine both of these paths to determine how the loop variables are initialized when we enter the loop. We do this by creating the SSA for the code on each path and then simplifying that SSA for the variables `n[4]` and `n[5]`. This process is the one described in Section 4.5.2. It shows that the only BBs in the paths that contain instructions that affect `n[4]` or `n[5]` are 2, 3, and 12 Table 4.12 shows the end result of this process with for those BBs. It contains the BB code and corresponding SSA where the analysis of the SSA shows

Table 4.12: Determining initial values for loop variables

BB	Line	Instruction	SSA
2	11	`i32.const 0`	$t_0 \leftarrow 0$
	12	`local.set 2`	$n[2] \leftarrow t_0$
	13	`local.get 0`	$t_1 \leftarrow n[0]$
	14	`i32.const -1`	$t_2 \leftarrow -1$
	15	`i32.add`	$t_3 \leftarrow t_1 + t_2$
	16	`local.tee 3`	$n[3] \leftarrow t_3$
	17	`local.set 4`	$n[4] \leftarrow t_3$
3	20	`i32.const 0`	$t_0 \leftarrow 0$
	21	`local.set 5`	$n[5] \leftarrow t_0$
12	76	`local.get 4`	$t_0 \leftarrow n[4]$
	77	`i32.const -1`	$t_1 \leftarrow -1$
	78	`i32.add`	$t_2 \leftarrow t_0 + t_1$
	79	`local.set 4`	$n[4] \leftarrow t_2$
	80	`local.get 2`	$t_3 \leftarrow n[4]$
	81	`i32.const 1`	$t_4 \leftarrow 1$
	82	`i32.add`	$t_5 \leftarrow t_3 + t_4$
	83	`local.tee 2`	$n[2] \leftarrow t_5$
	84	`local.get 3`	$t_6 \leftarrow n[3]$
	85	`i32.ne`	$t_7 \leftarrow t_5 \neq t_6$

that the loop variables are updated. Tables 4.13 and 4.14 show the simplification of the SSA to determine the how the variables are updated. These tables also only show the effect of instructions in the BBs that affect how the variables are updated. Blank entries in these tables occur when the instruction has no effect on `n[4]` or `n[5]`. From this we arrive at $\underline{\text{L3}} = \{n[4] \leftarrow n[0] - 1,\ n[5] \leftarrow 0\}$ for the first path and $\underline{\text{L3}} = \{n[4] \leftarrow n[4] - 1,\ n[5] \leftarrow 0\}$ for the second. The variable $n[0]$ is the first parameter of the sort function, namely the number of items to be sorted.

Finally, we've already shown in the previous section how to determine $\underline{\text{L4}}$. We saw that only $n[5]$ is updated in the loop and that $\underline{\text{L4}} = \{n[5] = n[5] + 1\}$.

To summarize, for our example we have

- $\underline{\text{L1}} = \{n[5] = n[4]\}$,
- $\underline{\text{L2}} = \{n[4], n[5]\}$,

Table 4.13: SSA simplification - first path

Instruction	$n[4]$	$[n5]$
21		$n[5] \leftarrow t_{10}$
20		$n[5] \leftarrow 0$
17	$n[4] \leftarrow t_3$	
16		
15	$n[4] \leftarrow t_1 + t_2$	
14	$n[4] \leftarrow t_1 + -1$	
13	$n[4] \leftarrow n[0] + -1$	
12		
11		

Table 4.14: SSA simplification - second path

Instruction	$n[4]$	$[n5]$
85		
84		
83		
82		
81		
80		
79	$n[4] \leftarrow t_2$	
78	$n[4] \leftarrow t_0 + t_1$	
77	$n[4] \leftarrow t_0 + -1$	
76	$n[4] \leftarrow n[4] + -1$	
21		$n[5] \leftarrow t_{10}$
20		$n[5] \leftarrow 0$
17	$n[4] \leftarrow t_3$	
16		
15	$n[4] \leftarrow t_1 + t_2$	
14	$n[4] \leftarrow t_1 + -1$	
13	$n[4] \leftarrow n[0] + -1$	
12		
11		

- <u>L3</u> $= \{n[4] \leftarrow n[0] - 1,\, n[5] \leftarrow 0\}$, or <u>L3</u> $= \{n[4] \leftarrow n[4] - 1,\, n[5] \leftarrow 0\}$
- <u>L4</u> $= \{n[5] \leftarrow n[5] + 1\}$.

From this information we can if N is the first parameter to the function that the inner loop is executed $N - 1$ times the first time it's entered and $N - n[4] - 1$ subsequent times. From this we see it's executed $(N - 1)/2$ times.

4.7 Cost Analysis

In this section we show how the methods described in the previous sections of the chapter can be generalized to allow us to bound the cost of the function. We begin with some terminology. When we use the phrase "EBB path" we mean an execution path that's made up of extended basic blocks. By "non-looping EBB paths" we mean paths that contain no loops. The EBBs that make up the path may contain loops, but at the level of the path there are no loops.

Similarly a "BB path" is an execution path that's made up of basic blocks and a "non-looping BB path" is a path with no loops..

4.7.1 Notation and Cost Equations

We introduce some notation:

- Use f to denote a function, X to denote the inputs to the function, ep for an EBB path, e and ec an EBB, bp a BB path and b a BB.

- Let $E(e)$ be the set of all non-looping EBB paths of EBBs in e that start at the first EBB of e and exit e. Two points must be made about this definition: we only consider EBB paths through e ignoring loops, and we only consider paths made up of EBBs that are contained directly in e. Not those contained in EBBs contained in e. $E(e)$ contains only paths that are non-looping and only paths that are one level below e. We extend this notation to let $E(f)$ denote all non-looping EBB paths through the function f.

- Let $nE(ep)$ be the set of EBBs without a loop that are on the EBB path ep.

- Let $lE(ep)$ be the set of EBBs with a loop that are on the EBB path ep

- For a looping EBB e, let $N(e, X)$ be a bound on the number of times the loop will iterate given function inputs X.

- Let $B(e)$ be the set of all non-looping BB paths of the EBB e.

With this terminology we can state the following inequalities for bounding the cost of a function:

$$cost_F(f, X) \leq \max_{ep \in E(f)} cost_{pE}(ep, f, X) \tag{4.1}$$

$$cost_{pE}(ep, ec, X) \leq \sum_{e \in nE(ep)} cost_E(e) + \sum_{e \in lE(ep)} [\max_{ep' \in E(ec)} cost_{pE}(ep', e, X)] * N(e, X) \tag{4.2}$$

$$cost_E(e) \leq \max_{bp \in B(e)} cost_{pB}(bp) \tag{4.3}$$

$$cost_{pB}(bp) = \sum_{b \in bp} cost_B(b) \tag{4.4}$$

$$cost_B(b) = \text{number of instructions in BB } b \tag{4.5}$$

To disambiguate the various cost functions we use a subscript on the function name. These subscripts are F for the cost of the function, pE for the cost of an EBB path, E for an EBB, pB for a BB path and B for a BB. The equations can be summarized as follows:

- 4.1 The cost of a function is bounded by the maximum of the cost of all non-looping EBB paths through that the function.

- 4.2 The cost of an EBB path is bounded by the sum of the cost of each non-looping EBB in the path plus, inductively, the cost of each of the looping EBBs times the number of iterations in the loop.

- 4.3 The cost of an EBB is bounded by the maximum cost of all BB paths of the EBB.

- 4.4 The cost of a BB path is the sum of the cost of the basic blocks in the path.

- 4.5 The cost of a BB is the number of instructions in the BB.

4.7.2 Bubble Sort Example

Throughout this section we've shown how to determine the basic blocks and extended basic blocks of a function, how to enumerate the paths in a function and how to bound the number of iterations of a loop. With these methods and the equations 4.1 through 4.5 we can produce a bound for the cost of a function.

We begin with equation 4.1. For this equation we need to know X, the inputs to the function and $E(f)$, the set of all non-looping EBB paths through the function. We can write $X = [N, A]$ where N is an integer representing the number of items to be sorted and A is the list of items to be sorted. We can determine $E(f)$ from the extended basic block control flow graph in Figure 4.2. We see that there are 2 non-looping paths that meet the criteria of the definition of $E(f)$:

- Path 1: [0 ... 2], [15]

- Path 2: [0 ... 2], [3 ... 13], [14], [15]

Notice that we're using the requirement from the definition that paths in $E(f)$ be at the top level of f and that we don't, at this point, use any of the lower level EBBs contained in the EBB [3 ... 13] to construct our paths. Equation 4.1 tells us that the cost of the function is cost of the max-cost path from these 2 paths.

In the following sections we walk through the cost calculations for each of these paths.

Path 1

We compute the cost of the paths using equation 4.2 to 4.5. We begin with path 1 and apply equation 4.2. This equation has two parts: the cost of EBBs that don't contain a loop and the cost of those that do. Since the first path doesn't contain any loops we can ignore the second part of the equation and arrive at this inequality for its cost:

$$cost_{pE}([0...2],[15]) \leq cost_E([0...2]) + cost_E([15...15])$$

We overload the use of *cost* as the name for these functions but the meaning should be clear from the arguments. On the left-hand side of the inequality we have the cost of a path and a cost of EBBs on the right hand side.

Applying equation 4.3 to simplify this requires that we enumerate the BB paths through the EBBs [0 ... 2] and [15]. In each of these cases there's only one path and so we can apply equation 4.4 to get:

$$cost_E([0...2]) \leq cost_{pB}([0,1])$$

$$cost_E([15...15]) \leq cost_{pB}([15])$$

Finally we calculate the cost of each of these BB paths using equation 4.4 to get

$$cost_{pB}([0,1]) = cost_B(0) + cost_B(1) = 1 + 4 = 5$$

$$cost_{pB}([15]) = cost_B(15) = 1$$

This gives us the final cost result for Path 1 of

$$cost_{pE}([0...2],[15]) \leq 5 + 1 = 6$$

Path 2

We follow a similar procedure for Path 2 with the added complication that it has an EBB that, in turn contains another EBB. This means that we'll need to apply both parts of equation 4.2. Applying equation 4.2 gives us

$$cost_{pE}([0...2],[3...13],[14],[15]) <= cost_E([0...2]) + cost_E([14]) + cost_E([15])$$

$$+ max_{ep' \in E([3...13])}(cost_{pE}(ep',[3...13],X) * N([3...13],X))$$

To simplify this we need to calculate each of the $cost_E$ values and to determine all the paths through the EBB [3 ... 13] and choose the maximum cost one. We see that $cost_E([0...2]) + cost_E([14]) + cost_E([15]) \leq 7$ similar to what we saw for Path 1 with the addition of $cost_E[15]$. For the paths through EBB[3 ... 13] we see that there are 2: [3 ... 5], [6 ... 10], [11], [12 ... 13] and [3 ... 5], [12 ... 13]. Since the first is a superset of the second we know that it will have the maximum cost. With these simplifications we arrive at:

$$cost_{pE}([0...2],[3...13],[14],[15]) <= 7$$

$$+ (cost_{pE}([[3...5],[6...10],[11],[12...13]],[3...13],X) * N([3...13],X))$$

From the analysis that was done in Section 4.6.2 we know that $N([3...13],X) \leq N-1$ where N is the number items to be sorted. We then recursively apply equation 4.2 to give:

$$cost_{pE}([0...2],[3...13],[14],[15]) <= 7+(N-1)*(cost_E([3...5])+cost_E([11)]+cost_E([12...13])$$

$$+ cost_{pE}([[6...10]], [6...10]], X) * N([6...10], X))$$

We can simplify this as follows:

- $cost_E([3...5]) = cost_B(3) + cost_B(4) + cost_B(5) = 3 + 8 + 1 = 12$
- $cost_E([11]) = cost_B(11) = 1$
- $cost_E([12...13]) = cost_B(12) + cost_B(13) = 11 + 1 = 12$
- $cost_{pE}([[6...10]], [6...10]], X) = cost_E([6...10]) = cost_B(6) + cost_B(7) + cost_B(8) + cost_B(9) + cost_B(10) = 1 + 21 + 7 + 4 + 1 = 34$
- $N([6...10], X) <= (N - 1)/2$ (from the previous section)

Putting this all together we get

$$cost_{pE}([0...2], [3...13], [14], [15]) <= 7 + (25 + (N - 1) * 34/2) * (N - 1)$$

Since this is clearly greater than the cost bound of Path 1 we can see that it's a bound on the cost of the function, i.e.

$$cost_F(f) <= 17N^2 - 9N - 1$$

This bound agrees with the well-known fact (see, for example, Edjal [22]) that the bubble sort algorithm is $O(n^2)$ in time complexity.

4.8 Summary

In this section we have described in detail the methods we use to analyze a WebAssembly function in order to determine a bound on its execution cost. We have

used Bubble Sort as a motivating example throughout and have, using these methods, shown how each of the parts of this analysis contribute to determining the bound and then synthesized a bound from these parts. In the next chapter we describe, in detail, the algorithms that have been implemented in OCaml to perform these analyses.

Chapter 5

WANALYZE Implementation

The sections of this chapter are organized as follows. In Section 5.1, Internal Representation, we provide a detailed description of the format of the binary WebAssembly file (typically a .wasm file). Section 5.2 describes how the code that we've written processes a file in this format. In subsequent sections we describe in detail how the ideas described in the previous chapter have been implemented in WANALYZE. This implementation consists of approximately 5,000 line of OCAML code that was developed over a period of about 10 months in 2021 and 2022. It's available on GitHub [37] as an open-source project with an nonrestrictive licensing requirement.

5.1 Internal Representation

To facilitate the analysis of the functions contained in a WebAssembly module we begin by creating an internal representation of the pertinent elements of the module. The starting point for this is the binary format (i.e .wasm file) that's described in Chapter 5 of the WebAssembly [75] specification. This binary format consists of a sequence of variable length sections each of which contains specific attributes of the

module. Table 5.1 contains a description of the contents of each of these sections. The sections appear in the binary file in the order given in the table and the order has been chosen to ensure that no forward references (to functions or data objects) are required in a module. Each section, with the exception of the start and data count sections, contains a vector of homogeneous elements of the type indicated by the section name. Generally elements in these vectors are accessed by an integer index into the vector.

The contents of each section and the order in which they appear in the binary module definition is as follows. Throughout these descriptions we refer to the "runtime environment". This is the execution time environment that hosts the WebAssembly module as described in Section 2.1.2.

Table 5.1: WebAssembly module sections

Name	Contents
Function Types	Function type signatures
Imports	Imported functions, tables, memories, variables
Functions	Types of functions implemented in this module
Tables	Function indexes used for indirect calls
Memories	Linear memory size and limits
Globals	Global variables
Exports	Functions, tables, memories, variable exported by this module
Start	The function to be invoked when the module is instantiated
Elements	Static data used for table initialization
Data Count	The number of data segments in the module
Code	The byte code for each function
Data	Static data used by the function

Function Types: The set of distinct function type signatures used either by imported functions or by functions defined in this module. A function type signature is, in turn, made up of two vectors that, respectively, specify the types of the function parameters and the types of the function outputs. A WebAssembly function can have multiple return values.

Imports: A WebAssembly module can require that the runtime environment provide it with functions, tables, memories or globals. This section contains the definition of these required import. This section contains the vector of descriptors of each of the required import items.

Functions: This section defines the type signature of each function defined in the module. Entries consist of an index into the Function Types section for each function.

Tables: A table contains a list of function indexes that can be used to call a function indirectly.

Memories: In WebAssembly memory is modeled as a linear array of addressable locations. This section allows for multiple memories to be defined but currently implementations are restricted to one memory. The definition includes the minimum and maximum number of pages for the memory.

Globals: This section defines the type and mutability of the global variables defined by this module.

Exports: A module may export functions, tables, memories and globals to the runtime environment. This section consists of a list of those items and their names.

Start: The function index of the function to be called when the module is loaded.

Elements: A vector of elements that are used to initialise tables.

Data Count: The number of items in the data section. Since this occurs before the code section it can be used to validate operations in the code section that operate on the data section.

Code: A vector that contains the WebAssembly code for each of the functions implemented in the module.

Data: Static, initialized, immutable data that's used by the module.

5.2 WebAssembly Module Reader

This chapter describes the OCaml code that we've written to read the WebAssembly binary file format. Since a single file contains a single WebAssembly module we term this code the module reader. The purpose of the module reader is to parse the binary WebAssembly module format and create an initial internal representation of the module that's suitable for further analysis. This is achieved by implementing a recursive descent parser that calls reader methods for each distinct object type that occurs in the binary file. The structure of the parser is derived directly from the binary file production rules found in Chapter 5 of the WebAssembly specification [76]. These reader methods correspond directly to the production rules and produce a wide range of objects, from an entire section down to a single byte, from the binary file. The production rules describe a hierarchical structure and the overall structure of the parser mirrors this hierarchical structure. The parser is simplified by the fact that the module structure does not require look-ahead. Any information required to parse a specific object successfully is found in the binary file before or when it's required. Additionally the length of a variable length object is always known when the object is read. So peeking forward to determine where an object ends is not required. These facts ensure that WebAssembly can be efficiently parsed at runtime. Since the module reader makes one linear pass through the WebAssembly binary it runs in $O(n)$ time where n is the size of the binary file. Further, since the initial internal representation uses the same data types as the binary file and there's no compression of objects in the binary file, the storage required to create the internal representation is also $O(n)$.

As an example, consider the production rules for the type section (the first in the module). This section defines the possible type signatures that a function in the module can have. A type signature is made up of the definitions of the data types

of the inputs to the function and those of the outputs of a function. Functions can return more than one value.

These production rules consist of three parts: the left-hand side which names the entity being defined, a middle section that defines the syntax (in the binary file), of the entity being defined, and the right-hand side which defines the values of the attributes of the entity being produced. The left-hand side is terminated by the ::= symbol and the right-hand side begins after the ⇒ symbol.

In the middle section, the definition section, variables that are created by the production rule appear in italics. A colon separates the name of a variable from the definition of the value that's assigned to it. Variables are used in the right-hand side of the rule when they are part of (or all of) the entity that's produced by the rule. Byte values that literally appear in the binary file as fixed values are represented in hexadecimal form $0xnn$. A fixed width font, for example `numtype`, is used for previously defined syntactic entities in the middle section.

These are taken verbatim from the WebAssembly specification:

$$reftype ::= \texttt{0x70} \Rightarrow \mathsf{funcref} \mid \texttt{0x6F} \Rightarrow \mathsf{externref}$$

$$numtype ::= \texttt{0x7F} \Rightarrow \mathsf{i32} \mid \texttt{0x7E} \Rightarrow \mathsf{i64} \mid \texttt{0x7D} \Rightarrow \mathsf{f32} \mid \texttt{0x7C} \Rightarrow \mathsf{f64}$$

$$valtype ::= t{:}\texttt{numtype} \Rightarrow t \mid t{:}\texttt{reftype} \Rightarrow t$$

$$vec(\texttt{B}) ::= n{:}\texttt{u32}\ (x{:}\texttt{B})^n \Rightarrow x^n$$

$$resulttype ::= t^*{:}\texttt{vec(valtype)} \Rightarrow [t^*]$$

$$functype ::= \texttt{0x60}\ rt_1{:}\texttt{resulttype}\ rt_2{:}\texttt{resulttype} \Rightarrow rt_1 \rightarrow rt_2$$

$$section_{\texttt{N}}(\texttt{B}) ::= N{:}\texttt{byte}\ size{:}\texttt{u32}\ cont{:}\texttt{B} \Rightarrow cont$$

$$typesec ::= ft^*{:}\texttt{section}_1\texttt{(vec(functype))} \Rightarrow ft^*$$

In describing these rules we give an operational reading from the point of view of

the parser. In this case the middle section is the input and the right-hand side is the output. With this in mind we can read each of these production rules as follows:

reftype: *reftype* is an attribute that's encoded as either the byte value $0x70$ for funcref or $0x6F$ for externref.

numtype: *numtype* is an attribute that encodes one of the numeric types i32, i64, f32 or f64.

valtype: *valtype* is either a numtype or a reftype. This rule stores the value of the attribute in the variable t and produces it as the output of the rule.

vec: This is an example of a parameterized production rule. Its purpose is to recognize the contents of a homogeneous vector whose values are produced by the production rule B. The rule stores the length of the vector in the variable n and the contents of the vector in variable x and produces it as a vector of the values of the n items in x.

resulttype: This uses the *vec* and *valtype* production rules to initialize the variable t^* which is a vector of the *valtype* attributes. The result, $[t^*]$, represents those attribute values in list form. Note that *valtype* can be either *numtype* or *reftype* and so list is non-homogeneous. It can contain values of either type.

functype: A *functype* consists of a byte containing 0x60 (which can be discarded) followed by two *resulttype*s. These are stored in the variable $rt1$ and $rt2$. The rule produces a function signature based on these two variables.

section: This is the parameterized production rule is used to produce the content of section N with that content being of produced by the rule B. The section number is stored in the variable N, its size in $size$ and the content in $cont$. The rule uses the variable $cont$ to produce the content.

typesec: A *typesec* is found in the module section at index 1 and consists of a vector of functypes. This is the rule that produces the content of the first section of a WebAssembly module.

In the OCaml implementation of the WebAssembly binary reader, a parser function is implemented for each production rule (i.e. the left-hand side of the rule) that implements the rule by reading bytes from the binary input, recognizing any constant values that the rule specifies (such as the $0x70$ or $0x6F$ value in the $reftype$ rule),and invoking additional parser functions that the right-hand side of the rule defines. This is done in the manner of a recursive descent parser whose recursion terminates with rules that use no other rules.

The final output of the binary reader is a set of populated OCaml data structures. For the example presented, the type section, the data structure and the associated OCaml type definitions are those shown in Figure 5.1. A similar hierarchy of type definitions exists for each of the other module sections. Some basic type definitions such as `numtype` are used in multiple section definitions.

The entire module is encapsulated in the OCaml type shown in Figure 5.2. In this definition most fields are marked as mutable since they are updated as the associated data is read from the binary module. Each of the `*_section` fields corresponds directly to a section found in the binary module.

5.3 Basic Blocks

From Definition 3 in Chapter 4 for WebAssembly basic blocks we can now describe the algorithm to produce the list of basic blocks for a WebAssembly function. It's a straightforward process of reading the instructions (and their operands) that make up the function one by one, and emitting a basic block whenever we encounter an instruction that's in the list of basic blocks terminators specified in the definition. This is detailed in Algorithm 1. Since this algorithm processes each instruction once it runs in $O(n)$ time where n is the number of instructions in the function.

```
type numtype = | I32 | I64 | F32 | F64

type reftype = | Funcref | Externref

type valtype =
  | Numtype of numtype
  | Reftype of reftype

type resulttype = valtype

type functype =
{
  rt1:    resulttype list;
  rt2:    resulttype list;
}

type wm (* wasm module *) =
{
  mutable type_section:     functype list;
  (* ... other section definitions *)
}
```

Figure 5.1: OCaml type definitions for the WebAssembly module type section

```
type wm (* wasm Module *) =
{
  module_name:               string;
  mutable data_count:        int;
  mutable type_section:      functype list;
  mutable import_section:    import list;
  mutable function_section:  typeidx list;
  mutable table_section:     tabletype list;
  mutable memory_section:    memtype list;
  mutable global_section:    global list;
  mutable export_section:    export list;
  mutable start_section:     funcidx option;
  mutable element_section:   element list;
  mutable code_section:      func list;
  mutable data_section:      data list;
  mutable last_import_func:  funcidx;
  mutable next_global:       int;
  mutable next_memory:       int;
  mutable next_table:        int;
  mutable next_data:         int;
}
```

Figure 5.2: OCaml type definitions for the WebAssembly module

Algorithm 1 Create basic blocks given function code, terminating instructions

```
1: procedure GetBasicBlocks(code, tinstrs)
2:     // process each instruction in the code one by one
3:     bbs ← []
4:     bb ← []
5:     for each instr in code do
6:         bb ← bb :: instr
7:         // if it's a control instruction then finish this BB and start a new one
8:         if instr in tinstrs then
9:             bbs ← bbs :: bb
10:            bb ← []
11:    return bbs
```

5.4 Extended Basic Blocks

Algorithm 2 shows the process for constructing a list of extended basic blocks for a WebAssembly function that takes as input the list of basic blocks for that function. It scans this list accumulating a list of basic blocks that will make up an extended basic block. This is the current EBB in the algorithm. The algorithm also keeps track of whether a loop has been encountered. It only processes loops at its current level and defers processing of nested loops to a recursive invocation of itself.

An extended basic block is emitted when a basic block satisfying one of these conditions is encountered:

- EBB1 The BB ends the current loop. In this case the procedure is called recursively on the loop EBB that's emitted.
- EBB2 It has a predecessor that's not in the current EBB. We start a new EBB in this case so that the flow control into it can be represented correctly in a control flow diagram.
- EBB3 It begins a loop and we're not currently processing a loop. We determine this by checking if the BB ends in a `loop` instruction and by keeping a flag that

states whether we're currently processing a loop or not.

- EBB4 It's the last basic block of the function.

These conditions are referred to in comments in Algorithm 2 to make it clear how they relate to the algorithm. EBB1 is recognized when we process the basic block that completes a loop. This basic block is an `end` basic block that's at the same level of nesting as the `loop` instruction at the beginning of the loop. EBB2 is recognized when we process a basic block that has a predecessor that's not in the extended basic block we're currently building. EBB3 is recognized when we start a loop and EBB4 is recognized after we process the last basic block.

Each basic block can be processed more than once by this algorithm. It's processed once for each loop that it's contained in. For example, a BB in an inner loop that's nested 3 levels deep will be processed 3 times. We can, therefore say that this algorithm is $O(N * M)$ where N is the number of basic blocks and M is the maximum loop nesting. In practice N is usually much larger than M and the algorithm runs in close to linear time.

Algorithm 2 Get extended basic blocks given the basic blocks of a function

```
 1: procedure GetEBBs(bbs)
 2:     ebbs ← [], current_ebb ← [], in_loop ← false
 3:     // process each basic block in turn
 4:     for each bb in bbs do
 5:         if in_loop then
 6:             // EBB1 does this BB end the loop?
 7:             // bb ends the loop if it's an end at the same nesting level
 8:             // of the current loop
 9:             if bb is end of loop then
10:                 // yes, finish this EBB
11:                 ebbs ← ebbs :: current_ebb
12:                 // process this ebb recursively since its the entire body of the loop
13:                 ebbs ← ebbs :: GetEBBs(current_ebb)
14:                 current_ebb ← [], in_loop ← false
15:             else
16:                 current_ebb ← current_ebb :: bb
17:         else
18:             // does this BB have a pred not in the EBB we're building?
19:             if preds(bb) − current_ebb is not empty then
20:                 // EBB2 yes, put it in a new EBB
21:                 ebbs ← ebbs :: current_ebb
22:                 current_ebb ← [bb]
23:             else
24:                 current_ebb ← current_ebb :: bb
25:             // is this BB starting a loop? i.e. is loop the last instruction?
26:             // in the BB
27:             if bb is loop then
28:                 // EBB3 yes, start a new EBB for the loop
29:                 ebbs ← ebbs :: current_ebb
30:                 current_ebb ← [], in_loop ← true
31:     // EBB4 is there an EBB to emit?
32:     if current_ebb ≠ [] then
33:         ebbs ← ebbs :: current_ebb
34:     return ebbs
```

5.5 Control Flow Graphs

Algorithm 1 shows how to build a basic block as a list of instructions. Similarly Algorithm 2 show how to build an extended basic block as a list of basic blocks. In

both cases we also would like to understand how these BBs or EBBs are connected as in a directed graph that represents the flow of control from one BB or EBB to another. This allows us to produce Control Flow Graphs as useful diagrams of a function and, more importantly, to be able to analyze the flow control within the function at the level of BBs or EBBs. Focusing on basic blocks, we need to determine the *successor* relationship between BBs. That its, we need to know to which BBs execution can flow from a given BB. The same ideas apply to EBBs where we'd like to know the successors to be able to understand execution flow control at the level of EBBs.

For both BBs and EBBs the fact that branching to the top of a loop is possible means that a BB or EBB that occurs later in the list can have a successor that occurs earlier in the list. This causes us to implement the process of determining successors to be one that is applied in a second pass after the list of BBs or EBBs has been created.

For BBs this pass consists of looking at the instruction at the end of the BB, the instruction that caused the BB to be created, and determining the successors of the BB based on that instruction. Table 5.2 lists all the possible instructions that can terminate a BB and the successors that this kind of BB will. Generally there are a 4 possibilities: none, the next BB, a BB that's a branch target, or the BB after the `end` BB corresponding to an `else` BB.

Table 5.2: Basic block successors

Terminating Instruction(s)	Successor(s)
`block, loop, else`	next BB
`else`	next BB of corresponding end BB
`if`	next BB, else BB
`br_if`	next BB, branch target BB
`br, br_table`	branch target BB(s)
`unreachable`	none
`return`	none

The successor relationship for EBBs can be built from the successors of BBs. Observe that a BB in a given EBB may have successors that aren't contained in that EBB. The EBB that contains that successor (every BB is in some EBB) is a successor to the given EBB. This can be implemented using OCaml list operations that set union and minus operations on lists of BBs. Algorithm 3 outlines this.

Algorithm 3 Create successor list for an extended basic block

```
1:  procedure GetSuccOfEBB(ebb, ebbs)
2:      succ_bbs ← [], succ_ebbs ← []
3:      // get all the BB successors of all the BBs of ebb
4:      for each bb in ebb do
5:          succ_bbs ← bbs ∪ successors(bb)
6:      // remove those that are in ebb
7:      succ_bbs ← succ_bbs − ebb
8:      // get the EBBs of those BBs
9:      for each bb in succ_bbs do
10:         // EbbOfBB returns the EBB that contains a given BB
11:         succ_ebbs ← succ_ebbs :: EbbOfBB(bb, ebbs)
12:     return succ_ebbs
```

Figures 5.5 and 5.5 contain the OCaml type definitions for the data structures that are created by a combination of the these algorithms and the algorithms that create BBs and EBBs. Fields in these types that are marked mutable are those whose contents are determined in a second pass through the list BBs or EBBs.

The basic blocks type contains the index of the basic block, the start and end indexes of the code of the basic block, the terminating instruction of the basic block, the successors, and for convenience the predecessors, the nesting level, indexes to the labels that the basic block creates, and optionally a reference to the basic block that that's the branch target of this basic block.

The extended basic blocks type has fields for the type of EBB (loop or non-loop), the first BB of the EBB, the list of BBs in the EBB, the successors of the EBB, and the nested EBBs in the given EBB.

```
type bb_type =
  BB_unknown
  | BB_exit_end | BB_exit_return | BB_exit_unreachable
  | BB_unreachable | BB_block | BB_loop | BB_if | BB_else
  | BB_end | BB_br | BB_br_if | BB_br_table | BB_return

type bb =
{
    (* index of this bb in the list of bblocks *)
    bbindex:  int;
    (* index into e of the first op in the expr *)
    start_op: int;
    (* index+1 of the last op in the expr *)
    mutable end_op:   int;
    (* bbs that can be directly reached by this bb *)
    mutable succ:     bb list;
    (* bbs that can directly reach this bb *)
    mutable pred:     bb list;
    (* effectively the control opcode that created this bb *)
    mutable bbtype:   bb_type;
    (* nesting level of the last opcode in the bb *)
    mutable nesting:  int;
    (* destination labels used in BR, BR_IF, BR_TABLE
            instructions *)
    mutable labels:   int list;
    (* the basic block of a label created by this
       basic bblock *)
    mutable br_dest:  bb option;
}
```

Figure 5.3: OCaml type definitions WebAssembly basic blocks

```
type ebb_type = EBB_loop | EBB_block

type ebb =
{
    (* either a block or a loop*)
    ebbtype:      ebb_type;
    (* bb that's the entry to the ebb *)
    entry_bb:     bb;
    (* list of bbs that make up the ebb *)
    bblocks:      bb list;
    (* list of ebblocks directly reachable from this one*)
    mutable succ_ebbs:    ebb list;
    (* ebbs containing nested loops *)
    nested_ebbs:  ebb list;
  }
```

Figure 5.4: OCaml type definitions WebAssembly extended basic blocks

5.6 Non-looping Extended Basic Block Algorithms

In Algorithm 2 we have shown how to create a list of extended basic blocks from a list of basic blocks. In doing so we create two kinds of EBBs: those with loops and those with no loops. In this section we focus on non-looping EBBs and describe algorithms that enumerate all BB paths through them and that determine the maximum cost BB path through them. These algorithms are based on the results of Algorithm 3 were we show how to create a control flow graph from a list of extended basic blocks by determining the successors of a given EBB. The approach used in the algorithm for finding all paths is based on following successors to enumerate them. The algorithm to find the maximum cost path through a non-looping EBB uses the fact that the cost of a BB in that EBB is the number of instructions in the BB.

Algorithm 4 consists of two procedures. The first, GetBBPathsOfEBB, takes an extended basic block as its input and calls the second procedure, GetPaths, with

an initial set of values created from the first BB in the EBB. This procedure then recursively processes the successors of each BB in the EBB to create the paths. The recursion terminates under two conditions: either it reaches a successor that's not in the EBB (this terminates the path) or it has processed all successors for the given BB. The worst case run time of this algorithm is $O(N * S)$ where N is the number of basic blocks and S is the number of successors. However, since this algorithm produces all paths its worst case memory usage is exponential in the number of basic blocks in the extended basic block, i.e. it's $O(2^N)$ where N is the number of basic blocks. This worst case memory usage occurs when the EBB contains a preponderance of basic blocks that have multiple successors. In Table 5.2 we identified these as basic blocks terminated by `if,` `br_if` or `br_table` instructions.

Algorithm 4 Generate BB paths of EBB

1: **procedure** GetBBPathsOfEBB(ebb)
2: // Root(ebb) returns the first BB of the EBB ebb
3: $paths \leftarrow GetPaths(ebb, Root(ebb), [Root(ebb)], [])$
4: **return** $paths$
5: **procedure** GetPaths($ebb, bb, path, paths$)
6: // process each BB that's a successor of the given BB
7: **for** each bb' in $Successors(bb)$ **do**
8: // is the successor is in the given ebb?
9: **if** bb' in ebb **then**
10: // yes, get the paths from successors,
11: // add them to the paths found so far
12: // + is the list append operator
13: $paths \leftarrow GetPaths(ebb, bb', path + [bb'], paths) + paths$
14: **else**
15: // no, add the path to the list of paths
16: $paths \leftarrow path :: paths$
17: // in the case where a BB has multiple successors that exit the EBB
18: // we'll have duplicate paths, so we de-duplicate the list
19: **return** DeDup($paths$)

Algorithm 5 also consists of two procedures and the pattern is similar to that of Algorithm 4: create an initial set of data based on the first BB in the BB and

then recursively iterate through successive successors. Since this algorithm's goal is to find the maximum cost path in an EBB, and its cost we initialize two arrays, *max_path* and *cost*, to hold our intermediate results. These arrays contain one entry for every BB in the EBB; the BB numbers are the indices in the arrays. The procedure GetCost updates these arrays as we find longer paths to a given array. The recursion terminates under the same circumstances as Algorithm 5: either we reach a successor that's not in the EBB or we process all successors. As with the previous algorithm, the worst case run time for this algorithm is $O(N * S)$ where N is the number of basic blocks and S is the number of successors. However the memory usage is linear, $O(N)$, in the number of basic blocks since we only store information for each basic block.

Algorithm 5 Find the maximum cost path through an EBB

```
 1: procedure GetMaxCostPathOfEBB(ebb)
 2:     // initialise cost of the path to each BB in our EBB to zero
 3:     // and the max cost path to each BB to be empty
 4:     for each bb in ebb do
 5:         cost[bb] ← 0, max_path[bb] ← []
 6:     // initialise the cost and its path for the root BB
 7:     // [the Cost function returns the cost of a BB, i.e. the number
 8:     // of instructions in it]
 9:     // Root(ebb) returns the first BB of the EBB ebb
10:     cost[Root(ebb)] ← Cost(bb), max_path[Root(ebb)] ← [Root(ebb)]
11:     // process successor BBs
12:     cost, max_path ← GetCost(ebb, Root(ebb), cost, max_path)
13:     // scan the costs to find the maximum
14:     max_path_bb ← bb of max(cost)
15:     return cost[max_path_bb], path[max_path_bb]
16: procedure GetCost(ebb, bb, cost, max_path)
17:     // process each BB that's a successor of the given BB
18:     for each bb′ in Successors(bb) do
19:         // is this successor in the EBB?
20:         if bb′ in ebb then
21:             // yes,
22:             // does it make a longer path than what we currently know?
23:             if cost[bb] + Cost(bb′) > cost[bb′] then
24:                 // yes, update the cost and make it the longest path to bb'
25:                 cost[bb′] ← cost[bb] + Cost(bb′)
26:                 max_path[bb′] ← max_path[bb] + [bb′]
27:         // recursively process the successor
28:         cost, max_path ← GetCost(ebb, bb′, cost, max_path)
29:     return cost, max_path
```

5.7 Symbolic Execution

Symbolic execution of WebAssembly code consists of processing a fragment of code instruction by instruction to create an expression that represents the result of that instruction while updating the virtual machine state appropriately with this expressions. This means that the virtual machine state (including the contents of the value stack, memory and variables) will be represented by the expressions that are produced

by symbolic execution.

An instruction may query the virtual machine state and perform an operation that depends on that state. Here are five examples of instructions that operate on the virtual machine state in different ways. In this description "item" refers to either a value or expression in the virtual machine state.

- The `add` instruction removes the first two items from the value stack, forms an expression for their sum and places this expression on the value stack.
- The `local.put` instruction removes the first two items from the value stack. It stores the first of these in the local variable indexed by the second.
- The `global.get` instruction removes the first item from the value stack. It gets the expression for the value of the global variable indexed by this item and puts it on the value stack.
- The `load` instruction removes the item at the top of the value stack and uses as an index into the virtual machine's linear memory to retrieve an item and place it on the value stack.
- The `br_table` instruction removes the item at the top of the value stack and uses it as an index into a table of labels to determine the location of the next instruction to be executed.

From these examples we can see that the virtual machine state must include the location of the next instruction to be executed, the value stack, local variables, global variables, and the linear memory. In fact these items are sufficient to determine the WebAssembly virtual machine state for the purpose of symbolic execution.

In order to be able to symbolically execute the instructions of any WebAssembly module it's necessary to be able to symbolically execute any WebAssembly instruction. To streamline this task we use the organization of groups and sub-groups of

instructions that the WebAssembly specification defines. The purpose of doing so in the specification is to reduce the amount of repetition that would be required if each instruction were defined individually. We can use this to organize the implementation so that it doesn't require specific code for every instruction. This re-use has the effect of making the code more reliable. Table 5.3 summarizes these groups and sub-groups. The largest group is the Numeric instructions. Instructions in this group operate on the value stack. The sub-groups provide the ability to define constants, perform unary and binary operations, test, compare and perform conversion using the value stack. Variable instructions provide the ability to set and get both local and variables from and to the value stack. Memory instructions provide the ability to load and store memory values to and from the value stack and, in addition, to control the size of memory, copy, and fill it. Control instructions can alter the normal flow of execution and can use values on the value stack to do so. Parametric instructions provide the ability to remove a value from the value stack and to conditionally select an item on the values stack. Table instructions provide the ability to get and set the values in a table and to control the contents of a table by growing, filling and copying it. Finally, Reference instructions provide the ability to produce and check for null references and to produce a reference to a specific function.

With these definitions the implementation of Symbolic Execution in WANALYZE consists of decoding an instruction to determine its group and sub-group and, based on these, updating the virtual machine state accordingly.

5.8 Static Single-Assignment Form

The purpose of using Static Single-Assignment Form in WANALYZE is twofold: to determine the initial value of a variable upon entry to a loop and to produce the assignment statements that cause a variable to be modified in the body of a loop. In

Table 5.3: Instruction groups and sub-groups

Group	Sub-groups
Numeric	constant, unary, binary, test, compare, conversion
Variable	local, global, load, store
Memory	load, store, control
Control	conditional, unconditional
Parametric	drop, select
Table	get, set, control
Reference	null, function

both cases this is achieved by converting the list of basic blocks that make up the code paths of interest (i.e. the path to the loop entry or the path through the body of the loop) into a list SSA forms. These SSAs are then reduced to assignment statements by iteratively replacing temporary variables on the left-hand side of an assignment statement with the right-hand side of the SSA that initializes the temporary variable. The nature of SSA, that a temporary variable is never modified after it's initialized, makes this possible.

5.8.1 Generation

Generation of SSA from a list of basic blocks proceeds instruction by instruction through each basic block. The names of temporary variables include a sequential index and so we keep a counter of the next temporary variable's index that's updated every time a new temporary variable is created. Not all instructions cause SSA to be generated.

Table 5.4 breaks down how SSA generation is performed based on the group and sub-group of an instruction. Several points need to be made about the contents of this table:

- Temporary variables are stored and retrieved from the stack. The $\uparrow$ and $\downarrow$ operators are applied to the temporary variable to the right of the operator and indicator that the value is, respectively, pushed to or popped from the stack.

Table 5.4: SSA generation by instruction groups and sub-groups

Group	Sub-groups	SSA
Numeric	constant	$\downarrow t_i \leftarrow c$
	unary, test, cvt	$\downarrow t_i \leftarrow op(\uparrow t_j)$
	binary, compare	$\downarrow t_i \leftarrow op(\uparrow t_j, \uparrow t_k)$
Variable	local, `get`	$\downarrow t_i \leftarrow n[\uparrow t_j]$
	local, `set`	$n[\uparrow t_i] \leftarrow \uparrow t_j$
	local, `tee`	$temp \leftarrow \uparrow t_i$, $n[\uparrow t_j] \leftarrow temp$, $\downarrow t_k \leftarrow temp$
	global, `get`	$\downarrow t_i \leftarrow g[\uparrow t_j]$
	global, `set`	$g[\uparrow t_i] \leftarrow \uparrow t_j$
	global, `tee`	$temp \leftarrow \uparrow t_i$, $g[\uparrow t_j] \leftarrow temp$, $\downarrow t_k \leftarrow temp$
Memory	load	$\downarrow t_i \leftarrow m[\uparrow t_j]$
	store	$m[\uparrow t_i] \leftarrow \uparrow t_j$
	control	None
Control	`br_if`, `if`	$\uparrow t_i$
	`br_table`	$\uparrow t_i, \uparrow t_j$
	unconditional	None
Parametric	drop	$\uparrow t_i$
	select	$\downarrow t_i \leftarrow select(\uparrow t_j, \uparrow t_k, \uparrow t_l)$
Table	get	$\downarrow t_i \leftarrow table[\uparrow t_j]$
	set	$table[\uparrow t_i] \leftarrow \uparrow t_j$
	control `size, grow, fill, drop`	$\uparrow t_i$
	control `copy, init`	$\uparrow t_i, \uparrow t_j$
Reference	`ref.null`	$\downarrow t_i \leftarrow nullref$
	`ref.is_null`	$\downarrow t_i \leftarrow isnullref(\uparrow t_j)$
	`ref.func`	$\downarrow t_i \leftarrow funcref(\uparrow t_j)$

- Any variable that's pushed onto the stack is a newly created variable. The index given to this variable is determined by a counter that's incremented every time a new variable is created. Any variable that's popped from the stack is an existing variable.

- In most cases temporary variables popped from the stack are used in the right hand side of an assignment statement but in some cases they're used as an index on the left hand side. Also in some instructions, like the Parametric `drop` instruction they're simply popped from the stack and discarded.

- all variables (temporary, local, global, memory, or table) have an implicit type. The WebAssembly runtime statically verifies at module load time that all instruction operands are of the correct data type. This means that we can ignore variable types in the SSA that we generate because we know they will be correct for the operation that's being performed.

As each instruction is processed this table is used to determine the form that the SSA that's generated for it and the operations that are performed on the stack. This process can be performed on a single instruction, a basic block, or a list of basic blocks representing an execution path in a function.

5.8.2 OCaml Types

Figure 5.8.2 shows the OCaml type definitions for the Static Single Assignment form. The three main types in this definition are *var* which represents a variable, *et* which represent an expression and *ssa* which is the actual SSA for.

The type *var* has two attributes: the collection that the variable belongs to (e.g. local, global, temporary, or memory variables) and its index in that collection.

The type *et* is a recursively defined type that represents an expression. An expression can consist of constants, variables, or nodes that contain an operator and its

```
(* Variables *)
type var_type = Var_parameter | Var_local | Var_retvalue
        | Var_global | Var_temp | Var_memory
type var =
  {
    vtype:  var_type;
    idx:    int;
   }

(* Expression trees *)
type constant_value = Int_value of int | Int64_value of int64
      | Float_value of float | String_value of string
type et = Empty | Constant of constant_value
            | Variable of var
            | Node of node
      and node = { op: string; args: et list }

(* SSA *)
type ssa = {
  lhs:           var;
  mutable rhs:   et;
}
```

Figure 5.5: OCaml type definitions for SSA

operands which consist of other expressions.

Finally, the *ssa* type has two attributes: a variable which represent the left hand side of the form and an expression which represents the right hand side.

5.8.3 Simplification

We now can use SSA to determine the initial value of a variable upon entry to a loop, and to determine the assignment statements that represent how a variable changes in a loop. We start by generating a list of basic blocks that make up the execution path for which we'll generate SSA. In the case of the initial value of a variable upon entry to a loop, it's the code path to the loop entry. In the case of the assignment statements that represent how a variable changes in a loop, it's the code path through the body of the loop. Our objective is to simplify this SSA in such a way that we're left with only the assignment statements of interest (initial value, or updated values) with no temporary variables.

Algorithm 6 describes the steps for doing this. The main idea is that the SSA forms for the execution path are traversed *in reverse order*. Each of these forms is used to simplify the expression for the variable which is initially set to be empty. If the form's left-hand side is the variable of interest then the expression for the variable becomes the right hand side of the form. Otherwise the form is used to resolve any occurrences of its left hand side in the right hand side of the expression.

This algorithm will ultimately produce an expression for the target variable that contains only variables from the virtual machine state (i.e. locals, globals or memory) and no temporary variables. Furthermore the expression will be only need to know the virtual machine state at the beginning of the code execution path in order to evaluate the expression. As we'll see in subsequent sections this makes the expression derived in this manner useful for the further analysis of loops. We use it to simplify expressions for variables that are found in loop conditions.

Algorithm 6 Simplify the SSA for a given variable

```
1: procedure SimplifySSAOfVar(SSAs, var)
2:     // initialize return expression to empty
3:     expr ← ϵ
4:     // process the list of SSAs in reverse order
5:     for each ssa in Reverse(SSAs) do
6:         // apply the definition in this SSA
7:         expr ← ApplySSA(ssa, var, expr)
8:     return expr
9: procedure ApplySSA(ssa, var, expr)
10:    // does this ssa update our variable?
11:    if var == ssa.lhs then
12:        // yes, return the rhs of the SSA
13:        expr ← ssa.rhs
14:    else
15:        // no, do we have an expression to expand?
16:        if expr is Variable or Node then
17:            // yes, replace occurrences of the lhs of the ssa in expr
18:            // with the rhs of the ssa
19:            expr ← Replace(expr, ssa.lhs, ssa.rhs)
20:    return expr
```

5.9 Loop Analysis

As we described in Section 4.6 we need four things to do the loop analysis that we envision:

- L1 Conditions that cause the loop to exit.
- L2 Variables that are used in evaluating these conditions.
- L3 Initial values of those variables when the loop is entered.
- L4 How the variables are updated in the loop.

The algorithms that we described in Section 5.8 are the algorithms that we use to produce L3 and L4. In this section we describe the algorithms that are used to produce L1 and L2

5.9.1 L1 Loop Exit Conditions

We determine the exit conditions of the loop by examining all paths through the loop body. We create these paths as lists of basic blocks. A loop make have multiple paths through its body and a given path may have multiple loop exit conditions. These multiple exit conditions may or not be distinct. For example a loop that has two paths through its body may or may not exit with the same loop exit condition for each of these paths.

In Section 4.6.1 we outlined the steps required to create the loop exit conditions. We can now add details to how these steps are performed in WANALYZE based on the type and algorithm definitions contained in the earlier sections of the chapter. In particular we draw from the type definitions in Figures 5.5 (basic blocks), 5.5 (extended basic blocks), and 5.8.2 (SSA), the description of path generation algorithms from based on control flow graphs in Section 5.5, and the description of symbolic execution in Section 5.7.

The procedure to find the conditions that cause a loop is as follows:

1. Find all basic blocks in the extended basic block of the loop that are terminated by a `br, br_if,` or `br_table` instruction that either continues or exits the loop. This can be done by looping through the *bblocks* attribute of the *ebb* type and examining the *bbtype* attribute of each *bb*. The *br_dest* attribute of the basic block tells us if this continues or exits the loop based on whether the basic block referenced by *br_dest* precedes the basic block or not.

2. For the cases where a `br` instruction is used find the corresponding `if` basic block. This can be done by scanning the list of basic blocks in reverse order to find the `if` basic block that the `br` is immediately nested in. If there is none then this is an unconditional branch and doesn't need to be examined further.

3. Enumerate all code paths from the first basic block to any of the basic blocks terminated by these `br_if`, `br_table`, or `if` instructions. This can be done using the path generation algorithms described in section sec:control-flow-graphs.

4. Symbolically execute each of these code paths to determine the loop exit condition. Either it or its logical negation will be the item on the on the top of the value stack before the `br_if`, `br_table`, or `if` instruction is executed.

This procedure examines each possible path through the loop body that results in the loop terminating and so generates all possible loop termination conditions. The final step is to scan the list of loop exit conditions that was found and discard any duplicate conditions.

5.9.2 L2 Loop Condition Variables

Once we've determined the loop exit conditions we can examine them to determine what variables they reference. The symbolic execution process that generates these loop exit conditions uses the type *et* that we introduced in figure 5.8.2 as part of the SSA discussion. Algorithm 7 takes as input a loop exit conditions and returns a list of the variables that it references. In order to allow it to be used recursively it also takes the list of variables found so far. This should be empty the first time it's called. Since we use a set union operator to construct the list of variables there's no need to remove duplicates from this list.

Algorithm 7 Get the variables of a condition

```
1: procedure GetLoopVars(condition, vars)
2:     // is the condition simply a variable?
3:     if condition is Variable var then
4:         // yes, add it to the list of variables we've found
5:         vars ← vars ∪ var
6:         // is the condition an expression?
7:     else if condition is Node node then
8:         // yes, add the variables from the sub-expressions
9:         for each arg in node.args do
10:            vars ← vars ∪ GetLoopVars(arg, vars)
11:    return vars
```

5.10 Cost Analysis

This final section outlines each algorithm that's required to be able to compute the costs as defined by equations 4.1 to 4.5 in Section 4.7.1.

Starting with the simplest example, Algorithm 8 shows how to calculate the cost of a basic block. This algorithm is defined based on the OCaml type in Figures 5.5. Specifically it uses the $start_op$ and end_op attributes, which are indexes to the first and to one after the last instruction in the basic block. The cost of the basic block is simply the difference between these two indexes. This gives us a way to evaluate equation 4.5.

Algorithm 8 Get the cost of a BB

```
1: procedure GetCostOfBB(bb)
2:     // return the number of instructions in bb
3:     return bb.end_op − bb.start_op
```

Algorithm 9 is also quite simple given the analysis that we've done. Given a list of basic blocks that represents a code execution path it computes the cost of that path by summing the cost of each of the basic blocks in that path. These produces the result of equation 4.4.

Algorithm 9 Get the cost of a BB path

```
1: procedure GetCostOfBBPath(bbs)
2:     // initialize the cost
3:     cost ← 0
4:     for each bb in bbs do
5:         cost ← cost + GetCostOfBB(bb)
6:     return cost
```

To get the result of equation 4.3 we can use Algorithm 5 from Section 5.6 to produce all non-looping paths in an extended basic block and take the maximum of the costs for all those paths. Algorithm 10 shows this.

Algorithm 10 Get the non-looping cost of an EBB

```
1: procedure GetCostOfEBB(ebb)
2:     // get the paths through this ebb
3:     paths ← GetBBPathsOfEBB(ebb)
4:     // initialize the cost
5:     cost ← 0
6:     for each path in paths do
7:         new_cost = GetCostOfBBPath(path)
8:         if new_cost > cost then
9:             cost ← new_cost
10:    return cost
```

Equation 4.2 has two terms on the right hand side. The first term is the sum of the costs of all non-looping extended basic blocks and can be computed using equation 4.3 and therefore Algorithm 10.

The second term is the sum of the costs of the maximum cost paths through looping extended basic blocks multiplied by the number of iterations of the loop. This equation is recursively defined because a loop may contain loops and so the cost of the inner loops must be carried into the cost of the outer loops.

Algorithm 11 Get the cost of an EBB path

```
1: procedure GetCostOfEBBPath(ebbs, ebb, X)
2:     // initialize the cost
3:     cost ← 0
4:     for each ebb′ in ebbs do
5:         if ebb′ has no loop then
6:             cost ← cost + GetCostOfEBB(ebb′)
7:         else
8:             costs ← []
9:             for each ebbs′ in GetPathsOfEbb(ebb) do
10:                costs ← costs :: GetCostOfEBBPath(ebbs′, ebb, X) ∗ N(ebb, X)
11:            cost ← cost + max(costs)
12:    return cost
```

Finally, to calculate the cost of the function, i.e. the result of equation 4.1, we calculate the cost of each extended basic block path through the function, using the result of equation 4.2 and Algorithm 11.

Chapter 6

Test Results and Evaluation

Three sets of experiments were performed to analyze and measure the success rate that WANALYZE achieved on a variety of input programs. Success was defined as both being able to enumerate all paths through a function and being able to determine the number of iterations for all loops within the function. If both of these can be determined for a function then a bound on the cost of that function can be determined.

6.1 Bubble Sort

The focus of the first experiment was the bubble sort function that we've been using as a running example. We wanted to determine if WANALYZE could successfully produce a prediction for the number of instructions to be executed and to determine how accurate that prediction was when compared to actual measurements.

This experiment was run in two parts. The first part involved running the bubble sort WebAssembly code from Section 2.6.2 using the **wasmtime** [11] WebAssembly runtime. This runtime has the ability to instrument WebAssembly code to measure the amount of "fuel" the code consumes when executed. Conveniently it measures fuel

as the number of instructions executed with the exception of `nop, drop, block,` and and `loop` instructions. The design of WANALYZE allows instructions to be counted in the same way.

The bubble sort WebAssembly function takes as input a fixed length array of 32 bit integers and sorts them in place. It's well known that the performance of bubble sort is dependent on the composition of the data. Worst case performance occurs when the input data is ordered in the complete reverse of the sorted data. And so two different tests were run to compare results. The first test was with the data in this worst-case, reverse-sorted order. The second test was with the data in a random order. Ten runs with input arrays sized between 10,000 and 100,000 in increments of 1,000 were run. In order to run this WebAssembly code using the **wasmtime** runtime it was necessary to write "glue" code in Rust that creates the data. This code can be found in Section 6.1.1.

The second part of the experiment involved running WANALYZE an the WebAssembly bubble sort function. The analysis was successful and produced this polynomial:

$$17n^2 - 9n - 1$$

as the bound on the number of instructions that would be executed when sorting n items. This agrees, with our result from Section 4.7.2 and with the well-known fact that bubble sort takes $O(n^2)$ time to execute. Table 6.1 contains the results of both parts of the experiment.

6.1.1 Glue Layer Code

This glue layer code, written in Rust, loads the WebAssembly module containing the bubble sort implementation and repeatedly initializes memory with data to be sorted and calls the bubble sort function to sort it. It measures the amount of fuel consumed

Table 6.1: Bubble sort bound vs. actual - **wasmtime** fuel

Items (K)	WANALYZE Bound (M)	Worst-case Actual (M)	Difference	Random Actual (M)	Difference
10	1700	1550	9.67%	1401	17.69%
20	6800	6200	9.67%	5601	17.81%
30	15300	13950	9.67%	12594	17.90%
40	27200	24800	9.67%	22401	17.85%
50	42500	38750	9.68%	35000	17.85%
60	61200	55800	9.68%	50397	17.86%
70	83300	75950	9.68%	68625	17.81%
80	108800	99200	9.68%	89597	17.86%
90	137700	125550	9.68%	11340	17.85%
100	170000	155000	9.68%	13999	17.86%

by this call by setting the amount of fuel available to a very large value to ensure fuel is not exhausted. It then reports on the amount of fuel consumed.

```
1  use anyhow::Result;
2  use wasmtime::*;
3  use std::env;
4  use rand::Rng;
5
6  fn main() -> Result<()> {
7      // get our command line arguments
8      let args: Vec<String> = env::args().collect();
9      let w = &args[1]; // wasm module file
10     let f = &args[2]; // function we call
11
12     // set up the wasm configuration
13     let mut config = Config::new();
14     config.consume_fuel(true);
15     let engine = Engine::new(&config)?;
```

```
16        let mut store = Store::new(&engine, ());
17        store.add_fuel(10_000_000_000_000)?;
18
19        // get the wasm module we'll run
20        let module = Module::from_file(store.engine(), w)?;
21        let instance = Instance::new(&mut store, &module, &[])?;
22
23        // get the function in the wasm module that we'll call
24        let f_w =
25          instance.get_typed_func::<(i32, i32), (), _>(&mut store, f)?;
26
27        // create an array that will hold the data we'll sort
28        let mut data: [i32; 100000] = [0; 100000];
29
30        // repeatedly call the function in the wasm module that we're
31        // measuring
32        for n in 1.. {
33            // determine how many items we'll sort this pass
34            let count = n*10_000;
35            if count > 100_000 {
36                break;
37            }
38
39            // initialise the array with data that causes worst-case
40            // bubble-sort behaviour
41            for i in 0..count {
42                data[i] = (count-i) as i32;
```

```
43              // or alternatively initialise with random data
44              // data[i] =
45              //    rand::thread_rng().gen_range(1, count as i32)
46          }
47          // copy it to the module's memory
48          let memory =
49            instance.get_memory(&mut store, "memory").unwrap();
50          unsafe {
51              let raw = memory.data_ptr(&mut store).offset(0);
52              raw.copy_from(data.as_ptr() as *const u8, 4*count);
53          }
54
55          // call the wasm code to sort the data
56          let fuel_before = store.fuel_consumed().unwrap();
57          let _ = match f_w.call(&mut store, (count as i32, 0)) {
58              Ok(v) => v,
59              Err(_) => {
60                  println!("Exhausted fuel sorting {} items", count);
61                  break;
62              }
63          };
64          let fuel_consumed = store.fuel_consumed().unwrap()
65                                  - fuel_before;
66          println!("sorting {} items, [consumed {} fuel]", count,
67              fuel_consumed);
68          store.add_fuel(fuel_consumed)?;
69      }
```

```
70        Ok(())
71    }
```

6.1.2 Results

The results for this test are shown in Table 6.1. The table shows that the test was run on data sets containing a range of items that number between 10,000 and 100,000 in increments of 10,000. The column titled "WANALYZE Bound" shows the bound that WANALYZE produced for the number of instructions (in millions) executed to sort that many items. The next column "Worst-case Actual" is the number of instructions reported by **wasmtime** when sorting a set of items that is initially in the complete reverse of sorted order. The column "Random Actual" is the number of instructions but this time the set to be sorted is initially in random order. The two percentage columns are the differences between the WANALYZE bound and the actual measured result in **wasmtime**. The last section of this chapter contains a discussion of these results.

6.2 Dhrystone Benchmark

The second experiment followed a similar same procedure as the first but this time the WebAssembly code generated by the **emscripten** tool-chain for the version 2.1 of the Dhrystone benchmark [41] was analyzed. The Dhrystone benchmark takes a single input that specifies the number of iterations of the benchmark test that are to be run and so one might expect that performance will be linear in that parameter. Inspection of the code verifies that this is the case and WANALYZE produces a cost bound of

$$1033n + 2578$$

where n is the number of Dhrystone iterations performed. See Table 6.2 for the detailed measurements.

In this case no glue layer code was necessary because the Dhrystone program has no input data. The number of iterations to be performed is a parameter that's compiled into the program.

The results in Table 6.2 follow a similar format to those for bubble sort. The difference is that, since there is no input data to consider, it was only necessary to run a single test for a given number of iterations.

Table 6.2: Dhrystone predicted / actual

Iterations	**wasmtime** Actual	WANALYZE Bound	Difference
12088	7,834	12,487	59.39%
15110	9,792	15,609	59.40%
18666	12,100	19,282	59.36%
20336	13,180	21,007	59.39%
31110	20,183	32,137	59.23%

6.3 Sample Programs

The third experiment performed was designed to determine how successful WANALYZE could be at determining the cost of every function found in larger applications available as WebAssembly modules. To this end, seven WebAssembly modules of various sizes, representing a diverse set applications were analyzed by WANALYZE. We recorded which functions could be successfully analyzed and the reasons for failure when the analysis failed. These results appear in Table 6.3.

This table shows the name and type of each application. Lines of code and numbers of functions are given to provide an idea of the size of the applications. WANALYZE performance data in the form of lines of code analyzed per unit time and the number of functions successfully analyzed is also given. The column "% Costed"

Table 6.3: Sample programs (AMD Ryzen 7 3800XT, 3900 MHz)

	Elapsed (sec)	KLOC	KLOC / sec	Function Count	Functions Costed	% Costed	Type
allium	1.14	62	54	68	67	98.5%	bitcoin miner
gifsicle	8.55	90	11	353	351	99.4%	codec
mozjpeg	6.44	81	13	350	348	99.4%	codec
oxipng	12.63	147	12	426	422	99.1%	codec
vim	69.35	644	9	4,833	4,827	99.9%	text editor
doom	179.83	905	5	2,631	2,628	99.9%	game
autocad	2901.87	22,641	8	93,664	93,642	99.9%	drawing tool

contains the percentage of functions in the application that WANALYZE was able to produce a cost estimate for.

The WebAssembly binaries for these applications can be found on GitHub in the WANALYZE project [37].

6.4 Observations and Discussion of Results

The results for both the bubble sort and Dhrystone test share some characteristics. In both cases the bound produced by WANALYZE is indeed a bound. It's greater than the measured actual value. It's also true, in both cases, that the difference measured is a similar percentage for tests with the same input data. However this percentage varies across different tests and tests with different input data.

This raises a few questions: Why does the WANALYZE bound differ from the actual measurement at all? Why is there a difference between tests of different applications? For example, bubble sort with random data has a difference of 18% (rounded) but Dhrystone has a difference of 59%. Why the difference between tests with different data? The two different bubble sort tests have differences of 10% and 18%.

The primary reason for the difference between the WANALYZE bound and the actual measurement is that WANALYZE necessarily chooses the most costly path

when evaluating alternative costs in an extended basic block or basic block. In practice this is a conservative approach because it will often be the case that a less costly path is taken. This means that WANALYZE will produce a bound and the accuracy of that bound will depend on how often the costliest path is taken.

This also explains the difference between different applications. In this case it's the fact that not only do the applications have lower cost paths that contribute to the WANALYZE estimate being higher but the differences in the path structure of the functions in the two applications also affects it. When looking at the extended basic block's flow control diagram for two functions this difference can be clearly seen. And so the impact of lower cost paths will be different between the two applications.

It also explains why applications run on different data will have different results. The effect of the lower cost paths will be different when different input data is used.

The results from the third experiment on seven different applications shows that WANALYZE can successfully produce a bound for a very large range of functions. This test included an analysis of over 200,000 functions consisting of over 24 million lines of WebAssembly code. Failures were caused by one issue: the number of possible extended basic block paths in the function exceeded a limit (1 million) that was put in place to ensure that the analysis completes in a reasonable amount of time. In all of the analysis of these seven applications there are 40 functions like this. When their extended basic block flow control diagrams are examined it is seen that they either have a few hundred or more conditional tree branches that are executed in serial fashion or that they have tens of multi-way branches with tens of possible paths each. The number of paths grows exponentially in the number of these tree branches and so more than a million paths are found under these circumstances.

Chapter 7

Conclusions and Future Work

In this final chapter we return to the research questions posed in the introduction chapter, summarize the conclusions from our work and outline some avenues for further research and development that should be explored.

7.1 Research Questions

In the introduction we posed three research questions that we're now in a position to answer.

Research Question 1 asked what properties or preconditions must be satisfied in order that we can bound the cost of a WebAssembly function. Our test of the seven real world applications that we described in Chapter 6 lead us to believe that we can produce this bound for almost all WebAssembly functions in such an application. The only circumstance under which we fail is when the extended basic block flow control diagram of the function contains a large number of possible paths.

Research Question 2 asked what proportion of functions in real-world applications could we analyse successfully. The success rate for the seven applications that

we tested ranged from 98.5% to over 99.9%. The overall success rate was over 99.9% of all functions with this result being dominated by the high success rate achieved in the largest application.

Research Question 3 asked whether the analysis could be done in a timely manner. The results in Table 6.3 show that this the case for moderately size applications. Applications with 150,000 or few lines of WebAssembly source can be analyzed in less than 15 seconds. Once the analysis is done it's valid until the application changes and so a system that pre-computes or caches the bound would reduce the importance on how quickly it can be done.

7.2 Future Work

This work raises additional software development work to be done and research questions that are unanswered at this point.

The first area is to further demonstrate the validity of WANALZE by increasing the amount of testing done and providing formal assurance of the results. We've been able to test our designs and code against a moderate number of diverse applications but to increase our confidence in the results a broader set of tests should be performed, and, perhaps more importantly, key aspects of the system should be formally proven to be correct. As well as providing greater assurance, a mechanization of this proof of correctness would also mean that the provably correct code could be generated from such a proof.

Forthcoming WebAssembly features will provide challenges in extending WANALYZE. In particular, exception handling and tail calls will require changes to our extended basic block definition and that this entails. As part of any effort to support these features it will be useful to extend WANALYZE to provide a bound on an entire function call graph rather than just a single function invocation.

An open question at this point is whether this work can be applied to other programming languages. Rust, in particular, may have a sufficient formal definition to allow this. At this point the greatest challenge would seem to be the fact that Rust, unlike WebAssembly, is decidedly not low level.

Another open question is whether the methods that we describe herein can be extended or used directly to solve other types of static analysis problems such as data flow analysis. This would have broad applicability to the security and quality of application and infrastructure systems.

7.3 Conclusion

WANALYZE can determine a bound on the execution cost for almost all functions in real world applications written in WebAssembly. For moderate-sized applications this bound can be produced in a timely manner. Strategies can be deployed to persist the bounds that can be pre-computed for very large applications. Aspects of WebAssembly, particular that has a formal definition, is low-level in nature and it has structured flow control contribute to make it possible to compute these bounds.

Bibliography

[1] Andreas Abel and Jan Reineke. uiCA: Accurate throughput prediction of basic blocks on recent Intel microarchitectures. In *Proceedings of the 36th ACM International Conference on Supercomputing*, pages 1–14, 2022.

[2] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[3] Andrew W Appel and Amy P Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 243–253, 2000.

[4] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[5] AutoCAD. Roundup: The AutoCAD Web App at Google I/O 2018. https://blogs.autodesk.com/autocad/autocad-web-app-google-io-2018/. Accessed December, 2022.

[6] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 119–128. IEEE, 2007.

[7] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[8] Enrico Bini and Giorgio C Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, 2004.

[9] Niklas Borselius. Mobile agent security. *Electronics & Communication Engineering Journal*, 14(5):211–218, 2002.

[10] Bytecode Alliance. WASI Overview. https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md. Accessed December, 2022.

[11] Bytecode Alliance. wasmtime - A Standalone Runtime for WebAssembly. https://github.com/bytecodealliance/wasmtime. Accessed December, 2022.

[12] Cillium. Cillium. https://cilium.io/. Accessed December, 2022.

[13] John Cocke. Global common subexpression elimination. In *Proceedings of a symposium on Compiler optimization*, pages 20–24, 1970.

[14] Collabora. An eBPF Overview. https://www.collabora.com/news-and-blog/blog/2019/04/05/an-ebpf-overview-part-1-introduction/. Accessed December, 2022.

[15] Keith D Cooper and Linda Torczon. *Engineering a Compiler*. Elsevier, 2011.

[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, third edition*. The MIT Press, 2009.

[17] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978.

[18] CPP Reference. Undefined behavior. https://en.cppreference.com/w/c/language/behavior. Accessed December, 2022.

[19] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, 1989.

[20] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. *Acm Sigplan Notices*, 48(10):443–456, 2013.

[21] ebpf.org. eBPF. https://ebpf.io/. Accessed December, 2022.

[22] Ramin Edjlal, Armin Edjlal, and Tayebeh Moradi. A sort implementation comparing with bubble sort and selection sort. In *2011 3rd International Conference on Computer Research and Development*, volume 4, pages 380–381. IEEE, 2011.

[23] Emscripten. Emscripten Compiler Tool Chain. https://emscripten.org/. Accessed December, 2022.

[24] Ethereum. Ethereum. https://ethereum.org/en/. Accessed December, 2022.

[25] Ethereum. Gas and Fees. https://ethereum.org/en/developers/docs/gas/. Accessed December, 2022.

[26] William Findlay, Anil Somayaji, and David Barrera. Bpfbox: Simple precise process confinement with ebpf. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 91–103, 2020.

[27] Go. Go 1.11 Release Notes. https://go.dev/doc/go1.11. Accessed December, 2022.

[28] Google. WebAssembly brings Google Earth to more browsers. https://blog.chromium.org/2019/06/webassembly-brings-google-earth-to-more.html. Accessed December, 2022.

[29] Michael S Greenberg, Jennifer C Byington, and David G Harper. Mobile agents and security. *IEEE Communications magazine*, 36(7):76–85, 1998.

[30] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In *International Conference on Computer Aided Verification*, pages 634–640. Springer, 2009.

[31] Dwight Guth. A formal semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, 2013.

[32] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[33] Christopher Healy, Mikael Sjodin, Viresh Rustagi, and David Whalley. Bounding loop iterations for timing analysis. In *Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium (Cat. No. 98TB100245)*, pages 12–21. IEEE, 1998.

[34] Thomas A Henzinger, Thibaud Hottelier, and Laura Kovács. Valigator: A verification tool with bound and invariant generation. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 333–342. Springer, 2008.

[35] Xuan Huang. A mechanized formalization of the Webassembly specification in Coq. *RIT Computer Science*, 2019.

[36] Wayne A Jansen. Countermeasures for mobile agent security. *Computer communications*, 23(17):1667–1676, 2000.

[37] John Shortt. WANALYZE - A tool to analyze WebAssembly functions. https://github.com/jsCarleton/wanalyze. Accessed December, 2022.

[38] Ralf Jung. *Understanding and Evolving the Rust Programming Language*. PhD thesis, Universität des Saarlandes, 2020.

[39] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976.

[40] Katran. Katran. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer//. Accessed December, 2022.

[41] Keith S. Thompson. Dhrystone v2.1. https://github.com/Keith-S-Thompson/dhrystone/tree/master/v2.1. Accessed December, 2022.

[42] Minseo Kim, Hyerean Jang, and Youngjoo Shin. Avengers, Assemble! Survey of WebAssembly Security Solutions. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 543–553, 2022.

[43] Laura Kovács. Reasoning algebraically about p-solvable loops. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 249–264. Springer, 2008.

[44] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *International Conference on Fundamental Approaches to Software Engineering*, pages 470–485. Springer, 2009.

[45] Peter Lee and George Necula. Research on proof-carrying code for mobile-code security. In *DARPA workshop on foundations for secure mobile code*, pages 26–28. Citeseer, 1997.

[46] K Rustan M Leino and Francesco Logozzo. Loop invariants on demand. In *Asian symposium on programming languages and systems*, pages 119–134. Springer, 2005.

[47] Borui Li, Hongchang Fan, Yi Gao, and Wei Dong. Bringing Webassembly to resource-constrained IOT devices for seamless device-cloud integration. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 261–272, 2022.

[48] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *2009 International Symposium on Code Generation and Optimization*, pages 136–146. IEEE, 2009.

[49] Pedro Daniel Rogeiro Lopes. Discovering vulnerabilities in Webassembly with code property graphs. *Técnico Lisboa*, 2021.

[50] Kuang-Chen Lu, Ben Greenman, Carl Meyer, Dino Viehland, Aniket Panse, and Shriram Krishnamurthi. Gradual Soundness: Lessons from Static Python. *arXiv preprint arXiv:2206.13831*, 2022.

[51] Salvador Lucas. The origins of the halting problem. *Journal of Logical and Algebraic Methods in Programming*, 121:100687, 2021.

[52] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional Programming for the Masses.* " O'Reilly Media, Inc.", 2013.

[53] George C Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, 1997.

[54] George C Necula and Peter Lee. Safe kernel extensions without run-time checking. In *OSDI*, volume 96-16, pages 229–243, 1996.

[55] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[56] José Carlos Palencia and M González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*, pages 26–37. IEEE, 1998.

[57] Benjamin C Pierce. *Types and Programming Languages*. MIT press, 2002.

[58] Quicknode. An overview of how smart contracts work on Ethereum. https://www.quicknode.com/guides/smart-contract-development/an-overview-of-how-smart-contracts-work-on-ethereum. Accessed December, 2022.

[59] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, 1988.

[60] Rust. WebAssembly. https://www.rust-lang.org/what/wasm, 2021.

[61] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. *Advances in Neural Information Processing Systems*, 31, 2018.

[62] Quentin Stiévenart, David W Binkley, and Coen De Roover. W: a WebAssembly Static Analysis Library (Extended Presentation Abstract). https://soft.vub.ac.be/Publications/2021/vub-tr-soft-21-04.pdf. Accessed December, 2022.

[63] Quentin Stiévenart, David W Binkley, and Coen De Roover. Static stack-preserving intra-procedural slicing of webassembly binaries. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2031–2042, 2022.

[64] Quentin Stiévenart and Coen De Roover. Compositional information flow analysis for Webassembly programs. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 13–24. IEEE, 2020.

[65] Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. Taint tracking for Webassembly. *arXiv preprint arXiv:1807.08349*, 2018.

[66] Tomoya Taka, Tadanori Mizuno, and Takashi Watanabe. A model of mobile agent services enhanced for resource restrictions and security. In *Proceedings 1998 International Conference on Parallel and Distributed Systems (Cat. No. 98TB100250)*, pages 274–281. IEEE, 1998.

[67] Discussion Thread. stackoverflow. \url{https://stackoverflow.com/questions/70841631/bpf-verifier-says-program-exceeds-1m-instruction}. Accessed December, 2022.

[68] Unity. WebAssembly is here! https://blog.unity.com/technology/webassembly-is-here. Accessed December, 2022.

[69] Cyril Wanner. wasm-codecs. https://github.com/cyrilwanner/wasm-codecs. Accessed December, 2022.

[70] Conrad Watt. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on certified programs and proofs*, pages 53–65, 2018.

[71] Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. Two Mechanisations of WebAssembly 1.0. In *International Symposium on Formal Methods*, pages 61–79. Springer, 2021.

[72] WebAssembly Community Group. WebAssembly. https://https://webassembly.org/. Accessed December, 2022.

[73] WebAssembly Community Group. WebAssembly Introduction. https://webassembly.github.io/spec/core/intro/introduction.html. Accessed December, 2022.

[74] WebAssembly Community Group. WebAssembly Roadmap. https://https://webassembly.org/roadmap. Accessed December, 2022.

[75] WebAssembly Community Group. WebAssembly specification, reference interpreter, and test suite. https://github.com/WebAssembly/spec/tree/w3c-1.0. Accessed December, 2022.

[76] WebAssembly Community Group. WebAssembly specification v2.0. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf. Accessed December, 2022.

[77] Ben Wegbreit. Property extraction in well-founded property sets. *IEEE Transactions on Software Engineering*, SE-1(3):270–285, 1975.

[78] Joachim Wegener and Frank Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-time systems*, 21(3):241–268, 2001.

[79] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, pages 352–357, 1984.

[80] Elliott Wen and Gerald Weber. Wasmachine: Bring the Edge up to Speed with A WebAssembly OS. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 353–360, 2020.

[81] Fengxiang Zhang and Alan Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Transactions on Computers*, 58(9):1250–1258, 2009.