WebAssembly Optimizations

Tadeo Volker Hees

March 20, 2025
Version: 484ec00d

Johannes Gutenberg University Mainz
FB08
Institute of Computer Science
Programming Languages

Bachelor thesis

WebAssembly Optimizations

Tadeo Volker Hees

1. Reviewer Sebastian Erdweg
Institute of Computer Science - Programming Languages
Johannes Gutenberg University Mainz

2. Reviewer André Brinkmann
Institute of Computer Science - Efficient Computing and Storage
Johannes Gutenberg University Mainz

Supervisors Sebastian Erdweg and André Brinkmann

March 20, 2025

Tadeo Volker Hees
WebAssembly Optimizations
Bachelor thesis, March 20, 2025
Reviewers: Sebastian Erdweg and André Brinkmann
Supervisors: Sebastian Erdweg and André Brinkmann

Johannes Gutenberg University Mainz
Programming Languages
Institute of Computer Science
FB08
Staudingerweg 9
55128 Mainz

Abstract

The thesis presents various static optimizations on WebAssembly code, given the analysis results of Sturdy-Wasm [1]. The optimizations were split into three parts: Dead code removal, constants optimizations and drops removal. All mentioned optimizations were implemented in Scala and their correctness tested with manually created unit tests as well as validation tests. The optimizations were then applied in a specified order on a large selection of WebAssembly files in Chapter 4 to test their effectiveness.

Abstract (german)

Die Bachelorarbeit stellt mehrere statische Optimierungen von WebAssembly Code vor, die auf Analyseergebnissen von Sturdy-Wasm [1] basieren. Die Optimierungen wurden in drei Teile geteilt: Dead-code Entfernungen, Optimierungen von Konstanten und die Entfernung von drops. Alle vorgestellten Optimierungen wurden in Scala implementiert und dessen Korrektheit mithilfe von manuell erstellten Testdateien geprüft, sowie mit einer großen Anzahl von Dateien validiert. Alle Optimierungen wurden schließlich in einer festgelegten Reihenfolge auf diese Dateien angewendet und dessen Ergebnisse in Kapitel 4 evaluiert.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Thesis structure . . . 1

2 WebAssembly **3**
2.1 Introduction . . . 3
2.2 Modules . . . 3
2.2.1 Basic types . . . 4
2.2.2 Functions . . . 4
2.2.3 Memory . . . 6
2.2.4 Control Flow . . . 6

3 Optimizations **9**
3.1 Overview . . . 9
3.1.1 Sturdy . . . 9
3.1.2 Overview of optimizations . . . 9
3.1.3 Implementation of the base module visitor . . . 11
3.2 Constants . . . 12
3.2.1 Idea . . . 12
3.2.2 Implementation . . . 16
3.3 Deadcode . . . 19
3.3.1 Idea . . . 19
3.3.2 Implementation . . . 22
3.4 Drops . . . 27
3.4.1 Idea . . . 27
3.4.2 Implementation . . . 28

4 Evaluation **31**

5 Related work **33**
5.1 Other optimizations based on Webassembly . . . 33
5.2 Optimizations on stack based languages . . . 33

6 Conclusion and future work **35**

Bibliography **37**

List of Figures **39**

List of Tables **41**

List of Listings 43

1 Introduction

1.1 Background

WebAssembly, often abbreviated to WASM, has become a relevant language in the landscape of web development. It offers a robust, efficient, and secure way to run high-performance code on the web. Unlike JavaScript, it is designed from the beginning to be a good compilation target for other languages. While there are already tools for optimizing WebAssembly code, such as Binaryen [@13], this thesis presents optimizations based on the static analysis results of Sturdy-Wasm [1].

1.2 Thesis structure

The thesis begins with a brief introduction to WebAssembly in Chapter 2. This chapter covers all the relevant information needed to understand the optimizations.

Chapter 3 contains the thesis's main content, presenting all performed optimizations divided into 3 parts. The constant optimization consists of replacing instructions that always return the same value and the removal of constant function parameters. The next optimization removes dead code and unused control-flow structures. The last optimization consists of removing the chain of unnecessary instructions from definition to drops. This optimization is mainly needed for the drops that are added in the previous 2 optimizations.

Chapter 4 mentions the validation tests that were performed for each optimization and the benchmark that reports the amount of instructions that were removed after running every optimization in a predetermined order. A filtered subset of WasmBench [4], a collection of real-world WebAssembly modules, was used for the entire evaluation.

Chapters 5 and 6 contain information about related works and the conclusion of the thesis.

2 WebAssembly

2.1 Introduction

WebAssembly [3] is a low-level assembly-like language based on a virtual stack machine. It is primarily intended to be used on the Web by embedding it into JavaScript but not limited to that use. It was designed to be fast with minimal overhead, safe and compact while addressing the shortcomings of JavaScript, especially when it comes to being a compilation target for other languages. Even though the code is low-level, WebAssembly makes no assumptions about the underlying hardware which makes it portable, as it is required by targeting the Web as a platform. To reduce the required bandwidth for transmitting the code, WebAssembly is usually encoded in a size efficient binary format, although it does also support an equivalent and human-readable text format.

Note: The entire thesis as well as the implementation of the optimizations assumes the usage of WebAssembly revision 1.0, since that is also the latest supported version by the Sturdy analysis and the Swam [@12] parser.

2.2 Modules

A binary in WebAssembly consists of a single module with definitions for *functions*, *globals*, *tables* and *memories* as well as potential *exports* of those under one or more names and *imports*. *Imports* are defined by specifying the

Fig. 2.1: WebAssembly abstract syntax [3]

(value types) $t ::= \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64}$
(packed types) $tp ::= \mathsf{i8} \mid \mathsf{i16} \mid \mathsf{i32}$
(function types) $tf ::= t^* \to t^*$
(global types) $tg ::= \mathsf{mut}^? \; t$

$unop_{\mathrm{iN}} ::= \mathsf{clz} \mid \mathsf{ctz} \mid \mathsf{popcnt}$
$unop_{\mathrm{fN}} ::= \mathsf{neg} \mid \mathsf{abs} \mid \mathsf{ceil} \mid \mathsf{floor} \mid \mathsf{trunc} \mid \mathsf{nearest} \mid \mathsf{sqrt}$
$binop_{\mathrm{iN}} ::= \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div\_}sx \mid \mathsf{rem\_}sx \mid \mathsf{and} \mid \mathsf{or} \mid \mathsf{xor} \mid \mathsf{shl} \mid \mathsf{shr\_}sx \mid \mathsf{rotl} \mid \mathsf{rotr}$
$binop_{\mathrm{fN}} ::= \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{min} \mid \mathsf{max} \mid \mathsf{copysign}$
$testop_{\mathrm{iN}} ::= \mathsf{eqz}$
$relop_{\mathrm{iN}} ::= \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt\_}sx \mid \mathsf{gt\_}sx \mid \mathsf{le\_}sx \mid \mathsf{ge\_}sx$
$relop_{\mathrm{fN}} ::= \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt} \mid \mathsf{gt} \mid \mathsf{le} \mid \mathsf{ge}$
$cvtop ::= \mathsf{convert} \mid \mathsf{reinterpret}$
$sx ::= \mathsf{s} \mid \mathsf{u}$

(instructions) $e ::= \mathsf{unreachable} \mid \mathsf{nop} \mid \mathsf{drop} \mid \mathsf{select} \mid$
$\mathsf{block}\ tf\ e^*\ \mathsf{end} \mid \mathsf{loop}\ tf\ e^*\ \mathsf{end} \mid \mathsf{if}\ tf\ e^*\ \mathsf{else}\ e^*\ \mathsf{end} \mid$
$\mathsf{br}\ i \mid \mathsf{br\_if}\ i \mid \mathsf{br\_table}\ i^+ \mid \mathsf{return} \mid \mathsf{call}\ i \mid \mathsf{call\_indirect}\ tf \mid$
$\mathsf{get\_local}\ i \mid \mathsf{set\_local}\ i \mid \mathsf{tee\_local}\ i \mid \mathsf{get\_global}\ i \mid$
$\mathsf{set\_global}\ i \mid t.\mathsf{load}\ (tp\_sx)^?\ a\ o \mid t.\mathsf{store}\ tp^?\ a\ o \mid$
$\mathsf{current\_memory} \mid \mathsf{grow\_memory} \mid t.\mathsf{const}\ c \mid$
$t.unop_t \mid t.binop_t \mid t.testop_t \mid t.relop_t \mid t.cvtop\ t\_sx^?$

(functions) $f ::= ex^*\ \mathsf{func}\ tf\ \mathsf{local}\ t^*\ e^* \mid ex^*\ \mathsf{func}\ tf\ im$
(globals) $glob ::= ex^*\ \mathsf{global}\ tg\ e^* \mid ex^*\ \mathsf{global}\ tg\ im$
(tables) $tab ::= ex^*\ \mathsf{table}\ n\ i^* \mid ex^*\ \mathsf{table}\ n\ im$
(memories) $mem ::= ex^*\ \mathsf{memory}\ n \mid ex^*\ \mathsf{memory}\ n\ im$
(imports) $im ::= \mathsf{import}\ \text{“}name\text{”}\ \text{“}name\text{”}$
(exports) $ex ::= \mathsf{export}\ \text{“}name\text{”}$
(modules) $m ::= \mathsf{module}\ f^*\ glob^*\ tab^?\ mem^?$

Figure 1. WebAssembly abstract syntax

names of a module and the entity being imported from it, as well as a type descriptor. A module can be instantiated by the embedder which requires giving definitions for all *imports* with matching types. Those definitions can also come from exports defined by other WebAssembly instances.

2.2.1 Basic types

The basic value types in WebAssembly are 32-bit and 64-bit integers and IEEE 754 floats. There are operations to convert between all value types and to reinterpret types of equal size. Signed and unsigned integers don't have separate types.

Global types also consist of an indicator whether the value is mutable or immutable.

2.2.2 Functions

Functions are classified by function types which indicate the types of values that are consumed as arguments and pushed as results of the computation. As of revision 1.0 of WebAssembly a function can return at most one value. The body of a function consists of instructions. The abstract syntax of all instructions can be seen in Figure 2.1.

Inside of a function all arguments are defined as the first **local variables**. Locals are accessed via their index and can be pushed to the stack with `local.get` and defined with `local.set` or `local.tee`, which acts like `local.set` but also leaves the value on the stack. All locals, including those defined by the function arguments, are mutable and can be overwritten.

The **stack** holds the values that are used by instructions as parameters. Instructions can manipulate the current stack by pulling and pushing values. During execution each function begins with an empty stack. Lst. 2.1 shows an example of a function with the concrete stack during execution as comments on the right.

```
1 (func "main" (result i32)
2   i32.const 2        ;; []
3   i32.const 3        ;; [2]
4   i32.add            ;; [3, 2]
5 )                    ;; [5]
```

Lst. 2.1: Concrete stack during execution

There is a `drop` instruction that just pulls the top value from the stack and discards it, but there are no other instructions to explicitly manipulate values on the stack, such as a duplication or swap instruction that are common in other stack based languages. The stack shape, which refers to the size and

types of contained values, can be statically determined at every point and has to match the function type at the end of the function for a successful validation. The end of a function can be reached by finishing the last instruction or by a jump targeting it. This jump can be archived with a `return` instruction or with any branch instruction targeting the most outer label, which is explained further in Section 2.2.4. When reaching the end of a function with a jump, all leftover values on the stack are discarded. Otherwise, the stack has to match the function type exactly without any leftover values. Lst. 2.2 shows an example of how the stack shape can be determined at any point. It should be noted that the body of an if construct begins with a new empty stack, which is explained in more detail in Section 2.2.4.

```
1  (func "main" (param i32) (param f32)
2    f64.const 2.3      ;; []
3    local.get 0        ;; [f64]
4    if (result f32)    ;; [i32, f64]
5      local.get 1      ;; []
6    else
7      f32.const 3.5    ;; []
8    end                ;; [f32]
9    drop               ;; [f32, f64]
10   drop               ;; [f64]
11 )                    ;; []
```

Lst. 2.2: Deterministic stack shape

All functions of a module are referenced with an index, starting with 0. Imported functions are indexed before explicitly defined functions. Directly calling functions via their index is possible with `call` instructions. The arguments are consumed from the stack and the return values pushed to the stack.

Tables are array-like data structures that store references to functions via their indices. The purpose of tables is to allow some functions to be called with an `indirect_call` instruction, which takes a table index from the stack to determine the called function. This means that the called function could depend on parameters of the current function or the memory and can't be statically determined, unlike direct calls. There can be at most one table per module. In the binary format, function references in tables are statically defined in a separate elements section. As of revision 1.0 it is not possible to dynamically insert elements into tables via instructions. To still allow for a type analysis with indirect calls, a reference to the expected function type has to be given, which will be checked dynamically against the called function. The first popped stack value is the table index to the desired function reference. Then all arguments are popped, and return values pushed like a direct `call`.

2.2.3 Memory

Modules in WebAssembly support memory as storage. Changes made to memory persist for the entire module instance and are not limited to the scope of the current function, unlike the stack. It is also possible to import and export memory to the embedder and other instances, which means that the changes aren't limited to the current instance either. This is an important fact for optimizations as it makes it impossible to statically determine the contents and accesses to the memory from just one module alone.

At most one memory can be defined by each module. The initial size at creation can be dynamically expanded with the `grow_memory` instruction, and its current size determined with the `current_memory` instruction.

Access to memory is possible with the `t.load` and `t.store` instructions, with *t* being one of the 4 possible value types. These instructions take optional static offset and alignment arguments as unsigned i32 (32bit integers). The load instruction then also consumes one dynamic unsigned i32 address from the stack and returns the value from memory at the effective address, which is the sum of the static offset and dynamic address. The store instruction first consumes the value to be stored from the stack and then the dynamic address. Both instructions can explicitly use 8, 16 and 32 bit-widths instead of the default one, defined by the value type. In these cases a load instruction must specify whether the value should be interpreted as signed or unsigned.

2.2.4 Control Flow

Control flow in WebAssembly is structured and only allows jumps to predefined labels. Arbitrary jumps to any line of code are not possible.

The three control flow constructs that define labels are `block`, `loop` and `if`. Each of them has a block signature that defines parameters and return values. All parameters are popped from the stack as arguments. As of revision 1.0 of WebAssembly the amount of return values can be at most 1. When entering the body of one of these constructs, a new stack is created and all the arguments pushed onto it. The stack has to match the signature exactly when exiting the construct, similar to exiting a function. After leaving the construct, return values are pushed onto the outer stack.

Labels are added by the control constructs and are nested. Adding a new label by entering a control construct can be seen as pushing to an implicit label stack and reaching the end of the control construct as popping from that stack. The first label in this stack is always the end of the function. After jumping to the index n, n+1 values are popped from the implicit label stack.

The `br` instruction takes an index starting with 0 and jumps to the target from the label stack at that position, starting from the top of the stack.

The `br_if` instruction first consumes a value from the execution stack and only performs the jump if it is not zero.

The `br_table` instruction consists of a list of m labels. It consumes one value from the execution stack to select the label, with the last label in the list always being the target for out of bound selections.

A `block` construct adds a label with its jump target at the end of the block. This means that a branch instruction targeting that label would jump to the end of the entire block construct. A `loop` works exactly the same as a `block` but defines the jump target of it's label to the first instruction of its body. Lst. 2.3 shows an example of an infinite loop where the jump conditional for `br_if` is always 1. An `if` construct consumes one value from the stack and executes the else branch if that value is zero. Otherwise, it executes the then branch. The `if` construct adds one label, with its target being at the end of the entire construct.

```
(loop
  (block
    br 0
    unreachable
  )
  i32.const 1
  br_if 0
  unreachable
)
unreachable
```

Lst. 2.3: Block jump and infinite loop

3 Optimizations

3.1 Overview

3.1.1 Sturdy

Sturdy [@9] is a tool for static analysis. In this thesis, only the constant analysis of Sturdy-Wasm [1] was used. The analysis generates a control flow graph starting from every entry point of a module, which are all the exported functions. Every imported function, except for host functions, has to be explicitly defined for the analysis to work. In this way, the analysis can span over multiple modules. All optimizations are currently only supported for the scope of a single module, which also means that the module is not allowed to have any imports, except for host functions.

The constant analysis reports dead code in form of dead instructions, which are unreachable nodes in the control flow graph. These instructions are always unreachable during execution.

Dead labels are reported as instructions that define a label which is never jumped to, according to the control flow graph. This could for example be a `block` construct that isn't referenced by any reachable branch instruction or whose referencing branch instructions are conditional and never executed.

Constant instructions are also reported as instructions that always return the same constant value.

For the optimizations only the control flow graph, dead code and constant instructions are needed from the analysis. The reported instruction nodes indicate the function index and the program counter inside of it's body, which allows the instruction to be located exactly inside the parsed module.

3.1.2 Overview of optimizations

Before delving into the split of the optimizations themselves, the strict separation between analysis and optimization is worth mentioning. In this thesis the analysis comes from sturdy-wasm and the optimizations are performed using all the available information from them. The analysis itself was not altered. Combining analysis and optimization could reduce computational overhead and allow for more elegant solutions in some cases. For example the optimization of control flow structures in Section 3.3.2 had to use workarounds

in some cases to work with only the provided control flow graph of sturdy, and unused `br_if` instructions aren't detected in all cases. This problem could also be solved by extending the analysis with a more verbose control flow graph, so a combination of analysis and optimization isn't strictly needed. Nonetheless, separating optimizations from the analysis generally allows for a simpler and more structured solution.

The constant optimization in Section 3.2 is the first of 3 optimizations. It technically consists of two different optimizations: A constant instruction optimization that replaces instructions that always return a constant value with that value, and a constant parameter removal that removes parameters which evaluate to the same constant value for every possible call of the function. They both only use constant instructions reported by Sturdy. Drops are prepended to preserve the stack signature after replacing instructions. The constant parameter optimization depends on reported constant `local.get` instructions, so it made sense to combine it with the constant instruction optimization. Both optimizations are useful as they avoid the unnecessary execution of instructions and the passing of unnecessary function arguments. Even if the total amount of instructions could increase during this optimization by prepending drops, the potential for saving computation can be significant, for example by replacing an entire call of a function with drops and a constant.

The second optimization is the dead code and unnecessary control flow structure removal in Section 3.3. The dead code removal consists of removing all dead instructions reported by the sturdy analysis. The control flow structure removal uses the reported control flow graph to detect constructs with dead labels, which means that no jumps occur to its label, dead else or then branches of an if construct or `br_if` instructions that never jump. These optimizations all remove instructions and rely on the reported dead code and dead labels, which is why a combination of them made sense. Drop instructions are also added in this optimization to preserve the correct stack signature. The dead code removal is useful to reduce the size of the module. The unnecessary control flow structure removal can also avoid the computation of control flow instructions by removing them.

The third and final optimization is the drop removal. This optimization was mainly required because the other optimizations add drop instructions. This leads to further potential for optimizations as all instructions that lead to the values on the stack that were dropped, without having any other effects like interacting with the memory, are unnecessary and can be removed. Removing them can save large amounts of space and avoid the computation of unnecessary instructions.

All optimizations where implemented in Scala 3 and use Swam [@12] for parsing WebAssembly modules in the binary or text format.

3.1.3 Implementation of the base module visitor

This section shows the implementation of the basic module visitor class that is extended for every other optimization. The class receives a parsed module from the Swam parser as an argument. The `visitModule()` method visits every section of the module and its contents, such as function bodies. The visited elements are returned as a sequence and flattened, which allows overwritten visit-methods to return more than one element. For example visiting a single instruction in the body of a function could return multiple instructions or no instruction at all. Without changes, the base module visitor just returns an unchanged module and is intended to be extended by the optimizations classes. This approach is useful to avoid redundant code between optimizations.

There is a mutable global `funcPC` integer variable that tracks the current program counter. This counter is used to identify the current instruction in relation to those reported by the sturdy analysis, while visiting the body of a function. As such the `funcPC` variable is reset in the `visitFunc(func, funcIdx)` method and increased in the `visitFuncInstr(funcInstr, funcIdx)` method. It is important to ensure that this counter is also updated correctly when overwriting these methods for the optimization classes.

```
1  class BaseModuleVisitor(mod: Module):
2
3    var funcPc: Int = 0
4
5    def visitModule(): Module =
6      val impFuncAmount = mod.imported.funcs.size
7      Module(
8        mod.types.flatMap(visitType),
9        // The index gets shifted by the amount of imported functions
             since those always come before!
10       mod.funcs.zipWithIndex.flatMap((func, funcIdx) => visitFunc(func,
              funcIdx+impFuncAmount)),
11       mod.tables,
12       mod.mems,
13       mod.globals.flatMap(visitGlobal),
14       mod.elem.flatMap(visitElem),
15       mod.data,
16       mod.start.map(visitStart),
17       mod.imports.flatMap(visitImport),
18       mod.exports.flatMap(visitExport)
19     )
20
21   def visitType(tpe: FuncType): Seq[FuncType] = Seq(tpe)
22
23   def visitFunc(func: Func, funcIdx: Int): Seq[Func] =
24     funcPc = -1
25     Seq(Func(func.tpe, func.locals.flatMap(visitFuncLocal(_, funcIdx)),
           func.body.flatMap(visitFuncInstr(_, funcIdx))))
26
27   def visitFuncLocal(local: ValType, funcIdx: FuncIdx): Seq[ValType] =
       Seq(local)
28
29   def visitFuncInstr(funcInstr: Inst, funcIdx: FuncIdx): Seq[Inst] =
```

```
30      funcPc += 1
31      Seq(funcInstr)
32
33    def visitGlobal(global: Global): Seq[Global] = Seq(global)
34
35    def visitImport(imprt: Import): Seq[Import] = Seq(imprt)
36
37    def visitExport(exprt: Export): Seq[Export] = Seq(exprt)
38
39    def visitElem(elem: Elem): Seq[Elem] = Seq(Elem(elem.table, elem.
          offset, elem.init.flatMap(visitElemInit)))
40
41    def visitElemInit(funcidx: FuncIdx): Seq[FuncIdx] = Seq(funcidx)
42
43    def visitStart(start: FuncIdx) = start
```

Lst. 3.1: The base module visitor

3.2 Constants

3.2.1 Idea

Constant Instructions

The constants reported by the constant analysis of Sturdy indicate which instructions are constant and to which constant value they evaluate. Constant in this context means that the instruction always evaluates to the same constant value across every possible execution. The main idea is to replace constant instructions with their reported values and preserve the stack signature by prepending drops. The following Lst. 3.2 shows an example where a `i32.add` instruction always evaluates to 3. Simply replacing the instruction with `i32.const 3` would break the signature of the stack, as the function would end with a stack shape of [i32, i32, i32] instead of the expected [i32]. Therefore, two drops need to be prepended. In total the entire optimization consists of replacing the `i32.add` instruction with `drop; drop; i32.const 3`

```
1  (func (export "main") (result i32)
2    i32.const 1       ;; []
3    i32.const 2       ;; [i32]
4    i32.add           ;; [i32, i32]
5  )                   ;; [i32]
```

Lst. 3.2: Constant add with annotated stack shape

The only part that differs between instructions to be replaced, is the amount of drops that have to be prepended. For all instructions except calls, the change to the stack is fixed. For direct and indirect calls the changes to the stack are specified with a function type. Replacing such an instruction requires prepending as many drops as function parameters. Lst. 3.3 shows an example

with constant calls. The comments show the replacements made. It should be noted that the function `const_func` is not exported and can therefore only be executed by its single direct call in the main function, where all passed parameters are always the same constant values.

```
(func $const_func (param i32 i32) (result i32)
  local.get 0       ;; i32.const 3
  local.get 1       ;; i32.const 2
  i32.sub           ;; drop; drop; i32.const 1
)
(func $return_3 (result i32)
  i32.const 3
)
(func (export "main") (result i32)
  call $return_3    ;; i32.const 3
  i32.const 2
  call $const_func  ;; drop; drop; i32.const 1
)
```

Lst. 3.3: Constant direct calls with annotated changes

Constant Parameters

Constant parameters refer to function parameters whose referencing `local.get` instructions are declared constant and which are never referenced by any `local.set` or `local.tee` instruction. These parameters are unnecessary after performing the constant instructions optimization and can be removed.

The removal of function parameters is performed by changing its function type. In Lst. 3.3, the function type of `const_func` would change from `(param i32 i32)(result i32)` to just `(result i32)`, as both parameters are constant. Technically in this case the function is not reachable anymore and could be removed entirely, but this dead code removal is performed in Section 3.3. In general let f be a function with constant parameters, A its original type and B the new type with removed constant parameters. If f is only referenced by direct calls, changing $f.type$ to B works out fine but indirect calls can cause problems, since the function type that is referenced by the indirect call, which is equivalent to A, would no longer match up with B. If it were known that every possible indirect call to f could only target functions with a type changed to B after performing the optimization, then it would be possible to also change the referenced type of the call. Nonetheless, the currently implemented optimization performs no removal for constant parameters at all if the function is referenced in a table.

The following Lst. 3.4 shows an example of the problem that can arise with indirect calls. Parameter 1 of the main function determines which of the two other functions is called. While parameter 1 of the function `const_param_1` is constant, the function type can't be changed to `(param i32) (result i32)` since it would require to change the type of the indirect call, which would in turn require the type of `no_const_param` to change as well. The type of

`no_const_param` can't be changed since none of its parameters are constant, as it is an exported function.

```
(type $func_type (func (param i32 i32) (result i32)))
(table 2 funcref)
(elem (i32.const 0) $const_func $non_const_func)
(func $const_param_1 (type $func_type)
  local.get 0
  local.get 1       ;; i32.const 3
  i32.add
)
(func $no_const_param (type $func_type)
  local.get 0
  local.get 1
  i32.add
)
(func $main (type $func_type)
  local.get 0
  i32.const 3
  local.get 1
  call_indirect (type $func_type)
)
(export "main" (func $main))
(export "non_const_func" (func $non_const_func))
```

Lst. 3.4: Constant parameter indirect call with annotated changes

Removing parameters from a function leaves leftover values on the stack for every call referencing that function. To preserve the signature of the stack, drops are prepended according to the removed parameters. This works only if all removed parameters consume values sequentially from the top of the stack. For example in the following Lst. 3.5 the parameters 0 and 2 of `add_func` are constant. Usually one would expect the type to be changed to `(param i32) (result i32)` and the values being passed as the removed parameters to be dropped right before the call, but this is not possible using only drops. The problem is that a drop can only remove the top value from the stack and can therefore not remove the first and third values from the stack without removing the second one, as it would be needed before executing the call. Therefore, constant parameters are only removed "from the top" until there is a non-constant parameter. In this case only the last parameter can be removed.

```
(type $func_type (func (param i32 i32 i32) (result i32)))
(type $main_type (func (param i32) (result i32)))

(func $add_func (type $func_type)
;; type changed to: (param i32 i32) (result i32)
  local.get 0       ;; i32.const 3
  local.get 1
  i32.add
  local.get 2       ;; i32.const 5
  i32.add
)
(func (export "main") (type $main_type)
  i32.const 3
```

```
14    local.get 0
15    i32.const 5
16    call $add_func    ;; drop; call $add_func
17  )
```

Lst. 3.5: Constant parameter restriction with annotated changes

In conclusion, there are 2 restrictions for removing constant parameters. If the function is referenced by any indirect calls, then no parameters are removed. In any other case the constant parameters are only removed "from the top". This second restriction is caused by prepending drops that are supposed to be removed in the third optimization, mentioned in Section 3.4. If all instructions leading to constant parameters were to be removed directly instead, then this second restriction would no longer apply. Def-use chains[2] would allow detecting these removable instructions.

Finally, some references to locals have to be shifted according to the removed parameters. There is also an edge case where a local instruction could still reference a removed parameter. This case can only occur for unreachable instructions, as they are not recognized as constant by Sturdy. Therefore, those instructions are replaced by the special `unreachable` instruction which also preserves the correct stack shape signature. This instruction is used more extensively in the deadcode optimization section. The following listing shows both of these cases. The local `$loc1` will have an index of 0 after removing the parameter of the function, which is why its references need to be shifted.

```
1  (type $func_type (func (param i32) (result i32)))
2  (memory $memory 1)
3  (func $func (type $func_type) (local $loc1 i32)
4  ;; type changed to: (result i32)
5    local.get 0       ;; i32.const 0
6    if (result i32)
7      local.get 0     ;; unreachable
8    else
9      i32.const 0
10   end
11   i32.load
12   local.set 1       ;; local.set 0
13   local.get 1       ;; local.set 0
14 )
15 (func (export "main") (type $func_type)
16   i32.const 0
17   local.get 0
18   i32.store
19
20   i32.const 0
21   call $func        ;; drop; call $func
22 )
```

Lst. 3.6: Local reference shift and unreachable

3.2.2 Implementation

The base visitor defined in Lst. 3.1 is extended by the `ConstantReplacer` class for constant instructions and by a private subclass `ConstantParameterRemover` for the parameters. The indices of constant parameters for every function are collected during the constant instruction optimization and are passed as arguments to run the `ConstantParameterRemover` as the last step when running the `visitModule()` method of `ConstantReplacer`.

ConstantReplacer class

The following global mutable Maps are defined for the sole purpose of being passed as arguments to the constant parameter remover:

1. `noGetParams: Map[FuncIdx, mutable.Set[Int]]`: Contains the indices of parameters whose referencing `local.get` instructions are declared constant.
2. `setParams: Map[FuncIdx, mutable.Set[Int]]`: Contains the indices of parameters which are referenced by `local.set` or `local.tee` instructions.

The actual argument passed to `ConstantParameterRemover` is the difference of `noGetParams` without `setParams`. This difference indicates all unused parameters after performing the constant instruction optimization.

The following private helper functions are defined:

1. `getConstantValue(funcIdx)`: Returns the constant value reported by the Sturdy analysis for the current instruction, at the position of the implicit global `funcPc` counter. If the instruction isn't constant `None` is returned instead.
2. `getFuncParams(funcIdx)`: Returns the function parameter types.

The following listing shows the implementation of `visitFuncInstr`. Scala pattern matching is performed on the visited instruction. The matches for if, block, and loop instructions just visit their bodies and perform no optimizations. For all other matches the optimization is performed as explained in Section 3.2.1. The values for the global `noGetParams` and `setParams` are also collected in this step when optimizing an applicable instruction targeting locals defined by parameters.

```
1 override def visitFuncInstr(funcInstr: Inst, funcIdx: FuncIdx): Seq[
    Inst] =
2   funcPc += 1
3   funcInstr match
4     case If(tpe, thenInstr, elseInstr) => Seq(
5       If(tpe, thenInstr.flatMap(visitFuncInstr(_, funcIdx)), elseInstr.
            flatMap(visitFuncInstr(_, funcIdx)))
6     )
```

```
7       case Block(tpe, blockInstr) => Seq(Block(tpe, blockInstr.flatMap(
            visitFuncInstr(_, funcIdx))))
8       case Loop(tpe, loopInstr) => Seq(Loop(tpe, loopInstr.flatMap(
            visitFuncInstr(_, funcIdx))))
9
10      case unop: (Unop | Testop | LoadInst) => getConstantValue(funcIdx)
            match
11        case Some(const: Inst) => Seq(Drop, const)
12        case None => Seq(unop)
13
14      case binop: (Binop | Relop) => getConstantValue(funcIdx) match
15        case Some(const: Inst) => Seq(Drop, Drop, const)
16        case None => Seq(binop)
17
18      case LocalGet(localIdx) => getConstantValue(funcIdx) match
19        case Some(const: Inst) =>
20          if (localIdx < getFuncParams(funcIdx).size) then noGetParams(
                funcIdx) += localIdx
21          Seq(const)
22        case None => Seq(LocalGet(localIdx))
23
24      case LocalSet(localIdx) if (localIdx < getFuncParams(funcIdx).size)
             =>
25        setParams(funcIdx) += localIdx
26        Seq(LocalSet(localIdx))
27      case LocalTee(localIdx) if (localIdx < getFuncParams(funcIdx).size)
             =>
28        setParams(funcIdx) += localIdx
29        Seq(LocalTee(localIdx))
30
31      case getInst: GlobalGet => getConstantValue(funcIdx) match
32        case Some(const: Inst) => Seq(const)
33        case None => Seq(getInst)
34
35      case selectInst: Select.type => getConstantValue(funcIdx) match
36        case Some(const: Inst) => Seq(Drop, Drop, Drop, const)
37        case None => Seq(selectInst)
38
39      case call: (Call | CallIndirect) => getConstantValue(funcIdx) match
40        case Some(const: Inst) =>
41          val paramSize: Int = call match
42            case Call(callFuncIdx) => getFuncParams(callFuncIdx).size
43            case CallIndirect(typeidx) => mod.types(typeidx).params.size
                  + 1 // +1 because of the table index
44          Seq.fill(paramSize)(Drop) ++ Seq(const)
45        case None => Seq(call)
46
47      case _ => Seq(funcInstr)
```

Lst. 3.7: The constant instruction removal implementation

ConstantParameterRemover class

The following global variables are defined for the constant parameter removal process:

1. `updatedTypes: mutable.ArrayBuffer[FuncType]`: Contains all function types with the addition of new types with removed parameters.
2. `remParams: mutable.Map[FuncIdx, Set[Int]]`: Contains all passed constant parameters without those of imported functions and functions contained in tables and also only contains sequential parameters taken from the end.
3. `funcTypeMapping: mutable.Map[FuncIdx, TypeIdx]`: Contains the mapping of functions to the new types where it is needed.
4. `callDrops: mutable.Map[FuncIdx, Int]`: Contains the amount of drops needed for a call targeting a function.

`updatedTypes` is initialized with all original function types, while the other 3 variables are initialized empty. The following listing shows the class constructor code that updated all 4 global variables. `constParams: Map[FuncIdx, Set[Int]]` contains all unused parameters collected during the constant instruction optimization. At first this Map is filtered by removing all elements referencing imported functions or functions contained in tables, as they are not supported. Then only sequential parameter indices starting from the end are collected as `actualRemParams` and later appended to `remParams`. The required `newFuncType` is generated accordingly. `funcTypeMapping` first checks if the required function type already exists in `updatedTypes`. Otherwise, it is appended.

```
1  constParams.filterNot((funcIdx: FuncIdx, paramIndices: Set[Int]) =>
2    paramIndices.isEmpty ||
3    // Only non imported functions that aren't referenced in any table (
         indirect call parameters removal not supported)
4    (funcIdx < mod.imported.funcs.size) || mod.elem.exists{ case Elem(_,
         _, init) => init.contains(funcIdx)}
5  ).foreach((funcIdx: FuncIdx, paramIndices: Set[Int]) =>
6    val oldFuncType: FuncType = mod.types(mod.funcs(funcIdx - mod.
         imported.funcs.size).tpe)
7    // Only remove constant parameters sequentially from the back until
         one is not constant
8    val actualRemParams = (oldFuncType.params.size-1).to(0, -1).takeWhile
         (paramIndices.contains).toSet
9    val newFuncType: FuncType = FuncType(
10     oldFuncType.params.zipWithIndex.filterNot((_, idx) =>
           actualRemParams.contains(idx)).map(_._1),
11     oldFuncType.t
12   )
13   remParams(funcIdx) = actualRemParams
14   funcTypeMapping(funcIdx) = updatedTypes.indexOf(newFuncType) match
15     case -1 => // case when the new required type with removed
           parameters doesn't exist yet
16       updatedTypes += newFuncType
17       updatedTypes.size - 1
18     case typeIdx: TypeIdx => typeIdx
19   callDrops(funcIdx) = oldFuncType.params.size - newFuncType.params.
         size
20 )
```

Lst. 3.8: Constant parameter type and drop collection

Within `visitFuncInstr`, the instructions are pattern matched. Calls and local instructions are changed as mentioned in section 3.2.1 by prepending drops to calls, shifting local references and replacing unreachable local instructions:

```
case Call(calledIdx: FuncIdx) =>
  val dropAmnt = callDrops.getOrElse(calledIdx, 0)
  totalAddedDrops += dropAmnt
  Seq.fill(dropAmnt)(Drop) ++ Seq(Call(calledIdx))
case LocalGet(localIdx) =>
  if remParams.getOrElse(funcIdx, Set.empty).contains(localIdx) then
      Seq(Unreachable)
  else Seq(LocalGet(localIdx - remParams.getOrElse(funcIdx, Set.empty).
      count(_ < localIdx)))
// LocalSet and LocalTee are identical
```

Lst. 3.9: Call and locals implementation

All other overwritten methods of the class update the types of the module to `updatedTypes` and apply `funcTypeMapping` for every function.

3.3 Deadcode

3.3.1 Idea

Dead Instructions

The dead instructions reported by the constant analysis of Sturdy represent all unreachabel instructions during any possible execution with any combination of input values. A naive approach would be to remove all dead instructions, as this wouldn't affect the actual stack or the memory at all during execution. However, the static signature of blocks, loops or if constructs might no longer match the statically calculated stack shape during validation.

The following Lst. 3.10 shows such an example. All instructions after the inner block are dead, since the `br 1` instruction always jumps over them while returning 87 for the outer block. Nonetheless, the instructions can't simply be removed as the static validation for the outer block doesn't account that the jump is always executed and expects the stack to contain an i32 when calculating the changes to the stack shape sequentially for every instruction.

```
(block (result i32)
  (block
    i32.const 87
    br 1
  )
  i32.const 3       ;; unreachable
  i32.const 2       ;; \
  i32.add           ;; \
)
```

Lst. 3.10: Deadcode removal necessary unreachable

Fortunately the special `unreachable` instruction has a valid type of `[t1*] -> [t2*]` for any possible sequence of value types t1 and t2. This means that ending a block with an `unreachable` always satisfies the required stack signature. If the `unreachable` were to be reached during execution an unconditional trap would be thrown and the execution would be terminated.

Taking this into account when dealing with instructions inside of blocks, the optimization replaces the first dead instruction with an unreachable and removes every following instruction in the same most inner block body. Since jumps are only possible to predefined labels, it is impossible for any instructions following the first dead instruction to be reachable, provided they are in the body of the same most inner block. Everything mentioned in this paragraph also counts for the other label defining constructs, which are loops and if constructs.

Lst. 3.10 therefore replaces the first dead instructions with an unreachable and removes the last two instructions completely. If there was one instruction in the inner block after the `br 1` it would also be replaced by an unreachable. This would technically not be needed since the inner block has no return value, so removing the instruction would cause no problems. Nonetheless, this edge case for control flow constructs without a return value is currently not implemented and an `unreachable` is always added for the first dead instruction.

Unnecessary control flow structures

Blocks and loops whose labels are never jumped to according to the analysis, and are therefore reported as dead labels, can be removed. In this case the deadcode optimization is applied to its body and the entire contruct replaced by those optimized instructions. The entire code in Lst. 3.11 would therefore be optimized to just `i32.const 8` without the block.

```
1 (block
2   i32.const 8
3 )
```

Lst. 3.11: Unnecessary block

`If` constructs with a condition always defaulting to either true or false can have their unreachable branch be discarged entirely. The entire construct is then replaced by the optimized instructions of the only reachable branch. If the label is not dead, then all instructions are additionally enclosed in a block with the same type signature as the original if construct. Since the if construct originally consumed one value from the stack as a conditional, a single drop is prepended.

```
1 i32.const 1
2 if (result i32)
3   i32.const 4
4 else
```

```
  i32.const 5
end

;; is optimized to:

i32.const 1
drop
i32.const 4
```

Lst. 3.12: Dead if branch with dead label

```
i32.const 0
if (result i32)
  i32.const 5
else
  i32.const 2
  br 0
  drop
  i32.const 4
end

;; is optimized to:

i32.const 1
drop
(block (result i32)
  i32.const 2
  br 0
  unreachable
)
```

Lst. 3.13: Dead if branch with alive label

`br_if` instructions that never jump, according to the control flow graph, can be removed and replaced with a drop.

```
(block
  i32.const 0
  br_if 0           ;; drop
  nop
)
```

Lst. 3.14: Br_if that never jumps

The label index for branch instructions has to be shifted in some cases when control flow constructs are removed to ensure that they still reference the same label. The following listing shows an example.

```
(block
  nop
  (block
    nop
    br 1
  )
)

;; is optimized to:

```

```
(block
  nop
  nop
  br 0
)
```

Lst. 3.15: Branch label shift

Dead functions

A dead function can be recognized as a function whose first instruction is dead. In some cases performing the constants optimizations can cause a function to become dead, like the function `const_func` in Lst. 3.3.

Dead functions are entirely removed. Since functions are referenced by index, removing one could require shifting all other references to functions, such as those in direct calls. More details are provided in the following implementation section.

3.3.2 Implementation

The deadcode optimization class extends the basic visitor. Apart from the dead instructions and dead label reports from Sturdy, an `ifTargets` Map is passed, which contains information on the behavior of `if` constructs and `br_if` instructions based on the outgoing edges of them according to the control flow graph. An `IfTarget` enum is calculated for every `if` and `br_if` instruction with the following possible values and meanings:

- For `if` instructions:
  - AllAlive: Both then- and else cases are reachable.
  - SingleInstructionTarget: Only one case is reachable (then or else).
  - EndLabelTarget: All reachable cases are empty (always goes to end of the entire `if` construct).
- For `br_if` instruction:
  - AllAlive: Both jumping and not jumping are reachable.
  - SingleInstructionTarget: The condition is never reached (never jump).
  - EndLabelTarget: Always goes to an EndLabel (unknown whether condition is always reached or never).
  - LoopJumpTarget: Always jumps to a loop (condition always reached).

It should be noted that this information always allows determining the behavior of `if` constructs exactly but is unable to do so for `br_if` instructions in the case of `ifTarget = EndLabelTarget`. Such a case occurs when the control flow graph reports that the end of a label, which is the end of a control flow construct, always follows after the instruction. In the case of `br_if` this could either mean that the jump always occurs to the end of a block/if or the jump never occurs, but it is the last instruction of a block/loop/if construct. In the case that the jump never occurs the instruction could be removed, but since these cases can't be differentiated in the current implementation using the control flow graph alone, the `br_if` instruction is never removed in the case of `ifTarget = EndLabelTarget`.

The following helper functions and variables are created:

- `var blockIsDead: Boolean`: A mutable variable that indicates if an unreachable has already been placed and all following instructions of the current block are are dead.
- `val deadFunctions: Seq[FuncIdx]`: An immutable sequence containing all dead functions.
- `def shiftFuncIdx(funcIdx): FuncIdx`: Shifts funcIdx to account for the removal of dead functions.
- `def instrIsDead(funcIdx: FuncIdx, shift: Int = 0): Boolean`: Indicates whether the instruction at funcPc+shift in funcIdx is dead.
- `def flattenLabelIdx(lbl: LabelIdx, lblDepth: Int, deadLblbDepths: Vector[Int]): LabelIdx`: Shifts label by amount of dead labels with a smaller index.
- `def visitFuncInstrCounter(funcInstr: Inst): Unit`: Increases funcPC without performing anything else.
- `def visitBlockBody(blockBody: Vector[Inst], funcIdx: FuncIdx, lblDepth: Int, deadLblDepths: Vector[Int]): Vector[Inst]`: Visits the block body and resets blockIdDead=false, since that global variable should only apply to the current block.

Dead functions are removed entirely.

```
override def visitFunc(func: Func, funcIdx: Int): Seq[Func] =
  if deadFunctions.contains(funcIdx) then Seq()
  else
    funcPc = -1
    blockIsDead = false
    Seq(Func(func.tpe, func.locals.flatMap(visitFuncLocal(_, funcIdx)),
         func.body.flatMap(visitFuncInstrExtended(_, funcIdx))))
```

Lst. 3.16: Dead functions removal

Instead of using `visitFuncInstr` from the base visitor an extended version with parameters for tracking dead label depths is used, which can be seen in Lst. 3.17. `lblDepth` indicates the current depth of nested block/loop/if constructs. A entry in `deadLblDepths` indicates the absolute position in the stack height of a removed label for the purpose of possibly shifing referenced labels of branch instructions. For example in Lst. 3.15 inside the inner function `lblDepth` would be 1 and `deadLblDepths` would only contain 0. Dead instructions are removed or replaced by an unreachable as it is mentioned in Section 3.3.1.

```
1  private def visitFuncInstrExtended(funcInstr: Inst, funcIdx: FuncIdx,
       lblDepth: Int = 0, deadLblDepths: Vector[Int] = Vector.empty[Int]):
        Seq[Inst] =
2    if blockIsDead then
3      visitFuncInstrCounter(funcInstr)
4      Seq.empty[Inst]
5    else if instrIsDead(funcIdx, 1) then
6      visitFuncInstrCounter(funcInstr)
7      blockIsDead = true
8      Seq(Unreachable)
9    else
10     funcPc += 1
11     funcInstr match {...}
```

Lst. 3.17: Dead instruction removal

Otherwise, pattern matching is performed on `funcInstr`. Blocks with dead labels are removed and replaced by their optimized body. The dead label depths are updated in this case. Loops are optimized in the same way.

```
1  case Block(tpe, blockInstr) =>
2    deadLabelMap.get(funcIdx) match
3      case Some(instrLocMap: Map[LabelInst, Seq[InstrIdx]]) if (
           instrLocMap(LabelInst.Block).contains(funcPc)) =>
4        visitBlockBody(blockInstr, funcIdx, lblDepth + 1, deadLblDepths.
           appended(lblDepth + 1))
5      case _ => Seq(Block(tpe, visitBlockBody(blockInstr, funcIdx,
         lblDepth + 1, deadLblDepths)))
```

Lst. 3.18: Block optimization

`If` constructs are optimized according to `IfTargets`, which can be seen in Lst. 3.19. If it is reported as `AllAlive` or `EndLabelTarget`, the constructs remains with an optimized body or gets removed entirely with its body respectively. Otherwise, it is known that only one of the branches (then or else) is reached during execution. In this case it is checked if `deadLabelMap`, which is the Sturdy report of control flow structures which are never jumped to, reports the instruction of having a dead label, to determine if its contents should be enclosed in a block.

```
1  case If(tpe, thenInstr, elseInstr) =>
2    val ifTarget = try
3      ifTargets(funcIdx)(funcPc)
4    catch
```

```
5       case e: NoSuchElementException => throw new NoSuchElementException(
            s"If instruction in line=${funcPc} is alive yet isn't contained
             in ifTargets Map!")
6     if (ifTarget == AllAlive) then
7       Seq(If(tpe,
8         visitBlockBody(thenInstr, funcIdx, lblDepth + 1, deadLblDepths),
9         visitBlockBody(elseInstr, funcIdx, lblDepth + 1, deadLblDepths)))
10    // Edge case where Then and Else branches are empty from the
          beginning or the only reached one is empty
11    else if (ifTarget == EndLabelTarget) then
12      (thenInstr ++ elseInstr).foreach(visitFuncInstrCounter)
13      Seq(Drop)
14    // One of the branches is dead and ifTarget == InstructionTarget
15    else
16      val thenBranchDead: Boolean = (thenInstr.isEmpty || instrIsDead(
            funcIdx, 1))
17      deadLabelMap.get(funcIdx) match
18        // Label is dead
19        case Some(instrLocMap: Map[LabelInst, Seq[InstrIdx]]) if (
              instrLocMap(LabelInst.If).contains(funcPc)) =>
20          val innerInstr =
21            if thenBranchDead then
22              thenInstr.foreach(visitFuncInstrCounter)
23              visitBlockBody(elseInstr, funcIdx, lblDepth + 1,
                    deadLblDepths.appended(lblDepth + 1))
24            else
25              val theninst = visitBlockBody(thenInstr, funcIdx, lblDepth
                    + 1, deadLblDepths.appended(lblDepth + 1))
26              elseInstr.foreach(visitFuncInstrCounter)
27              theninst
28          Seq(Drop) ++ innerInstr
29        case _ =>
30          val innerInstr =
31            if thenBranchDead then
32              thenInstr.foreach(visitFuncInstrCounter)
33              visitBlockBody(elseInstr, funcIdx, lblDepth + 1,
                    deadLblDepths)
34            else
35              val theninst = visitBlockBody(thenInstr, funcIdx, lblDepth
                    + 1, deadLblDepths)
36              elseInstr.foreach(visitFuncInstrCounter)
37              theninst
38          // If the if instruction is neither dead nor its label but one
                of the branches is empty, then it can be removed
39          // but we need to replace it with a block since its label is
                references by some branch
40          Seq(Drop) ++ Seq(Block(tpe, innerInstr))
```

Lst. 3.19: If optimization

Otherwise, the referenced label indices for branches are shifted accordingly and `br_if` instructions are removed according to the reported value for `ifTargets`. Indices for function calls are also shifted.

```
1 case Call(callFuncidx: FuncIdx) => Seq(Call(shiftFuncIdx(callFuncidx)))
2 case Br(lbl: LabelIdx) => Seq(Br(flattenLabelIdx(lbl, lblDepth,
      deadLblDepths)))
```

```
case BrIf(lbl: LabelIdx) =>
  // Note: For the case of EndLabelTarget can't determine whether the
      condition is always reached or never
  if (ifTargets(funcIdx)(funcPc) == SingleInstructionTarget ||
      deadLblDepths.contains(lblDepth-lbl)) then
    Seq(Drop)
  else
    Seq(BrIf(flattenLabelIdx(lbl, lblDepth, deadLblDepths)))
case BrTable(table: Vector[LabelIdx], lbl: LabelIdx) =>
  Seq(BrTable(table.map(flattenLabelIdx(_, lblDepth, deadLblDepths)),
      flattenLabelIdx(lbl, lblDepth, deadLblDepths)))

case _ => Seq(funcInstr)
```

Lst. 3.20: Rest instructions

Function indices for exports and the start section, which can contain at most 1 function, also have to be shifted.

```
override def visitExport(exprt: Export): Seq[Export] = Seq(Export(exprt
    .fieldName, exprt.kind, shiftFuncIdx(exprt.index)))
override def visitStart(start: FuncIdx): FuncIdx = shiftFuncIdx(start)
```

Lst. 3.21: Export function shift

References to dead functions must be removed from tables. While this might seem like a simple optimization at first, it is more complicated since multiple references to functions can be inserted sequentially with an offset, as a single elem of the *elem* section of a module. The solution is to split such an elem into two parts: one containing functions before the removed function and the other containing those after. The second elem's offset is adjusted to maintain function positions within the table, as those are critical for indirect calls. Lst. 3.22 shows such an example. The first argument of the elem indicates the table offset.

```
(elem (i32.const 1) $f1 $dead_func $f3)

;; split to:

(elem (i32.const 1) $f1)
(elem (i32.const 3) $f3)
```

Lst. 3.22: Example: Elem split for removed function

```
override def visitElem(elem: Elem): Seq[Elem] =
  elem.init.zipWithIndex.foldLeft(Seq[Option[Elem]](None)) {
    case (acc: Seq[Option[Elem]], (funcIdx: FuncIdx, elemIdx: Int)) =>
      if deadFunctions.contains(funcIdx) then
        acc.appended(None)
      else
        acc.updated(acc.size-1, acc.last match
          case Some(lastElem: Elem) => Some(Elem(lastElem.table,
              lastElem.offset, lastElem.init.appended(shiftFuncIdx(
              funcIdx))))
          case None =>
            if elemIdx == 0 then
```

```
11                Some(Elem(elem.table, elem.offset, Vector(shiftFuncIdx(
                      funcIdx))))
12              else
13                // Per spec the offset can only be a single constant
                      vector. Imported immutable globals are also allowed
                      which could
14                // cause an error here. This is okay for now since
                      imports are not supported by the optimization yet
15                val newOffset = elem.offset match
16                  case Vector(i32.Const(value: Int)) => Vector(i32.Const(
                        value+elemIdx))
17                  case Vector(glob: GlobalGet) => throw new
                        IllegalArgumentException("Imported global in elem
                        not allowed for this optimization!")
18                Some(Elem(elem.table, newOffset, Vector(shiftFuncIdx(
                      funcIdx))))
19          )
20    }.flatten
```

Lst. 3.23: Elem split implementation

3.4 Drops

3.4.1 Idea

The drops optimization aims to remove the chain of instructions that lead to pushing a value being dropped, without serving any other purpose. Since the stack changes performed by instructions are static, it is possible to determine every instruction that is responsible for producing the top value on the stack by going backwards from a drop instruction in a control flow graph. A naive approach would be to simply remove all of those instructions, including the drop instruction. For Lst. 3.24 this would result in removing all instructions with the exception of `local.get 0`, which would be correct and optimal for this simple example.

```
1  (func (export "main") (param i32 i32) (result i32)
2    local.get 0
3    local.get 1
4    i32.const 2
5    i32.add
6    drop
7  )
```

Lst. 3.24: Drops naive removal

Lst. 3.25 shows an example where the naive approach doesn't work. The example is similar to an example mentioned in Section 2.2 of the paper [11]. This paper introduces the stack types *mnd* and *opt* to indicate whether a value on the stack is mandatory or optional. The same types are used as comments in the listing for the stack. Values in this case are declared as optional if they are

only used in the chain of instructions producing a dropped value. Mandatory values in this example are the values consumed by `if` and `br_if` as conditions and the two values consumed by `i32.store`. Using only the naive approach of declaring every value leading to the drop as optional, one would assume that every instruction producing the optional returned value of the if construct, which are the `local.get 1` and both `local.get 2` instructions, are optional. This is however not the case since the first `local.get 2` instruction produces a mandatory value for the `store` instruction, in case the `br_if` instruction doesn't jump. Therefore, this instruction can't be removed and its result has to be returned for the if construct, in case the `br_if` instruction does jump. This means that the if construct must necessarily return a value and the last `local.get 2` and the `local.get 1` also can't be removed.

```
(memory $memory 1)
(func (export "main") (param i32 i32 i32 i32)
  i32.const 9       ;; []
  local.get 0       ;; [opt]
  (if (result i32)  ;; [mnd, opt]
    (then
      local.get 1   ;; []
    )
    (else
      local.get 2   ;; []
      local.get 3   ;; [mnd]
      br_if 0       ;; [mnd, mnd]
      i32.const 8   ;; [mnd]
      i32.store     ;; [mnd, mnd]
      local.get 2   ;; []
    )
  )
  i32.add           ;; [opt, opt]
  i32.const 4       ;; [opt]
  i32.sub           ;; [opt, opt]
  drop              ;; [opt]
)
```

Lst. 3.25: Drops removal problem

This last example shows that only performing a control flow analysis backwards from a drop is not sufficient, as there can be branches in the control flow. The paper [11] suggests that a bidirectional analysis is necessary. This would allow performing a forward analysis on the first `local.get 2` instruction, which would detect that its value is mandatory for the store instruction. Another possible approach could involve the use of def-use chains[2] for drops and instructions requiring mandatory values.

3.4.2 Implementation

The actual implementation for this thesis is simple but restrictive. The optimization is performed like the naive approach until a control flow instruction

is encountered, prompting the optimization to stop prematurely while appending drops to ensure a correct stack signature. In such restricted cases the mentioned problem can't occur. For Lst. 3.24 this performs the optimization optimally. For Lst. 3.25 all instructions after the if construct are removed and two drops are inserted in place of the `i32.add` instruction to preserve the stack signature. This is not the most optimal optimization since the first `i32.const 9` produces one of the optional values for the optional `i32.add` instruction and is therefore unnecessary.

A new `DropsAnalysis` class is initialized with all reversed edges from the Sturdy analysis as input. A `CfgNode` is a node of the control flow graph representing instructions and other possible types which are not relevant for the optimization.

```
1 private class DropsAnalysis(revEdges: Map[CfgNode, Seq[CfgNode]])
```

There are two global mutable collections as variables:

1. `remInst: mutable.Set[CfgNode]`: Will contain a set of all removable instructions nodes.
2. `neededDrops: mutable.Map[CfgNode, Int]`: Will contain a Map of needed drops that should be appended for an instruction node.

The following function exists to track changes to the stack for an instruction node. For every non-supported instruction, which include the control flow instructions and also stores, an exception is thrown. This exception is later caught.

```
1  /**
2   * Changes on the stack performed by the instruction of node
3   * @return (consumed amount, produced amount, StackType)
4   */
5  def stackChange(node: CfgNode): (Int, Int, StackType) = node match
6    case CfgNode.Instruction(inst: Inst, _) => inst match
7      // nop would technically be (0, 0, StackType.Opt) but it doesn't
           but it doesn't make a difference for this implementation
8      case _: (Unop | Testop | Convertop | LoadInst | LoadNInst) | Nop =>
           (1, 1, StackType.Opt)
9      case _: (Binop | Relop) => (2, 1, StackType.Opt)
10     case _: AConst | LocalGet(_) | GlobalGet(_) => (0, 1, StackType.Opt
           )
11     case Select => (3, 1, StackType.Opt)
12     case Drop => (1, 0, StackType.Opt)
13     case _ => throw new StackChangeUnsupported(s"Instruction type of
           $node not supported")
14   case _ => throw new StackChangeUnsupported(s"Instruction type of
         $node not supported")
```

Lst. 3.26: Stack change function

Then `processBackwards` is executed for every drop instruction that isn't already contained in `remInst`. This function updates the global `remInst` and `neededDrops` variables. `revEdges(node)` returns a list of all preceding nodes of

`node` according to the control flow graph. The function is executed recursively for all preceding nodes until a `StackChangeUnsupported` exception is thrown by `stackChange`.

```
def processBackwards(node: CfgNode, stack: Seq[StackType] = List.empty[
    StackType]): Unit =
  try {
    val (consumed, produced, stackType) = stackChange(node)
    val newStack =
      if stackType == StackType.Opt then
        if produced == 0 then
          List.fill(consumed)(StackType.Opt) ++ stack
        else // produced == 1
          List.fill(consumed)(stack.head) ++ stack.tail
      else ???

    remInst += node
    if newStack.nonEmpty then revEdges(node).foreach(processBackwards(_
        , newStack))
  } catch {
    case e: StackChangeUnsupported =>
      neededDrops(node) = stack.size
  }
```

Lst. 3.27: Drop removal backwards analysis

Finally, the instructions represented by the nodes in `remInst` are removed and drops are prepended accordingly to `neededDrops` using a simple visitor class.

4 Evaluation

To test the correctness of each individual optimization, unit tests were performed with assertions on manually created Wasm files with their expected optimized result. These test cases were chosen to cover most edge cases and restrictions of the current implementation. The code examples provided in every idea section of this thesis are all based on a subset of those tests. All tests ran successfully.

Next all optimizations were applied sequentially to a subset of 1000 files from WasmBench [4]. The subset was already chosen by Sturdy-Wasm [1] to be used for benchmarks and is therefore proven to work with the analysis. Since analyzing larger files can potentially require a significant amount of time, a timeout of 10 minutes was chosen and all files were ordered by size. The resulting file after applying all optimizations was also validated with Swam to further rule out optimization errors.

The order that was chosen for all optimizations in this evaluation was: Constants -> Dead code -> Drops. This order was chosen deliberately as the constant optimization can introduce new dead code by removing all calls to a function, as can be seen in Lst. 3.3. The drops optimization has to be the last one as all other optimizations add new drop instructions.

679 files were optimized in total, which excludes all optimizations that timed out and were dropped due to timing constraints. The validation succeeded for every one of those files. While this doesn't prove the correctness of all implemented optimizations, it rules out a large amount potential errors since WebAssembly validation is especially strict. The evaluation data includes for each optimization: A count of all instructions after performing the optimization (denoted by `*InstCnt`) and a count of all added drops (denoted by `*Drops`). For the dead code optimization all added unreachables were also counted as `deadUnr`. The percentage of reduced instruction counts was also calculated for each instruction individually and in total (denoted by `%red_*`).

The results in Table 4.1 show a particularly high standard deviation (std), which is mostly caused by a large amount of small files and total instruction counts ranging from 6 to 16277. Table 4.2 shows the results for only the last/largest 200 files, where the standard deviation is significantly lower.

In total, it has to be noted that instruction counts after every optimization also include all added drops. Therefore, the constant optimization can only increase the amount of instructions. The actual amount of replaced constant instructions wasn't counted in this test, but it would be equivalent to the reported constant instruction by Sturdy-wasm [1].

Tab. 4.1: Optimization results on all 679 files

	origInstCnt	constInstCnt	deadInstCnt	dropsInstCnt
mean	6038.23	6083.26	3918.22	3838.46
std	6489.04	6532.76	4610.12	4520.51

	constDrops	deadDrops	deadUnr	dropsDrops
mean	45.04	2.23	7.97	8.24
std	49.07	3.43	9.22	10.17

	%red_const	%red_dead	%red_drops	%red_total
mean	-2.655123	19.72	6.81	22.67
std	9.32	26.53	15.33	27.60

Tab. 4.2: Optimization results on last/largest 200 files

	origInstCnt	constInstCnt	deadInstCnt	dropsInstCnt
mean	13828.26	13920.01	9766.11	9569.13
std	1354.59	1363.59	1188.39	1164.65

	constDrops	deadDrops	deadUnr	dropsDrops
mean	91.76	4.96	19.08	20.55
std	14.81	0.86	3.29	2.87

	%red_const	%red_dead	%red_drops	%red_total
mean	-1.31	29.58	2.12	30.53
std	0.21	5.95	1.72	5.96

Dead code optimization has by far the most impact on removing instructions with an average of about 30% on the reduced dataset and 20% on the complete one. The amount of added unreachables is negligible with an average of 19 on the reduced dataset and a maximum of 39 in total on the entire dataset. These instructions are never reached during execution and therefore add no computational overhead.

The added drops after a drops optimization indicates all leftover drops. These exist because of the restrictive implementation that was chosen for this thesis. While the total amount might not seem to be too significant, the potential for removing additional instructions might not reflect this number. Performing an optimal optimization on a single drop could potentially lead to a large chain of instructions to be removed. If function calls are among them, then the saved computational expenses might be even larger.

5 Related work

5.1 Other optimizations based on Webassembly

Binaryen [@13] is a widely used compiler/toolchain library and optimizer written in C++ for WebAssembly. Modules are parsed to the Binaryen IR unto which optimization passes are performed. Dead code removal, constant propagation and constant parameter removal are all implemented alongside many other optimizations, such as merging of duplicate instructions and inlining of functions.

There are papers suggesting additional optimizations [7] and also questioning the use of function inlining [10]. Inlining has been shown to be counterintuitive in some cases for WebAssembly in particular since it doesn't account for the way compilation is usually handled in browsers. Usually browsers use more optimized compilers for frequently called functions. If those functions were to be inlined instead the compilation could be prevented, which could negatively affect performance.

5.2 Optimizations on stack based languages

There exist various optimization techniques for stack based languages but not all of them may be applicable to WebAssembly. One example is a stack scheduling optimization [8] [6], which uses stack manipulation instruction such as duplications to avoid multiple loads of the same variable. The duplication instruction doesn't exist in WebAssembly so the most similar way to implement this optimization would be to define a new local as a cache for the value instead, but there is no guarantee that this would be more efficient in a real life scenario for WebAssembly.

There was also research on optimizing stack based code via type systems [11]. The "load-pop pairs elimination" in particular is interesting for the drops optimizations mentioned in this thesis. The paper stresses the necessity of a bidirectional analysis for such a problem but also mentions that these aren't necessarily more complex than unidirectional ones [5].

Def-use chains [2] were already mentioned in this thesis as a possible approach for the drops optimization and a more complete constant parameter optimization. A def-use chain contains the definition of a variable and every instruction involved in its usage. For the drops optimization def-use chains

could be applied to all drops and also to all instructions requiring mandatory values on the stack, such as stores. For the constant parameter removal they could be used for the removed parameters of function calls.

6 Conclusion and future work

In conclusion, the constant analysis reports from Sturdy-Wasm were successfully applied to perform practical optimizations. Some of the problems that could occur when trying to apply those reports naively were shown, like the removal of dead instructions that could break the signature of blocks. It was also shown that not all analysis reports could always lead to an optimization. For example not all if constructs with a dead label can be optimized, as both then- and else branches might still be alive.

The implemented dead code optimization was shown to be effective in removing instructions. The constant instruction optimization replaces all reported constant instructions, which makes it as effective as the analysis reports [1]. The drops optimization was only implemented in a restrictive manner and has the most potential for improvement. Some ideas were mentioned, like the usage of a bidirectional analysis or def-use chains.

Another potential improvement would be to expand all optimizations to support imports from other modules. Sturdy supports running an analysis over multiple modules by specifying the definition of every imported function. This could expand the optimization to be applicable to more modules.

Bibliography

[1]Katharina Brandl, Sebastian Erdweg, Sven Keidel, and Nils Hansen. “Modular Abstract Definitional Interpreters for WebAssembly”. In: *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 5:1–5:28 (cit. on pp. v, 1, 9, 31, 35).

[2]R. Castillo, F. Corbera, A. Navarro, R. Asenjo, and E. L. Zapata. “Complete Def-Use Analysis in Recursive Programs with Dynamic Data Structures”. In: *Euro-Par 2008 Workshops - Parallel Processing*. Ed. by Eduardo César, Michael Alexander, Achim Streit, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 273–282 (cit. on pp. 15, 28, 33).

[3]Andreas Haas, Andreas Rossberg, Derek L. Schuff, et al. “Bringing the web up to speed with WebAssembly”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 185–200 (cit. on p. 3).

[4]Aaron Hilbig, Daniel Lehmann, and Michael Pradel. “An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases”. In: *Proceedings of the Web Conference 2021*. WWW ’21. Ljubljana, Slovenia: Association for Computing Machinery, 2021, pp. 2696–2708 (cit. on pp. 1, 31).

[5]Uday P. Khedker and Dhananjay M. Dhamdhere. “A generalized theory of bit vector data flow analysis”. In: *ACM Trans. Program. Lang. Syst.* 16.5 (Sept. 1994), pp. 1472–1511 (cit. on p. 33).

[6]Philip Koopman. “A preliminary exploration of optimized stack code generation”. In: (1992) (cit. on p. 33).

[7]Zhibo Liu, Dongwei Xiao, Zongjie Li, Shuai Wang, and Wei Meng. “Exploring Missed Optimizations in WebAssembly Optimizers”. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 436–448 (cit. on p. 33).

[8]Martin Maierhofer and M Anton Ertl. “Optimizing stack code”. In: *Forth-Tagung, Ludwigshafen* (1997) (cit. on p. 33).

[10]Alan Romano and Weihang Wang. “When Function Inlining Meets WebAssembly: Counterintuitive Impacts on Runtime Performance”. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. San Francisco, CA, USA: Association for Computing Machinery, 2023, pp. 350–362 (cit. on p. 33).

[11]Ando Saabas and Tarmo Uustalu. “Type Systems for Optimizing Stack-based Code”. In: *Electronic Notes in Theoretical Computer Science* 190.1 (2007). Proceedings of the Second Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2007), pp. 103–119 (cit. on pp. 27, 28, 33).

Webpages

[@9]Programming Languages @ JGU Mainz. *sturdy.scala*. URL: `https://gitlab.rlp.net/plmz/sturdy.scala` (visited on Feb. 23, 2025) (cit. on p. 9).

[@12]Lucas Satabin. *swam*. URL: `https://swam.gnieh.org/` (visited on Feb. 23, 2025) (cit. on pp. 3, 10).

[@13]WebAssembly. *binaryen*. URL: `https://github.com/WebAssembly/binaryen` (visited on Feb. 23, 2025) (cit. on pp. 1, 33).

List of Figures

2.1 WebAssembly abstract syntax [3] 3

List of Tables

4.1 Optimization results on all 679 files 32
4.2 Optimization results on last/largest 200 files 32

List of Listings

2.1 Concrete stack during execution 4
2.2 Deterministic stack shape 5
2.3 Block jump and infinite loop 7

3.1 The base module visitor 11
3.2 Constant add with annotated stack shape 12
3.3 Constant direct calls with annotated changes 13
3.4 Constant parameter indirect call with annotated changes 14
3.5 Constant parameter restriction with annotated changes 14
3.6 Local reference shift and unreachable 15
3.7 The constant instruction removal implementation 16
3.8 Constant parameter type and drop collection 18
3.9 Call and locals implementation 19
3.10 Deadcode removal necessary unreachable 19
3.11 Unnecessary block 20
3.12 Dead if branch with dead label 20
3.13 Dead if branch with alive label 21
3.14 Br_if that never jumps 21
3.15 Branch label shift 21
3.16 Dead functions removal 23
3.17 Dead instruction removal 24
3.18 Block optimization 24
3.19 If optimization 24
3.20 Rest instructions 25
3.21 Export function shift 26
3.22 Example: Elem split for removed function 26
3.23 Elem split implementation 26
3.24 Drops naive removal 27
3.25 Drops removal problem 28
3.26 Stack change function 29
3.27 Drop removal backwards analysis 30

Colophon

This thesis was typeset with LaTeX 2$_\varepsilon$. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.

Adaptations to the style of the Institute of Computer Science can be found at `https://gitlab.rlp.net/institut-fur-informatik/cleanthesis-jgu`.

Declaration of Authorship

I hereby declare that I have written the present thesis independently and without use of other than the indicated means. I also declare that to the best of my knowledge all passages taken from published and unpublished sources have been referenced. The thesis has not been submitted for evaluation to any other examining authority, nor has it been published in any form whatsoever. I duly noted the Regulations for Good Scientific Practice and Dealing with Scientific Misconduct.

Mainz, March 20, 2025

Tadeo Volker Hees