UTRECHT UNIVERSITY

MASTER THESIS

Superoptimization of WebAssembly Process Graphs

Author:
Dennis G. SPROKHOLT

Supervisor:
Prof. Dr. G.K. KELLER

2nd Supervisor:
Dr. S.W.B. PRASETYA

A thesis submitted in partial fulfilment (25 ECTS) of the requirements for the degree of Master of Science

in the

Department of Information and Computing Sciences

ICA-6605877

Abstract

Superoptimization of WebAssembly Process Graphs

by Dennis G. SPROKHOLT

The time needed for program execution is rarely minimal. Often, a faster program exists that produces the same output. Our superoptimizer aims to reduce the execution time of WebAssembly programs significantly. We propagate symbolic information over control flow, partially evaluate expressions and branch conditions using the Z3 SMT solver, synthesize alternate fragments for some small loops, and apply small structural changes to loops with low bounds. In particular, our approach of propagating symbolic information over control flow and *driving* loops with symbolic information is novel for superoptimization. Loop driving and synthesis make small artificial programs *several orders of magnitude* faster. On large programs, only partial evaluation already requires *multiple hours* of optimization time while finding only marginal (~1%) improvements. While the complete approach shows some potential, its application to large programs is currently infeasible, as it may require *months* of superoptimization time.

Acknowledgements

My supervisors Gabriele and Wishnu,
for their guidance, patience, and countless reviews.

My friends and family,
for their sympathy and support.

Contents

List of Abbreviations

AST	**A**bstract **S**yntax **T**ree
CEGIS	**C**ounter**E**xample-**G**uided **I**nductive **S**ynthesis
CFG	**C**ontrol **F**low **G**raph
CPU	**C**entral **P**rocessing **U**nit
FFI	**F**oreign **F**unction **I**nterface
GI	**G**enetic **I**mprovement
GMA	**G**uarded **M**ulti-**A**ssignment
GP	**G**enetic **P**rogramming
HNF	**H**ead **N**ormal **F**orm
IR	**I**ntermediate **R**epresentation
MCMC	**M**arkov **C**hain **M**onte **C**arlo
SAT	Boolean **Sat**isfiability Problem
SMT	**S**atisfiability **M**odulo **T**heories
SSA	**S**ingle **S**tatic **A**ssignment
WASM	**W**eb**A**sse**m**bly

Chapter 1

Introduction

Programmers describe a series of steps that allow a computer to solve a particular problem. The used programming language mainly prescribes the granularity of such steps. About this granularity, Perlis amusingly remarked:

> "A programming language is low level when its programs require attention to the irrelevant."
>
> —Alan J. Perlis[45]

While many may agree with this statement, it remains debatable what exactly is *irrelevant* in its context. For instance, most programmers opt for platform-agnostic languages[1]; which indicates that few programmers prefer to write architecture-specific machine instructions. Similarly, many programs are written in languages with *garbage collection*, meaning that many consider manual memory management irrelevant. As languages become higher-level - together with the programs written with them - programmers increasingly trust compilers and runtime systems to fill in the gaps; that is, ensure programs written at a high level of abstraction perform similarly to those same programs written at a lower level. While compiler technology has improved significantly over the years, none can claim these abstractions come entirely without cost. To alleviate programmers from focussing on the irrelevant, our superoptimizer *aims* to reduce the execution time of programs to a *minimum* so that programmers no longer have to; or at least, only to a lesser extent.

Every computer program takes time to execute. Yet, often a *faster* program exists that performs the same computation. Program optimizers aim to transform a program into another which takes less time (or resources) to produce an identical result. The relevance of program optimization was already understood before computers - as we know them - existed[2]:

> "In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them for the purposes of a Calculating Engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation."
>
> —Ada Lovelace (Note D)[20]

Unfortunately, program optimizers almost never produce *optimal* programs. This is because optimizing compilers apply a finite range of pre-defined transformations[36],

[1] `https://madnight.github.io/githut/` - GitHub language statistics

[2] Charles Babbage never completed the construction of his Analytical Engine, for which Ada Lovelace proposed the first algorithm.

which are never exhaustive[3]. Figure 1.1 depicts the space of all programs, within which a compiler generates a program.

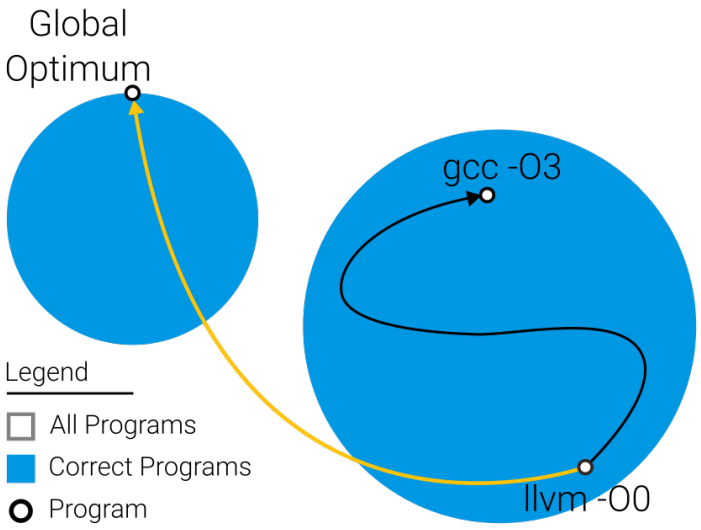


FIGURE 1.1: Abstract depiction of search space (adapted from [52])

The blue regions depict the set of programs that are correct by the specification. While an existing compiler may find a *reasonably fast* program in the space, a non-trivial transformation leading to the global optimum is out of reach. Think of the left region - which is out of reach - as the implementation of a wholly *different algorithm* that solves the same problem. As existing optimizers are incapable of reaching this optimum, a *superoptimizer* can attempt to cross this chasm.

1.1 Superoptimization

In 1987, Massalin created the first superoptimizer[37]. It *exhaustively searches* through the set of all programs - in increasing length - until a correct program is found. Massalin's superoptimizer produces optimal programs in the Motorola 68020 instruction set. The optimal program computing the *signum* function is shown in Listing 1.1.

$$\text{signum}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

LISTING 1.1: Massalin's optimal `signum` program[37]

```
(x in d0)
add.l d0,d0     |add d0 to itself
subx.l d1,d1    |subtract (d1 + Carry) from d1
negx.l d0       |put (0 - d0 - Carry) into d0
addx.l d1,d1    |add (d1 + Carry) to d1
(signum(x) in d1) (4 instructions)
```

This program - consisting of only 4 instructions - is shorter (and faster) than the straightforward implementation with conditional branches, which consists of 9 instructions.

Unfortunately, this superoptimizer does not *solve* program optimization. After all, the search space of programs scales exponentially with the program length,

[3] If these transformations *were* sufficient to obtain optimality, any non-terminating program (without side-effects) could be optimized to a trivial loop; which solves the halting problem.

which makes finding larger optimal programs very computationally expensive. Additionally, it is generally undecidable to determine the equivalence between two looping or recursive programs. However, Massalin's paper sparked much subsequent research in the area of superoptimization.

Later superoptimizers built upon this work by providing alternate search strategies for non-looping code fragments (e.g., stochastic[52], symbolic[21], or enumerative[47]). These approaches have in common that non-looping fragments are extracted from the programs and independently superoptimizer; after which these fragments are placed back into the program. As the superoptimization of fragments does *not* take into account their location and *context* within the program, some optimizations may be missed. After all, program fragments need only produce correct results for the states on which they are executed.

Souper[50] is a recent superoptimizer, which optimizes LLVM[36] IR. Arteaga et al.[11] included Souper in a pipeline to superoptimize C programs before converting them to WebAssembly. Souper is particularly effective at synthesizing linear instruction sequences. However, it only minimally considers the context of these fragments within the full program.

While superoptimization of loop-free programs is well researched, few superoptimizers handle loops or recursion. Although the generation of a looping program is trivial - by generating backward branching statements - their verification is problematic. One method[53] for verifying equivalence between two looping programs was added to STOKE[52]'s verifier. It relies upon the notion of cutpoints[62], which requires one program to *simulate* the other. Intuitively, a program simulates another if both encounter the same states at the cutpoints. This relation is only preserved when both programs have similar loop structures. This requirement inhibits many improvements from being found.

A full account of previous research on superoptimization is included in section 6.1.

1.2 Process Graphs

We apply superoptimization to programs containing arbitrary control flow by applying simple behavior-preserving structural changes. Our superoptimizer infers the *context* of contained program fragments from information propagation over this control flow. To achieve this, we rely upon the notion of *process graphs*[61, 19, 28], which are finite graphs that represents *all* execution traces of an (imperative) program[4]. An execution trace is the sequence of statements encountered while concretely executing the program. Multiple execution traces can be grouped together, which are parametric over their input variables; this can be represented through a - potentially infinite - *process tree*[19]. Figure 1.2 displays an example partial process tree.

Every node in the process tree corresponds to a set of *program states*, each of which describes a particular concrete assignment to every program variable. This set of possible states at any node can be perfectly determined from an initial states and the execution trace reaching it; this is called *perfect information propagation*[19]. This information makes process trees particularly useful for *program specialization*[28], as redundant statements can be eliminated and infeasible branches omitted. For superoptimization, this information allows finding better optimizations; after all, an improved instruction sequence need only be correct for the set of states upon which it

[4]Intuitively, it looks like a control flow graph where the *edges* contain the instructions

is executed. For example, any instruction sequence following the node marked with p in Figure 1.2 can be optimized with the knowledge that $x < 2$ (until x changes).

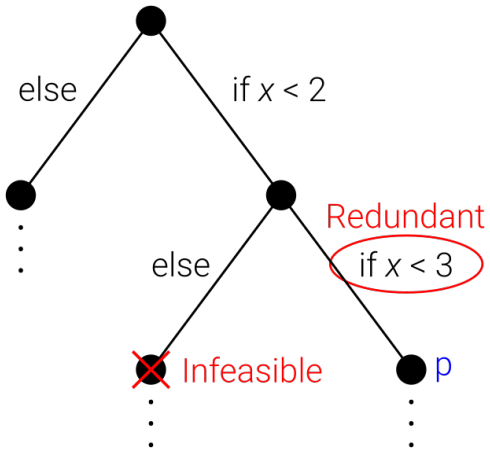


FIGURE 1.2: Example Process Tree
(for some imperative pseudo language)

As process trees are potentially infinite, they do not directly correspond to a finite program. However, process trees can always be converted into an a finite process graph[19, 61] (which often sacrifices some optimization).

1.3 WebAssembly

WebAssembly[23, 48] is a safe, fast, compact, and portable low-level language. Its main design goal is to provide near-native performance on the Web, while observing its security model[2]. Safety is important as code is obtained from untrusted sources, which should not compromise the executor. WebAssembly is an abstraction over modern hardware; thus, it can easily be translated into machine code for common instruction sets (e.g., x86, ARM64). WebAssembly computation is based on a *stack machine*; this ensures a compact binary representation[55], while its absence of register count favors no particular instruction set. While low-level, WebAssembly enforces *structured control flow* through its abstract syntax; this ensures by construction that control flow cannot form irreducible loops or branch into the middle of a multi-byte instruction.

As WebAssembly is a relatively recent language whose specification is frequently extended[1], only few tools are available and these are still under active development[66]. As WebAssembly programs often under-perform compared to their native counterparts[26], better optimization techniques are very beneficial. Thus, our superoptimizer specifically targets WebAssembly programs; however, the proposed techniques should be applicable to any (imperative) language.

1.4 Research Questions

Previous superoptimizers are limited by failing to consider the *context* within which instruction sequences appear, and fail in superoptimizing program fragments containing loops. Now that the previous sections established the notion of superoptimization and process graphs, the research questions - which aim to address these issues - can be understood.

- **RQ1: How can superoptimization be extended to handle statically bounded loops?** Usually, *high-level observations* can be made over loops which either allow the loop to be entirely eliminated, or assist in finding improvements in a fraction of its iterations. In this research, non-terminating loops are of lower interest, as these occur infrequently in WebAssembly programs[5].
  - *RQ1A: How can superoptimization be applied to a process tree?* Process trees contain non-looping execution traces, whose superoptimization should be similar to linear execution sequences. The propagation of information to nodes in this tree, as well as the extension of these trees, should allow for better optimizations than previously possible.
  - *RQ1B: How can these optimizations be applied to a finite process graph?* Programs are generally required to have a finite size. The *potentially infinite* process trees rarely correspond to a program. Process graphs, however, are finite structures which similarly capture the execution traces. Applying the optimizations to a process graph should assist in producing a valid WebAssembly program.
- **RQ2: How effective are these techniques in improving the performance of WebAssembly functions?** The objective of the developed superoptimizer is to reduce the execution time of WebAssembly programs. So, it is necessary to systematically determine whether the execution time of any WebAssembly function decreases after it is superoptimized.

1.5 Structure

Two unrelated concepts are brought together in this thesis; namely *superoptimization* and *process graphs*. Chapter 2 gives some preliminary definitions. In chapter chapter 3, superoptimization is explained irrespective of process graphs, but instead assumes non-looping instruction sequences. Following that, chapter 4 extends superoptimization to process graphs. Chapter 5 describes the empirical evaluation of the superoptimizer and its generated programs, which is mainly performed through benchmarking. In chapter 6, related work is given. Chapter 7 answers the research questions. Finally, chapter 8 describes possibilities for future research.

[5]Currently, modern web browsers *abort* programs several dozen seconds after their execution to avoid resource over-use, which surely happens to non-terminating programs.

Chapter 2

Preliminaries

This chapter establishes several preliminary definitions and notational conventions.

2.1 Programming Languages

Convention 2.1 (Code Segment Notation)**.** Code fragments which illustrate optimizations are written in either the WebAssembly text format or the Rust[38] programming language. While the research pertains to WebAssembly superoptimization, its text format is not always intuitive to convey ideas. For those cases, Rust is used instead. Rust supports primitive types (`i32`,`i64`,`f32`,`f64`) equal to WebAssembly's, and its compiler ecosystem targeting WebAssembly is quite mature[1].

LISTING 2.1: Sample Rust Function

```
fn mul2( x: u32 ) -> u32 {
  x * 2
}
```

LISTING 2.2: Sample WebAssembly Function

```
(func $mul2 (param $x i32) (result i32)
  get_local $x
  i32.const 2
  i32.mul
)
```

Convention 2.2 (Source Code)**.** The superoptimizer artifact is written in the *Haskell* programming language. Occasionally, details on the implementation are easier explained through its source code.

LISTING 2.3: Sample Haskell Function

```
mul2 :: Int -> Int
mul2 x = x * 2
```

[1] https://rustwasm.github.io/docs/book/

2.2 A WebAssembly Introduction

The superoptimizer aims to superoptimize WebAssembly programs. This procedure involves transforming WebAssembly programs, checking equivalence between program fragments, and generating alternate fragments, to name a few. All those operations rely heavily on the WebAssembly specification[48]. While reiterating the specification in its entirety is surely futile, delineating some essentials should aid the understanding of important decisions later on; this explanation is given below.

2.2.1 Stack machine

WebAssembly programs are defined using a structured *stack machine*. Instructions interact with the stack by popping its inputs before pushing its outputs. For example, the `i32.add` instruction pops two `i32` (32-bit integer) values from the stack, adds them together (modulo 2^{32}), and pushes that result back to the stack. Listing 2.4 displays an example WebAssembly program.

LISTING 2.4: Some function `$f`

```
(func $f (result i32)
  i32.const 4  ;; push 4
  i32.const 5  ;; push 5
  i32.add      ;; pop 5. pop 4. push 9
  i32.const 3  ;; push 3
  i32.mul      ;; pop 3. pop 9. push 27
)
```

LISTING 2.5: Sugared `$f`

```
(func $f (result i32)
  (i32.mul
    (i32.add
      (i32.const 4)
      (i32.const 5)
    )
    (i32.const 3)
  )
)
```

The function `$f` starts with an empty stack. Then 4 and 5 are pushed to the stack. `i32.add` pops those operands, computes $4 + 5 = 9$, and pushes 9 to the stack. Then, after pushing 3 to the stack, the stack contains $[9, 3]$ (right-most is the top). Finally, `i32.mul` pops those values, computes $9 \cdot 3 = 27$, which is pushed back to the stack. The function returns the stack's top value - being 27 - as its result.

Listing 2.5 shows an alternative notation of the same program as Listing 2.4, where the program is denoted as an S-expressions. Often, S-expressions are easier to understand. However, instructions producing multiple results cannot generally be written in the S-expression format. Either notation may be used throughout this document.

2.2.2 Structured Programming & Labels

WebAssembly operates as a *structured* stack machine, which implies programs must adhere to *structured control flow*. Intuitively, a structured program may contain instruction (1) sequences, and both (2) `if-else` and (3) `loop` constructs, but *no* (arbitrary) `goto` statements. Those three constructs are sufficient to represent any computable function[7]. While structured programming is commonplace in recent high-level languages, it is less so in low-level languages. For languages written and read by programmers, (abundant) use of `goto` statements make programs hard to understand[16]. For machine languages that problem does not exist, as few programmers write it directly. WebAssembly, however, is both *structured* and low-level. WebAssembly's program structure eliminates some static errors, such as branches

into the middle of a multi-byte instruction. Additionally, *streaming* validation and compilation become easier[23], which enables web browsers to start compilation to machine-code before the file is fully downloaded.

Listing 2.6 shows an annotated WebAssembly program containing an if-statement. Whenever a block (`if`, `block`, `loop`) statement is encountered, a *label* is pushed to the stack. The label effectively delineates the block *scope*; instructions within the block cannot modify values below the label, as those belong to its outer scope. Upon exiting a scope, the label is popped again. Blocks may have inputs and outputs. Before entering a block, its parameters are popped from the stack. After pushing the scope label, those parameters are pushed again. Similarly, upon exiting, the results are popped before popping the label. After popping the label, those results are pushed to the stack again.

LISTING 2.6: Function with an if-statement

```
(func (result i32)
  i32.const 9  ;; push 9
  i32.const 5  ;; parameter passed to if-statement
  i32.const 1  ;; condition (true)
  if (param i32) (result i32) ;; pop 1. pop 5. push label. push 5.
    i32.const 3  ;; push 3
    i32.mul      ;; pop 3. pop 5. push 15
  end  ;; pop 15. pop label. push 15.
  i32.add  ;; pop 15. pop 9. push 24.
)
```

Break statements While *arbitrary* jumps are disallowed, jumps to surrounding scopes *are* allowed; This is done through `br`, `br_if`, and `br_table` instructions. Listing 2.7 shows a program with a conditional break (`br_if`) statement.

LISTING 2.7: Function with conditional break

```
1 (func (param $x i32) (result i32)
2   block $B0 (result i32)
3     i32.const 20  ;; push 20
4     i32.const 10  ;; push 10
5     get_local $x  ;; push the value of $x
6     br_if $B0     ;; break if $x != 0
7     drop          ;; drop 10
8   end
9 )
```

Within the block, 20 and 10 are pushed to the stack. Then the value of local[2] variable `$x` is pushed to the stack; if that value is true ($\neq 0$), it breaks, and execution resumes *after* the block. As block `$B0` must return a `i32` value, before breaking, 10 is preserved to the other scope. Any other values remaining on the stack (being 20) are discarded with the label. However, if `$x` is false ($= 0$), the break is *not* performed and execution resumes at line 7. There, 10 is explicitly dropped from the stack. Then, block `$B0` naturally exits with 20 as its result.

While labels are typically given identifiers (e.g., `$B0`) in the text format, in the binary format, labels are referenced by an index. This index denotes the number of

[2]Locals are explained in subsection 2.2.3

labels between the referencing instruction and the referenced label; this is akin to *de Bruijn indices*[14]. In the example, `$B0` on line 6 may thus be replaced by 0, as there are no other scopes between the break statement and block `$B0`.

2.2.3 Locals

Every function has a *fixed* number of local variables. The function parameters are also considered local variables, whose value is provided by the function caller. Non-parameter local variables are initialised to their respective zero values upon function entry. All local variables are mutable. Consider Listing 2.8 below.

LISTING 2.8: Function with local variables

```
(func (param $x i32) (param $y i32) (result i32) (local $z i32)
  (tee_local $z
    (i32.add
      (get_local $x)
      (get_local $y)
    )
  )
)
```

Local variables `$x` and `$y` are function parameters, and generally no assumptions can be made about their values. `$z` is a non-parameter local variable, and is initialized to 0 upon function entry. In the binary format, variables are referenced by their index. Indices may also be used in the text format, though names are usually more readable. `$x` has index 0, `$y` has index 1, and `$z` has index 2.

2.2.4 Globals

A WebAssembly module may contain global variables. Any function may access the global variables. Global variables are statically determined; that is, no new global variables can emerge at runtime. Global variables are statically defined as either mutable or immutable. Consider Listing 2.9.

LISTING 2.9: Program with global variables

```
(module
  (global $p i32 (i32.const 3))
  (global $q (mut i32) (i32.const 0))

  (func (param $x i32)
    (set_global $p
      (i32.add
        (get_global $q)
        (get_local $x)
      )
    )
  )
)
```

The global variables are `$p` and `$q`. Both have type `i32`, though only `$q` is mutable. As `$p` is immutable, its value may never change at runtime. Similarly to locals,

global variables may also be referenced by their index (in order of definition); `$p` has index 0 and `$q` has index 1.

2.2.5 External Functions

While WebAssembly modules are frequently called *programs*, these modules cannot be executed on their own. Instead, WebAssembly modules are always executed in a *host environment*. The host may invoke (exported) functions defined within the module. WebAssembly functions may invoke functions exposed by the host environment. Consider Listing 2.10.

LISTING 2.10: Program with global variables

```
(module
  (import "env" "print" (func $print (param i32)))
  (func (export "foo")
    (call $print
      (i32.const 42)
    )
  )
)
```

In this module, the host exposes a print function (`env.print`), which accepts a single `i32` value. It is the host's responsibility to provide a sensible implementation for that function; which likely involves printing the value to the console. WebAssembly has no other means of accessing the outside world. This mechanism ideally provides a perfect *sandbox* within which the module runs, as any outside access must be explicitly granted.

Often, a web browser acts as a host, where a WebAssembly module is spawned by a JavaScript program. This way, websites can spawn performant applications on-demand. Compilers (e.g., Emscripten[65]) usually generate WebAssembly together with corresponding JavaScript "glue code", which web-developers use to interact with the module.

Though, WebAssembly is not limited to the web, as desktop runtimes[3] also exist. In those cases, a user program encapsulating the runtime acts as the host, which guards system interaction, such as system calls for file access.

In practice, WebAssembly programs often have *several dozens* of imported functions; each of which handles a specific interaction with the host. Examples of these interactions are printing to the user console, writing to a WebSocket network connection, or updating a display component.

[3] See: https://github.com/bytecodealliance/wasmtime or https://wasmer.io/

2.2.6 Memory

WebAssembly programs may optionally contain a block of linear memory, as illustrated by Listing 2.11. Memory consists of a vector of uninterpreted bytes.

LISTING 2.11: Module with memory

```
(module
  (func $f
    (f64.store
      (i32.const 99)   ;; address
      (f64.const 42.0) ;; value
    )
  )
  (memory (export "mem") 128 256)
)
```

This module has a memory block consisting of at least 128 and at most 256 memory pages; each of which has a size of 64 KiB. At runtime, the size of memory may grow upon request through the `grow_memory` instruction. The function `$f` writes the binary representation of the 64-bit float 42.0 to memory at index 99. Memory may also be read from or written to by *the host*. Note, though, that this memory is independent from the host's memory; meaning a WebAssembly program can never access the host's memory.

2.2.7 Uninterpreted Integers

WebAssembly programs operate on variables which are either *uninterpreted integers* (`i32`, `i64`) or floating-point numbers (`f32`, `f64`).

> **Definition 2.1** (Uninterpreted integer)**.** *Uninterpreted integers* are integers whose signedness interpretation varies depending on context[48]. For some operations, signedness does not matter; for example, for addition. Whereas for other operations the signedness is important, in which case it is enforced by the *instruction*; for example, `i32.lt_u` versus `i32.lt_s` for unsigned and signed variants of the less-than instruction, respectively.

2.3 Definitions

In this section, we give definitions for several terms used within this thesis.

Any computer program can eventually be executed on some concrete machine which exists in our reality. As the program executes, this reality changes. This reality is modelled as an infinite sequence of *worlds*.

Definition 2.2 (World)**.** The world is all that is the case[64], which is the totality of facts. (Except the internal state)

A *world* thus represents the state of *all that is* at some *specific point in time*. The world that *is now* is different from the world that *was* 2 seconds ago. A running program can observe facts about the world, or transform the world into another. For convenience, the program's internal state (e.g., stack and memory) is *not* considered part of the world. Observation of the world, as well as transformations of the current world, are considered *side effects* of the program.

Definition 2.3 (Side Effect)**.** A side effect is an action that observes or transforms the world outside the local environment. For WebAssembly, this is performed through *host calls*. The host system can explicitly provide access to host functions[48]. The WebAssembly instance may call these functions to change the system state or interact with peripheral devices.

Examples of side effects are writing to a file, or launching nukes.

The *internal* state of the program (which is *not* considered part of the world), is defined as follows:

Definition 2.4 (Program State)**.** A program state s is defined as a triple (K, M, G). K represents the program stack. As WebAssembly's semantics are defined over a stack machine, this stack contains primitive values (of type `i32`,`i64`,`f32`, or `f64`), labels, and *activations*. Activations are the *call frames* of active functions. Labels represent block scopes on the stack. M is the *program memory*; which is a bounded vector of bytes whose length is always a multiple of $64 \cdot 1024$ (which is the page size). G is the vector of *global variables*.

This definition roughly corresponds to its definition in the WebAssembly specification[48]. WebAssembly implicitly enforces that a memory block cannot be larger than 2^{33} bytes; as memory is addressed by 32-bit indices added to another 32-bit constant offset. Whenever a stack is explicitly stated, it grows from *left to right*.

Example 2.1 (Stack)**.** Consider the stack listed below:

$$(\texttt{f32.const}\ 42.0)\ (\texttt{i32.const}\ 99)$$

The 32-bit integer 99 is *on top* of the stack, while the 32-bit float 42.0 is the second (and bottom) element of the stack.

A program transitions through multiple program states by executing instructions. Every concrete execution of a program follows an *execution trace*.

Definition 2.5 (Execution Trace)**.** An *execution trace* is the *sequence* of instructions that is traversed during program execution from a *single* initial state. This sequence need not be finite; which is the case for non-terminating programs.

At any program point there is a limited set of states that can *possibly* occur. Simultaneously, many states *cannot* occur at that program point; as no execution trace to that program point exists which produces those states. For example, it may be the case that variable `x` always contains an *even* integer at some program point p. The knowledge of possible states at a program point is essential for improving performance. A set of such states is called a *configuration*[61] (Jones[28] calls these *stores*).

Definition 2.6 (Configuration)**.** A *configuration* $\mathcal{C}$ is a *set of states* that have identical *static structures*. Two states s_1 and $_2$ have identical static structures if all the following conditions hold:

- Their stacks have an identical number of elements and pairwise[4] these elements have the same static structure; which involves:
  - For primitive values, both have identical primitive types
  - For activations, the number of local variables are identical and their elements pairwise have identical types.
- Their number of global variables are identical and pairwise these variables have identical types.

The structural restriction is placed upon configurations as no two states with unequal structures can occur at the same program point; this is enforced by validation[48] of WebAssembly programs.

Every concrete execution of a program transitions through a sequence of program states. When considering a process tree - which contains *all* execution traces - every node corresponds to a configuration. A program can thus be fully described through its execution traces between configurations and worlds. This was observed by Turchin[61], and later adopted by Jones[28] in his formal system of *program specialization*. In this context, an abstract program is defined as follows:

Definition 2.7 (Abstract Program)**.** An *abstract program*[28] is a quintuple $\pi = (P, S, W, \rightarrow, p_0)$, where $p_0 \in P$ and $\rightarrow \subseteq (P \times W \times S) \times (P \times W \times S)$. P is the set of *program points*. S is the set of *states*. W is the set of *worlds*. $\rightarrow$ is the *translation relation*, which is written in *infix* notation[5]. p_0 is the *initial program point*.

Note that this definition differs from Jones's[28], as it includes the set of *worlds*. This opaque model of worlds is used to represent side-effects of abstract programs (See subsection 3.2.7). The concrete execution of such a program is defined as follows:

Definition 2.8 (Concrete Execution)**.** An execution $\mathbf{exec}(\pi, w_0, s_0)$ of some program π (where $\pi = (P, S, W, \rightarrow, p_0)$), for some valid $s_0 \in S$ and $w_0 \in W$ corresponds to a finite or infinite sequence:

$$(p_0, w_0, s_0) \rightarrow (p_1, w_1, s_1) \rightarrow (p_2, w_2, s_2) \rightarrow \dots$$

Usually, execution halts in a terminal state; for WebAssembly, this returns control to the caller together with the produced results.

[4] pairwise P for vectors A and B of length n: $\bigwedge_i^n P(A_i, B_i)$
[5] Infix '$\rightarrow$': $(p_0, w_0, s_0) \rightarrow (p_1, w_1, s_1)$

Definition 2.9 (Terminal state)**.** A state pair (p, w, s) is *terminal* if $(p, w, s) \rightarrow (p', w', s')$ holds for no p', w', s'.

Definition 2.10 (Terminal relation)**.** $\rightarrow^*$ is the reflexive transitive closure of $\rightarrow$. $\rightarrow^F$ denotes the *terminal relation*. That is, $(p, w, s) \rightarrow^F (p', w', s')$ iff $(p, w, s) \rightarrow^* (p', w', s')$ *and* (p', w', s') is terminal.

Alternatively, the following notation may be used for program $\pi = (P, S, W, \rightarrow, p)$: $\mathbf{exec}^F(\pi, w, s)$, which produces (p', w', s'). This denotes the terminal state of program π when executed in world w from initial state s.

Note that $(p, w, s) \rightarrow^F (p', w', s')$ need not be defined for every (p, w, s); which happens for non-terminating programs. Furthermore, most programs are *deterministic*; which means:

Definition 2.11 (Determinism)**.** A program is *deterministic*[28] if for every $p \in P$, $w \in W$, $s \in S$, it holds that $(p, w, s) \rightarrow (p', w', s')$ and $(p, w, s) \rightarrow (p'', w'', s'')$ always means that $(p', w', s') = (p'', w'', s'')$.

Whenever the program is deterministic, the terminal state (p', w', s') is *unique* for (p, w, s) in $(p, w, s) \rightarrow^F (p', w', s')$.

In superoptimization, program fragments are replaced by faster program fragments. Clearly, this new fragment must "behave identically" to the original. Formally, this means the program fragments must be *extensionally equivalent* within a particular context. Typically, *extensional* equivalence is distinguished from *intensional* equivalence, which Example 2.2 illustrates.

Example 2.2 (Extensional vs Intensional equivalence)**.**
Consider the two functions below:

$$f(x) = (x + 2) * 3$$
$$g(x) = x * 3 + 6$$

The functions f and g have a different *internal structures*, as the order of operations and the used constant values differ; as their definitions are not syntactically equal, the functions f and g are *not* intensionally equal.

However, both f and g map values in their domain to *identical* values in their co-domain (So, $\forall x.f(x) = g(x)$); as their external properties match, f and g *are* extensionally equal.

Program fragments need only be extensionally equivalent for the set of states upon which the fragment is executed. For mathematical functions, extensional equivalence over some domain is defined as follows:

Definition 2.12 (Extensional Function Equivalence over Domain)**.** Two functions f and g are *extensionally equivalent* over some domain $\mathcal{D}$ if f and g map every input value $x \in \mathcal{D}$ to identical output values. That means:

$$f \equiv_{\mathcal{D}} g \quad \iff \quad \forall x \in \mathcal{D}.(\ f(x) = g(x)\ )$$

An example of this equivalence is as follows:

Example 2.3 (Extensional Function Equivalence over Domain)**.** Consider the functions:

$$f(x) = x^2$$
$$g(x) = x + 2$$

The functions f and g are extensionally equivalent over the domain $\mathcal{D} = \{-1, 2\}$: $f \equiv_{\mathcal{D}} g$. This is because $f(-1) = g(-1) = 1$ and $f(2) = g(2) = 4$. Note that the output values for any other input $x \notin \mathcal{D}$ do not affect this extensional equivalence over $\mathcal{D}$.

Convention 2.3 (Extensional Function)**.** Functions with a finite domain may be written *extensionally*. The function $f = \{3 \mapsto 4,\ 5 \mapsto 6\}$ is a function with domain $\{3, 5\}$, where $f(3) = 4$ and $f(5) = 6$.

Much of program equivalence checking involves logic formulas, which are mainly stated in *first-order logic with equality*. Below, conventional terminology for this domain is stated:

Definition 2.13 (Interpretation)**.** An *interpretation* is a mapping that assigns values to all free variables in a formula. This may also be referred to as a *model*.

Example 2.4 (Interpretation)**.** Consider the formula:

$$a < b \wedge b < c$$

One interpretation function is $u = \{a \mapsto 2,\ b \mapsto 1,\ c \mapsto 4\}$, as it assigns a sort-correct value to each variable. Note, though, that the formula is *not* true under this interpretation.

Definition 2.14 (Satisfiable)**.** A formula is *satisfiable* iff there exists at least one interpretation under which the formula is true.

Its inverse is:

Definition 2.15 (Unsatisfiable)**.** A formula is *unsatisfiable* iff there exists *no* interpretation under which the formula is true.

Definition 2.16 (Valid)**.** A formula is *valid* iff it is true under *every* interpretation.

Note that iff a formula A is *unsatisfiable*, its negation $\neg A$ is *valid*. As A is false under every interpretation, $\neg A$ is true under every interpretation.

Chapter 3

Program Synthesis

Superoptimization is a special case of program synthesis. Program synthesis involves the construction of a program satisfying some specification. For superoptimization, the source program embodies this specification.

This chapter elaborates on the main concepts and challenges within this synthesis process. Mainly, *loop-free* program fragments are considered for superoptimization, while chapter 4 applies these techniques to cyclic process graphs.

3.1 CounterExample-Guided Inductive Synthesis

Programs synthesis aims to find a program π for some given specification π_{spec} over some (large) set of input states $\mathcal{C}$ and a common set of worlds W, which satisfies:

$$\forall w \in W. \forall s \in \mathcal{C}. (\mathbf{exec}^F(\pi, w, s) = \mathbf{exec}^F(\pi_{spec}, w, s))$$

Assume only *terminating* programs without side-effects are considered[1]. The specification can be entirely described by a (possibly infinite[2]) set of input/output pairs which is the test set, that is obtained as:

$$T_{\mathcal{C}} = \{(s,\ \mathbf{exec}^F(\pi_{spec}, w, s)) \mid s \in \mathcal{C}\}$$

Finding a program π that is consistent with *all* these pairs is computationally expensive; as it requires execution of every candidate program π' on this set, which is likely already infeasible for a single candidate π'. To resolve this, the program can be made consistent with an *iteratively growing* set of input/output pairs. This corresponds to the prevalent synthesis technique: CounterExample-Guided Inductive Synthesis (CEGIS)[59, 58]. Figure 3.1 depicts its structure.

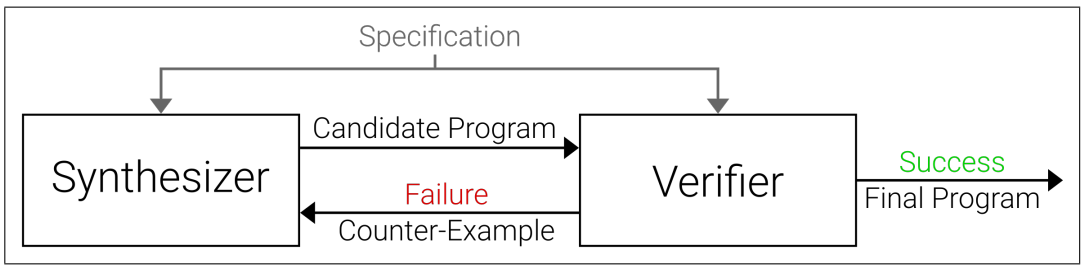


FIGURE 3.1: CounterExample-Guided Inductive Synthesis

[1] Side-effects are separately described in subsection 3.2.7

[2] For WebAssembly, this set is finite, but usually humongous.

Denote T_C^n as some particular set of test cases $T_C^n \subseteq T_C$ where $|T_C^n| = n$, for any $n \leq |T_C|$. Starting at $n = 1$, a candidate program π' is synthesised that is consistent with all test cases in T_C^n. Their correctness can be determined by executing π' on each test input. Once π' is consistent with all test cases in T_C^n, π' is given to a *verification oracle*. This oracle determines whether π' corresponds to the specification (i.e., it satisfies all test cases in T_C), or finds another test case t with which it is inconsistent. This produces another test set $T_C^{n+1} = T_C^n \cup \{t\}$, upon which the algorithm repeats. This continues until a correct program is found[3]. Note that n never exceeds $|T_C|$, as π_{spec} and π' match on all inputs when $n = |T_C|$.

The produced program becomes gradually consistent with an increasing number of counter examples. As it "learns" from these examples, the synthesis may be considered inductive.

3.1.1 Program Equivalence

This synthesis procedure relies upon some *verification oracle* to determine equivalence of two programs. Below, we define program equivalence. This definition relates to *extensional function equivalence* (Definition 2.12 on page 14). However, contrary to mathematical functions, programs may produce side-effects or run indefinitely. Thus, extensional equivalence of *program fragments* requires a more elaborate definition, which is given below:

Definition 3.1 (Extensional Program Equivalence over Configuration)**.** Two program fragments π_1 and π_2 are extensionally equivalent over a *configuration* $\mathcal{C}$ when *both* the following conditions hold for *every* state $s \in \mathcal{C}$ and every initial world w:

- Either both $\textbf{exec}(\pi_1, w, s)$ and $\textbf{exec}(\pi_2, w, s)$ terminate in the same state, or both run indefinitely.
- The *sequence of worlds* observed by $\textbf{exec}(\pi_1, w, s)$ and $\textbf{exec}(\pi_2, w, s)$ must be identical; this ensures both programs have identical side-effects.

WebAssembly programs can also *trap*. Trapping immediately aborts execution[48]. This happens, for instance, upon division by zero. Whether an execution trapped, must thus be included in the state representation. For the purpose of extensional equivalence, trapping is no different from successful program termination. The enforced equality between the sequences of observed worlds is stricter than it need be. However, as no knowledge about the behavior of side-effects is assumed, this is a safe assumption. (This is further elaborated in subsection 3.2.7)

3.2 Program Equivalence Checking

Now remains the *checking* of this equivalence. For the program fragments considered in the superoptimizer, bounded *symbolic execution*[32] is used to determine their equivalence. Both programs are symbolically executed, which produces a symbolic representation of their respective final program state. Effectively, every variable is assigned a formula (which is parametric over the input variables). Every such symbolic representation completely describes the input/output relation of the corresponding program. Equality between such states can be described through a logic

[3]Synthesis is undecidable; if no such program exists, it searches forever.

formula, equating the symbolic states. The Z3[15] SMT solver determines validity of such formulas, which means the programs are equal.

Intuitively, symbolic execution differs from concrete execution as follows: When a program is concretely executed, its input variables have concrete values (e.g., the value 42 for a `i32` variable). When a program is symbolically executed, its input variables are assigned *symbolic values*. A symbolic value can be considered to represent *any particular* value in a set of possible values. Through symbolic execution, a program can be executed for *multiple inputs* simultaneously. While this is similar to the previously-seen execution over configurations, it differs subtly yet crucially. The full process of symbolic execution and equivalence checking is elaborated below.

3.2.1 Symbolic State

While we previously used *configurations* to describe the possible states at program points, these are unsuitable for checking program equivalence. This is partially because configurations do *not* include information about the encountered worlds; but mainly because configurations disregard the *dependency* between a program's input and output. This issue is explained through the following example:

Example 3.1 (Symbolic State Rationale)**.** Consider symbolic execution of the two programs below, starting with the configuration $\mathcal{C}_I$ where x is represented by a symbolic value a, where $a \in \{5, 10\}$.

LISTING 3.1: Program π_1

```
// { Pre:  x = a ∧ ( a = 5 ∨ a = 10 ) }
x = 10 - x;
// { Post: x = 10 - a ∧ ( a = 5 ∨ a = 10 ) }
```

LISTING 3.2: Program π_2

```
// { Pre:  x = a ∧ ( a = 5 ∨ a = 10 ) }
x = x - 5;
// { Post: x = a - 5 ∧ ( a = 5 ∨ a = 10 ) }
```

After executing *both* programs on the same configuration $\mathcal{C}_I$, their final configuration $\mathcal{C}_O$ (containing two states) can be described as: $\{\{\mathrm{x} \mapsto 0\}, \{\mathrm{x} \mapsto 5\}\}$.

While both programs terminate in the same configuration $\mathcal{C}_O$, these two programs are *not* equal. With configurations, the *relation to the program input* is lost. After all, the input/output relation of variable x in π_1 is characterised by $\{5 \mapsto 5,\ 10 \mapsto 0\}$, while in π_2 it is characterised by $\{5 \mapsto 0,\ 10 \mapsto 5\}$.

Only checking for the equivalence of configurations is insufficient to determine program equivalence. Instead, through symbolic execution, programs are determined equal iff their outputs are equal for every input value. While a configuration may also be symbolically represented, it fundamentally differs from a symbolic state. A configuration simultaneous represents *all* contained states, while a symbolic state symbolically represents a single state which is parametric over its symbolic inputs (in this case a).

Symbolic states are defined similarly to concrete states (as defined in Definition 2.4).

Definition 3.2 (Symbolic State)**.** A *symbolic* program state $\mathcal{Y}$ is defined as a quadruple (K, M_d, M_s, G, W, T) over some environment $E_{\mathcal{C}}$. K is a symbolic representation

of the stack, containing primitive values and activations. M_d is a symbolic representation memory block, while M_s symbolically represents its size. G is the vector of symbolic global variables. W is the symbolic representation of the world. T is a symbolic boolean indicating whether the execution has trapped.

The *static structure* of the stack K is concretely defined (i.e., it is not parametric over any program inputs), whereas its contained primitive values and local variables are represented by symbolic values. This is similar for global variables, where only their values are symbolically represented.

The memory block is described with the two variables M_d and M_s. The reason for this dual description is quite subtle. In the logic, M_d is an *uninterpreted function* of type:

$$M_d : [\![33]\!] \rightarrow [\![8]\!]$$

$[\![33]\!]$ and $[\![8]\!]$ are bitvectors of size 33 and size 8, respectively. This function is defined over its entire domain, which corresponds to 2^{33} memory addresses. However, a program's memory is bounded by its number of assigned memory pages. This bound on the domain of M_d is externally enforced by M_s, which symbolically represents the memory's size. Any memory access at some address a is bounds-checked:

$$0 \leq a < M_s$$

When this check fails, program execution traps. So, together, M_d and M_s are sufficient to represent program memory.

Implementation of Symbolic States

The symbolic state (inside a function activation) from our superoptimizer's implementation in Haskell is listed in Listing 3.3.

LISTING 3.3: Symbolic State Implementation (in Haskell)

```
data SymbolicProgramState env =
  SymbolicProgramState {
    localState  :: Maybe (SymbolicLocalState env)
  , globals     :: [SymbolicGlobal env]
  , mem         :: Maybe (SymbolicMem env)
  , world       :: Symbolic env World
  }

data SymbolicLocalState env =
  SymbolicLocalState {
    isTrapped   :: Symbolic env Bool
  , activation  :: SymbolicActivation env
  , stack       :: SymbolicStack env
  }
```

The `localState` field reflects an important detail; namely, that the `SymbolicLocalState` is absent (`Nothing`) on paths that are *unconditionally trapped* upon encountering an `unreachable` statement. Consider the program below, in Listing 3.4 and Listing 3.5.

LISTING 3.4: Checked 0 division π_A

```
if x == 0 {
  memory[42] = 3;
  unreachable!( );
}
return 10 / x;
```

LISTING 3.5: Unchecked 0 division. π_B

```
return 10 / x;
```

These programs are subtly *unequal*. Surely, both programs trap whenever `x` has the value 0. However, π_A modifies memory before doing so. As memory updates are externally observable - even on trapped execution - it must be represented within the state. However, after trapping (unconditionally), execution halts and the stack values are dropped. Other execution paths (when $\texttt{x} \neq 0$) *do* continue execution. A symbolic representation is required that characterizes the full input/output relation of the program (over some $\mathcal{C}$). So, all final symbolic states at the end of execution paths must be merged under their respective (mutually exclusive) path conditions. For trapped executions the non-stack components (i.e., globals, memory, and world) are preserved. Figure 3.2 aims to illustrate this. Note that two states A and B are equal (over some $\mathcal{C}$) whenever:

$$
\begin{aligned}
&\text{globals}_A = \text{globals}_B \\
\wedge\ &\text{mem}_A = \text{mem}_B \\
\wedge\ &\text{world}_A = \text{world}_B \\
\wedge\ &\text{trapped}_A = \text{trapped}_B \\
\wedge\ &\neg\,\text{trapped}_A \rightarrow (\text{activation}_A = \text{activation}_B \ \wedge\ \text{stack}_A = \text{stack}_B)
\end{aligned}
$$

So, when *both* programs trapped for a particular execution, the stack and activation may be considered *vacuously* equal. As Figure 3.2 illustrates, the path conditions of unconditionally trapped branches are included in the merged final state. After all, in the merged state that path (when $\alpha = 0$) is no longer *unconditionally* trapped, but conditionally trapped under its path condition (being $\alpha = 0$). Merging states is further elaborated in subsection 3.2.3, after symbolic execution is established (in subsection 3.2.2).

The *reason* for keeping the local state absent, is because branches that trap unconditionally provide no guarantee on their stack state. Consider Listing 3.6.

LISTING 3.6: Branches with unequal stacks

```
if (result i32)
  (i32.const 3)
  (i32.const 2)
  unreachable ;; traps
else
  (i32.const 1)
end
```

While type-checking demands that the `if`-block produces a `i32` value, the `unreachable` instruction has *any* type, as it aborts execution. The branches have unequal stacks; in the if-case there are two `i32` values on the stack, while in the else-case there is only one. Still, this program is type-correct, as stacks are discarded upon trapping.

Merging stacks with unequal static structures is impossible. However, as the stack for the trapped execution is irrelevant anyway, it may be modeled by `Nothing`.

Memory is wrapped inside a `Maybe` for a much simpler reason: Not all programs have a memory block.

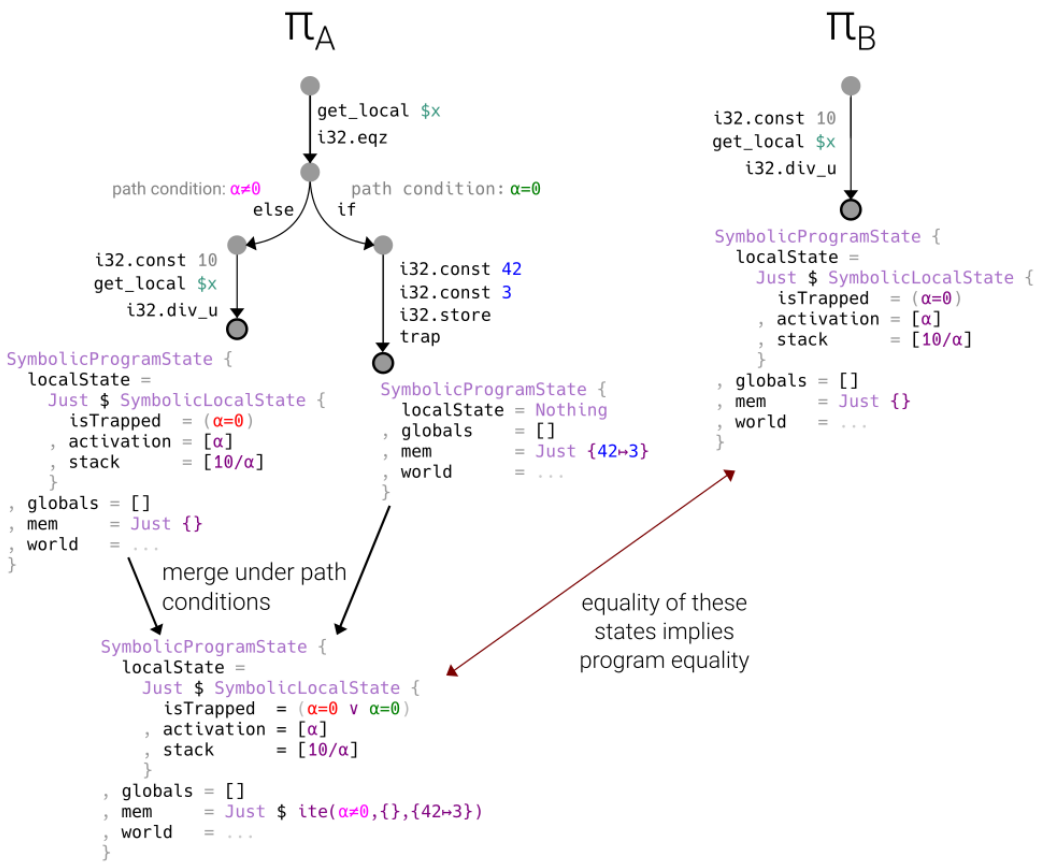


FIGURE 3.2: Symbolic Execution and Equivalence Checking example (Not strictly Haskell)

Symbolic Environments

The symbolic values within the symbolic state are defined with logical predicates, which constrain the possible assignments to the values; for instance, by stating '$a < b \wedge b + 1 < c$'. As every symbolic state is composed of such values, these predicates also constrain symbolic states. Consider a symbolic state with the following stack:

$$K\ \texttt{(i32.const}\ a\texttt{)}\ \texttt{(i32.const}\ b\texttt{)}\ \texttt{(i32.const}\ c\texttt{)}$$

The predicates ensure that the following interpretation is *not* valid (as $a < b$ does *not* hold):

$$K\ \texttt{(i32.const 100) (i32.const 75) (i32.const 50)}$$

As the symbolic execution concerns WebAssembly programs, the majority of the operators correspond directly to WebAssembly's. These predicates are included in the environment $E_{\mathcal{C}}$ over which the program state $\mathcal{Y}$ is defined. The input configuration $\mathcal{C}$ - which is associated with the fragment's initial program point - already constrains the values. Hence, these constraints are adopted by the symbolic state. The environment is defined as follows:

Definition 3.3 (Symbolic Variable Environment)**.** A symbolic variable environment E_C is defined as a tuple (V, R). V is the set of symbolic values available in the environment. R is the set of constraints placed upon these symbolic values; these constraints are usually relational between several values.

The possible constraints within R correspond roughly to the relational instructions in WebAssembly specification[48]. These operators are defined as follows:

- *Numeric Operators* - These operators define relations between numeric symbolic values. This includes equating a symbolic value to a constant value (which roughly corresponds to an equality with a 't`.const` c' instruction).
- *Select Operator* - This operator performs as an if-then-else construct. For example, $\texttt{select}(0, 42, 99) = 42$ and $\texttt{select}(1, 42, 99) = 99$.
- *Memory instructions* - As the memory block is also a symbolic value (of type byte-vector), it must be included in the environment, as memory changes during symbolic execution. An example constraint is: $M_{d1} = \texttt{store}(M_{d0}, i, v)$, which states the contents of memory block M_{d1} are identical to those of M_{d0}, except it contains value v at location i.
- *Extra Conversion Operators* - The WebAssembly specification only includes conversion between its primitive types. Additional conversion operators are defined for extra uninterpreted integer types (`i8` and `i33`). A example conversion constraint is $a = \texttt{i32.convert\_i8}(b)$.

An example environment is described below:

Example 3.2 (Environment)**.** Consider a WebAssembly function with 2 parameters p_0 and p_1 of type `i32`. Assume the initial configuration $\mathcal{C}$ - as obtained from the fragment's context - exclusively contains those states where $p_0 < p_1$.

An environment $E_C = (V, R)$ is derived from $\mathcal{C}$. A fresh symbolic value a is introduced, which is used as the symbolic value for parameter p_0. Similarly, fresh symbolic value b is assigned to parameter p_1. Another two fresh symbolic value m_d and m_s are introduced to symbolically represent program memory. w is a fresh symbolic value representing the world. Then $V = \{a, b, m_d, m_s, w\}$.

As $\mathcal{C}$ was constrained, its constraints are extracted to the environment; which is included in $R = \{\texttt{i32.le\_u}(a, b)\}$. `i32.le_u` defines the less-than operator for 32-bit unsigned integers.

The obtained symbolic state is given as $\mathcal{Y}_0 = (K, m_d, m_s, \emptyset, w)$. The stack K contains a single activation for the function, where the parameters are assigned symbolic values a and b.

Note that all satisfying interpretations to the symbolic values in a symbolic state $\mathcal{Y}$ over E_C exactly describe all states contained in $\mathcal{C}$.

3.2.2 Symbolic Execution

The extensional equivalence of two loop-free program fragments π_1 and π_2 over an initial configuration $\mathcal{C}$ is determined through *symbolic execution*[32]. Assume that π_1 and π_2 have no side-effects (side-effects are described in subsection 3.2.7). As both π_1 and π_2 are loop-free, both programs terminate. Then π_1 and π_2 are equivalent iff:

$$\forall s \in \mathcal{C}.(\mathbf{exec}^F(\pi_1,\ w,\ s) = \mathbf{exec}^F(\pi_2,\ w,\ s))$$

Clearly, it is computationally infeasible to iterate over all initial states $s \in \mathcal{C}$. Instead, π_1 and π_2 are *symbolically executed* starting at the same *symbolic state* $\mathcal{Y}$ in some environment $E_\mathcal{C}$. Both programs π_1 and π_2 are executed, and terminate in their own symbolic states $\mathcal{Y}_1$ and $\mathcal{Y}_2$, respectively. So:

$$(p_{f1}, w_f, \mathcal{Y}_1) = \mathbf{exec}^F(\pi_1,\ w,\ \mathcal{Y})$$
$$(p_{f2}, w_f, \mathcal{Y}_2) = \mathbf{exec}^F(\pi_2,\ w,\ \mathcal{Y})$$

When programs π_1 and π_2 are not *intensionally* equivalent, the symbolic representations of $\mathcal{Y}_1$ and $\mathcal{Y}_2$ are likely[4] different. However, π_1 and π_2 may be extensionally equivalent over $\mathcal{C}$, which is the case when their symbolic final states are:

$$\pi_1 \equiv_\mathcal{C} \pi_2 \quad \iff \quad E_\mathcal{C} \vdash \mathcal{Y}_1 = \mathcal{Y}_2$$

The equivalence of $\mathcal{Y}_1$ and $\mathcal{Y}_2$ is determined by the Z3 SMT solver. SMT solvers - like SAT solvers - can only find *satisfying* assignments. A *single* satisfying assignment for $\mathcal{Y}_1 = \mathcal{Y}_2$ demonstrates equality for a *single* concrete execution trace. Clearly, satisfiability is insufficient, as equality for *all* executions traces (starting at any $s \in \mathcal{C}$) is required. $\mathcal{Y}_1$ and $\mathcal{Y}_2$ are equal when their equality is *valid*. Luckily, this problem is representable as a satisfiability problem. $\mathcal{Y}_1$ and $\mathcal{Y}_2$ are equivalent if there exists *no* satisfying assignment for:

$$E_\mathcal{C} \vdash \mathcal{Y}_1 \neq \mathcal{Y}_2$$

The equality between two symbolic states relies upon the equality between their components; which are their stacks, memory blocks, and global variables. There are some considerations for determining this equivalence, which are described in subsection 3.2.6 and subsection 3.2.7.

3.2.3 Merging States

As execution paths may diverge under branch conditions, (when assuming determinism) all terminal states are reached under mutually-exclusive branch conditions. Once all paths are fully explored[5], the *pairs* of symbolic states are merged under the branch condition that caused them to diverge. In practice, states are merged bottom-up over the execution tree. Each path after a conditional branch - under some branch-condition φ - can be described by a symbolic state describing the input/output relation of that particular branch. Note that any conditional branch, with branch condition φ, spawns two new paths A and B. In path A (the if-case) φ holds, while in B (the else-case) $\neg\varphi$ holds. Let $\mathcal{Y}_A$ be the symbolic state describing the input/output relation of path A, and $\mathcal{Y}_B$ that same state for path B. Then, intuitively, their combined state can be given as:

$$E_\mathcal{C} \ \vdash \ \mathcal{Y} = (\mathcal{Y}_A \ \ \|_\varphi \ \ \mathcal{Y}_B)$$

[4]Potentially, syntactically distinct programs may produce identical symbolic states; for instance, when the application of *common sub-expression elimination* to π_1 produces π_2.

[5]Or a bound is reached, and symbolic execution aborts

Within an environment $E_{\mathcal{C}}$, this procedure traverses upward in the execution tree, until a single symbolic state is obtained. This "all-encompasing" symbolic state captures the full input/output relation of the program. The rule below states how two symbolic *values* are merged under a path condition.

$$\frac{E_{\mathcal{C}} \vdash \varphi : \text{SymbolicBool}, \quad a : \tau, \quad b : \tau, \quad c \text{ fresh in } E_{\mathcal{C}}, \quad c : \tau}{E_{\mathcal{C}}\{c \mapsto \text{select}(\varphi, a, b)\} \vdash a \ \|_{\varphi}\ b \ \Rightarrow c}\ \texttt{Merge}$$

Note that a and b are any two values (of the same type) contained in the symbolic environment. In practice, these may be (symbolic variations) of types `i32`, `i64`, `f32`, `f64`, memory block (uninterpreted function), or world. Then, two states of equal static structures are pairwise merged with the branch condition φ. This procedure merges elements (pairwise) over the full structure, being the local state, globals, memory, and the world. These are rather straightforward, except perhaps for merging the symbolic local state, as those are contained in a `Maybe`.

When both paths trap unconditionally, then their combination also traps unconditionally.

$$\frac{E_{\mathcal{C}} \vdash \varphi : \text{SymbolicBool}}{E_{\mathcal{C}} \vdash \quad \texttt{Nothing} \ \|_{\varphi}\ \texttt{Nothing} \ \Rightarrow \texttt{Nothing}}\ \texttt{MergeLocalNeither}$$

When only the left path (if-case) traps unconditionally, their combination traps when the path condition φ of the if-path holds. The trapping-behaviour of the right path (when b holds) is also preserved.

$$\frac{\begin{array}{c} E_{\mathcal{C}} \vdash \varphi, b : \text{SymbolicBool} \qquad c \text{ fresh in } E_{\mathcal{C}} \\ s\ \{\texttt{isTrapped} \mapsto b\} : \texttt{SymbolicLocalState env} \end{array}}{\begin{array}{l} E_{\mathcal{C}}\{c \mapsto (\varphi \vee b)\} \vdash \quad \texttt{Nothing} \ \|_{\varphi}\ \texttt{Just}\ (\ s\ \{\texttt{isTrapped} \mapsto b\}\ ) \\ \qquad\qquad \Rightarrow \texttt{Just}\ (\ s\ \{\texttt{isTrapped} \mapsto c\}\ ) \end{array}}\ \texttt{MergeLocalRight}$$

Similarly, when only the right path (else-case) traps unconditionally, the combined paths trap when the path condition $\neg\varphi$ of the else-path holds. The trapping-behaviour of the left path (when a holds) is also preserved.

$$\frac{\begin{array}{c} E_{\mathcal{C}} \vdash \varphi, a : \text{SymbolicBool} \qquad c \text{ fresh in } E_{\mathcal{C}} \\ s\ \{\texttt{isTrapped} \mapsto a\} : \texttt{SymbolicLocalState env} \end{array}}{\begin{array}{l} E_{\mathcal{C}}\{c \mapsto (a \vee \neg\varphi)\} \vdash \quad \texttt{Just}\ (\ s\ \{\texttt{isTrapped} \mapsto a\}\ ) \ \|_{\varphi}\ \texttt{Nothing} \\ \qquad\qquad \Rightarrow \texttt{Just}\ (\ s\ \{\texttt{isTrapped} \mapsto c\}\ ) \end{array}}\ \texttt{MergeLocalLeft}$$

When neither path traps unconditionally, their combination does not do so either. The trapping condition from both paths (a and b) are preserved. $(s_1 \ \|_{\varphi}\ s_2)$ refers (somewhat informally) to the pairwise application of the above `Merge` rule to the remaining elements in both s_1 and s_2.

$$\frac{\begin{array}{c} E_{\mathcal{C}} \vdash \varphi, a, b : \text{SymbolicBool} \qquad c \text{ fresh in } E_{\mathcal{C}} \\ s_1\ \{\texttt{isTrapped} \mapsto a\} : \texttt{SymbolicLocalState env} \\ s_2\ \{\texttt{isTrapped} \mapsto b\} : \texttt{SymbolicLocalState env} \end{array}}{\begin{array}{l} E_{\mathcal{C}}\{c \mapsto (a \vee b)\} \vdash \\ \quad \texttt{Just}\ (\ s_1\ \{\texttt{isTrapped} \mapsto a\}\ ) \ \|_{\varphi}\ \texttt{Just}\ (\ s_2\ \{\texttt{isTrapped} \mapsto b\}\ ) \\ \qquad\qquad \Rightarrow \texttt{Just}\ (\ (s_1 \ \|_{\varphi}\ s_2)\ \{\texttt{isTrapped} \mapsto c\}\ ) \end{array}}\ \texttt{MergeLocalBoth}$$

3.2.4 SMT solver

The superoptimizer uses the Z3[15] SMT solver to determine the equivalence of formulas obtained through symbolic execution. While a SAT solver can determine the satisfiability of a formula in *propositional logic*, an SMT solver extends this to (quantified) First-Order logic which may contain terms in a background theory (e.g. theory of integers).

Example 3.3 (SMT solver steps)**.** Consider the formula over integers x and y:

$$x > y \ \wedge \ (x < 10 \ \vee \ y > 42)$$

This formula is abstracted to an propositional logic formula:

$$A \ \wedge \ (B \ \vee \ C)$$

Note that A corresponds to $x > y$, B corresponds to $x < 10$, and C corresponds to $y > 42$. The SAT solver can, for example, find the satisfying interpretation $\{A \mapsto \top, \ B \mapsto \top, \ C \mapsto \top\}$. The *integer solver* is then given the set of terms $\{x > y, \ x < 10, \ y > 42\}$. Clearly, no satisfying assignment exists for x and y. Thus, the propositional formula is extended:

$$(A \ \wedge \ (B \ \vee \ C)) \wedge \neg(A \wedge B \wedge C)$$

Another satisfying SAT interpretation $\{A \mapsto \top, \ B \mapsto \top, \ C \mapsto \bot\}$ is found. The integer solver is given the corresponding set of terms $\{x > y, \ x < 10, \ \neg(y > 42)\}$; for which a satisfying model is found where $\{x \mapsto 5, \ y \mapsto 4\}$, which satisfies the original formula.

If no satisfying model exists, this is discovered once no satisfying formula exists for the corresponding propositional formula. Note that this example illustrates a high-level (and only *conceptually* accurate) depiction of a SMT solver. For the remainder of this thesis, the internals of the SMT solver are regarded as a black-box.

3.2.5 Empty Theory

SMT solvers often support a wide range of built-in theories; for example, Z3 supports linear integer arithmetic, bitvector arithmetic, and array operations, among others[15, 6]. However, many (uncommon or domain-specific) theories may still be missing. To still determine satisfiability of formulas relying upon such theories, the new theory can be described (either precisely or approximately) within the *empty theory*. This theory corresponds to the theory of *Equality logic and Uninterpreted Functions* (EUF)[35]. This theory is called *empty*, as its set of sentences is empty.

Example 3.4 (Model with uninterpreted functions)**.** Consider the following formula:

$$f(x) = y \ \wedge \ x < y \ \wedge \ f(y) = 6$$

Note that f is an *uninterpreted* function, which means it has no semantics at all. Note, though, that it is *pure*, which means that repeated application of this function to the same argument must produce the same result. One satisfying interpretation for this formula is:

$$\{f \mapsto \{3 \mapsto 4, \ 4 \mapsto 6\}, \ x \mapsto 3, \ y \mapsto 4\}$$

Note that the function f is now given a (partial) interpretation.

A satisfying model is usually found through *syntactic unification*. Terms within the language (of this logic) are partitioned into different *congruence classes* by a *congruence relation*[6, 17].

Definition 3.4 (Functional Congruence)**.** For every (uninterpreted) function f with arity $n > 0$:

$$\forall x_1, \ldots, x_n, y_1, \ldots, y_n. \left[\left(\bigwedge_{i=1}^{n} x_i = y_i \right) \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \right]$$

Congruence builds on the equalities that are already present within the universe (e.g., as obtained from other theories). Congruence is an equivalence relation, which means it is *reflexive*, *symmetric*, and *transitive*.

3.2.6 Equivalence Checking of Unknown Operators

WebAssembly programs operate on variables that are either uninterpreted integers (**i32** and **i64**) or IEEE 754-2019[3] Floating-Point numbers (**f32** and **f64**). The satisfiability (or unsatisfiability) of an expression consisting of values of these types can be determined. Z3 has good support for bitvector arithmetic. However, WebAssembly's floating-point qualification requirements (e.g., signaling NaNs)[48] are difficult to represent in Z3[49]. Also, if representing floats were easy, satisfiability checking is computationally expensive[9]. To work around this, symbolic execution of floating-point arithmetic is *defined stricter*[6]. That is, floating point operations are represented by another *sort* within Z3. This is described below.

The sort of these abstract 32-bit floating points is denoted $\mathcal{F}32$. For abstract 64-bit floating points this is $\mathcal{F}64$. These two sorts are *uninterpreted*, which means no concrete value assignment exists for values of this type. Z3 assigns abstract values to elements of these sorts. Conceptually, one may regard the set of abstract inhabitants of these sorts as isomorphic to $\mathbb{N}$. Within a program, floating points are represented as bitvectors, which are given specific sorts. **bvf32** denotes the sort of floating-point bitvectors of size 32. **bvf64** denotes the sort of floating-point bitvectors of size 64. The sorts of (integer) bitvectors of size 32 and 64 are denoted by $[\![32]\!]$ and $[\![64]\!]$, respectively[27].

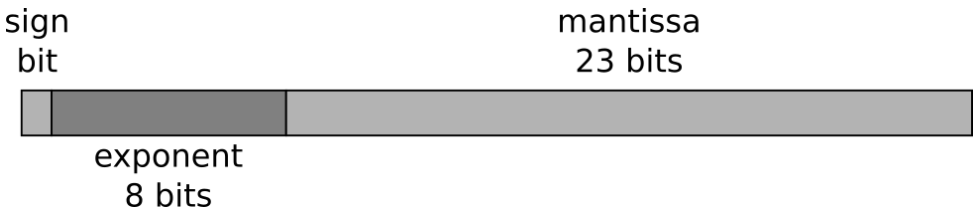


FIGURE 3.3: Typical interpretation of a **bvf32**[3]

The floating-point bitvectors are precisely that, bit vectors. Most inhabitants of **bvf32**s *could* be interpreted as real numbers[7]; Figure 3.3 depicts this correspondence. However, our verifier does *not* consider this interpretation, and treats values of sort **bvf32** and **bvf64** as typical bitvectors (which do not have the same operators as $[\![32]\!]$ and $[\![64]\!]$).

[6] It corresponds to an *Herbrand interpretation* for floating-point operations

[7] This is roughly what happens during concrete execution

While we introduce no formal SMT theory for floating-point bitvectors, we describe this new system through axioms over the empty theory. The set of functions is defined as $\Sigma^f_{\mathcal{F}} = F \cup P$. F is the set of functions defined as:

$$\begin{aligned} F = &\{n^{\mathcal{F}32} \mid n \in \mathbb{N}\} \\ &\cup \{n^{\mathcal{F}64} \mid n \in \mathbb{N}\} \\ &\cup \{0^{\mathbf{bvf32}}, \ldots, (2^{32}-1)^{\mathbf{bvf32}}\} \\ &\cup \{0^{\mathbf{bvf64}}, \ldots, (2^{64}-1)^{\mathbf{bvf64}}\} \\ &\ldots \end{aligned}$$

The remaining elements of F are functions which are evident from Table 3.1.
The set P denotes the predicates (which are functions that map to booleans):

$$P = \{\texttt{f32.relop}, \texttt{f64.relop}\}$$

$\tau_{\mathcal{F}}(f)$ denotes the sorts of symbols, as defined in Table 3.1.

Symbol	Sort $\tau_{\mathcal{F}}(f)$	Intended meaning
$n^{\mathcal{F}32}$	$\mathcal{F}32$	abstract 32-bit float
`F32.to`	$\mathbf{bvf32} \to \mathcal{F}32$	conversion
`F32.from`	$\mathcal{F}32 \to \mathbf{bvf32}$	conversion
`i32.reinterpret_f32`	$\mathbf{bvf32} \to [\![32]\!]$	reinterprets bitvector
`f32.reinterpret_i32`	$[\![32]\!] \to \mathbf{bvf32}$	reinterprets bitvector
`F32.unop`	$\mathbb{N} \to \mathcal{F}32 \to \mathcal{F}32$	unary f32 operator
`F32.binop`	$\mathbb{N} \to \mathcal{F}32 \to \mathcal{F}32 \to \mathcal{F}32$	binary f32 operator
`F32.relop`	$\mathbb{N} \to \mathcal{F}32 \to \mathcal{F}32 \to \mathit{Bool}$	f32 comparison
$n^{\mathcal{F}64}$	$\mathcal{F}64$	abstract 64-bit float
`F64.to`	$\mathbf{bvf64} \to \mathcal{F}64$	conversion
`F64.from`	$\mathcal{F}64 \to \mathbf{bvf64}$	conversion
`i64.reinterpret_f64`	$\mathbf{bvf64} \to [\![64]\!]$	reinterprets bitvector
`f64.reinterpret_i64`	$[\![64]\!] \to \mathbf{bvf64}$	reinterprets bitvector
`F64.unop`	$\mathbb{N} \to \mathcal{F}64 \to \mathcal{F}64$	unary f64 operator
`F64.binop`	$\mathbb{N} \to \mathcal{F}64 \to \mathcal{F}64 \to \mathcal{F}64$	binary f64 operator
`F64.relop`	$\mathbb{N} \to \mathcal{F}64 \to \mathcal{F}64 \to \mathit{Bool}$	f64 comparison
`i32.trunc_F32`	$\mathrm{Sx} \to \mathit{Bool} \to \mathcal{F}32 \to [\![32]\!]$	truncate to i32
`i32.trunc_F64`	$\mathrm{Sx} \to \mathit{Bool} \to \mathcal{F}64 \to [\![32]\!]$	truncate to i32
`i64.trunc_F32`	$\mathrm{Sx} \to \mathit{Bool} \to \mathcal{F}32 \to [\![64]\!]$	truncate to i64
`i64.trunc_F64`	$\mathrm{Sx} \to \mathit{Bool} \to \mathcal{F}64 \to [\![64]\!]$	truncate to i64
`promote`	$\mathcal{F}32 \to \mathcal{F}64$	promotes precision
`demote`	$\mathcal{F}64 \to \mathcal{F}32$	demotes precision

TABLE 3.1: Abstract floats and their operations

For notational simplicity, the operators are grouped by sort. For each group, a natural number corresponds to a particular member of the group (e.g., for unary operators: *add*, *sub*, etc.), which are ordered as in the WASM specification[48]. The 'Sx' type marks the signedness of the result[48]. The truncation functions are provided with a boolean which identifies whether it performs saturating conversion.

The intended meaning for these functions is stated, which relates them to their corresponding WASM instructions. These functions are, however, all *uninterpreted*. The SMT solver is responsible for assigning a valid (but arbitrary) satisfying interpretation. Intuitively, when determining equality over uninterpreted functions - or inhabitants of uninterpreted sorts - the values must be equal *under any interpretation*.

Two symbolic states $\mathcal{Y}_1$ and $\mathcal{Y}_2$ are equal if their inequality is unsatisfiable:

$$E_{\mathcal{C}} \not\models \mathcal{Y}_1 \neq \mathcal{Y}_2 \quad \Rightarrow \quad E_{\mathcal{C}} \models \mathcal{Y}_1 = \mathcal{Y}_2$$

The inequality is determined through the disjunction over the pairwise inequality between their components. Also, note that the other way around is *not* generally true. Subterms within the components (stack, memory, globals) of $\mathcal{Y}_1$ and $\mathcal{Y}_2$ may contain expressions of sort $\mathcal{F}32$. When two expressions a and b of sort $\mathcal{F}32$ are unequal, it is *sound* to assume `F32.from(a)` and `F32.from(b)` are also unequal (for our purposes). However, as **bvf32** has a bounded number of inhabitants (2^{32}), `F32.from(a)` and `F32.from(b)` could be equal. So, this assumption may cause some equal symbolic states to be marked as unequal. This is illustrated in Example 3.5. In practice this causes some optimizations to be missed, but no incorrect optimizations are ever accepted.

The SMT solver only marks an inequality as unsatisfiable if no interpretation *exists* for any of the used *functions* or symbolic values; this implies that programs which contain unknown functions are only equal if they are equal under *every interpretation* of those functions. As the universe of elements for $\mathcal{F}32$ and $\mathcal{F}64$ is infinite, a satisfying interpretation for any of these functions - which map to either of those domains - can often be found. One scenario where the inequality is unsatisfiable, is when the program π_2 is obtained by performing *common subexpressions elimination* upon π_1; as formulas obtained from their symbolic execution can be syntactically unified.

Example 3.5 (Missed equal float programs)**.** Consider the programs below.

LISTING 3.7: Program 1

```
x = 2.0 * 4.0
// { Post: x = F32.from(F32.mul(F32.to(2.0bvf32), F32.to(4.0bvf32))) }
```

LISTING 3.8: Program 2

```
x = 4.0 * 2.0
// { Post: x = F32.from(F32.mul(F32.to(4.0bvf32), F32.to(2.0bvf32))) }
```

The two expressions are *not* equal under functional congruence:

- `F32.from( F32.mul( F32.to( 2.0`$^{\texttt{bvf32}}$ `), F32.to( 4.0`$^{\texttt{bvf32}}$ `) ) )`
- `F32.from( F32.mul( F32.to( 4.0`$^{\texttt{bvf32}}$ `), F32.to( 2.0`$^{\texttt{bvf32}}$ `) ) )`

However, the floating point expressions *are* equal: $2.0 * 4.0 = 4.0 * 2.0$. Floating-point multiplication is notoriously *not* commutative, in general. It is *safe* to err on the side of caution, and assume multiplication is *never* commutative.

3.2.7 Equivalence Checking of Side Effects

Many practical WebAssembly programs produce side-effects; usually, these side-effects involve interaction with the user. While the superoptimizer assumes no knowledge about external functions, it is beneficial when it can reason about program fragments containing unknown functions; otherwise programs with side-effects must be avoided altogether.

As no knowledge about specific side-effects is assumed, verification must remain sound in a "worst case scenario"; which is defined as follows:

Definition 3.5 (Diverging Worlds Assumption)**.** Under this assumption, no two distinct side effects can *ever* produce the same world.

So, if one execution externally calls `bar(3)`, while another calls `bar(5)`, then the executions *will not ever* traverse a common world in the future.

An abstract depiction of example diverging world transitions by functions that produce side-effects is depicted in Figure 3.4.

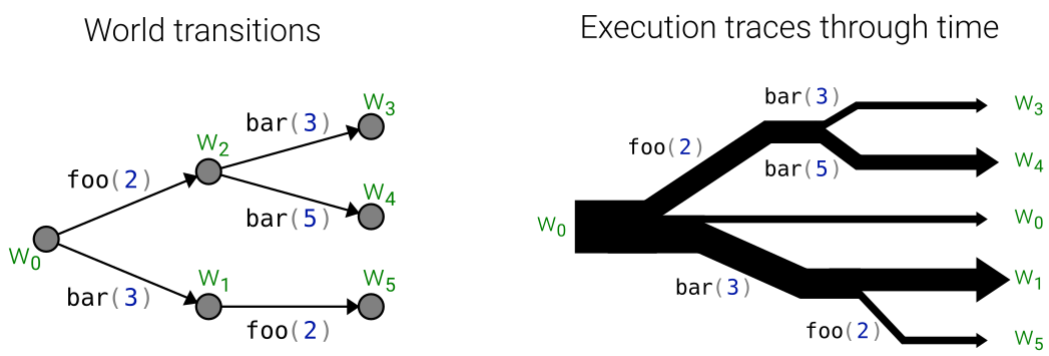


FIGURE 3.4: Example abstract depiction of diverging world transitions

Note that this is a very strict assumption; because in reality, `bar(3)` and `bar(5)` *may* actually have the same effect on the world. However, as the behavior of side-effects is unknown, assuming divergence is the only sound approach. Yet, sometimes equality of side effects *can* be proven (e.g., two executions *both* calling `foo(42)` in the same world), which enables some optimization opportunities. This is elaborated below.

Example 3.6 (Side-effects)**.** Consider the two programs below, where the functions `foo` and `bar` produce *unknown side effects*.

LISTING 3.9: Program 1

```
// { Pre: a = 0 ∨ a = 1 }
if a == 0 {
  c = foo(0, 5);
} else {
  c = foo(a, b);
}
d = bar();
```

LISTING 3.10: Program 2

```
// { Pre: a = 0 ∨ a = 1 }
b = if a == 0 { 5 } else { b };
c = foo(a,b)
d = bar();
```

Both programs produce *identical* side effects; this is regardless of the value of b. Namely:

- When $a = 0$, first `foo(0,5)` is called, followed by `bar()`.
- When $a = 1$, first `foo(1,b)` is called, followed by `bar()`.

The produced side-effects are also independent from the implementations of `foo` and `bar`.

WebAssembly modules import external functions, which are assumed to always produce side-effects. All imported functions - as well as their type signatures - are statically included in a WebAssembly module. This allows all these functions to be assigned a corresponding uninterpreted function in the logic. WebAssembly's syntax requires functions to have a type signature as follows:

$$vec(valtype) \to vec(valtype)$$

However, these type signatures do *not* consider the side effects of functions. Intuitively, their corresponding uninterpreted functions are given the following type signature (or rather, *sort* signature):

$$W \times (M_d \times [\![32]\!]) \times vec(valtype) \to W \times (M_d \times [\![32]\!]) \times vec(valtype)$$

Here, W is the set of *worlds*. The tuple $(M_d, [\![32]\!])$ describes the program *memory*; M_d is an array whose values are bitvectors of size 8, while the value of sort $[\![32]\!]$ is the memory's size. The signature includes memories, because external function have the opportunity to modify the memory. While the worlds abstractly represent the state of reality, these are modelled as an uninterpreted sort, whose opaque elements can be considered isomorphic to $\mathbb{N}$. Similarly to the equivalence checking over unknown operators, modifying the type signature as such is sufficient to soundly determine equivalence between two programs. After all, programs whose execution traces invoke different external functions produce distinct output worlds (by Definition 3.5). An example of proving inequality in the presence of side effects is given below.

Example 3.7 (Determining unequality)**.** Consider the following programs:

LISTING 3.11: Program π_1

```
x = foo(x + x);
```

LISTING 3.12: Program π_2

```
x = foo(3 * x);
```

The function `foo` is assumed to produce unknown side-effects. Under the sound assumption of diverging worlds, both programs surely end up in different worlds; which makes the program unequal. The operations that establish this are described below.

First, both programs are symbolically executed on the same symbolic state - which, for this example is $s_0 = \{\mathtt{x} \mapsto a\}$; `x` is the program variable `x`, and a is a symbolic value. Both executions start in the initial world w_0. Executing both programs gives the following final states:

$$\mathbf{exec}^F(\pi_1, w_0, s_0) = (p_{\text{final}},\ \text{foo}_w(w_0,\ a + a),\ \{\mathtt{x} \mapsto \text{foo}(w_0,\ a + a)\}))$$
$$\mathbf{exec}^F(\pi_2, w_0, s_0) = (p_{\text{final}},\ \text{foo}_w(w_0,\ 3 \cdot a),\ \{\mathtt{x} \mapsto \text{foo}(w_0,\ 3 \cdot a)\}))$$

The procedure `foo` produces two values: its output and the modified world. The function foo_w uniquely tracks its modifications to the world. So, the world produced by '$\text{foo}_w(w_0,\ a + a)$' is different from w_0.

The programs π_1 and π_2 are only equal iff all program variables map to the same values *and* both final worlds are equal, for any assignment of a and w_0. This corresponds to the following inequality being *unsatisfiable*:

$$\text{foo}(w_0,\ a + a) \neq \text{foo}(w_0,\ 3 \cdot a)\ \vee\ \text{foo}_w(w_0,\ a + a) \neq \text{foo}_w(w_0,\ 3 \cdot a)$$

However, one particular satisfying assignment is:

$$\{a \mapsto 1,\ \texttt{foo} \mapsto \{(w_0, 2) \mapsto 0,\ (w_0, 3) \mapsto 1\},\ \texttt{foo}_w \mapsto \{* \mapsto w_1\}\}$$

As a satisfying assignment exists, π_1 and π_2 are unequal.

While the previous example demonstrated the discovery of *inequality*, Example 3.8 demonstrates that equality may be determined in the presence of side-effects.

Example 3.8 (Proving equality)**.** Consider the following programs:

LISTING 3.13: Program π_1

```
x = foo(x + x);
```

LISTING 3.14: Program π_2

```
x = foo(2 * x);
```

These programs are surely extensionally equal, which is elaborated below.

Both programs are executed on the initial symbolic state $s_0 = \{\texttt{x} \mapsto a\}$; `x` is the program variable `x`, and a is a symbolic value. Both executions start in the initial world w_0. Executing both programs gives the following final states:

$$\textbf{exec}^F(\pi_1, w_0, s_0) = (p_{\text{final}},\ \text{foo}_w(w_0,\ a + a),\ \{\texttt{x} \mapsto \text{foo}(w_0,\ a + a)\})$$
$$\textbf{exec}^F(\pi_2, w_0, s_0) = (p_{\text{final}},\ \text{foo}_w(w_0,\ 2 \cdot a),\ \{\texttt{x} \mapsto \text{foo}(w_0,\ 2 \cdot a)\})$$

The programs π_1 and π_2 are only equals iff all program variables map to the same values *and* both final worlds are equal, for any assignment of a and w_0. This corresponds to the following inequality being *unsatisfiable*:

$$\text{foo}(w_0,\ a + a) \neq \text{foo}(w_0,\ 2 \cdot a)\ \vee\ \text{foo}_w(w_0,\ a + a) \neq \text{foo}_w(w_0,\ 3 \cdot a)$$

However, no interpretation of a, w_0, 'foo', and 'foo_w' exists such that this expression is true. Thus, it is unsatisfiable; which means programs π_1 and π_2 are equal.

The inverse of the diverging world assumption states: When the output worlds of two terminating programs are equal, both programs also encountered an equal sequence of worlds during their execution. This is a crucial observation, as it implies that program fragments can be freely replaced by other program fragments that equally affect the internal state and the world. When a fragment is replaced by another extensionally-equal fragment, in a non-terminating program, then the sequence of worlds observed by that non-terminating program remains unchanged.

Similarly to the programs over floating points, inequality between programs *cannot* be soundly determined; this is illustrated below. This inherently means the verifier errs on the side of caution.

Example 3.9 (Missed equal programs with side-effects)**.** Consider the two programs below. As a side-effect, the function `writeFile` writes a string to a file.

LISTING 3.15: Program 1

```
writeFile("hello.txt", "hello");
writeFile("world.txt", "world");
```

LISTING 3.16: Program 2

```
writeFile("world.txt", "world");
writeFile("hello.txt", "hello");
```

As both commands write to different files, these commands are *probably* commutative. This means the programs - in reality - produce identical side effects. However, the verifier assumes no knowledge of side-effect produced by functions. Thus it determines these programs as unequal.

Side-Effects of Memory

WebAssembly has one built-in instruction that is non-deterministic, namely `memory.grow`. This instruction either increases the size of the memory block by the requested amount (multiplied by the pagesize) and returns the previous size, or fails and returns -1. The success of this operations depends mainly on resources available to the host system. Luckily, W models a world that includes the host system. So the type signature of the instruction becomes:

$$\texttt{memory.grow} : W \times [\![32]\!] \to W \times [\![32]\!]$$

This function is given the current world and the *memory size*. Remarkably, this function does *not* require knowledge of the memory data (M_d). The memory data is merely an uninterpreted function which maps addresses to values. The bound on these addresses is externally enforced. As growing memory does not change the *contents* of memory, it is not included in the signature of `memory.grow`.

3.3 Program Search

Now the synthesis of program fragments remains. For this process, our superoptimizer uses a simple *enumerative* algorithm that is adapted to WebAssembly. This algorithm is *very* simple in comparison to synthesis algorithms used by existing superoptimizers (as discussed in section 6.1). However, the synthesizer applies several WebAssembly-specific pruning rules, which is sufficient to produce small instruction sequences. The synthesis algorithm is basically an instance of Dijkstra's algorithm, where edges correspond to instructions. First, the used cost function is described below.

3.3.1 Cost function

Instructions are assigned an execution cost. As the purpose of optimization is to reduce execution time, the cost of an instruction is often represented by its latency in clock cycles on a specific CPU. WebAssembly instruction do not have an associated latency, as WebAssembly is defined over an *abstraction* of modern hardware. However, some reasonable approximations can be made. The full table of instruction costs is not given here; instead, a few instruction costs are listed in Table 3.2. Most costs roughly correspond to the costs associated with recent Intel instruction sets[8], with some arbitrary multiplication factor.

Instruction	Cost
c.const	5
i*n*.add	5
i*n*.mul	15
i*n*.div	80
i*n*.shr	3
f*n*.abs	10
f*n*.add	20
f*n*.div	80
...	

TABLE 3.2: Sample of the cost function
($n \in \{32, 64\}$; $c \in \{\text{i32}, \text{i64}, \text{f32}, \text{f64}\}$)

While this list is not exhaustive, it displays relations between instruction costs that reflect those on modern CPUs. For instance, integer multiplication is more expensive than addition, which is more expensive than logical operators. By modelling the costs after common CPUs, it should reasonably reflect the execution times.

3.3.2 Dijkstra's algorithm

The synthesis is roughly an instance of Dijkstra's algorithm. Nodes correspond to configurations, while edges correspond to instructions. The algorithm expands nodes with the lowest cost first; expanding nodes continues until a state is reached whose static structure corresponds to that of the final state.

Consider the WebAssembly program in Listing 3.17, which multiplies its input by 4.

LISTING 3.17: Program 2

[8] https://www.agner.org/optimize/instruction_tables.pdf

```
(func (param $x i32) (result i32)
  (i32.mul
    (get_local $x)
    (i32.const 4)
  )
)
```

Figure 3.5 displays the (partial) search graph for the program. While traversing, instructions are (concretely) executed on the elements in the test set. All paths which reach the same state are extensionally equivalent under the current test set, causing their nodes to be merged. Merging their nodes ensures similar paths are not traversed separately, which reduces the search space.

The nodes in the search graph are traversed in increasing cost order (as per Dijkstra's algorithm). The first node matching the outputs of the test set corresponds to a potentially correct program. In the example, when $\$x = 2$, then $\$x \cdot 4 = 8$. Following symbolic execution, the SMT solver determines whether $\$x \cdot 4$ equals 8 for every value of $\$x$; which it does not. In particular, $\$x \cdot 4 \neq 8$ when $\$x = 5$, which is used as a new test case. All paths within the graph are (concretely) executed for this test case, causing some previous equivalence classes to be split. The search procedure resumes, until another candidate is found; namely, the program performing a left-shift by 2. Finally, the SMT solver determines $\$x \cdot 4$ equals '$\$x \ll 2$' for any value of $\$x$.

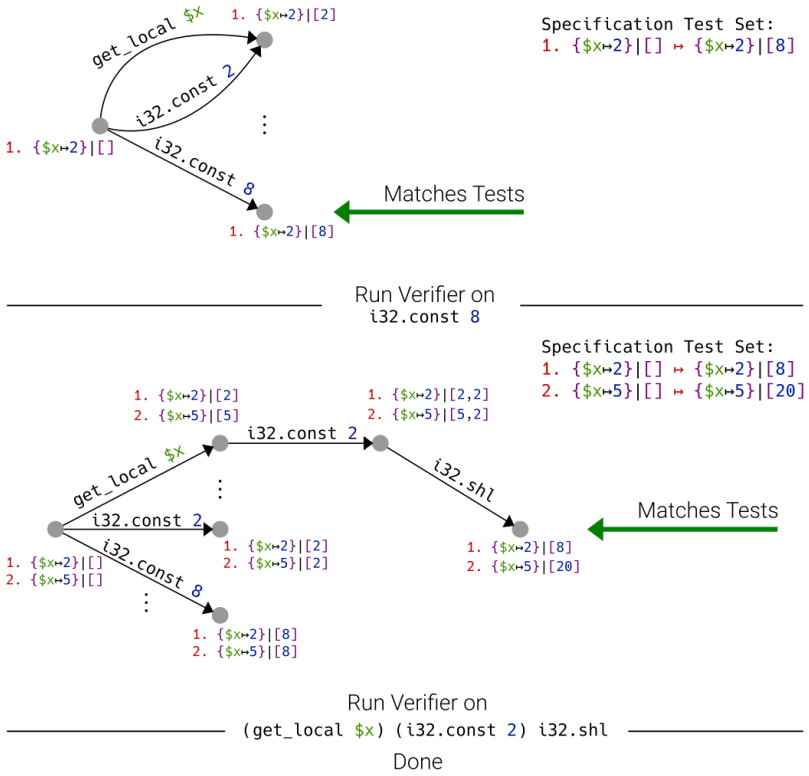


FIGURE 3.5: CEGIS Search Graph for $\$x * 4$ (partial & simplified)
(The tuple contains the stack and local variables; the stack grows to the right)

In practice, nodes also store global variables and memory. The advantage of this (CEGIS) search is that the SMT solver need only be invoked sparingly, as program

inequality can often be determined over a limited set of test cases. Unfortunately, this procedure cannot generate programs containing floating-point arithmetic or function calls. After all, floating-point values and side-effects have no representation in the test set; nor is concrete execution possible for side-effects. For those programs, the CEGIS aspect is omitted; which means the synthesis uses plain brute-force. Often, issues arise when the program state contains floating-point values - even when they reside lower on the stack and are not accessed by the fragment; the solver still attempts to find a satisfying assignment (representing a counter-example) for all symbolic values in the state. A possible solution is to omit parts of the state that are unused by the current fragment; our superoptimizer does not currently do this.

3.3.3 Pruning

The program search may be optimized by using information from the original program, which effectively prunes the search space. Some pruning strategies are:

- Use instructions similar to the original program. For example, if the input program only operates on `i32` values, generating programs which perform `i64` arithmetic may be omitted. The following properties are extracted from the input program:
  - Whether it uses any `i32` arithmetic operations
  - Whether it uses any `i64` arithmetic operations
  - Whether it uses any `f32` arithmetic operations
  - Whether it uses any `f64` arithmetic operations
  - The identifiers (= indices) of *read* global variables
  - The identifiers (= indices) of *modified* global variables
  - The identifiers (= indices) of *read* local variables
  - The identifiers (= indices) of *modified* local variables
  - The identifiers (= indices) of called functions

  The synthesizer only generates programs with instructions corresponding to those sets. Technically, this strategy may inhibit the discovery of an *optimum*. After all, non-obvious bit-twiddling hacks *may* emerge between - for example - `i32`-`i64` conversion. In practice, those are unlikely and do not (currently) warrant the increase in computation time.

- Limit generating (`i32` & `i64`) constants. As there are a total of 2^{32} unique inhabitants for the type `i32`, generating every single constant takes too long. In practice, the following sets - or combinations thereof - work well (for `i32`), depending on the program:
  - $\{2^i \mid i \in [0, 31]\}$ - All powers of 2.
  - $\{2^i - 1 \mid i \in [1, 32]\}$ - Any number of trailing one-bits.
  - Prime numbers

- Eliminate "stupid" programs; meaning any program containing fragments which are surely non-optimal. For example, any (pure) operation executed on constant inputs may be omitted, as it could be replaced by another constant. Surely, an infinite set of qualifying "stupid" fragments exists; we experimentally selected some simple patterns for our superoptimizer.

- When the original program has multiple (feasible) branches with unique side effects (e.g., one branch calls a function, while the other does not), then surely no linear program sequence exists that replaces it. (So synthesis aborts)
- When considering side-effects, the generated program must surely call the same functions as the original program, in the same order.

Further improvements to this synthesis can be achieved by considering the context of the fragment within the program; this is explained in chapter 4.

Chapter 4

Process Graphs

Chapter 3 established the synthesising and verification of loop-free programs. This chapter elaborates on their application to process graphs. Section 4.1 gives an overview of our approach.

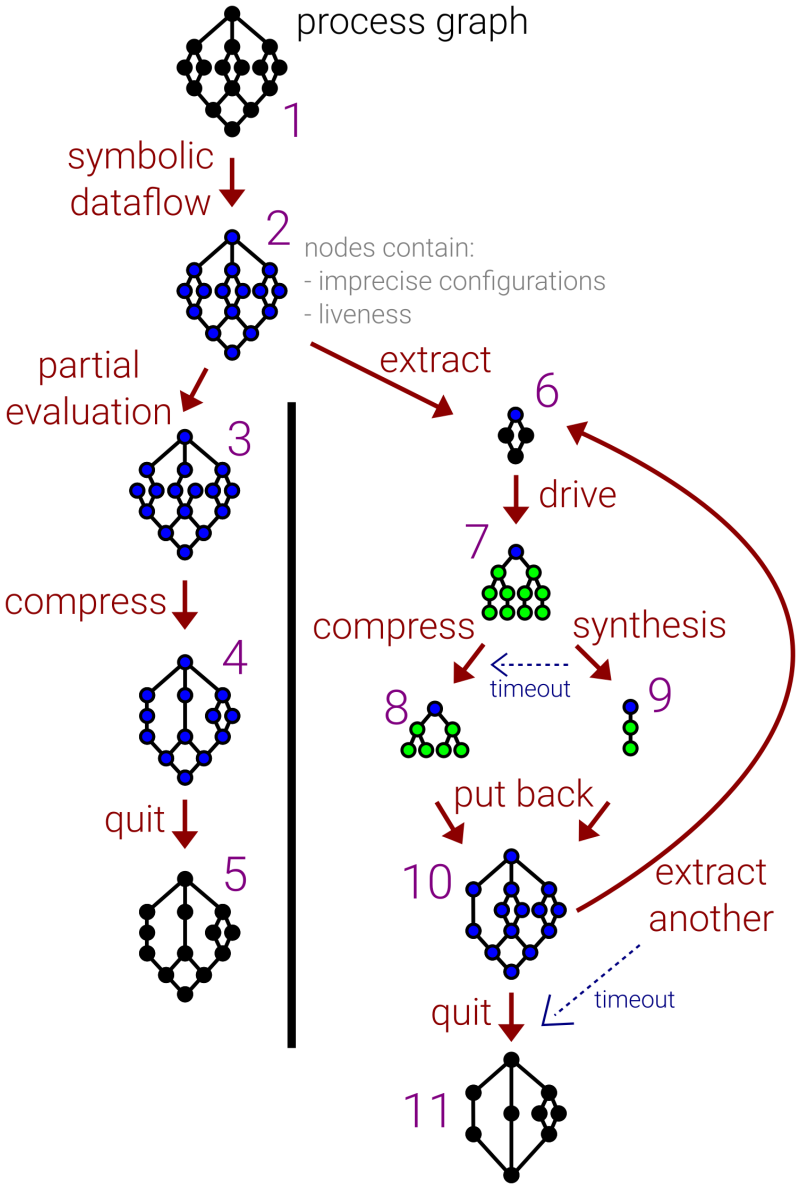


FIGURE 4.1: Superoptimization Procedure Overview

4.1 Superoptimizer Overview

Consider Figure 4.1; it depicts the general approach as applied by our superoptimizer. While the stages are numbered, it does *not* imply their order; those are merely for easy reference. The arrows convey the order between the stages, on which we elaborate below.

Stage 1. The procedure starts with a process graph of a single WebAssembly function. If a module contains multiple functions[1], we apply this procedure to every function. We discuss the representation of process graphs in section 4.4. Every node is assigned an (imprecise) configuration and variable liveness information; this is discussed in section 4.5.

Stage 2. At this point, the nodes contain an imprecise (but still useful) configuration. Ideally, optimization continues at stage 6, where a fragment is extracted, optimized, and placed back; in practice, that proves difficult. For most programs, it is easier (and computationally feasible) to continue to stage 3. While we do demonstrate the effectiveness from stage 6 onward, its application within larger graphs is subject to further research (as later discussed in section 8.1). Partial evaluation, which occurs when going to stage 3, is discussed in subsection 4.5.4.

Stage 3. After partial evaluation, some instructions are replaced by constants and branches eliminated. While non-optimal, this operation gives us insight in the expected *computation cost* of superoptimization on process graphs (as discussed later in subsection 5.4.1). From stage 3, the procedure continues to stage 4, where transition chains are compressed. Transition chain compression eliminates redundant instructions from the graph, which we discuss in subsection 4.6.4.

Stage 4/5. Finally, after eliminating redundant instructions, the (slightly) faster function is placed back into the module. Once all functions are optimized like this, the module is written back to a WebAssembly file. These programs are later evaluated in section 5.2.

Stage 6. This stage commences with some small WebAssembly graph fragment. In the subsequent stages, the fragment is optimized. We later evaluate the success of these stages in section 5.1. To go to stage 7, the fragment is *driven*. We give the theoretical basis for driving in section 4.2. In subsection 4.6.2 its application to WebAssembly is discussed. Note that driving subsumes partial evaluation (as discussed in subsection 4.5.4).

Stage 7. Given some imprecise initial configuration (as obtained by symbolic dataflow), the fragment is driven into a *finite* process tree. Within this tree, every node is associated with a *perfect*[2] configuration. From this tree, either *transition chain compression* or synthesis may be applied. Synthesis was previous discussed in chapter 3. We discuss transition chain compression in subsection 4.6.4.

[1] Modules often contain *hundreds* (if not *thousands*) of functions.

[2] These configuration contain as much information as possible, *given the imprecise initial configuration*. Surely, improving the initial configuration will further increase precision at nodes.

Stage 8/9/10/11. Finally, the optimized fragment is placed back into the graph. Only synthesised fragments are optimal; compressed driven fragments are merely faster (and often much larger). As we do not currently extract fragments from the graph (see stage 2), we do not actually put them back. The overview merely illustrates their envisioned place within the procedure. We perform stage 6, 7, 8, and 9 in isolation, and evaluate their effectiveness in section 5.1.

Final notes Note that *only* the synthesis step (between stage 7 and 9) currently uses *symbolic states*. Symbolic states *very precisely* capture an *input/output relation*, which is necessary when checking program equivalence (as discussed in chapter 3). Inside the graph, we maintain *configurations*, which are also symbolically represented; those are *much less precise*[3] than symbolic states. The representation of configurations is further discussed in section 4.3.

The stages above give a top-down overview of the graph optimization procedure. Section 4.2 elaborates on the background theory. From section 4.3 and onward, we discuss its application to WebAssembly.

4.2 Perfect Process Tree

Turchin introduced the notion of process trees[61] for his *supercompiler* (not to be confused with a super*optimizer*). Intuitively, a supercompiler *simulates* program execution with partial knowledge of the input, with the purpose of constructing a faster program; this is further discussed in section 6.3. Turchin theorised that a supercompiler relates to the way humans think. Humans observe phenomena, generalize observations, and construct mental models from these observations. For computer programs, this gives rise to the notion of *driving*:

Definition 4.1 (Driving)**.** Driving transforms a program π into another program π_d that is equivalent to π for any valid initial state[28]. Effectively, driving *simulates* program execution with knowledge of those initial states.

Through driving, the aim is to make observations about the program's execution. Mostly, these "observations" correspond to *partial evaluation* of program fragments. For example, an (sub-)expression may be replaced by a constant, or a conditional branch may be eliminated after its branch condition is evaluated. In the context of abstract programs (see Definition 2.7 on page 13), program transitions are specialized to their context. The transitions of the original program are defined as follows:

$$(p, w, s) \to (p', w', s') \to \cdots \to (p'', w'', s'')$$

In its driven form, every program point in the process tree is associated with a configuration for which it is specialized. Driving thus transforms the transition chains[4] into:

$$((p, C), w, s) \to ((p', C'), w', s') \to \cdots \to ((p'', C''), w'', s'')$$

A *perfect process tree*[19] is a process tree which contains exactly the set of execution traces that can occur during concrete execution. This means that any unfeasible

[3] A configuration may *over-approximate* the states reaching a program point.

[4] Effectively, a transition chain is a linear instruction sequence within the tree.

branches are entirely eliminated. *Perfect information propagation*[5] may be used to attain this result. The propagated must also be "perfect" to eliminate *all* branches. Effectively, every node in the tree must know precisely the set of states that can possibly reach it. In practice, though, finding perfect configurations is *very* difficult. Instead, *sound over-approximations* of configurations are often used.

When propagating insufficient information, driving cannot evaluate expressions that *could* be partially evaluated when sufficient information had been available. Previous research illustrated over-approximation by propagating only constants[19], or equality/inequality constraints[28]. Propagating more extensive (symbolic) information over control flow may enable better partial evaluation using an SMT solver. Generally, driving does not fundamentally change transition chains. As transition chains are effectively linear instruction sequences, a superoptimizer could aim to do so. Entire subtrees may be replaced, as long as the input/output relation remains unchanged. This means - when assuming termination - a set of execution traces (characterised by $\rightarrow$) at some program point $(p, \mathcal{C})$ may be safely replaced by another set of execution traces (characterised by $\rightarrow_2$); if for every state $s \in \mathcal{C}$:

$$((p, \mathcal{C}), w, s) \rightarrow^F ((p', \mathcal{C}'), w', s')$$
$$((p, \mathcal{C}), w, s) \rightarrow_2^F ((p', \mathcal{C}'), w', s')$$

Note that driving enforces that $\mathcal{C}$ is chosen such that $s \in \mathcal{C}$. At any time, it is safe to loosen the requirement on $\mathcal{C}$; instead choosing a $\mathcal{C}_G \supset \mathcal{C}$. After all, the invariant that $s \in \mathcal{C}_G$ at its program point p, is preserved. ($s \in \mathcal{C} \wedge \mathcal{C} \subset \mathcal{C}_G \Rightarrow s \in \mathcal{C}_G$).

4.2.1 Generalization

A common issue with driving is that it may run indefinitely, which produces an infinite tree. Some specialized program points may be *generalized*[61] again, which weakens the associated configuration $\mathcal{C}$ into another $\mathcal{C}_G$ where $\mathcal{C} \subset \mathcal{C}_G$. Clearly, doing so *arbitrarily* makes little sense. However, this is a good idea when two specialized program points $(p, \mathcal{C})$ and $(p, \mathcal{C}')$ are generalized to the same program point $(p, \mathcal{C}_G)$, where $\mathcal{C}, \mathcal{C}' \subseteq \mathcal{C}_G$. This operation converges two independent execution traces into a single new one. Additionally, program points may be combined with parents, which forms a cycle. If it is guaranteed that every infinite chain eventually generalizes with a parent node, a finite process graph is obtained.

Our configurations may contain "perfect" information, as a set of states can be precisely described through symbolic expressions; there is no "need" for over-approximating. However, at the same time, it implies that generalization is *next to impossible*. Generalization *could* be applied by picking $\mathcal{C}_G$ as:

$$\mathcal{C}_G = \mathcal{C} \cup \mathcal{C}'$$

However, when generalizing a program point $(p, \mathcal{C})$ with a parent program point[6] $(p, \mathcal{C}')$ - when assuming $\mathcal{C} \not\subseteq \mathcal{C}'$ - then $(p, \mathcal{C}')$ is replaced by $(p, \mathcal{C}_G)$, which *requires* that it is driven again. $(p, \mathcal{C}')$ is only correct for any $s \in \mathcal{C}'$. It is *not* generally correct for any $s \in (\mathcal{C}_G \setminus \mathcal{C}')$. Hence, it must be driving again from $(p, \mathcal{C}_G)$.

Surely, it is possible to drive $(p, \mathcal{C}_G)$ again. Though, at some point, one of its child nodes $(p, \mathcal{C}'')$ needs to be generalized with this parent yet again. While the $\cup$ and

[5] It turn out this means *perfect 'information propagation'*, and *not* 'perfect information' propagation

[6] Both must share the same p; as obtained from the original program. These represent different specialization of the same program point. Distinct program points cannot (in general) be combined to begin with.

$\subseteq$ can be implemented for symbolically represented configurations (as discussed in subsection 4.3.3 and subsection 4.3.2), this entire process gets *very* computationally expensive. While an argument may be made for its termination[7], it may take a number of iterations that is *exponential* in the size of the state[8].

4.3 Configuration Representation

We defined configurations in Definition 2.6 (on page 13) as sets of states with identical structures. While that description is accurate, it is *not* how we *represent* configurations. A configuration is usually a *humongous* set. For instance, a configuration with *five* unconstrained `i32` value on its stack has $2^{32 \cdot 5}$ members. Likely, no machines can keep these concretely in memory. Instead, configurations are - like symbolic states - symbolically represented. Conceptually, configurations differ from symbolic states (as indicated by Example 3.1 on page 18); a symbolic state is parametric over some input, which allows it to characterize an input/output relation. A configuration is a set of states.

A configuration has no knowledge of the *world* which holds at a program point, as the set of worlds is infinite (and opaque). Neither does a configuration have a notion of whether a "program point is trapped" - there is no such thing.

Example 4.1 (Configuration vs Symbolic State)**.** Consider a simple program computing $10/x$. During (concrete) execution, that program traps whenever $x = 0$. So, for equivalence checking, trapping behaviour must be preserved and is thus included a the symbolic *state* (as described in subsection 3.2.1). Though, as far as configurations are concerned, the final program point corresponds to the configuration $\{\{x \mapsto i\} \mid i \neq 0\}$; as that is the set of states reaching it.

As we represent configurations symbolically, operations on configurations require an SMT solver. These operations are performed similarly to those on symbolic states, albeit with different regards. For instance, a symbolic state may be unsatisfiable, indicating it *represents* no particular concrete state. When a configuration is unsatisfiable, the configuration is *empty*, indicating it *contains no* concrete states.

4.3.1 Configuration Implementation

Listing 4.1 shows our implementation of configurations; it resembles the implementation of symbolic states (as included in Listing 3.3 on page 19) - which is reasonable as both are modeled over WebAssembly states. However, it differs *crucially*.

[7] Through set union, the configuration can only grow. In the worst case, the "unconstrained" configuration is reached.

[8] Note that state *structures* - and thus their size - are statically known for any program point.

LISTING 4.1: Configuration Implementation (in Haskell)

```
data Configuration env =
  Configuration {
    activation  :: Activation env
  , stack       :: Stack env
  , globals     :: [Global env]
  , mem         :: Maybe (SymbolicMem env)
  , constraint  :: Symbolic env Bool
  , symbolics   :: Symbolics env
  }
```

A configuration contains its own symbolic environment (field `symbolics`)[9], as it makes no sense to define a configuration *over* some external environment; the configuration *represents* the environment[10] with its constraints. Configurations also do *not* include the world, nor contain a trapping condition. Additionally, configurations do *not* contain floating point values; meaning that values contained (deeper) inside this data structure are represented as shown in Listing 4.2.

LISTING 4.2: Configuration Value Implementation (in Haskell)

```
data Val env
  = VI32 (Symbolic env TI32)
  | VI64 (Symbolic env TI64)
  | VF32 ()
  | VF64 ()

-- The components in a Configuration are as follows (simplified)
data Activation env = Activation [Val env] [Val env] -- params, locals
type ScopeStack env = [Val env]
type Stack env      = NonEmpty (ScopeStack env)
data Global env     = Global Mut (Val env)
```

So, configurations do *not actually* represent sets of program states perfectly. Instead, the set of values assignable to a floating point variable is grossly over-approximated. Implicitly - as configurations represent *sets* - a floating-point values may be *any* value in its domain[11]. This over-approximation may cause some optimizations to be missed, as expressions containing floating-points can never be partially evaluated. As evaluation of floating-point expressions is difficult anyway (as discussed in subsection 3.2.6 on page 26), this is perfectly acceptable.

[9] `env` is *not* the environment; it is effectively a phantom parameter. Bottled all the way down, it occurs *nowhere* on the right-hand side. This trick assists in ensuring variables from different environments don't accidentally get mixed up.

[10] Remember, symbolic states are defined over an *external shared* environment, within which they are proven equivalent. (See subsection 3.2.2)

[11] While over-approximating is fine for configurations, it is *incorrect* for symbolic states, as discussed in subsection 3.2.6.

4.3.2 Configuration Subset

For generalization, it is necessary to decide whether one configuration is a subset of another. This problem can be translated into a logic formula using the definition of the subset relation; for any A,B:

$$A \subseteq B \quad \equiv \quad \forall x.(\, x \in A \rightarrow x \in B \,) \quad \equiv \quad \neg\exists x.(\, x \in A \wedge x \notin B \,)$$

The right-most formula is akin to any other satisfiability problem, where *unsatisfiability* of $x \in A \wedge x \notin B$ proves the subset relation; this is illustrated by Example 4.2.

Example 4.2 (Subset as Satisfiability Problem)**.** Consider the following two sets:

$$A = \{(x,y) \mid 2 \leq x \leq 7,\ 4 \leq y \leq 5\}$$
$$B = \{(x,y) \mid 0 \leq x \leq 10,\ 4 \leq y \leq 7\}$$

For these sets, $A \subseteq B$ holds, which means:

$$A \subseteq B \quad \equiv \quad \neg\exists x.(\, x \in A \wedge x \notin B \,)$$

The question of whether $A \subseteq B$ holds can thus be formulated as a satisfiability problem. Namely, $A \subseteq B$ holds iff the following formula is *unsatisfiable*:

$$(2 \leq x \leq 7 \ \wedge\ 4 \leq y \leq 5) \ \wedge\ \neg(0 \leq x \leq 10 \ \wedge\ 4 \leq y \leq 7)$$

In practice, such a formula is obtained by pairwise equating values between two configurations with equal static structures. While we found generalization with parent nodes to be unfeasible (see subsection 4.2.1), generalization with adjacent paths (i.e., multiple if-branches) is likely feasible. Additionally:

$$A \subseteq B \quad \Rightarrow \quad A \cup B = B$$

Whenever some configuration A is a subset of some configuration B, then their union need not be computed, as it is just B. While this seems useless in theory, our configurations often contain *thousands* of terms. Computing one less union operation may significantly reduce these representations; see also subsection 8.2.4.

4.3.3 Configuration Union

Finding the *union* over two symbolically represented sets gets a bit trickier. First, recall the definition of set union:

$$x \in (A \cup B) \quad \equiv \quad x \in A \vee x \in B$$

Example 4.3 (Union of Symbolic Representations)**.** Consider the following two sets:

$$A = \{2 \cdot x \mid x < 4\}$$
$$B = \{y \mid y > 1\}$$

The union of these sets is given as:

$$A \cup B = \{\ \text{select}(\ x < 4,\ 2 \cdot x,\ y\ ) \mid x < 4 \ \vee\ y > 1\}$$

Here, 'select' refers to the if-then-else operations in the logic (`ite` in Z3). Note that, *intuitively*:

$$v \in \{2 \cdot x \mid x < 4\} \Rightarrow v \in \{\ \text{select}(\ x < 4,\ 2 \cdot x,\ ?_1\ ) \mid x < 4 \vee ?_2\}$$
$$v \in \{y \mid y > 1\} \Rightarrow v \in \{\ \text{select}(\ ?_3,\ ?_4,\ y\ ) \mid ?_3\ \vee\ y > 1\}$$

Here, $?_1$ and $?_2$ are unknown expressions that do not reference x, and $?_3$ and $?_4$ are unknown values that do not reference y.

That example should illustrate how the union over configurations may be implemented. Given two configurations $\mathcal{C}_1$ and $\mathcal{C}_2$ with equal static structure, their disjoint symbolic environments (field `symbolics`) are combined. The *disjunction* over their constraint (field `constraint`) is taken as the new constraint. Finally, the 'select' operator - with the constraint of $\mathcal{C}_1$ as conditional - is pairwise applied to all fields inside the configurations.

4.4 Graph Representation

In the sections below, the data structures we use to represent process graphs are discussed.

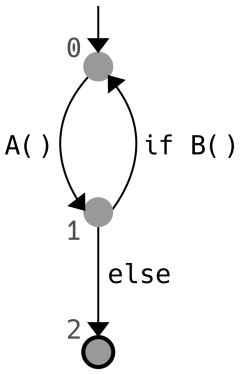
4.4.1 The Bad Graph

The theory demands[28] that transitions in process graphs represent "clean" transitions between configurations. For example, unconditional break (`br`) instructions are implicit within such graphs, as jumps are no different from regular instruction traversal; that is, considering a process graph contains execution traces. While it is surely possible to define such a graph for WebAssembly, after superoptimization, a program must be re-obtained from that graph. As the WebAssembly syntax demands structured control flow, it is surely non-trivial to find such a program. In theory, any computable function can be represented using structured control flow.

Definition 4.2 (Structured Program Theorem)**.** Any computable function can be represented with a control flow graph consisting of only *sequence*, *selection* (if-else), and *iteration* (loops)[7].

However, this definition makes no claims on whether program reconstruction is *easy*; in practice, program reconstruction may require the introduction of arbitrary control flow structures. Consider Figure 4.2 and its corresponding reconstructed program in Listing 4.3[12].

FIGURE 4.2: Some process graph (pseudo-language)



LISTING 4.3: Loop and switch (Rust)

```
let mut label = 0;
loop {
  match label {
    0 => { A( ); label = 1; },
    1 => if B( ) { label = 0; }
         else { label = 2; },
    2 => break,
    _ => unreachable!( )
  }
}
```

The reconstructed program contains the notorious *loop-and-switch* anti-pattern. A seemingly simple control-flow structure reconstructs into a complicated structured program, where a `label` variable tracks which branch to take next. Surely, this kind of overhead is *very* costly during execution; if arbitrary overhead is introduced during program reconstruction, many performance improvements from superoptimization may be lost altogether. Note, however, that often a good program with few loop-and-switch statements *may* still be found; for instance, when using the prevalent Relooper[65] algorithm. Relooper is a great algorithm that can transform many arbitrary control flow graphs into rather good structured programs. Occasionally, however, the loop-and-switch structure remains necessary. Within our superoptimizer, we avoid this issue altogether by preserving the control structures from the original program.

[12] Example adapted from [25]

4.4.2 The Good Graph

The data type for the process graphs is given in Listing 4.4; effectively, this data type is *isomorphic* to the WebAssembly AST[48]. While it does *not* strictly represent a process graph, the main benefits of a process graph are preserved; namely, that every program point may be associated with a configuration. Additionally, *process trees* may be defined over a subset of this representation, which preserves the ability to drive (as later discussed in subsection 4.6.1). A reasonable case can be made for claiming that this structure *is* a process graph as defined over structured control flow. After all, a "true" process graph from which no program can be reconstructed serves little purpose anyway.

LISTING 4.4: Graph Data Structures

```
data Node
  = Node Edge     -- ^ graph inner node
  | NodeEnd       -- ^ terminal node in a scope
  | NodeReturn    -- ^ explicit function return
  | NodeTrapped   -- ^ unconditionally trapped

data Edge
  = EdgeInstr SimpleInstr NodeId
  | EdgeIf FuncType SubGraph SubGraph NodeId
  | EdgeBlock FuncType SubGraph NodeId
  | EdgeLoop FuncType SubGraph NodeId
  | EdgeBr LabelIdx
  | EdgeBrIf LabelIdx NodeId
  | EdgeBrTable [LabelIdx] LabelIdx
  | EdgeCall FuncIdx NodeId
  | EdgeCallIndirect FuncType NodeId

type NodeId = Int

data Graph =
  Graph {
    rootI      :: NodeId
  , nodes      :: IntMap Node
  , terminalI  :: NodeId
  }
```

A graph is represented as an `IntMap`. This representation allows every node - which corresponds to a program point - to be individually addressed. Most operations and modifications are defined over this graph structure. Note that all edges correspond exactly to instructions in the WebAssembly AST[48]. For some entities, it is debatable whether they should be an *edge* or a *node*. For instance, `EdgeBr` has no (explicit) target, but returns control to a parent scope. As a rule of thumb: If an entity needs to be individually addressable as a program point, it is a node. The node type is reused by the tree representation, which also has some bearing on the

choice of representation (not the edges, though, as discussed in subsection 4.6.1)[13]. Additionally, some "edges" contain entire sub-graphs; Surely that is *strange*, but it is necessary to guarantee program reconstruction. Some constructors are briefly discussed below.

- `NodeEnd` - The final node in any scope; execution resumes in the surrounding scope. If it occurs in the outer-most scope, it represents the graph's terminal node.
- `NodeReturn` - Corresponds to the `return` instruction. It explicitly returns from the function; effectively, this causes execution to "resume" at the graph's terminal node, where execution halts.
- `EdgeBr`, `EdgeBrIf`, `EdgeBrTable` - These corresponds to the `br`,`br_if`,`br_table` instructions, which break to a surrounding scope. (See also subsection 2.2.2)
- `EdgeCallIndirect` - Corresponds to the `call_indirect` instruction, which calls a function in the function table. Intuitively, it calls a function referenced by a pointer.

[13]The presented structures vary slightly from the actual implementation; particularly in their type-parameters and strictness annotations. `Node` and `Tree` are identical in the implementation, except for their (parametric) edges; this is irrelevant for discussion here.

4.5 Dataflow Analysis

Dataflow analysis[31, 43] is a method for finding information about variables at program points. This information propagates over flow edges, which represent *transitions*. These transitions correspond roughly to those on process graphs. Dataflow analysis aims to find consistent information over larger control flow structures; this varies somewhat from the established configurations. For instance, *constant propagation* propagates known constants through variables at program points in the graph. *Zero analysis* keeps track of which variables are surely *zero*, *non-zero*, or neither. Within our context, those analysis values may be considered *abstractions* or *over-approximations* of perfect configurations at program points. Though, these analyses are commonplace because of their (relatively) low computation cost.

4.5.1 Lattices

The information stored at a program point by a dataflow analysis corresponds (often) to an element in some lattice. A lattice is a *partially ordered set*, where every two elements have a *join* $\vee$ (least upper bound) and a *meet* $\wedge$ (greatest lower bound). Similarly to how configurations may be generalized, the *join* over information obtained from multiple incoming edges can be taken, which causes the value to "ascend" in the traversed lattice. Every *finite*[14] lattice has a $\top$ value.

Definition 4.3 (Lattice Top)**.** The *top* $\top$ value in a *finite* lattice L satisfies for any value $a \in L$:

$$\top \quad = \quad a \vee \top \quad = \quad \top \vee a$$

Dataflow analysis aims to find a *fixpoint*[43].

Definition 4.4 (Fixpoint)**.** In general, a fixpoint of some function f is obtained at some value x when $f(x) = x$. For any dataflow analysis (on process graphs), a fixpoint is reached whenever for all nodes A with flow edge E to some node B satisfy:

$$\text{transfer}(v_A, E) \vee v_B \quad = \quad v_B$$

Here, v_A and v_B are the values stored at nodes A and B, respectively.

Whenever a fixpoint is reached, the system is *stable*; this means the information stored at each node is *consistent* with information observed during any concrete execution. In practice, a *least fixpoint* has the most value. After all, storing $\top$ at every node constitutes as a fixpoint, but conveys no information. Instead, values residing lower in the lattice correspond[15] to smaller configurations, which are surely more descriptive than $\top$. The application of dataflow to our process graphs is discussed below.

[14] Dataflow analysis is possible for *some* infinite lattices. In this work, only finite lattices are of interest.

[15] Formally, the connection between configurations and lattice elements is established through *abstract interpretation*; for instance, as discussed by Jones[28]. We don't elaborate on this relation further.

4.5.2 Flow Edges

Listing 4.5 displays the flow edges as implemented in our superoptimizer.

LISTING 4.5: Flow Data Structures (Haskell)

```haskell
type Flow = NodeId -> [(FlowEdge, NodeId)]

data FlowEdge
  = FlowInstr SimpleInstr
  | FlowCall FuncIdx
  | FlowCallIndirect FuncType
  | FlowIfTrue KeptVals    -- ^ enters an if-block
  | FlowIfFalse KeptVals   -- ^ enters an else-block
  | FlowEnter KeptVals     -- ^ enters a block/loop
  | FlowBr KeptVals (NonEmpty DroppedVals) IsLoopLabel
  | FlowBrIf KeptVals (NonEmpty DroppedVals) IsLoopLabel
  | FlowBrElse
  | FlowBrTable JmpTableIdx KeptVals (NonEmpty DroppedVals) IsLoopLabel
  | FlowMagic [ValType] (NonEmpty DroppedVals)

data JmpTableIdx
  = JmpTableIdx Word32
  | JmpTableGeq Word32

type DroppedVals = [ValType]
type KeptVals    = [ValType]
type IsLoopLabel = Bool
```

Most of these constructors correspond to those of the process graph (as discussed in subsection 4.4.2). Though, these edges are represented closer to the transitions as (implicitly) evident in the WebAssembly specification. While the process graph ensures by construction that every transition is deterministic - for instance, `EdgeIf` contains *both* mutually-exclusive branches - this information is implicit in the flow edges. Every branch represents an individual transition. Some notes on these constructors are given below:

- `FlowEnter` - This edge is encountered upon block scope entry (e.g., `block`, `loop`). It pushes a new label to the stack, and appropriately transfers its parameters from the parent scope to the new scope. (As illustrated in subsection 2.2.2)
- `FlowBrIf`, `FlowBrTable` - These edges represent a conditional jump to an outer scope. Upon leaving a scope, some values from the local scope are transferred to the outer scope, while explicitly dropping other scope stacks (As illustrated in subsection 2.2.2). Note particularly for `FlowBrTable`: The `br_table` instruction may contain an arbitrary number of jump targets, where a popped index determines the target at runtime. Each of these targets is individually represented as a flow edge, which contains its specific branch condition (i.e., `JmpTableIdx`).

- `FlowMagic` - The transfer function for this edge introduces values of the given types, while dropping some scope stacks; this ensures type-correct stacks are produced during backward analysis over `unreachable` instructions.
- `IsLoopLabel` - While not a constructor, its presence relates to a subtle property in WebAssembly. When jumping to some label n, out of a `block` or `if` statement, then n labels are popped and execution resumes $n + 1$ scopes upward. However, when that label corresponds to a `loop`, then n labels are popped, the loop's label is *pushed again* and execution resumes within the loop body. `IsLoopLabel` distinguishes those cases.
- `KeptVals` / `DroppedVals` - Flow edges that relate to control instructions (e.g., `br`) may discard entire scope stacks upon traversing to a parent scope. During *backward analysis*, these scope stacks must be reconstructed, and are thus explicitly included in the flow edges.

We implement dataflow analysis parametrically over the WebAssembly semantics, which ensures some common behaviour is shared. This shared behavior mainly consists of pushing and popping type-correct values from the stack for each instruction. The superoptimizer instantiates (parts[16] of) this framework for different analyses and executions, such as liveness analysis, symbolic information propagation, concrete execution, symbolic execution, and type-checking. To increase confidence toward correctness of this framework, we implemented both forward and backward type-checking over the graph flow. As WebAssembly validation ensures the structure of states at program points is statically known, type-checking aims to find consistent structures[17] for every program point. The instantiation of this framework for *symbolic information propagation* and *liveness analysis* is discussed in the sections below.

4.5.3 Symbolic Information Propagation

In this subsection, we discuss the propagation of symbolic information (configurations) over non-looping control flow.

Transfer function

Configurations represent sets of states. The WebAssembly specification[48] elaborates on the concrete semantics, which represent the transitions corresponding to instructions. These semantics are not reiterated here. Intuitively, consider the transfer function - where E is the edge - as:

$$\text{transfer}(E, \mathcal{C}) = \{\text{step}(E, s) \mid s \in \mathcal{C},\ \text{step}(E, s) \text{ not trapped}\}$$

Here, the 'step' function references the concrete behaviour of the edge; for instructions these correspond directly to the small-step operational semantics[48]. Flow edges that do not directly correspond to instructions can be trivially inferred from it (and some are discussed in subsection 4.5.2). One thing that is important to note, though, is that on function calls *no assumptions* are made. A function call may produce any output values in the domain of its output type (e.g., any $i \in [0, 2^{32})$ for

[16] Symbolic execution and concrete execution are *not* implemented over flow; they do share a lot of the parametric type-correct pushing/popping implementation, though.

[17] Type-checking determines types for stack values, local variables, and global variables at every program point. We implemented this over both forward and backward flow.

`i32`). Additionally, the analysis assumes that function calls may modify memory and modify mutable global variables. As the implementation of functions is unknown, their behaviour must be over-approximated; meaning this is the only sound approach.

Strategy

Propagating configurations over control flow entails some subtleties in the implementation. After all, the analysis should *terminate*, be *consistent*, and preferably be *fast*. A naive iterative dataflow solver will likely not suffice, as it surely takes too long; for similar reasons as for generalization (as discussed in subsection 4.2.1). Instead, we use a *single pass* solver to reach a fixpoint. The general approach is as follows:

1. The graph root is initialised to the *initial configuration*. Function parameters, memory, and global variables may have any value; as these are unknown. Non-parameter local variables are initialized to their respective zero values.
2. Loop entries are initialised to their respective $\top$ values. Effectively, these configurations are entirely unconstrained; no assumptions are made about its contained values.
3. Every node is assigned a configuration by traversing the forward edges in *reverse postorder*. Configurations obtained from multiple incoming edges are combined through set-union ($\cup$ - see subsection 4.3.3).

This approach reaches a fixpoint. While this fixpoint is not generally a *least fixpoint*, it is *consistent*; meaning all states observed at runtime at some program point p are contained in the configuration $\mathcal{C}$ stored at the program point. Though, these configurations may grossly over-approximate the actual set of observed states. The approach is further elaborated below.

Note that our process graphs corresponds to a *reducible graph*, which means every edge is unambiguously a *backward* or *forward* edge.

Definition 4.5 (Reducible Graph)**.** The edges of a reducible graph can be partitioned in two sets[24]:

- Forward edges (forming a Directed Acyclic Graph)
- Backward edges (a, b), where b dominates a

When a graph is *irreducible*, a loop body may be entered through multiple edges. In that case, it is ambiguous for some edges whether they are forward or backward edges. Our process graph is a *reducible* graph, as it consists of structured control flow (see also Definition 4.2 on page 45). The DAG of forward edges may thus be traversed in *reverse postorder*.

Definition 4.6 (Reverse postorder)**.** Reverse postorder is a linear order on nodes in a DAG where every node is visited before its children.

As nodes are traversed in reverse postorder, every node and edge is traversed only once. Thus, the system terminates in $\mathcal{O}(n + e)$; where n is the number of nodes and e is the number of edges. As it may be non-obvious that the obtained solution is a fixpoint, that is explained in Proof 4.1.

Proof 4.1 (Fixpoint)**.** Assume the system has *not* reached a fixpoint. That means there exists some flow edge E from some node A with configuration $\mathcal{C}_A$ to a node B with configuration $\mathcal{C}_B$ such that:

$$\mathit{transfer}(\mathcal{C}_A, E) \cup \mathcal{C}_B \quad \neq \quad \mathcal{C}_B$$

Node B is either a loop entry, or it is not. These cases are handled separately:

- Node B is a loop entry. Loop entries are assigned configuration $\top$, so $\mathcal{C}_B = \top$. When taking the definition of $\top$ into account:

  $$\mathit{transfer}(\mathcal{C}_A, E) \cup \mathcal{C}_B \quad = \quad \mathit{transfer}(\mathcal{C}_A, E) \cup \top \quad = \quad \top \quad = \quad \mathcal{C}_B$$

  Hence, B cannot be a loop entry.

- Node B is *not* a loop entry. Then, as the graph (within which nodes A and B are defined) is *reducible*, edge E must be a *forward* edge. Hence, node A occurs *before* node B in the reverse postorder (of the forward edges). $\mathcal{C}_A$ at A is obtained before its outgoing edge E is traversed. Node B is visited after *all* its parents (in the forward DAG) are visited, which means:

  $$\mathcal{C}_B \quad = \quad \bigcup\{\, \mathit{transfer}(\mathcal{C}_X, E_X) \mid (\mathcal{C}_X, E_X) \in \mathit{in}(B)\}$$

  Here, $\mathit{in}(B)$ represents the set of B's incoming edges (in the forward DAG) with the configurations stored at those parent nodes. As A is a parent of B, $(\mathcal{C}_A, E) \in \mathit{in}(B)$. Then:

  $$\begin{aligned} & (\mathcal{C}_A, E) \in \mathit{in}(B) \\ \Rightarrow \quad & \mathit{transfer}(\mathcal{C}_A, E) \subseteq \bigcup\{\, \mathit{transfer}(\mathcal{C}_X, E_X) \mid (\mathcal{C}_X, E_X) \in \mathit{in}(B)\} \\ \Rightarrow \quad & \mathit{transfer}(\mathcal{C}_A, E) \subseteq \mathcal{C}_B \\ \Rightarrow \quad & \mathit{transfer}(\mathcal{C}_A, E) \cup \mathcal{C}_B = \mathcal{C}_B \end{aligned}$$

  Hence, B cannot be a regular node either.

As B can be neither of these node types, B cannot exist. Hence, the system has reached a fixpoint.

Speedup While this algorithm already throws away lots of information (by setting loop entries to $\top$), computing a single forward pass is often *still* too expensive. A lot of the computation cost is incurred by taking the union over configurations on converging flow, as terms from multiple incoming edges are *combined* into a new configuration (See subsection 4.3.3). For larger programs, a single configuration may contain environments with *multiple thousands* of symbolic terms. Storing that many terms *for every node* causes significant memory use[18]; additionally, passing these large environments to Z3 introduces quite some overhead. To eliminate this overhead, a configuration may be replaced by $\top$ if its term count exceeds some parametric value (e.g., $3,000$); this is not a problem, as it over-approximates the configuration. Realistically, some information is thrown away and some optimization opportunity is lost.

[18]While value/memory sharing is *sometimes* possible, every environment contains its own set of terms and (unknown) values, which (currently) requires terms to be copied between environments.

4.5.4 Partial Evaluation

We propagate configurations to nodes in the process graph with the hopes that their information enables optimization opportunities. A simple optimization is to perform *partial evaluation* on terms in the program. Consider Listing 4.6.

LISTING 4.6: Comparison transitivity (Rust)

```
1 if a > b {
2   if b > c {
3     if a <= c { // Never true
4       ...
5     }
6   }
7 }
```

There *exists no* concrete execution where the boolean conditional on line 3 ($a \leq c$) is true. Hence, the expression may be safely replaced by *false*, which enables elimination of that entire `if`-block. For every (integer) expression, the superoptimizer asks Z3 whether a constant replacement exists. This constant replacement for some expression is found as follows:

Algorithm 4.1 (Partial Evaluation)**.** Given some expression that is represented by a symbolic value q in some configuration $\mathcal{C}$, partial evaluation aims to find a constant replacement. The configuration $\mathcal{C}$ is passed to Z3, and Z3 finds a satisfying model for q.

- If no model exists, the configuration is empty; this means the branch is unreachable.
- If a model with some constant c_1 exists, a *possible* replacement exists. Now, the configuration is passed to Z3 again, with the additional constraint $q \neq c_1$.
  - If *another* satisfying constant c_2 exists for q, then *no* unique constant replacement exists for q. (This must also be assumed when Z3 times out)
  - If *no* other satisfying constant exists for q, then q may be safely replaced by c_1.

It is crucial that the replacement constant is *unique*. Whenever multiple satisfying models are found, no unique constant exists. Example 4.4 illustrates this issue.

Example 4.4 (Constant Replacement)**.** Consider Listing 4.7.

LISTING 4.7: Invalid Constant Replacement (Rust)

```
// { pre: z > 3 }
y = 4 * z
```

Whenever Z3 is requested to provide a satisfying model for `y`, Z3 may provide $\{\texttt{y} \mapsto 16\}$. After all, $\texttt{z} = 4$ satisfies the precondition, and then `y` evaluates to 16. However, when Z3 is asked for a *different* model, $\{\texttt{y} \mapsto 20\}$ may be given. That model *also* satisfies the precondition ($\texttt{z} = 5$), and is thus a valid assignment for `y`.

The crucial observation is that SMT solvers find *satisfying* models; of which there may be many. Only when the model value is *unique* may that value be used as a valid replacement.

At any node, when the stack's top value can be provably replaced by a constant, our superoptimizer introduces a `in.const` instruction, preceded by a `drop` instruction. The `drop` instruction drops the *computed* value, while the constant instruction pushes the value it always produces (without computing it). Later, during *transition chain compression* (discussed in subsection 4.6.4), the `drop` instructions are eliminated with their corresponding computations.

4.5.5 Liveness Analysis

A final important prerequisite for synthesis of fragments in the graph is *liveness analysis* (also called *live variable analysis*). This analysis finds which variables are *live* at each program point. A variable is *live* at some program point p if its value is possibly read before it is written to again. Listing 4.8 displays an example of liveness analysis.

LISTING 4.8: Live Variables (Rust)

```
let x = 4;
// Live: {x}; x is live because y needs it
let y = 2 * x;
// Live: {y}; y is live because it is returned;
//            x is dead because it is not needed
return y;
```

Variables that affect a function's result or the external state are live. Any variables that contribute to those live variables are also live. For WebAssembly, the function's result and global variables are live at the end of a function. Inputs to functions and inputs to memory operations are also live. Note that it is safe to assume all variables are live ($\top$), though that provides no information whatsoever. Knowing which variables are *not* live *is* useful. Liveness analysis is performed as an instance of *backward* dataflow analysis.

Except for the terminal program point, all variables in the other program points are initialised as *dead*. As liveness analysis is backwards, the live variables at every program points depends on the liveness of variables in the next state and the (backward) transfer function for the corresponding edge. The transfer function is related to the edge's operations. The analysis is performed by repeatedly updating the liveness state at every program point. Formally, this ascends a lattice for every variable, which is given as:

$$\begin{aligned} \text{live} &= \top = \top \vee \top = \top \vee \bot = \bot \vee \top \\ \text{dead} &= \bot = \bot \vee \bot \end{aligned}$$

The $\vee$ (join / confluence)[19] operator denotes the lowest upper bound of its arguments. The confluence over two vectors A and B of length n produces another vector of length n with:

$$\forall i \in [0..n) \quad . \quad (A \vee B)_i = A_i \vee B_i$$

We implemented this backward analysis within our dataflow framework, where a transfer function is associated with every flow edge. Consider the rules below;

[19] In general, the combination operator in dataflow analysis is refered to as the *confluence* operator. In this specific instance, it corresponds to the *join* operator on the lattice.

where L represents the local variables and K is the stack (globals and memory are omitted here). n is the *previous* node of i in regular execution; it is the next node in *backward* flow.

If the output of a binary operator is live, then both its inputs are also live. Conversely, if its output is dead, then so are its inputs. Note that `binop`* represents the group of all binary operators.

$$\frac{i \rightarrow_{\text{flow}} (\texttt{FlowInstr binop}^*, n)}{L, K\ a \quad \Rightarrow_{\text{transfer}(i,n)} \quad L, K\ a\ a}\ \texttt{binop}^*$$

If the value taken from a local is live (on the stack), then that local was surely live before. If that stack value is *not* live, then the liveness of the local depends on other reads of that local (Hence, the $a \vee b$).

$$\frac{i \rightarrow_{\text{flow}} (\texttt{FlowInstr (local\_get } v), n)}{L[v \mapsto a], K\ b \quad \Rightarrow_{\text{transfer}(i,n)} \quad L[v \mapsto a \vee b], K}\ \texttt{local\_get}$$

Whenever a value is written to a local, that local was surely dead before. However, the written value (as taken from the stack) is only life if the local's value is live afterwards.

$$\frac{i \rightarrow_{\text{flow}} (\texttt{FlowInstr (local\_set } v), n)}{L[v \mapsto a], K \quad \Rightarrow_{\text{transfer}(i,n)} \quad L[v \mapsto \bot], K\ a}\ \texttt{local\_set}$$

"Tee'ing" a local is the operation where a value is written to it, but the value remains on the stack afterward. Like with `set_local`, the local is dead before the write. However, the liveness of the stack value beforehand depends on *both* the liveness of the local and that value afterwards.

$$\frac{i \rightarrow_{\text{flow}} (\texttt{FlowInstr (local\_tee } v), n)}{L[v \mapsto a], K\ b \quad \Rightarrow_{\text{transfer}(i,n)} \quad L[v \mapsto \bot], K\ (a \vee b)}\ \texttt{local\_tee}$$

This list is not exhaustive, but shows the essential rules to convey the essence of the analysis. Further rules are not given, as they are similar those stated above. Appendix A depicts the application of this analysis to an actual program.

Dataflow Speedup The liveness analysis is solved with an iterative algorithm[5, Chapter 17.4] which finds the least-fixed point. The algorithm visits nodes only after they have been updated by changes to previous nodes (being the next nodes in forward execution). Additionally, nodes are prioritized by their location in a (quasi-)postorder, which (roughly[20]) causes any node to be visited before its dependents. Only after a node's parents have converged, is the child updated again. If this child loops back to the parent, the parent may require re-visiting later. In practice, this traversal strategy shows a 8x speedup over non-ordered traversal for larger graphs.

[20] As the graph is cyclic, not every node may be visited before *all* its dependants. Choosing a "wrong" priority order does not affect the correctness of the result, only the time taken for convergence.[5]

Application to Synthesis The liveness analysis provides another degree of context to the fragments in the graph, which is particularly useful during synthesis and verification. Variables which are dead at the end of a fragment do not affect the behaviour of the program afterwards. Thus, dead variables may be ignored during equivalence checking (i.e., they are vacuously equal). As fewer equivalences between variables need to be verified, the verification may go quicker. Listing 4.9 and Listing 4.10 illustrates the irrelevance of dead variables.

LISTING 4.9: Program π_A

```
(func (param $x i32) (result i32)
    (local $y i32)

  (get_local $x)
)
```

LISTING 4.10: Program π_B

```
(func (param $x i32) (result i32)
    (local $y i32)

  (tee_local $y
    (get_local $x)
  )
)
```

The program states at the end of the functions are *not* generally equal. In program π_A, $y contains 0 at the end. In program π_B, $y contains the function's output value. Yet, clearly both functions return the same value; local values are dropped upon function exit.

The liveness information lowers the constraints placed on replacements; a larger set of programs may satisfy the specification. Thus, hopefully the synthesizer finds a faster program. That is, assuming that programs actually contain dead variables. Appendix C shows the proportion of live variables for programs used in the benchmarks. On average, about 40% of the variables available at any program point are *dead*.

WebAssembly The benefit of liveness information is somewhat WebAssembly-specific. Other superoptimizers[52, 47] optimize programs for CPU architectures with about a dozen registers. WebAssembly functions, on the other hand, may contain *hundreds* of variables. Often, a program fragment uses several variables to store *intermediate* values, while other variables contain the fragment's "output". The intermediate values are dead after contributing to that output; this allows many variables to be ignored during equivalence checking (and synthesis). Typical compilers apply *register allocation* before generating machine code; intermediate registers are often reused. Though, liveness analysis could still be beneficial for (some) other superoptimizers.

4.6 Graph Superoptimization

Now that symbolic configuration and liveness information is available in for all nodes in our graphs, further transformations can be applied. After all, the configurations may enable the discovery of better optimizations on the graph.

4.6.1 Tree Conversion

A simple structural transformation of control flow follows from driving, which we previously discussed for abstract programs (in Definition 4.1 on page 39). As driving simulates program execution with partial knowledge of program input, the superoptimizer requires another data structure to represent these execution traces. This data structure is included in Listing 4.11. In practice, its edges are (roughly) *a subset* of those defined for process graphs.

LISTING 4.11: Process Tree Implementation (in Haskell)

```haskell
-- Similar to graph nodes
data Tree a
  = Tree a (TreeEdge a)  -- ^ tree inner node
  | TreeEnd a            -- ^ terminal node in a scope
  | TreeReturn a         -- ^ explicit function return
  | TreeTrapped a        -- ^ unconditionally trapped

-- A subset of the constructors for graph edges (except keep/drop)
data TreeEdge a
  = TreeEdgeInstr SimpleInstr (Tree a)
  | TreeEdgeIf FuncType (Tree a) (Tree a)
  | TreeEdgeCall FuncIdx (Tree a)
  | TreeEdgeCallIndirect FuncType (Tree a)
  | TreeEdgeKeepDrop KeptVals DroppedValsNE (Tree a)

type DroppedValsNE = NonEmpty ValType
```

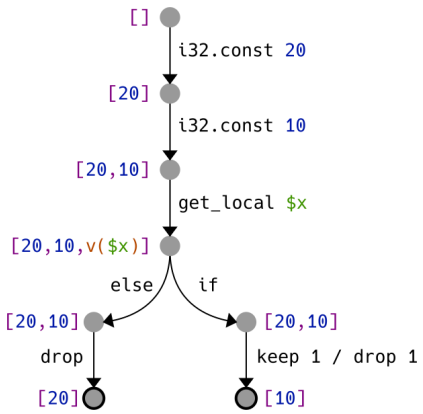
One *crucial* difference is in the representation of break statements, which are entirely absent in the tree. To illustrate the necessity for this change, consider Example 4.5.

Example 4.5 (Flattened Block)**.** Consider Listing 4.12 with its corresponding *process tree* in Figure 4.3.

LISTING 4.12: Program with `br_if`

```
1 (func (param $x i32) (result i32)
2   (block $b (result i32)
3     (i32.const 20)
4     (i32.const 10)
5     (get_local $x)
6     (br_if $b)
7     drop
8   )
9 )
```

FIGURE 4.3: Execution traces (Stack grows to the right)



Whenever `$x` is *true* the `br_if` on line 6 breaks out of the block, while passing 10 to the outer scope; 20 is discarded from the stack inside the block scope. If `$x` is *false*, 10 is explicitly dropped, while 20 is passed to the outer scope.

While these transitions are somewhat implicit in the WebAssembly semantics, this transition must be explicitly represented in the tree edges. Figure 4.3 illustrates that behaviour for the example, where the "keep 1 / drop 1" edge (`TreeEdgeKeepDrop`) represents behaviour associated with breaking from a block.

All `br` instructions are eliminated, and directly connected to the trees corresponding to their targets. All `br_if` instructions are replaced by `TreeEdgeIf` conditional statements. Currently, `br_table` instructions *cannot* be represented in the tree. The entire scope stack is passed as parameters to sub-scopes. Values that would have been dropped by explicit breaks (in the graph), are represented by `TreeEdgeKeepDrop` edges.

Note that the tree edges are *a subset* of those in the process graph. This property enables easy conversion *back* into the graph, provided that the tree is finite. The exception to this rule is the `TreeEdgeKeepDrop` edge, which corresponds to an empty block (in the AST) as follows:

$$\texttt{TreeEdgeKeepDrop}\ x\ y \quad \Rightarrow_{\text{toAST}} \quad \texttt{block}\ ((y +\!\!+ x) \to x)\ (\texttt{br}\ 0)\ \texttt{end}$$

Note that "$(y +\!\!+ x) \to x$" is the block's type signature. Arguably, this approach is a bit of a cheat; though, it is the only way to discard values that reside lower on the stack. The entire stack (containing values with types $y +\!\!+ x$) enters the block's scope stack. Upon explicitly breaking from the block, only the kept values (of types x) are passed to the parent scope, while the remaining values (of types y) on the scope stack are discarded.

4.6.2 Driving

While we previously defined driving in Definition 4.1 (on page 39), this section elaborates on its application to WebAssembly process trees.

Cyclic graphs may be converted to *infinite* trees. For the implementation, that is no problem, as Haskell expressions are evaluated *lazily*. However, process graphs (and their resulting programs) *are* finite, and only *finite* trees may be converted back into a graph. In practice, trees are driven *up to a bound* (e.g., 10,000 edges deep). Upon exceeding this bound, the tree is assumed to be infinite and *discarded*. The driving procedure attempts to evaluate symbolic values to constants, which includes branch conditions (akin to partial evaluation on graphs in subsection 4.5.4). When a branch condition is proven, it means every concrete execution - that reaches its program point - will take that branch; the other branch is never taken, and may be cut off. When all infinite branches are provably discarded, the tree is surely finite. Consider Listing 4.13 and its corresponding process tree in Figure 4.4.

LISTING 4.13: Bubble Sort (Rust)

```
static mut arr: [u32; 3] = [ 0, 0, 0 ]; // externally modifiable

unsafe fn bubble_sort( ) {
  for i in 0..arr.len() {
    for j in 0..arr.len() - 1 - i {
      if arr[j] > arr[j + 1] {
          arr.swap(j, j + 1);
      }
    }
  }
}
```

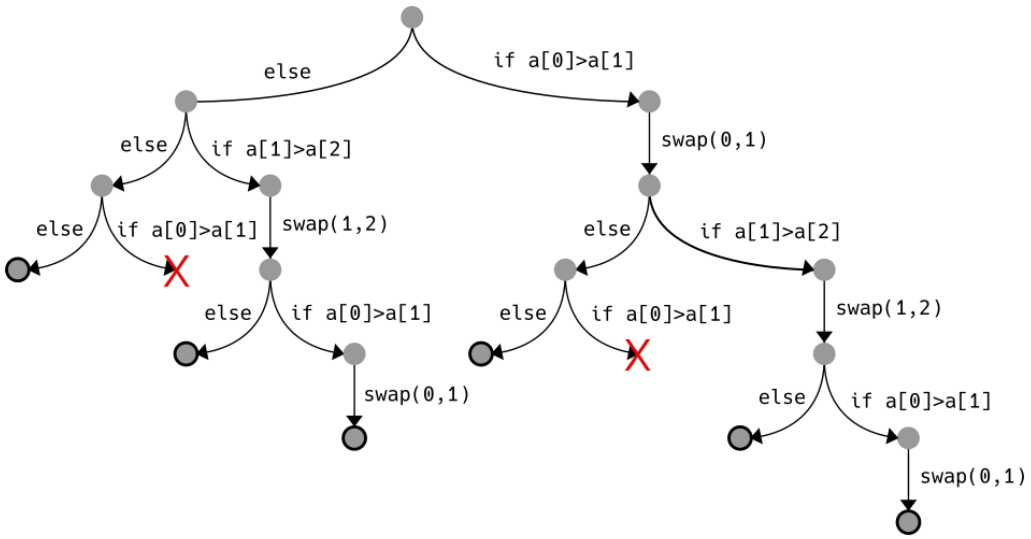


FIGURE 4.4: Bubblesort of *three* elements (pseudo-language)

The outer loop body is executed 3 times; the inner loop body is also executed a total of 3 times (2, 1, and 0 times, respectively). The process tree contains no edges that traverse the inner-most body more than thrice. All contained execution traces

are finite. Additionally, infeasible branches are marked with a red 'X'. In the actual driven tree, those branches are absent altogether (but are included in the figure for illustrative purposes). Note also that the indices are replaced by their respective constants. Effectively, driving makes those two modifications: Constants are replaced and branches are eliminates.

Likely, the driven bubblesort is faster than the original. Though, if the number of array elements were any larger than *four*, this transformation may not be so appealing; after all, the code size likely increases by quite a bit. Nonetheless, for small loops with low bounds, this transformation provides an interesting improvement.

4.6.3 Synthesis

Some program fragments may be replaced in their entirety by a linear sequence of instructions. Surely, this is the case whenever the original fragment is also a linear sequence of instructions. For some programs, branching or looping structures may be replaced by a linear instruction sequence in their entirety; one example is the computation of a *population count* as included in Listing 4.14, which counts the number of one-bits in a 32-bit integer.

LISTING 4.14: Population Count (Rust)

```
fn popcount( mut x: u32 ) -> u32 {
  let mut count = 0;
  while x != 0 {
    if ( x & 1 ) != 0 {
      count += 1;
    }
    x >>= 1;
  }
  count
}
```

While this program *does* loop, it never loops indefinitely; a 32-bit integer contains at most 32 one-bits. This entire loop can be replaced by a single `i32.popcnt` instruction, together with a `get_local` on the argument. By applying the synthesis procedure on this fragment, a large speedup may be achieved.

More often, though, a fragment cannot be entirely replaced by a linear sequence. Whenever a tree cannot be entirely replaced by a linear sequence, a subtree *could* still be replaceable by a linear sequence. In practice, it is difficult to find a replaceable subtree. A naive approach is to try replacing every subtree, until an appropriate replacement is found. The synthesis pruning rules (in subsection 3.3.3) can assist with eliminating trees (and their parents) for replacement. Still, in practice this is currently too expensive (as later discussed in subsection 5.4.1).

4.6.4 Transition Chain Compression

After fully driving a fragment, redundant instruction sequences may remain in the tree. For instance, the instructions computing an eliminated branch condition may remain, which are followed by a `drop` instruction (as produced by partial evaluation, as discussed in subsection 4.5.4). After driving terminates, the superoptimizer repeatedly applies the following compression rules to the transition chains:

$$
\begin{aligned}
\texttt{(}i\textit{n}\texttt{.const }c\texttt{) drop} &\Rightarrow_{\text{compress}} \epsilon \\
\texttt{(get\_local }v\texttt{) drop} &\Rightarrow_{\text{compress}} \epsilon \\
\texttt{(tee\_local }v\texttt{) drop} &\Rightarrow_{\text{compress}} \texttt{(set\_local }v\texttt{)} \\
\texttt{unop}^{*}\texttt{ drop} &\Rightarrow_{\text{compress}} \texttt{drop} \\
\texttt{binop}^{*}\texttt{ drop} &\Rightarrow_{\text{compress}} \texttt{drop drop} \\
\texttt{ternop}^{*}\texttt{ drop} &\Rightarrow_{\text{compress}} \texttt{drop drop drop} \\
i\textit{n}\texttt{.load}^{*}\texttt{ drop} &\Rightarrow_{\text{compress}} \texttt{drop}
\end{aligned}
$$

Dropped constants may be omitted entirely; similarly for dropped variable fetches. Whenever the result of an operator is dropped, the compressor eliminates the operator, and its inputs are dropped instead. Whenever the result of a memory load operation is dropped, the compressor eliminates the load operation, and drops its input (memory address) instead.

Note that these rules are *not* sufficient to remove all drops introduced by driving. In pathological cases, `drop` instructions would have to be propagated *into* block statements, or *over* larger sequences. As those cases are rare, we do not currently perform those "advanced" compressions. Ideally, synthesis (and/or verification) may be used to provably eliminate `drop` instructions in more complicated constructs. Synthesis may also reduce transition chains further, by synthesising alternate transitions chains. Sadly, synthesis is currently too costly to apply crudely in large programs (as further discussed in section 5.4).

Driving Heuristic Currently, we enforce a user-determined bound and timeout on the driving procedure. Ideally, driving should not expand "very large" process trees (e.g., like bubblesort on larger arrays - Figure 4.4). Sometimes, however, an optimization is only discovered after *many* expansions. For instance, for the `popcount` program (Listing 4.14 on page 60), the optimization is only possible after traversing the loop body 32 times. Consider also Listing 4.15.

LISTING 4.15: `babbage` (Rust)

```
fn run( ) -> u32 {
  let mut i = 0;
  while ( i * i ) % 1_000_000 != 269_696 {
    i += 1;
  }
  i
}
```

For this `babbage` program, the body must be traversed 25,264 times to establish the loop bound. While driving produces a very long transition chain, transition

chain compression eliminates it almost entirely. In general, it is difficult to infer *beforehand* how large the resulting program becomes *after* synthesis (for `popcount`) or transition chain compression (for `babbage`). Yet, both driving and synthesis require *many* Z3 invocations. Applying some heuristic to limit time spent on unprofitable expansions would be very beneficial. For this research, no heuristics have been discovered.

Instead, the superoptimizer enforces *timeouts* on distinct optimization tasks. Additionally, the superoptimizer enforces (user-specified) bounds on the driving depth. At least, these measures ensure that the superoptimizer eventually terminates; in the worst case, without producing any optimizations.

Chapter 5

Evaluation

We evaluate the superoptimizer by benchmarking optimized programs - whose performance is compared to the original program - and by manual inspection of performed modification. This evaluation consists of two parts:

- Small artificial programs - We constructed several small artificial programs. These programs contain several remaining optimization opportunities, which we expect our superoptimizer to find.
- Large realistic programs - The techniques applied to the smaller programs are not quite feasible for larger programs. Instead, we apply only partial evaluation to these larger programs. This category includes some projects that trivially compile to WebAssembly, such as the Lua interpreter. Additionally, programs from the *Rosetta Code Corpus* are included; another superoptimizer[11] used Souper[50] to optimize these same programs.

5.1 Small Artificial Programs

While these small programs offer obvious opportunity for optimization, it is surely useful to quantify the degree of achievable speed-up. Below, we describe the evaluation setup for small programs.

Benchmarking of compiler optimizations notoriously introduces *measurement bias*[42]; seemingly innocuous factors may significantly affect performance analysis. Examples of these factors are the *link order* of object files, and the length of the environment variable strings[1]. While our benchmarking setup may *not entirely* eliminate variation, caution is taken through the following methods:

- All programs are executed in three different *runtimes*; namely the Chrome webbrowser, the Firefox webbrowser, and the Wasmer[2] desktop runtime.
- Within each runtime, every program is executed a 1000 times, whose average is used.

Note that executing a WebAssembly function has some overhead cost by *marshalling* a value from the host environment into WebAssembly's environment. For example, all JavaScript numbers are stored as 64-bit floating points; when calling a WebAssembly function which accepts a `i32` argument, this value needs to be converted. Additional overhead may be present, depending on the runtime. This means, plainly measuring the speed-up of a single function call gives:

[1] Amusingly, running a program as Bob performs differently from running it as Alexander, as the environment variables include the username.

[2] `https://github.com/wasmerio/wasmer`

$$\frac{\textit{call-overhead} + \textit{time}(\pi_{\text{opt}})}{\textit{call-overhead} + \textit{time}(\pi)}$$

While this poses no issue whenever $\textit{time}(\pi_{\text{opt}})$ dominates *call-overhead*, the minuteness of the benchmarked programs may cause this overhead to dominate the execution time. To improve the accuracy of the measurements, we inserted another `runall` function into the WebAssembly modules (after superoptimization) that executes the function-under-measurement n times. Here, n represents the number of invocations of the measured function by WebAssembly itself. n is experimentally chosen depending on the program. Intuitively, think of the programs as illustrated in Listing 5.1.

LISTING 5.1: Measured program with `runall` (Rust)

```
fn run( arg: u32 /* other args here */ ) -> u32 {
  ... /* Measured function body */
}

fn runall( n: u32, arg: u32 /* other args here */ ) -> u32 {
  let mut res = 0;
  for i in 0..n {
    res = run( arg /* other args */ );
  }
  res
}
```

This approach ensures the measurements are closer to:

$$\text{speedup} \approx \frac{\textit{call-overhead} + n \cdot \textit{time}(\pi_{\text{opt}})}{\textit{call-overhead} + n \cdot \textit{time}(\pi)}$$

With a large enough value of n this approximates the intended measurement:

$$\text{speedup} = \frac{\textit{time}(\pi_{\text{opt}})}{\textit{time}(\pi)}$$

While this `runall` function may *also* introduce overhead, it is reasonable to assume that it is similar between runtimes; whereas for the call overhead that is less likely to be the case.

The benchmarked programs have between 0 and 3 `i32` inputs and produce 1 output; except for `bubblesort4`, which takes *four* `i32` values as input, produces no output, but modifies memory instead. The benchmarking programs sample uniformly over all possible inputs; this *should* produce an accurate portrayal of the program's "true" performance.

5.1.1 Results

The relative execution time of superoptimized programs is depicted in Figure 5.1. While some of these results seem impressive, keep in mind these programs were specifically selected because of optimizations missed by regular compilers, and a specific superoptimization was targeted at these problems. Nonetheless, these programs are illustrative of how a superoptimizer may go about optimizing them.

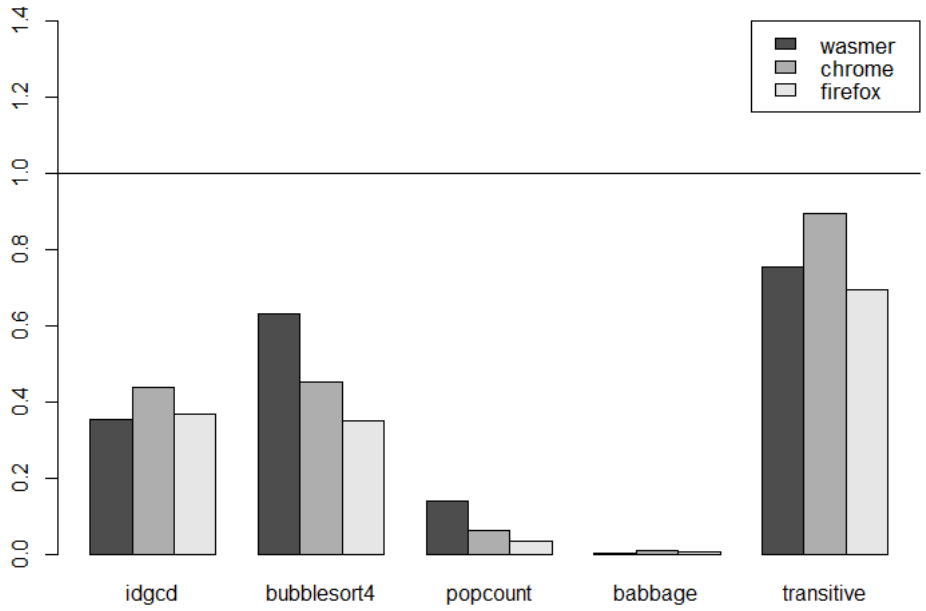


FIGURE 5.1: Execution time of optimized programs proportional to original programs (lower is better)

Table 5.1 contains the actual numbers associated with Figure 5.1. To produce the relative execution times, a Student's t-test with $\alpha = 0.05$ is performed, which produces a 95% confidence interval. Samples may vary slightly between executions, as caused by system interference (e.g., context switching). Reasonably, the interference should follow a normal distribution with mean 0. From the confidence interval, the *least optimistic* bound is taken; this means, for `idgcd` in Wasmer, the resulting execution time is 0.352 times the original time, *or less*, with 95% confidence. Surely, the performance of some programs increased much more than others; this variation depends largely on the non-optimality of the original programs. In the following sections, these results are analysed and discussed.

TABLE 5.1: Optimization times and proportional execution times of programs

File	Optimization Time	Execution Time Proportion		
		Wasmer	Chrome	Firefox
idgcd	563ms	0.352	0.438	0.367
bubblesort4	12,474ms	0.666	0.453	0.349
popcount	3,145ms	0.138	0.062	0.035
babbage	67,191ms	0.002	0.007	0.006
transitive	31ms	0.755	0.894	0.695

5.1.2 Runtime variation

An obvious observation is that the speedups vary wildly between runtime environments for the same program. While the execution time of `bubblesort4` decreased to only 66.6% of its original execution time in Wasmer, it decreased to 34.9% in Firefox. A reasonable explanation for these disparities is that these runtimes utilise different mechanisms for converting WebAssembly to machine code[3], or that the call overhead still dominates execution. In any case, all three WebAssembly runtimes display performance *improvements* for the listed programs.

[3] Admittedly, both Wasmer and Firefox seemingly use Cranelift for code generation.

5.1.3 Program: idgcd

Consider the function in Listing 5.2. The function returns the Greatest Common Divisor of `x` and "$2 \cdot$ `x`", which is surely `x` itself (after eliminating the possibility of integer overflow). While a human may trivially observe this input/output relation, these higher level observations may be non-trivial for a machine. Especially so when considering that the bound on the loop is non-obvious; that is, it does not iterate over a finite range. In reality, though, *no* concrete execution exists that traverses the loop body more than *twice*; Figure B.1 in Appendix B illustrates this.

After expanding the corresponding tree, the synthesizer observes only `i32`-arithmetic instructions are performed and some local variables (`$x`, `$b`, `$c`) are read and written to (see the full WebAssembly code in Listing B.1). The synthesizer aims to find a program with only those instructions, while also using the information that `$b` and `$c` are dead at the end. Finally, the synthesizer finds the simple replacement program as listed in Listing 5.3. Note, though, that unused locals are not currently removed.

LISTING 5.2: `idgcd` (Rust)

```rust
fn run( x: u32 ) -> u32 {
  if ( x <= 0x7FFFFFFF ) {
    // gcd( x, 2 * x )
    let mut a = x;
    let mut b = 2 * x;

    while b != 0 {
      let c = b;
      b = a % b;
      a = c;
    }
    a
  } else { // u32 overflow
    x
  }
}
```

LISTING 5.3: `idgcd` optimum (Wasm)

```wasm
(func $run
  (param $x i32) (result i32)
  (local $b i32) (local $c i32)

  get_local $x
)
```

5.1.4 Program: bubblesort4

The `bubblesort4` program contains an application of the *bubblesort* algorithm, which is specialized to an array of length *four*; its implementation in Rust is included in Listing 5.4. The main observation in this optimization is that infeasible branches are eliminated. Figure 5.2 depicts the tree of bubblesort applied to *three* elements (the tree for *four* elements is too large to display). The original implementation will always perform *six* conditional checks with corresponding branch. After elimination of infeasible branches, between *three* and *six* conditional breaks may be performed, depending on the input array. Surprisingly, this transformation makes the resulting program resemble an expansion of *insertion sort*. This optimization may shave off up to 75% of the execution time, depending on runtime and input, at an 774 byte increase in code size. Nonetheless, this optimization is surely interesting and - to the best of our knowledge - not found by other superoptimizers.

LISTING 5.4: Bubble Sort (Rust)

```
static mut arr: [u32; 4] = [ 0, 0, 0, 0 ]; // externally modifiable

unsafe fn bubble_sort( ) {
  for i in 0..arr.len() {
    for j in 0..arr.len() - 1 - i {
      if arr[j] > arr[j + 1] {
          arr.swap(j, j + 1);
      }
    }
  }
}
```

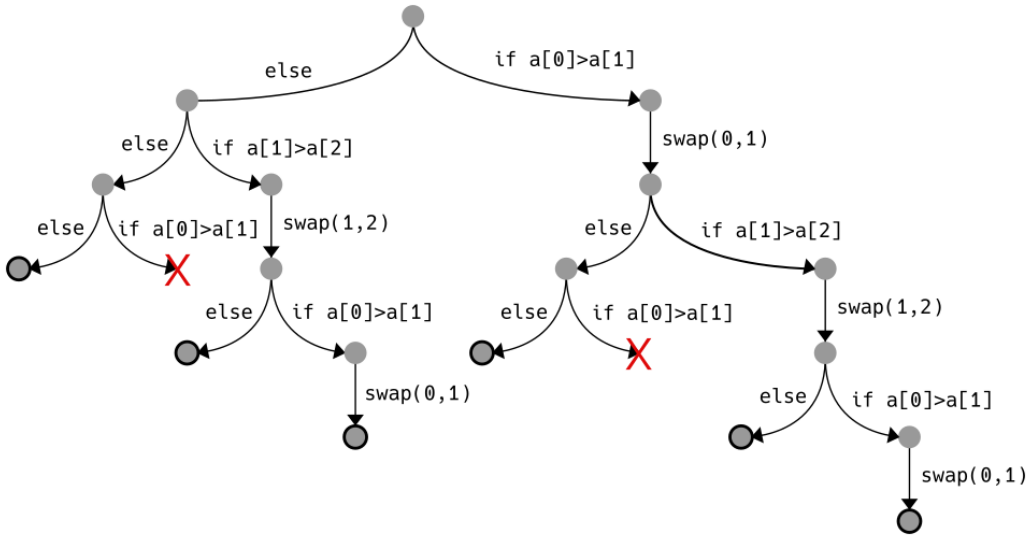


FIGURE 5.2: Process Tree for Bubblesort on *three* elements (Pseudo-language - copy of Figure 4.4)

5.1.5 Program: babbage

The `babbage` program answers Babbage's question:

> "What is the smallest positive integer whose square ends in the digits 269,696?"
>
> —Charles Babbage

The program is given in Listing 5.5. Our superoptimizer finds the optimization in Listing 5.6 by driving the loop for 25,264 iterations, after which the superoptimizer finds it to terminate. The transition chains reaching the final state are statically eliminated. While driving is implemented with symbolically represented configurations - surprisingly - Z3 *is never invoked* for this problem. As values for all variables are statically known, the "symbolic expressions" can always be simplified to constants at any point in the transition chains. Nevertheless, finding this optimization still takes about 67 seconds, which indicates there exists quite some overhead in driving for 25,264 loop iterations. Reportedly[11], the Souper[50] superoptimizer also finds this optimization; Souper uses an advanced CEGIS synthesizer to find the satisfying constant.

LISTING 5.5: `babbage` (Rust)

```
fn run( ) -> u32 {
  let mut i = 0;
  while ( i * i ) % 1_000_000 != 269_696 {
    i += 1;
  }
  i
}
```

LISTING 5.6: `babbage` optimum (Wasm)

```
(func $run (result i32)
  (local $l0 i32)

  i32.const 25264
)
```

5.1.6 Program: popcount

The `popcount` program counts the number of 1 bits in a 32-bit word; it is included in Listing 5.7. The loop iterates for no more than 32 iterations, as there are never more than 32 bits in a 32-bit word. The applied strategy is similar to the strategy of `idgcd` (see subsection 5.1.3). This entire body is replaced by a single `i32.popcnt` instruction with local variable fetch.

LISTING 5.7: `popcount` (Rust)

```
fn popcount( mut x: u32 ) -> u32 {
  let mut count = 0;
  while x != 0 {
    if ( x & 1 ) != 0 {
      count += 1;
    }
    x >>= 1;
  }
  count
}
```

LISTING 5.8: `popcount` optimum (Wasm)

```
(func $run
  (param $x i32) (result i32)

  get_local $x
  i32.popcnt
)
```

5.1.7 Program: transitive

`transitive` is a simple demonstration of how outputs from infeasible branches are not considered during equivalence checking. After driving, the synthesis procedure promptly finds the constant 1 as a replacement. Note, however, that if the original function returned 49 instead of 1, finding this replacement would be *considerably* harder; considering that only a select set of constants can currently be synthesized (See also subsection 3.3.2). However, in that case, the optimum *could* still be found by merging all equivalent feasible branches (which then all return 49).

LISTING 5.9: `transitive` (Rust)

```
fn run(a: u32, b: u32, c: u32) -> u32 {
  if a > b {
    if b > c {
      if a <= c { // Never true
        return 2;
      }
    }
  }
  return 1;
}
```

LISTING 5.10: `transitive` optimum (Wasm)

```
(func $run
  (param $a i32)
  (param $b i32)
  (param $c i32)
  (result i32)

  i32.const 1
)
```

5.2 Large Realistic Programs

On the large programs, we evaluate the usefulness of symbolic information - in the form of configurations - propagated over control flow. That is, some instructions are replaced by constants and branches are eliminated if their branch-condition is provable. Currently, applicability of the (superior) techniques as used on the smaller programs is infeasible for larger programs, as indicated by the superoptimization time measurements listed below.

While we *could* benchmark these programs, this is *not* done for the following reasons:

- WebAssembly modules are *not* programs. Instead, these modules depend on a large amount of host glue code (e.g., in JavaScript - See also subsection 2.2.5). Placing optimized WebAssembly modules back into their JavaScript context is surely useful to roughly *test* their correctness. However, manually augmenting (compiler-generated) JavaScript glue code to include performance measurements is error-prone.
- The improvements are sadly not expected to provide speed-ups larger than 1%, as indicated by results below. Going through the trouble of benchmarking these programs is thus rather futile.

For these programs, we run the superoptimizer on a Google Cloud Compute Engine *N2D* instance - running a 2nd Gen AMD EPYC CPU - with 256 GB of RAM. Realistically, 64GB of RAM should be sufficient to reproduce these results, though 256GB is used to avoid the occasional out-of-memory error after 5+ hours of execution (depending on parameters). While superoptimization *should* be inherently parallelizable, all programs are generated using a *single CPU*; this is currently necessary to avoid the occasional segmentation fault[4]. The results are listed in Table 5.2.

File	Timeout	Constants Replaced	Branches Eliminated	Time Taken	Relative Output File Size
⋆ bitwise_IO	1000ms	5/394	1/43	2,970 ms	99.56%
⋆ lua_mini	1000ms	6/1,280	0/78	168,392 ms	99.76%
sha256	1000ms	0/580	0/17	14,065 ms	99.91%
raytracer	200ms	N/A	29/2,277	137,784 ms	99.48%
raytracer	200ms	115/27,682	30/2,277	1,811,354 ms	99.35%
lua	200ms	N/A	15/5,125	234,514 ms	99.78%
lua	200ms	47/48,383	15/5,125	2,672,929 ms	99.75%
z3	200ms	N/A	803/487,686	35,171,365 ms	99.06%
z3†	50ms	807/2,086,551	437/262,036	~20hr	N/A

TABLE 5.2: Results of Constant Replacement and Branch Elimination
(More replacements is better)

[4]By some speculation, the segmentation fault seems to originate in either Z3 or its bindings with Haskell. As it often occurs only after *many* hours, it is hard to debug. It does not occur in single-threaded mode.

While all these programs are considered larger, their respective sizes vary wildly. The first three programs are small enough to assign a timeout of one second *per constant*. The programs below those are much larger, and are given a lower timeout of only 200ms. Also, measurements are included for either *only* branch elimination, or *both* branch elimination and constant replacement. While no performance improvements are listed, it stands to reason that *not* computing a value at runtime is surely faster than having to compute it.

In particular, the Z3 program (compiled to WebAssembly) is very large, at a file size of 17.4 MiB. It took almost 10 hours to eliminate a mere 803 branches. Constant replacement on Z3 (marked with †) was aborted after 20 hours, as it exceeded our willingness to wait. The entries marked with ⋆ were priorly optimized with Souper[50], on which some optimizations were still found. Below, the results are discussed.

5.2.1 Program: bitwise_IO

We took the `bitwise_IO` program from another superoptimizer[11], which compiled this same program to WebAssembly using Souper[50]. Our superoptimizer still found optimizations *after* Souper optimized it, which are shown in Listing 5.11 and Listing 5.12. Listing 5.13 shows the fragment before applying transition chain compression.

LISTING 5.11: input fragment

```
...
get_local $p0
i32.const 8
i32.add
tee_local $l0
i32.eqz
br_if $B4
...
```

LISTING 5.12: output fragment

```
...
i32.const 8
set_local $l0
...
```

LISTING 5.13: intermediate fragment

```
...
get_local $p0    ;; $p0 is proven to be always 0
drop
i32.const 0      ;; replace $p0 by constant 0
i32.const 8
i32.add
drop
i32.const 8      ;; ($p0 + 8) is proven to be always 8
tee_local $l0
i32.eqz
drop
i32.const 0      ;; (8 == 0) is proven to be always False
drop             ;; discard the `br_if`, because it never jumps
...
```

This optimization is not particularly profound. What is interesting, though, is that Souper did not find it. Further inspection indicates that between the assignment

of 0 to `$p0` and the value retrieval, an *external function* is called. Souper aborts the collection of path conditions upon encountering a function call[50]; even when the call has no opportunity to modify the variable. Our superoptimizer does not have *that* limitation.

Souper Comparison We emphasize that Souper *first* optimized the program; then we fed the (WebAssembly) program produced by Souper into our superoptimizer. Indubitably, Souper finds *many* great optimizations, for which our superoptimizer *pales in comparison*, in particular because Souper has a vastly superior fragment synthesizer. Nonetheless, it is interesting to see that opportunities for optimization remain for an alternate approach.

5.2.2 Program: lua_mini

Another program priorly superoptimized by Souper is `lua_mini`. This program is a Lua interpreter which does not include Lua's standard libraries (and is thus called "mini"). An interesting fragment is listed in Listing 5.14, while the corresponding optimized fragment is listed in Listing 5.15. The optimization is rather straightforward; a value written to memory is immediately read back again (and added to 1). Surely, that memory access operation can be eliminated. Souper missed this optimization as it lacks a model for memory[50].

LISTING 5.14: original fragment

```
...
get_local $10
i32.const 1336
i32.store offset=4
get_local $10
get_local $10
i32.load offset=4
i32.const 1
i32.add
...
```

LISTING 5.15: optimized fragment

```
...
get_local $10
i32.const 1336
i32.store offset=4
get_local $10
i32.const 1337
...
```

5.2.3 Program: sha256

While no replacements are applied to `sha256`, its file size is reduced by 2 bytes. This reduction is caused by a post-processing step where a write to a dead variable (`$g0`) is eliminated, as illustrated by Listing 5.16 and Listing 5.17.

LISTING 5.16: original fragment

```
...
tee_local $10
set_global $g0
...
```

LISTING 5.17: optimized fragment

```
...
set_local $10
...
```

5.2.4 Program: raytracer

The `raytracer` program generates images by tracing light rays into a 3D scene; it contains mostly floating-point arithmetic. This program I wrote myself priorly and spent considerable time manually optimizing it. It is surprising to see still more than 1% of the branches could be eliminated using the superoptimizer. The difference between 29 and 30 eliminated branches in the two runs can be explained by noise (e.g., OS-level context switching) interfering with Z3's ability to prove the branch condition within the alloted 200 milliseconds.

29 branches were eliminated in 2:17 minutes. *Also* replacing 115 constants took an additional 18 minutes. While these optimizations surely exceed those of plain LLVM[5], it is debatable whether *these* marginal gains are worth the required time.

[5] `raytracer` is written in Rust. The Rust compiler uses LLVM as backend.

5.2.5 Program: lua

The `lua` program is the full Lua interpreter compiled to WebAssembly. As it includes the standard libraries, it is remarkably larger than `lua_mini`. Relatively few improvements were found. Though, one interesting optimization is shown in Listing 5.18 and Listing 5.19. Two replacements are performed:

- First, `$l1` is provably 0 on line 13, as it is zero whenever `$p2` is zero. `$p2` is zero because it is used as the branch condition (and the else-case is considered). A simple dataflow zero analysis is incapable of determining `$l1` to be 0, as it is conditional on `$p2` being zero, which is yet unknown when `$l1` is set on line 6. The symbolic dataflow thus gives an interesting advantage.
- Finally, because `$l1` is dead after the if-block (as determined by liveness analysis) it is now also dead before line 8. Hence, the assignment to `$l1` on line 6 can be safely eliminated.

In principle, the subsequent `set_local` and `get_local` statements to `$p2` (in Listing 5.19) could be replaced by a `tee_local`, though this is not currently performed by the superoptimizer. Also, on line 14, `$p2` is already 0; yet it is assigned 0 again. Reassignments of previous values are not currently considered; though, doing so (in general) requires another Z3 call per assignment.

LISTING 5.18: original fragment

```
1  ...
2  i32.load8_s
3  tee_local $p2
4  i32.const 255
5  i32.and
6  set_local $l1
7  get_local $p2
8  if $I15
9    i32.const 0
10   set_local $l1
11   ...
12 else
13   get_local $l1
14   set_local $p2
15 end
16 ... ;; $l1 dead
```

LISTING 5.19: optimized fragment

```
...
i32.load8_s
set_local $p2
get_local $p2
if $I15
  i32.const 0
  set_local $l1
  ...
else
  i32.const 0
  set_local $p2
end
... ;; $l1 dead
```

5.2.6 Program: z3

The `z3` program is the Z3 SMT solver compiled to WebAssembly. The replacements are akin to those observed on previous programs. What makes `z3` unique is its enormous *size*, at 17.4MiB; it contains over 8 million instructions. Yet, at the same time, it represents the kind of realistic performance-intensive programs that would particularly benefit from superoptimization. It took roughly 10 hours to attempt elimination of almost a half million branches; these computation times are indicative of expected *costs* of superoptimization. While a single branch elimination is *relatively* cheap in computation time, the cumulative time amasses quickly to many hours.

These timings may also aid speculation on economical feasibility of superoptimization.

Only 0.16% of all branches were eliminated within 10 hours. Yet, the file size reduced by 0.94%. As WebAssembly contains `if`-blocks, whenever a branch condition is proven, either the entire `if` or `else` body may be eliminated. While only few branches are eliminated, increasing timeouts likely improves this a bit, at the cost of greater computation time. Additionally, note that Z3 was invoked for 487,686 branch conditions at 200ms each; this *would* equal a total time of 97,537,200ms (~27 hours). In practice, "only" 10 hours were spent, which happens - in many cases - because Z3 can prove no replacement is possible before the 200ms timeout is exceeded.

5.3 Synthesis and Verification

In this section the effectiveness of synthesis and verification is evaluated. We evaluate the enumerative synthesis (i.e., no CEGIS). Table 5.3 displays the synthesis times for some small programs.

Programs	Liveness		No Liveness		Equal Proof
	Time	Candidates	Time	Candidates	Time
popcount	3,145ms	80	timeout (120s)	348	499ms
idgcd	563ms	15	timeout (120s)	587	482ms
popcount64	45,519ms	142	segfault	N/A	5,624ms
mulshl$^{\epsilon}$	1,202ms	57	1,258ms	57	2ms
leqand$^{\epsilon}$	87ms	10	83ms	10	2ms
memarray	N/A	N/A	N/A	N/A	21,746ms

TABLE 5.3: Some synthesis timings

The 'Candidates' columns display the number of programs verified by Z3 before finding a correct program or synthesis timed out after 2 minutes. Programs marked with ϵ contain no dead variables.

One observation is that proving equality is generally more expensive than eliminating incorrect candidates. For `popcount`, 16% of the total time was taken to prove correctness. The 79 incorrect candidates were eliminated in the remaining 2,646ms (84%) of the time. On average, eliminating an incorrect candidate took only 1% of the time. For `idgcd` this disparity is even larger, where proving equality takes 85% of the time. Eliminating an incorrect candidate takes (on average) 1% of the time. Surely, when more candidates are considered, most of total time is likely spent on eliminating them. Generally, though, it seems that proving equality is more expensive than eliminating an unequal program.

The `popcount64` program is the 64-bit version of the `popcount` program. It has to consider a much larger set of instructions, namely those for `i32`-arithmetic (`i32`s are used as booleans) and `i64`-arithmetic. Its loop is driven for 64 iterations, which results in a large symbolic state. While otherwise similar to `popcount`, it takes almost 15 times as long to find a replacement. Verifying the correct program for `popcount64` takes 11 times the time taken by `popcount`. These examples indicate a large difference in computation cost may exist between seemingly similar programs.

5.3.1 Effects of Liveness Analysis

When variable liveness is *not* considered during synthesis, the synthesizer has to generate significantly more programs. As the liveness information lowers the requirement on equality - by considering dead variables as vacuously equal - more candidates satisfy the specification. When liveness information is considered, the synthesizer finds a replacement for `popcount` after eliminating 79 incorrect candidates within ~3 seconds. Without liveness information, no replacement is found after considering 348 candidates in 2 minutes. Similar behaviour can be observed for the `idgcd` program.

For the programs (`mulshl` & `leqand`) that contain no dead variables, liveness analysis has clearly no effect.

5.3.2 Memory

This section elaborates on the verification cost of programs containing memory.

Program: memarray

Consider the programs in Listing 5.20 and Listing 5.21. The left program traverses all *four* array elements *twice*, while the second only traverses them *once*. Both programs surely return the same output. When their corresponding WebAssembly programs are passed to our verifier, it takes over 20 seconds to prove their equivalence. Currently, synthesizing programs with memory operations is likely too costly for our superoptimizer.

LISTING 5.20: `memarray1` (Rust)

```
;; DATA has fixed length 4

let mut sum = 0;
for x in &DATA {
  sum += x;
}
for x in &DATA {
  sum += x;
}
return sum;
```

LISTING 5.21: `memarray2` (Rust)

```
;; DATA has fixed length 4

let mut sum = 0;
for x in &DATA {
  sum += x;
}
return 2 * sum;
```

Program: mem_le

Another program that operates on memory is `me_le` (LE stands for Little Endian). Consider two programs in Listing 5.22 and Listing 5.23. The former extracts the lowest 8 bits of `$a`, and stores the (zero-padded) 32-bit integer to memory address 0. The latter program stores the lower 8 bits of `$a` at this same address. These programs are subtly unequal.

LISTING 5.22: `mem_le` 1

```
i32.const 0
get_local $a
i32.const 255
i32.and
i32.store
```

LISTING 5.23: `mem_le` 2

```
i32.const 0
get_local $a
i32.store8
```

While both store the lower 8 bits of `$a` to address 0, the left program *also* stores 0 bytes at addresses 1, 2, and 3. Our verifier finds this error in 7ms with the following counterexample:

$$\text{model} = \{\text{params} = \{\texttt{\$a} \mapsto 0\},\ \text{mem} = \{2 \mapsto 32\},\ ...\}$$

It is remarkable to observe that subtle differences over memory are caught by the verifier (and Z3) within very little time.

5.4 Discussion

Below, we discuss the results and their applicability to practical programs.

5.4.1 Generalizability of small optimizations

The optimizations on the smaller programs are usually timed out at 10 *minutes*. Within that time limit, significant improvements are found on specific (artificial) control flow structures, on a *very small* scale. The *minuscule* timeouts (of one second or less) set for *very simple* improvements on larger realistic programs still causes *colossal* overall superoptimization times of up to 10 hours. If those individual timeouts were set to 10 minutes to apply better optimizations, the total time would (naively) extrapolate to roughly 6000 hours; which is 250 days.

Surely, the small programs are often optimized in about 1 minute; that is *already* difficult and costly. Yet, that is whenever it is *known* that the opportunity is present. *Finding* these opportunities in larger programs is much more difficult. Any program contains a number of sub-fragments that is exponential in the program size. After extracting an arbitrary fragment, there is little guarantee that - after driving and/or synthesis - it results in an improvement. So, even when one of our small programs occurs *exactly* inside a larger program, it requires significant brute-force effort to find and optimize them. Surely, that issue delineates the essence of superoptimization: *Finding non-obvious improvements*. While we currently limit search by timeouts, a better solution would be to (also) find and apply sensible *heuristics*. For instance, by using properties observed in the original fragments or by targeting the search to performance sensitive regions. These possibilities are further discussed in chapter 8.

5.4.2 Synthesis Limitations

Our synthesis algorithm is vastly inferior to those of many other superoptimizers (as discussed in chapter 6). Effectively, it is an enumerative brute-force synthesizer which uses some domain-specific pruning rules. Currently, this simple synthesizer is sufficient to find some simple improvements. Most of our approach side-steps synthesis by propagating symbolic information, applying partial evaluation, and driving loops. This approach enables the discovery of some unique optimizations. Particularly, driving naturally finds the `babbage` and `bubblesort4` replacements. However, the application of a *better* synthesis algorithm - which considers the propagated symbolic information - will likely find much better fragment replacements; this is also further discussed in chapter 8.

Chapter 6

Related Work

In this chapter, we elaborate on previous research on superoptimization, program synthesis, and supercompilation.

6.1 Superoptimizers

Massalin[37] coined the term *superoptimization* in 1987, as discussed in section 1.1. Later superoptimization research focused on achieving either or both of the following:

- *Generate Larger Programs* - Generate larger instruction sequences in reasonable time. Often this includes generating sequences containing forward-branches and sometimes backward-branches (i.e., loops).
- *Generate Compiler Optimizations* - Optimizations for a particular instruction set are discovered by traversing its program search space. These obtained optimizations are then implemented as *peephole optimizations* in an existing compiler; this reduces manual labor when constructing optimizing compilers for unknown CPU architectures.

Most unique contributions in previous research relate to obtaining (near-)optimal programs *faster*, through sophisticated search techniques and equivalence checks. Several of these are elaborated on below.

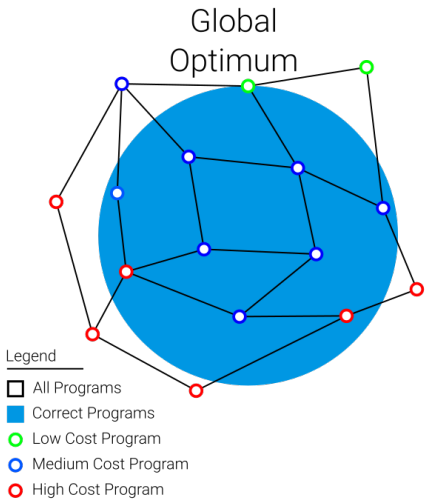
6.1.1 STOKE

As the program search space scales exponentially in the length of the program, there exists a limit on the number of states that can be considered in reasonable time. STOKE[52] instead traverses the high-dimensional irregular search space *stochastically* using *Markov Chain Monte Carlo* (MCMC). While this method is not guaranteed to find an optimal program, it can obtain significant improvements for much larger programs than otherwise possible. The optimization task is formulated as a cost minimization problem. The cost captures both competing requirements of speed and correctness.

Figure 6.1 abstractly depicts the search space. Correctness is captured by the bit-difference between the actual output and expected output when running the program on a set of input vectors. Correct and near-correct programs are very likely to be visited early because of their low cost. With MCMC sampling, a program *transforms* into an "adjacent" one in the search space. A single transformation is either of the following:

- Replace an opcode by a random opcode.
- Replace an operand by a random operand with equivalent type.
- Interchange two randomly selected instructions.
- Insert a random instruction.
- Remove a random instruction.

FIGURE 6.1: Abstract depiction of stochastic search space



Transformations that lower the cost are typically preferred, while deteriorations are occasionally allowed to avoid *local* minima. Repeated application of these last two rules (insertion and deletion) can transform any program into any other. Thus all programs in the search space can be encountered while likely encountering improvements quickly. Absolute correctness of a candidate solution is determined using an SMT solver.

Floating Points

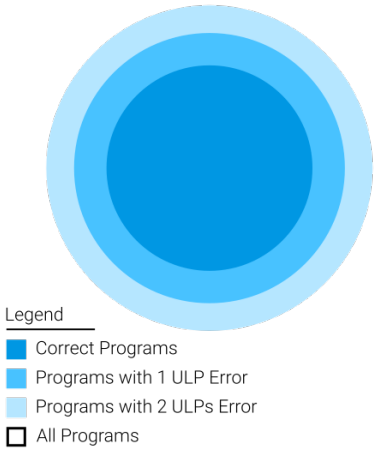


FIGURE 6.2: Abstract depiction of search space allowing errors

An addition to STOKE enables superoptimization of programs with floating-point calculations[51]. Notably, this technique allows errors, within reason. The error in the Units-in-the-Last-Place (ULPs) typically determines the accuracy of floating-point computations. Minor deviations in floating-point computations are usually acceptable, especially when performance significantly improves. Figure 6.2 abstractly depicts this search space. When allowing greater error, the global optimum differs. The programmer typically configures the allowed error margin.

So, STOKE traverses the high-dimensional irregular search space of loop-free programs quickly and is capable of finding (near-)correct near-optimal solutions. Correctness requirements are configurable by the programmer. When STOKE is run

for a sufficient amount of time, it finds an actually optimal program, because any program can transform into any other.

Conditionally Correct Superoptimization

Later, STOKE was extended with *Conditionally Correct* Superoptimization[54]. That work relies largely on an assumption we made as well: Program fragments need only be correct for the inputs upon which they are executed. While *we* aimed to establish fragment inputs through configurations tracked through our program, in that work, fragment preconditions are guessed from test cases. STOKE largely relies upon user-specified test cases to determine initial correctness. After synthesizing a fragment that is consistent with those test cases, formal verification would otherwise commence. In that work, however, the verifier also produces a *precondition* under which the fragments are correct. If the user is convinced that the precondition captures the set of realistic inputs, the program is accepted. Otherwise, the user provides additional test cases that are inconsistent with that precondition, and another fragment is synthesized. This process continues until STOKE finds a fragment with acceptable precondition.

Loops & Cutpoints

Churchill et al.[13] attempted to extend STOKE's stochastic search to loops. Program equivalence is initially determined by a bounded verifier. The bounded verifier determines the equivalence of all output for paths that traverse no single basic block more than k times in either the source or target program, where the user provides k. This method handles memory read/write operations. It is computationally expensive to determine the equivalence of memory layouts in the presence of pointers, as pointers may partially reference (*alias*) the same memory location. This may generate SMT terms that are exponentially large in the number of memory operations. To avoid this, the authors apply *alias relationship mining*, which attempts to place symbolic pointers in fixed-offset equivalence classes. Absolute equivalence is determined by a "sound verifier". This extends previous work[53], where a *simulation relation* is maintained while executing both programs. This simulation relation consists of *cutpoints*[62] and *invariants*. Each cutpoint λ corresponds to a location in both the source and target program, while an invariant ψ_λ describes their relationship. Upon execution of the instructions following λ in both the source and target program, both executions reach another cutpoint λ' where the obtained states satisfy $\psi_{\lambda'}$. Every loop has at least one cutpoint, and the source and target program must agree on the heap state at every cutpoint. As both programs must reach all cutpoints in the same order, it intuitively means both programs progress together on state changes. The invariants are obtained by looking for patterns between the programs' configurations when executed on test cases; a SMT solver formally verifies these. The observed relations are limited to register equalities and null checks, thus not all invariants are found. While not all program equivalences are discovered, no invalid program is ever produced.

Machine Learning

Another work[10] extends STOKE with *reinforcement learning*. The stochastic transformations applied by STOKE do not depend on the semantics of the considered

program, nor depend on past behavior. When the input program uses mainly bitwise operations, the optimal program does likely not rely on floating point operations. Bunel et al.[10] use a *neural network* to learn a better sampling distribution for the random transformations applied by STOKE. This approach ensures a more *goal-directed* stochastic traversal of the search space.

6.1.2 Denali

The Denali[30] superoptimizer relies on techniques otherwise used in SMT solvers, as the authors observed[30]:

> A refutation-based automatic theorem-prover is in fact a general-purpose goal-directed search engine, which can perform a goaldirected search for anything that can be specified in its declarative input language. Successful proofs correspond to unsuccessful searches, and viceversa.

Denali converts its input programs (written in a C-like language) into sets of *Guarded Multi-Assignments* (GMAs). An example GMA is as follows:

$$\texttt{p < r} \rightarrow \texttt{(M[p], p, q) := (M[q], p+8, q+8)}$$

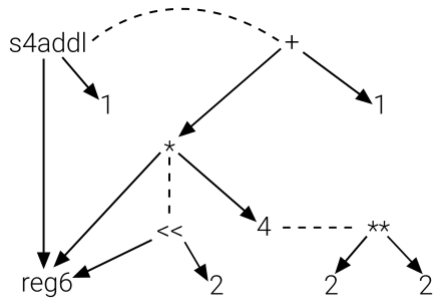
The values on the right-hand side are assigned to the variables on the left-hand side, but only if the condition (`p < r`) is true. Note that `M` represents program memory.

These GMAs are then converted into an *E-graph*, which is a DAG augmented with an equivalence relation on its nodes. Two nodes are equivalent when their terms represent identical values. Figure 6.3 represents such a graph. The equivalences are determined by encoded *mathematical axioms*, such as:

$$\forall x : 2 * x = x << 1$$

This process (called *matching*) builds a large E-graph representing all possible ways to compute the expressions in a GMA expression. An E-graph of size $\mathcal{O}(n)$ can represent $\Theta(2^n)$ ways of computing an expression of size n. As these cannot all be enumerated in reasonable time, an SMT solver is used to find the shortest expression.

FIGURE 6.3: An example E-graph for '`reg6*4+1`'. Dashed arcs represent equivalences. (Adapted from Denali paper[30])



The SMT solver is repeatedly asked (for different values K) whether a valid program of length K exists, while no program of length $K-1$ exists. Eventually, an optimal program is found while no faster program exists.

This technique can handle conditional forward-branches and memory access. However, the resulting program is not strictly optimal, but only "mathematically optimal" instead; that is, it is only optimal when considering the defined axioms, which are rarely comprehensive. Thus, the non-obvious bit-twiddling machine-specific programs are likely never found. Also, the size of the E-graph can scale exponentially with the input expressions, which causes Denali to perform poorly for larger programs; for example, consider '`a+b+c+d`' with commutativity and associativity axioms on '+'.

Equality Saturation

A line of research that was partially inspired by Denali is Equality Saturation[60], which is a general-purpose compilation paradigm. While it is *not* strictly considered superoptimization, we include it here because of its similarity to Denali. While Denali uses E-graphs, this technique represent *entire programs* in Program Expression Graphs (PEGs). These PEGs are referentially transparent, similar to gated SSA representations. A PEG completely represents a program. A *Saturation Engine* repeatedly applies a set of transformation axioms to saturate an E-PEG. This E-PEG simultaneously represents *multiple* versions of the input program; which is similar to - but more general - than an E-graph. This approach enables *non-destructive* application of optimizing transformations. As all optimizations are simultaneously applied to a program, it solves the prevalent *phase-ordering problem* for compilers (at the cost of increased compilation time). E-PEGs may also be used for *translation validation*; when two programs have equal saturated E-PEGs, then those programs are equal.

Typical compilers combine the decision of an optimization's *applicability* with the *profitability* of the optimization. This causes profitability of optimizations to be determined very locally; namely, potential future optimizations are not considered. Equality saturation allows the utilization of a *global profitability heuristic*, which picks the lowest-cost program from a saturated E-PEG.

A recent paper[63] describes `egg` (**e**-**g**raphs **g**ood), which is a Rust implementation of extensible E-graphs, specialized for performant equality saturation. The authors evaluated `egg` against an existing expression simplifier, which it outperformed by 3000×.

6.1.3 Souper

Souper[50] is a recent superoptimizer that operates on a purely functional directed acyclic dataflow graph resembling those in the LLVM IR. Notably, Souper stores optimizations in a Redis cache, which is a networked key-value store. Their entry in the cache replaces subsequent encounters of a previously superoptimized sequence; this allows superoptimization to be used similarly to incremental compilation, making it more suitable for practical development. Additionally, Souper obtains both static and dynamic profile counts. A static count is the number of times an instruction sequence is present in the input file, while the dynamic count is the number of times it is executed (which differ for instructions inside a loop). These counts can advise compiler developers on useful (peephole) optimizations on common cases that are currently missing. In the case of Souper, this resulted in the implementation of several optimizations in LLVM. In particular, Souper can quickly synthesize program fragments using its symbolic CEGIS synthesizer. This synthesis algorithm was inspired by the Brahma[22] tool, which we discuss in subsection 6.2.1. A recent extension[41] made Souper 2.32× faster by considering *dataflow facts*; these facts are

obtained *from the original fragment*. From the original fragment it observes, for instance, which bits are constant in the output (known bits analysis), or which input bits may affect the output (required bits).

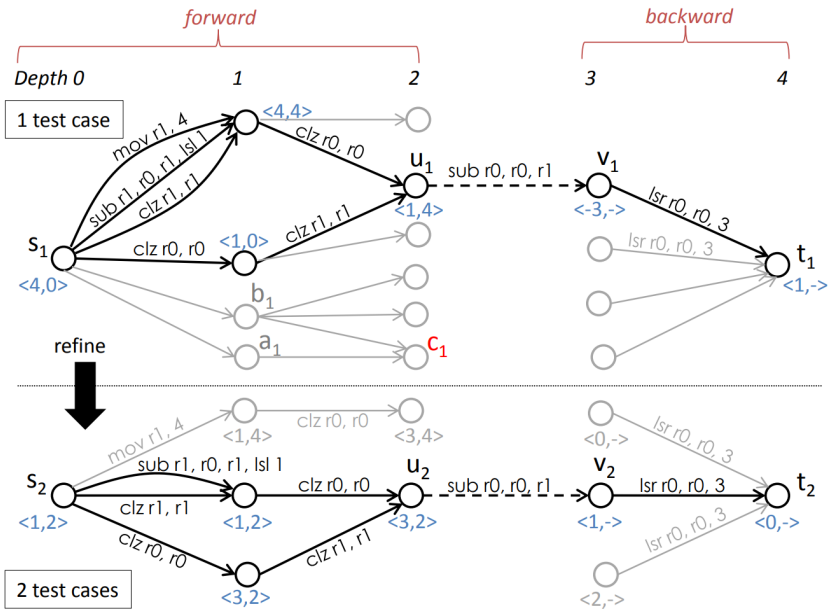
Arteaga et al.[11] applied Souper to WebAssembly, where C/C++ is compiled to WebAssembly through LLVM with Souper. Their objective was reducing binary size instead of execution time. We use their pipeline to evaluate our superoptimizer against (as discussed in chapter 5).

6.1.4 Lens

Lens[47] performs an enumerative *bidirectional* search while invalid candidates are pruned; this makes lens about 11 times faster than naive exhaustive search techniques. Figure 6.4 depicts an example search tree. The nodes represent configurations. Two paths are considered (preliminarily) equivalent if their value assignments match when executed on a test set. As paths may no longer be equivalent under different test cases, Lens continuously generates new test cases to refine the search space.

The backward search generates instructions at the end of the program, while "executing" the program in reverse on the test output. Equivalent states in either direction are easily identified through their variable-assignments for each test case.

FIGURE 6.4: Bidirectional search graph. Highlighted paths pass test cases (Taken from [47])



Once a seemingly correct program is obtained, a constraint solver determines absolute equivalence. Otherwise, a counterexample is obtained, which acts as another test case to refine the search space; this repeats until it finds a correct program.

The authors of Lens also experimented with a *cooperative search* technique, where multiple search instances (enumerative, stochastic, and symbolic) run simultaneously while sharing their current best solutions to aid the other instances. This technique allows Lens to outperform other superoptimizers (such as STOKE[52]) while generating truly optimal program fragments.

Note that executing instructions backward maps states one-to-many. Consider '$x + y = A$' where x and y are variable, while A is a known constant; in 32-bit

arithmetic there are 2^{32} distinct assignments to x and y summing to A. To make the backward search realistically tractable, Lens operates on a 4-bit instruction set instead. In the end, the obtained program is converted to its 32-bit equivalent.

6.2 Program Synthesis

Alonzo Church[12] performed early work on program synthesis, who aimed to generate a circuit from mathematical requirements. Later synthesizers attempt to generate a program satisfying some formal specification. We do not give a full account of synthesis research, but mention some interesting approaches.

6.2.1 Brahma

Brahma[22] is a *component-based symbolic synthesizer*, which Souper[50] also adapted to superoptimization (as discussed in subsection 6.1.3). Components available in the language (i.e., instructions) are described by a logical relation between input and output. Every such component is modeled as a *resource*, where the programmer is responsible for providing an upper-bound on the number of times each component is used. For instance, that the `add` instruction may be used at most twice; this ensures the set of resources is *finite*. A *synthesis constraint* describes the (large) set of programs that can be constructed from those resources. A *verification constraint* describes the correctness of such programs. This technique heavily relies on an SMT solver to propose candidate programs that are consistent with the *synthesis constraint*. A proposed candidate is verified using the SMT solver with the verification constraint. If incorrect, a test case is obtained and used to prune the set of candidates; this is thus an application of CEGIS. The synthesis-verification loop repeats until a correct program is found; in practice, 2 to 14 iterations were often sufficient.

Interestingly, during the ICFP 2013 contest, team Unagi[4] outperformed the Brahma approach when applied to an artificial[1] functional language. Within 72 hours - which was the time alloted to the contest - they wrote a brute-force enumerative synthesizer that distributed over 32 cores; domain-specific rules were applied to prune the search space. Their approach indicates that the application of large-scale computing resources can be surprisingly effective for superoptimization.

6.2.2 Functional Languages

Synthesis of functional languages is also extensively researched. These synthesizers leverage *type information* to prune the search space; after all, only constructors satisfying a term's expected type need to be generated. One approach[44] can generate programs satisfying user-provided input-output examples and (first order) type signatures. Later work[34] performed CEGIS synthesis on a functional language for a given *refinement type*. We do not elaborate on the synthesis of functional programs further, as it only minimally relates to our work.

[1] A simple functional language specifically constructed for the contest.

6.2.3 Automatic Program Repair

A related line of research aims to automatically repair bugs in programs. Repairing a program often involves modifying it such that it satisfies a test set. Angelix[39] performs symbolic execution on programs to extract *angelic paths*, which are execution traces that violate a test case. After finding multiple such paths (an angelic forest), their synthesiser aims to find a *patch*; this patch modifies an expression in the program such that those paths satisfy the test cases.

Genetic Improvement[46] (GI) improves programs for some given cost function. GI modifies programs through *Genetic Programming* (GP), where program fragments are moved within and between programs. GP can lead to improvements to execution time, energy and memory consumption, but also to the introduction of entirely new functionality. Especially when introducing new functionality, the output program is not identical in behaviour to the improved one; thus GI programs are often exclusively determined equal over a particular set of test cases. Though, in practice, a small amount of test cases is often sufficient to infer invariants for a program[18]. Often, accuracy can be sacrificed for greater performance (e.g., in video encoders or machine learning algorithms), which GI can automatically achieve[56, 57].

6.3 Supercompilation

Turchin[61] introduced supercompilation for this REFAL language (recursive functions algorithmic language). Turchin noted[61]:

> [Supercompilation] traces the possible generalized histories of computation by the original program, and compiles an equivalent program, reducing in the process the redundancy that could be present in the original program.

A program corresponds to a machine. A machine computing $g(x) = f(x, Y)$ where Y is constant can typically be more rigorously optimized than the machine computing $f(x, y)$ where x and y are variable. Even when x and y are variable, redundancies can often be eliminated in practice. While our work aligns with Turchin's view of programs and processes in spirit, Turchin's supercompiler is defined over his REFAL language, which is (more-or-less) a functional language. So, our implementation does not correspond directly with his theory (our application to process graphs was discussed in chapter 4).

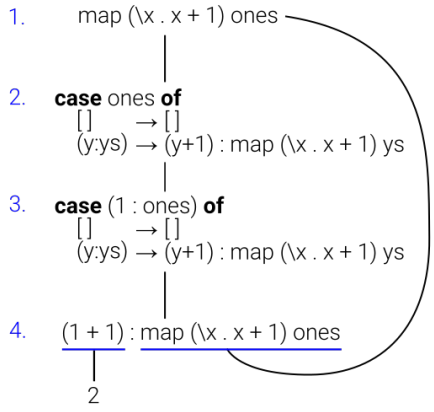
6.3.1 Purely Functional Supercompilation

Supercompilation is particularly useful for *purely functional languages*, where research is (arguably) most mature. Considerable optimization may be achieved; such as deforestation where intermediate data structures are entirely removed. One application of supercompilation to Haskell is by Bolingbroke[8], which extends techniques developed in preceding work[40]. An example optimization that can be achieved is as follows:

$$\begin{array}{l}\textbf{let}\ ones\ =\ 1\ :\ ones\ ;\ map\ =\ ...\\ \textbf{in}\ map\ (\lambda x\ .\ x\ +\ 1)\ ones\end{array} \quad\Longrightarrow\quad \textbf{let}\ xs\ =\ 2\ :\ xs\ \textbf{in}\ xs$$

Though this is a simple example, existing compilers do not typically perform this optimization as it applies a function to an infinite list. In supercompilation (on functional languages), optimization is achieved through *reducing*, *splitting*, and *matching*; Figure 6.5 depicts an example of this.

FIGURE 6.5: Haskell supercompilation



In this example, regard `map` and `ones` as defined within the context. Supercompilation starts at term 1, which is in Head Normal Form (HNF). It reduces to the definition of `map` in term 2. Then the definition of `ones` is required, which is substituted. In term 3 the case is known, thus it reduces to term 4. At that point no further reduction are applicable, so it *splits* on the cons constructor (:) where both subexpressions are independently supercompiled. `1+1` trivially reduces to `2`. However, the tail *matches* a previously encountered expression which is still being supercompiled. It is replaced by the result of supercompiling term 1 (which, by recursion, is the result of term 4). This process thus obtains the final expression as listed above.

Note that the number of encountered sub-terms is finite, as all sub-terms originate from the source programs. When expanding the process tree, a history of encountered states is maintained. When a *well-quasi-order* on the history is observed, expansion terminates. Conceptually, it terminates when no *new* expressions are encountered along the expanding tree. This termination ensures that diverging fragments are not expanded forever; which, for example, happens when recursing with an accumulator.

Supercompilation is notorious for its explosion of code size, as all operations are specialised to their context; this effectively inlines them. While execution speedups of a 100% are occasionally observed, the binary size can be 10 times larger for practical projects. One way[29] of avoiding this explosion is through *speculative* supercompilation, where terms that grow too big are discarded and thus not supercompiled. Many *small* functions - with few syntactic nodes each - account for most execution time; supercompiling these improves performance a lot, while binary size increases only slightly.

6.3.2 Imperative Supercompilation

The application of supercompilation to imperative languages has been sparse. In one approach[33], supercompilation is applied to Java. Java operates - like WebAssembly - as a stack based machine. The evaluation stack is explicitly included in their configurations when driving. Within their configurations, only simple constraints are placed on values: $v \geq c$, where v is a variable and c is a constant. While these constraints are simple, it allows for easy generalization. The author defines an *homeomorphic embedding* on their configurations, which is used to determine when to terminate driving. A homeomorphic embedding is one specific well-quasi order (as

described in Haskell supercompilation). In this application to Java, the obtained process graph is not used for optimization; instead, it is used for sound verification. It determines whether some condition holds at *any* terminal configuration following every valid process in the graph.

Chapter 7

Conclusion

In this work we consider superoptimization as finding non-obvious program optimizations with little regard for time taken by the optimizer. The targeted programs are more complicated than linear instruction sequences; in particular, they containing statically bounded loop. Concretely, we aim to answer the following questions:

- How can superoptimization be extended to handle statically bounded loops?
  - How can superoptimization be applied to a process tree?
  - How can these optimizations be applied to a finite process graph?
- How effective are these techniques in improving the performance of WebAssembly functions?

RQ1A: How can superoptimization be applied to a process tree? In chapter 4, we describe how our superoptimizer expands (small) graphs into finite process trees. Our approach of driving process trees with an SMT solver (Z3) is novel. Through driving, our superoptimizer removes infeasible branches; eliminating all *infinite* branches produces a *finite* tree. Sometimes, our synthesizer can replace a finite tree by a linear instruction sequence; it prunes the search space using static information extracted from the tree. Chapter 3 elaborates on that approach.

RQ1B: How can these optimizations be applied to a finite process graph? In chapter 4 we discussed the correspondence between process graphs and trees. Small graphs can sometimes be converted into trees and optimized as such. For larger programs, we propagate symbolic information as dataflow over control flow structures, but *not into* loops. We use this information to partially-evaluate expressions and eliminate branches. Brute-force partial evaluation of expression with contextual symbolic information is also novel.

RQ1: How can superoptimization be extended to handle statically bounded loops? Programs containing loops may be superoptimized by converting them into process graphs and applying our techniques as discussed in RQ1A and RQ1B.

RQ2: How effective are these techniques in improving the performance of WebAssembly functions? In chapter 5, we evaluate the performance of discovered optimizations, and the time taken by our superoptimizer. While the execution time of small artificial programs is reduced *by several orders of magnitude*, the used techniques are currently infeasible for larger programs. On larger programs, simple improvements such as branch elimination and constant substitution already take multiple hours but improve programs by no more than 1%.

Chapter 8

Future Work

During our research, we discovered problems that we were unable to solve; these are delineated below. Additionally, we elaborate on potential for research opportunities that are similar to our approach.

8.1 Fragment Search

We were unable to apply the techniques used on small programs to larger programs. A large limiting factor is the inability to discover *were* to look for opportunities. Extracting arbitrary fragments through brute-force proves infeasible. Two approaches *could* assist in resolving this (either independently or together); these are discussed below.

8.1.1 Heuristic

Ideally, the superoptimizer would *predict* where optimization opportunities reside. Though, by definition, superoptimization involves finding *non-obvious* optimizations; this implies that prediction is likely hard. In practice, we observe that some optimization opportunities only emerge after driving a loop for *many* iterations. For instance, the tree for the `popcount` program (in subsection 5.1.6) traverses the loop body 32 times and has 32 leaves. The driven tree of the `babbage` program (in subsection 5.1.5) consists of a single linear chain with over 50,000 transitions (which are later eliminated). Yet, perhaps some observations may be made about their common behaviour that allows the superoptimizer to make an *informed guess* about optimization opportunities. Effectively, this would establish a *heuristic* to guide the search for profitable fragments. One (rough) heuristic could be:

- If the original fragment *does* access memory in *more than one* path, then do *not* spend time on it.
- Otherwise, fully drive it and attempt synthesis.

The issue remains on where to start looking. After all, any program point may be selected as the root of some fragment. Similarly, any (dominated & post-dominating) descendant may be selected as the fragment's terminal node. The heuristic above could *possibly* be used to select such a fragment. Likely, some better heuristic is necessary to find fragments.

8.1.2 Profiling

Another potential solution is to target the search effort at performance-critical sections. Instead of attempting superoptimization for *every* fragment, it may only be attempted for, say, 5% of the most performance-critical regions. Surely, this approach requires knowledge of *where* those regions are. Possibly, those regions could be identified through *profiling*; which involves observing realistic program executions and reporting on often-visited program points. As web browsers contain extensive CPU profiling tools, this should be achievable. Though, those tools seemingly measure the time spent on WebAssembly *function calls*, and may - to no surprise - identify that most time is spent on the 1,000+ instruction-long main functions. With some manual effort and inspection, smaller critical WebAssembly functions could be discovered. Finally, this profiling data must be communicated to the superoptimizer.

Whenever an application developer *knows* where the performance-critical regions in a program are (possibly also through profiling), then superoptimization could be manually targeted to some regions. A caveat is that developers likely identify performance-critical sections in the *source code*. Finding the corresponding fragments in the compiled WebAssembly program remains non-trivial.

8.2 Symbolic Dataflow Analysis

Through the propagation of symbolic configuration over (forward) dataflow analysis, we discovered (in section 5.2) that programs often contain expressions that may be replaced by constants; particularly, regular compilers missed those optimizations. While *our* gains were marginal, future techniques could benefit.

8.2.1 Abstract Interpretation

With *abstract interpretation*, program points are assigned elements obtained from a lattice of reasonable size; for instance, the *constant propagation* lattice. While less powerful than symbolic information, its computation is relatively cheap; even *into* loops (depending on the lattice). We envision a system where abstract interpretation is combined with forward symbolic information propagation. For instance, from the fixpoint value at loop entries, a symbolic configuration may be constructed. In turn, from the propagated symbolic information, lattice values may increase (using Z3). Consider Listing 8.1. Constant propagation fails to observe that the expression on line 3 always evaluates to *false*. Using Z3, that may be proven with symbolic information and used to improve the corresponding value in the constant propagation lattice.

LISTING 8.1: False by transitivity

```
1 if a > b {
2   if b > c {
3     let x = ( a <= c ); // Always false. Tell constant propagation
4     ...
5   }
6 }
```

Generalization

We failed to appropriately *generalize* program points during driving (see subsection 4.2.1); this was particularly so because our symbolic representation is effectively an element in a *humongous* (but finite) lattice. Driving and generalization is quite effective[28] when using a smaller (but imprecise) lattice. Similar to the previous point, a smaller lattice may be used to *generalize* program points, while using symbolic information to provably eliminate branches. In our view, that approach inherits the best of both worlds.

8.2.2 Simple Changes

Currently, our superoptimizer only uses symbolic information to replace constant expressions in larger programs. A simple extension is to provably eliminate value *reassignments*. That is, if variable `x` already contains some value *a*, then another assignment of that same value '`x := a`' is surely futile. Those imperfections do exist in programs (see subsection 5.2.4); though, it is unknown *how often*. Similarly, memory *re-access* may be avoided. If an existing variable `y` already contains the value read from `mem[i]`, then an assignment `z := mem[i]` may be replaced by `z := y`. Again, it is unknown how often those imperfections occur or what their performance gain is. On the other hand, these checks should be relatively cheap in the context of superoptimization.

8.2.3 Larger Observations

Our superoptimizer may propagate symbolic information out of loops. Yet, we overapproximate significantly, which could be improved. Consider Listing 8.2 and Listing 8.3. While the branch condition on line 4 is always true - as it follows from the postcondition of `sort` - it is very hard (and very undecidable) to prove this in general. However, when `xs` is *known to be* sufficiently short, this optimization can be found. Proving this requires propagating a configuration *into* the `sort` function, fully drive its body, combine the configurations at the tree leaves, and return the combined configuration to the caller.

LISTING 8.2: Suboptimal `f`

```
1 pub fn f( xs: &mut [u32] ) {
2   // assume: xs.len( ) >= 2
3   sort( &mut xs );
4   if xs[ 0 ] <= xs[ 1 ] {
5     foo( );
6   } else {
7     bar( );
8   }
9 }
```

LISTING 8.3: Optimal `f`

```
pub fn f( xs: &mut [u32] ) {
  // assume: xs.len( ) >= 2
  sort( &mut xs );
  foo( );
}
```

8.2.4 Configuration Reductions

Our configurations are represented symbolically, which means their representation is much smaller than concretely enumerated sets. Still, their expression count often ranges into the *multiple thousands* (see subsection 4.5.3), which incurs a high memory cost; and their size likely also incurs a cost on the verification time spent in Z3. While we apply some expression simplification rules, it could be useful to expend some Z3 calls toward simplifying the configurations. Probably, the necessity for exploring this idea is predicated on the success of the previous points.

8.3 Larger Superoptimizer

Most existing superoptimizers have been research projects, while widespread industry adoption has been minimal. Yet, we argue that implementing a superoptimizer - like many compilers - is more a software engineering challenge than a research one. In particular, distributing the search over multiple machines may be *necessary* to obtain good results. As program fragments may be independently superoptimized, this should pose little conceptual problems; only an implementation one. During the ICFP 2013 programming contest, team Unagi[4] showed the surprising effectiveness of distributed brute-force enumerative synthesis. Additionally, many great synthesis algorithms[50, 47, 52] exist, which are time-consuming to reproduce, but are very useful when implementing a superoptimizer. Particularly, we envision a better synthesizer may leverage the symbolic information we propagate over control flow to find better replacements.

In any case, to effectively research superoptimization, the existence of a solid *superoptimizer framework implementation* would be very beneficial. This is, arguably, why recent research[11, 41] builds on Souper[50].

8.3.1 Side-effects

While we propose a way of proving equivalence over side-effects (in subsection 3.2.7), it does not quite contribute to our results. We expect its application together with a better synthesis algorithm may be very profitable.

8.4 High-level Language

Supercompilation[61] (not super*optimization*), from which we took process graphs, was largely established to translate *between* languages. We see potential in a high-level language, whose execution is simulated with a process graph. Through synthesis, the represented process may be projected as an efficient program in some low-level language.

When a regular compiler transforms a program – written in a high-level language – to machine code, that compiler already makes *many* decisions. For instance, on the chosen memory layout, or by deciding how undefined behaviour (at the high level) translates to machine code. Simultaneously, a lot of the high-level information is lost during conversion. When synthesis is applied *during* supercompilation from a high-level language, the restrictions on synthesis are lowered as few decisions are yet made. Success in this direction may produce programs that are faster than are possible with superoptimization alone.

Appendix A

GCD Liveness Analysis

Figure A.1 shows the application of liveness analysis to a GCD (Greatest Common Divisor) function. The annotations consist of a tuple representing the local variables (`$a`, `$b`, `$c`) and the program stacks. Every entered scope adds another scope stack. The analysis starts at the terminal node, where only the stack value is live.

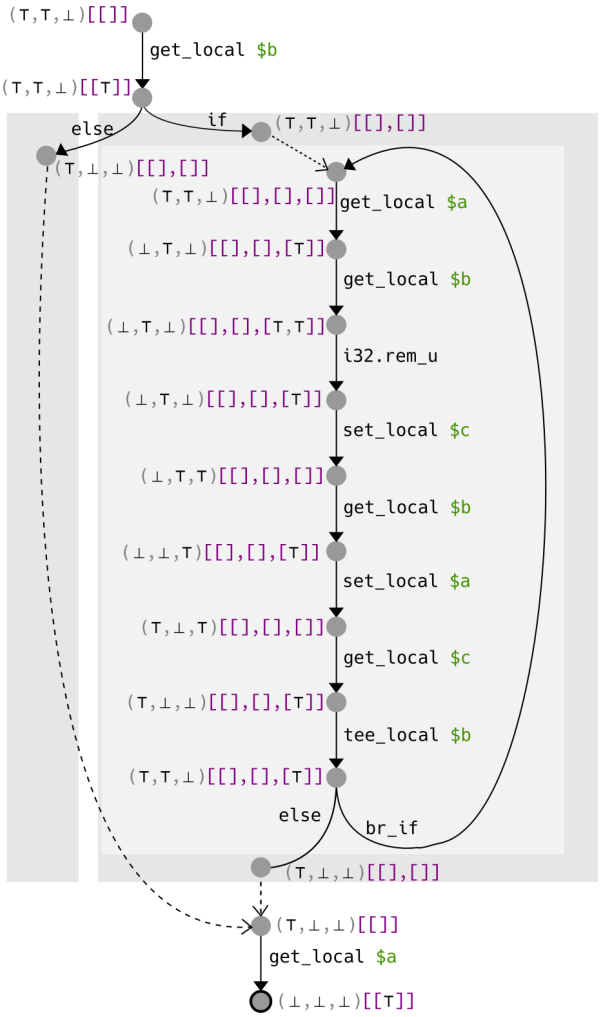


FIGURE A.1: GCD Liveness Annotations

Appendix B

Expanded GCD tree

The code in Listing 5.2 (in subsection 5.1.3) shows a strange identity function, which is computed using the Greatest Common Divisor. The (static) bound on its execution is non-obvious. However, no concrete execution will execute the loop body more than twice; Figure B.1 illustrates this.

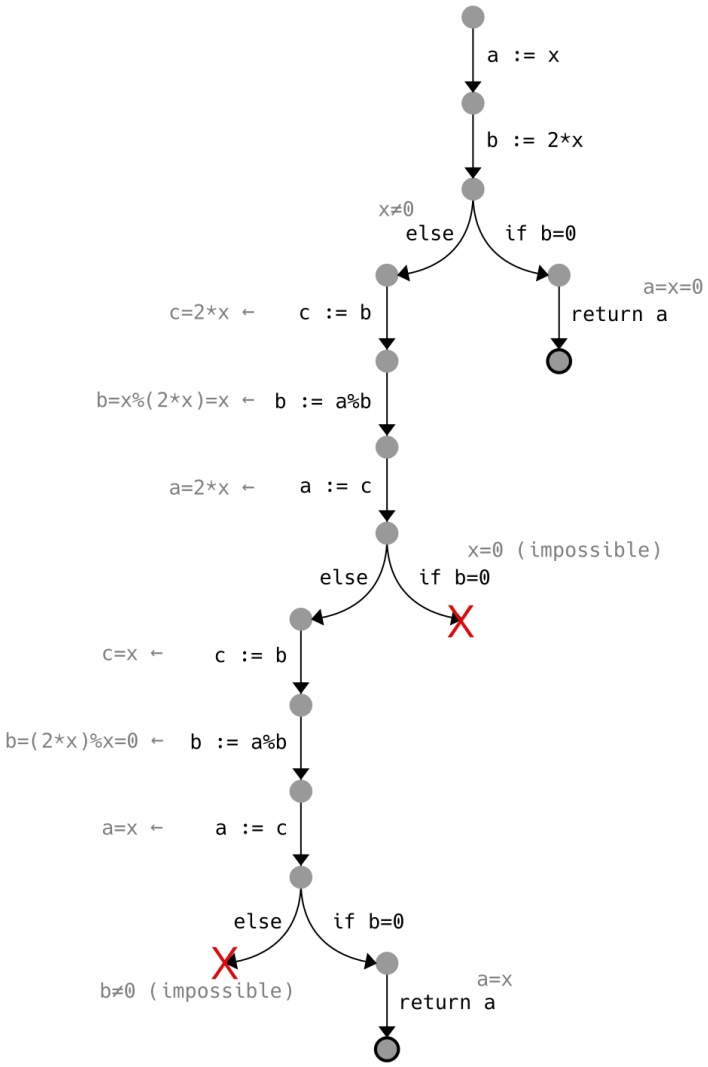


FIGURE B.1: Fully expanded `idgcd`
(without overflow check - pseudo language)

The compiled WebAssembly version of the full program is included in Listing B.1.

LISTING B.1: Strange identity function (WebAssembly)

```
(func $run (export "run") (type $t0) (param $x i32) (result i32)
  (local $b i32) (local $c i32)
  block $B0
    get_local $x
    i32.const 0
    i32.lt_s
    br_if $B0
    get_local $x
    i32.const 1
    i32.shl
    tee_local $b
    i32.eqz
    br_if $B0
    loop $L1
      get_local $x
      get_local $b
      tee_local $c
      i32.rem_u
      set_local $b
      get_local $c
      set_local $x
      get_local $b
      br_if $L1
    end
    get_local $c
    return
  end
  get_local $x
)
```

Appendix C

Liveness Proportions

Below, the proportion of dead variables over all program points in the benchmark programs are listed. Variables which are *dead* at the *end* of a program fragment need not be considered during equivalence checking. See subsection 4.5.5 for further elaboration. Some programs contain no global variables.

Do keep in mind that a local variable is dead at the program point *preceding* a write instruction. So, a program with a single local, that consists only of alternating read (`get_local`) and write (`set_local`) instructions, has 50% dead locals. Realistically, programs contain longer chains where a variable remains dead (or alive).

Program	**Stack**	**Locals**	**Globals**	**Total**
idgcd	0.000	0.593	N/A	0.471
bubblesort4	0.000	0.542	N/A	0.481
popcount	0.000	0.535	N/A	0.430
babbage	0.000	0.476	N/A	0.250
transitive	0.000	0.667	N/A	0.522
bitwise_IO	0.009	0.292	0.003	0.235
lua_mini	0.012	0.604	0.041	0.230
sha256	0.002	0.557	0.109	0.434
raytracer	0.001	0.582	0.116	0.539
lua	0.005	0.417	0.004	0.207
z3	0.002	0.466	0.025	0.335

TABLE C.1: Proportion of dead variables over all program points

Appendix D

Unpredictability of Z3

Both SAT and SMT solving are computationally expensive (NP-complete[1]). It is surely impressive that Z3 (v4.8.8) finds satisfying assignments to large formulas in little time. Occasionally, however, Z3 has trouble finding a model at all. Consider the polynomial $(a+1)^4$ with its expanded form, where a is a 32-bit bitvector:

$$(a+1)^4 \quad \equiv \quad a^4 + 4 * a^3 + 6 * a^2 + 4 * a + 1$$

Also consider the following axiom on bitvectors, where $\ll$ denotes the left-shift operator:

$$4 * a \quad \equiv \quad (a \ll 2)$$

Z3 can prove the respective equality of the following two expressions almost instantaneously:

$$4 * a \quad \equiv \quad (a \ll 2) \tag{D.1}$$
$$(a+1)^4 \quad \equiv \quad a^4 + (a^3 \ll 2) + 6 * a^2 + 4 * a + 1 \tag{D.2}$$

Yet, proving the equality of the following expression takes near-infinite time:

$$(a+1)^4 \quad \equiv \quad a^4 + (a^3 \ll 2) + 6 * a^2 + (a \ll 2) + 1 \tag{D.3}$$

This disparity in time is interesting, as Equation D.1 is substituted in Equation D.2 to produce Equation D.3. Apparently, some axioms are not applied in a larger context. In general, it is hard to reason about the expected time it takes for Z3 to prove no satisfying assignment exists (which we need to determine equality). In practise, we found that enforcing a timeout while accepting missed optimizations is the only way around this issue.

[1] SMT solving is *undecidable* in general. On *quantifier-free* formulas with bitvectors, bitvector arrays, and uninterpreted functions, it is NP-complete.

Bibliography

[1] WebAssembly Roadmap. https://webassembly.org/roadmap/ [Online; accessed Mar 12, 2021].

[2] WebAssembly Use Cases. https://webassembly.org/docs/use-cases/ [Online; accessed Mar 12, 2021].

[3] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84.

[4] AKIBA, T., IMAJO, K., IWAMI, H., IWATA, Y., KATAOKA, T., TAKAHASHI, N., MOSKAL, M., AND SWAMY, N. Calibrating Research in Program Synthesis Using 72,000 Hours of Programmer Time.

[5] APPEL, A. W., AND PALSBERG, J. *Modern Compiler Implementation in Java*, 2nd ed. Cambridge University Press, USA, 2003.

[6] BJØRNER, N., DE MOURA, L., NACHMANSON, L., AND WINTERSTEIGER, C. M. *Programming Z3*. Springer International Publishing, Cham, 2019, pp. 148–201.

[7] BÖHM, C., AND JACOPINI, G. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. *Commun. ACM 9*, 5 (May 1966), 366–371.

[8] BOLINGBROKE , M., AND JONES, S. P. Supercompilation by Evaluation. *ACM SIGPLAN Notices 45*, 11 (2010), 135–146.

[9] BRAIN, M., SCHANDA, F., AND SUN, Y. Building better bit-blasting for floating-point problems. In *Tools and Algorithms for the Construction and Analysis of Systems* (Cham, 2019), T. Vojnar and L. Zhang, Eds., Springer International Publishing, pp. 79–98.

[10] BUNEL, R., DESMAISON, A., KUMAR, M. P., TORR, P. H. S., AND KOHLI, P. Learning to superoptimize programs. *CoRR abs/1611.01787* (2016).

[11] CABRERA ARTEAGA, J., DONDE, S., GU, J., FLOROS, O., SATABIN, L., BAUDRY, B., AND MONPERRUS, M. Superoptimization of WebAssembly bytecode. *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming* (Mar 2020).

[12] CHURCH, A. Applications of recursive arithmetic to the problem of circuit synthesis. *Summaries of the Summer Institute of Symbolic Logic* (1957), 3–50.

[13] CHURCHILL, B., SHARMA, R., BASTIEN, J., AND AIKEN, A. Sound Loop Superoptimization for Google Native Client. *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Apr 2017).

[14] DE BRUIJN, N. G. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)* (1972), vol. 75, Elsevier, pp. 381–392.

[15] DE MOURA , L., AND BJØRNER, N. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), TACAS'08/ETAPS'08, Springer-Verlag, p. 337–340.

[16] DIJKSTRA, E. *Go to Statement Considered Harmful*. Yourdon Press, USA, 1979, p. 27–33.

[17] DOWNEY, P. J., SETHI, R., AND TARJAN, R. E. Variations on the Common Subexpression Problem. *J. ACM 27*, 4 (Oct. 1980), 758–771.

[18] ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. Dynamically discovering likely program invariants to support program evolution. *Proceedings of the 21st international conference on Software engineering - ICSE '99* (1999).

[19] GLÜCK , R., AND KLIMOV, A. V. Occams razor in metacomputation: the notion of a perfect process tree. *Static Analysis Lecture Notes in Computer Science* (1993), 112–123.

[20] GREEN, C. D. Classics in the History of Psychology. http://psychclassics.yorku.ca/Lovelace/lovelace.htm [Online; accessed Mar 12, 2021].

[21] GULWANI, S., JHA, S., TIWARI, A., AND VENKATESAN, R. Synthesis of Loop-Free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, Association for Computing Machinery, p. 62–73.

[22] GULWANI, S., JHA, S., TIWARI, A., AND VENKATESAN, R. Synthesis of Loop-Free Programs. *SIGPLAN Not. 46*, 6 (June 2011), 62–73.

[23] HAAS, A., ROSSBERG, A., SCHUFF, D. L., TITZER, B. L., HOLMAN, M., GOHMAN, D., WAGNER, L., ZAKAI, A., AND BASTIEN, J. Bringing the web up to speed with WebAssembly. *ACM SIGPLAN Notices 52*, 6 (2017), 185–200.

[24] HECHT, M. S., AND ULLMAN, J. D. Flow Graph Reducibility. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1972), STOC '72, Association for Computing Machinery, p. 238–250.

[25] IOZZELLI, Y. Solving the structured control flow problem once and for all, Apr 2019. https://medium.com/leaningtech/solving-the-structured-control-flow-problem-once-and-for-all-5123117b1ee2 [Online; accessed Mar 12, 2021].

[26] JANGDA, A., POWERS, B., BERGER, E. D., AND GUHA, A. Not so Fast: Analyzing the Performance of Webassembly vs. Native Code. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (USA, 2019), USENIX ATC '19, USENIX Association, p. 107–120.

[27] JONÁŠ, M. *Satisfiability of Quantified Bit-Vector Formulas: Theory and Practice*. PhD thesis, Masarykova univerzita, Fakulta informatiky, 2019.

[28] JONES, N. D. The Essence of Program Transformation by Partial Evaluation and Driving. *Lecture Notes in Computer Science Perspectives of System Informatics* (2000), 62–79.

[29] JONSSON , P. A., AND NORDLANDER, J. Taming code explosion in supercompilation. *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation - PERM 11* (2011).

[30] JOSHI, R., NELSON, G., AND RANDALL, K. Denali. *ACM SIGPLAN Notices 37*, 5 (2002), 304.

[31] KILDALL, G. A. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1973), POPL '73, Association for Computing Machinery, p. 194–206.

[32] KING, J. C. Symbolic Execution and Program Testing. *Commun. ACM 19*, 7 (July 1976), 385–394.

[33] KLIMOV, A. V. A Java Supercompiler and Its Application to Verification of Cache-Coherence Protocols. *Perspectives of Systems Informatics Lecture Notes in Computer Science* (2010), 185–192.

[34] KNOTH, T., WANG, D., POLIKARPOVA, N., AND HOFFMANN, J. Resource-Guided Program Synthesis. *CoRR abs/1904.07415* (2019).

[35] KROENING, D., AND STRICHMAN, O. *Equality Logic and Uninterpreted Functions*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 59–80.

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

[37] MASSALIN, H. Superoptimizer: A Look at the Smallest Program. *ACM SIGPLAN Notices 22*, 10 (1987), 122–126.

[38] MATSAKIS, N. D., AND KLOCK, F. S. The Rust Language. *Ada Lett. 34*, 3 (Oct. 2014), 103–104.

[39] MECHTAEV, S., YI, J., AND ROYCHOUDHURY, A. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (2016), pp. 691–701.

[40] MITCHELL, N. Rethinking supercompilation. *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming - ICFP '10* (2010).

[41] MUKHERJEE, M., KANT, P., LIU, Z., AND REGEHR, J. Dataflow-Based Pruning for Speeding up Superoptimization. *Proc. ACM Program. Lang. 4*, OOPSLA (Nov. 2020).

[42] MYTKOWICZ, T., DIWAN, A., HAUSWIRTH, M., AND SWEENEY, P. F. Producing Wrong Data without Doing Anything Obviously Wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2009), ASPLOS XIV, Association for Computing Machinery, p. 265–276.

[43] NIELSON, F., NIELSON, H. R., AND HANKIN, C. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.

[44] OSERA, P.-M., AND ZDANCEWIC, S. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015), PLDI '15, Association for Computing Machinery, p. 619–630.

[45] PERLIS, A. J. Special Feature: Epigrams on Programming. *SIGPLAN Not. 17*, 9 (Sept. 1982), 7–13.

[46] PETKE, J., HARALDSSON, S. O., HARMAN, M., LANGDON, W. B., WHITE, D. R., AND WOODWARD, J. R. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation 22*, 3 (2018), 415–432.

[47] PHOTHILIMTHANA, P. M., THAKUR, A., BODIK, R., AND DHURJATI, D. Scaling up Superoptimization. *ACM SIGOPS Operating Systems Review 50*, 2 (2016), 297–310.

[48] ROSSBERG, A. WebAssembly Specification. https://webassembly.github.io/spec/core/ [Online; accessed Mar 12, 2021].

[49] RÜMMER, P., AND WAHL, T. An SMT-LIB theory of binary floating-point arithmetic.

[50] SASNAUSKAS, R., CHEN, Y., COLLINGBOURNE, P., KETEMA, J., LUP, G., TANEJA, J., AND REGEHR, J. Souper: A Synthesizing Superoptimizer, 2017.

[51] SCHKUFZA, E., SHARMA, R., AND AIKEN, A. Stochastic optimization of floating-point programs with tunable precision. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 14* (2013).

[52] SCHKUFZA, E., SHARMA, R., AND AIKEN, A. Stochastic Superoptimization. *ACM SIGPLAN Notices 48*, 4 (2013), 305–316.

[53] SHARMA, R., SCHKUFZA, E., CHURCHILL, B., AND AIKEN, A. Data-Driven Equivalence Checking. *ACM SIGPLAN Notices 48*, 10 (Dec 2013), 391–406.

[54] SHARMA, R., SCHKUFZA, E., CHURCHILL, B., AND AIKEN, A. Conditionally correct superoptimization. *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2015* (2015).

[55] SHI, Y., CASEY, K., ERTL, M. A., AND GREGG, D. Virtual machine showdown. *ACM Transactions on Architecture and Code Optimization 4*, 4 (2008), 1–36.

[56] SIDIROGLOU-DOUSKOS, S., MISAILOVIC, S., HOFFMANN, H., AND RINARD, M. Managing performance vs. accuracy trade-offs with loop perforation. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11* (2011).

[57] SITTHI-AMORN, P., MODLY, N., WEIMER, W., AND LAWRENCE, J. Genetic programming for shader simplification. *Proceedings of the 2011 SIGGRAPH Asia Conference on - SA '11* (2011).

[58] SOLAR-LEZAMA, A., JONES, C. G., AND BODIK, R. Sketching Concurrent Data Structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), PLDI '08, Association for Computing Machinery, p. 136–148.

[59] SOLAR-LEZAMA, A., TANCAU, L., BODIK, R., SESHIA, S., AND SARASWAT, V. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2006), ASPLOS XII, Association for Computing Machinery, p. 404–415.

[60] TATE, R., STEPP, M., TATLOCK, Z., AND LERNER, S. Equality saturation. *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '09* (2008).

[61] TURCHIN, V. F. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems 8*, 3 (1986), 292–325.

[62] TURING, A. Checking a large routine. *The early British computer conferences* (1989), 70–72.

[63] WILLSEY, M., NANDI, C., WANG, Y. R., FLATT, O., TATLOCK, Z., AND PANCHEKHA, P. egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages 5*, POPL (Jan 2021), 1–29.

[64] WITTGENSTEIN, L. Tractatus Logico-Philosophicus. *London: Routledge, 1981* (1922).

[65] ZAKAI, A. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (New York, NY, USA, 2011), OOPSLA '11, Association for Computing Machinery, p. 301–312.

[66] ZAKAI, A., ET AL. Binaryen. https://github.com/WebAssembly/binaryen/ [Online; accessed Mar 12, 2021].