DTU

21/02 2024

Design and Implementation of a WebAssembly Compiler Back-End

for the High-Level Programming Language Hygge

Troels Lund (s161791)

Master Thesis

Supervisors
Alceste Scalas
Radoslaw Jan Rowicki

Agenda

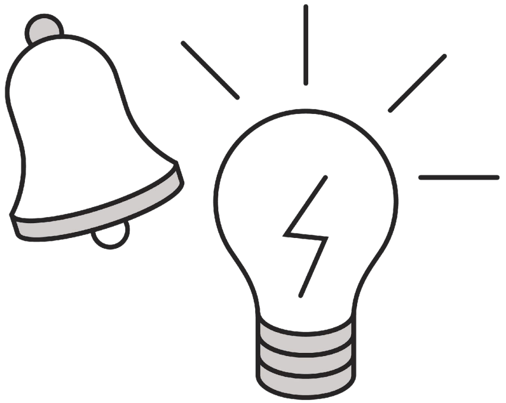
1. **Problem statement**
2. **Why** – Motivation and background
3. **What** – What was achieved
4. **How** – Design and implementation
5. **Evaluation**

Problem statement

1. How can high-level programming language features of Hygge be **synthesized** to the low-level constructs found in the target language of WebAssembly (Wasm)?

2. Are there any specific **limitations** or **challenges** in the Hygge-to-Wasm compilation process, and what are the potential solutions or workarounds?

3. How can the WebAssembly code be **optimised**, and how does the optimised code compare to the non-optimised version?



Why?
Motivation and background

Motivation

- WebAssembly shows **great potential** as a technology, consequently gaining momentum as a widely adopted **compilation target**.

- WebAssembly is a **stack-based** virtual ISA.
- With a secondary focus on **teaching-related aspects**.

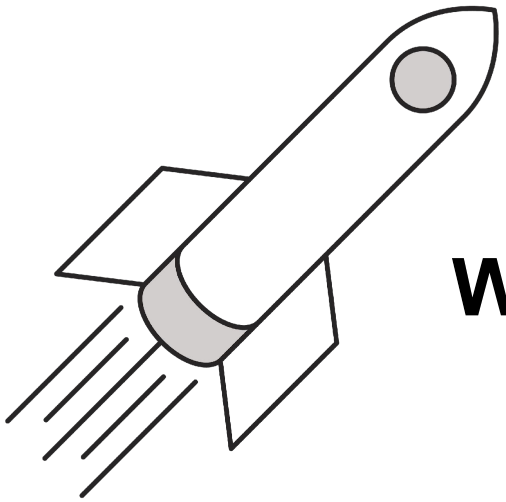
Related work

Inspiration has been drawn from WebAssembly **compiler toolchains** and **literature**.

Binaryen

emscripten

AS



What was achieved?

Deliverables of the project

A compiler with a **back-end targeting WebAssembly**, named **HyggeWasm.**

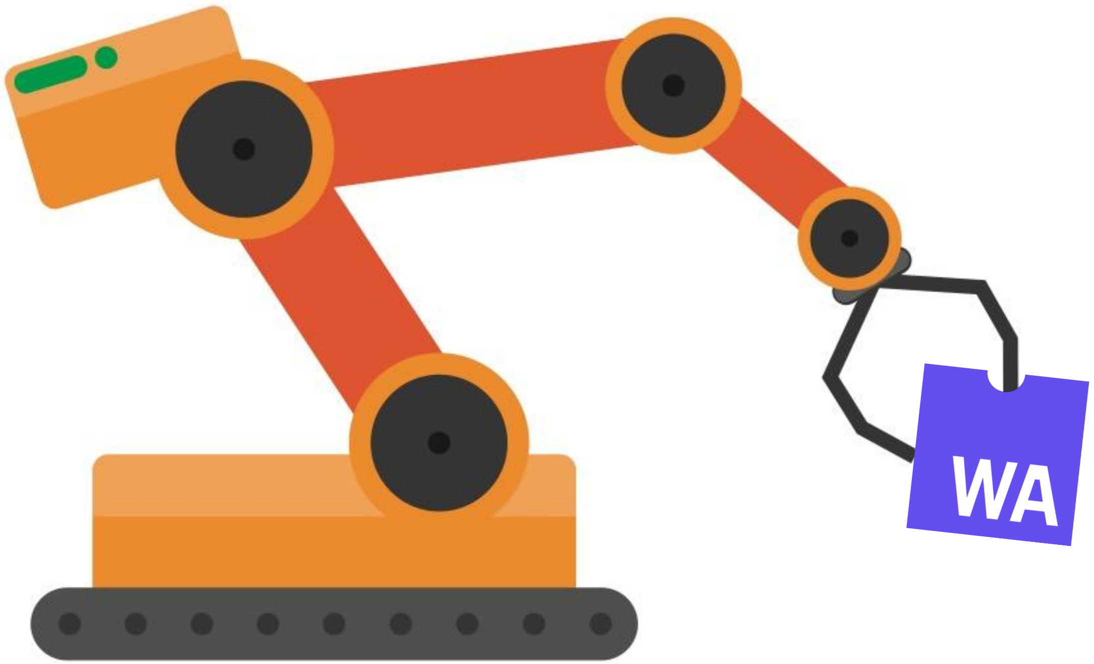
1. An **intermediate representation** of the WebAssembly (WAT) module and an algorithm for **translation into the textual format**.
2. A **runtime** for handling **I/O** and **memory allocation** in both C# and Typescript.
3. A **web application** that makes it easy **to load, run and debug programs**.

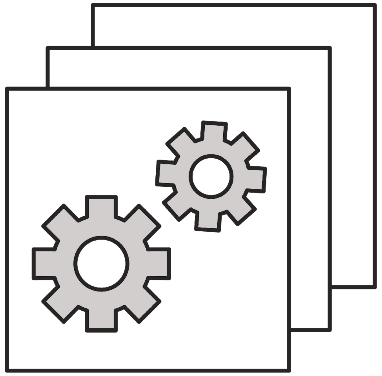
Language feature highlights

- **Functions**
  - Functions as first-class citizens
  - Recursive functions
  - Anonymous functions
  - Closures (with shared mutable variable)
- **Control flow**
  - Pattern matching
  - If-then-else,
  - Loops (while-loop, for-loop and do-while-loop)
- **Data structures**
  - Structs
  - Arrays
  - Discriminated union types
- **Logical operators with short-circuit evaluation**

Demonstration

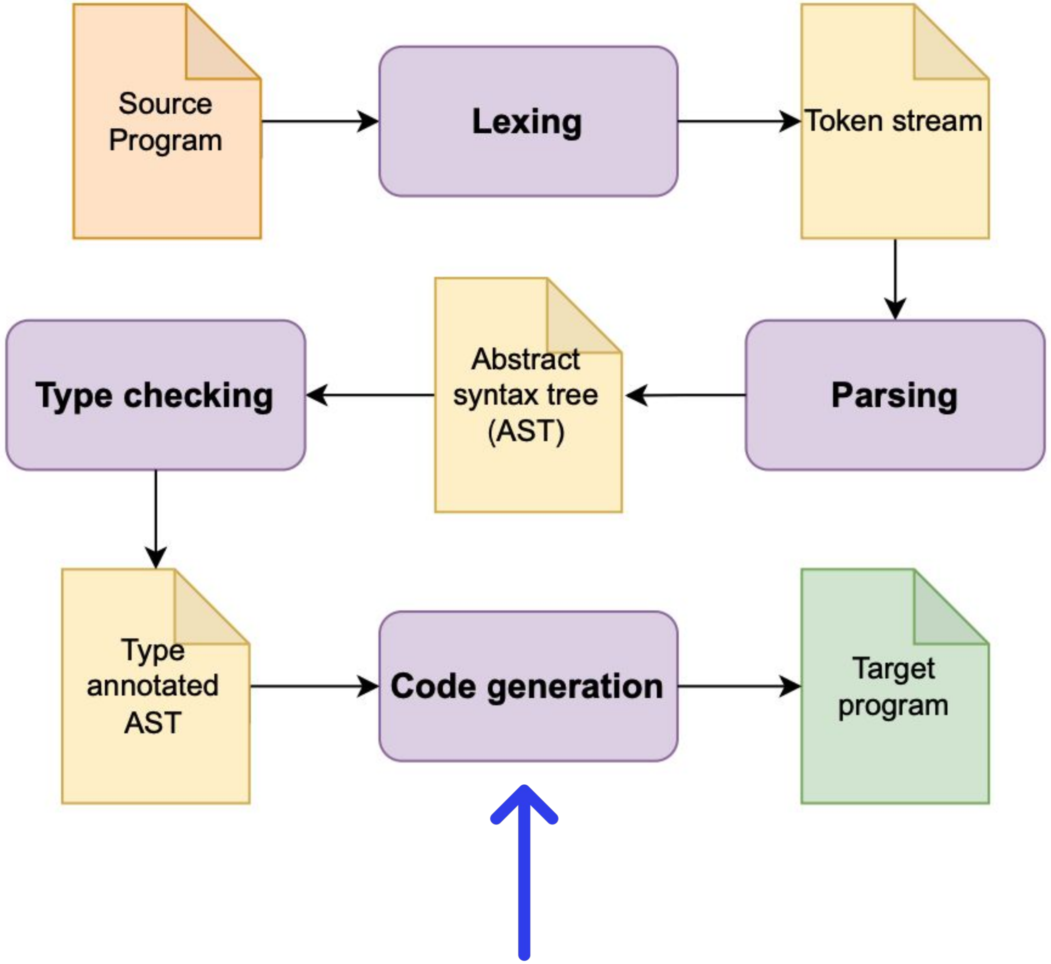
- **Input/output** of Hygge program
- **Recursive** functions
- Learning and Development tool



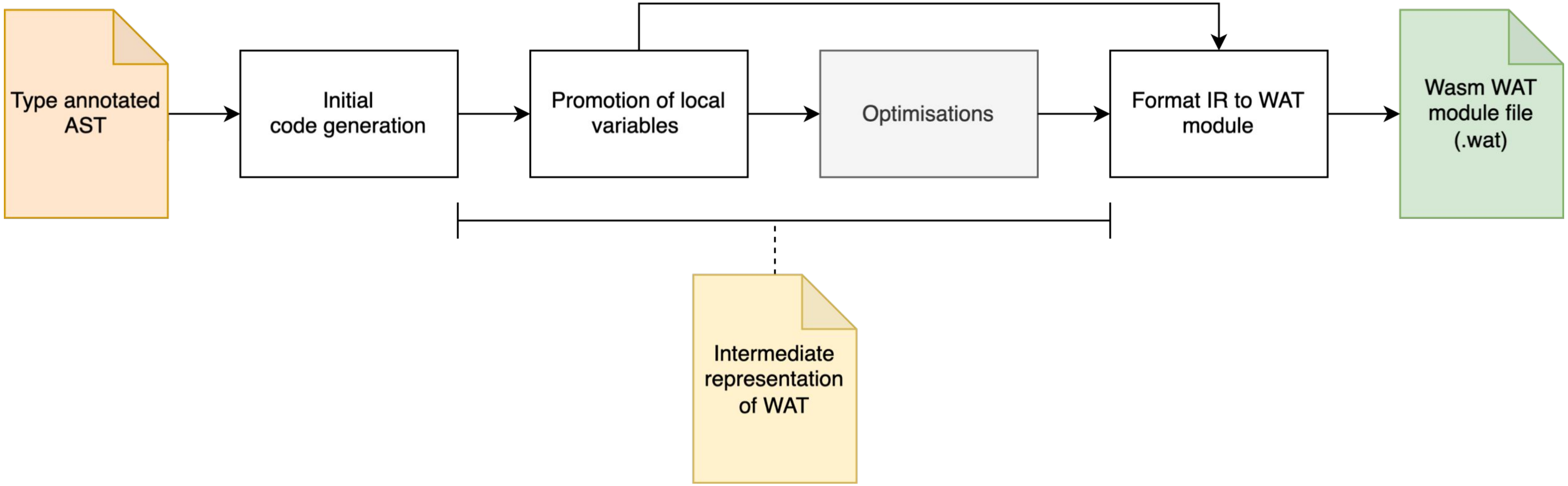


How was it achieved?

General structure - Phases of the compiler



General structure - Phases of code generation



- **Initial code generation** produces a **first draft** of the code.
- **Promotion of variables** in the global scope of the Hygge program.
- Optionally **optimisations** are applied.
- Targets the **WebAssembly text format (WAT)**.

✅ **Valid Wasm program!**

General structure - Intermediate representation defined by WGF

```
Module()
    .AddGlobal(("result", (I32, Mutable), (I32Const 0, "initialize to 0")))
    .AddCode(
        [ (GlobalSet(
                Named("result"),
                [ (I32Add(
                    [ (I32Const 5, "push 5 to stack")
                      (I32Const 6, "push 6 to stack") ]),
                "add 5 and 6") ]
              ),
              "store result in global variable") ]
    )
```

- Designed for easy **manipulation** of the symbolic code.
- Designed to be used in the **optimisation** stage.
- Allows for **comments** associated with one or multiple instructions.

* WGF (WAT Generation Framework)

Key challenges

- Implementing non-trivial language features:
  - Functions as first-class citizens
  - Recursive functions
  - Closures (with mutable shared variables)
  - Pattern matching
  - And more…

- Enabling **Input/Ouput** of the compiled Hygge programs
- **Memory** allocation and management
- Assembler support

General design challenges

Code generation strategies (Operation modes)

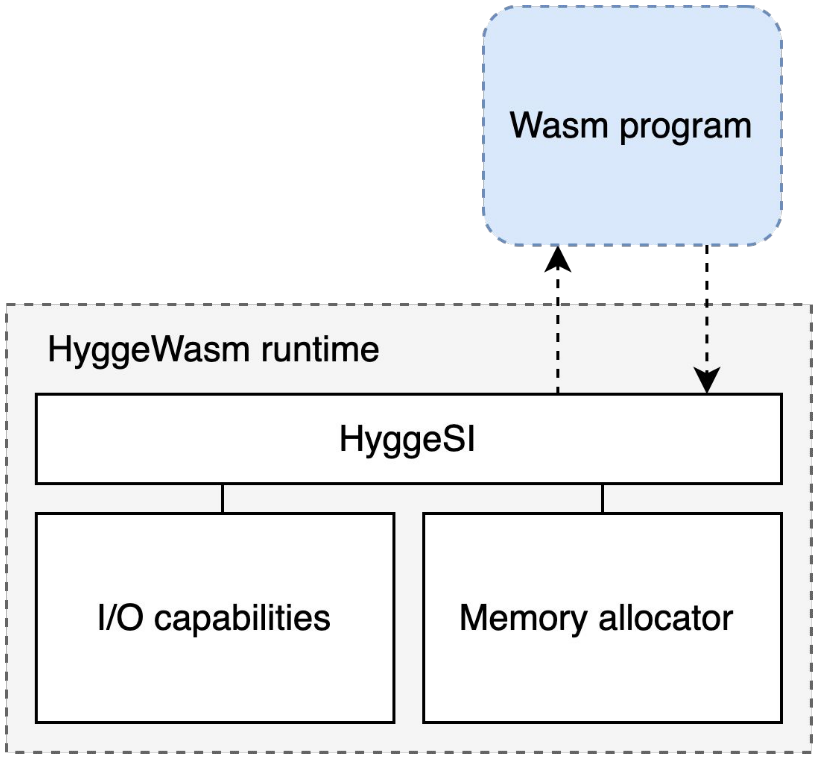
- **System Interfaces** Enabeling **Input/Ouput** of the compiled Hygge programs
  - HyggeSI (Hygge System Interface)
  - WASI (Standard)
- **Memory strategies** **Memory** allocation and mangement
  - Internal
  - External
  - Heap
- **Writing Styles** Assembler support
  - Linear
  - Folded

Operation modes influence the strategy used for generating code.

System interfaces

- **WASI** (WebAssembly System Interface, **standard**)
  - Read integer
  - Write string

- **HyggeSI** (Hygge System Interface)
  - Allocate memory block
  - Read integer
  - Read floating point
  - Write integer
  - Write floating point
  - Write string



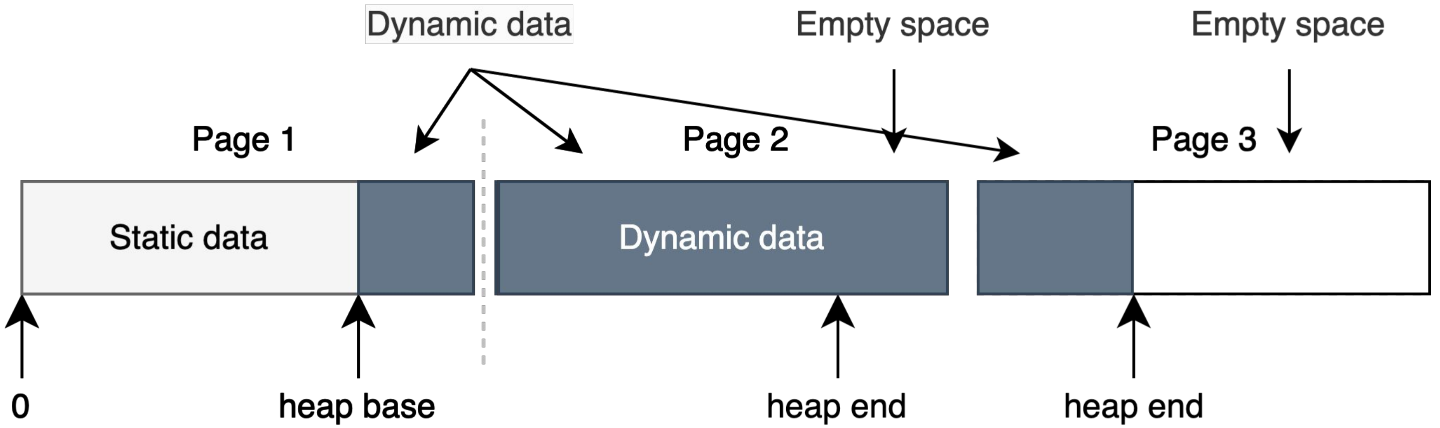
Code generation strategies (Operation modes)

- **System Interfaces**
  - HyggeSI (Hygge System interface)
  - WASI (standard)
- **Memory strategies**
  - Internal
  - External
  - Heap
- **Writing Styles**
  - Linear
  - Folded

Operation modes influence the strategy used for generating code.

Memory strategies (linear memory)

- The **external** and **internal** modes operates on **linear memory.**
- Space for **static data** is allocated during **compile time** and is placed first in linear memory**.**
- The **Bump allocation** algorithm is used to allocate memory blocks.
- In **external** mode, Bump allocation is implemented by the host system and in **internal** mode memory management is **embedded into the generated code.**
- The **accessible memory space** can be **grown** during **runtime** in both modes.



Memory strategies (heap)

- The **heap** mode uses the **WasmGC** extension, this **enables garbage collection.**

- Different **memory model**

- New **type declarations**

- New **instructions**

Instruction	Description
`struct.new`	Create a new struct
`struct.get`	Access a field in a struct
`struct.set`	Set a field in a struct
`array.new`	Create a new array

```
(type $s|i-i32|a-f32|b-i32 (;0;) (struct
        (field $i (mut i32))
        (field $a (mut f32))
        (field $b (mut i32))))

(global $var_s (;2;) (mut
        (ref null $s|i-i32|a-f32|b-i32))
        (ref.null $s|i-i32|a-f32|b-i32))
```

Code generation strategies (Operation modes)

- **System Interfaces**
  - HyggeSI (Hygge System Interface)
  - WASI (standard)
- **Memory strategies**
  - Internal
  - External
  - Heap
- **Writing Styles**
  - Linear
  - Folded

Operation modes influence the strategy used for generating code.

Writing styles

The **writing style** changes the **shape of the code** significantly by using another **syntax**.
The **writing style influence assembler support.**

```
global.get $var_a
global.get $var_b
i32.add
global.set $var_c
```

(a) Linear writing style

```
(global.set $var_c
    (i32.add
        (global.get $var_a)
        (global.get $var_b)
    )
)
```

(b) Folded writing style

Evaluation

Testing - methodology

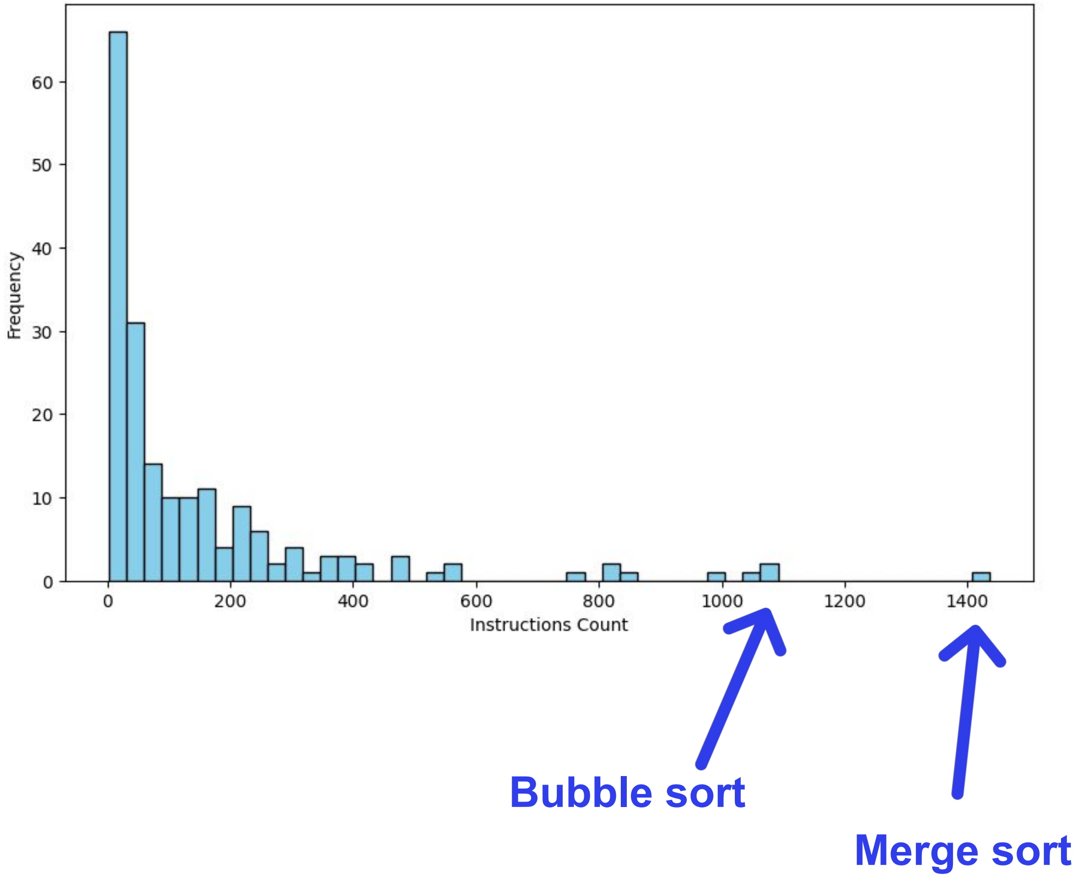
- Testing has followed a **test driven development (TDD)** methodology.
- Hygge programs are written to **test functionality**.

```
// creating a struct with the field 'f' with the value 3
let s: struct {f: int} = struct {f = 1 + 2};
s.f <- 5; // assign a new value to field 'f'
assert(s.f = 5) // assert the value of field 'f' is now 5.
```

Assert expressions check condition of the code, if the condition is **false** a **trap** is triggered, and the **program terminates**.

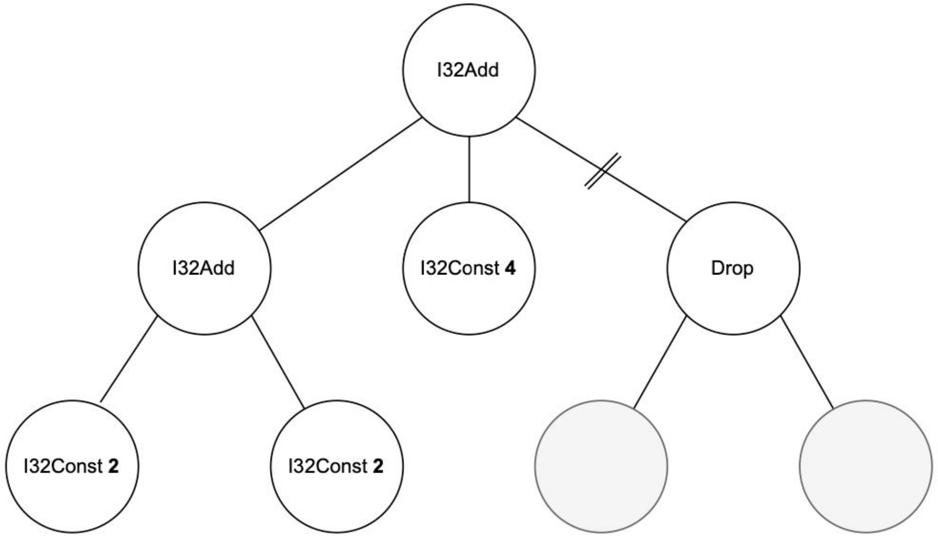
Testing

- All language features have been tested.
- Most tests are written to **target a specific feature**.
- Test suite
  - **212** test programs
  - **1.040** of them target code generation
  - The entire test suite has **1.203** distinct tests



Optimisations (IR)

- Local variable read and write optimisation
- Dead-code elimination
- Constant folding
  - Branch-level tree shaking



Optimisations are performed on the **symbolic code** (**IR)** and is implemented as **peephole optimisations**.

Evaluation of optimisations

The **dataset** was created by compiling all programs with and without optimisations applied and **count the number of executable instructions** in each program.

A mean reduction of **14.62%** measured by instruction count across all tests.

Constant folding stands out as the most impactful optimisation, contributing a **12.36%** reduction.

Future work

- Full support for WasmGC
- Improve dead-code elimination
- Tail recursion optimisation

Key takeaways

- **Multiple non-trivial language features implemented**:
  - Functions as first-class citizens
  - Recursive functions
  - Closures with mutable shared variables
  - Pattern matching

- **Multiple code generation strategies**
  - Three **memory management** strategies and one of the few languages that utilize **WasmGC** at the moment.
  - Supports two **system interfaces** to enable **I/O**.
  - Two **writing styles**.

- **Robust testing of all features** ✅
- The thesis addresses all the questions in the problem statement.

DTU

Hygge - Insertion sort

```
let insertionSort: (array{int}) -> unit = fun(arr: array{int}) -> {
    let len: int = arrayLength(arr);
    let mutable i: int = 0;

    for (i <- 1; i < len; ++i) {
        let key: int = arrayElem(arr, i);
        let mutable j: int = i - 1;

        // Move elements of arr[0..i-1] that are greater than key to one
    position ahead of their current position
        while (j >= 0 && (arrayElem(arr, j) > key)) do {
            arrayElem(arr, j + 1) <- arrayElem(arr, j);
            j <- j - 1
        };
        arrayElem(arr, j + 1) <- key
    }
};
```

Hygge - Higher-order functions

```
fun doOperation(x: int, y: int, operation: (int, int) -> int): int = {
    operation(x, y)
};

fun add (a: int, b: int): int = {
    a + b
};

assert(doOperation(5, 3, add) = 8)        // Output: 8
```

Hygge – Fibonacci (recursive functions)

```
// declare n as an integer and assign it the value 16
let n: int = 16;
// function to calculate the nth term of the Fibonacci sequence
fun fibRec(n: int): int = {
    if (n <= 1) then {
        n
    }
    else {
        fibRec(n - 1) + fibRec(n - 2)
    }
};
// print the result
println("The 16th term of the Fibonacci sequence is:");
println(fibRec(n))
```

Hygge – Fibonacci (imperative)

```
// Number of terms of the Fibonacci sequence to print (minimum 2).
let n: int = 16;

let mutable t0: int = 0;    // First term in the Fibonacci sequence
let mutable t1: int = 1;    // Second term in the Fibonacci sequence
println(t0);
println(t1);

let mutable i: int = 2;     // Counter: how many terms we printed
let mutable next: int = 0; // Next term in the Fibonacci sequence

while (i < n) do {
    next <- t0 + t1;
    println(next);
    t0 <- t1;
    t1 <- next;
    i <- i + 1
}
```

Hygge – Simple closure

- *i* is captured

```
fun f(): () -> int = {
    let mutable i: int = 0;
    fun () -> {
        i++
    }
};

let f0: () -> int = f();
assert(f0() = 0);
assert(f0() = 1)
```

Hygge - FizzBuzz

```
let mutable i: int = 0;
let mutable y: int = readInt();

for ((); (i < y); {i <- i + 1}) {

    let by3: bool = ((i % 3) = 0);
    let by5: bool = ((i % 5) = 0);

    if (by3) then {
        if (by5) then {
            println("FizzBuzz")
        }
        else {
            println("Fizz")
        }
    }
    else {
        if (by5) then {
            println("Buzz")
        }
        else {
            println(i)
        }
    }
}
```

Can run only using WASI features

Variable promotion

- **Variables** in the global scope of the Hygge program are promoted.
- The declaration of **local variables are removed** from the function level and **added to the global section of module**.
- All instructions operating on a variable is substituted with equivalent ones for global variables.

```
let x: int = 40; // <-- Variable x becomes global value
let f: () -> int = fun() -> 40 + x; // <-- Accsess value of x

assert(f() = 80)
```

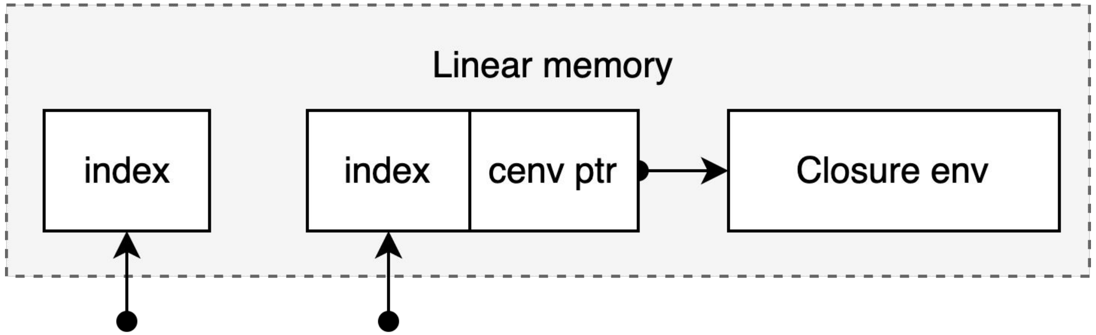
Closures

- Function **signatures are rewritten** to include *cenv.*

- **Variable storage** keeps track of where in the closure **captured variables are stored**.
  - This is the *offset* storage type.

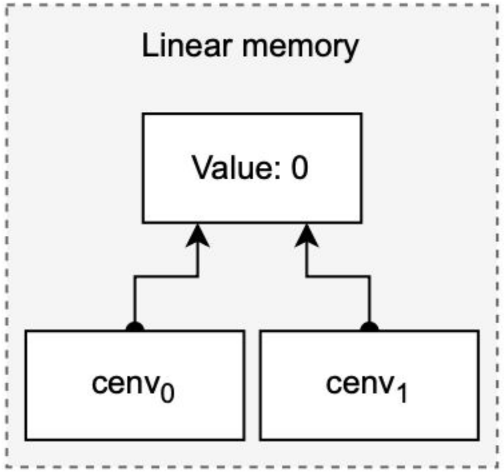
- **Mutable variables** are **encapsulated in a struct** so that the reference can be shared.
- **Access** to the mutable variable is **rewritten to a field selection**.

- A **function instance** consists of the pair of **index** and **closure environment**.



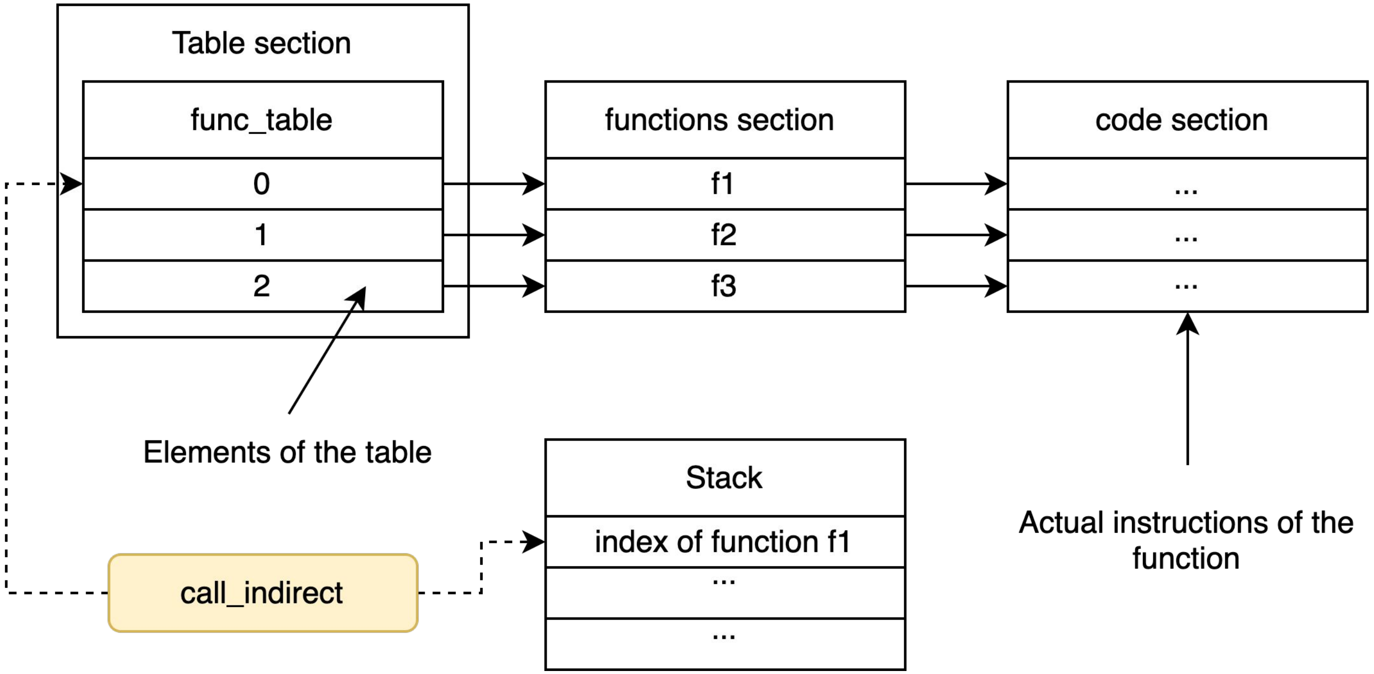
Hygge – Closure with shared mutable variable(s)

```
type Counters = struct {increment: () -> int; decrement: () -> int};

// Return a structure with two functions that share a counter.
// The 'count' is initialized to 0.
// The 'count' can be either incremented or decremented
fun makeCounters(): Counters = {
    let mutable count: int = 0; // The mutable variable 'count'

    // The lambda terms below capture 'count' twice
    struct { increment = fun () -> { count <- count + 1 };
             decrement = fun () -> { count <- count - 1 } } : Counters
};
// create a counter
let c1: Counters = makeCounters();
assert(c1.increment() = 1);
assert(c1.increment() = 2);
// create a counter more
let c2: Counters = makeCounters();
assert(c2.increment() = 1); // Output: 1 (independent of c1)
assert(c2.increment() = 2);
assert(c2.decrement() = 1);
assert(c2.decrement() = 0)
```



Indirect calls in WebAssembly (Functions as first-class citizens)

- A function is reduced to a **index** that can be **stored in memory**.
- Memory address is an **offset** (*i32*) in *linear memory* that can be parsed around.
- An **indirect call** take a function **type defination** and an **index**.



Anonymous functions

- An anonymous function is named based on the scope it is placed in.

```
fun sum(a: int): (int) -> (int) -> int = { // <-- Named "$fun_sum"
    fun (b: int) -> {          // <-- Named "$fun_sum/anonymous"
        fun (c: int) -> {      // <-- Named "$fun_sum/anonymous/anonymous"
            a + b + c
        }
    }
}
```

Loops

- Loops use **control structures** and branch instructions.
- This is the skeleton of a while-loop:

```
(block $loop_exit
  (loop $loop_begin
    (br_if $loop_exit ;; if false break
      (i32.eqz ;; evaluate loop condition
        ;; the condition itself
      )
    )
    ;; the loop body
    (br $loop_begin) ;; jump to the beginning of the loop
  )
)
```

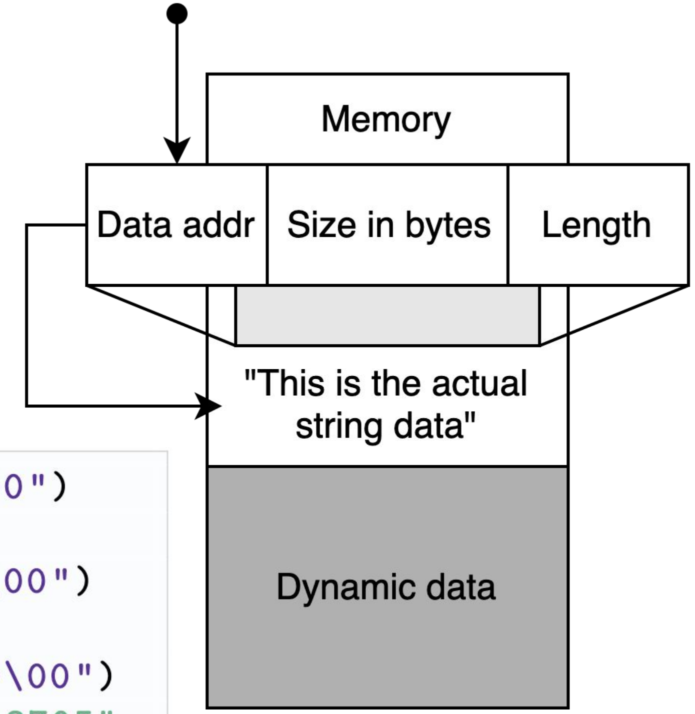
Stack management

- The **last element** in a sequense of expressions is the **return value.**
- All other expressions that **leave a value** on the stack are **discarded**.
- Must be done to conform with **result types** of **control structures.**

```
fun f(arr: array {int}, i: int): array {int} = {
    if (i < arrayLength(arr)) then {   // <-- Result type of (i32)
        arrayElem(arr, i) <- i + 1;    // <-- Push i32 value
        f(arr, i + 1)                  // <-- Function will push address (i32)
    }
    else {
        arr                            // <-- Push i32 value
    }
};
```

Strings

- Strings are placed in memory at **module instantiation** with a data string of 8-bit hex segments.




```
(data (i32.const 0) "\0c\00\00\00\12\00\00\00\12\00\00\00")
(data (i32.const 12) "hygge println test")
(data (i32.const 30) "\2a\00\00\00\10\00\00\00\10\00\00\00")
(data (i32.const 42) "hygge print test")
(data (i32.const 101) "\71\00\00\00\03\00\00\00\01\00\00\00")
(data (i32.const 113) "0x2705") ;; Unicode Character "U+2705"
```

Combining modules

```
| StringLength e ->
    let m' = doCodegen env e m

    m'
        .ResetAccCode()
        .AddCode([ (I32Load_(None, Some(8), m'.GetAccCode()), "load string
    length") ])
```

Union type constructor and pattern matching

```
type t = union {
     Some: int;
     None: unit
};

// union-type constructor
let x: t = Some{42};
let n: t = None{()};

match x with {
     Some{v} -> println(v);
     None{_} -> println("None")
}
```

Arrays

```
fun f(arr: array {int}, i: int): array {int} = { // recursive function
    if (i < arrayLength(arr)) then { // read length of array as part of condition
        arrayElem(arr, i) <- i + 1; // assign value to array element
        f(arr, i + 1) // recursive function call
    }
    else {
        arr // return modified array
    }
};

f(arr, x / 2); // modified array returned

x <- 0; // reset x

do { // do-while to print array data
    println(arrayElem(arr, x)); // read array element
    x <- x + 1 // increment
} while (x < arrayLength(arr)); // read length of array as part of condition

println("--------------------------");

let sliced: array {int} = arraySlice(arr, x / 2, arrayLength(arr)); // slice array in half

x <- 0;

do { // do-while to print array data
    println(arrayElem(sliced, x)); // read array element
    x <- x + 1 // increment
} while (x < arrayLength(sliced)); // read length of array as part of condition

println("--------------------------");

type OptionalArray = union { // read length of array as part of condition
    Some: array {int};
```

Constant folding

```
(func $_start (;0;)
;; execution start here:
(if
    (i32.eqz ;; invert assertion
      (i32.eq ;; equality check
        (i32.add
          (i32.add ;; <-- become 'i32.const 9'
            (i32.const 4) ;; push 4 on stack
            (i32.const 5) ;; push 5 on stack
          )
          (i32.const 3) ;; push 3 on stack
        )
        (i32.const 12) ;; push 12 on stack
      )
    )
  (then
    (global.set $exit_code ;; set exit code
      (i32.const 42) ;; error exit code push to stack
    )
    (unreachable) ;; exit program
  )
)
;; if execution reaches here, the program is successful
)
```

Figure 6.8: Before constant fold

```
(func $_start (;0;)
;; execution start here:
(if
  (i32.const 0) ;; condition
  (then
    (global.set $exit_code ;; set exit code
      (i32.const 42) ;; error exit code push to stack
    )
    (unreachable) ;; exit program
  )
)
;; if execution reaches here, the program is successful
)
```

Pattern matching in WAT

```
(block $match_end (result i32) ;; <-- result type of the block
;; case for id: $1, label: Some
(if
    (i32.eq ;; check if index is equal to target
      (i32.load ;; load label
        (global.get $var_x) ;; get local var: var_x, have been promoted
      )
      (i32.const 1) ;; put label id 1 on stack
    )
  (then
    (global.set $match_var_x ;; set local var, have been promoted
      (i32.load offset=4
        (global.get $var_x) ;; get local var: var_x, have been promoted
      )
    )
    (global.set $var_i ;; set local var, have been promoted
      (i32.add
        (global.get $match_var_x) ;; get local var: match_var_x, have been promoted
        (i32.const 1) ;; push 1 on stack
      )
```

Language features

- **Arithmetic operators** (-, +, %, /, sqrt, max and min)
- **Logical operators** (or, and, xor - && and || (short-circuit evaluation))
- **Relational operators** (=, <, >, <= and >=)
- **Variables** (++var, var++, --var, var--, +=, -=, *=, /= and %=, var <- value)
- **Control flow** (if-then-else, while-loop, for-loop and do-while-loop)
- **Data structures**
  - **Structs** (Constructor, field access, assign field value)
  - **Tuples** (Constructor, field access, assign field value)
  - **Arrays** (Constructor, element access, assign element value, slice array)
  - **Discriminated union types**
- **Functions** (first-class citizens, recursive functions, anonymous functions and closures)
- **Pattern matching**
- **I/O** (Read integer or float, Write integer, float, and string values to output stream)