Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Department of Computer Science

Software Engineering Orientation

Bachelor thesis

2024

Functional language compiler to WebAssembly

Technical documentation

Noah Godel

Supervisors:	Jacques Supcik
	Serge Ayer
Experts:	Baptiste Wicht
	Valentin Bourqui

Fribourg, 19 June 2024

Version 0.2

Hes·so

Table of versions

Version	Date	Modifications
0.2	19.6.2024	Analysis section
0.1	12.6.2024	First version - introduction and chapter titles

Contents

TODOs

Section 1
Introduction

This report documents the development of a functional language compiler to WebAssembly (Wasm). The project was conducted as part of the Bachelor's thesis at the Haute école d'ingénierie et d'architecture de Fribourg (HEIA-FR). The goal of the project was to design and implement a compiler for a functional language that targets Wasm. The project was supervised by Dr. Jacques Supcik and Dr. Serge Ayer, with Dr. Baptiste Wicht and Valentin Bourqui as experts. For further details, please refer to the requirement specification document [4]. The project repository can be found at the following URL.

https://gitlab.forge.hefr.ch/noah.godel/24-tb-wasm-compiler

1.1. Context

The functional programming paradigm offers advantages for certain types of problems like data transformations, parallel processing, and mathematical computations. However, it has limitations, and many use cases are better suited for imperative or object-oriented programming. Ideally, developers should be able to leverage the strengths of different paradigms within the same codebase. Unfortunately, integrating functional languages into existing codebases written in other languages can be challenging.

Wasm is a bytecode format designed to execute code at near-native speeds across different environments like web browsers and Wasm runtimes. By developing a compiler for a functional language, or in the context of this project, a subset of an existing one, that compiles to Wasm, we can combine functional programming benefits with Wasm's performance and portability. This enables seamless integration of high-performance functional code into codebases of different languages, allowing developers to utilize functional programming strengths for specific components.

The project aims to demonstrate embedding the new functional language compiled to Wasm into existing codebases, showcasing interoperability and the potential for combining paradigms within the same project. For more details on the context, refer to the requirements specification document [4].

Figure 1 illustrates the concept of embedding a Wasm module into a codebase.
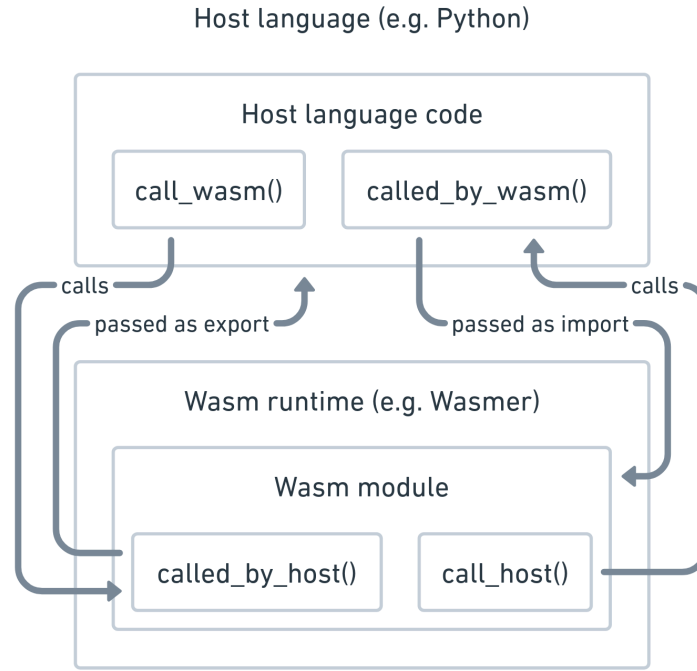
Figure 1: Illustration of embedding a Wasm module into a codebase.

1.2. Objectives

Upon completion of the project, the following key objectives will be achieved:

- **Functional Programming Language Specification**: Define a functional programming language that is a subset of an existing language, tailored for efficient Wasm compilation and seamless embedding into other codebases. A subset of the standard library supporting the language features will also be defined.
- **Functioning Compiler**: Develop a fully operational compiler capable of translating the defined functional language into efficient Wasm bytecode for high-performance execution across environments. The compiled code should interoperate with other languages it is embedded into.
- **Language Documentation**: Provide documentation detailing the usage and development of the new language, including examples, references, and demonstrations of embedding into different language codebases to facilitate learning and adoption.

While not production-ready after 7 weeks, the project will serve as a proof of concept and foundation for potential further development by delivering the defined language, compiler, documentation, and embedding examples. Refer to the requirements specification for more details on the objectives [4].

1.3. Document structure

This document is structured as follows.

- *Introduction*: Provides an overview of the project and its context.

- *Analysis*: Describes the context, objectives, and requirements of the project.
- *Design*: Details the design of the functional language, compiler, and standard library.
- *Implementation*: Explains the implementation of the compiler and standard library.
- *Evaluation*: Discusses the evaluation of the compiler and standard library.
- *Conclusion*: Summarizes the project, highlights achievements, and outlines future work.

! TODO !
maybe add a chapter about planning and sprints

Section 2
Analysis

This section presents the constraints, the exploration of different technologies and features and the technological choices.

2.1. Choice of language for the subset

In this project, the choice of the language subset is crucial. The language should be expressive enough to demonstrate the functional programming paradigm's benefits while being simple enough to implement within the project's timeframe. The language should also be a subset of an existing language to so that the task of having to specify the language's syntax and semantics is simplified.

The following languages were considered for the project.

2.1.1. OCaml

OCaml is a general-purpose, multi-paradigm programming language that extends the ML language with object-oriented features. It has a strong type system, automatic memory management, and supports functional, imperative, and object-oriented programming styles. OCaml is widely used in academia and industry, particularly in areas such as theorem proving, compiler development, and systems programming.

Advantages:
- Strong static type system can facilitate efficient compilation and optimization.
- Since OCaml is often used in compiler development, its compiler is well documented and can serve as a reference for the project.
- Already supports <u>Wasm</u> compilation, which can serve as a reference for the project.

Disadvantages:
- Multi-paradigm nature and complex syntax may complicate the task of creating a purely functional subset.
- OCaml is a vast language with many features, which may make it challenging to define a subset that is both expressive and manageable, given the author's limited experience with the language.

2.1.2. F#

F# is a multi-paradigm programming language that encompasses functional, imperative, and object-oriented styles. It is a part of the .NET ecosystem and can be seamlessly integrated with other .NET languages such as C# and Visual Basic. F# is particularly

well-suited for data-oriented programming tasks, parallel programming, and domain-specific language development.

Advantages:
- Seamless integration with the .NET ecosystem and interoperability with other .NET languages.
- Already supports <u>Wasm</u> compilation through Bolero (which uses Blazor), providing a reference for the project.

Disadvantages:
- Limited adoption and smaller community compared to more popular languages like C#.
- Multi-paradigm nature may complicate the task of creating a purely functional subset.
- The author is not familiar with F# and would need to learn the language from scratch.

2.1.3. The Lisp languages (Common Lisp, Clojure)

Lisp (List Processing) is a family of programming languages with a long history and a distinctive syntax based on parentheses and lists. Common Lisp and Clojure are two prominent dialects of Lisp.

Common Lisp

Common Lisp is a multi-paradigm language that supports functional, imperative, and object-oriented programming styles. It is used in artificial intelligence, computer algebra, and symbolic computation applications.

Advantages:
- Very simple and consistent syntax, which makes it easy to define a subset.
- Established language with a rich ecosystem of libraries and tools.

Disadvantages:
- No built-in support for <u>Wasm</u> compilation, which means there is no reference implementation for the project.
- The author is not familiar with Common Lisp and would need to learn the language from scratch.

Clojure

Clojure is a modern Lisp dialect that runs on the Java Virtual Machine (JVM) and emphasizes immutable data structures and functional programming. It is designed for concurrent and parallel programming, and is often used in web development and data analysis applications.

Advantages:

- Functional programming paradigm aligned with the project's goals.
- Runs on the JVM, which has existing tooling and libraries for Wasm compilation.

Disadvantages:
- No *direct* support for <u>Wasm</u> compilation, which means there is no reference implementation for the project.
- The author has limited experience with Clojure and defining a subset may be challenging.

2.1.4. The BEAM languages (Erlang, Elixir)

The Beam languages, Elixir and Erlang, are functional programming languages that run on the Erlang Virtual Machine (BEAM). They are designed for building scalable, fault-tolerant, and distributed systems.

Erlang

Erlang is a general-purpose, concurrent programming language with built-in support for distributed computing. It is widely used in telecommunications, banking, and e-commerce systems that require high availability and fault tolerance.

Advantages:
- Functional programming paradigm aligned with the project's goals.
- There are alternative compilers for BEAM languages that target <u>Wasm</u>, which can serve as a reference for the project.

Disadvantages:
- The author has limited experience with Erlang, which may complicate the task of defining a subset.

Elixir

Elixir is a more recent functional language that builds upon the strengths of Erlang's VM and ecosystem. It aims to provide a more modern and productive syntax while maintaining the robustness and concurrency features of Erlang.

Advantages:
- Functional programming paradigm aligned with the project's goals.
- Elixir has a more modern syntax and tooling compared to Erlang.
- As with Erlang, there are alternative compilers for BEAM languages that target <u>Wasm</u>, which can serve as a reference for the project.

Disadvantages:
- The author has limited experience with Elixir, which may complicate the task of defining a subset.

2.1.5. Haskell

Haskell is a purely functional programming language with a strong static type system and lazy evaluation. It is known for its elegance, conciseness, and expressive type system, which facilitates safe and modular code development.

Haskell's functional paradigm and powerful abstraction mechanisms make it well-suited for a wide range of applications, including data analysis, concurrent and parallel programming, domain-specific language development, and cryptography.

Advantages:
- Purely functional programming paradigm, aligning perfectly with the project's goals.
- Advanced type system can facilitate efficient compilation and optimization.
- Existing tools and libraries for Wasm compilation, such as the Glasgow Haskell Compiler (GHC) and its support for various intermediate representations.
- Author's familiarity with the language can facilitate implementation and understanding of language intricacies.

Disadvantages:
- Lazy evaluation may introduce complexities in the compilation process and performance considerations.
- Haskell's advanced type system may require additional effort to define a subset that is both expressive and manageable within the project's timeframe.

Considering the project's goals of creating a functional language subset tailored for efficient compilation to WebAssembly (Wasm), Haskell stands out as the most suitable choice. Its purely functional nature, advanced type system, existing tooling for Wasm compilation, and the author's familiarity with the language make it an ideal foundation for this project. Since the project has a limited timeframe of 7 weeks, the choice of a language subset that the author is most comfortable with, is crucial.

Since Haskell is a purely functional language, defining a subset that is both expressive and manageable within the project's timeframe should be feasible. Additionally, the motivation behind the project is to be able to leverage the strengths of functional programming within existing codebases, and Haskell's functional paradigm aligns perfectly with this goal.

While other languages like OCaml, F#, Lisp dialects, and the Beam languages have their strengths, their multi-paradigm nature or limited direct support for Wasm compilation could introduce additional complexities or hinder the efficient realization of the project's objectives.

2.2. Wasm extensions

Wasm is a stack-based virtual machine designed to execute code at near-native speeds across different environments. It is used in web browsers, server-side applications, and other environments where performance and portability are essential. Wasm bytecode is

generated from high-level languages and can be executed on any platform that supports the Wasm runtime.

In its current form, Wasm provides a set of core features that are sufficient for executing code efficiently. However, there are several extensions and proposals that aim to enhance Wasm's capabilities and make it more versatile for different use cases. The following Wasm extensions were considered for the project.

2.2.1. Component model

One of the main limitations of Wasm (especially in the context of embedding it into existing codebases) is the small number of types it supports (essentially integers and floats). The component model proposal [3] aims to address this limitation by introducing a new language that allows developers to define custom types and interfaces and an ABI for interacting with Wasm modules. This extension could be beneficial for the project as it would enable more seamless integration of the functional language subset into other codebases.

Using this new language, developers can define interfaces using the .wit file format and implement these interfaces in Wasm modules. To use the generated component, bindings need to be generated in the host codebase that match the interface, its types and functions that are defined in the .wit file. This allows the host codebase to interact with the Wasm component using the defined interface.

The problem with this extension is that it is still in the proposal stage and Wasm components can only be run in a few languages (Rust, JavaScript and partially Python) using the Wasmtime runtime. This could limit the project's ability to demonstrate embedding the functional language into different codebases.

Listing 1 shows an example of an interface using the .wit file format, and Listing 2 shows an example of the implementation of the interface in Wasm.

```
package example:add;

world root {
  export add: func(x: s32, y: s32) -> s32;
}
```

Listing 1: Example of a Interface using the .wit file format.

```
(module
  (func (export "example:add/root#add") (param i32) (param i32) (result i32)
    local.get 0
    local.get 1
    i32.add
  )
)
```

Listing 2: Example of the implementation of the Interface.

2.2.2. Reference types and function references

The reference types proposal [10] aims to allow for reference types (function references or external references) to be used as values. This extension could be beneficial for the project since this extension simplifies the implementation of functions as first-class citizens.

In core Wasm, function references are only used inside function tables (necessary for indirect calls). The reference types proposal extends this to allow function references to be used as values in the functions themselves and not only as indices into the function table. It also introduces new instructions to interact with the function table to dynamically add and remove functions from it.

The proposal is still in the proposal stage, but it is supported by the Wasmer, Wasmtime and WasmEdge runtimes and practically everywhere else. This means that the project could leverage these runtimes to demonstrate the embedding of the functional language into different codebases.

The function references proposal [11] is an extension of the reference types proposal that simply enables function references to be called directly. It also makes a distinction between nullable and non-nullable function references. This extension could be beneficial for the project as it simplifies the implementation of functions as first-class citizens even further.

The function references proposal is still in the proposal stage and is less supported than the reference types proposal. It is supported by the Wasmtime and WasmEdge runtimes and in the browser.

Listing 3 shows an example of reference types and function references in Wasm.

```
(module
    (table 1 funcref)
    (type $type0 (func (result i32)))
    (type $type1 (func (param i32) (result i32)))

    (func $foo (result i32) i32.const 42)

    ;; This function calls the function referenced in the table with
    ;; the index returned by "add_func_to_tabel"
    (func $ref_types_example (result i32)
        (call_indirect 0 (type $type0) (call $add_func_to_tabel))
    )

    ;; This function adds the function reference to the table and
    ;; returns the index
    (func $add_func_to_tabel (result i32)
        (table.set 0 (ref.func $foo) (i32.const 0))
        i32.const 0
    )

    ;; This function takes a int and returns it
    (func $bar (param i32) (result i32) local.get 0)

    ;; This function takes a int and calls "call_passed_func" with
    ;; it and the function reference
    (func $func_types_example (param i32) (result i32)
        (call $call_passed_func (local.get 0) (ref.func $bar))
    )

    ;; This function takes a int and a function reference and calls
    ;; the function reference with the int
    (func $call_passed_func (param i32) (param (ref $t1)) (result i32)
        (call_ref $type1 (local.get 0) (local.get 1))
    )
)
```

Listing 3: Example of reference types and function references in Wasm.

2.2.3. Garbage collection

The garbage collection proposal [12] aims to introduce garbage collection support in Wasm. This extension could be beneficial for the project as it would simplify memory management and resource cleanup in the functional language subset. It is a quite complex proposal and is still in the proposal stage. Since the support for garbage collection in Wasm is as of now limited to the browser and node.js, this could limit the project's ability to demonstrate embedding the functional language into different codebases.

The proposal bases itself on the reference types and function references proposals and introduces new types (so-called heap types) like structs, arrays, and references to these

types. It also introduces new instructions to allocate and modify these types on the heap.

2.2.4. Tail call optimization

The tail call optimization proposal [13] aims to introduce tail call optimization support in Wasm. This extension could be beneficial for the project as it would optimize the performance of recursive functions in the functional language subset. The proposal is still in the proposal stage and is supported by the Wasmtime and WasmEdge runtimes and practically everywhere else.

Listing 4 shows an example of tail call optimization in Wasm.

```wasm
(module
    (func $factorial (param $x i64) (result i64)
        (return (call $factorial_aux (local.get $x) (i64.const 1)))
    )

    (func $factorial_aux (param $x i64) (param $acc i64) (result i64)
        (if (i64.eqz (local.get $x))
            (then (return (local.get $acc)))
            (else
                (return
                    (call $factorial_aux
                        (i64.sub (local.get $x) (i64.const 1))
                        (i64.mul (local.get $x) (local.get $acc))
                    )
                )
            )
        )
        unreachable
    )

    (func $factorial_tail (param $x i64) (result i64)
        (return_call $factorial_tail_aux (local.get $x) (i64.const 1))
    )

    (func $factorial_tail_aux (param $x i64) (param $acc i64) (result i64)
        (if (i64.eqz (local.get $x))
            (then (return (local.get $acc)))
            (else
                (return_call $factorial_tail_aux
                    (i64.sub (local.get $x) (i64.const 1))
                    (i64.mul (local.get $x) (local.get $acc))
                )
            )
        )
        unreachable
    )
    (export "factorial" (func $factorial))
    (export "factorial_tail" (func $factorial_tail))
)
```

Listing 4: Example of tail call optimization in Wasm.

Listing 5 shows a performance comparison between a factorial function with and without tail call optimization.

```
factorial(20): 2432902008176640000 in 12.41µs
factorial_tail(20): 2432902008176640000 in 1.319µs
```

Listing 5: Example of tail call optimization performance comparison.

2.3. Embedding the Wasm module into a codebase

The embedding of the Wasm module into a codebase is a crucial aspect of the project. The Wasm module should be able to interact with the host codebase seamlessly.

Since Wasm is originally designed to run in web browsers, the embedding of Wasm modules into web applications is well supported. However, embedding Wasm modules into other codebases, such as server-side applications or desktop applications, can be more challenging. To be able to interact with the Wasm module, the host codebase needs a runtime that can load and execute the Wasm module (see Figure 1). The runtime should also provide mechanisms for passing data between the host codebase and the Wasm module.

The following technologies were considered for the project.

2.3.1. Wasmer

Wasmer s a standalone Wasm runtime that supports running Wasm modules outside the browser. It provides a set of APIs for loading and executing Wasm modules, as well as mechanisms for interacting with the host codebase. Wasmer also has a registry of Wasm modules that can be used to share and distribute Wasm modules. These modules can also be deployed one the cloud using Wasmer's cloud service.

However, Wasmer does supports the least amount of Wasm proposals out of the three runtimes (see Table 1), which could limit the project's ability to demonstrate seamless embedding the functional language into different codebases.

On the other hand, Wasmer provides a large set of SDKs for different programming languages (see Table 2 [8]). Compared to the other runtimes Wasmer has the most extensive support for different programming languages. Which is important for the project since the functional language subset should be able to be embedded into different codebases.

2.3.2. Wasmtime

As with Wasmer, Wasmtime is a standalone Wasm runtime that supports running Wasm modules outside the browser. It is developed by the Bytecode Alliance, a group of companies and individuals working on WebAssembly and related technologies. To see the full list supported Wasm proposals see Table 1 [9] and for the supported programming languages see Table 2.

Wasmtime is the only runtime that supports the component model proposal, which could be beneficial for greatly simplifying the embedding of the functional language into different codebases.

2.3.3. WasmEdge

As with the other runtimes, WasmEdge is a standalone Wasm runtime that supports running Wasm modules outside the browser. It is developed by the Second State, a company that provides a platform for building and deploying Wasm applications. WasmEdge supports almost all Wasm proposals (see Table 1 [6]). It achieves this by running JavaScript code in a sandboxed environment and can so support the browser's implementation of Wasm.

However, WasmEdge supportes fewer programming languages than Wasmer and Wasmtime (see Table 2 [7]). This could limit the project's ability to demonstrate embedding the functional language into different codebases.

2.3.4. Wasm proposal compatibility and language support

Table 1 shows a summary of the compatibility of the Wasm proposals with the different runtimes [14].

Proposal	Wasmer	Wasmtime	WasmEdge	Browser
Reference types	✓	✓	✓	✓
Function references	✗	✓	✓	✓
Garbage collection	✗	✗	✓	✓
Tail call optimization	✗	✓	✓	✓

Table 1: Summary of Wasm proposal compatibility with different runtimes.

Table 2 shows a summary of the language support of the different runtimes [14].

Language	Wasmer	Wasmtime	WasmEdge
Rust	✓	✓	✓
C/C++	✓	✓	✓
.NET (C#, F#, VB)	✓	✓	✗
D	✓	✗	✗
Python	✓	✓	✓
JavaScript	✓	✗	✗
Go	✓	✓	✓
PHP	✓	✗	✗
Ruby	✓	✓	✗
Java	✓	✗	✓
R	✓	✗	✗
Postgres	✓	✗	✗
Swift	✓	✗	✗
Zig	✓	✗	✗
Dart	✓	✗	✗
Crystal	✓	✗	✗
Common Lisp	✓	✗	✗
Julia	✓	✗	✗
V	✓	✗	✗
OCaml	✓	✗	✗
Elixir	✗	✓	✗
Perl	✗	✓	✗

Table 2: Summary of language support of different runtimes.

2.4. Choice of compiler technology

The choice of compiler technology is essential for the project's success. The compiler should be able to translate the functional language subset into efficient Wasm bytecode. The following technologies were considered for the project.

2.4.1. LLVM

LLVM is a collection of modular and reusable compiler and toolchain technologies. It is widely used in industry and academia for developing compilers, static analysis tools, and runtime environments. LLVM provides a set of libraries and tools for building com-

pilers, including a compiler front-end (Clang), a compiler back-end (LLVM Core), and a set of optimization passes.

By making a compiler front-end that translates the functional language subset into LLVM intermediate representation (IR), the project could leverage LLVM's existing infrastructure for optimizing and generating efficient machine code. The LLVM IR can then be translated into Wasm bytecode using the Binaryen toolchain. Additionally targeting LLVM IR would allow the project to compile the functional language to other targets like x86, ARM, or RISC-V.

However, LLVM's complexity and the learning curve associated with it could make it challenging to implement within the project's timeframe. The project would also need to define a subset of the functional language that can be efficiently translated into LLVM IR.

Advantages:
- Efficient optimization and code generation capabilities.
- Support for multiple targets and architectures.
- Existing infrastructure for building compilers and toolchains.

Disadvantages:
- Complexity and learning curve associated with LLVM.
- Need to define a subset of the functional language that can be efficiently translated into LLVM IR.

2.4.2. Manual translation

Manual translation refers to the process of writing a custom compiler that directly translates the functional language subset into Wasm bytecode without using an intermediate representation like LLVM IR. This approach would involve defining a custom compiler architecture that parses the functional language syntax, performs semantic analysis, and generates Wasm bytecode.

While manual translation provides full control over the compilation process and allows for tailoring the compiler to the project's specific requirements, it can be time-consuming and error-prone. The project would need to implement lexing, parsing, type checking, and code generation from scratch.

Advantages:
- Full control over the compilation process (e.g., Wasm proposal compatibility, embedded runtime support, etc.).
- Tailoring the compiler to the project's specific requirements.

Disadvantages:
- Time-consuming and error-prone implementation.
- No existing infrastructure for optimization and code generation.

2.4.3. Binaryen

2.5. How the GHC Haskell compiler works

This chapter is inspired by notes from a lecture on the GHC compiler at Stanford University [2].

The Glasgow Haskell Compiler (GHC) is the most widely used Haskell compiler and provides a reference implementation for the Haskell language. GHC translates Haskell source code into intermediate representations (IRs) and eventually into machine code. The compilation process in GHC involves several stages, each performing specific tasks to optimize and generate efficient code.

In summary, the compilation process in GHC consists of the following stages (see Figure 2):

1. First the Haskell source code typed checked and desugared into a simplified intermediate representation (Core). This representation is very similar to the original Haskell code but all syntactical constructs are removed or transformed into only let and case statement. All pattern matching definitions for functions are also reduced to a lambda abstraction with a case statement. This enables to simplify the code and make it easier to optimize. The places where allocations take place (let bindings) and the place expressions get evaluated (case statements) are more clear to see and it is easier to reason about the execution order of the program.

2. The Core representation is then optimized using a set of optimization passes. These passes include inlining, constant folding, dead code elimination, and other optimizations that aim to improve the performance of the code. The optimizations are applied in a sequence of passes, each pass transforming the Core representation to a more optimized version.

3. The optimized Core representation is then translated into a lower-level intermediate representation called STG (Spineless Tagless G-machine). The STG representation is closer to the actual execution model of the Haskell runtime system and provides a more detailed view of the program's evaluation. In this representation the allocations only take place in the let bindings and evaluation only takes place in the case statements. The difference to the Core representation is that the STG representation is more detailed and provides more information about the evaluation order of the program, e.g. function application are represented as a thunk (a closure that takes no arguments) that gets evaluated when needed.

4. The STG representation is further optimized using a set of machine-independent optimizations. The optimizations are applied in a sequence of passes, each pass transforming the STG representation to a more optimized version.

5. The optimized STG representation is then translated into a lower-level intermediate representation called Cmm (C minus minus). The Cmm representation is a low-level imperative language that closely resembles the actual machine code that will

be generated. In this representation, the program is represented as a sequence of instructions that manipulate memory and perform computations.

6. The Cmm representation is further optimized using a set of machine-dependent optimizations. Compiler backends can then generate efficient machine code for the target architecture, C code, or LLVM IR.
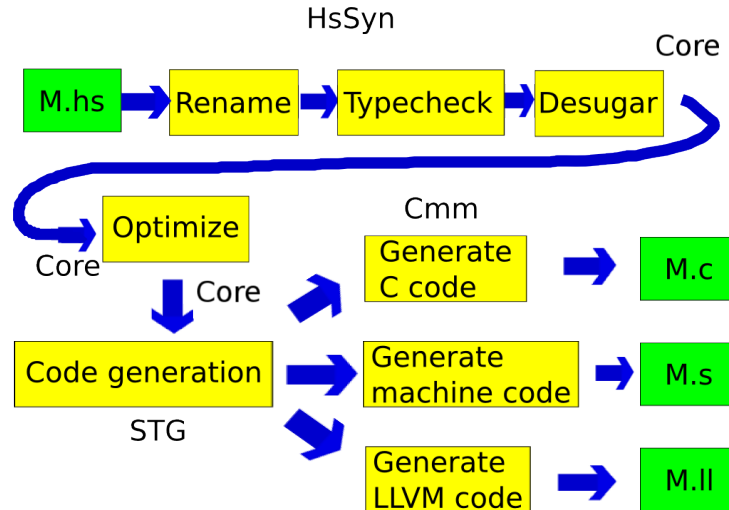
Figure 2: Compilation process in the GHC Haskell compiler (taken form the lecture [2]).

2.6. Other technological choices

The following technological choices were made for the project.

2.6.1. Gitlab

Gitlab is a web-based DevOps lifecycle tool that provides a Git repository manager providing wiki, issue-tracking, and CI/CD pipeline features. It is widely used in industry and academia for managing software projects and collaborating on code development. In the context of the project, Gitlab will be used to host the project's source code, issue-tracking, documentation, and CI/CD pipelines. The CI/CD pipelines in Gitlab will be used to automate the linting, testing, and deployment of the build artifacts.

2.6.2. Typst

The documentation for the project will be written in Typst, a typesetting system that allows for the creation of structured documents designed to be a modern alternative to LaTeX. Typst provides a simple and intuitive syntax for writing documents, including support for figures, tables, code blocks, a package ecosystem, and mathematical expressions.

Mr. Supcik provided a template for the project's documentation, which includes a title page, table of contents, list of figures, list of tables, a header and footer, and a bibliography. The author introduced a glossary, a syntax file for the Wasm text format to enable syntax highlighting, and some styling changes to the template.

2.6.3. Language for the compiler

Rust is a systems programming language that focuses on safety, speed, and concurrency. It is widely used in industry and academia for developing high-performance software, operating systems, and embedded systems. Rust's memory safety features, zero-cost abstractions, and modern tooling make it an ideal choice for implementing the compiler for the functional language subset.

Rust also has great support for WebAssembly and all its runtime environments. The Rust compiler can target WebAssembly directly and the Rust ecosystem provides tools and libraries for working with WebAssembly.

Section 3
Design

This section describes the design of the functional language, compiler, and standard library. It includes the lexical and context-free syntax of the language, the compiler's architecture, and the standard library's design.

3.1. Language specification

The syntax of the functional language (named "Waskell") is as already mentioned a subset of Haskell. There is a language specification document that defines the syntax and semantics of Haskell made by the Haskell team [5]. The Lexical syntax and context-free syntax of the functional language are based on this document. The lexical syntax refers to the rules that define how the characters in the source code are grouped into tokens. The context-free syntax refers to the rules that define how the tokens are grouped into expressions, declarations, and other constructs.

3.1.1. Lexical syntax

The lexical syntax of Waskell is identical to Haskell. Listing 3 shows the lexical syntax of the functional language (the text after the annotations is the token's name used in the compiler).

$$
\begin{array}{llll}
\text{Expr} & e & ::= x & \textit{variable} \\
& & \mid \; \lambda x.e & \textit{abstraction} \\
& & \mid \; e\ e & \textit{application}
\end{array}
$$

Listing 3: Lexical syntax of the functional language.

! TODO !
add the lexical syntax

3.1.2. Context-free syntax

The context-free syntax of Waskell is a subset of Haskell (defined in report [5]). The context-free syntax of Waskell is based on this document. Listing 4 shows the context-free syntax of the functional language (the text after the annotations is the non-terminal's name used in the compiler).

$$
\begin{aligned}
\textsf{Expr}\ e \Coloneqq\ & x && \textit{variable} \\
\mid\ & \lambda x.e && \textit{abstraction} \\
\mid\ & e\ e && \textit{application}
\end{aligned}
$$

Listing 4: Context-free syntax of the functional language.

! TODO !
add the context-free syntax

3.1.3. Language features

In short the subset of Haskell that is used in Waskell is the following:
- Let bindings (and where bindings)
- Lambda abstractions
- Function application and currying
- If expressions
- Case expressions
- Guards on function definitions and case expressions
- Simple pattern matching (no irrefutable patterns)
- Forced type annotations (no type inference)
- Simple types (Int, Bool, Char, String, List, Tuple)
- Simple type polymorphism (no type classes)
- Support for custom operators (no fixity, associativity, precedence)

! TODO !
keep language feature list up to date because that might change and add examples of the features and their limitations

3.1.4. Standard library design

! TODO !
keep the standard library design up to date

The standard library of the functional language is a subset of the Haskell standard library (or the Prelude). The standard library provides a set of functions and types that are commonly used in functional programming. The standard library includes functions for working with lists, tuples, numbers, and other data types. The design of the standard library is based on the Haskell standard library documentation [1].

Basic types

Listing 6 shows the list of basic types in the standard library.

```
-- Boolean type with values True and False.
data Bool = True | False
-- Character type representing Unicode characters.
data Char = ...
-- Integer type with fixed precision.
data Int = ...
-- Floating-point type with single precision.
data Float = ...
-- Floating-point type with double precision.
data Double = ...
-- String type representing lists of characters (alias for [Char]).
type String = [Char]
-- List of elements of type a (two constructors: [] and :).
data [a] = [] | a : [a]
-- Tuple type with n elements of types a, b, ..., z.
data (a, b, ..., z) = ...
-- Unit type with a single value denoted by ().
data () = ...
-- Maybe type representing optional values.
data Maybe a = Nothing | Just a
-- Either type representing disjoint unions.
data Either a b = Left a | Right b
```

Listing 6: The list of basic types in the standard library.

Boolean functions

Listing 7 shows the list of functions for working with booleans in the standard library.

```
not :: Bool -> Bool -- Negates a boolean value.
(&&), (||) :: Bool -> Bool -> Bool -- Logical AND and OR operations.
otherwise :: Bool -- Always returns True (useful in guards).
```

Listing 7: The list of functions for working with booleans.

Numeric functions

Listing 8 shows the list of functions for working with numbers in the standard library.

! TODO !

remove use of type classes and replace by implemented solution

bool

```haskell
data Ordering = LT | EQ | GT -- Ordering type for comparison results.
compare :: Ord a => a -> a -> Ordering -- Compares two values.
(+), (-), (*) :: Num a => a -> a -> a -- Addition, subtraction, and
multiplication.
negate, abs, signum :: Num a => a -> a -- Absolute value and signum.
(==), (/=), (<), (<=), (>), (>=) :: Ord a => a -> a -> Bool -- Comparison
operations.
min, max :: Ord a => a -> a -> a -- Minimum and maximum of two values.
minBound, maxBound :: Bounded a => a -- Smallest and largest value of a type.
quot, rem, div, mod :: Integral a => a -> a -> a -- Quotient and remainder
operations.
quotRem, divMod :: Integral a => a -> a -> (a, a) -- Quotient and remainder
as a pair.
(/) :: Fractional a => a -> a -> a -- Division operation.
recip :: Fractional a => a -> a -- Reciprocal of a value.
pi :: Floating a => a -- The value of pi.
exp, log, sqrt :: Floating a => a -> a -- Exponential, logarithm, and square
root.
(**), logBase :: Floating a => a -> a -> a -- Exponentiation and logarithm
base.
sin, cos, tan :: Floating a => a -> a -- Trigonometric functions.
asin, acos, atan :: Floating a => a -> a -- Inverse trigonometric functions.
sinh, cosh, tanh :: Floating a => a -> a -- Hyperbolic functions.
asinh, acosh, atanh :: Floating a => a -> a -- Inverse hyperbolic functions.
isNaN, isInfinite :: RealFloat a => a -> Bool -- Checks for NaN and infinity.
truncate, round, ceiling, floor :: RealFrac a => a -> Int -- Rounding
operations.
even, odd :: Integral a => a -> Bool -- Checks for even and odd numbers.
(^) :: (Num a, Integral b) => a -> b -> a -- Exponentiation operation.
```

Listing 8: The list of functions for working with numbers.

List functions

Listing 9 shows the list of functions for working with lists in the standard library.

```
head :: [a] -> a -- Returns the first element of a list.
(!!): [a] -> Int -> a -- Returns the element at a specific index.
last :: [a] -> a -- Returns the last element of a list.
tail :: [a] -> [a] -- Returns the list without the first element.
init :: [a] -> [a] -- Returns the list without the last element.
take :: Int -> [a] -> [a] -- Takes the first n elements of a list.
drop :: Int -> [a] -> [a] -- Drops the first n elements of a list.
splitAt :: Int -> [a] -> ([a], [a]) -- Splits a list at a specific index.
takeWhile :: (a -> Bool) -> [a] -> [a] -- Takes elements from a list while
a predicate is true.
dropWhile :: (a -> Bool) -> [a] -> [a] -- Drops elements from a list while
a predicate is true.
span :: (a -> Bool) -> [a] -> ([a], [a]) -- Splits a list into two parts
based on a predicate.
break :: (a -> Bool) -> [a] -> ([a], [a]) -- Splits a list into two parts
based on a predicate.
map :: (a -> b) -> [a] -> [b] -- Applies a function to each element of a
list.
filter :: (a -> Bool) -> [a] -> [a] -- Filters a list based on a predicate.
foldr :: (a -> b -> b) -> b -> [a] -> b -- Folds a list from the right.
foldl :: (b -> a -> b) -> b -> [a] -> b -- Folds a list from the left.
foldr1 :: (a -> a -> a) -> [a] -> a -- Folds a non-empty list from the right.
foldl1 :: (a -> a -> a) -> [a] -> a -- Folds a non-empty list from the left.
scanr :: (a -> b -> b) -> b -> [a] -> [b] -- Scans a list from the right.
scanr1 :: (a -> a -> a) -> [a] -> [a] -- Scans a non-empty list from the
right.
scanl :: (b -> a -> b) -> b -> [a] -> [b] -- Scans a list from the left.
scanl1 :: (a -> a -> a) -> [a] -> [a] -- Scans a non-empty list from the
left.
iterate :: (a -> a) -> a -> [a] -- Generates an infinite list by repeatedly
applying a function.
repeat :: a -> [a] -- Generates an infinite list with a single element.
cycle :: [a] -> [a] -- Generates an infinite list by cycling a list.
zip :: [a] -> [b] -> [(a, b)] -- Zips two lists together.
(++), concat :: [a] -> [a] -> [a] -- Concatenates two lists.
null :: [a] -> Bool -- Checks if a list is empty.
length :: [a] -> Int -- Returns the length of a list.
reverse :: [a] -> [a] -- Reverses a list.
any :: (a -> Bool) -> [a] -> Bool -- Checks if any element satisfies a
predicate.
all :: (a -> Bool) -> [a] -> Bool -- Checks if all elements satisfy a
predicate.
and :: [Bool] -> Bool -- Checks if all elements are true.
or :: [Bool] -> Bool -- Checks if any element is true.
sum :: [Int] -> Int -- Sums the elements of a list.
product :: [Int] -> Int -- Multiplies the elements of a list.
maximum :: [a] -> a -- Returns the maximum element of a list.
minimum :: [a] -> a -- Returns the minimum element of a list.
```

Listing 9: The list of functions for working with lists.

Tuple functions

Listing 10 shows the list of functions for working with tuples in the standard library.

```
fst :: (a, b) -> a -- Returns the first element of a tuple.
snd :: (a, b) -> b -- Returns the second element of a tuple.
curry :: ((a, b) -> c) -> a -> b -> c -- Curries a function.
uncurry :: (a -> b -> c) -> (a, b) -> c -- Uncurries a function.
```

Listing 10: The list of functions for working with tuples.

String functions

Listing 11 shows the list of functions for working with strings in the standard library.

```
lines :: String -> [String] -- Splits a string into lines.
words :: String -> [String] -- Splits a string into words.
unlines :: [String] -> String -- Joins lines into a string.
unwords :: [String] -> String -- Joins words into a string.
```

Listing 11: The list of functions for working with strings.

Miscellaneous functions

Listing 12 shows the list of miscellaneous functions in the standard library.

```
maybe :: b -> (a -> b) -> Maybe a -> b -- Handles optional values.
either :: (a -> c) -> (b -> c) -> Either a b -> c -- Handles disjoint unions.

id :: a -> a -- Identity function.
const :: a -> b -> a -- Constant function.
flip :: (a -> b -> c) -> b -> a -> c -- Flips the arguments of a function.
($) :: (a -> b) -> a -> b -- Function application operator.
(.) :: (b -> c) -> (a -> b) -> a -> c -- Function composition operator.
until :: (a -> Bool) -> (a -> a) -> a -> a -- Repeatedly applies a function
until a predicate is true.
error :: String -> a -- Throws an error with a message.
undefined :: a -- Throws an undefined error.
```

Listing 12: The list of miscellaneous functions.

3.2. Compiler architecture

Section 4
Implementation

Section 5
Evaluation

Section 6
Conclusion

6.1. Challenges

6.2. Future work

6.3. Personal opinion

Declaration of honor

In this project, we used generative AI tools, namely <u>GitHub Copilot</u> for coding and <u>Claude AI</u> for paraphrasing. Copilot was employed as an advanced autocomplete feature, but it did not generate a significant portion of the project. I, the undersigned Noah Godel, solemnly declare that the submitted work is the result of personal effort. I certify that I have not resorted to plagiarism or other forms of fraud. All sources of information used and author citations have been clearly acknowledged.

Glossary

Claude AI: Claude AI is a LLM based AI chat bot by Anthropic. <u>30</u>

GitHub Copilot: GitHub Copilot is an AI pair programmer that uses AI to make context related code suggestions. <u>30</u>

HEIA-FR – Haute école d'ingénierie et d'architecture de Fribourg: The Haute école d'ingénierie et d'architecture de Fribourg (HEIA-FR) is a technical school of applied sciences located in Fribourg, Switzerland. The name in english is: School of Engineering and Architecture of Fribourg. <u>2</u>

Wasm – WebAssembly: WebAssembly (often shortened to Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications. <u>2</u>, <u>5</u>, <u>6</u>, <u>7</u>, <u>16</u>, <u>17</u>

Wasm runtime: A WebAssembly runtime is a software that executes WebAssembly code. It can be a standalone runtime like Wasmtime or integrated into a larger software like a web browser. <u>2</u>

Lisp dialects

Wasm runtimes

Bibliography

[1]   The Haskell community. 2024. Haskell Prelude documentation. Retrieved from https://hackage.haskell.org/package/base-4.7.0.1/docs/Prelude.html

[2]   Brian O'Sullivan. 2012. Lecture notes on the GHC compiler. Retrieved from https://www.scs.stanford.edu/11au-cs240h/notes/ghc.html

[3]   Bytecode Alliance. 2024. WebAssembly Component Model proposal. Retrieved from https://component-model.bytecodealliance.org/

[4]   Noah Godel. 2024. *Functional language compiler to WebAssembly - Specification.*

[5]   Simon Marlow. 2010. Haskell language specification. Retrieved from https://www.haskell.org/onlinereport/haskell2010/

[6]   WasmEdge. 2024. List of supported Wasm proposals for the Wasmedge runtime. Retrieved from https://wasmedge.org/docs/start/wasmedge/extensions/proposals/#standard-webassembly-features

[7]   WasmEdge. 2024. List of SDKs for the Wasmedge runtime. Retrieved from https://wasmedge.org/docs/embed/overview

[8]   Wasmer. 2024. List of SDKs for the Wasmer runtime. Retrieved from https://github.com/wasmerio/wasmer?tab=readme-ov-file#wasmer-sdk

[9]   Wasmtime. 2024. List of SDKs for the Wasmtime runtime. Retrieved from https://github.com/wasmerio/wasmer?tab=readme-ov-file#wasmer-sdk

[10]   WebAssembly Community Group. WebAssembly Reference Types proposal. Retrieved from https://github.com/WebAssembly/reference-types

[11]   WebAssembly Community Group. WebAssembly Function References proposal. Retrieved from https://github.com/WebAssembly/function-references

[12]   WebAssembly Community Group. 2024. WebAssembly Garbage Collection proposal. Retrieved from https://github.com/WebAssembly/gc

[13]   WebAssembly Community Group. 2024. WebAssembly Tail Call proposal. Retrieved from https://github.com/WebAssembly/tail-call

[14]   WebAssembly Community Group. 2024. WebAssembly feature extensions. Retrieved from https://webassembly.org/features/

Table of figures