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, 31 May 2024

Version 0.1

Hes·so

Table of versions

Version	**Date**	**Modifications**
0.1	31.5.2024	First version

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 [1]. 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 [1].

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 [1].

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.

! TODO !

add choice of language subset (specify the Wasm compatibility, the functional features, the standard library)

2.1.1. OCaml

2.1.2. F#

2.1.3. Lisp languages

2.1.4. BEAM languages

BEAM

2.1.5. Haskell

2.2. 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.

! TODO !

add choice of compiler technology (specify the Wasm compatibility, the functional features, the standard library)

2.2.1. LLVM

2.2.2. Manual translation

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. The following technologies were considered for the project.

! TODO !

add embedding technologies (specify the Wasm compatibility, the functional features, the standard library), interface types ??? => (https://docs.wasmer.io/wai)

2.3.1. Wasmer

2.3.2. Wasmtime

2.3.3. Wasmtime with WebAssembly Component Model

2.4. How the GHC Haskell compiler works

2.5. Other technological choices

! TODO !

add other technological choices

2.5.1. Gitlab

2.5.2. Typst

2.5.3. Language for the compiler

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.

! TODO !

add design of the functional language, compiler, and standard library

3.1. Lexical syntax

3.2. Context-free syntax

3.3. Compiler architecture

3.4. Language features

3.5. Standard library design

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 GitHub Copilot for coding and Claude AI 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

BEAM: BEAM is the virtual machine at the core of the Erlang Open Telecom Platform (OTP). It is a process virtual machine that is responsible for executing Erlang and Elixir code. 4

Claude AI: Claude AI is a LLM based AI chat bot by Anthropic. 11

GitHub Copilot: GitHub Copilot is an AI pair programmer that uses AI to make context related code suggestions. 11

***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. 2

***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. 2, 5

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. 2

Wasm runtimes

Bibliography

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

Table of figures