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

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 [2]. 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 [2].

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

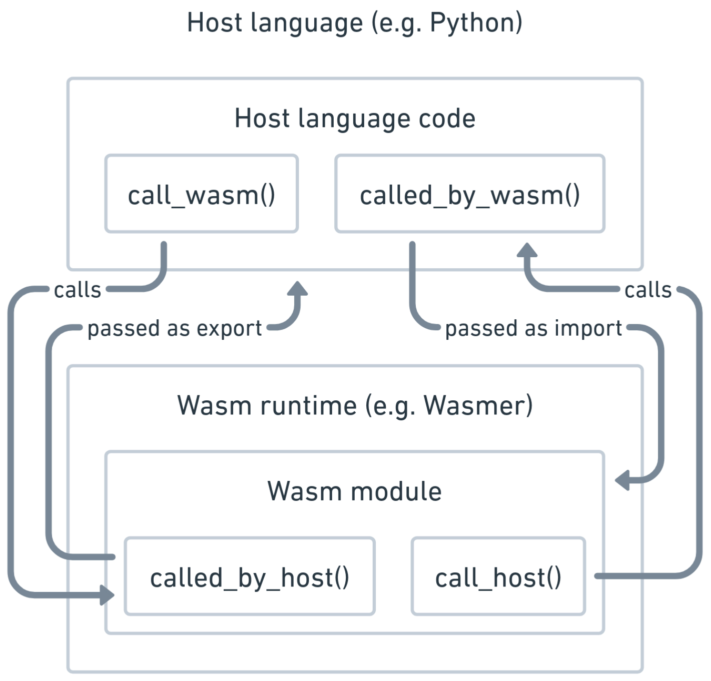

Figure 1: Illustration of embedding a Wasm module into a codebase.

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 [2].

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 Wasm 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 Wasm 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 Wasm 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 Wasm 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 Wasm, 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 Wasm, 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 [1] 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 [3] 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 [4] 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 [5] 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 [6] 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.

```
(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. 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. WasmEdge

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.

! TODO !

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

2.4.1. LLVM

2.4.2. Manual translation

2.4.3. Binaryen

2.5. How the GHC Haskell compiler works

2.6. Other technological choices

! TODO !

add other technological choices

2.6.1. Gitlab

2.6.2. Typst

2.6.3. 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

Skipped features:

- irrefutable patterns + pattern bindings
- typed expressions
- holes
- operator stuff ?? (precedence, associativity, fixity, arity > 2 operators / paren func def thing)
  - "(a &* b) c = …"

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

Claude AI: Claude AI is a LLM based AI chat bot by Anthropic. 20

GitHub Copilot: GitHub Copilot is an AI pair programmer that uses AI to make context related code suggestions. 20

***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, 6, 7, 14

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

Lisp dialects

Wasm runtimes

Bibliography

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

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

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

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

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

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

Table of figures