






Student/in:
Godel Noah

Zusammenfassung Bachelorarbeit 2024
INFORMATIK UND KOMMUNIKATIONSSYSTEME

DOZENT/IN: Jacques Supcik, Serge Ayer	
AUFTRAGGEBER/IN: HTA-FR	
INTERNER KONTAKT: jacques.supcik@hefr.ch	
KURZZEICHEN PROJEKT: FLC-WASM	INTERNE PROJEKT-NR.: B24ISC07
VERTIEFUNGSRICHTUNGEN: Software-Engineering	EXPERTE/EXPERTIN: Baptiste Wicht, Valentin Bourqui
NACHHALTIGKEITSZIELE:   	

Compiler für eine funktionale Programmiersprache zu WebAssembly (Wasm)

Das Hauptziel dieses Projekts ist die Entwicklung eines Compilers für eine funktionale Sprache zu Wasm. Das Projekt zielt darauf ab, ein Subset von der Haskell Sprache zu definieren, einen Compiler zu entwickeln, der die Sprache in Wasm-Bytecode übersetzt, und eine Dokumentation für die Sprache und den Compiler bereitzustellen. Durch die Verwendung von Wasm-Runtimes kann der kompilierte Code in verschiedenen Umgebungen ausgeführt werden, wie zum Beispiel in anderen Programmiersprachen oder Webbrowsern.

Das funktionale Paradigma

Das funktionale Paradigma bietet mehrere Vorteile, wie Modularität, Prägnanz, Sicherheit, Parallelität, höhere Ordnungsfunktionen und einen deklarativen Stil. Haskell wurde als Grundlage für das Subset aufgrund seiner rein funktionalen Natur, des fortgeschrittenen Typsystems, der bestehenden Werkzeuge für die Wasm-Kompilierung und der Vertrautheit des Autors mit der Sprache ausgewählt. Das Subset der Sprache wird so gestaltet, dass sie innerhalb des Projektzeitrahmens ausdrucksstark und überschaubar ist.

Im Vergleich zu einer imperativen Sprache stellt die Implementierung einer funktionalen Sprache Herausforderungen dar, wie z.B. die Unterstützung der Lazy Evaluation, der Garbage collection, der Funktionen als Werte usw. Das Projekt wird einen naiven Ansatz zur Lazy Evaluation verwenden, denn

eine effiziente Lösung wäre nicht vernünftig in der kurzen Zeit, die zur Verfügung steht. Das Projekt wird kein Garbage collecting und keine Tail-Call-Optimierung haben. Der Fokus des Projekts liegt nicht auf der Implementierung eines vollständig optimierten Compilers, sondern darauf, die Einbettung der funktionalen Sprache in andere Codebasen zu demonstrieren.

Beispiel Code

Das folgende Code Beispiel zeigt eine Funktion in Waskell (der Name des Subset von Haskell), die die Fibonacci-Folge für eine gegebene Zahl berechnet. Die Funktion wird mithilfe von Pattern Matching und Rekursion definiert, gängige Techniken in der funktionalen Programmierung. Die Fibonacci-Funktion nimmt eine Ganzzahl n und gibt die n -te Fibonacci-Zahl zurück. Die Basisfälle sind für $n = 0$ und $n = 1$ definiert, und der rekursive Fall ist für $n > 1$ definiert. Die Funktion ruft sich selbst rekursiv auf, um die Fibonacci-Zahlen für $n - 1$ und $n - 2$ zu berechnen und gibt deren Summe zurück. Die Fibonacci-Funktion demonstriert die Eleganz und Prägnanz der funktionalen Programmierung in Haskell.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Wasm-Runtimes



Student/in:
Godel Noah

Zusammenfassung Bachelorarbeit 2024
INFORMATIK UND KOMMUNIKATIONSSYSTEME

Wie bereits erwähnt, zielt das Projekt darauf ab, die Einbettung der neuen funktionalen Sprache, die zu Wasm kompiliert wurde, in bestehende Codebasen zu demonstrieren und dabei die Interoperabilität und das Potenzial zur Kombination von Paradigmen innerhalb desselben Projekts aufzuzeigen. Dies funktioniert, da es mehrere Wasm-Runtimes gibt (wie Wasmtime, Wasmer, WasmEdge usw.), die die Ausführung von Wasm-Modulen in verschiedenen Sprachen unterstützen. Einige dieser Runtimes ermöglichen sogar die bidirektionale Kommunikation zwischen der Host-Codebasis und dem Wasm-Modul.

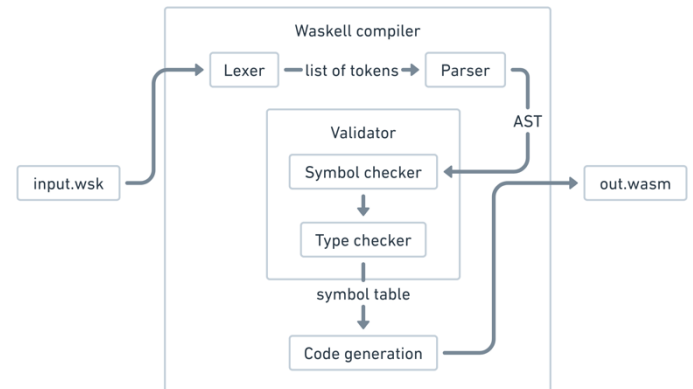
Das folgende Code-Beispiel zeigt ein Beispiel für die Verwendung der Fibonacci-Funktion, die in der Waskell-Sprache definiert ist, in einer Python-Codebasis unter Verwendung der Wasmtime-Runtime. Der Python-Code lädt das Wasm-Modul, das die Fibonacci-Funktion enthält, ruft die Funktion mit einem Argument auf und druckt das Ergebnis aus. Dies demonstriert die Interoperabilität der Waskell-Sprache mit anderen Sprachen durch Wasm-Runtimes.

```
import wasmtime.loader
import fib_module

print(fib_module.fib(10))
# Output: 55
```

Struktur des Compilers

Die folgende Abbildung illustriert die Struktur des Compilers für funktionale Sprachen zu WebAssembly. Der Compiler besteht aus mehreren Komponenten, darunter ein Lexer, ein Parser, ein Validator und ein Codegenerator.



- **Lexer:** Der Lexer liest den Quellcode und wandelt ihn in einer Liste von Tokens um. Der Lexer erkennt Schlüsselwörter, Bezeichner, Literale und andere sprach Konstrukte.
- **Parser:** Der Parser liest den Strom von Tokens und konstruiert einen abstrakten Syntaxbaum (AST), der die Struktur des Programms darstellt.
- **Validator:** Der Validator überprüft den AST auf Syntax- und Semantikfehler und transformiert ihn anschliessend in eine Symboltabelle.
- **Codegenerator:** Der Codegenerator übersetzt die Symboltabelle in Wasm-Bytecode und generiert den endgültigen ausführbaren Code.

Für detailliertere Informationen und eine tiefere Einsicht in die technischen Aspekte dieses Projekts lade ich Sie herzlich ein, die vollständige Dokumentation zu lesen. Sie bietet eine umfassende Beschreibung der Architektur, der Implementierung und der praktischen Anwendungen des Compilers für funktionale Sprachen zu Wasm. Viel Vergnügen beim Lesen!