





Etudiante·es :
Godel Noah

Résumé travail de Bachelor 2024
INFORMATIQUE ET SYSTÈMES DE COMMUNICATION

PROFESSEUR·ES : Jacques Supcik, Serge Ayer	
MANDANT : HEIA-FR	
CONTACT INTERNE : jacques.supcik@hefr.ch	
ACRONYME DU PROJET : FLC-WASM	NO INTERNE OU FILIÈRE : B24ISC07
ORIENTATION : Ingénierie logicielle	EXPERT·ES : Baptiste Wicht, Valentin Bourqui
OBJECTIFS DE DEV. DURABLE :   	

Compilateur pour un langage fonctionnel vers WebAssembly

L'objectif principal de ce projet est de développer un compilateur pour un langage fonctionnel vers Wasm. Le projet vise à définir un sous-ensemble du langage Haskell, à développer un compilateur qui traduit le langage en bytecode Wasm, et à fournir une documentation pour le langage et le compilateur. En utilisant les runtimes Wasm, le code compilé peut être exécuté dans divers environnements, tels que d'autres langages de programmation ou des navigateurs web.

Le paradigme fonctionnel

Le paradigme fonctionnel offre plusieurs avantages, tels que la modularité, la concision, la sécurité, le parallélisme, les fonctions d'ordre supérieur et un style déclaratif. Haskell a été choisi comme base pour le sous-ensemble en raison de sa nature purement fonctionnelle, de son système de types avancé, des outils existants pour la compilation Wasm, et de la familiarité de l'auteur avec le langage. Le sous-ensemble du langage est conçu pour être expressif et gérable dans les délais du projet.

La mise en œuvre d'un langage fonctionnel pose des défis par rapport à un langage impératif, tels que la prise en charge de l'évaluation paresseuse, de la collecte des ordures et des fonctions comme valeurs. Le projet utilisera une approche naïve de l'évaluation paresseuse, car une solution efficace ne serait pas raisonnable dans le temps imparti. Le projet n'inclura pas de collecte des ordures ni d'optimisation des appels en queue. L'objectif n'est

pas de mettre en œuvre un compilateur entièrement optimisé, mais de démontrer l'intégration du langage fonctionnel dans d'autres bases de code.

Exemple de code

L'exemple suivant montre une fonction en Haskell (le nom du sous-ensemble de Haskell) qui calcule la suite de Fibonacci pour un nombre donné. La fonction est définie en utilisant le pattern matching et la récursion, des techniques courantes en programmation fonctionnelle. La fonction Fibonacci prend un entier n et retourne le n ème nombre de Fibonacci. Les cas de base sont définis pour $n = 0$ et $n = 1$, et le cas récursif est défini pour $n > 1$. La fonction s'appelle récursivement pour calculer les nombres de Fibonacci pour $n - 1$ et $n - 2$ et retourne leur somme. Cela démontre l'élégance et la concision de la programmation fonctionnelle en Haskell.

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



Runtimes Wasm

Comme mentionné précédemment, le projet vise à démontrer l'intégration du nouveau langage fonctionnel, compilé en Wasm, dans des bases de code existantes, en montrant l'interopérabilité et le potentiel de combinaison des paradigmes au sein du même projet. Cela est possible car il existe plusieurs runtimes Wasm (tels que Wasmtime, Wasmer, WasmEdge, etc.) qui supportent l'exécution de modules Wasm dans différents langages. Certains de ces runtimes permettent même la communication bidirectionnelle entre la base de code hôte et le module Wasm.

L'exemple de code suivant démontre l'utilisation de la fonction Fibonacci, définie dans le langage Waskell, au sein d'une base de code Python en utilisant le runtime Wasmtime. Le code Python charge le module Wasm contenant la fonction Fibonacci, appelle la fonction avec un argument et imprime le résultat. Cela démontre l'interopérabilité du langage Waskell avec d'autres langages grâce aux runtimes Wasm.

```
import wasmtime.loader
import fib_module

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

Structure du compilateur

Le diagramme suivant illustre la structure du compilateur pour les langages fonctionnels vers WebAssembly. Le compilateur se compose de plusieurs composants, dont un lexer, un parser, un validateur et un générateur de code.

- **Lexer:** Le lexer lit le code source et le convertit en une liste de tokens. Le lexer reconnaît les mots-clés, les identifiants, les littéraux et autres constructions du langage.
- **Parser:** Le parser lit le flux de tokens et construit un arbre syntaxique abstrait (AST) qui représente la structure du programme.
- **Validateur:** Le validateur vérifie l'AST pour des erreurs de syntaxe et de sémantique, puis le transforme en une table des symboles.
- **Générateur de code:** Le générateur de code traduit la table des symboles en bytecode Wasm et génère le code exécutable final.

Pour des informations plus détaillées et une compréhension approfondie des aspects techniques de ce projet, je vous invite à lire la documentation complète. Elle offre une description complète de l'architecture, de l'implémentation et des applications pratiques du compilateur pour les langages fonctionnels vers Wasm. Bonne lecture!