プログラミング言語 Wasabi の設計と実装

hotaka B1 環境情報学部

目的

- WebAssembly の学習
- Rust の学習
- ゲーム開発やクリエイティブコーディングに使いやすい言語の設計

WebAssembly について

- Webブラウザで高速にプログラムを実行できるスタックマシンベースの仮想命令セット
- バイナリ形式でプログラムを実行する
- 中間表現としてS式を利用できる
- Webブラウザ以外の環境(OS上)でも動作するようになってきている
- 低水準のホスト言語(Rust や C++)からコンパイルして利用する
- DOM操作やブラウザのAPIを操作する際は JavaScript で定義した関数を呼び出して行う

対応ブラウザの例

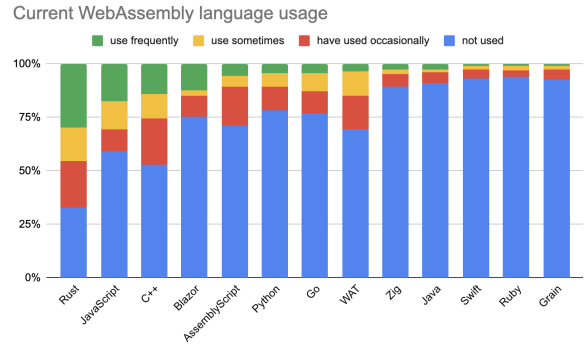
- Google Chrome
- Firefox
- Safari

WebAssembly Text Format(WAT)

```
;; スタックマシンで処理される

i32.const 1
i32.const 2
i32.add
;; 3

;; JavaScript の関数を呼び出す

(import "js" "print" (func $print (param i32)))
i32.const 10
call $print
```

背景

- WebAssembly は JavaScript よりもネイティブに近いパフォーマンスを出すことができる
- WebAssembly は主に Rust や C++ といった低水準のプログラミング言語からコンパイルしてアプリに組み込む
  - JavaScript という言語の構文に囚われずにWebアプリケーションを記述することができる
  - 特に Rust は WebAssembly のホスト言語として使われることが多い
- これらの言語は付属のツールチェーンを用いてプロジェクトを作成しコンパイラのターゲットに WebAssembly を指定してからコンパイルを行う必要がある



"The State of WebAssembly 2022" by Colin Eberhardt より引用

https://blog.scottlogic.com/2022/06/20/state-of-wasm-2022.html

問題

- JavaScript ユーザには低水準の言語の学習コストが高い
  - 構文やパラダイムの違い
- 多くの WebAssembly のホスト言語はその言語中心のプロジェクトになることが多い
  - 既存のアプリケーションやアプリケーションに部分的に WebAssembly を取り込むのにコストがかかる
    - 例: プロジェクトの作成 / ビルド / グルーコードの生成

目標

- JavaScript ユーザが手軽に WebAssembly を扱えるようにする

関連技術: AssemblyScript

- TypeScript(型付きの JavaScript のスーパーセット)から WebAssembly を出力する言語
- TypeScript がベースなので JavaScript ユーザでも親しみやすい構文を持つ
- WebAssembly の型による制限はあるが TypeScript の型システムを使いながら WebAssembly を利用できる
- **TypeScript の構文に縛られているため言語としての表現力が弱い**

提案手法

- 以下の特徴を持つプログラミング言語「Wasabi」の設計と実装を行う
  - Rust と TypeScript をベースに構文を設計する
    - **JavaScript ユーザでも親しみやすい構文を心がける**
  - クラスやインターフェイスといったオブジェクト指向に対応する
    - **DOM や Canvas API を JavaScript のコードと同じように表現できるようにする**
  - 演算子や関数のオーバーロードに対応する
    - **ゲームやシミュレーションのコードをより簡潔に書けるようにする(例: ベクトル演算)**
- この言語のファイルを JavaScript から直接読み込めるようにするための Webpack プラグインの開発
  - Webpack = オープンソースの JavaScript モジュールバンドラー
    - プラグインを用いることで `` `import classes from 'style.css'` `` のように別のフォーマットのファイルを解析して JavaScript に変換することができる。
  - `` `import App from 'app.was'` `` のようにWasabi言語の関数やクラスを読み込めるようにする

設計

- 以下のモジュールを実装する
  - `tokens`
    - トークンの定義
  - `lexer`
    - 字句解析器
  - `ast`
    - AST ノードの定義
  - `parser`
    - 構文解析器
  - `compiler`
    - WebAssembly へのコンパイラ
  - `wasa`
    - CLI

開発している言語

- 言語名: Wasabi
- 開発言語: Rust
- 論理LOC: 2494行
- 実装済みの機能
  - WebAssembly Text Format へのコンパイル(S式)
  - 関数定義, JavaScript関数のインポート, 変数定義
  - if, while
  - 四則演算, 基本的な論理演算(and, or, not)
  - 型: i32, i64, f32, f64, bool

実行までの流れ : Wasabi のコード

`1 + 2` の結果を JavaScript の `print` 関数に渡すコード。

```
// JavaScript の関数をインポート
import js {
  fn alert(i32);
}

// `export` で関数を JavaScript 側に公開
export fn main() {
  alert(1 + 2);
}
```

実行までの流れ : Wasabi のコード

`1 + 2` の結果を JavaScript の `print` 関数に渡すコード。

```
// JavaScript の関数をインポート
import js {
  fn alert(i32);
}

// `export` で関数を JavaScript 側に公開
export fn main() {
  alert(1 + 2);
}
```

実行までの流れ : Wasabi のコード

`1 + 2` の結果を JavaScript の `print` 関数に渡すコード。

```
// JavaScript の関数をインポート
import js {
  fn alert(i32);
}

// `export` で関数を JavaScript 側に公開
export fn main() {
  alert(1 + 2);
}
```

実行までの流れ : WasabiからText Format(WAT)へのコンパイル

Wasabi言語のコンパイラCLIの `wasa` を利用する。

```
$ ./wasa main.was > main.wat
```

main.wat の内容

```
$ cat main.wat
(module
  (import
    "js"
    "print"
    (func
      $alert
      (param i32)
    )
  )
  (func
    $main
    (export "main")
    (call
      $alert
      (i32.add
        (i32.const 1)
        (i32.const 2)
      )
    )
  )
)
```

実行までの流れ : WAT からバイナリへの変換

WebAssembly が公式に提供しているツールチェインの `wabt` に付属している `wat2wasm` を使用する。

```
$ wat2wasm main.wat -O main.wasm
$ file main.wasm
main.wasm: WebAssembly (wasm) binary module version 0×1 (MVP)
```

実行までの流れ : JavaScript コードの準備

```
// JavaScript
const imports = {
  js: {
      alert(value) {
          alert(value)
      }
  }
}
const { instance } = await WebAssembly.instantiateStreaming(fetch('/main.wasm'), imports)
instance.exports.main()
```

```
// 先程の Wasabi コード
import js {
  fn alert(i32);
}

export fn main() {
  alert(1 + 2);
}
```

実行までの流れ : JavaScript コードの準備

```
// JavaScript
const imports = {
  js: {
      alert(value) {
          alert(value)
      }
  }
}
const { instance } = await WebAssembly.instantiateStreaming(fetch('/main.wasm'), imports)
instance.exports.main()
```

```
// 先程の Wasabi コード
import js {
  fn alert(i32);
}

export fn main() {
  alert(1 + 2);
}
```

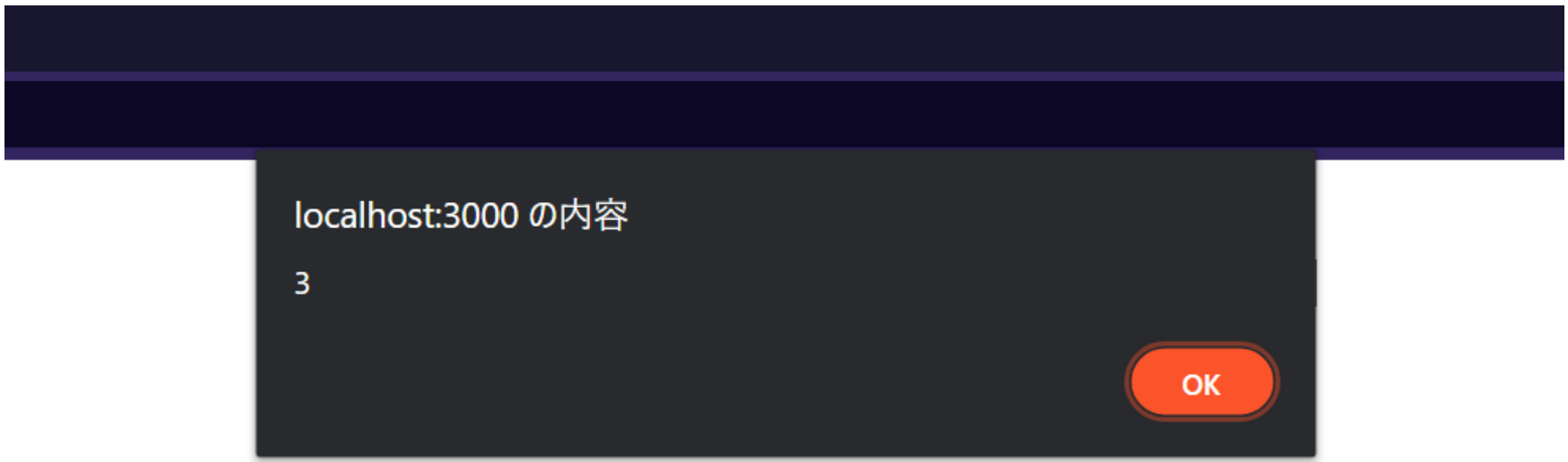
実行までの流れ : JavaScript コードの準備

```
// JavaScript
const imports = {
  js: {
      alert(value) {
          alert(value)
      }
  }
}
const { instance } = await WebAssembly.instantiateStreaming(fetch('/main.wasm'), imports)
instance.exports.main()
```

```
// 先程の Wasabi コード
import js {
  fn alert(i32);
}

export fn main() {
  alert(1 + 2);
}
```

実行時のスクリーンショット

環境: Brave



構文紹介：`import <モジュール名> { <関数定義 ... > }`

JavaScript の `WebAssembly.instantiate` で渡された関数をインポートします。

```
// Wasabi
import js {
  fn alert(i32);
}
```

```
// JavaScript
WebAssembly.instantiate(source, {
  js: {
    alert(value) {
      alert(value)
    }
  }
})
```

構文紹介：`fn <関数名>(<引数...>): <返り値> { <本体> }`

```
// 返り値が無い場合は省略可能
fn main() {
  print(1 + 2);
}
```

```
fn add(a: i32, b: i32): i32 {
  // return a + b;
  a + b // ブロックの最後が式ならそのまま返り値になる。
}
```

構文紹介：`let <変数名>: <型> = 初期化`

ローカル変数を定義します。

```
fn main() {
  let year: i32 = 2003;
  let is_leap_year: bool = true;

  // 型を省略すると初期化式から自動的に型が決定されます。
  let radius = 10; // i32
  let pi = 3.14;   // f64
}
```

構文紹介：`while <式> { <処理> }`

```
fn main() {
  let i = 0;

  while i < 100 {
    print(i);
  };
}
```

評価

- 環境: Windows 11, Brave(Chromium 109.0.5414.87)
- 比較言語: JavaScript, Rust
- 比較手法
  - 処理速度の比較(Perfomance API の `performance.now()` 関数を用いて計測を行う)

評価の内容

- 各言語で `1` から 与えられた引数 `max` までの合計値を計算して返す関数を実装する。
- JavaScript からそれぞれの関数に `1,000,000` を渡して「1 から 1000000 までの合計値を計算する」処理の時間を計測する。

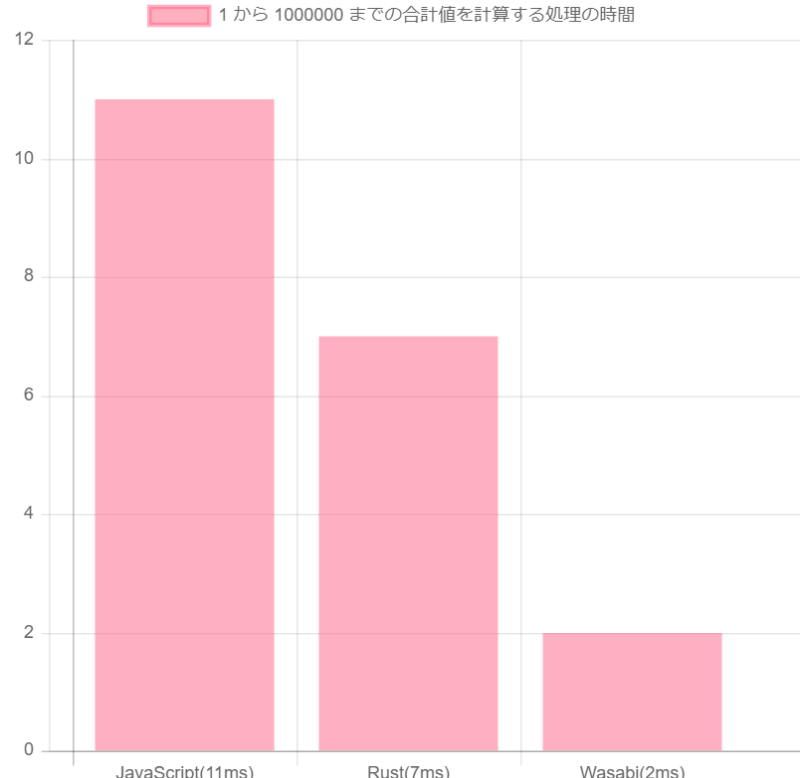
```
// JavaScript
const sum = (max) => {
  let i = 0;
  let r = 0;

  while (i < max) {
    i += 1;
    r += i;
  }

  return r;
}
```

```
// Rust
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn sum(max: i64) -> i64 {
    let mut i = 0;
    let mut r = 0;

    while i < max {
        i += 1;
        r += i;
    }

    r
}
```

```
// Wasabi
export fn sum(max: i64): i64 {
  let i: i64 = 0;
  let r: i64 = 0;

  while i < max {
    i += 1 as i64;
    r += i;
  };

  r
}
```

評価の結果

- Wasabiの出力した WebAssembly が JavaScript に比べて約1.5倍早い, Rust の出力した WebAssembly に比べて約3倍早い結果となった



今後

- 言語の設計と実装
  - 配列やクラスなどの実装
  - 関数や演算子のオーバーロードの実装
- CLI / ツールチェイン
  - TypeScript の型定義ファイルの生成機能
  - Webpack プラグインの開発
  - Language Server Protocol の対応