WebAssemblyを対象にしたSchemeコンパイラの作成に向けて

荒井滉平 増原英彦 叢悠悠

WebAssembly はスタックベースの仮想機械語であり，Java バイトコードや CIL のような仮想機械語と比較してより機械語に近い抽象度を持つ． Scheme のように第一級継続を有する言語からそのような対象言語へのコンパイルに対する取り組みが行われていないため,．本論文では第一級継続をサポートする Scheme から高速な WebAssembly へのコンパイル手法を検討する準備として，簡単なコンパイラと基本的な最適化のいくつかを実現し，その効果を測定した．今後の展望として，コンパイラの対象言語を WebAssembly に限定継続を導入した wasm/k [6] に変更し，call-with-current-continuation をその限定継続を用いて実装することで，今回作成したコンパイラとどのような性能差が生じるかについて議論する．

1 はじめに

近年，高級言語からのコンパイル対象として，仮想機械語を用いる言語処理系が増えている．機械語を直接生成する言語処理系と比べて，多様な実行環境に容易に対応できることや，プロファイル情報を用いた実行時最適化・実行時コンパイルなどの技術の発展により，機械語を生成する場合よりも性能向上が望める場合もあることなどの点から期待されている．仮想機械後には Java バイトコード [7]・CIL [4]・WebAssembly [2] などがある．WebAssembly は近年登場した仮想機械語で，主要ブラウザや Node.js などの JavaScript 実行環境上で動作する特徴がある．

一方，仮想機械語が提供する抽象度には様々なものがあり，ソース言語をその仮想機械語にコンパイルする場合には，ソース言語に備わるオブジェクト・ごみ集め・第一級関数・第一級継続などの高度な機能に適切な表現方法を見つけることや，仮想機械による実行時最適化がどの程度期待できるかが問題となる．例えば，Java バイトコードは Java 言語のための仮想機械語として設計されているため，関数型言語の第一級関数を表現するためには Java のオブジェクトを用いなければいけない．第一級継続を実装する場合，コールスタックをヒープメモリに保存するなどの操作する必要がある．しかし，仮想機械語はコールスタックを操作できない設計になっている場合が多い．その代わりに例外機構を利用して実装している．

我々は第一級関数と第一級継続を持つソース言語から WebAssembly を対象言語としたコンパイラを研究する．特に第一級継続を持つソース言語を WebAssembly へとコンパイルする処理系の研究はまだ行われていない．第一級継続を持つ言語から WebAssembly へのコンパイラは存在はするが，第一級継続自体のサポートはされていない．

一方，第一級継続を仮想機械自体で提供する Wasm/k [6] のような研究もあるため，第一級継続を WebAssembly 上でどのように実現するかについて検討することが主な目標である．そのための最初の段階として，本論文では，第一継続を持たない Scheme 言語のサブセットを WebAssembly へとコンパイルする処理系を試作し，いくつかの基本的なコード最適化の効果を調べる．これによって，現在の WebAssembly は仮想機械にどの程度の実行時最適化が期待できるかを明らかにすることが目標である．

Kohei Arai, Hidehiko Masuhara, Youyou Cong, 東京工業大学情報理工学院数理・計算科学系, Dept. of Mathematical and Computing Science, Tokyo Institute of Technology.

```
(table 1 anyfunc)
(elem (i32.const 0) $add1)
(func $add1 (param $x i32) (result i32)
  (local.get $x)
  (i32.const 1)
  i32.add)
(func $main
  (;関数の直接呼び出し;)
  (i32.const 100)
  (call $add1)
  (;関数の間接呼び出し;)
  (i32.const 100)
  (i32.const 0)
  (call_indirect (type (func (param i32
      ) (result i32))))
  ...)
```

図 1: WebAssembly の関数の例

2 背景

2.1 WebAssembly の仕様

WebAssembly とは，仮想機械語の一つで C/C++ や Rust などの様々な高級言語からコンパイルされる．Chrome・FireFox・Safari などの主要ブラウザや Node.js などの JavaScript 実行環境上で動作する．WebAssembly は関数を実行単位とする．WebAssembly の仮想機械はスタックベースであり，WebAssembly の各命令はオペランドスタックを操作する．例えば，(i32.const 3) はオペランドスタックに i32 型の 32bit 整数の 3 を積む．i32.add はオペランドスタックの先頭から i32 型の値を 2 つ取り出し，それらの和をオペランドスタックに積む．

2.1.1 データ型

WebAssembly の基本データ型は i32・i64・f32・f64 の 4 種類である．それぞれ順に 32bit 整数・64bit 整数・32bit 浮動小数点数・64bit 浮動小数点数を表す．WebAssembly では i32 は真理値の表現にも利用される．

2.1.2 関数

WebAssembly の関数はパラメータを受け取り，本体を実行した結果をオペランドスタックに積む．関数実行終了時にオペランドスタックに積まれていた値が返り値となる．result によって関数実行後のオペランドスタックに積まれている値の数と型を宣言する．WebAssembly は関数の実行終了時のオペランドスタックの状態と result で宣言された型が一致するか検証する．例えば，図 1 の 3 行目にある (result i32) は関数実行終了時に i32 型の値 1 つがオペランドスタックに残っていることを表す．関数は名付けることができる．図 1 において$add1・$main がそれぞれ関数を名付けている．

2.1.3 型宣言

WebAssembly の関数は param・result によって関数の入力値と返り値の型と個数を宣言する．param には名前をつけることができ，関数の中で名前で参照することができる．例えば，図 1 の 4 行目では$x と名付けた引数を参照している．

2.1.4 直接呼び出し

WebAssembly の関数はモジュール内に宣言された順にインデックスが割り当てられる．関数を呼び出すときはこのインデックスを指定するか関数につけた名前を指定して図 1 の 10 行目のように直接呼び出すことができる．この例では名前によって呼び出す関数を指定している．

2.1.5 間接呼び出し

関数はテーブルという構造に登録することができる．テーブルは関数参照のベクターである．図 1 の 1 行目でテーブルを作成し，2 行目で関数$main をテーブルのインデックス 0 に登録している．13，14 行目のようにテーブルでのインデックス 0 を指定して call_indirect を実行するで目的の関数を動的に呼び出すことができる．call_indirect で宣言されている type は呼び出す関数の型を指定している．この例では i32 型の値を 1 つ受け取り，i32 型の値を 1 つ返す型を指定している．指定した型と実際に登録されていた関数の型が異なっていた場合，ランタイムエラーが発生する．

2.1.6 ヒープメモリ

WebAssembly のヒープメモリは，バイト単位の番地を用いて基本データ型の値を読み書きできる．メモリアドレスは i32 型の値によって表現される．このメモリは 1 ページを 64KB として伸張させることができる．ガベージコレクションは現在提供されていないが将来的にサポートされる可能性がある．[1]

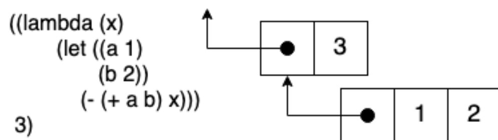

図 2: Scheme プログラムとフレームの対応

2.2 Call/cc

Call-with-current-continuation(`call/cc`) とは，Scheme で継続を取り出す関数である．`call/cc` は 1 引数関数 1 つを受け取る．`call/cc` は呼び出されるとその時点での継続を生成し，その継続を引数として受け取った関数を実行する．`call/cc` で呼び出した関数の実行中に継続が呼び出されたとき，関数の実行を終了し，継続への引数を `call/cc` の返り値として継続を実行する．

2.3 Scheme の環境

Scheme では手続きが呼び出されたときや let 式が評価されたときなどに変数が束縛される．プログラム実行中のある時点における有効な束縛の集合は環境と呼ばれる．即ち，新たに変数を束縛することは環境を拡張することと言える．

3 コンパイラの設計

この節では，現在のコンパイラの設計について述べる．

3.1 フレーム

本コンパイラでは Scheme の環境をフレームと呼ぶ構造のリストとして表現する．フレームは親フレームへのアドレスと複数の変数の値の配列によって構成され，WebAssembly ヒープメモリに格納する．変数の値の配列の順番は変数が束縛された順番と一致する．図 2 のように，新たに変数が束縛される毎にフレームを構築し，変数の値の配列を格納する．以上の説明から分かる通り，フレームは変数名を管理しない．

3.2 変数の値の管理

Scheme はブロック構造を持つ言語であるため，束縛の有効範囲に注意しなければならない．変数の値はフレームのエントリか，WebAssembly のローカル変数に格納する．WebAssembly のローカル変数に格納される場合については 3.4.1 で述べる．フレームは変数名を管理しないため，変数を参照するときは現在のフレームからの相対位置を指定することによって行う．

3.3 Scheme データと WebAssembly コードの対応

すべての Scheme データは WebAssembly の i32 型の値を用いて表現する．下位 1,2bit をタグとして，固定長整数と組み込み定数を埋め込み表現する．それ以外のデータはヒープメモリに格納し，そのアドレスを i32 型に埋め込む．

下位 2bit が 10 の場合

真偽値や null などの定数に番号を付け，上位 30bit にその番号を埋め込む．

下位 1bit が 1 の場合

固定長整数に対応する．上位 31bit で整数を表す．

下位 2bit が 00 の場合

上位 30bit に WebAssembly ヒープメモリのアドレスを埋め込む．アドレス先にデータの内容を格納する．アドレスが指す手前の 4byte はメタ情報で，そのうち 8bit をデータ型を表すタグに割り当てる．以下にヒープメモリに格納されるデータの表現方法を示す．

数値

32bit 長以上の整数，浮動小数点数

クロージャ

Scheme の第一級関数値である procedure に対応する．クロージャを関数本体と環境の組として表現する．具体的には，関数のインデックスを表す i32 型の値と，フレームのアドレスを表す i32 型の値の組である．

cons セル

先頭 4byte で car のデータを，続く 4byte で cdr のデータを格納する．

上記で述べた以外のデータ型は表現方法を検討中である．

3.4　式のコンパイル

図 3 に Scheme プログラムのコンパイル例を示す.

3.4.1　変数参照

`(lambda (y) (+ x y))` のコンパイル結果は図 3(b) において `$lambda2` に対応する．この式において `x` は 1 段上のフレームの 0 番目に格納されているため，51-54 行目のようにコンパイルする．`y` はこの式内からのみ参照されるため，55 行目のようにローカル変数の参照としてコンパイルする．

3.4.2　lambda 式のコンパイル

Scheme の lambda 式は WebAssembly の関数に対応させる．その際に関数の第一引数にフレームを受け取るようにすることで，lambda 式が定義されたときの環境を取得できるようにする．WebAssembly では関数が第一級値ではないため，関数を返り値にすることができない．そのため，関数を返したい場合はクロージャを返すように実装する．また，WebAssembly の関数はネストして定義することができない．そのため，プログラム中の各 lambda をそれぞれ独立に WebAssembly モジュールのトップレベル関数としてコンパイルする．WebAssembly のテーブルを利用することで，各関数のインデックスと本体を関連付ける．

3.4.3　If

WebAssembly の if 命令にコンパイルする．Scheme では `#f` 以外のすべての値が真として扱われるが，WebAssembly も `i32.const 0` 以外の値は真として扱われる．そこで条件式の実行後，独自に実装したランタイム関数 `not_false` を呼び，条件式の返り値が true ならば `(i32.const 1)` に，false ならば `(i32.const 0)` に変換する．

3.4.4　Begin

最後の式以外のコンパイル結果のあとに `drop` を挿入する．`drop` はオペランドスタックの先頭にある値を捨てる命令である．これは関数の実行終了時のスタックに積まれている要素が関数自身が期待するものと異なるとランタイムエラーを発生させるためである．このため，`set!` のように返り値が規定されていない式であっても undefined を表現する定数を 1 つだけ返すように設計する．

```
1 (let ((f (lambda (x)
2           (lambda (y) (+ x y)))))
3    ((f 1) 2))
```

(a) 入力コード

```
1  (module
2     (;import runtime functions ;)
3     (import "skismer"
4             "frame_get"
5             (func $frame_get (param i32
                   i32 i32) (result i32)))
6     ...
7     (import "skismer" "table" (table
          $table 100 anyfunc))
8     (elem (i32.const 28) $lambda1
          $lambda2)
9     (export "memory" (memory 0))
10    (func (export "main") (result i32)
11       (i32.const 123)
12       (local.get $fp)
13       (global.get $fp)
14       (call $lambda0))
15    (func $lambda0 (param $fp i32) (
          result i32)
16       (local $f i32)
17       (local $anorm0 i32)
18       (; extend frame here ;)
19       (i32.const 28)
20       (local.get $fp)
21       (call $make_closure)
22       (local.set $f)
23       (local.get $f)
24       (call $closure_get_env)
25       (i32.const 1)
26       (call $make_i32)
27       (local.get $f)
28       (call $closure_get_func_id)
29       (call_indirect (type ($type1)))
30       (local.set $anorm0)
31       (local.get $anorm0)
32       (call $closure_get_env)
33       (i32.const 2)
34       (call $make_i32)
35       (local.get $anorm0)
36       (call $closure_get_func_id)
37       (call_indirect (type $type1)))
38    (func $lambda1 (param $fp i32) (
          param $x i32) (result i32)
39       (; extend frame here ;)
40       (local.get $x)
41       (local.get $fp)
42       (i32.const 0)
43       (i32.const 0)
44       (call $frame_set)
45       (i32.const 29)
46       (local.get $fp)
47       (call $make_closure))
48    (func $lambda2 (param $fp i32) (
          param $y i32) (result i32)
49       (; extend frame here ;)
50       (i32.const 11111111)
51       (local.get $fp)
52       (i32.const 1)
53       (i32.const 0)
54       (call $frame_get)
55       (local.get $y)
56       (call $add))
57    (func $initialize
58       (; initialize runtime ;)))
```

(b) 出力される WebAssembly コード

図 3: コンパイルの例

```
1    (f a (lambda (r) (k r)))
```

(a) uncps 前のプログラム

```
1    (let ((r (f a)))
2      (k r))
```

(b) uncps 後のプログラム

3.4.5 Call/cc

call/cc を実装するには，一般的に CPS 変換を行うか，コールスタックを操作するかの 2 通りがある．WebAssembly では第一級継続を持たず，コールスタックを操作することができないため，WebAssembly で call/cc をサポートするには元のプログラムに CPS 変換を施すことになる．現在の実装では実験的に CPS 変換によって call/cc をサポートしているが，実行時に WebAssembly のコールスタックの上限にかかってしまう．

4 最適化

この節では，本コンパイラが出力するコードを高速化するために行った最適化を述べる．

4.1 Uncps

Uncps の目的は，CPS 変換したことによって生成された lambda の個数を減らすことで，実行速度を向上させることである．uncps は手続き呼び出し式において，演算子が組み込み手続きかつ被演算子の最後の要素が CPS 変換によって生成された lambda の場合に適用できる．

Uncps は 2 段階で行う．はじめに CPS の手続き呼び出し式を，継続式を演算子とする手続き呼び出し式に変換する．その後，継続式が lambda であった場合に更に等価な let 式に変換する．

4.2 Let の平坦化

let の平坦化の目的は，フレームの拡張を必要最低限で行うようにすることである．この最適化を行う前にアルファ変換を行い，変数のシャドーイングを取り除く必要がある．$lambda-ir とは，この変換によって生成される独自の形式である．構文は ($lambda-ir

```
1    (lambda (f)
2      (let ((a 1) (b 2))
3        (f a b)))
```

(a) let の平坦化前のプログラム

```
1    ($lambda-ir (f) (a b)
2       (begin
3           (set! a 1)
4           (set! b 2)
5           (f a b)))
```

(b) let の平坦化後のプログラム

$\langle args \rangle \langle variables \rangle \langle body \rangle$) である．$\langle args \rangle$ は lambda の引数，$\langle variables \rangle$ は変換前の lambda の本体中の let 式で束縛される変数名のリストである．let で行われる束縛はすべて set!式による代入で置換される．この形式に変換することで，フレームの拡張は$lambda-ir1 個につき一回だけ行われるようになり，フレームの拡張に関連するオーバーヘッドを減少させることが期待できる．

4.2.1 変換の手順

let の平坦化は 2 つの段階を経て行われる．

1. プログラムの起点から lambda を探索する．見つかった場合，その lambda 式の本体を探索する．
2. let 式が見つかった場合，その let で束縛される変数名のリストを取得する．let による束縛をすべて set!による代入に置換する．
3. $lambda-ir 式を生成し，$\langle args \rangle$ に lambda の引数，$\langle variables \rangle$ に取得した変数名のリスト，*body* に lambda 式の本体を束縛する．

4.3 ローカル変数化

ローカル変数化の目的は，変数参照の際にフレームの代わりに WebAssembly のローカル変数を利用することでパフォーマンスを向上させることである．フレームを介する変数参照はメモリアクセスを伴うためパフォーマンスが悪化する．この最適化は変数のコンパイル時に，その変数がスコープ内の lambda から参照されないときに適用できる．この最適化を適用すると，図 6 のように変数の参照はフレーム経由ではなく WebAssembly のローカル変数として変数名で直接参照するようになる．変換手順は以下の通りである．

```
1    (func $lambda0 (param $fp i32) (
         result i32)
2        (local.get $fp)
3        (i32.const 0)
4        (i32.const 0)
5        (call $frame_get))
```

(a) フレームによる変数参照の例

```
1    (func $lambda0 (param $fp i32) (param
         $x i32) (result i32)
2        (local $a i32)
3        (local.get $a))
```

図 6: WebAssembly のローカル変数による変数参照の例

1. 変数のコンパイル時，その変数の内側のスコープの lambda または$lambda-ir から参照されているか検査する．
2. 参照されていない場合，その変数を (local $var i32) としてコンパイルする．

4.4 間接呼び出しから直接呼び出しへの変更

この最適化の目的は手続き呼び出しのコストを削減することである．手続きが組み込みのものである

```
1    (local.get $f)
2    (call $closure_get_env)
3    (; args here;)
4    (local.get $f)
5    (call $closure_get_func_id)
6    (call_indirect (type $f))
```

(a) 間接呼び出しを利用している WebAssembly コード

```
1    (local.get $f)
2    (call $closure_get_env)
3    (; args here;)
4    (call $f)
```

(b) 直接呼び出しを利用している WebAssembly コード

図 7: 関数呼び出し方式の最適化

```
1    (let ((f +))
2      (f 1 2))
```

図 8: WebAssembly 上で直接呼び出しに変換できない例

```
1    (letrec ((ack (lambda (m n)
2                    (if (= m 0)
3                        (+ n 1)
4                        (if (= n 0)
5                            (ack (- m 1) 1)
6                            (ack (- m 1)
                                 (ack m (- n
                                 1))))))))
7        (ack 3 7))
```

図 9: Scheme のベンチマークプログラム

```
1    int ack(int m, int n) {
2        if (m == 0) {
3            return n + 1;
4        } else if (n = 0) {
5            return ack(m - 1, 1);
6        } else {
7            return ack(m - 1, ack(m, n -
                 1));
8        }
9    }
10
11   int main() {
12       ack(3, 7);
13       return 0;
14   }
```

図 10: C のベンチマークプログラム

か，変数に割り当てられている場合に適用できる．Scheme では図 8 のように組み込み手続きを変数に割り当てるプログラムも書けるため，組み込み手続きにも間接参照できるよう関数インデックスを振る必要がある．ただし，組み込み手続きが直接呼び出されている場合はコンパイル時に組み込み手続きであることが簡単にわかるため，その場合に直接呼び出し方式でコンパイルする．

5 最適化の効果測定

前節で今回実装した最適化の内容を述べた．この章では，各最適化がどれほどの実行速度向上に繋がるかを計測する．

5.1 ベンチマークの実行方法

ベンチマークランナーは Node.js で作成した．ベンチマークランナーは対象の WebAssembly コードをロードし，初期化を済ませた後にベンチマーク対象プログラムを複数回呼び出す．この回数は Node.js による JIT コンパイルがいつ行われるかに依存する．測定値が安定してから再びベンチマークプログラムを 100 回呼び出し，平均値を測定値とした．

5.2 ベンチマークプログラム

図 9・図 10 に今回使用したベンチマークプログラムを示す．ベンチマークプログラムには，アッカーマン関数 Ack(3, 7) の実行を選定した．時間の都合により，他のベンチマークプログラムでの計測はできていない．また，現在の実装では CPS 変換を施すと uncps を施してもアッカーマン関数の実行は WebAssembly のコールスタックの上限に到達してしまうため，uncps については計測の対象外とした．

5.3 ベンチマークの計測方法

WebAssembly コードの実行時間の測定には Node.js の `process.hrtime()` を利用した．この関数は呼び出されると現在時刻をナノ秒単位で取得する．CPU のクロックの分解能は 1000ns である．計測の対象時間は，ベンチマークランナーがコンパイルされた図 9 の WebAssembly コードを呼び出してから値が返ってくるまでの時間を対象とした．この計測を各最適化を施した WebAssembly コードに対して行った．比較対象として，C で同等の動作を行うプログラムも計測した．

5.4 実行環境

コンパイルした WebAssembly コードの実行環境は Node.js 15.14.0 である．C のベンチマークプログラムのコンパイルには Apple clang version 12.0.0 (clang-1200.0.32.29) を利用した．

実行マシンの CPU は Intel Core i5-8210Y 1.6 GHz デュアルコア, RAM は 16 GB 2133 MHz LPDDR3, OS は mac OS Big Sur である．今回実行したベンチマークプログラムを図 9・図 10 に示す．

表 1: ベンチマークの実行結果

bench	平均 [ms]	標準偏差
wasm(no opt)	406.25	43.760
wasm(var)	289.57	25.575
wasm(flatten)	401.01	42.666
wasm(direct)	383.96	24.049
wasm(Full)	250.86	15.527
C	3.1080	5.523

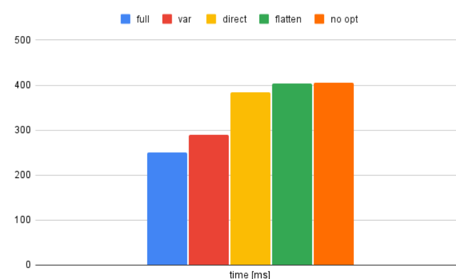

図 11: WebAssembly コードのベンチマーク実行結果のグラフ

5.5 実行結果

表 1 に計測結果を示す．それぞれ () の中に適用した最適化を記している．var はローカル変数化を，flatten は let の平坦化を，direct は関数の直接呼び出し化を，Full はすべての最適化を適用したものを表している．また，図 11 に WebAssembly コードのベンチマークプログラムの実行結果のグラフを示す．青棒が全ての最適化，赤棒がローカル変数化，黄棒が関数の直接呼び出し化，緑棒が let flatten に対応している．オレンジ色の棒は最適化をかけていないベンチマークプログラムの実行結果である．

let の平坦化については，今回のベンチマークプログラムの性質上最適化の恩恵を受けられない結果になった．

最も高速化に寄与した最適化はローカル変数化であった．関数のインデックスは WebAssembly のヒープメモリに存在するので，関数のインデックスを得る処理はヒープメモリにアクセスする処理である．一

方，フレームを介する変数参照もヒープメモリにアクセスする処理である．そこで最適化によって削減されたメモリアクセスの回数を調べる．Ack(3, 7) の場合，図 9 の 2 行目が実行される回数が 692,852 回，３行目が実行される回数が 346476 回，4 行目が実行される回数が 347,489 回，５行目が実行される回数が 1013 回，6 行目が実行される回数が 346,475 回である．関数呼び出し一回に付き一回，関数のインデックスを得る処理によるメモリアクセスが発生するので，関数の直接呼び出し化によって削減されたメモリアクセスの回数は 2,426,342 回である．同様にローカル変数化によって削減されたメモリアクセスの回数は 3,812,244 回である． よって，ローカル変数化が関数の直接呼び出し化より約 1.57 倍メモリアクセス回数を削減している．一方，この 2 つの最適化は，最適化をしていないプログラムと比べて関数の直接呼び出し化が 22.29ms，ローカル変数化が 116.68ms と，ローカル変数化の方が約 5.23 倍実行時間の短縮効果がある．削減されたメモリアクセスの 1 回の処理にかかる時間がローカル変数化と関数の直接呼び出し化で等しいと仮定すると，ローカル変数化ではメモリアクセスの削減と別に約 81.85ms の実行時間が短縮できたことになる．これは WebAssembly のローカル変数が仮想レジスタとして機能するため，レジスタに関する最適化が WebAssembly の仮想機械によってなされたと考えられる．

C の実行結果と比べて，Full で約 80 倍遅い結果になった．これは，関数の呼び出しが発生するたびにフレームの拡張によって起こるメモリアクセスが発生していることや，今回実装した四則演算が浮動小数点数の足し算にも対応するために，値の型の確認の処理が追加されていることなどが原因である．

6 今後の研究

6.1 末尾呼び出し最適化の実装

3.4.5 で述べたとおり，現在の実装では CPS 変換を施したプログラムをコンパイルし実行すると，WebAssembly のコールスタックの上限にかかってしまう．これは末尾呼び出し最適化を行っていないためである．今後はトランポリンによる末尾呼び出し最適化の実現を検討する．

6.2 Wasm/k を利用した call/cc の実装

本コンパイラでは，WebAssembly がコールスタックを操作する命令を提供していないために `call/cc` を CPS 変換によって実験的に実装している．ただし，5.2 で述べたように，現在の実装では CPS 変換を施すと uncps の最適化を行ったとしても WebAssembly のコールスタックの上限に到達してしまう．この対策として，トランポリンを用いてコールスタックの使用を抑えることを検討する．併せて，WebAssembly コードに第一級継続を実装した wasm/k [6] をコンパイルの対象言語にすることを検討する．`call/cc` を wasm/k の提供するコントロールオペレータによって実装し，コンパイル後の WebAssembly コードの実行時間，コードサイズなどが本コンパイラの現状の実装と比べどう変わるかなどについて議論する．wasm/k では，継続テーブルという独自の構造に継続を保存する．`call/cc` を wasm/k で実装した場合に，この継続テーブルに継続を保存する処理，継続テーブルから継続を復元する処理にどれほどのコストがかかるかは今後の実験で調査する．

7 関連研究

7.1 Schism

schism [5] は，Scheme から WebAssembly へのコンパイラである．schism はセルフコンパイルできることを念頭に設計されたコンパイラである．現在の設計では文字列は Cons セルを利用したリストで表現されている．セルフコンパイルには適している設計であるが，メモリを大量に消費する課題がある．schism では初期化で巨大なメモリを用意しておくことで対処しているが，今後はより効率的な設計を模索すると述べられている．

7.2 Kawa

Kawa [3] は，Scheme を含む様々な動的型付き言語を Java プラットフォーム上で動作するようにするフレームワークである．Kawa では，Scheme の `call/cc` を Java の例外処理を利用して実装している．

ただし，この方法では取得した継続を最大 1 回までしか呼び出せないという制限が伴うため，何回でも呼び出せる継続を完全にはサポートしていない．今後我々がコンパイルのターゲットにする wasm/k では限定継続命令が提供されているため，継続の呼び出し回数に制限をかけることなく `call/cc` を実装することができる．

8 まとめ

本研究では，基本的な最適化を数個実装した Scheme から WebAssembly へのコンパイラを作成し，最適化の効果を測定した．ベンチマークの結果，Scheme の変数を WebAssembly のローカル変数で表現する最適化が最も実行速度の向上につながった．WebAssembly のローカル変数を利用することで，レジスタに関わる最適化が WebAssembly の仮想機械によって行われたと考えられる．

参 考 文 献

[1] : GC Proposal for WebAssembly, `https://github.com/WebAssembly/gc`.

[2] : WebAssembly Core Specification.

[3] Bothner, P. and Terrace, C.: Kawa: compiling scheme to java, 1998.

[4] Ecma, E.: 335: Common Language Infrastructure, *ECMA (European Association for Standardizing Information and Communication Systems)*, (2012).

[5] Holk, E.: Schism: A Self-Hosting Scheme to WebAssembly Compiler, EasyChair Preprint no. 500, EasyChair, 2018.

[6] Pinckney, D., Guha, A., and Brun, Y.: Wasm/k: Delimited Continuations for WebAssembly, *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2020, New York, NY, USA, Association for Computing Machinery, 2020, pp. 16–28.

[7] Tim Lindholm, Frank Yellin, G. B. A. B. D. S.: The Java® Virtual Machine Specification Java SE 12 Edition, `https://docs.oracle.com/javase/specs/jvms/se12/html/index.html`.