2022 年度 修士論文

仮想機械語を用いた第一級継続の実現方法とその比較

指導教員 増原英彦

東京工業大学　情報理工学院

数理・計算科学系　数理・計算科学コース

20M30002 荒井滉平

2022 年 2 月 28 日提出

概要

第一級継続とは，プログラムのある時点における残りの計算を表す手続きである継続を第一級値である継続オブジェクトして扱う言語機能である．Scheme 言語ではこの継続オブジェクトを call-with-current-continuation(call/cc) 関数を用いて作成，利用することができる．第一級継続を有する高級言語から仮想機械語にコンパイルする場合，この第一級継続は継続渡し方式 (CPS) に変換する，仮想機械の拡張命令によりサポートするなどによって実現することができる．仮想機械語の１つである WebAssembly(wasm) にコンパイルする場合，WebAssembly の仮想機械を拡張し継続命令を導入した Wasm/k という研究もある．しかし Wasm/k のように仮想機械を拡張し第一級継続を実現することにどれほど利点があるのかが不明瞭であった．そこで本研究では仮想機械上において第一級継続を CPS 変換によって実現する方法 (CPS 方式) と Wasm/k の継続命令を用いて実現する方法 (Wasm/k 方式) を実行速度の観点から比較する．この比較を実現するために Scheme 言語のサブセットから WebAssembly にコンパイルし，Wasm/k によって拡張された仮想機械上でも動作する処理系である Skismer を開発した．この処理系では第一級継続を CPS 方式と Wasm/k 方式で実装し，容易に両者の実行速度やコードサイズの比較することができる．この Skismer を用いて，call/cc を利用するものと利用しないベンチマークプログラムを実行した．実際に利用したベンチマーク群を Skismer で call/cc を CPS 方式と Wasm/k 方式でコンパイルし実行することで実行方式の違いによる性能差を測定した．この結果，call/cc を使ったベンチマークでは Wasm/k 方式のほうが約 2 倍から約 6 倍遅くなり，使わないベンチマークでは CPS 方式の方が約 2 倍から約 5 倍遅いという結果を得た．これはそれぞれ Wasm/k の継続命令の実装に用いられているスタックコピー方式はスタックが深くなるほど継続の利用にオーバーヘッドがかかるため，末尾呼び出し最適化を適用できなかったためである．今後の展望として，Wasm/k の実装方式の改良によるパフォーマンスの向上，および WebAssembly のプロポーザルによってより Scheme の仕様に準拠した処理系，及び効率的な第一級継続の実現の可能性について述べる．

謝辞

この研究は増原先生と叢先生の多大なサポートのもと行うことができました．修士課程から研究室に入ったこともあり，研究テーマも環境も全て 1 から始まったものの，新型コロナウイルスの影響で大学に通うこともできず，対面で雑談をすることもできず，しまいには完全に精神的限界を迎えてしまった私に真正面から向き合ってくださった先生方には感謝してもしきれません．また共に輪講会に参加したり，オンラインでも楽しめるよう様々な宴会を企画してくださった研究室の皆様にも感謝いたします．

コロナ禍で対面で会う機会は片手で数えるほどしかありませんでしたが，この研究室で様々な研究に触れ，様々な経験を得る事ができました．改めて 2 年間という短い間でしたが，本当にお世話になりました．

目次

1 はじめに

現在では数多くのプログラミング言語が存在するが，近年では仮想機械を利用して実現している言語が普及している．仮想機械を利用することで，様々なハードウェアをサポートすることが容易になる，その仮想機械に特化したプロファイル情報を用いた実行時最適化や実行時コンパイルを行うことで直接ネイティブコードを生成するより高効率に実行できる可能性がある．また，最適化を仮想機械上に実現した言語に共通して適用できる利点もある．

プログラミング言語の機能の中でも高度なものの一つに第一級継続がある．第一級継続は，プログラムのある時点における残りの計算を表す継続を第一級のオブジェクトとして扱うものである．第一級継続を有する言語に Scheme[18] や Standard ML of New Jersey[3] などがある．これらの言語では，call-with-current-continuation(call/cc) という手続きを使うことで継続オブジェクトの作成，利用を行うことができる．継続オブジェクトは 1 引数の手続きのように振る舞い，呼び出されると現在の計算を捨てて与えられた引数を返り値として call/cc から戻り継続の計算を行う．第一級継続を利用することで，例外機構やコルーチンのような疑似スレッドなどのプログラムの大域制御機能を言語処理系がサポートすることなく実装することができる．

この継続オブジェクトを実装するには，コールスタックを直接操作して実装する方法やコード変換を経由して実現する方法がある．また，実行環境が例外機構などの大域制御を提供している場合，それを用いて動作の一部を実現することもできる．ただしこの大域制御利用する方法では一部の第一級継続の動作のみ実現できる．

ここで，第一級継続を持つ高級言語を仮想機械語にコンパイルすることを考える．一般に仮想機械語は自身の仮想機械のコールスタックを操作する命令を提供していない．そのため，第一級継続を持つ高級言語を実現する方法には２つある．一つは高級言語の継続を含むプログラムを変換し，仮想機械上で継続を含まない仮想機械語にコンパイルする方法である．もう一つは仮想機械を拡張して継続命令を導入することである [15]．しかし，仮想機械に継続命令を導入する研究はあまり行われておらず，第一級継続を実現するために仮想機械を拡張することにどのような利点があるのかが不明瞭であった．

そこで本研究では，第一級継続を実現する際に仮想機械を拡張することでどれほどの利点があるのかを従来のプログラム変換による実装方法とプログラムの実行速度の観点から比較した．入力言語は Scheme のサブセット，対象言語は WebAssembly とし，Scheme の call/cc を両者の方法で WebAssembly 上で実現する処理系である Skismer を開発した．

この論文は２章で背景について述べた後，３章で課題と提案を述べる．４章では Skismer の設計と WebAssembly における call/cc の実装方法を説明する．５章でベンチマークを行い，その結果と考察を述べる．６章で WebAssembly 上で第一級継続を実現することについての展望を述べる．７章で関連研究について言及した後，８章でまとめを行う．

2 背景

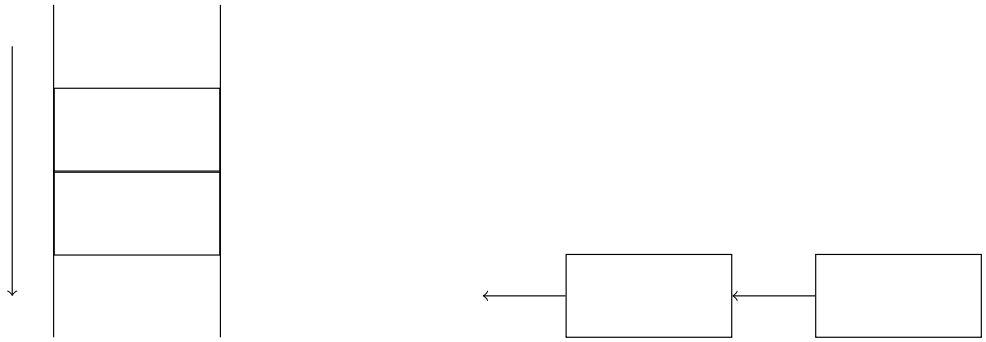
2.1 継続

継続とは，プログラムのある地点の残りの計算のことである．例えば，(+ (+ 1 2) 3) の計算において，(+ 1 2) 時点の継続は (+ [] 3) である．ここで [] は現在実行している (+ 1 2) の計算結果を表す．

継続は変数の値・関数本体の処理終了後の戻り先アドレスなどの情報を含んでいる．これらの情報はコールスタックに含まれているため，継続はコールスタックの状態そのものということができる．

継続はプログラミング言語に普遍的な概念であり，特に Scheme や Standar ML of New Jersey など一部の言語では継続を第一級値として扱うことができる．つまり指定した位置の環境を保存し，あとで評価を再開するすることができる．継続を第一級値とするとコルーチン・例外機構などをプログラマが実装することができる．

2.2 コールスタックの実装別の継続の実装



(a) 配列方式　　(b) フレーム連鎖方式

図2.2.1: コールスタックの実装

継続はコールスタックの状態そのものであるため，継続を第一級値として実装するにはコールスタックがどのように実装されているかが重要である．一般にコールスタックは 2 つの方法で実装される．一つは連続メモリ空間を利用する配列方式，もう一つはスタックフレームを線形リストのように連鎖させるフレーム連鎖方式である．配列方式のモデル図を図 2.2.1aに示す．各四角形はスタックフレーム，矢印の向きはスタックフレームが積まれる方向を表す．フレーム連鎖方式のモデル図を図 2.2.3に示す．矢印の向きはそのフレームの処理終了後に戻る先のフレームを指す．

2.2.1 配列方式

この方式は一般に多くの言語処理系で採用されているコールスタックの実装方法である．この方式では利用を終えたスタックフレームのデータは上書きされる．コード 2.2.1を用いて図 2.2.2に例を示す．最初に 5 行目の func1 の呼び出しで func1 のスタックフレームが作成される．func1 内で func2 を呼び出すことで func1 の次に func2 のスタックフレームが作成される（図 2.2.2b）．func2 の実行終了後，フレームは破棄されるがデータはその場に残る (図 2.2.2c)．続いて func1 内で func3 が呼ばれると，func2 のフレームは解放されているのでその位置を上書きするように func3 のフレームが作成される（図 2.2.2d）．従って，配列方式で func2 を実行中の継続を再開したい場合は何らかの方法でこの時点のスタック全体を保存しておく必要がある．一般的には継続を保存するときはヒープに保存し，継続を再開するときにヒープからコールスタックに復元するという操作が行われる．

配列方式で継続を実現する場合，継続の処理時にコールスタックのコピー，ヒープからコールスタックへの復帰などが発生するため，対象とする継続のサイズに比例し継続処理のオーバーヘッドが大きくなる．

コード 2.2.1: example.scm

```
1 (define (func1)
2   (func2)
3   (func3))
4
5 (func1)
```

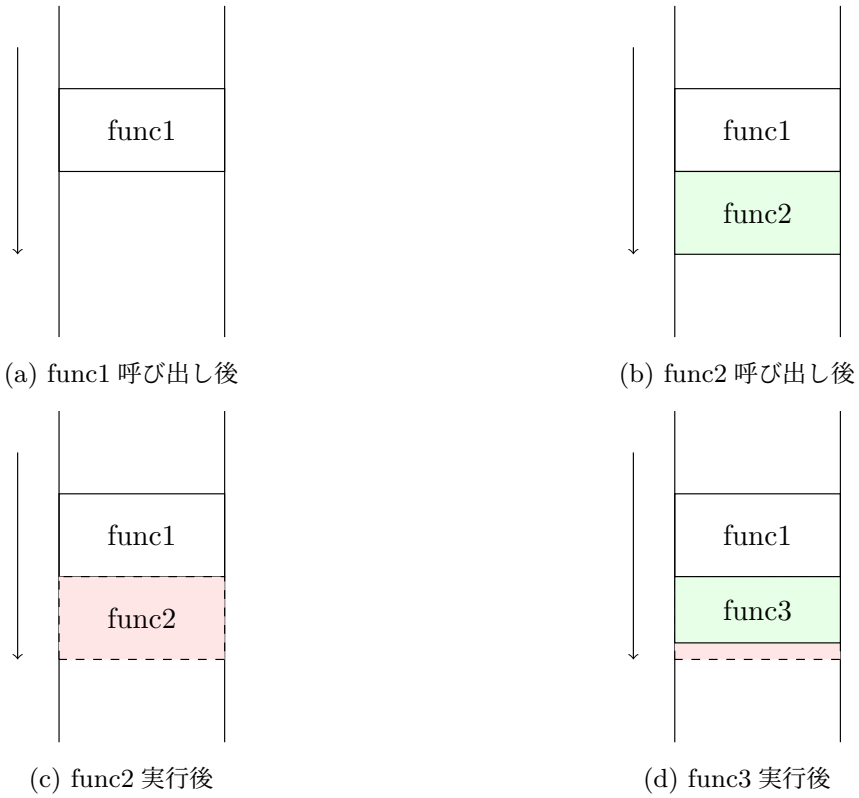
2.2.2 フレーム連鎖方式

フレーム連鎖方式は Standard ML of New Jersey などの言語で採用されているコールスタック実装方式である．この方式では利用を終えたフレームがガベージコレクタに回収されるまで残り続ける．同じくコード 2.2.1を用いて図 2.2.3に例を示す．

func1 の呼び出し以降，func2 の実行後までは配列方式と同様である．しかし func3 を呼び出した時，func3 用のフレームが新たにアロケートされ，func2 のフレームが上書きされることはない．従ってフレーム連鎖方式で func2 呼び出し時点の継続を再開したい場合は単に func2 のフレームに再移動[*1]し，処理を再開するだけでよい．

フレーム連鎖方式では継続の処理は配列方式に比べ非常に軽量に行うことができるが，使われることがなくなったフレームを回収するコストが発生する．フレームは関数を呼び出すと生成されるため，結果として関数呼び出し全体にコストが掛かる．

[*1] 継続の処理を再開できるようにスタックポインタやプログラムカウンタなどを書き換えること



(a) func1 呼び出し後　(b) func2 呼び出し後

(c) func2 実行後　(d) func3 実行後

図2.2.2: 配列方式のコールスタックのフレームの変遷

2.3 Call/cc

コード 2.3.2: call/cc のイディオム

```
1 (call/cc (lambda (k) body))
```

Scheme や Standard ML of New Jersey は継続を第一級オブジェクトとして定義している．これらの言語では，call-with-current-continuation (call/cc) という手続きを使って継続を取り扱うことができる．コード 2.3.3に call/cc を用いて (+ (+ 1 2) 3) の継続を扱う例を示す．call/cc は 1 引数関数を受け取り，call/cc が呼び出された時点の継続を作成し，その関数 proc に渡して実行する．この継続は「エスケープ手続き」と呼ばれ，1 引数関数のように振る舞い，その引数は call/cc 関数の返り値のように扱われる．proc 内でエスケープ手続きが呼び出された場合，call/cc の「返り値」としてエスケープ手続きに渡された引数を返す．proc 内でエスケープ手続きが呼び出されなかった場合，通常の関数と同様に返り値を返す．コード 2.3.3では k がエスケープ手続きとして，(+ [] 3) を表す．call/cc が呼び出した関数内では k に (+ 1 2) を渡して呼び出しているので，call/cc の返り値として (+ 1 2) の計算結果である 3 が返る．k の内容は計算結果に 3 を足すことなので，その後 (+ 3 3) が行われ全体の計算結果は 6 になる．

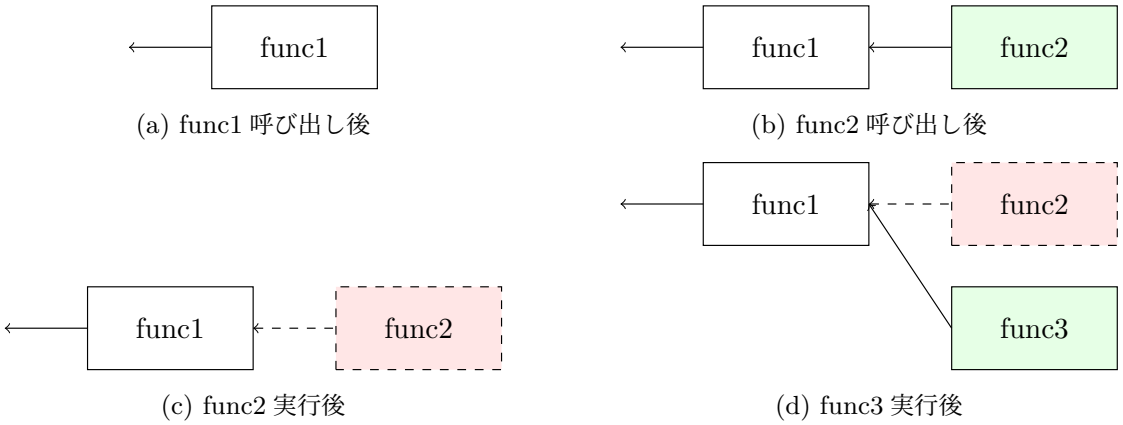


図2.2.3: フレーム連鎖方式のコールスタックのフレームの変遷

コード 2.3.3: call/cc の簡単な利用例

```
1 (+ (call/cc (lambda (k) (k (+ 1 2)))) 3)
2 ;;
3 ;; k: (lambda (result) (+ result 3))
4 ;; (k (+ 1 2))
5 ;; (k 3)
6 ;; (+ 3 3)
7 ;; => 6
```

Call/cc が呼び出した関数内で渡した継続が呼び出された場合，call/cc の呼び出しから直ちに脱出し，継続に渡された値を call/cc の返り値とする．コード 2.3.4に call/cc を用いて整数リストの積を求めるプログラムの例を示す．この例では，product が受け取ったリストに 0 が含まれていた場合，渡された継続に引数として 0 を与えて呼ぶことでそれまでのリスト要素の積を計算することなく直ちに関数を終了し 0 を返す．継続が呼ばれることなく関数の実行が終了した場合，その関数の返り値がそのまま call/cc の返り値として以降の計算が続行される．このプログラムのように，call/cc は継続を暗黙的に取得し，与えられた関数の引数 return に束縛している．

Call/cc で扱う継続は一度作成したものを何度でも呼び出すことができる．call/cc を利用することで例外機構・バックトラック探索・ジェネレータ・スレッドなどの機能を実装することができる．

コード 2.3.4: 継続を利用した整数リストの積を求めるプログラム例

```
1 (define (product lst)
2   (call/cc (lambda (return)
3     (let loop ([lst lst])
4       (if (null? lst)
5         1
```

```
6           (if (= 0 (car lst))
7             (return 0)
8             (* (car lst) (loop (cdr lst)))))))))
9
10
11  (product '(1 2 3))
12  (product '(1 0 2))
```

2.4 Call/cc の実装

Call/cc で扱う継続は第一級値なのでコールスタックの実装方法によって call/cc の実装方法が異なる．

2.4.1 配列方式

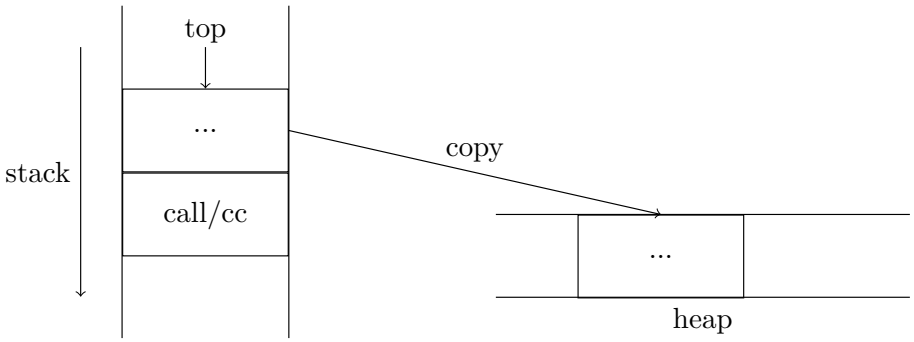


図2.4.4: 配列方式の call/cc 呼び出し時の動作

配列方式でコールスタックを実装している場合はスタックコピー方式が第一級継続を実現する単純な方法の 1 つである．`call/cc` が呼び出された時点のコールスタックを継続オブジェクトとして全てヒープに保存し（図 2.4.4），エスケープ手続きに継続を渡してを実行する．継続が呼ばれた場合，その時点のコールスタックを全て捨てヒープに保存していたコールスタックを復元する．

この方法では継続の作成・呼び出し両方に継続作成時のスタックの大きさに比例したオーバーヘッドが発生する．ただし，通常の関数呼び出しには影響を及ぼさない．

2.4.2 配列方式の最適化

単純なスタックコピー方式を用いた継続の実装方法では，call/cc を呼び出す度にスタックのコピーを行う．しかし call/cc が呼び出した関数の中で継続を呼び出さない場合にオーバーヘッドとなる．また，ネストした call/cc 呼び出しをすると同じ範囲を含むコールスタックを複数回コピーしてしまうため非効率である．

そのため，先行研究でスタックのコピーを必要になるまで行わないよう最適化する方法が提案さ

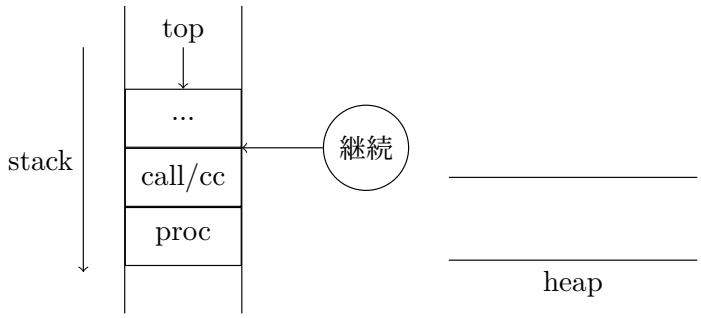


図2.4.5: 最適化された継続呼び出し.

れている [19]．提案手法では call/cc を呼び出した時点ではコピーを行わず，現在のスタックポインタの位置を記録する．

提案手法では Call/cc が呼び出した関数内で継続が呼び出された場合は従来の方法と同じ動作をする．しかし関数内で継続の参照[*2]が行われなかった場合はそのままスタックのコピーを行わないまま call/cc を終了し呼び出し元に制御を戻す．このようにすることで不必要なスタックのコピーを取り除くことができる．また，ネストした call/cc を呼び出している状況で継続を呼び出した場合は内側の call/cc が作成した継続に外側の call/cc が作成した継続が含まれるため，スタックの内容を共有することができる．そのため同じ内容のスタックのコピーを作成することを防ぐことができ，メモリ使用量とスタックコピーにかかるオーバーヘッドを低減することができる．

2.4.3 フレーム連鎖方式

コールスタックが線形リストで実装されている場合，スタックの操作によってデータが上書きされないためコピーを行う必要がない．継続の作成は call/cc 呼び出し時のスタックポインタを保存し，継続の呼び出し時はスタックポインタを継続作成時のものに置き換えて実行を再開するだけで完了する．第一級関数を用いてスタックを消費せずにフレーム連鎖方式と同等の動作を実現する方法に CPS 変換がある

この方式では第一級継続の実装が容易になる利点があるが，使用済みのスタックがメモリに残ってしまうため全ての関数呼び出しにオーバーヘッドが発生する．配列方式ではスタックフレームの作成・解放がレジスタの加減算を一回行うことで可能である．しかしフレーム連鎖方式ではフレームの作成は高速な方法ではレジスタの加算一回で実現できるが，フレームの解放にはガベージコレクタを動作させるためレジスタの減算一回以上の処理が必要になってしまうためである．

2.5 CPS 変換

継続渡し形式 (Continuation Passing Style/CPS)[16] とは，関数から返る代わりに継続を表す関数に返り値を渡して実行するように書くプログラミングスタイルである．CPS で書かれたプロ

[*2] ここで継続の参照とは，継続に引数を与えて呼び出す・継続を返り値とする．ある変数に継続を代入することを指す.

グラムでは継続が陽に表される．プログラムを CPS に書き換える変換のことを CPS 変換という．CPS では全ての関数は新たな引数として継続を表す関数を受け取る．この引数は慣例的に k と表される．関数が第一級値である言語ではこの継続はクロージャとして表される．このクロージャを渡し，実行していくさまはリスト方式でコールスタックを構築してるものとみなすことができる．よって，コールスタックが配列方式で実装されている言語であっても CPS 変換を行うことでフレーム連鎖方式のコールスタックの動作を実現することができる．

CPS で書かれたプログラムの具体例を示す．コード 2.5.5で定義された factorial，fact-aux 関数は CPS 変換されたコード 2.5.6では新たな引数 k を受け取る．このプログラム例では factorial-cps を呼び出すときに，トップレベルの継続として計算結果をそのまま返す関数を与えている．ここで，=&，-&，*& は CPS に対応したプリミティブであり，2 つの引数に加え継続を受け取る．

コード 2.5.5: CPS 変換前の factorial 関数の例

```
(define factorial
  (lambda (n) (fact-aux n 1)))
(define fact-aux
  (lambda (n acc)
    (if (= n 1)
      acc
      (fact-aux (- n 1) (* acc n)))))

(factorial 5) ;;=> 120
```

コード 2.5.6: CPS 変換後の factorial 関数の例

```
(define factorial-cps
  (lambda (n k) (fact-cps-aux n 1 k)))
(define fact-cps-aux
  (lambda (n acc k)
    (=& n 1 (lambda (b)
              (if b
                (k acc)
                (-& n 1 (lambda (n1)
                  (*& n acc (lambda (nk)
                    (fact-cps-aux n1 nk k)))))))))

(factorial-cps 5 (lambda (k) k)) ;;=> 120
```

CPS 変換をすることでコールスタックを操作する命令を直接扱えない言語でも call/cc を実装することができる．ここでは，CPS における call/cc を call/cc-cps と呼ぶことにする．call/cc-cps は新たに継続 cont を受け取り，call/cc-cps に渡される関数 f も新たな引数 k' を受け取る．call/cc-cps が受け取る cont は継続を表す 1 引数関数である．k' は f の実行結果を渡す先の 1 引数関数である．ここで，CPS 変換する前の f の引数 k は call/cc が作成する継続を受け取ることと，k' は

call/cc-cps の継続と同一であるから，k と k' は同じ継続を受け取る．従って，call/cc-cps は受け取った関数 f に引数として cont を 2 個渡して実行する関数となる．

コード 2.5.7: CPS 変換による call/cc の実装

```
;; before CPS conversion
(define call/cc
  (lambda (f)
    (f CONT))) ;; CONT is the continuation captured by call/cc
;; how to use
(call/cc (lambda (k) (k 10)))

;; CPS conversion version
(define call/cc-cps
  (lambda (f k)
    (f k k)))
(call/cc-cps (lambda (k k') (k 10)) cont)
```

この方法を採用している Scheme 処理系に Chicken Scheme[4] がある．Chicken Scheme は入力ソースの Scheme プログラムに CPS 変換を施し，最適化をかけた後に C プログラムにコンパイルする．

2.5.1 CPS 変換後の最適化

CPS 変換後のプログラムはクロージャの呼び出しを大量に行うことになる．クロージャの呼び出しはクロージャ変数を参照するためにメモリアクセスが生じる，高階関数として呼び出された場合は関数を間接的に呼び出す必要があるなどのオーバーヘッドがある．しかし，CPS 変換後のプログラムは静的解析を行うことが，最適化を適用することが容易でありこれらのオーバーヘッドを減らすことができる [2]．例えば，継続関数を可能な限りインライン展開することで余分な関数呼び出しを減らすことができる．また，クロージャ変換を行うことでコンパイルの対象言語上で関数呼び出しを間接呼び出しから直接呼び出しに戻すことを可能にし，クロージャ変数を局所変数に変換することができる．これにより CPS 変換したことで生じる関数呼び出しのオーバーヘッドを大幅に削減することができる．

さらに，CPS 変換したプログラムでは関数から返る代わりに次の関数を呼び出すため，再び同じフレームに帰ってくることがない．すなわち，継続関数を呼び出したあとは呼び出し元の関数のスタックフレームはガベージコレクタによって回収可能である．そのため，メモリアロケータにリニアアロケーションを行うものを採用し，ガベージコレクタに世代別コピー GC を採用することで効率よく実行することができる [3]．これはコピー GC が生存しているデータが少ないほど効率よく実行することができるガベージコレクタだからである．

2.6 Wasm/k

Wasm/k[15] は WebAssembly に継続オブジェクトを作成する際に範囲を限定する継続 (delimited continuation) に関する命令を拡張実装した研究である．この研究の動機は，Go などのスレッドを持つ言語から WebAssembly へコンパイルする際に手動でスタックコピーを行っていたものを WebAssembly 仮想機械上で行う事によって，より効率的にスレッドを実現することである．Wasm/k では `(control h)` と `(restore v)` 命令を限定継続命令として追加している．

2.6.1 WebAssembly

WebAssembly[9] は，W3C によって開発されているコンパクトで効率的に実行でき，安全で移植性に優れるよう設計されたスタックマシン上で動作する仮想機械語である．ブラウザ上で JavaScript を補完しながら Web アプリケーションのパフォーマンスを高める目的で設計された．現在では WebAssembly の堅牢性・移植性を活かし，Wasmtime[6] や WasmEdge[12] などのブラウザ外でも実行できる環境が存在している．

WebAssembly はごく一部の命令を除き，すべての命令に型がついている．型検査を行うことで WebAssembly コードを実行する前にある程度正しく動作することを保証することができる．型検査は関数の動的呼び出し時にも行われる．

2.6.2 仕様

データ型

WebAssembly は `i32`，`i64`，`f32`，`f64` の 4 つの値型のみを提供する．それぞれ 32/64 ビット整数・32/64 ビットの IEEE754-20196[1] に準拠した浮動小数点を表す．特に 32 ビット整数は命令によって 2 の補数で表現される符号付き整数・符号なし整数・真偽値・メモリアドレスに解釈される．

モジュール

WebAssembly はモジュールを単位としてインスタンスを作成する．モジュール内にはグローバル変数・関数・関数テーブル・メモリが含まれる．

関数

WebAssembly の関数はパラメータを受け取り，コード実行後のオペランドスタックに残っていた値が返り値となる．可変長のパラメータを持つ関数を定義することはできない．関数同士は再帰的に呼ぶことが可能である．また，仮想レジスタとして機能するローカル変数を定義することもできる．

関数はモジュール内に定義された順番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，関数に直接つけたラベルを用いて `call` 命令で呼び出すことができる．関数がテーブルに登録されている場合，そのテーブル上のインデックスを指定して `call_indirect` 命令で間接的に呼び出すこともできる．関数テーブルを参照する

処理が入るため，call_indirect による関数呼び出しは call による関数呼び出しよりコストがかかる．

WebAssembly の関数は第一級オブジェクトではないため，返り値として関数を返すことはできない．この動作を模倣する場合，呼び出したい関数のインデックスを返し，呼び出し元でそのインデックスを元に間接的に呼び出す必要がある．

コード 2.6.8: 関数を直接呼び出しする wasm コードの例

```
(func $add (param $lhs i32) (param $rhs i32) (result i32)
  (local.get $lhs)
  (local.get $rhs)
  (i32.add))

(func $main (result i32)
  (i32.const 1)
  (i32.const 2)
  (call $add))
```

コード 2.6.9: 関数を間接呼び出しする wasm コードの例

```
(table 1 anyfunc)
(elem (i32.const 0) $add)
(type $op (func (param i32 i32) (result i32)))
(func $add (param $lhs i32) (param $rhs i32) (result i32)
  (local.get $lhs)
  (local.get $rhs)
  (i32.add))

(func $main (result i32)
  (i32.const 1)
  (i32.const 2)
  (i32.const 0) ;; index of $add at table
  (call_indirect (type $op)))
```

メモリ

WebAssembly のヒープメモリは 0 ベースでインデックスされた線形バイト配列である．そのため，ロード/ストアするデータの解釈はそのバイト列を利用する命令によって異なる．1 ページ 64KB 単位とし，最大で 65535 ページまでメモリを持つことができる．メモリはコード実行中に動的に拡張できるが，縮小させることはできない．現在の仕様ではモジュールに付き 1 つのメモリのみを定義することができる．

コントロールフロー

if や block,loop といった基本的で局所的なコントロールフローを提供している．一方現在の WebAssembly は例外やスレッドなどの非局所的なコントロールフローに関する命令を

提供していない．

2.6.3 Wasm/k の継続命令

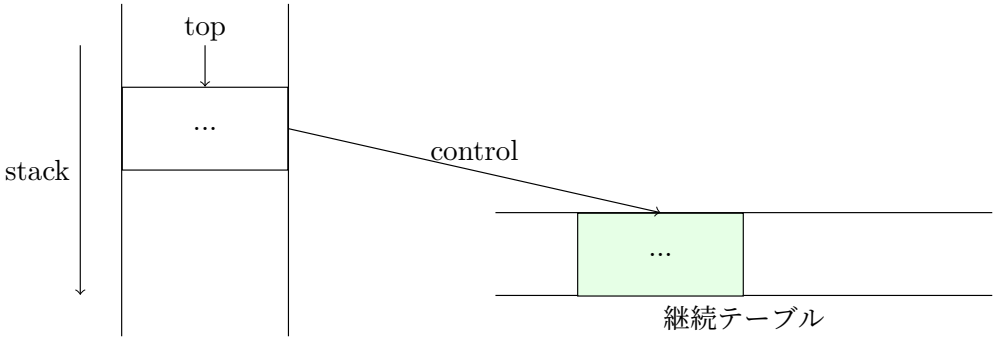
Wasm/k の提供する継続命令は単純なスタックコピー型で実装されている．各命令の動作モデル図を図 2.6.6に示す．

control　実行時点のコールスタックを取得し，それを Wasm/k 独自の継続テーブルというヒープに保存する．その後，それに対応するインデックスである i64 型の値を作成し h に渡して実行する命令である．このインデックスは暗黙的に作成されるため外部から指定することはできない．h を呼び出す際，任意の i64 型の値を 1 つ渡すことができる．これは h 内で関数を動的に呼び出したい場合，そのインデックスを外部から渡す必要があるからである．control で呼び出した関数は後述する restore 命令によって帰ることを要求される．そのため，h は返り値を持たない関数として定義しなければならない．

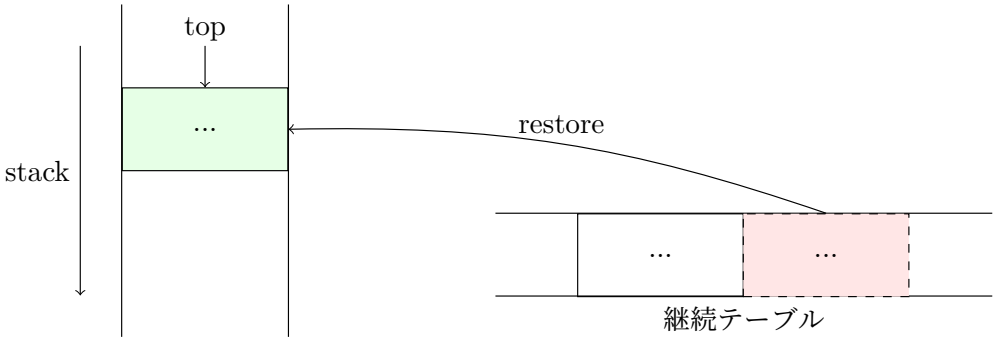
restore　指定された i64 型の値に対応する継続オブジェクトをコールスタックに復帰させ，その後継続テーブルから削除する命令である．第一級継続では同じ継続オブジェクトを複数回呼び出せるが，これを実現するためには後述する continuation_copy 命令を用いて継続オブジェクトを複製し，それを呼び出すことで可能になる．

continuation_copy　指定された i64 型の値に紐付けられた継続テーブル上の継続オブジェクトを複製し，それに対応する新たな i64 型の値を返す．第一級継続を実現するために導入された命令である．restore 命令を呼び出す前に continuation_copy を呼び出すことで一度作成した継続オブジェクトを何度も呼び出すことができるようになる．

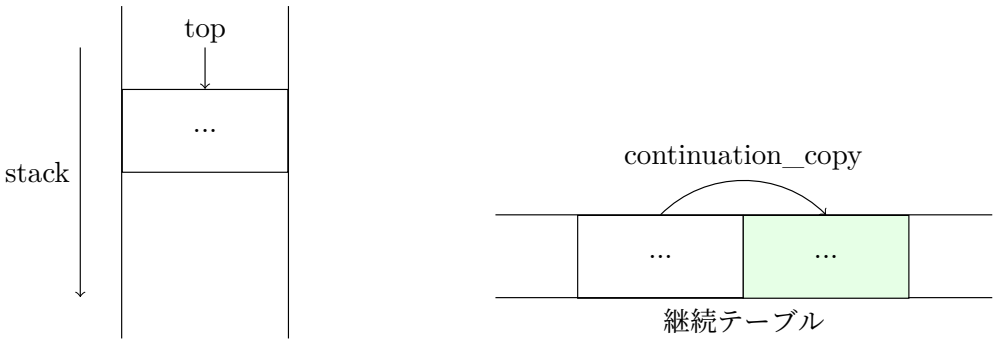
continuation_delete　指定された i64 型の値に紐付けられた継続テーブル上の継続オブジェクトを破棄する．



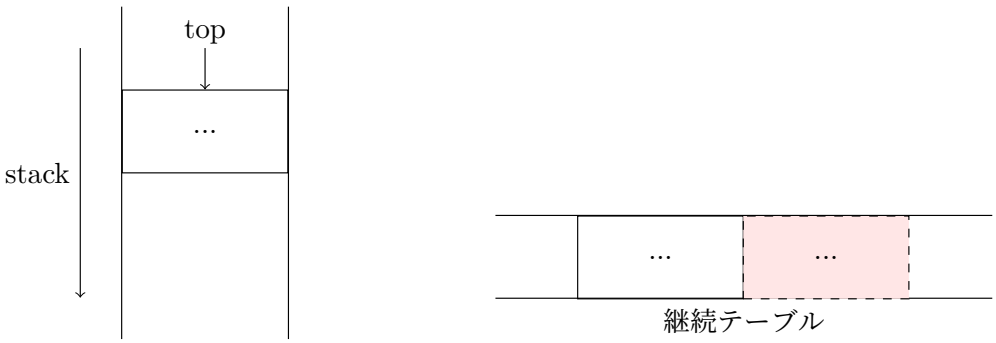
(a) control の呼び出し



(b) restore の実行



(c) continuation_copy の実行



(d) continuation_delete の実行

図2.6.6: Wasm/k 命令の動作．緑のフレームは新規作成されたもの，赤いフレームは削除されたものを表す

3　研究の課題と方針

第一級継続を持つ言語から仮想機械語にコンパイルすること考える．一般に，仮想機械語は自身の仮想機械のコールスタックを操作できる命令を提供していない．そのため，仮想機械上で第一級継続を実現するには一般的には入力プログラムを CPS 変換してからコンパイルする必要がある．一方で，Wasm/k のように仮想機械にコールスタックの操作を抽象化した継続命令を導入する研究もある．しかし，Wasm/k のように仮想機械を拡張して継続を実現する研究はあまり行われていない．そのため，第一級継続を実装する際に，仮想機械を拡張する方法が CPS 変換により実現する方法と比較して最適化などを含めてよりよいパフォーマンスを出すことができるのか不明であった．また，Wasm/k で継続命令が導入された動機も第一級継続を実現することではないため，この目的の比較は行われていない．

そのため，仮想機械を拡張する方法による第一級継続の実現の利点を明らかにするため，実行速度の観点から両者を比較する．本研究では Scheme 言語のサブセットから WebAssembly にコンパイルし，call/cc を CPS 変換を利用して実装する方法と Wasm/k の提供する継続命令を用いて実装する方法でコンパイルするコンパイラである Skismer を開発し，実際にコンパイルしたコードをベンチマークを用いて計測する．

4 Skismer

この章では 3 章で述べた課題を調査するために開発した Skismer の設計について述べる．

4.1 Skismer の概要

Skismer は Scheme のサブセットから WebAssembly へのコンパイラである．Scheme の基本的な構文やデータ構造の一部，call/cc をサポートしている．call/cc は Wasm/k の提供する継続命令を利用して実現する「Wasm/k 方式」と，CPS 変換を利用して実現する「CPS 方式」の 2 方式でコンパイルすることができる．生成された WebAssembly コードは単体で実行可能な形式で出力される．

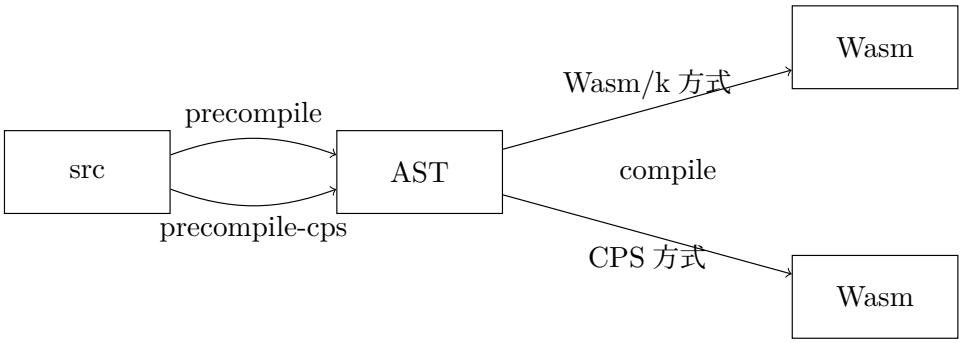


図4.1.7: Skismer のコンパイルフロー

Skismer では図 4.1.7の流れでコンパイルを行う．入力プログラムはまず precompil フェーズで各変換及び最適化が行われる．CPS 方式で call/cc を実装する場合は図 4.1.7の precompile-cps の流れで CPS 変換が行われる．precompile フェーズを経た compile フェーズで，precompile フェーズで出力された AST に対しコンパイルを行い WebAssembly ファイルを出力する．出力された WebAssembly ファイルは WebAssembly 実行環境上で読み込み，エントリーポイントである main 関数を呼び出すことで実行することができる．

4.2 Scheme データと WebAssembly コードの対応

4.2.1 静的スコープ

Scheme 言語は静的スコープを持つ言語である．従って，すべての変数は定義された位置によって値が決定する．Let 式の評価時や lambda 式の呼び出し時などで新たな変数束縛が発生する度に新たなスコープが作成される．Skismer ではこのスコープをフレームと呼ぶ構造のリストで表現する．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 ヒープメモリ上に格納する．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 4.2.8に例を示す．このモデルではルートフレームがトップレベルのスコープを表し，親フレームとして常

に自分自身を指す．Let 式などで変数 v_1, v_2 を定義するとフレームが作成され，そのフレームの親にルートフレームを指定する．図 4.2.8の v_1, v_2 の要素にはそれぞれ v_1, v_2 の値が格納される．

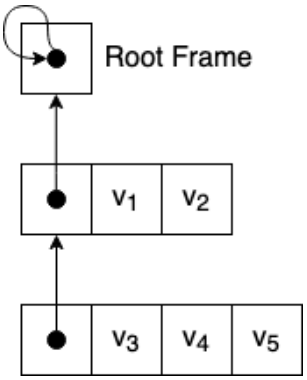


図4.2.8: フレームのモデル図

一方，WebAssembly の関数ではローカル変数を定義することができる．このローカル変数は仮想レジスタとして動作するため高速である．そこで，コンパイル時に自由変数でないことがわかる変数の値は WebAssembly のローカル変数に割り当て，自由変数はフレームのエントリに割り当てるよう実装した．このため，全ての関数はフレームを表す引数 `$fp` を必ず第一引数に受け取り，自由変数を参照する際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よう実装した．

4.2.2 Scheme データの WebAssembly 上の表現

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 1,2 ビットをタグとして，ビットの組み合わせによって残りのビットで表現するデータの内容を示す．

下位 2bit が 10 の場合

上位 30 ビットで真偽値や null などの定数に対応する番号を埋め込む．

下位 1bit が 1 の場合

固定長整数に対応する．上位 31bit で符号付き整数を表す．

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 4 バイトはメタ情報で，そのうち 8 ビット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

32 ビット長以上の整数及び浮動小数点数

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．

4.3 precompile フェーズ

この節では実際に precomple フェーズで行っている各種変換，最適化について述べる．

4.3.1 parse-exp

この段階では，入力ソースを 0 引数の lambda でラップしたコードをパースし，AST に変換する．この理由は，WebAssembly を実行する際にエントリーポイントとなる main 関数を作るためである．AST の定義は付録 Aを参照されたい．

4.3.2 cps-convert

call/cc を CPS 方式で実装する場合に行う．2 章でも述べたとおり，CPS 変換で実装した call/cc は通常の関数とみなせるためプログラムが call/cc を含むか含まないかに依らず統一的に変換を行うことができる．CPS 変換を行うコードは付録 Aを参照されたい．

4.3.3 UnCPS

CPS 変換を行うと組み込み関数の呼び出しも CPS に対応したものに直す必要がある．一方組み込み関数自体の内部では call/cc 呼び出しによる継続呼び出しが発生しない．そのため組み込み関数は直接呼び出しに直すことができる．この最適化では組み込み関数を CPS から直接呼び出しに直すことによって組み込み関数呼び出しのオーバーヘッドを低減する．

4.3.4 expand-letrec

コード 4.3.10: letrec 展開前のプログラム

```
1 (letrec ([loop
2             (lambda (n)
3               (if (= n 0)
4                 #t
5                 (loop (- n 1))))])
6   (loop 10))
```

コード 4.3.11: letrec 展開後のプログラム

```
1 (let ([loop 'undefined])
2   (begin
3     (set! loop (lambda (n)
4                  (if (= n 0)
5                    #t
6                    (loop (- n 1)))))
7     (loop 10)))
```

図4.3.9: letrec の展開例

この変換では，`letrec` を全て `let` と `set!` を使ったプログラムに変換する．これは `letrec` を用いた再帰関数の定義を Skismer が正しくコンパイルするために行う．図 4.3.9に変換例を示す．

4.3.5 アルファ変換

アルファ変換は変数の名前を変更する変換である．アルファ変換によって全ての変数にはユニークな名前が与えられる．これは後に行う Let の平坦化を実現するために不可欠なことである．

4.3.6 A 正規化

コード 4.3.12: A 正規化の例

```
1  ;; before
2  ((lambda (x) x) 10)
3
4  ;; after
5  (let ([anorm0 (lambda (x) x)])
6    (anorm0 10))
```

A 正規化はプログラムを A 正規形 [8] に変換することである．A 正規形では各計算の途中結果が変数に束縛される．この形に変換することで，`((lambda (x) ...) e)` のように無名関数を定義し即座に呼び出すようなプログラムをコンパイルする処理を簡略化することができる．

4.3.7 Let の平坦化

Let の平坦化はフレームの作成コストを低減するために行う最適化である．この最適化では，lambda 式およびその本体中に出現する let 式をまとめて lambda-IR という独自の構文に変換する．lambda-IR に変換することによって，let 式の評価，lambda 式の呼び出しごとに行われていたフレームの拡張が lambda-IR の呼び出し毎に一回のみ行われるようになるので，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できる．lambda-IR の構文は図 4.3.10のとおりである．

`($lambda-ir` $\langle args \rangle \langle variables \rangle \langle body \rangle$`)`

図4.3.10: lambda-IR の構文

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．この内，以後自由変数として出現しない束縛をすべて `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．

4.3.8 間接呼び出しの直接呼び出し化

実装の単純化のため，全ての関数は間接呼び出しでコンパイルされる．ただし，組み込み関数に関しては Scheme プログラム上で直接呼び出されている場合は事前に呼び出し位置を特定できるため，wasm 関数の直接呼び出しに直すことができる．これにより関数呼び出しのオーバーヘッドを減少させることができる．

コード 4.3.13: 変換前の wasm コード

```
(i32.const DUMMY_FRAME_POINTER)
(i32.const 1)
(call $make_i32)
(i32.const 2)
(call $make_i32)
(i32.const 10) ;; index of $add at table
(call_indirect (type $type2))
```

コード 4.3.14: 変換後の wasm コード

```
(i32.const DUMMY_FRAME_POINTER)
(i32.const 1)
(call $make_i32)
(i32.const 2)
(call $make_i32)
(call $add)
```

4.4 comiple フェーズ

コンパイルフェーズでは 2 段階の処理を行う．

- AST 内の全ての lambda-IR を集め，それぞれをコンパイルする．
- コンパイルされた lambda-IR の情報を元に，実行可能な WebAssembly モジュールを生成する．

lambda-IR を収集する目的は lambda-IR を WebAssembly の関数へとコンパイルし，それを Wasm モジュールに含ませるためである．

4.4.1 データのコンパイル

数値

各数値型に対応した `make_num` 関数を呼び出すことで Skismer のデータ型に対応した数値を生成する．4.2で述べたとおり，`i32` 型以外の数値は全てヒープ上に確保される．

コード 4.4.15: 数値型のコンパイル例

```
    ;; 0
    (i32.const 0)
    (call $make_i32)

    ;; 3.141592
    (f64.const 3.141592)
    (call $make_f64)
```

変数

変数がローカル変数であるかフレーム変数であるかでコンパイル方法が変わる．ローカル変数である場合，他の関数から直接参照されることはないため，WebAssembly のローカル変数としてコンパイルすることができる．

フレーム変数の場合現在のフレームの位置からの相対位置を指定することで変数を参照する．

コード 4.4.16: ローカル変数のコンパイル

```
(func $lambda0 (param $fp i32) ... (result i32)
  (local $x i32) (local $y i32)
  ...
  (local.get $x)
  ...)
```

コード 4.4.17: フレーム変数のコンパイル

```
(func $lambda0 (param $fp i32) ... (result i32)
  ...
  (local.get $fp)
  (i32.const 1)    ;;relative frame address
  (i32.const 0)    ;;index of x at the frame
  (call $frame_get)
  ...)
```

変数に値をセットするときも同様に参照し，`local.get`, `frame_get` 関数の代わりに `local.set`,`frame_set` 関数を用いる．

lambda-IR

lambda-IR は関数本体のコンパイルと，クロージャとしてコンパイルする場合に分かれる．

クロージャの場合　呼び出す関数が登録されている関数テーブル上のインデックスを表す `i32` 型の整数と，評価時の環境を元にクロージャを作成する `make_lambda` 関数を呼び出す. 関数テーブル上のインデックスは precompile フェーズで事前に割り当てられている. `make_lambda` 関数は受け取ったインデックスとフレームポインタを WebAssembly ヒープ上に保存し，そのアドレスを返す.

コード 4.4.18: クロージャの作成

```
(i32.const 10)
(local.get $fp)
(call $make_lambda)
```

関数本体のコンパイル　lambda-IR が呼ばれるとき，新たな変数束縛が発生するのでフレームを拡張する必要がある．そのため関数の呼び出し直後にフレーム拡張の処理を行う．関数の引数と let 変数からフレーム変数の個数を調べ，その個数分の要素を確保したフ

レームを新規作成する．コード 4.4.1では 1215 行目が該当する．フレームの拡張後，各変数の値をセットする．コード 4.4.1では 1618 行目がローカル変数の値のセット，1924 行目がフレーム変数の値のセットを行う命令である．
初期処理が完了した後，本体のコードをコンパイルした結果を並べる．

コード 4.4.19: lambda-IR のコンパイル例

```
;; input code
;; let x is local var, y is frame var.
(lambda-ir (arg) (x y)
  (begin
    (set! x 1)
    (set! y 2)
    bodies ...))

;; compiled code
(func $lambda0 (param $fp i32) (param $arg i32) (result i32)
  (local $x i32)
  (local.get $fp)
  (i32.const 1)
  (call $frame_extend)
  (local.set $fp)
  (i32.const 1)
  (call $make_i32)
  (local.set $x)
  (i32.const 2)
  (call $make_i32)
  (i32.const 0)
  (i32.const 0)
  (local.get $fp)
  (call $frame_set)
  ;; compiled body here ...)
```

関数呼び出し

呼び出す関数が組み込み関数かユーザー定義関数かでコンパイル方法が変わる．

組み込み関数　組み込み関数は事前に定義された対応する関数を直接呼び出すようにコンパイルする．現在実装している組み込み関数は新たに変数を束縛しないのでフレームを拡張する必要はないが，Skismer で関数のコンパイルに関わる処理を共通化するため，組み込み関数も第 1 引数にフレームを受け取るものとしている．そのため，フレームポインタとしてダミー値を渡している．

コード 4.4.20: 組み込み関数の呼び出し例

```
;; (+ 1 2)
(i32.const DUMMY_FRAME)
```

```
3  (i32.const 1)
4  (call $make_i32)
5  (i32.const 2)
6  (call $make_i32)
7  (call $add)
```

ユーザー定義関数　現状の Skismer では全てのユーザー定義関数は間接呼び出しで行われる．関数はクロージャになっているため，クロージャから呼び出す関数の定義時の環境，関数テーブル上のインデックスをそれぞれ `closure_get_env` 関数，`closure_get_func_id` 関数を呼び出すことで取得する．WebAssembly では `call_indirect` を用いて関数を呼び出す際，動的検査のために呼び出す関数の引数と返り値の型を指定しなければならない．Skismer では現状全ての関数が `i32` 型の値を返すため，引数の個数に応じた事前に定義した関数型を指定することで対応する．コード 4.4.1の例では 2 引数関数を呼んでいるので，12 行目で 2 引数の関数型に対応する `$type2` を指定している．

コード 4.4.21: ユーザー定義関数の呼び出し例

```
1  ;; let f is 2-params function.
2  ;; (f 1 2)
3  (local.get $f)
4  (call $closure_get_env)
5  (i32.const 1)
6  (call $make_i32)
7  (i32.const 2)
8  (call $make_i32)
9  (local.get $f)
10 (call $closure_get_func_id)
11 (call_indirect (type $type2))
```

if

if 式は WebAssembly の if 命令に対応させる．条件分岐をする際，WebAssembly ではスタックトップの値をポップし，`i32` 型の 0 を偽，それ以外の値を真として扱う．Skismer では独自の真偽値を定義しているため，Skismer のランタイム関数の `isnot_false` 関数を通して `i32` 型の 0 と 1 に変換する．

コード 4.4.22: if のコンパイル

```
1  (i32.const 10)
2  (call $make_i32)
3  (i32.const 20)
4  (call $make_i32)
5  (call $le)
6  (call $isnot_false)
```

```
(if (result i32)
  (then
    (i32.const 0x0010))  ;; TRUE at skismer
  (else
    (i32.const 0x0110))) ;; FALSE at skismer
```

4.4.2 実行可能 WebAssembly モジュールの生成

全ての関数をコンパイルした後，実行可能 webAssembly モジュールを生成する．この段階では，ユーザー定義関数の関数テーブルへの登録，メモリの作成，ランタイム関数と組み込み関数の追加，エントリーポイントの作成，モジュール初期化時に行う処理の定義を行う．

ユーザー定義関数の関数テーブルへの登録

WebAssembly では `elem` 命令を用いて，i32 型の値と登録したい関数の名前を指定することで関数テーブルへ関数を登録することができる．ここで i32 型の値は関数テーブル上でのインデックスであり，テーブルに登録できる個数以下の任意の数値を指定できる．

コード 4.4.23: 関数テーブルへの登録

```
(table $table 1000)
(elem (i32.const 13) $lambda0)
(elem (i32.const 14) $lambda1)
```

メモリの作成

WebAssembly のメモリは `memory` 命令で定義できる．このメモリには 1 以上の数値を渡し，メモリ作成時に最低限必要なページ数，確保できる最大のページ数を指定することができる．

コード 4.4.24: メモリの定義

```
(memory $memory (i32.const 65535))
```

メモリアロケータの追加

WebAssembly ではアロケートはユーザーが指定して実行しなければならない．Skismer ではリニアアロケーションを行うアロケータを実装し，アロケーション時にアロケータ関数を呼び出す．アロケータはフレームを操作する `frame_init`，`frame_get`，`frame_set` などの関数やクロージャを作成する `make_lambda` などのアロケーションを必要とする関数内にインライン展開されている．

コード 4.4.25: メモリの定義

```
(memory $memory (i32.const 65535))
```

ランタイム関数と組み込み関数の追加　別途作成した組み込み関数，ランタイム関数をモジュールに組み込む．組み込み関数に関しては同時に関数テーブルへの登録も行う．これはコード

4.4.2のような動的な関数呼び出しに対応するためである．ランタイム関数はプログラムから直接呼ばれることも動的に呼ばれることもないため，関数テーブルへの登録は行なわない．

コード 4.4.26: 動的に組み込み関数を呼び出す必要の例

```
(let ([apply-f (lambda (f) (f 1 2))])
  (apply-f +))
```

エントリーポイントの作成

WebAssembly では export 属性を指定した関数を外部から呼び出すことができる．この機能を利用し，main 関数を作成し，export 属性を付与することでコンパイルしたコードを実行できるようにした．main 関数内で呼び出している lambda0 はパースする際にソースコード全体をラップした lambda に対応する．`$init_fp` は後に述べる初期化処理が完了した後に設定されるフレームポインタを表すグローバル変数である．

コード 4.4.27: main 関数

```
(func $main (export "main") (result i32)
  (global.get $init_fp)
  (call $lambda0))
```

初期化処理の作成　WebAssembly では `start` 命令で指定した関数をモジュール作成後に自動的に実行できる．この関数は主に初期化処理に利用される．Skismer ではルートフレームの作成，ビルトイン関数のシンボルのフレームへの登録を行う．拡張後のフレームへのポインタをグローバル変数にセットする．

コード 4.4.28: 初期化処理

```
(func $init
  (local $fp i32)
  (call $frame_init)
  (i32.const 16)
  (call $frame_extend)
  (local.set $fp)
  ;; register $add to frame
  (i32.const 0)
  (local.get $fp)
  (local.get $fp)
  (i32.const 0)
  (i32.const 0)
  (call $frame_set)
  ;; register $sub to frame
  (i32.const 1)
  (local.get $fp)
  (call $make_lambda)
  (local.get $fp)
```

```
    (i32.const 0)
    (i32.const 1)
    (call $frame_set)
    ...  ;; register other builtin functions as well
    (local.get $fp)
    (global.set $init_fp))
```

4.5 call/cc の実装

コード 4.5.29: CPS 方式の call/cc 実装

```
(func $call/cc (param $fp i32) (param $body i32) (param $k i32) (result i32)
          (local.get $body)
          (i32.const CONT_WRAPPER_ID)
          (local.get $k)
          (call $make_lambda)
          (local.get $k)
          (call $closure/2))
(func $closure/2 (param $closure i32) (param $arg1 i32) (param $arg2 i32) (result i32)
          (local.get $closure)
          (call $closure_get_env)
          (local.get $arg1)
          (local.get $arg2)
          (local.get $closure)
          (call $closure_get_func_id)
          (call_indirect (type $type2)))
(func $cont_wrapper (param $closure i32) (param $arg i32) (param i32) (result i32)
          (local.get $closure)
          (call $closure_get_env)
          (local.get $arg)
          (local.get $closure)
          (call $closure_get_func_id)
          (call_indirect (type $type1)))
```

4.5.1 CPS 方式

2.5で述べたとおり，CPS 方式で実装された call/cc は単純な 2 引数関数になる．このとき，継続として渡される関数も 2 引数関数になる．継続が呼び出されたときの関数を動的に呼び出すた

めに $cont_wrapper 関数でクロージャを作成し，$closure/2 の呼び出し時に call/cc の第一引数に渡される関数の第 1 引数として渡す．$closure/2 の第 2 引数が評価されることはないため，$cont_wrapper の第 3 引数は無視される．

4.5.2 Wasm/k 方式

Wasm/k の提供する control 命令で呼び出す関数は restore 命令で帰ってくることを期待しており，通常の WebAssembly 関数として値を返してはならない仕様になっている．(control h) で呼び出される h には第一引数として control が作成し，継続テーブルに保存した継続の id が i64 型で渡される．第 2 引数として任意の i64 型の値を与えることができる．また，restore 命令で呼び出した継続は継続テーブルから消去されてしまうため，複数回呼び出せる第一級継続の性質を保つために継続をコピーしてから呼び出すという操作を加える．加えて，Wasm/k が提供する命令は i64 型で実装されている．Skismer では基本的にすべてのデータは i32 型で実装されているので，Skismer と Wasm/k で処理を行き来する場合にデータの変換処理を行う．

call/cc　実行するクロージャを i64 型に拡張し，それを引数に $callcc_helper を呼び出す．返り値を再び i32 型に変換しその値を返す．

call/cc_helper　受け取ったクロージャを再び i32 型に戻し，Skismer 関数として実行する．この関数に渡す継続は $cont_wrapper 関数の実行とする．

$cont_wrapper　継続が呼び出されたときに実際に実行される関数である．継続に渡す値を i64 型に変換し，継続を呼び出す $copy_and_reset を呼び出す．この関数は実装の都合上 i32 型の値を返すよう実装しているが，実際は copy_and_reset で実行する restore 命令で直接 control に戻る．すると WebAssembly のバリデータによって返り値の方が合わないことを指摘されるため，それを回避するためにダミー値を最後に返すように見せかけている．

$copy_and_reset　呼び出す継続をコピーし，受け取った値を引数に restore を実行する．

4.6 実装されていない機能

Skismer では時間の都合上，文字列やシンボル，マクロ機構やガベージコレクタ機構が実装されていない．また，インライン展開やクロージャ変換などの最適化も CPS 方式では行えていない．これらの機能については今後の課題の 1 つである．

コード 4.5.30: Wasm/k 方式の call/cc 実装

```
(func $call/cc (param $dummy_fp i32) (param $proc i32) (result i32)
        (local.get $proc)
        (i64.extend_i32_u)
        (control $callcc_helper)
        (i32.wrap/i64))

(func $callcc_helper (param $k i64) (param $proc i64)
        (local $proc32 i32)
        (local.get $proc)
        (i32.wrap/i64)
        (local.set $proc32)

        (local.get $k)

        (local.get $proc32)
        (call $closure_get_env)
        (i32.const ,CONT_WRAPPER_ID)
        (local.get $k)
        (i32.wrap/i64)
        (call $make_lambda)
        (local.get $proc32)
        (call $closure_get_func_id)
        (call_indirect (type $type1))

        (i64.extend_i32_u)
        (call $copy_and_reset))

(func $copy_and_reset (param $k i64) (param $value i64)
        (local.get $k)
        continuation_copy
        (local.get $value)
        restore)

(func $cont_wrapper (param $frame i32) (param $value i32) (result i32)
        (local.get $frame) ;;actually cont id
        (i64.extend_i32_u)
        (local.get $value)
        (i64.extend_i32_u)
        (call $copy_and_reset)
        SKISMER_DUMMY_VALUE) ;;dummy return value for type checker
```

5　ベンチマークの実行

この章では実際に Skismer でベンチマークコードをコンパイルし CPS 方式と Wasm/k 方式の実行速度の比較を行う．ベンチマークの目的はまず背景で述べた通りの第一級継続の実装方式による定性的なオーバーヘッドが，仮想機械上では実際にどれほど現れるのかを計測すること，そして結果から仮想機械に継続命令を拡張することの現状の利点を考察することである．

5.1　前提

４章で述べたとおり，Skismer はガベージコレクタ機構を搭載していない．そのため，フレームの回収コストが発生すべき CPS 方式が有利な条件になる．一方でインライン展開やクロージャ変換を行っていないため，CPS 方式で関数の呼び出し回数が増える．そのため CPS 方式は不利な条件も課されている．従って，非常に不公平な条件でベンチマークが行われていることに留意する必要がある．

5.2　実行環境と実行方法

表 5.2.1に実行環境を示す. ベンチマークプログラムはそれぞれ Skismer で CPS 方式と Wasm/k 方式によってコンパイルした．仮想機械の条件を同一にするため，実行環境には Wasm/k によって継続命令が拡張された Wasmtime を採用した．Wasmtime は Rust 言語向けの API を公開しているため，Rust でベンチマークランナーを記述，実行した．このベンチマークランナーは各ベンチーマーク WebAssembly コードをインスタンス化し，10 回事前実行を行い Wasmtime の JIT コンパイラによる最適化を完了させた後に 100 回実行する．

表5.2.1: 実行環境

OS	Mac OS Monterey
CPU	1.6 GHz デュアルコア Intel Core i5
メモリ	16 GB 2133 MHz LPDDR3
Wasmtime	0.2.0(Wasm/k による拡張がされたもの)
Rust	1.53.0

5.3　ベンチマークプログラム

ベンチマークプログラムには call/cc を使用するプログラムとそうでないプログラムを採用した．ただし Skismer では前章で述べたとおり実装されている言語機能が非常に限られているため，現段階では大きなプログラムや複数の言語機能を使うプログラムをベンチマークに用いることが

できない．Call/cc を利用しないベンチマークコードは Scheme のマイクロベンチマーク群である Larceny Benchmarks にあった 53 個のベンチマークのうち，CPS 方式で実行できた 8 個,Wasm/k 方式で実行できたものが 10 個で，両者で実行できた 4 個を採用し性能を比較した．

Call/cc を利用したプログラムのベンチマークには継続の作成・呼び出しに関する基本性能を測るものと，ジェネレータや非決定的計算を行うプログラムを実行した．それぞれのベンチマークプログラムは付録 Bに示す．

5.3.1 Call/ss を利用しないプログラム

このベンチマーク群の目的は CPS 変換を行う事による通常の関数呼び出しへの影響を計測することである．Skismer で CPS 変換されたプログラムは元のプログラムで組み込み関数以外の関数呼び出しにつき一回関数呼び出しが増える．そのため，CPS 方式でコンパイルしたコードは Wasmk 方式でコンパイルしたコードより遅くなることが予想される．

5.3.2 Call/cc を利用するプログラム

Call/cc の基本性能を測るプログラム このベンチマーク群の目的は call/cc を用いた継続の作成・呼び出しの基本性能を計測することである．capture.scm ではループするごとに call/cc を呼び出し継続を作成する．call.scm ではトップレベルで継続を作成し，それをループごとに呼び出す．capture_and_call.scm ではループするごとに継続の作成と呼び出しを行う．

fibgen

このベンチマークは呼ばれるごとに次のフィボナッチ数を返すジェネレータを call/cc を使って定義し，fib(40) の値を求めるまで呼び出す．call/cc の評価をループ関数に合わせて実行するので，ジェネレータが呼び出されるたびにスタックが深い位置で継続オブジェクトが作成，呼び出しされる．したがって，このベンチマークでは継続オブジェクトを作成する際にスタックをコピーする方式を取る処理系では遅くなる．

amb

このベンチマークは非決定演算子 amb を call/cc で実装し，リストの中からピタゴラスの定理を満たす整数の組を求める．このベンチマークでは call/cc が作成した継続オブジェクトが複数回呼び出される．したがって，作成された継続オブジェクトの呼び出しのコストを計測することができる．

5.4 実行結果

5.4.1 Call/cc を利用しないプログラム

実行結果を表 5.4.11，図 5.4.12に示す．グラフの縦軸は実行時間の平均値を表す．全てのベンチマークで CPS 方式が Wasm/k 方式に比べ約 2 倍から 6 倍遅くなった．

図5.4.11: call/cc を利用しないプログラムの実行結果の表

bench	cps[ns]	wasmk [ns]
fib	17904	4346
sum	3157560	626378
tak	3227982	507566
ack	10037376	4849955

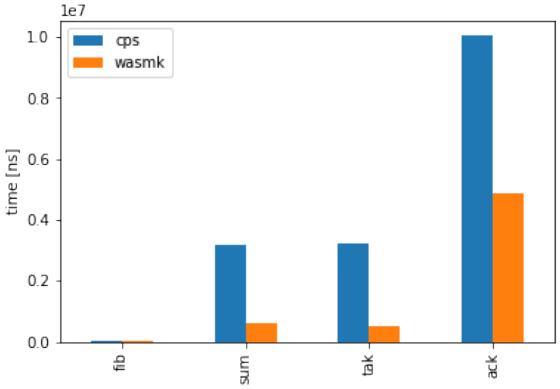


図5.4.12: call/cc を利用しないプログラムの実行結果のグラフ

5.4.2 Call/cc を利用するプログラム

call/cc の基本性能

図 5.4.13aは capture.scm の実行結果，図 5.4.13bは call.scm の実行結果，図 5.4.13cは capture_and_call の実行結果を示す．それぞれ継続の作成，呼び出し，作成と呼び出しに関する基本性能を測定するベンチマークプログラムである．実際のプログラムを B.2に示す．ループ回数は 100 から 400 回までステップ数を 100 として実行した．全てのベンチマークでループ回数に比例して実行時間がかかっていることが見られる．

継続の作成では，Wasm/k 方式が 6 倍以上遅くなった．これは Wasm/k の control 命令がスタックコピー式で実装されているため，継続の作成にスタックの深さに比例したオーバーヘッドがかかるためである．一方，CPS 方式の場合は CPS 変換時に継続の作成は完了しておりオーバーヘッドがかからないため，ループごとに関数を 1 回呼び出すプログラムと同等の速度になる．

継続の呼び出しでは CPS 方式のほうが約 2 倍以上遅くなった．これは継続を取得した時点のスタックが浅く restore 命令のオーバーヘッドが抑えられた一方，CPS 方式では継続の呼び出しでオーバーヘッドが発生したためである．

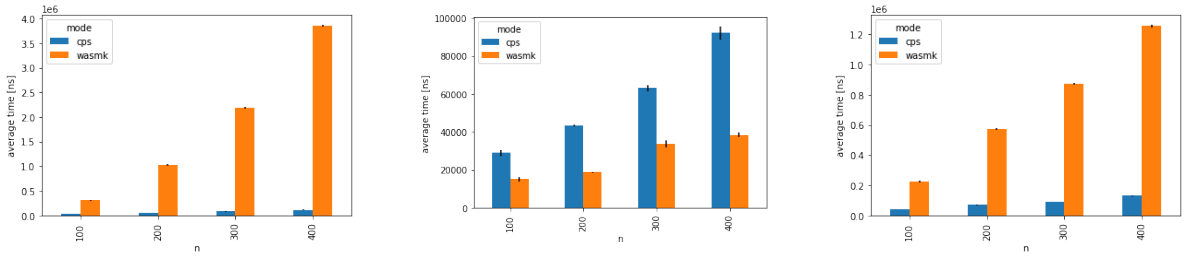
継続の作成と呼び出しでは Wasm/k 方式が 6 倍以上遅くなった．これは継続の作成と同様のオーバーヘッドに加え，呼び出す継続がループするごとに大きくなるため継続呼び出しにもオーバーヘッドがかかったためである．

fibgen

Wasm/k 方式が 6 倍以上遅くなる結果となった．継続の作成と呼び出しに準じた call/cc の利用をしているため，同様のオーバーヘッドがかかっている．

amb

Wasm/k 方式のほうが約 2 倍遅い結果になった．fibgen と比べると CPS 方式と Wasm/k



(a) 継続の作成　(b) 継続の呼び出し　(c) 継続の作成と呼び出し

図5.4.13: call/cc の基本性能の測定結果のグラフ

表5.4.2: call/cc の基本性能の測定結果

n	mode	capture[ns]	call[ns]	capture_and_call[ns]
100	cps	34602	28876	43325
200	cps	61873	43374	71770
300	cps	84567	63136	91260
400	cps	119079	92236	132446
100	wasmk	308819	15002	224810
200	wasmk	1026833	18769	573505
300	wasmk	2188913	33712	870733
400	wasmk	3852785	38419	1254309

方式のパフォーマンスの差が 3 倍以上縮まったが，これは同じ継続オブジェクトを何度も使い回すためである．したがって，amb 関数を呼び出したスタックの深さによって実行パフォーマンスが決定されることになる．call/cc の基本性能の測定で行った継続の呼び出しに関する結果と合わせると，これは control/reset によるスタック操作のオーバーヘッドが非常に大きいことを示している．

5.5 考察

ベンチマーク結果より，call/cc を使わないプログラムでは Wasm/k 方式が，call/cc を使うプログラムでは CPS 方式が高速に実行できる．これには WebAssemblyn が提供する命令による．

call/cc を使わない場合　CPS に変換したプログラムは末尾呼び出しを大量に行うプログラムに変換される．これは関数が値を返す代わりに継続を表す関数に返り値を渡して実行するからである．一般に末尾呼び出しは最適化によって関数呼び出しから jump 命令に置き換えること

図5.4.14: fibgen の実行結果

	cps	wasmk
平均 [ns]	9258	60200
標準偏差	5354	24834
標準誤差	535	2483

図5.4.15: fibgen ベンチマークの実行結果のグラフ

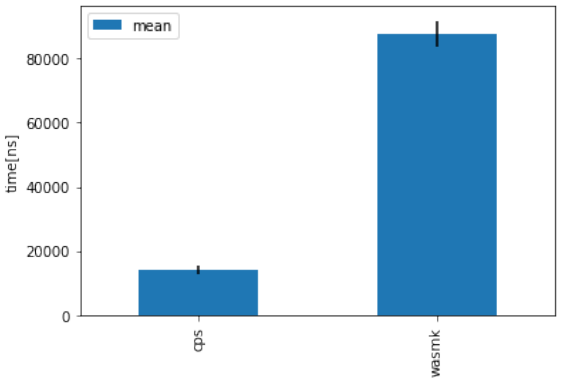
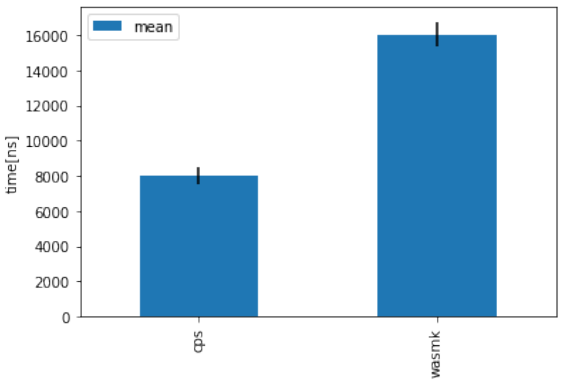


図5.4.16: amb の実行結果

	cps	wasmk
平均 [ns]	7993	16066
標準偏差	5043	7036
標準誤差	504	704

図5.4.17: base ベンチマークの実行結果のグラフ



ができる．Java バイトコード [14] や CIL[7] では jump 命令を提供しているため，CPS 変換したプログラムはこの最適化によって高速なコードにコンパイルできる．しかし 2 章でも述べたとおり，WebAssembly は現在非ローカルな制御を行う命令を提供していない．そのためこの最適化を適用できず，関数呼び出しによるオーバーヘッドの影響を受けてしまった．

call/cc を使う場合　一般に継続を取り扱う場合は CPS 変換したプログラムが高速に扱える．これは継続は事前に CPS 変換によって陽に与えられており，純粋な関数呼び出しとして扱われるためである．一方 Wasm/k 方式では，継続作成時に WebAssembly のスタックをコピーしてヒープに保存する処理が入り，また継続呼び出し時には reset 命令の特性上呼び出す継続のコピーを作成してそれを復帰させるため，実質的に継続呼び出しに通常のスタックコピー方式を採用した実装より 2 倍オーバーヘッドがかかっている．

6　課題と展望

この章ではベンチマークの結果を受けて課題と展望を述べる．

6.1　ベンチマークの拡充

５章で実行したベンチマークでは，call/cc の性能のみを図るのではなく，call/cc の実装方式の違いによって他の処理に影響を及ぼしていないかを図れるようなプログラムを採用している．しかし，提案されたベンチマークだけでは未だ不十分であることに言及する必要がある．例えば，実装方式の違いによって影響を受けることがないとわかっている純粋な四則演算のみで構成された浮動小数点計算や，文字列操作などである．実際に影響を受けていないことを確認するには，同じ計算を行うプログラムの整数版と浮動小数点版を複数用意しそれらを比べることによって目的を達成することができる．現状の Skismer では，浮動小数点の取り扱いに難があるため，今回は採用を見送る結果になった．

また，call/cc を大量に呼び出すが，作成した継続をほとんど使用しない場合をカバーするベンチマークプログラムを採用していない．これは実世界プログラムでこの事例に該当するプログラムを探すことが非常に困難であり，マイクロベンチマークしか作成できないことに由来する．この場合をカバーするベンチマークプログラムの提案は今後の研究の課題の 1 つである．

6.2　Wasm/k の改良

ベンチマーク結果では，Wasm/k 方式を採用した実装では 2 倍以上パフォーマンスが悪くなることがわかった．これは Wasm/k が単純なスタックコピー方式によって継続を実装しているからである．また，同じ継続を何度も呼び出さす際に，継続オブジェクトのコピーを作成し，その継続を利用した後にコピーを削除するという動作を行っており，この操作が継続呼び出しのオーバーヘッドにつながっている．

6.2.1　Wasm/k 命令の振る舞いの変更

これを改善する 1 つの手法は Wasm/k 命令の振る舞いを変更することである．現在は `restore` 命令の実行時に継続オブジェクトを削除している．そのため `continuation_copy` 命令によって対象の継続オブジェクトをコピーする必要があった．しかし `restore` 呼び出し時に継続を削除せずコピーのみ行うようにすることで同じ動作を実現できる．この方法により，継続オブジェクトのコピーと削除にかかるコストを削減することができる．一方でこの手法を用いると `control` 命令が作成した元々の継続オブジェクトが継続テーブルに残ったままになってしまう．そのため，GC を組み合わせることで，オリジナルの継続を参照する変数がゴミとなった場合に `continuation_delete` 命令を呼ぶことで削除する．

6.2.2 継続作成の遅延

また，2.4.2で述べた最適化を Wasm/k の実装に適用することも考えられる．`control` 命令の呼び出し時には継続オブジェクトを作成せず，参照フラグを含むダミーの継続オブジェクトを作成する．`control` が呼び出した関数内で継続が参照されたときに初めて継続オブジェクトを作成するようにする．`restore` は対象の継続オブジェクトの参照フラグを参照し，継続が参照されていないときは `return` 命令と同様に値を返し，継続が参照されているときは継続テーブルからコールスタックを復元する．このようにすることで不必要な継続オブジェクトの作成を防ぎ，オーバーヘッドを低減することができると考えられる．

6.3 末尾呼び出し除去の適用

WebAssembly には jump 命令がないため末尾呼び出し除去を適用することができなかった．しかし，現在 WebAssembly に末尾呼び出しに最適化された関数呼び出し命令である `return_call` と `return_call_indirect` を追加する提案がされている [17]．この提案の動機はまさに末尾呼び出し最適化の要求や CPS のプログラムの最適化要求に応えるためである．この命令を利用することで最適な末尾呼び出しをトランポリンを用いることなく実現することができる．また，Skismer の構成も単純化することができる可能性がある．末尾位置で呼び出している `call_indirect` を `return_call_indirect` に置き換えるだけで済むため，移行も容易である．

7　関連研究

7.1　Scheme から他の仮想機械語へコンパイルするソフトウェア

7.1.1　Kawa

Kawa[5] は動的言語から Java バーチャルマシン (JVM) バイトコードを生成するライブラリで，Scheme にも対応している．Kawa では，Scheme の call/cc を Java の例外機構を用いて実装している．ただし，この方法では継続を一回までしか呼ぶことができない．更に，エスケープのように外側へと向かう継続の呼び出ししか行えないという制限がつく．したがって... のようなコードをコンパイル，実行することができない．

コード 7.1.31: call/cc.java

```
1 (let ([cc (call/cc (lambda (k) (k k)))])
2   (cond
3     ([continuation? cc] (cc 123))
4     ([number? cc] (displayln cc))
5     (else (error "unreachable"))))
```

このプログラムは，変数 cc に call/cc で作成した継続オブジェクトを束縛している．この継続は cc に値を束縛し，let 式の本体を実行することを表す．例外機構を用いて継続オブジェクトを実装した場合，try ブロックが call/cc を囲むように位置する．1 行目の (k k) で 1 回目の継続の呼び出しが行われると，継続オブジェクトはこの try に正しく捕捉される．しかし，cc は継続オブジェクトであるため，cond の 3 行目にマッチし 2 度めの継続呼び出しが行われると，対応する try ブロックが存在しないためトップレベルにまでさかのぼってしまう．継続オブジェクトは例外オブジェクトであるため，結果として Uncaught Error が発生することになる．このように，対応する try ブロックが存在する範囲でしか呼ぶ事ができず，完全な第一級継続を実現しているとは言えない．

7.1.2　IronScheme

IronScheme[13] は Microsoft が開発している.NET フレームワーク上の Scheme 実装である．IronScheme も C# の例外機構を利用して Scheme の継続を実装しているので，Kawa と同様の問題を抱えている．

7.2　Scheme から WebAssembly へのコンパイラ

7.2.1　Schism

Schism[11] 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．そのため，現在の実装では call/cc はサポートされていない．著者はもし call/cc を実装する場合は CPS 変換を介して実装することになるだ

ろうと述べている．

7.3 WasmFX

WasmFX[10] は WebAssembly に型付きの継続を導入するものである．この研究は WebAssembly のマルチスタックプロポーザルの一環で行われている．WasmFX によって導入される継続は WebAssembly の型体型に対応するものであり，Wasm/k で導入された control/reset 命令より汎用性が高い．WasmFX の動機は WebAssembly 上で効率的で安全な非ローカルな制御を行えるようにすることであり，これにより async/await やグリーンスレッド及び第一級継続を実現することが可能になる．

8 まとめ

この論文では，第一級継続を WebAssembly 上で実現する際に入力プログラムを CPS 変換する方法と WebAssembly の仮想機械を拡張する方法のどちらが良いかを，実行速度の観点から比較した．この比較を行うために WebAssembly 上で動作する Scheme 言語処理系である Skismer を開発した．Skismer は call/cc 関数を実装しており，コンパイル時に CPS 方式または Wasm/k 方式を選んでコンパイルする事ができる．Skismer で call/cc を含まないプログラムと継続の利用例に合わせたプログラムを両者の方式でコンパイルし，実行速度を計測した．ベンチマークの結果，call/cc を使わないプログラムでは CPS 方式が約 3 倍から 6 倍遅くなり，call/cc を使ったプログラムでは Wasm/k 方式が 2 倍以上遅くなる結果が得られた．call/cc を使わない場合で CPS 方式が遅くなったのは WebAssembly が jum 命令を提供していないため，末尾呼び出し最適化を適用できなかったためである．call/cc を使った場合で Wasm/k 方式が遅くなったのは，一般的な継続の実装方法によるオーバーヘッドに加え，通常のスタックコピー方式の実装より実質的に継続の作成に 2 倍のオーバーヘッドがかかっているためである．ただし，ベンチマークプログラムはマイクロベンチマークでしか測定できていない．これは現在の Skismer の設計上の制限による．また，Wasm/k の現在の実装方式は改善の余地のあるものであるため，その改善例を示した．最後に将来の WebAssembly に組み込まれるであろう提案を紹介し，その機能によって WebAssembly 上でどのように効率的に第一級継続を実現できるか展望を示した．

付録 A　Skismer のコード

Skismer は Scheme 言語の一種である Racket 言語で作成した.

ast.rkt　eopl ライブラリの提供する `define-datatype` を利用して AST を定義した.

コード 1.0.32: ast.rkt

```
(require eopl)
(define-datatype expression expression?
  (num-exp
    (num number?))
  (bool-exp
    (bool boolean?))
  (var-exp
    (var symbol?))
  (let-exp
    (vars (list-of symbol?))
    (vals (list-of expression?))
    (body expression?))
  (letrec-exp
    (vars (list-of symbol?))
    (vals (list-of expression?))
    (body expression?))
  (lambda-exp
    (args (list-of symbol?))
    (body expression?)
    (label number?))
  ;;expression that merges lambda and let-ir
  (lambda-IR
    (lambda-args (list-of symbol?))
    (let-bound-vars (list-of symbol?))
    (body expression?)
    (label number?))
  (call-exp
    (rator expression?)
    (rand (list-of expression?)))
  (if-exp
    (condition expression?)
    (then expression?)
    (else expression?))
  (block-exp
    (stmt (list-of expression?)))
  (set-exp
    (var symbol?)
```

```
38      (val expression?)))
```

CPS 変換　cps-convert はこれから変換する AST と継続を受け取り，各要素に対応した変換関数に渡して変換を実行する．top-level-cps は CPS 変換を行う際のエントリーポイントで，恒等関数を継続として入力プログラムの CPS 変換を行う．cps-gensym 関数は CPS 変換を行うことで増える関数の引数や関数に対応する一意な変数を生成する．

コード 1.0.33: cps-convert.rkt

```
1  (define cps-convert
2    (lambda (expr cont)
3      (define cnt-if 0)
4      (define cnt-k 0)
5      (define cnt-r 0)
6      (define (cps-gensym sym)
7        (let ((cnt (cond ((eq? sym '$k)
8                          (let ((cnt cnt-k))
9                            (set! cnt-k (+ 1 cnt-k))
10                           cnt))
11                         ((eq? sym '$r)
12                          (let ((cnt cnt-r))
13                            (set! cnt-r (+ 1 cnt-r))
14                            cnt))
15                         ((eq? sym '$if-k)
16                          (let ((cnt cnt-if))
17                            (set! cnt-if (+ 1 cnt-if))
18                            cnt))
19                         (else (error 'cps-gensym "unknown sym:~s\n"
                                    sym)))))
20           (string->symbol
21             (string-append (symbol->string sym) (number->string cnt)))))
22
23     (define (require-convert? expr)
24       (cases expression expr
25              (num-exp (num) #f)
26              (bool-exp (bool) #f)
27              (var-exp (var) #f)
28              (lambda-exp (args body label) #f)
29              (else #t)))
30
31     ;;cps-convert-k
32     ;;-> var (if not required convert)
33     ;;-> converted form (else)
34     ;;k : scheme procedure (i.e. not lambda-exp)
35     (define cps-convert-k
```

```
36        (lambda (expr cont)
37          (cases expression expr
38            (num-exp (num) (cont expr))
39            (bool-exp (bool) (cont expr))
40            (var-exp (var) (cont expr))
41            (let-exp (vars vals body)
42                     (cps-convert-let vars vals body cont))
43            (letrec-exp (vars vals body)
44                        (cps-convert-letrec vars vals body cont))
45            (lambda-IR (args vars body label)
46                       (error 'cps-convert-k "unexpected expr: ~s\n"
                              expr))
47            (if-exp (cond then else)
48                    (cps-convert-if cond then else cont))
49            (lambda-exp (args body label)
50                        (cps-convert-lambda args body label cont))
51            (call-exp (rator rands)
52                      (cps-convert-call rator rands cont))
53            (block-exp (stmts)
54                       (cps-convert-block stmts cont))
55            (set-exp (var val)
56                     (cps-convert-set var val cont))
57            )))
58
59      (define cps-convert-lambda
60        (lambda (args body label cont)
61          (let ((added-k (cps-gensym '$k)))
62            (cont
63              (lambda-exp (append args (list added-k))
64                          (cps-convert-k body
65                                         (lambda (x) (apply-cont
                                              added-k x)))
66                          0)))))
67      ;; let
68      ;; if all vals are simple case, just apply cont to body
69      ;; else convert not simple val
70      (define cps-convert-let
71        (lambda (vars vals body cont)
72          (let loop ((vals vals)
73                     (acc null))
74            (if (null? vals)
75              (let-exp
76                vars
77                acc
78                (cps-convert-k body cont))
```

```
79              (cps-convert-k
80                (car vals)
81                (lambda (x)
82                  (loop (cdr vals)
83                        (append acc (list x)))))))))
84
85      (define cps-convert-letrec
86        (lambda (vars vals body cont)
87          (let loop ((vals vals)
88                     (acc null))
89            (if (null? vals)
90              (letrec-exp
91                vars
92                acc
93                (cps-convert-k body cont))
94              (cps-convert-k
95                (car vals)
96                (lambda (x)
97                  (loop (cdr vals)
98                        (append acc (list x)))))))))
99
100     (define (cps-convert-if cond then else cont)
101         (cps-convert-k
102           cond
103           (lambda (x)
104             (let* ((if-k (cps-gensym '$if-k))
105                    (if-cont (lambda-exp (list if-k) (cont (var-exp
                           if-k)) 0)))
106               (let-exp
107                 (list if-k)
108                 (list if-cont)
109                 (if-exp x
110                         (cps-convert-k then (lambda (x) (call-exp
                              (var-exp if-k) (list x))))
111                         (cps-convert-k else (lambda (x) (call-exp
                              (var-exp if-k) (list x)))))))))))
112
113     (define (cps-convert-call rator rands cont)
114       (cps-convert-k
115         rator
116         (lambda (x)
117           (let loop ((rands rands)
118                      (acc null))
119             (if (null? rands)
120               (call-exp x (append acc (list
```

```
121                 (let ((arg (cps-gensym '$r)))
122                   (lambda-exp (list arg) (cont (var-exp arg)) 0)))))
123                 (cps-convert-k
124                   (car rands)
125                   (lambda (y)
126                     (loop (cdr rands)
127                           (append acc (list y))))))))))
128
129
130     (define (cps-convert-block stmts cont)
131         (let loop ((stmts stmts)
132                    (acc null))
133           (cond
134             ((null? stmts) (cps-convert-k (var-exp '$null) cont))
135             ((null? (cdr stmts))
136              (cps-convert-k
137                (car stmts)
138                cont))
139             (else (cps-convert-k
140                     (car stmts)
141                     (lambda (y)
142                       (loop (cdr stmts)
143                             (append acc (list y))))))))))
144
145     (define cps-convert-set
146       (lambda (var val cont)
147         (cps-convert-k
148           val
149           (lambda (x)
150               (block-exp
151                 (list (set-exp var x)
152                       (cont (num-exp 987654321))))))))
153
154     (cps-convert-k expr cont)))
155
156 (define (top-level-cps src)
157     (cps-convert src
158       (lambda (x)
159         (lambda-exp '()
160           (call-exp x (list (lambda-exp '($top-level-cont) (var-exp
                '$top-level-cont) 1))) 0))))
```

付録 B　ベンチマークコード

B.1　call/cc を利用しないプログラム

tak

コード 2.1.34: tak.scm

```
(letrec ([tak (lambda (x y z)
                (if (>= y x)
                  z
                  (tak (tak (- x 1) y z)
                       (tak (- y 1) z x)
                       (tak (- z 1) x y))))])
  (tak 10 1 1))
```

ack

コード 2.1.35: ack.scm

```
(letrec ([ack (lambda (m n)
                 (if (= m 0)
                   (+ n 1)
                   (if (= n 0)
                     (ack (- m 1) 1)
                     (ack (- m 1) (ack m (- n 1))))))])
  (ack 3 4))
```

fib

コード 2.1.36: sum.scm

```
(letrec ([fib-aux (lambda (n1 n2 i)
                      (if (= i 0)
                        n2
                        (fib-aux (+ n1 n2) n1 (- i 1))))]
         [fib (lambda (n)
                (fib-aux 1 0 n))])
  (fib 40))

```

sum

コード 2.1.37: sum.scm

```
(letrec ([loop (lambda (i sum)
                 (if (= i 0)
                   sum
                   (loop (- i 1) (+ i sum))))])
  (loop 1000 0))
```

B.2 call/cc の基本性能を計測するプログラム

継続の作成

コード 2.2.38: capture.scm

```
(letrec ([loop (lambda (i)
                 (begin
                   (if (= 0 i)
                     #t
                     (loop (call/cc (lambda (k)
                                      (- i 1)))))))])
  (loop N))
```

継続の呼び出し

コード 2.2.39: call.scm

```
(let ([loop #f])
  ((lambda (i)
                 (if (= 0 i)
                   #t
                   (loop (- i 1))))
   (call/cc (lambda (k)
              (begin
                (set! loop k)
                N)))))
```

継続の作成と呼び出し

コード 2.2.40: capture_and_call.scm

```
(let ([loop #f])
  ((lambda (i)
     (if (= 0 i)
       #t
       (loop
         (call/cc (lambda (k)
                    (k (- i 1)))))))
   (call/cc (lambda (k)
              (begin
                (set! loop k)
                (- N 1))))))
```

B.2.1 call/cc を利用するプログラム

fibgen

コード 2.2.41: fibgen.scm

```
(let ([fibgen
        (lambda ()
          (let ([return #f])
            (letrec ([fib-iterator
                      (lambda ()
                        (letrec ([loop (lambda (fn next)
                                         (begin
                                           (call/cc
                                             (lambda (resume)
                                               (begin
                                                 (set! fib-iterator
                                                   (lambda ()
                                                     (resume #f)))
                                                 (return fn))))
                                           (loop next (+ fn
                                                         next)))))])
                          (loop 1 1)))])
              (lambda ()
                (call/cc
                  (lambda (k)
                    (begin
                      (set! return k)
                      (fib-iterator)))))))))])
  (let ([fg (fibgen)])
    (letrec ([loop (lambda (n)
                     (if (= n 1)
                       (fg)
                       (begin
                         (fg)
                         (loop (- n 1)))))])
      (loop 40))))
```

amb

コード 2.2.42: amb.scm

```
(letrec ([fail-stack null]
         [fail (lambda ()
```

```
3                   (begin
4                     (let ([save (car fail-stack)])
5                       (begin
6                         (set! fail-stack (cdr fail-stack))
7                         (save save))))])
8           [amb (lambda (choices)
9                  (let ([cc (call/cc (lambda (k) (k k)))])
10                   (if (null? choices)
11                     (fail)
12                     (let ([choice (car choices)])
13                       (begin
14                         (set! choices (cdr choices))
15                         (set! fail-stack (cons cc fail-stack))
16                         choice))))])
17    (let ([x (amb (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 (cons
          8 (cons 9 (cons 10 null))))))))))]
18          [y (amb (cons 1 (cons 2 (cons 3 (cons 12 (cons 5 (cons 6
                (cons 4 (cons 8 (cons 11 null))))))))))]
19          [z (amb (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 13
                (cons 7 (cons 8 (cons 9 null))))))))))])
20      (if (= (+ (* x x) (* y y)) (* z z))
21        (cons x (cons y z))
22        (amb null))))
```

参考文献

[1] 754-2019 - IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, 11 1991.

[3] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 528 LNCS:1–13, 1991.

[4] Henry G. Baker. CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A. *ACM SIGPLAN Notices*, 30(9):17–20, 1995.

[5] Per Bothner. Kawa-Compiling Dynamic Languages to the Java VM, 1998.

[6] Bytecode Alliance. Wasmtime —a small and efficient runtime for WebAssembly & WASI.

[7] ECMA. ECMA-335 - Ecma International.

[8] Cormac Flanagan, Amr Sabry, Duba Matthias Felleisen, and Departmentof Rice Houston. The Essence of Compiling with Continuations. *PLDI '93:Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, 28(April 92):237–247, 1993.

[9] Andreas Haas, Andreas Rossberg, SchuffDerek L., TitzerBen L., Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. *ACM SIGPLAN Notices*, 52(6):185–200, 6 2017.

[10] Daniel Hillerström, Sam Lindley, Andreas Rossberg, K C Sivaramakrishnan, Daan Leijen, and Matija Pretnar. WasmFX: Typed Continuations in Wasm, 2021.

[11] Eric Holk. Schism. Technical report, 2018.

[12] Ju Long, Hung Ying Tai, Shen Ta Hsieh, and Michael Juntao Yuan. A Lightweight Design for Serverless Function as a Service. *IEEE Software*, 38(1):75–80, 1 2021.

[13] Microsoft. IronScheme.

[14] Oracle. The Java® Virtual Machine Specification, 2021.

[15] Donald Pinckney, Arjun Guha, and Yuriy Brun. Wasm/k: Delimited Continuations for WebAssembly. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, page 16–28, Virtual, USA, 2020. Association for Computing Machinery.

[16] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 12 1975.

[17] Andreas Rossberg. WebAssembly Core Specification. Technical report, W3C, 2019.

[18] Alex Shinn, John Cowan, Arthur A Gleckler Steven Ganz Alexey Radul Olin Shivers

Aaron W Hsu Jeffrey T Read Alaric Snell-pym Bradley Lucier David Rush Gerald J Sussman Emmanuel Medernach Benjamin L Russel Richard Kelsey, William Clinger, Jonathan Rees, Michael Sperber, R Kent Dybvig, Matthew Flatt, Anton Van Straaten, Guy Lewis Steele Jr, and Gerald Jay Sussman. Revised 7 Report on the Algorithmic Language Scheme. Technical report, 2013.

[19] Ugawa Tomoharu, Minagawa Nobuhisa, Komiya Tsuneyasu, Yasugi Masahiro, and Yuasa Taiich. 継続の生成におけるスタックコピーの遅延. 情報処理学会論文誌プログラミング（*PRO*）, 44(SIG13(PRO18)):72–83, 10 2003.