平成20年度 修士論文

継続の共有化による継続ベースWebサーバのメモリ使用量削減

電気通信大学 大学院情報システム学研究科

情報システム基盤学専攻

0753014 新宮 澄夫

指導教員 小宮 常康 准教授
多田 好克 教授
本多 弘樹 教授

提出日 平成21年1月29日

目 次

図 目 次

第 1 章

はじめに

1.1 研究背景

従来の古典的な CGI による Web アプリケーションでは，ユーザとの対話のたびにプログラムの実行を終了する．その際中断したプログラムを再開するときのために，アプリケーションがどこまで進んだかを表す情報を保存し，再開するときには情報を取り出す．基本的に情報の管理は開発者が記述しなければならず，大きなアプリケーションにおいては管理が煩雑である．また管理コードが明示的に埋め込まれるため，そういった Web アプリケーションのプログラムは可読性が悪いものとなる．

そこで継続ベース Web サーバというものがある [1, 2, 3, 4]．これは中断したプログラムの再開に必要な情報の保存，復元に第一級継続を利用したもので，CUI によるコンソールアプリケーションを記述するときと同様に，アプリケーションの処理の流れの通りプログラムを記述することができる．継続ベース Web サーバによる Web アプリケーションでは従来のスタイルの Web アプリケーションよりプログラムが可読性が良くなるという利点がある．

1.2 目的

継続ベースWebサーバによるWebアプリケーションの継続オブジェクトのサイズは従来のWebアプリケーションが残す情報のサイズに比べ大きい．さらに対話の際に生成された継続を再開するリクエストが，いつユーザから送られるかは分からない．そのため，生成された継続はタイムスタンプなどで管理し，しばらくの間消さずに保存しなければならない．ユーザとの対話が少ないうちは問題にならないが，対話が多くなってくると継続ベースWebサーバは多くのメモリを消費するようになってしてしまう．サーバの負担を軽減するために，継続オブジェクトのサイズを小さくしたい．

継続ベースWebサーバによるWebアプリケーションでは，ユーザとの対話のたびに継続オブジェクトが生成される．継続を捕捉する式で生成される継続のうち，プログラムの字面上同じ式で生成されるものは内容に共通部分がある可能性が高い．．継続ベースWebサーバによるWebアプリケーションの異なるユーザセッションで生成される継続について，内容の一部を共有して保存することによりメモリ使用量を削減できると思われる．そこで本研究では，継続の共有化による継続ベースWebサーバのメモリ使用量削減を提案する．

しかし全ての継続の捕捉において共有化が行えるわけではない．継続を捕捉する関数が呼び出されたときの制御スタックに積まれたフレームの並びが異なる場合，継続の共有化は困難である．そこで継続を捕捉する関数が呼び出されたときの制御スタックに積まれたフレームの並びが常に1つかどうか調べ，常に1つである場合のみ共有化を行うことにする．フレームの並びは制御フロー解析により調べることにする．また，継続には生成されたとき制御スタックに置かれていた局所変数の内容も含まれているが，字面上同じ場所で生成される継続であっても生成のたびに局所変数の値が異なる可能性がある．そのため異なるユーザセッションの継続間で内容が異なる可能性のある局所変数については共有化することができない，そこで

共有化できない局所変数については，従来通り共有化せずに保存することにする．全ての局所変数のうちどれが共有化可能でどれが共有化不可能かを調べるために，局所変数についての解析により共有化できない局所変数の選び出しをする．

本研究ではScheme言語処理系に継続の共有化機能を追加した．さらに継続ベースWebサーバでの利用をシミュレーションして従来法による継続とメモリ使用量や時間的オーバーヘッドについて比較し，継続の共有化の有効性を確認する．

1.3 論文の構成

本論文は以下の構成になっている．

- 第2章ではまず従来の古典的なCGIによるWebアプリケーションの仕組みと欠点について説明する．次に継続ベースWebサーバによるWebアプリケーションについて利点と，大きなメモリ使用量という欠点について説明する．また継続ベースWebサーバのメモリ使用量を削減する手法として，継続の共有化を提案する．

- 第3章ではコンパイル前に行うプログラム解析について述べる．`call/cc`の置き換えを判断するための制御フロー解析について説明する．また局所変数の共有化を判断するための解析について考察する．

- 第4章では共有化を行う継続の設計と実装を述べる．共有化した継続を生成する関数`call/csc`を設計し，TUTSchemeというScheme処理系に実装した．

- 第5章では提案法の評価について述べる．時間的オーバーヘッドと継続ベースWebサーバにおけるメモリ使用量について，従来法と比較検討した．

第 2 章

提案手法

2.1 CGIによるWebアプリケーション

CGI（Common Gateway Interface）とは Web サーバがクライアントからの要求に応じて外部プログラムを実行するための仕組みである．CGI を用いない Web においては，Web サーバは HTML などの静的なページを送るだけである．しかし CGI を用いることにより，プログラムの実行結果に基づいて動的にページを生成することができる．図 2.1 に CGI による Web ページ生成の動作を示す．ユーザから CGI によるページのリクエストを受け取ると，Web サーバは CGI ファイルを実行する言語処理系を起動する．そして URL により指定された CGI ファイルの内容を言語処理系に入力として与え，出力結果をユーザへ送る Web ページとする．

1 つの CGI プログラムにより複数の対話を行う Web アプリケーション，例えば 2 つの数値をユーザから受け取りその和を計算し表示するという簡単なものを実現する場合を考えてみる．まずユーザが初めてリクエストを送ると，1 つ目の数値の入力を促すページが生成されユーザに送られる．そのページには例えば次のようなフォームがある．

```
<FORM METHOD="GET" ACTION="http://www.mywebserver.com/add.cgi">
input first value <INPUT NAME="value"> <P>
Submit button: <INPUT TYPE="submit" VALUE="Submit"> <P>
</FORM>
```

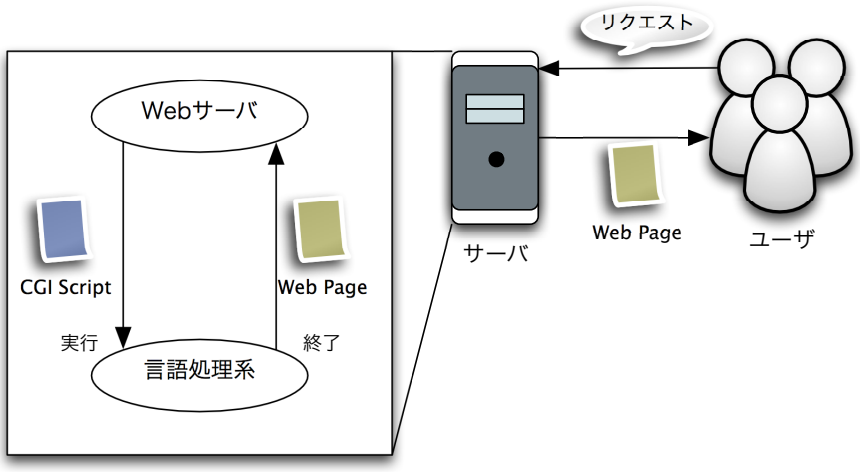

図 2.1: CGI による Web ページ生成の動作

ユーザのブラウザには 1 つの入力フォームと Submit という名前のボタンが表示される．仮にユーザが入力フォームに 10 を入力し Submit ボタンを押すと，Web サーバには次のようなリクエストが送られる．

```
http://www.mywebserver.com/add.cgi?value=10
```

2 つ目のリクエストを受け取ったときは，CGI プログラムは次のような 1 回目とは内容の異なるページを出力する．

```
your first input value = 10
<FORM METHOD="GET" ACTION="http://www.mywebserver.com/add.cgi">
input second value <INPUT NAME="value"> <P>
Submit button: <INPUT TYPE="submit" VALUE="Submit"> <P>
</FORM>
```

ここで問題がある．2 つ目のリクエストを受けたときは，1 つ目の数値の入力を促すページでも加算結果を表示するページでもなく，2 つ目の数値の入力を促すページを作るべきである．CGI プログラムの実行中に，どのようにして出力すべきペー

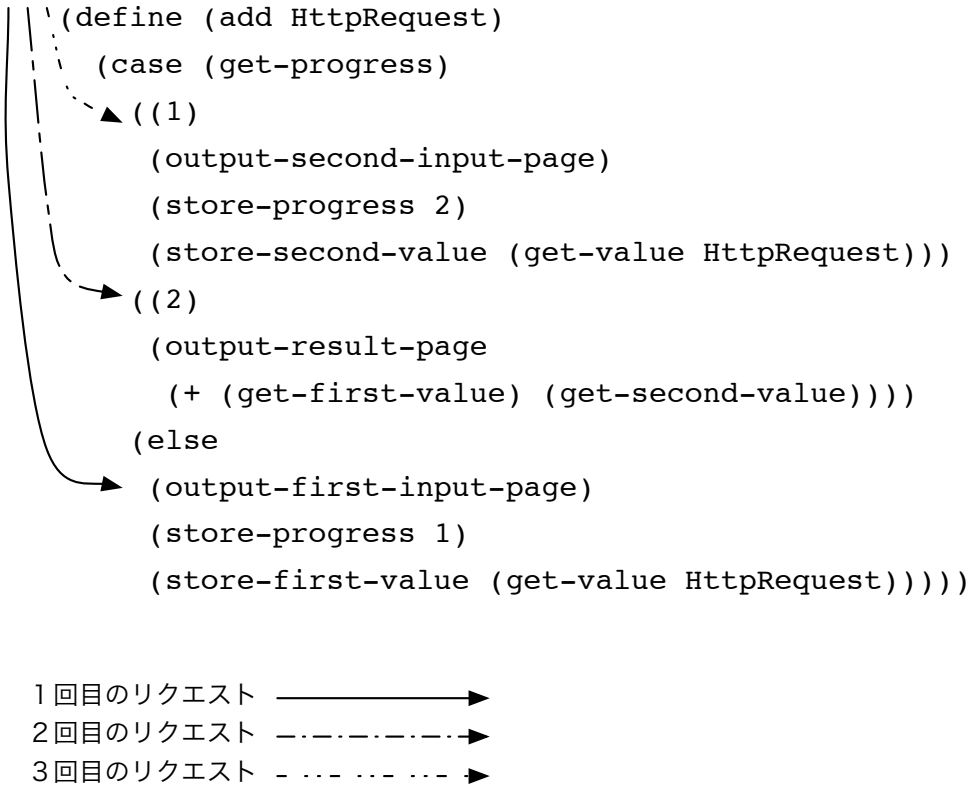

図 2.2: 従来の Web アプリケーションにおける情報の保存

ジを知ればよいのだろうか．方法はいくつかあるが基本的には CGI プログラムの終了前に，いったいアプリケーションの実行がどこまで進んだかを知らせる情報を作り，CGI の実行が終了しても情報が失われない場所に保存している．加算の例で言うと，1 回目の対話の際に次の処理は 2 つ目の数字の入力を促すことであるという情報を保存しておく．例えば加算アプリケーションを実現する CGI プログラムは図 2.2 のように書ける．関数 add にユーザからのリクエストが与えられると，まずアプリケーションの実行がどこまで進んでいるかを調べる．進捗を表す情報に応じて異なる処理を行い，ふさわしい内容のページを出力する．またページ内容を出力した後に，図 2.3 のように入力された値や新しい進捗情報をセッションオ

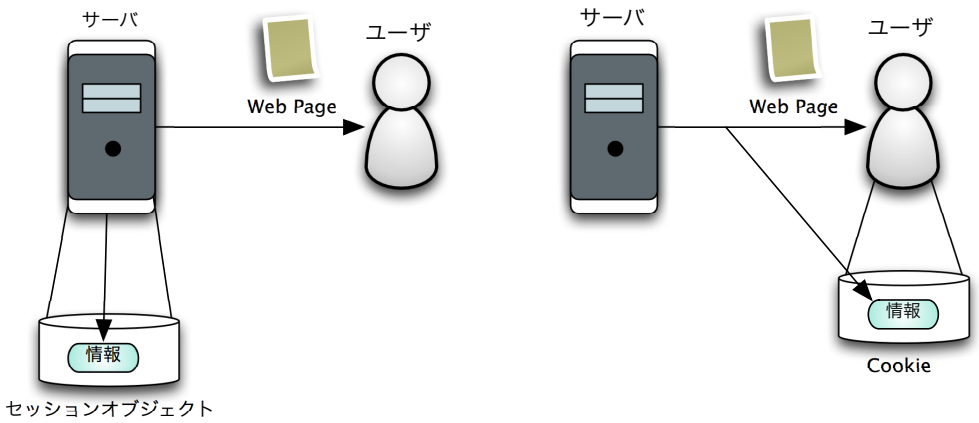

図 2.3: 従来の Web アプリケーションにおける情報の保存

```
(define (add)
   (let* ((val1 (read))  ;; 入力 1
          (val2 (read)))  ;; 入力 2
      (display (+ val1 val2))))  ;; 出力
```

図 2.4: コンソールプログラムによる加算アプリケーション

ブジェクトなどに保存する．情報の保存場所としてはセッションオブジェクト以外に Cookie やフォームの hidden フィールドなどがある．

図 2.4 にコンソールプログラムにより加算アプリケーションを実現するときのコードの例を示す．CGI による Web アプリケーションでは，基本的に進捗情報や入力された値などの管理は開発者がプログラムに明示的に記述しなければならない．大規模なアプリケーションにおいてはその管理が煩雑である．また本来行いたい処理の間にそのような管理コードが埋め込まれているため，そういった Web アプリケーションのプログラムは可読性が悪いものとなる．

2.2 継続ベース Web サーバ

2.2.1 継続

継続とは，計算過程のある時点における「残りの計算全て」を表す概念である [6]. Scheme[7]， Ruby[8] などいくつかのプログラミング言語では継続を第一級のデータとして取り入れており，明示的に使用することができる．Scheme では継続を捕捉する関数 call-with-current-continuation（以後，省略形の call/cc を用いる）という組み込み関数を用いることで，第一級継続を扱うことができる．call/cc は次のような形で使われる．

```
(call/cc <function>)
```

call/cc が呼び出されると，システムは call/cc 式実行後の継続を生成し，その継続を引数として<function>を呼び出す．もし<function> の実行中に継続が呼び出されると call/cc 式の実行を直ちに終了し，call/cc 式終了後の残りの計算にとりかかる．このとき，呼び出した継続への引数が call/cc 式の値として使われる．<function>実行中に継続が呼び出されたなかった場合は，<function>の返す値が call/cc 式の値となる．また，継続はグローバル変数などに束縛しておいて call/cc 式終了後に呼び出すこともできる．継続の主な利用方法には

- 大域脱出
- コルーチン
- マルチタスク

などがある．

図 2.5 に call/cc を使ったプログラム例を示す．read-value はユーザからの入力を受け取る関数である．ユーザからの入力が関数 f へ引数として渡され，局所変数 n に束縛される．関数 f の中では継続を生成し変数 esc に束縛している．もし n

```
(define (f n)
  (call/cc
    (lambda (esc)
      (+ 10
         (* 5
            (/ 1 (if (zero? n)
                     (esc 1)
                     n)))))))
(define (g)
  (f (read-value)))
(g)
```

図 2.5: call/cc を用いた大域脱出

の値が 0 であれば，ゼロ除算を回避するために esc を呼び出し大域脱出を行う．また図 2.5 の call/cc 呼び出し時の制御スタックの内容を図 2.6 に示す．まず関数 h を呼び出すと，制御スタックに h のフレームが積まれる．同様に関数 g と関数 f のフレームも制御スタックに積まれる．call/cc を呼び出したときの制御スタックの内容は図 2.6 の通りである．call/cc で生成される継続すなわち call/cc 式実行後の仕事というのは，

- f の残りの処理
- g の残りの処理

である．この処理を行うのに必要な情報というのは，図 2.6 の太枠で囲んだ部分に全て含まれている．つまり継続オブジェクトの内容は call/cc を呼び出したとき制御スタックに積まれている call/cc 以外の関数のフレームである．

継続の実装は処理系依存であり GC 戦略，スパゲッティ戦略，heap 戦略，stack 戦略，chunked-stack 戦略，stack/heap 戦略，incremtal stack/heap 戦略などさま

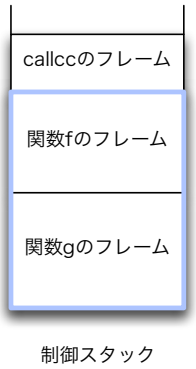

図 2.6: 図 2.5 の call/cc 呼び出し時の制御スタック

ざまな実装戦略が存在する [9]．ここでは制御スタックの内容をヒープにコピーする方法である stack/heap 戦略を説明する．図 2.7 に stack/heap 戦略による継続生成時の動作を示す．call/cc が呼び出された時点で，制御スタックのトップには call/cc の関数フレームが積まれている．そこには引数として受け取った関数 (仮に fun とする) も含まれている．その時点の制御スタック（call/cc の関数フレームを除く）の内容をスタックからヒープへ退避する．この時点で制御スタックは空になる．call/cc の引数として与えられた関数 fun のフレームを空の制御スタックに追加する．「制御スタックの内容をクリアし，退避していた内容を制御スタックに戻す」という動作をする手続きを作る．fun の引数に継続が与えられることになっているので，その手続きを制御スタックに push する．仮に次のように継続が呼び出されたときの動作を述べる．

```
(k val)
```

1. その時点での制御スタックの内容をクリアする．
2. 制御スタックからヒープへ退避していた内容を，制御スタックに戻す．

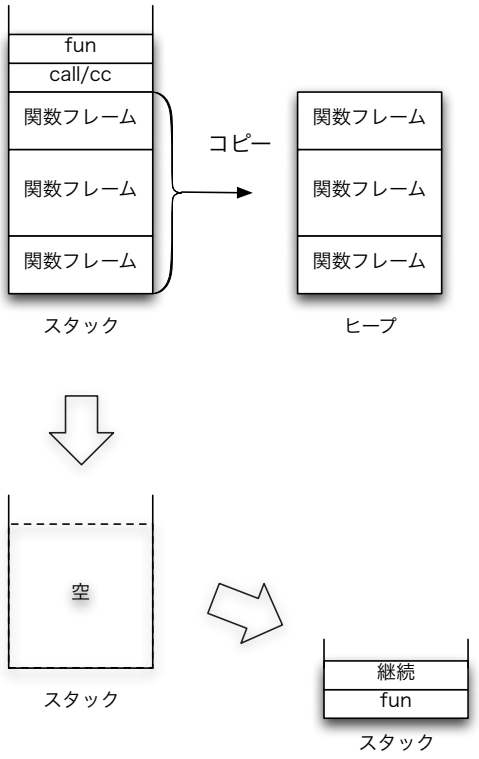
fun
call/cc
関数フレーム
関数フレーム
関数フレーム
コピー
関数フレーム
関数フレーム
関数フレーム
スタック
ヒープ
空
スタック
継続
fun
スタック

図 2.7: 継続の実装例

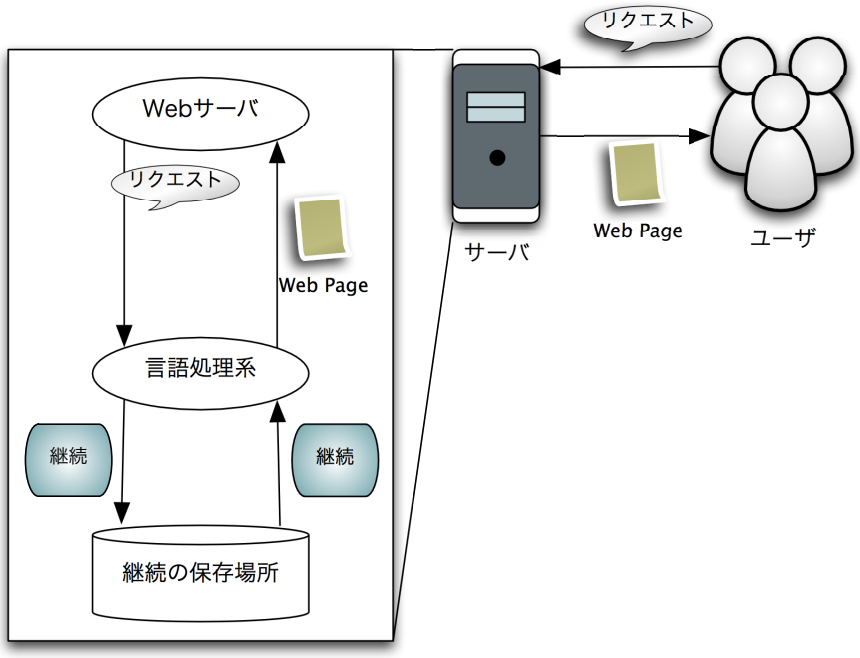

図 2.8: 継続ベース Web サーバの概観

3. 関数 `call/cc` の結果が val であるものとして計算を再開することになっているので，`val` を制御スタックに push する．

4. プログラムの実行を再開する．

2.2.2 継続による Web プログラミング

従来の古典的な CGI による Web アプリケーションにおいて開発者が明示的に書いていた管理コードの働きは，言語処理系の継続機能によって実現することができる．図 2.8 に継続ベース Web サーバの概観を示す．古典的な CGI による Web アプリケーションとの違いは，中断した実行の再開に必要な情報の保存に第一級継続を利用していることである．図 2.9 に第一級継続を利用した実行再開のための情報の保存の例 [5] を示す．

関数 `show` が行う処理は以下の通りである．

```
(define (show html-generator)
  (call/cc
   (lambda (k)
     (display (html-generator (k->url k))
                                ;;create URL from Continuation
              (current-connection) )
     (close-output-port (current-connection))
     (suicide) ) ) )

(define (server httpRequest)
  (let ((url (get-request-url httpRequest)))
    (let ((k (url->k url)))      ;;get Continuation from URL
      (if k
          (k httpRequest)
          <application> ) ) ) )
```

図 2.9: 第一級継続を用いた実行再開のための情報の保存

1. 関数 show の内部で call/cc が呼び出され，関数 show からリターンした後の「残りの計算全て」を含んだ継続が捕捉される．
2. 捕捉された継続が変数 k に束縛される．
3. 関数 k->url によりユニークな URL を新しく生成する．
4. さらに生成された URL から k に束縛された継続へのマッピングを記憶する．
5. k に束縛されている継続をデータベースなどへ保存する．
6. 関数 html-generator により生成された URL を含んだ html ページが作られる．
7. ユーザへ html ページを送信する．
8. 関数 sucide により実行を終了する．

このように関数 show の内部では，第一級継続を利用して実行再開のための情報の保存を行っている．ユーザからリクエストを受け取ったときサーバが行う処理は以下の通りである．

1. 関数 get-request-url により受け取ったリクエストから URL を取得し，url に束縛する．
2. 関数 url->k により url に対応する継続が存在するか調べる．
3. url に対応する継続が存在するときはデータベースなどから継続を取り出し，リクエストを関数 show の結果として継続を再開する．
4. url に対応する継続が存在しないとき通常のリクエスト処理として<application>を実行する．

```
(define (input-val-from-client)
  (get-request-val (show input-page)))
(define (output-val-to-client val)
  (show (output-page val)))

(define (add)
  (let ((val1 (input-val-from-client)) ;; 入力 1
        (val2 (input-val-from-client))) ;; 入力 2
    (output-val-to-client (+ val1 val2)))) ;; 出力
```

図 2.10: 第一級継続を利用した加算アプリケーション

2.2.3 利点と欠点

CUI によるコンソールプログラムで 2 つの数値をユーザから受け取り加算結果を出力するというプログラムを書くと図 2.4 のようになる．図 2.10 に同じアプリケーションを第一級継続を利用して実現したプログラムを示す．第一級継続を利用することで，コンソールアプリケーションを記述する場合と同様に，アプリケーションの処理の流れの通りプログラムを記述することができる．

このように継続ベース Web サーバによる Web アプリケーションのプログラムは従来の Web アプリケーションのプログラムに比べ，以下のような利点がある．

- 管理コードで行っていたことを言語処理系の継続機能にまかせることができる．
- プログラムの可読性が良くなる．

しかし継続ベース Web サーバによる Web アプリケーションで保存する継続のサイズは，従来の Web アプリケーションにおいてセッションオブジェクトや Cookie に保存していた情報のサイズよりも大きく数 Kbyte にもなる [5]．また対話の際に

保存した継続を呼び出すリクエストが，いつユーザから送られるかは不明である．生成された継続はタイムスタンプなどで管理し，寿命が過ぎるまでは消さずに保存しておかなければならない．そのため継続ベース Web サーバによる Web アプリケーションには従来のものよりも多くのメモリを消費してしまうという欠点がある．

2.3 継続の共有化

継続ベース Web サーバの欠点である大きなメモリ使用量を削減したい．継続ベース Web サーバが従来の Web アプリケーションに比べメモリ使用量が大きいのは，継続オブジェクトのサイズに原因がある．この継続オブジェクトのサイズを従来の Web アプリケーションにおける実行再開のための情報のサイズに近づけることができれば，継続ベース Web サーバ全体のメモリ使用量も従来の Web アプリケーションのメモリ使用量に近づけることができる．この節ではいかにして継続オブジェクト 1 つあたりのサイズを小さくするか，その基本的アイデアを明らかにする．

2.3.1 共有化によるメモリ使用量削減

次のプログラムで生成される継続について考える．

```
(define (f)
 ... (call/cc ...) ...)
(define (g)
 ... (f) ...)
(g)
```

まず関数 g が呼び出され，制御スタックに g のフレームが積まれる．同様に f のフレームも積まれる．次に call/cc が呼び出され，制御スタックにある g と f の

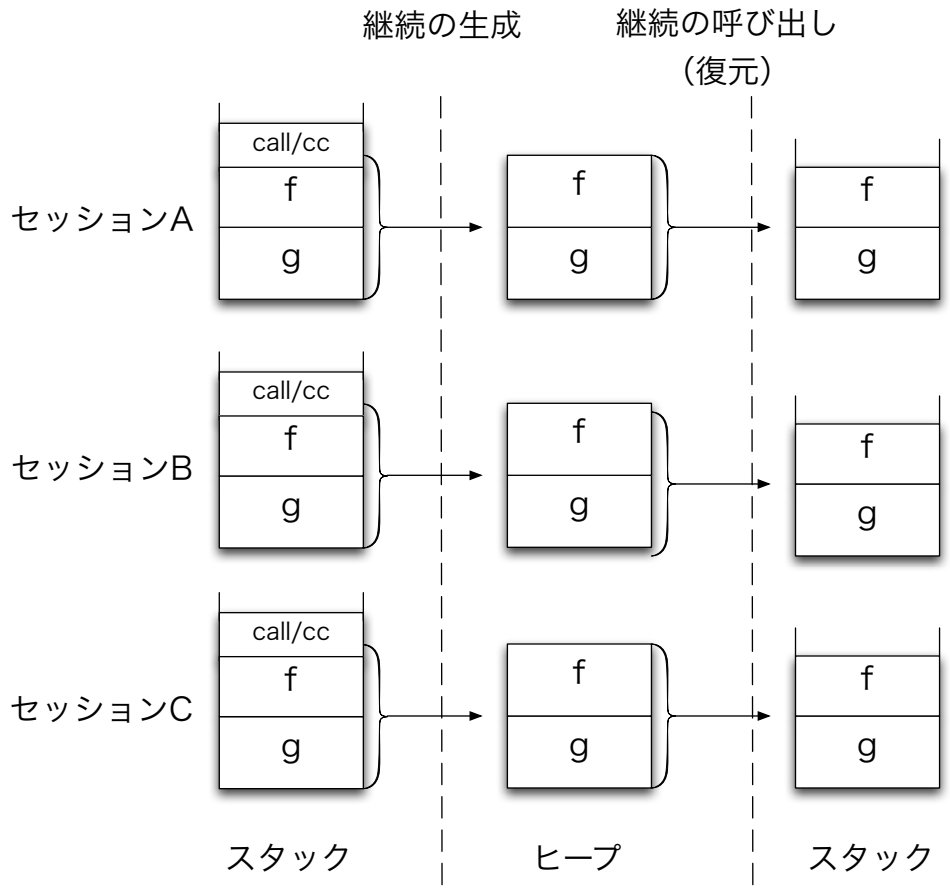

図 2.11: 従来法による継続の生成と復元

フレームが継続としてヒープにコピーされる．図 2.11 に従来法における複数セッションでの継続の生成と復元の様子を示す．call/cc を何度も呼び出すと，g と f のフレームからなる継続が何個も生成される．このとき図 2.12 のように継続を 1 つにまとめて保存することでメモリを節約することができる．

具体的には，1 回目の call/cc の呼び出しではヒープへの退避を行うが，2 回目以降の call/cc の呼び出しではすでに生成済みの継続を共有して使用する．また継続呼び出しの際のヒープからスタックへの復元は通常通りに行う．

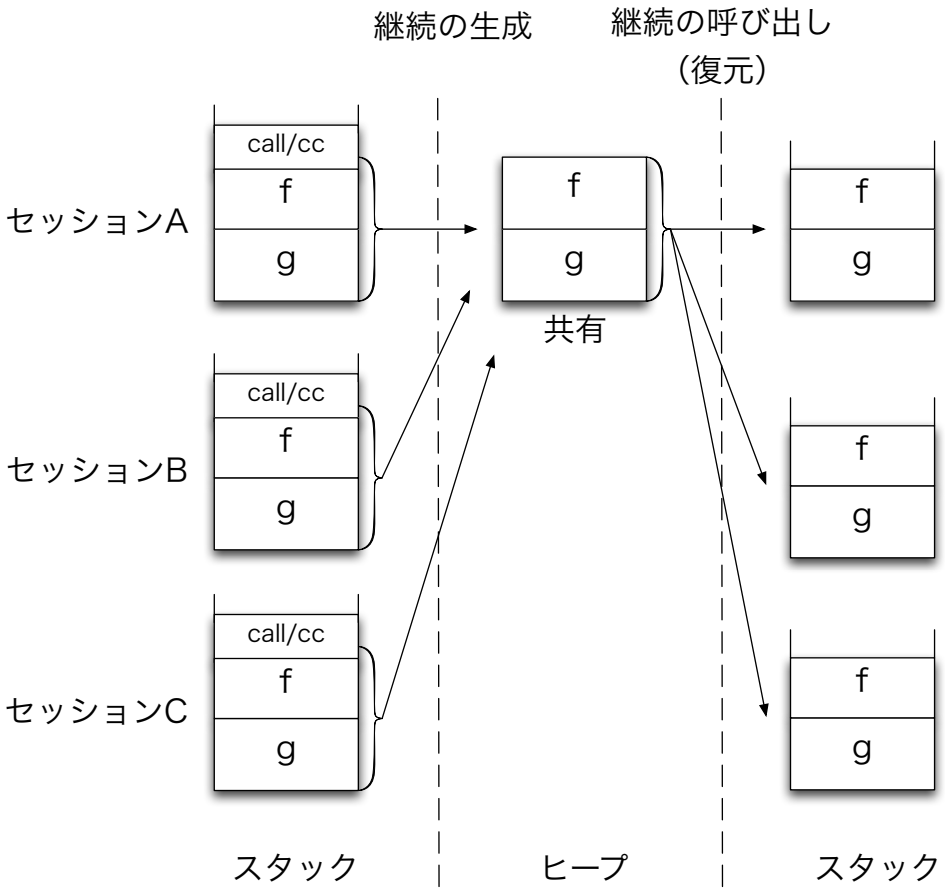

図 2.12: 共有の基本アイデア

```
(define (add)
    (let ((val1 #f)
          (val2 #f))
        (set! val1 (get-request-val (show input-page))) ;;input1
        (set! val2 (get-request-val (show input-page))) ;;input2
        (show (output-page (+ val1 val2)))))          ;;output
```

図 2.13: 単純な加算アプリケーション

2.3.2 継続ベース Web サーバへの適用

継続ベース Web サーバによる簡単な Web アプリケーション，例えばプログラム 2.13 のような加算アプリケーションを多数のユーザが並行に実行しているケースについて考える．

それぞれのユーザがサーバにリクエストを送り，ユーザ毎に異なるセッションが開始される．さらに各セッションで input1 を行うための関数 show が呼び出される．関数 show の内部で call/cc を呼び出し継続を生成し，保存し，URL と関連づけ，その URL を送信するページに埋め込む．各ユーザとの対話のたびに継続が生成されるため，異なるセッションで生成された多くの継続オブジェクトが存在することになる．さらにプログラムの字面上で同じ call/cc 式で生成される継続オブジェクトはフレームの並びが同じで，内容に共通部分が多い可能性がある．この例のように call/cc で生成される継続オブジェクトの内容について，関数フレームの並びが常に同じであることが明らかならば，生成される継続を異なるセッション間で共有化する余地がある．

2.3.3 問題点と解決法

異なるセッションの call/cc で生成される継続オブジェクトについて，常に共有化して保存することができるわけではない．同じ call/cc で生成される継続でも，call/cc を呼び出すときの制御スタックの内容が異なることがある．

例えばプログラム中のある call/cc を異なる制御フローを通じて呼び出すことが可能な場合，制御スタックに積まれるフレームの並びが異なる可能性がある．call/cc を呼び出すときの制御スタックに積まれたフレームの並びが異なる場合，共有化するのは困難である．そこで共有化を行うのは call/cc を呼び出すときのフレームの並びが常に一定であることが保証できる call/cc 式についてのみということにした．

制御スタックの内容は図 2.14 のようにいくつかの関数のフレームに，さらにフレームは制御情報と局所変数に分けることができる．制御情報はリンクやリターンアドレス等である．トップレベルから call/cc 式へ至る制御フローが 1 つである call/cc で生成される継続は，制御情報については異なるユーザセッションの継続同士で全く同一である．そこで共有化すると判断した call/cc で継続オブジェクトを生成する場合，制御情報についてはいつでもその全体を異なるセッション間で共有することができる．しかし制御スタックの中で局所変数が占める割合は少なくない．そこで局所変数についても共有化できる部分については共有化したい．しかしフレームの並びが同じでも，局所変数の内容が異なる可能性がある．何故ならば異なるセッションの同じ関数の引数に，異なる値が与えられる可能性があるためである．ある局所変数の内容が異なる可能性があるならば，その局所変数についてはセッション毎に個別に保存する必要がある．またある局所変数について共有できるか否かを，継続を生成するたびに共有している継続オブジェクトのものと比較して決めるのはコストを考えると望ましくない．そこで代わりに共有化を行う call/cc へ引数などで，どの局所変数が共有できないかを知らせる情報を与えることで，継続オブジェクトを生成する際に発生する時間的オーバーヘッドを小

図 2.14: 制御フレームの内容

さくすることができる．そのためあらかじめ各局所変数について共有できるか否かを調べ，共有しない局所変数を選び出しておく必要がある．

継続の共有化を自動的に行う場合，以下のような処理をコンパイル前に行う必要がある．

- プログラム中の各 call/cc について呼び出される際のフレームの並びを調べ，常に同じ並びであると保証できるものがあれば共有化を行う call/cc へ置き換える．共有化を行う call/cc は初回の呼び出しでは制御スタックをヒープにコピーし，２回目以降はすでに退避したものを共有して使用する．また共有化する以外の機能は変わらない．

- 局所変数について異なるセッション間で内容が異なる可能性があるかどうか調べ，内容が異なる可能性がある局所変数を選び出しておく．共有化を行う call/cc へ引数などで制御スタック上における共有化しない局所変数の位置情報を与え，その部分のみセッション毎に個別に保存する．

第 3 章

プログラム解析

3.1 制御フロー解析

3.1.1 call/cc 置き換えの判断

共有化する call/cc 式に置き換える call/cc 式と，置き換えない call/cc 式を判別する必要がある．どのように行えば良いだろうか．どのようなケースでは共有化が困難かについて考える．まずプログラム中の異なる call/cc で生成される継続は明らかに共有が困難である．何故ならばプログラムの字面上で異なる call/cc が呼び出されるときの制御スタックのフレームの並びが同じであるというのはかなり特殊なケースである．一方，プログラムの字面上で同じ call/cc が呼び出されるときの制御スタックのフレームの並びが同じであるケースは一般的にあり得る．

プログラムの字面上で同じ call/cc で生成される継続でも，そこへ至るまでの制御フローが複数存在することがある．その場合，制御スタックのフレームの並びが異なるため共有化は困難である．例えば次のプログラムのように関数 f から call/cc が，関数 g と関数 h から f が呼ばれるプログラムについて考える．

```
(define (f)
 ... (call/cc ...) ...)
(define (g)
 ... (f) ...)
```

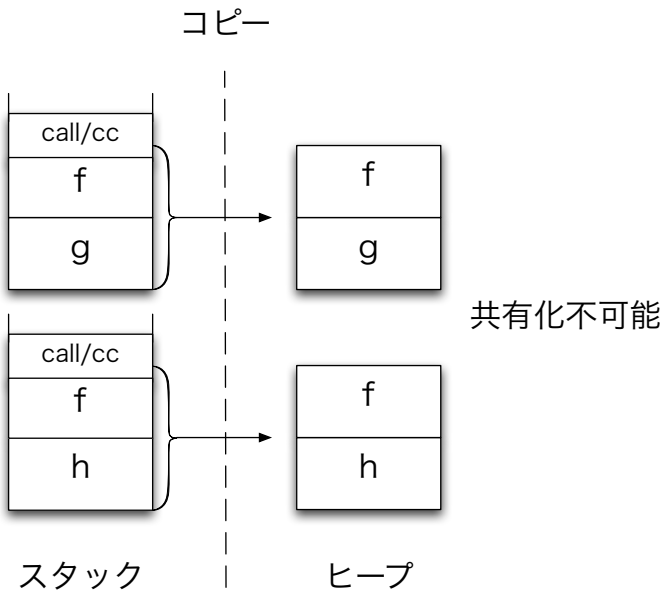

図 3.1: 共有化できないケース 1

```
(define (h)
 ... (f) ...)
(g)
(h)
```

call/cc で生成される継続には図 3.1 のように，call/cc で生成される継続オブジェクトには g の関数フレームが含まれるものと h の関数フレームが含まれるものの 2 種類が考えられるため共有化は困難である．

また次のプログラムのように f が g のみから，g が関数 g1 と関数 g2 から呼ばれる場合も同様である．

```
(define (f)
 ... (call/cc ...) ...)
(define (g)
```

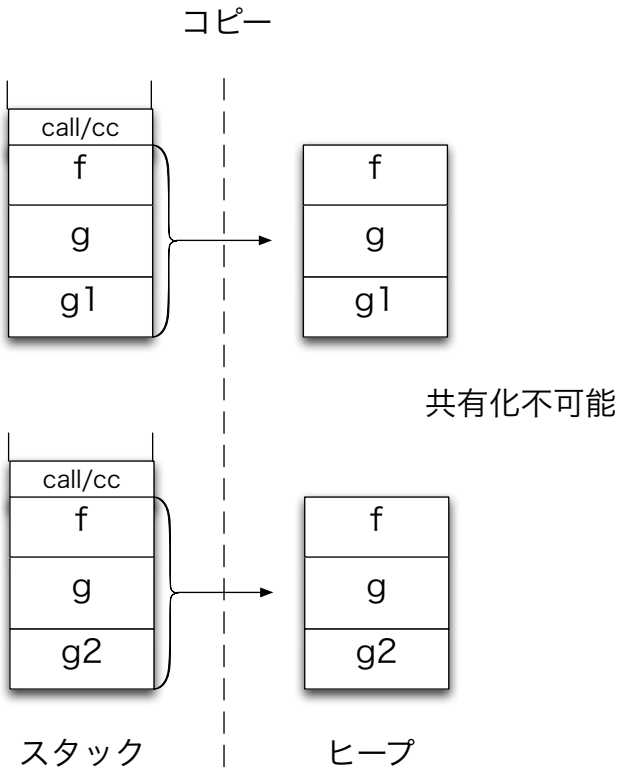

図 3.2: 共有化できないケース 2

```
 ... (f) ...)
(define (g1)
 ... (g) ...)
(define (g2)
 ... (g) ...)
(g1)
(g2)
```

図 3.2 のように call/cc で生成される継続オブジェクトには g1 の関数フレームが含まれるものと g2 の関数フレームが含まれるものの 2 種類が考えられるため共有化が困難である．ただし，次のプログラムのように g が g1 から呼び出されたときのみ f を呼び出すような場合，call/cc が呼び出されるときの制御スタックのフレームの並びは一種類なので共有可能である．

```
(define (g)
 (if (caller-is-g1?)
     (f)
     ...)
 ...)
```

また再帰呼び出しが存在し，再帰呼び出し関数のフレームが制御スタックに含まれる場合，共有化は困難である．何故ならば，例えば自己再帰呼び出しされる関数 r のフレームが制御スタックに含まれているとする．r を呼び出すたびに制御スタックにはそのフレームが追加されるが，再帰呼び出しの回数によりそのフレームの数が変わるため，call/cc を呼び出すたびに制御スタックに含まれる r のフレームの数により制御スタックのサイズやフレームの並びが異なることが考えられる．

ただし呼び出し回数が必ず同じ回数で止まる場合はその限りではない．再帰呼び出し関数の存在は継続の共有化を行える場面を増やすことへの障害になってしまう．

ある call/cc 式を共有化するかどうかは，call/cc を呼び出すときの制御スタックのフレームの並びが常に同一であるかどうかで決める．そのフレームの並びはトップレベルから call/cc へ至る制御フローの数だけ存在する．すなわち継続の共有化を自動的に行うためには，トップレベルから各 call/cc への制御フロー解析を行えばよい．プログラム中の全ての call/cc についてトップレベルからその call/cc 式へ至るまでの制御フローを調べ，トップレベルからの制御フローが 1 つしかない call/cc 式についてのみ共有化を行うようにすればよい．

上記の関数 caller-is-g1?が使われているプログラムでも共有化可能であると判別することで，共有化する場面を多くできる．そのためにはフロー解析に g1 から呼ばれた g と，g2 から呼ばれた g を区別する精度が求められる．また継続ベース Web サーバによる Web アプリケーションのプログラミングについて，再帰呼び出し関数や相互再帰呼び出しを禁止するようなポリシーを導入することでも，共有化する場面を多くできると思われる．

3.1.2 解析方法

プログラム中の各 call/cc 式について共有化が可能かどうか判断するために制御フロー解析を行う．本研究では Shivers の抽象解釈 [11] によるフロー解析を利用した．また解析プログラムは Shivers のプログラムを使用した．その理由は以下の通りである．

- 本研究と同じく，Scheme 言語を対象としていること．
- 本研究が要求する解析精度を満たすこと．
- プログラムの構造に従って構築された解析器であり，本手法で必要となる局所変数の解析を行うよう拡張することが比較的容易なこと．

共有化する場面を多くするためには，同じ関数でも異なる呼び出し元から呼び出されたときは違いを区別して解析する精度が必要になる．そのため 1CFA というフロー解析を使用する．これは同じ関数でもそこまでの呼び出し順序が異なる場合は区別して扱うため，上記の要求を満たす．

図 3.5 に簡単なプログラムと，そのプログラムの 1CFA によるフロー解析結果を示す．このプログラムは x，y の 2 つの引数を受け取り，x に 4 を足し，それに y をかけるというものである．この解析は CPS （Continuation Passing Style）という中間的な表現を使う [11]．そのためフロー解析をする前にまず CPS 変換を行い，プログラムを CPS 形式にする．C1 や L1 といった記号は call site と `lambda` 式につけられたラベルである．本解析器では変数とその値への束縛を，変数名と contour のセットから値へのマッピングにより管理している．局所変数は `lambda` 式と `letrec` 式で新たに作られるが，contour は全ての `lambda` 式と `letrec` 式にユニークに与えられる整数である．contour の違いにより，異なる `lambda` 式または `letrec` 式で作られる同名の局所変数を区別して扱うことができる．BENV は contour 環境と呼ばれるもので，構文上のバインディング（`lambda` 式や `letrec` 式）から contour へのマッピングをしている．1CFA では字面上で同じ `lambda` 式が何度も呼ばれるとき，各呼び出し毎に異なる contour を与えている．最後の CCache（CallCache）はある関数が，どういった contour 環境で，どこから呼び出されたを記録したキャッシュである．このキャッシュを調べることで，プログラム全体の詳細な制御フローを知ることができる．なお，図 3.5 の中の `XLAM`，`XCALL`，`XCONT`，`XRET` の意味はそれぞれ外部の `lambda` 式，外部の呼び出し元，外部の継続，外部のリターンである．

- 関数*は (<b2> <b3> <b4>) という contour 環境で，call site C1 から呼び出されている．つまり図 3.3 のようにトップレベルから `L3` がついた `lambda` 式，`L2` がついた `lambda` 式，`L1` がついた `lambda` 式の順で制御フローが進んだ状態で `C1` から呼び出されたということが分かる..

- 関数+は (<b3> <b4>) という contour 環境で，call site C2 から呼び出され

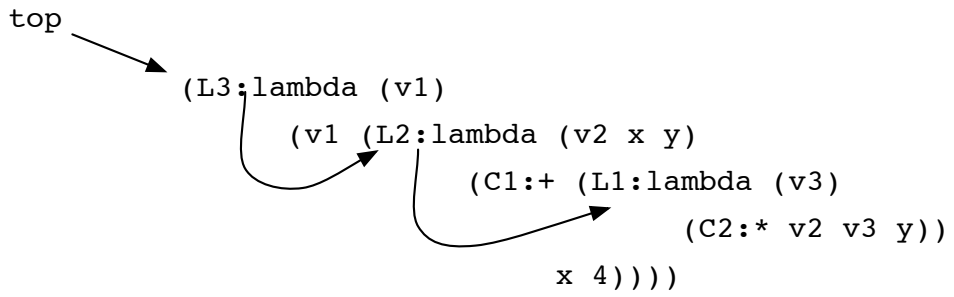

図 3.3: トップレベルから関数*へフロー

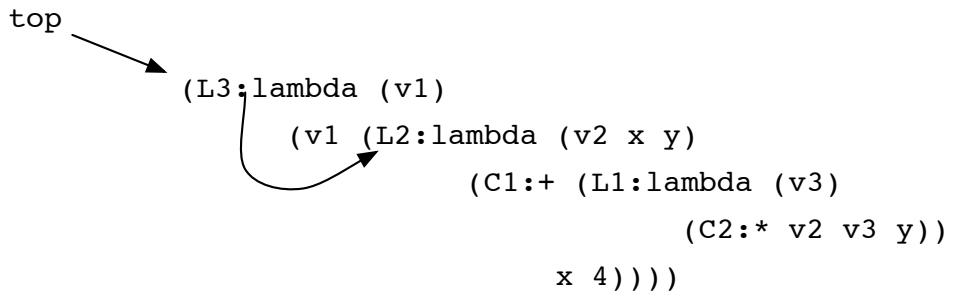

図 3.4: トップレベルから関数+へフロー

ている．つまり図 3.4 のようにトップレベルから L3 がついた lambda 式，L2 がついた lambda 式，の順で制御フローが進んだ状態で C2 から呼び出されたということが分かる..

図 3.8 に単純な加算を行う Web アプリケーションのプログラムとその 1CFA による解析結果の一部を示す．web-read は継続を利用してユーザから入力を受け取る関数，display は継続を利用してユーザへデータの表示を行う関数である．トップレベルから関数 display 内で呼び出される call/cc へ至る制御フローは 1 つなのでこの call/cc 式は置き換え可能であることが分かる．しかし一方トップレベルから関数 web-read 内で呼び出される call/cc への制御フローは 2 通りあるため，この call/cc は共有化が困難であることが分かる．関数 web-read の呼び出しには関数+の第 1 引数の位置で呼び出されるときと，関数+の第 2 引数の位置で呼び出されるときの 2 つの場合がある．実際にそれぞれの場合では，call/cc を呼び出すときの制御スタックの内容が少し異なる．

- 関数 display 内の call/cc の呼び出しは (<b13> <b14> <b15>) という contour 環境で呼び出されるケースの 1 通りである．図 3.6 にトップレベルから関数 display への制御フローを示す．

- 関数 web-read 内の call/cc の呼び出しには (<b20> <b14> <b15>) という contour 環境で呼び出されるケースと，(<b22> <b14> <b15>) という contour 環境で呼び出されるケースの 2 通りがある．図 3.7 にトップレベルから関数 web-read への制御フローを示す．

web-read 内で呼び出される call/cc が共有化できなかったのは web-read が関数であり，その関数が複数の場所で呼び出されたためである．display は同じく関数だが 1 箇所でしか呼び出されなかったために，トップレベルから display 内で呼び出している call/cc への制御フローの数は 1 つだった．共有化できる場面を増やす方法として，例えば web-read や display などの関数をマクロで定義することが考えられる．web-read や display を関数からマクロに変更しても同様の機能が実現できる．マクロ展開することでプログラム 3.8 は図 3.9 のように展開される．関数+の引数部分に注目する．図 3.8 では関数 web-read の呼び出し式が 2 つ

```
(lambda ()
   (lambda (x y)
      (* (+ x 4) y)))
;; CPS form
(L3:lambda (v1)
      (v1 (L2:lambda (v2 x y)
                (C1:+ (L1:lambda (v3)
                            (C2:* v2 v3 y))
                      x 4))))

L3 =    (lambda (v1) C3)
L2 =    (lambda (v2 x y) C2)
L1 =    (lambda (v3) C1)
C3 =    (v1 L2)
C2 =    (+ L1 x 4)
C1 =    (* v2 v3 y)
;;; BENV = <b?> (BINDER Cfrom)
<b6>    (XLAM #f)
<b5>    (+ C2)
<b4>    (L3 XCALL)
<b3>    (L2 XCALL)
<b2>    (L1 ic+)
<b1>    (* C1)
;;; ^CCache = (CALL . ^BENV) → ^D
(ic* <b1>) → (XCONT)
(C1 <b2> <b3> <b4>) → (*)
(ic+ <b5>) → (#(L1 (<b3> <b4>)))
(C2 <b3> <b4>) → (+)
(C3 <b4>) → (XCONT)
(XCALL <b6>) → (#(L2 (<b4>)) #(L3 empty-benv) XFUN)
(XRET <b6>) → (XCONT)
```

図 3.5: フロー解析の例 1

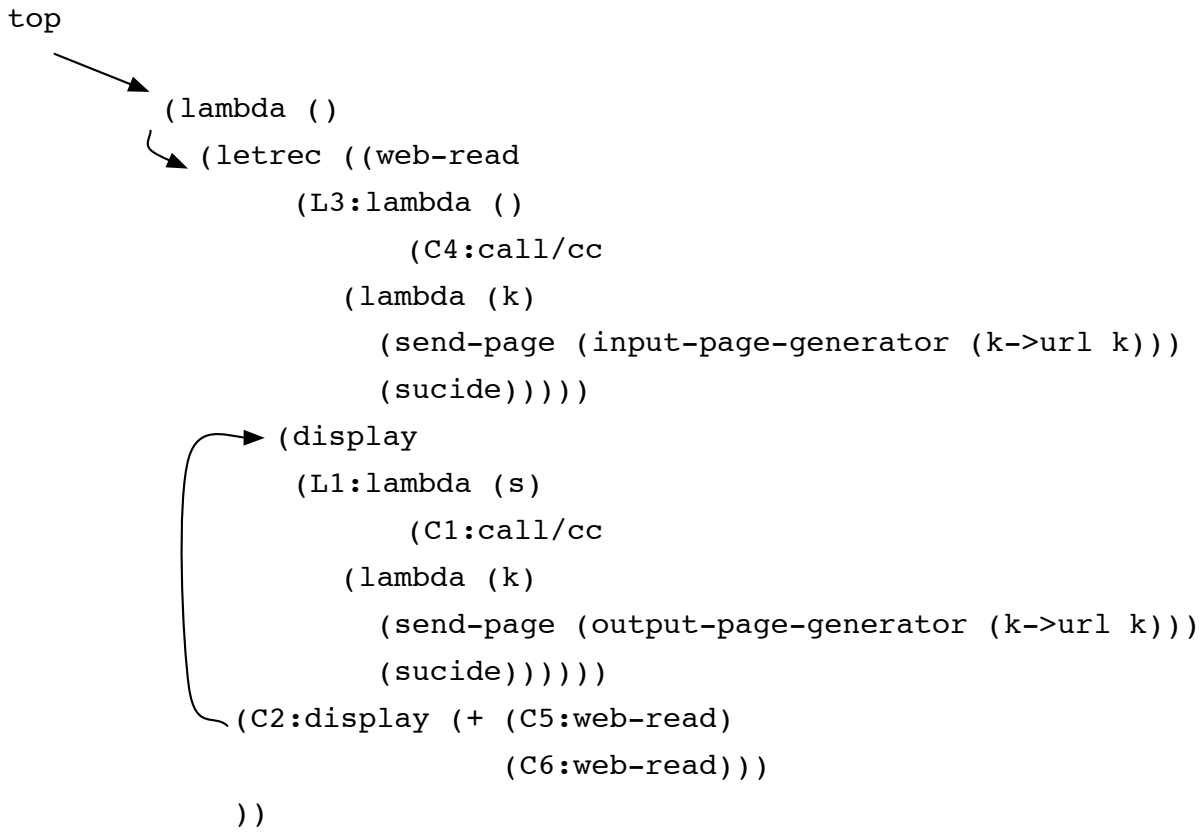

図 3.6: トップレベルから関数 display へフロー

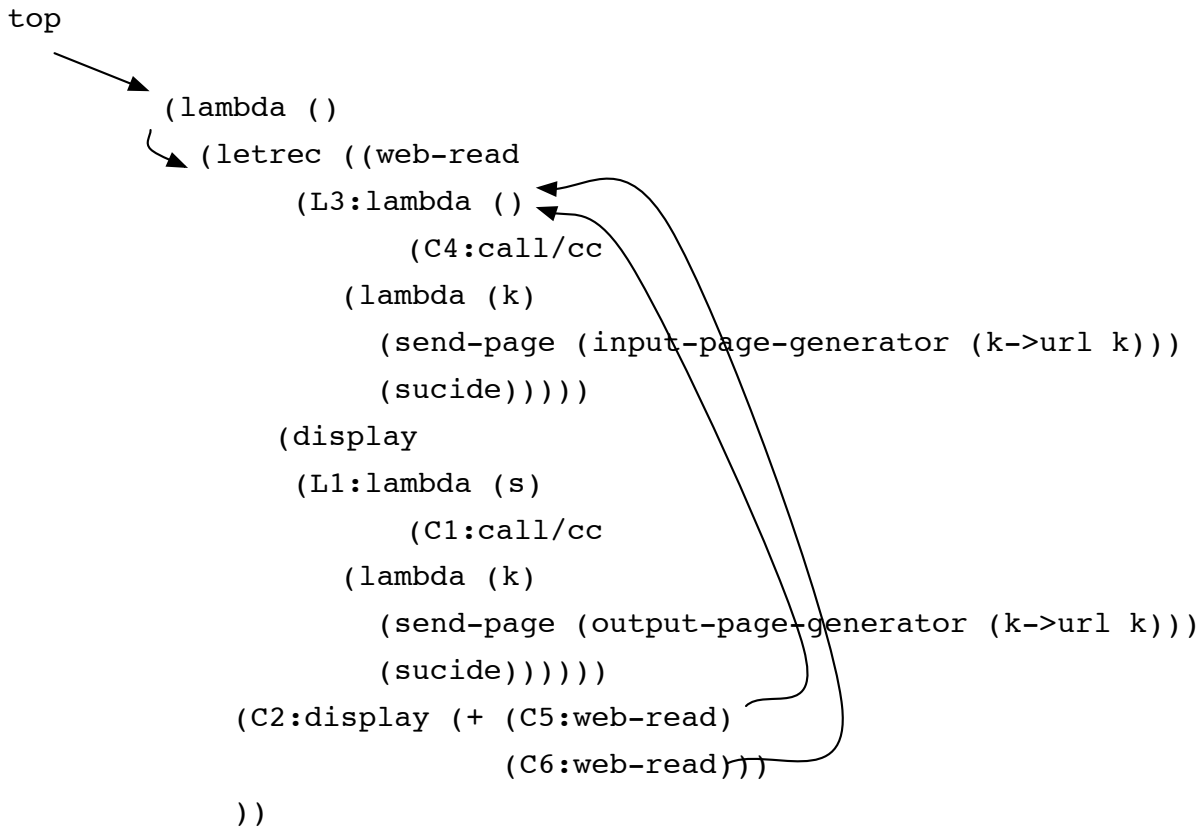

図 3.7: トップレベルから関数 web-read へフロー

```
(lambda ()
  (letrec ((web-read
            (L3:lambda ()
               (C4:call/cc
                   (lambda (k)
                      (send-page (input-page-generator (k->url k)))
                      (sucide)))))
           (display
            (L1:lambda (s)
               (C1:call/cc
                   (lambda (k)
                     (send-page (output-page-generator (k->url k)))
                     (sucide))))))
    (C2:display (+ (C5:web-read)
                   (C6:web-read)))
    ))

;; 解析結果の一部
;;; BENV = <b?> (BINDER Cfrom)
 <b22>  (L3 C6)
 <b20>  (L3 C5)
 <b15>  (L6 XCALL)
 <b14>  (LAB1 LAB1)
 <b13>  (L1 C2)

 ;;; ^CCache = (CALL . ^BENV) → ^D
 (C1 <b13> <b14> <b15>) → (call/cc)
 (C4 <b20> <b14> <b15>) → (call/cc)
 (C4 <b22> <b14> <b15>) → (call/cc)
```

図 3.8: フロー解析の例 2

;; マクロ定義

```
(macro web-read
  (lambda (form)
    '(call/cc
      (lambda (k)
        (send-page (input-page-generator (k->url k)))
        (sucide)))))
(macro display
  (lambda (form)
    '(call/cc
      (lambda (k)
        (send-page (output-page-generator (k->url k) ,(cadr form)))
        (sucide)))))
```

;; 展開後の式

```
(lambda ()
  ((call/cc
    (lambda (k)
      (send-page (input-page-generator (k->url k)))
      (sucide)))
   (+   (call/cc
         (lambda (k)
           (send-page (output-page-generator (k->url k) ,(cadr form)))
           (sucide)))
        (call/cc
         (lambda (k)
           (send-page (output-page-generator (k->url k) ,(cadr form)))
           (sucide))))))
```

図 3.9: フロー解析の例 3

存在していた．それぞれの web-read 呼び出し式を通してプログラムの字面上で同じ call/cc 式が評価されるため，関数 web-read 内の call/cc は共有化できなかった．しかし図 3.9 では関数+の引数部分には 2 つの lambda 式が存在し，それぞれの lambda 式の中でプログラムの字面上異なる call/cc 式が呼び出される．そのためトップレベルからそれぞれの call/cc への制御フローは 1 つになり共有化を行うことができる．このマクロ展開されたコードに対して制御フロー解析を行う場合では，全ての call/cc を共有化する call/cc へ置き換えることができる．内部で call/cc を呼び出しておりさらに複数の場所で使われるような関数（例の display や web-read）について，関数ではなくマクロで定義することで共有化する場面を増やすことができると考えられる．

3.2 局所変数の解析

局所変数は異なるセッション間で常に同じ値のものと異なる可能性があるものとの 2 種類に分けられ，同じ値のものについては共有可能である．例えば図 3.10 のプログラムについて考える．まず関数 h が関数 g を引数に値 10 を与えて呼び出す．関数 g のフレームに含まれる局所変数 y は値が常に一定なので，異なるセッション間で共有してもよい．次に g の中の条件分岐により，処理の流れが 2 つに分かれる．関数 f は引数として 20 か 30 のどちらかが与えられる．関数 f のフレームに含まれる局所変数 x は値がセッションにより異なる可能性があるため，異なるセッション間で共有することは止める．

次のプログラムについて，局所変数 a が異なるセッション間で共有できるか考えてみる．

```
((lambda (a)
   (call/cc (k)
     ...))
```

```
(define (f x)
  (call/cc (lambda (return)
             (if (zero? x)
                 (return 1)
                 (/ 1 x)))))
(define (g y)
  (if <<条件>>
          (f (* 2 y))
          (f (* 3 y))))
(define (h)
  (g 10))
(h)
```

図 3.10: 局所変数の共有化についてのプログラム例

```
     <expression>)
```

まず<expression>が定数である場合について考える．lambda 式の引数には定数が与えられる．call/cc で生成される継続に保存される局所変数 a の値は常に一定である．次に<expression>が定数以外，変数や関数呼び出しである場合について考える．式が変数や関数呼び出しであるとき，式の値は実行してみなけらば分からない．1 つの方法としては局所変数の共有化の判断を，関数呼び出しの引数部分に置かれた式が定数か否かで行うことが考えられる．しかし引数に定数を与える呼び出しというのはそれほど多くない．そこで引数に変数や関数呼び出しが与えられた場合でも，局所変数を共有化する方法について考える．

図 3.11 で定義されている局所変数について，セッション間で共有できるか，それとも個別に保存しなければならないか考える．まず仮に<expression>が定数 1

```
(define (f a b)
  (display (- b a)))
(define (g c d)
  (f (+ 10 c) (+ 20 d)))
(define (h e)
  (g (* 2 e) (* 3 e)))
(h <expression>)
```

図 3.11: 局所変数を共有化できるプログラム例

である場合について考える．関数 h に 1 が与えられている．h の局所変数 e には必ず 1 が束縛される．次に関数 g に e に 2 を掛けた数値と，e に 3 を掛けた数値が与えられている．g の局所変数 c には必ず 2 が，局所変数 d には必ず 3 が束縛される．最後に関数 f に c に 10 を足した数値と，d に 10 を足した数値が与えられている．f の局所変数 a には必ず 12 が，局所変数 b には必ず 23 が束縛される．図 3.11 のプログラム<expression>が定数の場合は，全ての局所変数を共有化することができる．

次に read-value-from-client というユーザへ Web ページを送り，ページに入力された数値を受け取る関数の呼び出し式の場合について考える．関数 h に与えられる数値は，ユーザの入力に依存するため異なるセッション間で異なる可能性がある．<expression>が定数の場合とは逆に h の局所変数 e にどんな数値が束縛されるかは不定である．e の値が定まらないことで他の局所変数 a，b，c，d についても値が定まらない．図 3.11 のプログラムで<expression>が値の定まらない式の場合，全ての局所変数を異なるセッションで個別に保存しなけらばならない．

図 3.11 の<expression>が定数のときと値の定まらない式のときとでは関数 h に与える引数しか違いがない．<expression>が定数のときは全ての局所変数が共有化できる．これは値が常に定まっている変数のみが含まれた式の値もまた常に一定

であるためである．<expression>が値の定まらない式のときは全ての局所変数が共有化できない．これは値が不定である変数が含まれた式の値もまた不定であるためである．実際にプログラムを実行するまで関数 read-value-from-client の値などは不明であり，その値が含まれる式の値は分からない．しかしある式の値が一定か不定かの判断は，実行前に行うことができる．関数呼び出しの引数部分の式が一定か不定か判断できれば，式の値が束縛される局所変数について共有化の判断が行える．局所変数の共有化の判断をする方法として，静的解析により各局所変数の取りうる値の集合を求め，集合の要素が 1 つの局所変数については共有化できると判断する方法が考えられる．

第 4 章

共有化する継続の設計と実装

Scheme 処理系の 1 つである TUTScheme[10] に，共有化した継続を捕捉する関数 `call-with-current-shared-continuation`（以後，省略形の `call/csc` を使う）を実装した．なお TUTScheme の継続の実装戦略は incremental stack/heap 戦略 [9] である．

4.1 設計

図 4.2 のように継続オブジェクトをテンプレートと差分情報から構成する方法を提案する．具体的には異なるセッションの継続間でテンプレートを共有し，差分情報には復元したい内容とテンプレートとの差分を保存する．テンプレートには共通部分すなわち制御情報と共有可能な局所変数を，差分情報には非共通部分すなわち共有しない局所変数を保存させる．図 4.1 に従来法における継続オブジェクトと，提案法における継続オブジェクトを示す．提案法の方がサイズが大きいが，異なるユーザセッションで生成された継続間でテンプレートを共有するためユーザセッションの数が多ければ提案法の方がメモリ使用量の点では優位である．

プログラムの字面上で異なる `call/csc` 呼び出し式（以後 `call/csc` 式を使う）の呼び出しでは，異なるテンプレートを使用する．`call/csc` を呼び出す際にその呼び出しがプログラムの字面上でどの `call/csc` 式によるものか伝えるために，各 `call/csc` 式にはユニークな値を与えてやる必要がある．そこで全ての `call/csc`

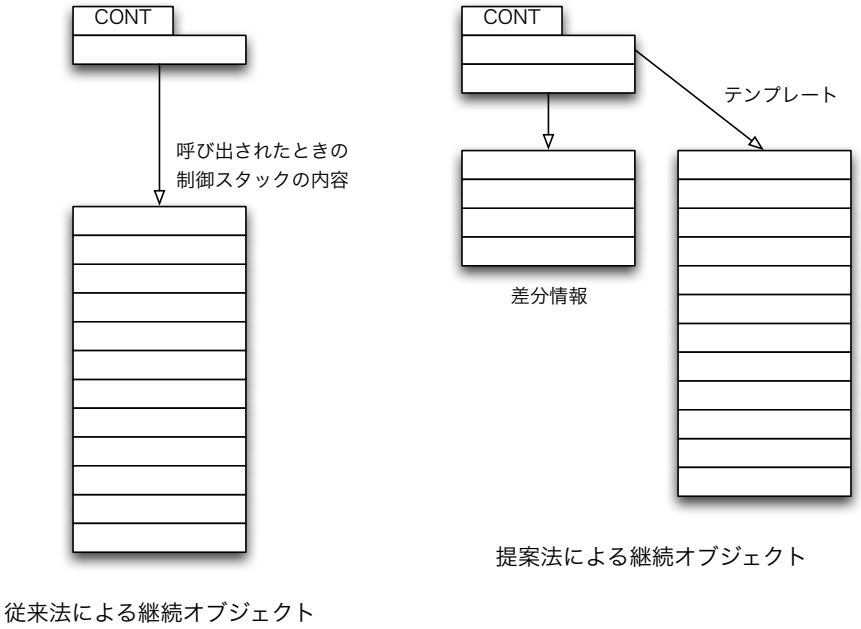

図 4.1: 従来法と提案法における継続オブジェクト

式には，呼び出されたときどのテンプレートを使えばよいのかを示した情報 ID を引数として与えることにする．またあらかじめ選び出しておいた共有しない局所変数の情報 lval-info も，各 call/csc 式に引数として与えることにした．共有化すると判断した call/cc 式は，次の call/csc 式に置き換えられる．fun は関数オブジェクトである．

```
(call/cc fun) -> (call/csc fun ID lval-info)
```

継続の共有化を自動的に行うためにはコンパイル前に静的解析を行い，その結果に基づきコード変換をする必要がある．プログラム中の各 call/cc 式について call/csc 式に置き換えられるかを調べるために，プログラムの制御フロー解析を行う． また個別に保存しなければならない局所変数を選び出すために，局所変数についての解析を行う．継続の共有化を自動的に行うためにコンパイル前に行う処

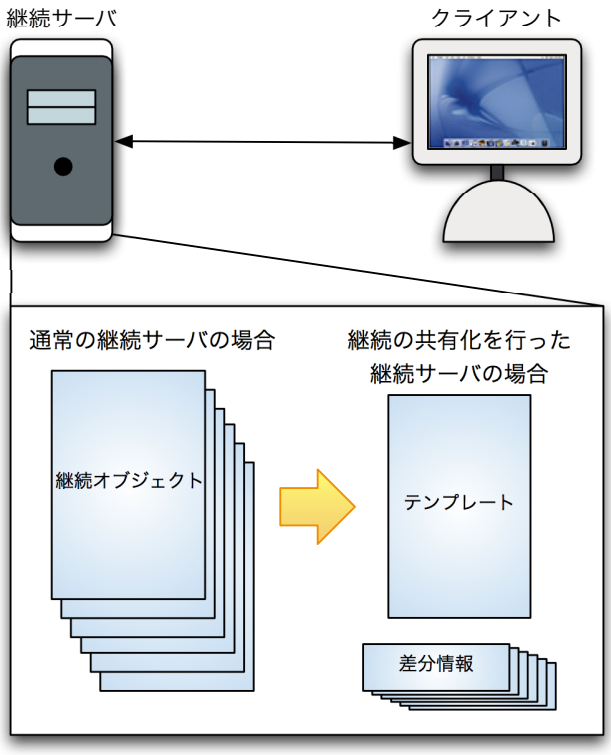
継続サーバ
クライアント
通常の継続サーバの場合
継続の共有化を行った
継続サーバの場合
継続オブジェクト
テンプレート
差分情報

図 4.2: 提案法の概観

理を以下に示す.

1. 全ての call/cc 式について，トップレベルからその call/cc 式に至るまでの制御フローの数を調べる.
2. 制御フローの数が 1 つの call/cc 式について，継続を共有化する call/csc 式に置き換える.
3. 各 call/csc は呼び出されるときに，どのテンプレートを使うのかを示した情報を受け取る必要がある．そこでユニークな ID を作り各 call/csc の第 2 引数に与える.
4. 各 call/csc について共有化できない局所変数を調べ，その情報を第 3 引数に与える.
5. 置き換え終了後コンパイル，実行する.

ただし，今回はコード解析による call/cc 式から call/csc 式への自動的な置き換えの実現が間に合わなかったため，目視で置き換えできるか判断している．また call/csc 式への書き換え，ID と lval-info の追加は手動で行った.

テンプレートはプログラム実行前に作るのではなく，実行中に作ることとした．call/csc は呼び出されるとまず，過去に同じ ID で呼び出されたことがあるかどうか調べる．具体的には ID と対応するテンプレートが既に存在するかによって判断する．初めての呼び出しであればそのときにテンプレートを作る．そうでなければテンプレートは既に作られているので差分情報だけを作る．call/csc が呼び出されたときに行う動作を以下にまとめる.

1. 第 2 引数として与えられた ID と対応するテンプレートが存在するか調べる.
2. ID と対応するテンプレートが存在しないときは通常の call/cc と同様の手順で現在の制御スタックの内容をヒープにコピーする．さらにヒープに退避した制御スタックの内容をテンプレートとして ID と関連づける.

3. 第 3 引数として与えられた共有しない局所得変数の情報を基に，共有しない局所変数の値をヒープにコピーする．それらをまとめて差分情報とする．

4. 新しく作られた差分情報と ID と対応したテンプレートのセットを継続オブジェクトとする．

5. call/csc の第 1 引数として与えられた関数のフレームを制御スタックに追加する．

6. 継続を制御スタックに追加する．

7. プログラムの実行を再開する．

継続が呼び出され，テンプレートと差分情報から構成した継続を再開するときに行う動作を以下に示す．

1. 継続オブジェクトに関連づけられたテンプレートを，制御スタックに戻す．

2. 継続オブジェクトに関連づけられた差分情報に含まれた，共有化できなかった局所変数の値を制御スタック上に上書きする．

3. プログラムの実行を再開する．

4.2 実装

図 4.3 に従来の call/cc により生成される継続オブジェクトのデータ構造を示す．incremental stack/heap 戦略では，フレームはスタック表現とリスト表現に分けて実現される．通常の実行はスタックを用いて行われるが，call/cc が呼び出され継続を生成するときにリスト表現に変換される．incremetal stack/heap 戦略ではヒープに退避した継続を再開するとき，全てのフレームを 1 度に制御スタックへ戻さずに，一部戻して残りは残しておく．制御スタックのフレームが全て消費さ

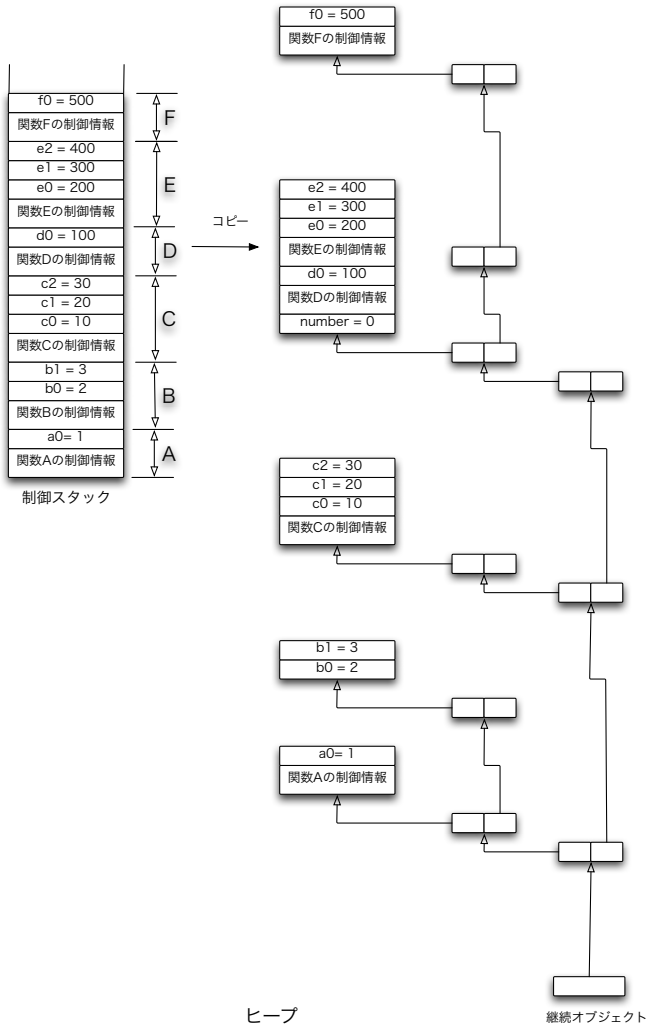

図 4.3: call/cc により生成される継続オブジェクト

れる，すなわちスタックアンダーフローが起こると新たにフレームの一部を制御スタックに戻す．スタックアンダーフローが起きた時に戻すべきフレームの先頭へのポインタは常に記録しておかなければならない．このポインタを保存する場所を more と呼ぶこととする．継続の実行もスタックアンダフローが発生した時と同様の処理を行えばよい．

図 4.4 にある call/csc 式を初めて呼び出すときに生成される継続オブジェクト

を示す．call/cc で生成される継続との違いは以下の通りである．

- 各フレームにはフレームに含まれる局所変数のうち，共有化しない局所変数の数とその局所変数の保存場所を示すオフセットが記録されている．
- 差分情報がフレームと同様のリスト表現により保存される．
- 継続オブジェクトはテンプレートと差分情報のセットである．

call/csc 式が初めて呼び出されるときの動作を以下で示す..

1. 初めて呼び出されたかどうかを確認するために，call/csc が呼び出されると直ぐに第 2 引数で与えられた ID と対応するテンプレートを確認する．ID と対応するテンプレートが何も指しておらず空のとき，call/csc 式が初めて呼び出されたと判断する．call/csc 式が初めて呼び出されるときはテンプレートと差分情報の両方を作る．
2. 従来の call/cc と同様にスタック表現のフレームをリスト表現に変換する．ただしリスト表現におけるフレームには，フレームに含まれる共有化しない局所変数の数とその位置情報を含む．共有化しない局所変数の数とその位置情報は call/csc 式の第 3 引数 lval-info を使い与えられる．
3. ID と対応するテンプレートにリスト表現されたフレームへのポインタを保存する．
4. 共有化しない局所変数の値をリスト表現で保存する．
5. テンプレートと差分情報のセットを継続オブジェクトとする．

図 4.5 に，テンプレートが既に存在する状態で call/csc 式を呼び出すときに作られる継続オブジェクトを示す．テンプレートが既に存在する状態で call/csc を呼び出すときの動作を以下で示す..

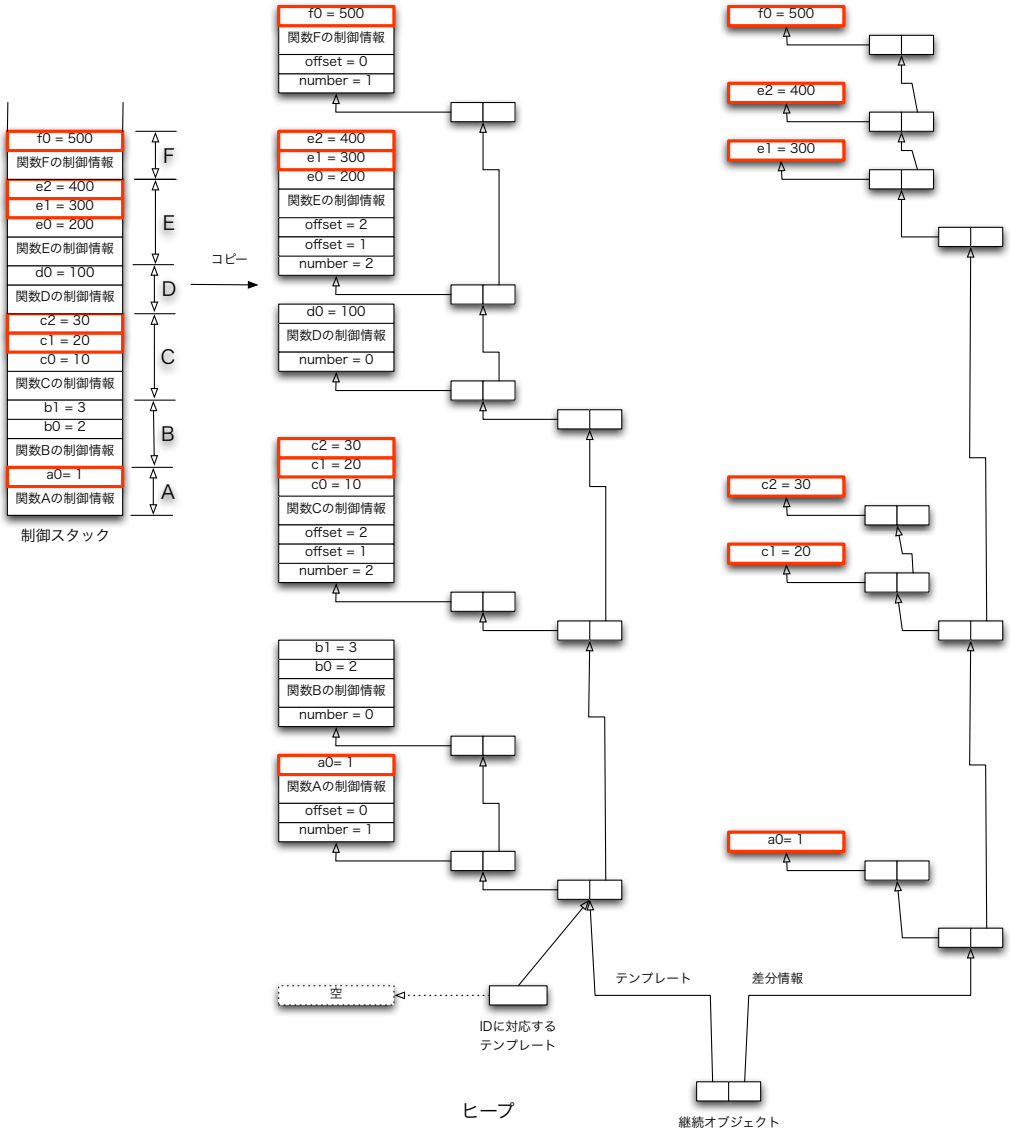

図 4.4: `call/csc` を初めて呼び出すときに作られる継続オブジェクト

1. ID と対応するテンプレートにはすでに作られたフレームへのポインタが存在するため，ポインタで指されたフレームを流用し差分情報のみを作る．
2. 共有化しない局所変数の値をリスト表現で保存する．
3. 差分情報と既に作られていたテンプレートとのセットを継続オブジェクトとする．

図 4.6 に，テンプレートと差分から構成した継続オブジェクトを再開するときの動作を示す．従来の `call/cc` により生成された継続を戻すときとの違いはフレームを制御スタックに戻す際，共有化しなかった局所変数があれば，差分情報から値を取り出し制御スタック上の局所変数の内容を上書きするという点である．

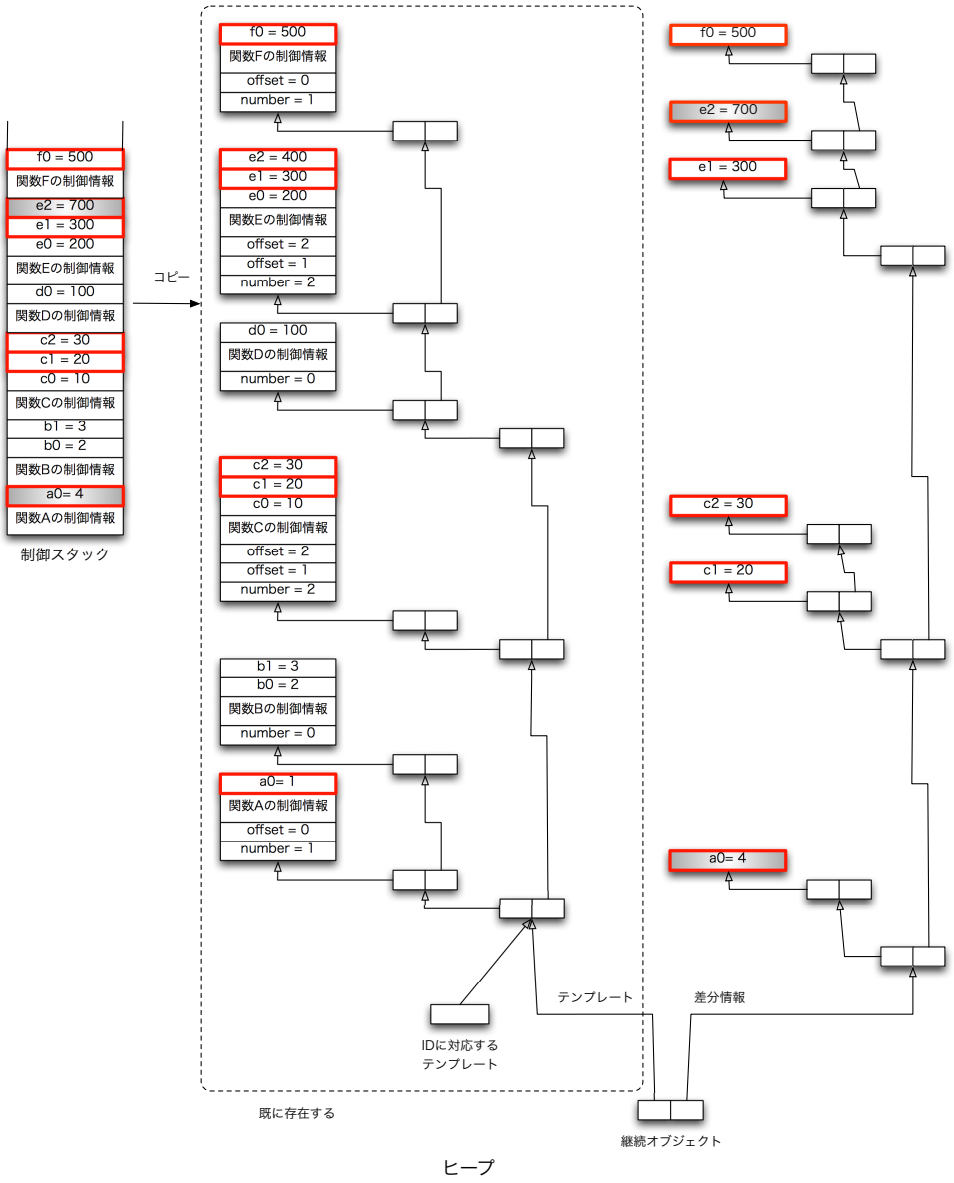

図 4.5: 2 度目以降で call/csc を呼び出すときに生成される継続オブジェクト

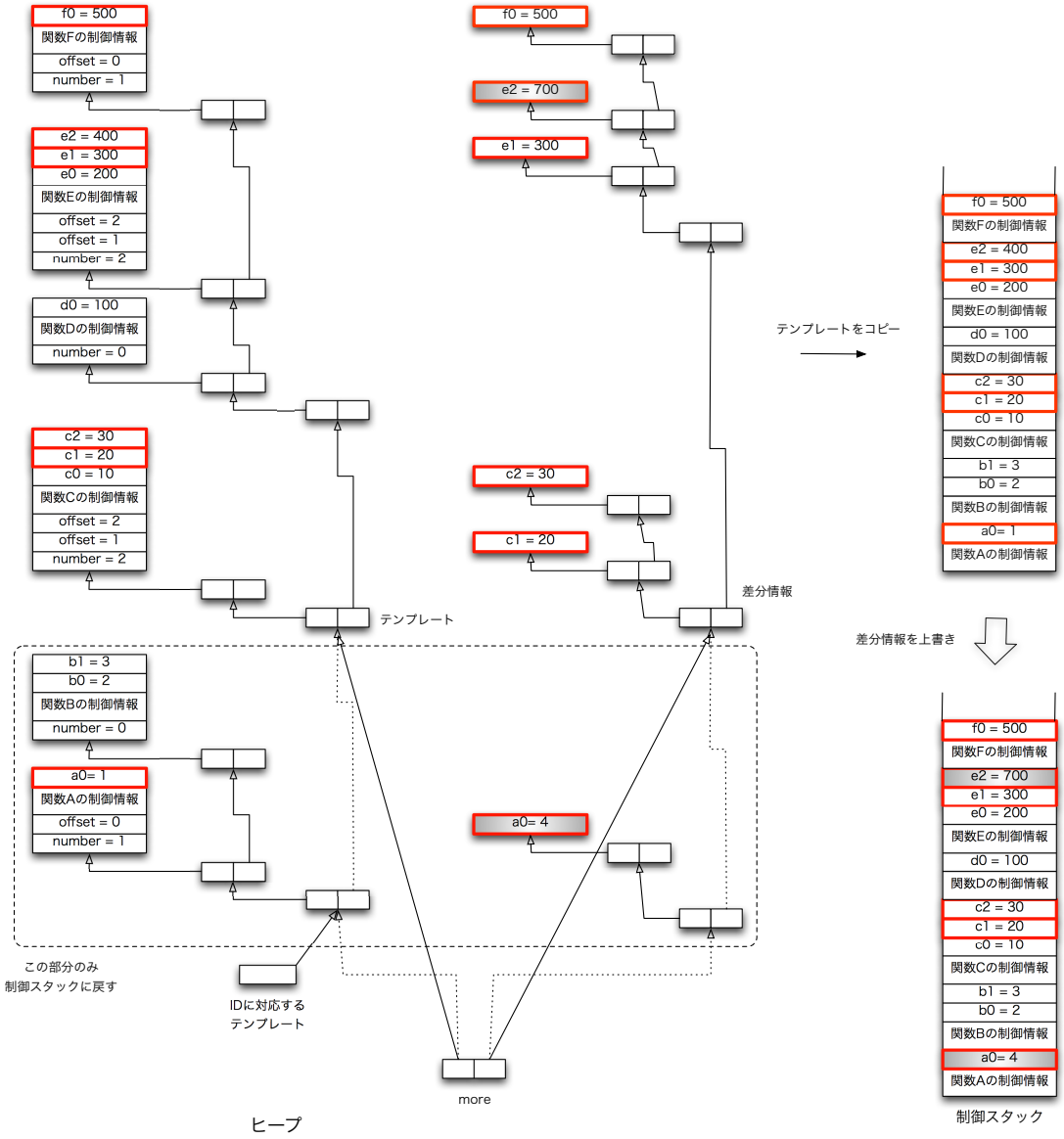

図 4.6: `call/csc` で生成された継続オブジェクトを再開するときの動作

第 5 章

評価

従来の call/cc と共有化機能を追加した call/csc について時間的なオーバーヘッドと，継続ベース Web サーバの利用をシミュレーションしたプログラムにおけるメモリ使用量を比較した．

5.1 時間的オーバーヘッド

継続オブジェクトの生成にかかる時間的オーバーヘッドを測定するためにシミュレーションを行った．継続の捕捉にかかる時間を知るために，図 5.1 ようなプログラムを用意した．1 つは call/cc が使われていないプログラム．もう 1 つは call/cc が関数 e の中で使われている以外は全く同じプログラム．この二つを実行し，call/cc が使われているプログラムの実行時間から call/cc が使われていないプログラムの実行時間を引けば継続の生成にかかった正味の時間が分かる．

さらに call/cc と同様に図 5.2 のようなプログラムを用意し，call/csc による継続の生成にかかった時間も求めた．call/csc はテンプレートと差分情報を作る場合と差分情報のみを作る場合で実行時間が異なる．そこで id を毎回変えることで常にテンプレートと差分情報両方を作るようにしたプログラムと，id を固定することで最初の 1 度だけテンプレートを作り後は差分情報のみを作るようにしたプログラムの 2 つを用意した．また call/csc は共有化しない局所変数の数によっても実行時間が異なる．そこで id の変化が違う二つのプログラムについてさらに

```
(define (calc_none)
  (let* ((h (lambda (e)
              (- e
                 ((lambda (k)
                    30) 'dummy))))
         (g (lambda (c d)
              (+ c (h 100) d)))
         (f (lambda (a b)
              (- a (g 10 20) b))))
    (f 200 5)
    ))

(define (calc_callcc)
  (let* ((h (lambda (e)
              (- e
                 (call/cc
                  (lambda (k)
                    30)))))
         (g (lambda (c d)
              (+ c (h 100) d)))
         (f (lambda (a b)
              (- a (g 10 20) b))))
    (f 200 5)
    ))
```

図 5.1: 継続の生成にかかる時間を測定するためのプログラム

```
(define (calc_callcsc_best)
  (let* ((h (lambda (e)
              (- e
                 (call/csc
                  (lambda (k)
                    30)
                  <id>
                  <lval-info>))))
         (g (lambda (c d)
              (+ c (h 100) d)))
         (f (lambda (a b)
              (- a (g 10 20) b))))
    (f 200 5)
    ))
```

図 5.2: call/csc の継続の生成にかかる時間を測定するためのプログラム

全ての局所変数をテンプレートに保存するプログラムと，全ての局所変数を差分情報に保存するプログラムの 2 通りを用意した．条件が異なる call/csc による継続の生成にかかる時間を求めるために合計 4 つのプログラムを用意した．

上記のプログラムを 100 回ループさせ継続を 100 個生成し，かかった時間を測定した．表 5.1 に実験結果を示す．program 列はプログラムの種類を表している．fix-id の付いたものは id が固定であるプログラム，change-id が付いたものは id が変化するプログラムである．template が付いたものは局所変数を全てテンプレートに保存するプログラム，diff が付いたものは局所変数を全て差分情報に保存するプログラムである．time 列はプログラムの実行にかかった時間．diff between no cont 列は継続の生成をしないプログラムの実行時間との差，すなわち継続の生成にかかった時間である．/100 は継続の生成にかかった時間を 100 で割った時間，つ

表 5.1: 継続オブジェクト生成に要した時間比較

program	time[ms]	diff between no cont[ms]	/100 [ms]
no cont	0.124	0	0
`call/cc`	1.319	1.195	0.1195
`call/csc` change-id template	1.358	1.234	0.1234
`call/csc` change-id diff	1.564	1.440	0.1440
`call/csc` fix-id template	1.101	0.977	0.0977
`call/csc` fix-id diff	1.471	1.347	0.1347

まり継続の生成 1 回あたりにかかる時間である．

表の一番右側の列の時間が，それぞれのケースで継続の生成一回あたりにかかった時間である．change-id の付いたプログラムの時間は，厳密には違うが差分情報のみを作る場合の継続の生成一回あたりにかかる時間とみなせる．初めて呼び出されたためにテンプレートを作る場合の `call/csc` 呼び出しでは，`call/cc` 呼び出しに比べ多少時間が多くかかる．しかし 2 回目以降の呼び出しで差分情報のみを作るときはむしろ時間が短縮されている．ただし全ての局所変数を共有しない場合の `call/csc` では，二回目以降の呼び出しでも `call/cc` よりも多く時間がかかり，初めての呼び出しではさらに多くの時間を要する．

この結果から分かることは

- ある `call/csc` が初めて呼び出されるときは，二回目以降のときよりもテンプレート作成の分多くの時間を要する．
- 共有しない局所変数が多いと 1 回目も 2 回目以降もより多くの時間を要する．
- 多くの時間を要すると言っても，従来の `call/cc` に比べてそれほど大きな差

は無かった．

5.2 メモリ使用量

複数のユーザが継続ベース Web サーバによるアドレス管理アプリケーションを利用する状況をシミュレーションした．シミュレーションのシナリオは各ユーザが 3 回の対話により自分の名前とアドレスをサーバに登録し，2 回の対話により自分の名前から対応する自分のアドレスを検索するというものである．なお継続オブジェクトには寿命をもうけ，古い物からゴミとなるようにしている．ユーザ数は 10，100，1000，10000，50000 人の 5 通りで行った．

図 5.3 から図 5.7 に異なるユーザ数でそのアドレス管理アプリケーションを利用したときの，対話によりメモリ使用量が増えていく様子を示す．ある程度メモリ使用量が増加すると急激にそれが下がるのは GC(ガーベジコレクタ) によりゴミとなったデータ（継続オブジェクトを含む）が回収されたためである．`call/csc` のメモリ使用量の増加は従来の `call/cc` のそれに比べ傾きが小さい．提案法による継続オブジェクト 1 つあたりのサイズは従来法によるそれよりも小さいことが分かる．またメモリ使用量の増加が緩やかなため，GC が起動される頻度も少ない．

以上の結果から，提案法の継続の共有化が継続ベース Web サーバのメモリ使用量削減に効果があることが確認できた．

5.3 セッション数が少ない場合

シミュレーションにおけるユーザ数すなわちセッション数が多いと，共有化の効果が十分発揮されメモリ使用量削減の効果が現れる．しかしセッション数が少ない（例えば 1）ときは，むしろ従来のものより多くのメモリを消費すると思われる．そこでアドレス管理アプリケーションの少ないユーザでの利用をシミューレションし，`call/cc` と `call/csc` についてメモリ使用量の比較を行った．図 5.8 に

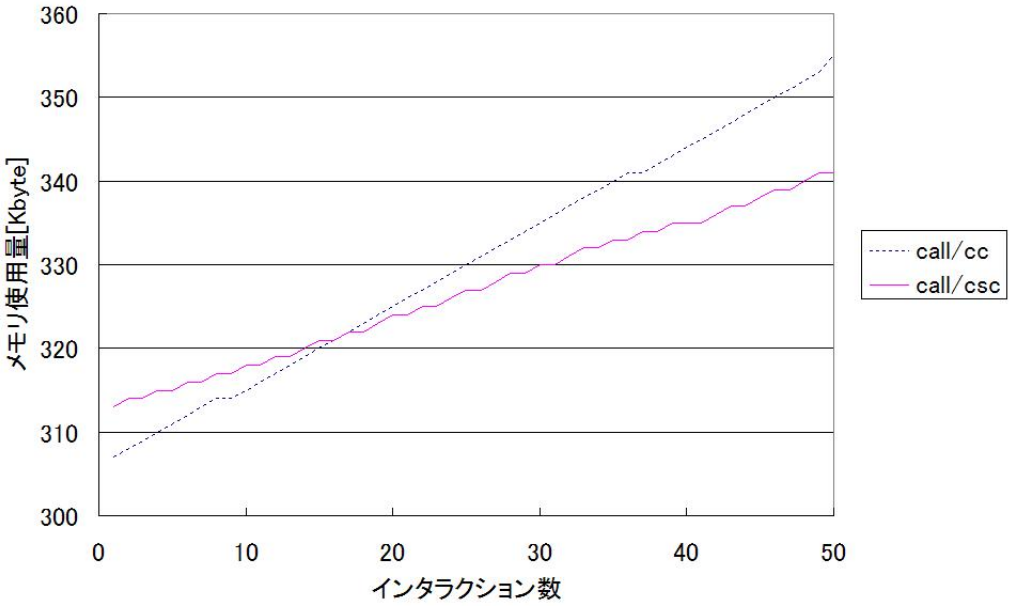

図 5.3: ユーザ数 10 人によるメモリ使用量の増加

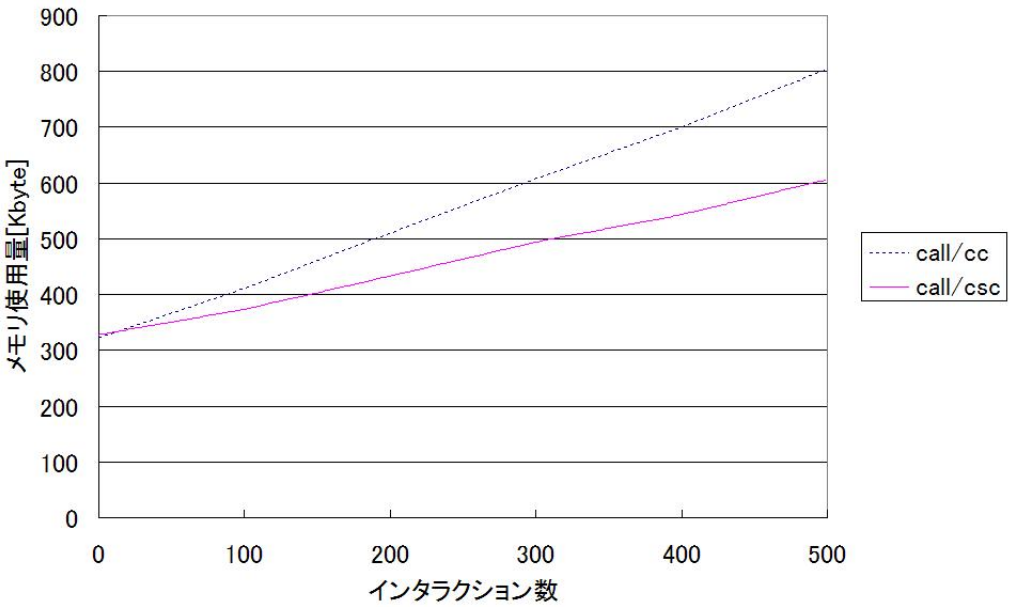

図 5.4: ユーザ数 100 人によるメモリ使用量の増加

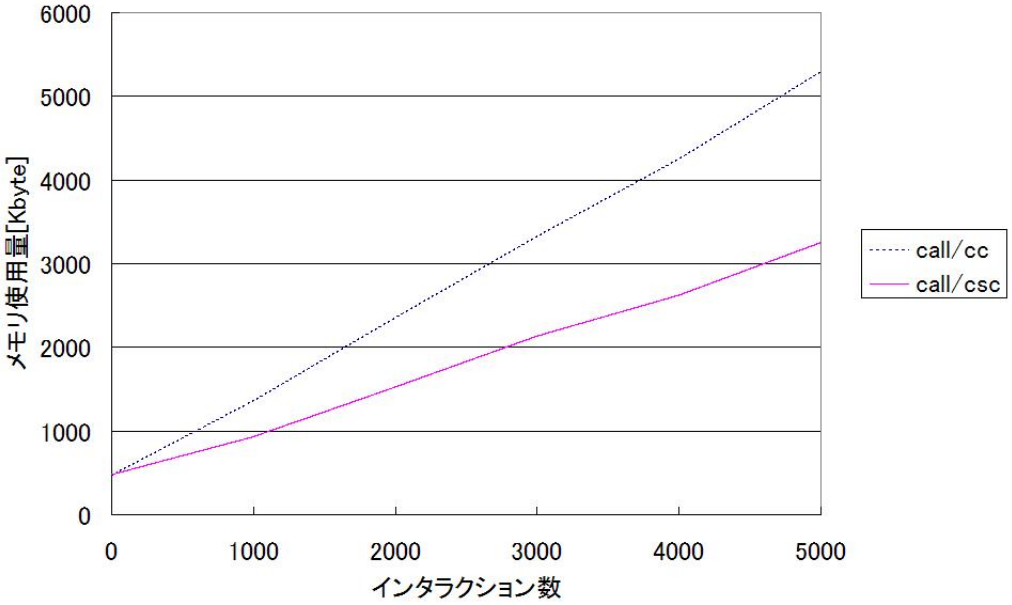

図 5.5: ユーザ数 1000 人によるメモリ使用量の増加

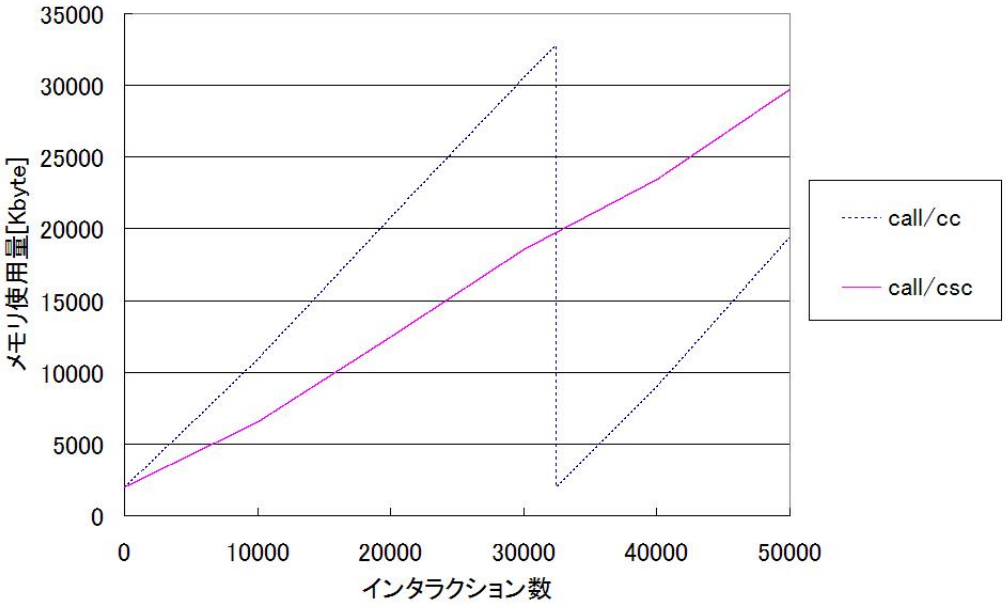

図 5.6: ユーザ数 10000 人によるメモリ使用量の増加

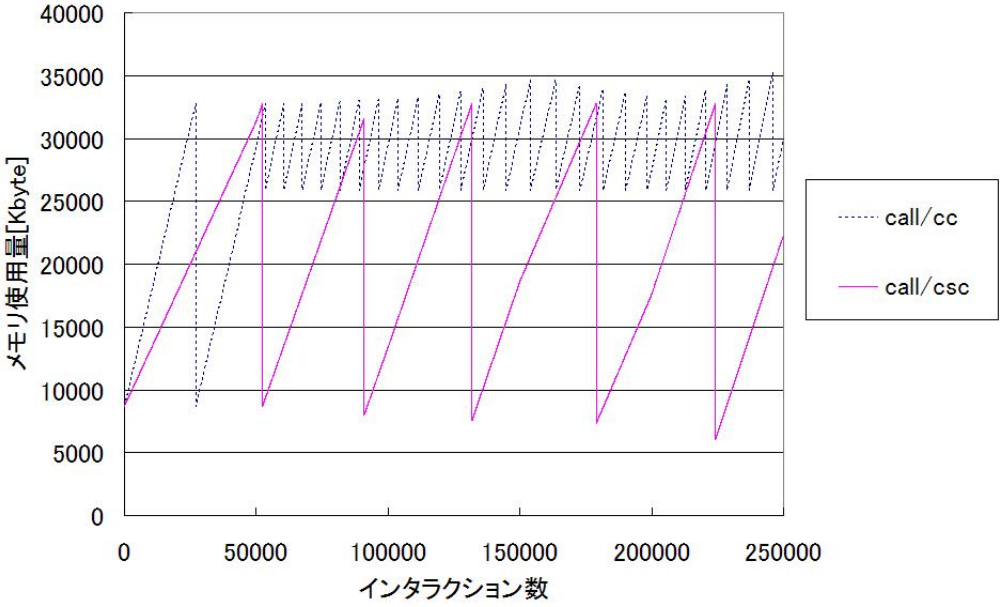

図 5.7: ユーザ数 50000 人によるメモリ使用量の増加

その結果を示す．ただし，call/csc は局所変数を全て共有化する場合と，全て共有化しない場合の 2 通りについて測定した．ユーザ数が 1 人の場合，共有化によるメモリ使用量削減の効果はなく，むしろ従来より多くのメモリを消費した．しかしユーザ数が 2 人以上ならば従来法よりも提案法のほうがメモリ消費量が少ないことが分かる．また共有化する局所変数の数がメモリ削減量に無視できない影響を与えていることも分かる．

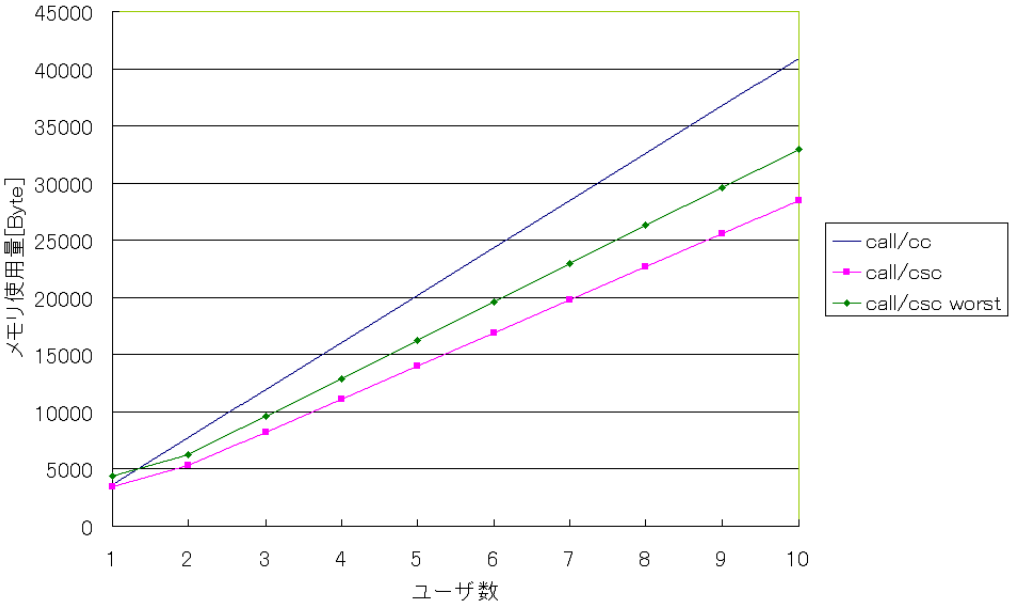

図 5.8: 少ないユーザ数でのメモリ使用量

第 6 章

おわりに

本研究では異なるセッションの call/cc で生成される継続について内容の一部を共有する余地があると考え，継続の共有化による継続ベース Web サーバのメモリ使用量削減を提案した．

字面上同じ場所で生成される継続でも内容が異なる可能性がある．継続生成時における制御スタックのフレームの並びが異なる場合共有化が困難であるため，フレームの並びが常に同じである継続生成についてのみ共有化することにした．継続生成時のフレームの並びは 1CFA という抽象解釈による制御フロー解析を使い調べる．また生成のたびに局所変数の値が異なる可能性もあるので，共有化できない局所変数については共有化せずに保存することにした．そのため共有化できない局所変数を調べるための解析も行う．

display や web-read などの内部で call/cc を呼び出しておりさらに複数の場所で使われるような関数について，関数ではなくマクロで実現することにより継続の共有化を行う場面を増やせると考えられる．継続ベース Web サーバによる Web アプリケーションのプログラミングについて，再帰呼び出し関数や相互再帰呼び出しを禁止するようなポリシーを導入することでも共有化する場面を増やせると考えられる．

本研究では Scheme 言語処理系に継続の共有化機能を実装した．さらに時間的オーバーヘッドや継続ベース web サーバにおけるメモリ使用量について，従来法と提案法を比較検討した．その結果提案法である継続の共有化が，継続ベース Web

サーバのメモリ使用量削減に効果があることが確認できた．他には，共有しない局所変数が多いと生成により多くの時間がかかること，継続生成にかかる時間は従来の call/cc と比べてそれほど大きく違わないこと，ユーザセッション数が少ないときむしろ従来より多くのメモリを消費することもあるということが分かった．

今後の課題は以下の通りである．

- 今は制御フロー解析を目視で確認し手動で call/cc から call/csc への置き換えをしているが，解析結果を利用し自動的に置き換えを行えるようにする．
- 共有化できない局所変数を選び出すための解析を実装する．
- 今は call/csc の引数を手動で追加しているが，解析結果を利用し自動的に行うようにする．
- 実際の継続ベース Web サーバによる多くの Web アプリケーションプログラムについて実験を行う．

謝辞

本研究を行う中で，いろいろな方々に力を貸していただきました．

指導教員の小宮常康先生には，日頃から研究にかかわる基礎知識や研究の進め方について丁寧なご指導を賜りました．疑問質問に対しては，毎回時間をかけて分かりやすく教えていただきました．ここに厚く御礼申し上げます．

佐藤喬先生，多田好克先生には研究を進める上で多くの有益な議論をさせていただきました．本当にありがとうございました．

そして二年間共に研究生活を送ってきた基盤ソフトウェア学講座の学生のみなさん．研究方針や方法論について様々なことを話し合ってきました．特に石橋崇さんには特に重要な議論や助言を頂きました．心より感謝いたします．

参考文献

[1] Queinnec, C.: “The Influence of Browsers on Evaluators or, Continuations to Program Web Servers”, *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, pp.23–33, 2000.

[2] McCarthy, J. and Krishnamurthi, S.: “Interaction-Safe State for the Web”, *Proceedings of the 2006 Scheme and Functional Programming Workshop*, pp.137–146.

[3] Krishnamurthi, S., Hopkins, P.W., McCarthy, Jay., Graunke, P.T., Pettyjohn, G. and Felleisen, M. : “Implementation and use of the PLT scheme Web server”, *Journal of Higher Order and Symbolic Computation*, pp.431–460, 2007.

[4] Khua Project: `http://www.kahua.org/`

[5] Queinnec, C.: “Inverting back the inversion of control or, Continuations versus page-centric programming”, *ACM SIGPLAN Notices Volume 38 , Issue 2 (February 2003)*, pp.57–64.

[6] Felleisen, M., Wand, M., Friedman, D. and Duba, B.: “Abstract Continuation: A Mathmetical Semantics for Handling Full Functional Jumps”, *Proceedings of the 1988 ACM conference on LISP and functional programming*, pp.52–62.

[7] 湯浅 太一: Scheme 入門, 岩波書店, 1991, ISBN 4000077015

[8] まつもと ゆきひろ: プログラミング言語 Ruby, オライリージャパン, 2009, ISBN 4873113946

[9] Clinger, W., Hartheimer,A. and Ost, E.: "Implementation strategies for first-class continuations", *Journal of Higher Order and Symbolic Computation*, pp.7–45, 1999.

[10] TUTScheme: `http://www.spa.is.uec.ac.jp/~komiya/download/`

[11] Shivers, O.: "The semantics of Scheme control-flow analysis", *Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pp.190–198 1991.