

お茶の水女子大学大学院 博士前期課程

人間文化創成科学研究科 修士論文

代数的効果を含むプログラムのステップ実行



著者氏名 : 理学専攻 情報科学コース 2 年 古川 つきの

指導教官 : 理学部 情報科学科 准教授 浅井 健一

令和 2 年 3 月

要旨

ステッパとはプログラミング教育やデバッグのために使うツールであり、プログラムが代数的に書き換わる様子を出力することで実行過程を見せるものである。

これまでに Racket の教育用に制限した構文などを対象にステッパが作られてきたが、継続を扱うことができる代数的効果のような、複雑なプログラム制御をする言語機能に対応したステッパは作られていなかった。そのような複雑な機能を含むプログラムの挙動を理解するのは特に困難なので、ステッパでプログラムの動きを観察できるようにしたい。本研究では例外処理や継続操作の機能を含む言語に対応したステッパを実装した。

ステッパは簡約のたびにその時点でのプログラム全体を出力するインタプリタなので、実行している部分式のコンテキスト (周りの式) の情報が常に必要になる。このコンテキストの情報をどのように得るかというのが、ステッパの実装における最大の問題である。本研究ではコンテキストを自分で設計する方法と機械的に導出する方法の 2 つを示す。

実装したステッパを利用してみると、長いプログラムを入力すると実行に長い時間がかかってしまい使用できないことが分かった。その解決の為、ステップ実行処理の終了を待たずに利用できる「incremental なステッパ」を実装した。

また、OCaml の一部の構文に対するステッパを大学の授業で学生に使用してもらい、その実行ログなどからステッパの教育上の効果について考察し、ステッパが有用な場面があることを示した。

キーワード：プログラミング教育、デバッグ、OCaml、代数的効果、CPS 変換、非関数化、インタプリタ

Abstract

A stepper, which display all the reduction steps of a given program, is a novice-friendly tool for understanding program behavior and debugging. So far, the tool was available only in the pedagogical languages of the DrRacket programming environment; therefore, we have not been able to step through programs that use advanced features such as exception handling. We implemented steppers for constructs including exception handling and various control operators.

To implement a stepper, we need information on the context surrounding the redex, because stepper is a kind of interpreter which outputs the whole programs at each reduction step. The biggest problem in implementing a stepper is how to get the information. In this paper, we suggest two ways for that.

Using the stepper, we realized that it takes so long time to evaluate long programs. In order to get over this problem, we designed an "incremental stepper" that can be used without waiting for the end of the step execution process. This made it possible to create a stepper in a server-client system.

We asked university students to use our OCaml stepper in a course, and examined the educational effects of the stepper from the execution logs and their comments. We showed that there were some situations where the stepper was useful.

Keywords: programming education, debugging, OCaml, algebraic effects, CPS transformation, defunctionalization, interpreter

目 次

第 1 章	序論	1
第 2 章	関連研究	5
2.1	ステップの実装	5
2.2	ステップと似たツール	5
2.3	ステップ実行によるプログラミング学習	6
2.4	algebraic effects	6
第 3 章	try-with のステップ実行	7
3.1	はじめに	7
3.2	対象言語とインタプリタの定義	7
3.3	ステップ関数の実装	9
3.3.1	一次元的なコンテキストでの実装	10
3.3.2	二次元的なコンテキストでの実装	11
3.4	実際の OCaml ステップ	15
3.4.1	型エラーについて	15
3.4.2	対象構文	16
3.4.3	関数適用のスキップ	17
3.5	この章のまとめ	17
第 4 章	algebraic effects のステップ実行	19
4.1	はじめに	19
4.2	algebraic effects とインタプリタの定義	19
4.2.1	algebraic effects	20
4.2.2	構文の定義	20
4.2.3	CPS インタプリタによる意味論	21
4.3	インタプリタの変換	24
4.3.1	非関数化	24
4.3.2	CPS 変換	25
4.3.3	非関数化	28

4.3.4	出力	28
4.3.5	CPS インタプリタに基づいたステップ	31
4.4	他の言語への対応	33
4.4.1	型無し λ 計算	33
4.4.2	try-with	33
4.4.3	shift/reset	33
4.4.4	Multicore OCaml	33
4.5	この章のまとめ	34
第 5 章	incremental なステップの実装	35
5.1	ステップの動作	35
5.1.1	DrRacket のステップ	35
5.1.2	incremental でない OCaml ステップ	35
5.1.3	提案するステップ	35
5.2	OCaml の attribute	36
5.2.1	式の attribute	36
5.2.2	プログラムの attribute	37
5.3	生じる問題と解決方法	38
5.3.1	情報の消失	38
5.3.2	表示の崩れ	39
5.4	λ 計算に対する実装	40
5.4.1	incremental でないステップ関数	41
5.4.2	incremental なステップ関数	42
5.4.3	実際のステップ	46
5.4.4	ツールの実装	48
5.5	予想される問題点とその経過	48
5.5.1	文字数の爆発	49
5.5.2	実行時間	49
5.5.3	関数適用評価スキップ後の前ステップ出力	50
5.6	この章のまとめ	51
第 6 章	評価	52
6.1	授業の内容	52
6.2	結果	53
6.2.1	ステップの使用状況	53
6.2.2	ステップの効果	54

6.2.3 学生による評価	56
第 7 章 結論	58
付 録 A 学生の実行ログから得られたデータ	63

目 次

1.1	DrRacket のステップ	1
1.2	DrRacket のステップを進めた様子	2
1.3	本研究で実装した OCaml のステップ	3
3.1	対象言語の定義	7
3.2	big-step インタプリタ	8
3.3	コンテキストと再構成関数 (試作版)	10
3.4	コンテキストと再構成関数 (最終版)	12
3.5	ステップ関数	14
3.6	関数 memo とメイン関数	15
3.7	実際のステップでプログラムを実行する様子	16
3.8	階乗関数をスキップする様子	17
3.9	関数適用をスキップするためのステップ関数	18
3.10	関数適用をスキップするためのステップ関数の出力	18
4.1	対象言語の構文	21
4.2	対象言語の定義	22
4.3	継続渡し形式で書かれたインタプリタ	23
4.4	非関数化後の継続の型	25
4.5	CPS インタプリタを非関数化したプログラム	26
4.6	CPS インタプリタを非関数化して CPS 変換したプログラム	27
4.7	2 回目の非関数化後の継続の型	28
4.8	CPS インタプリタを非関数化して CPS 変換して非関数化したプログラム	29
4.9	変換の後、出力関数を足して得られるステップ	30
4.10	継続を外側に拡張する関数	31
4.11	継続の情報を保持するための言語やコンテキストの定義	31
4.12	変換の結果得られた、CPS インタプリタを基にしたステップ関数	32
5.1	attribute を含む構文木の例	36
5.2	ハイライトのための attribute の利用	37

5.3	対象言語の定義	41
5.4	incremental でないステップ関数の実装	42
5.5	ステップ出力関数	43
5.6	実行の種類とステップ番号の定義	43
5.7	incremental なステップのための出力関数	44
5.8	incremental なステップ関数	47
5.9	ステップのスキップ機能	48
6.1	ステップが使用された回数	54
6.2	ステップの実行のうち各構文を含むプログラムの実行の回数	55
6.3	学生が正答するまでの時間の比較	56

表 目 次

6.1 学生による点数評価	57
A.1 ステップが使用された回数	63
A.2 拡張されたステップを利用した年と他の年の学生が正答するまでの時間の比較 . .	64

第1章 序論

書いたプログラムが思った通りの挙動をしない時、プログラマはデバッグをする必要がある。

単純なデバッグはプログラムを実行した際の出力から推測したりソースコードを眺めることで行われるが、そのようなデバッグは「ソースコードのどの部分が間違っているか」を示すものが無く、多くの時間や労力を要することがある。特にプログラミングにまだ慣れていない初学者にとっては、デバッグの経験や言語に対する理解が乏しい為、より困難な作業になると考えられる。

そこで色々な言語にデバッガが用意されているが、デバッガを利用するには、デバッガのコマンドの文字列や意味を覚えたり、ブレイクポイントを設定する箇所を考えたりといった、初学者にとってやはり困難な操作が必要になる。また、一般的なデバッガで表示されるのは「ソースコード中の実行中の行」であり、どこで今の関数を呼び出されたのか、この後どんな計算があるのかなどといったプログラム全体の流れが分かりにくい。

我々は、プログラミング初心者がデバッグをするのに最適な方法は、ステップを使うことだと考える。ステップは Racket 言語の統合開発環境 DrRacket において提供されているツール [2] である。ユーザがエディタにプログラムを書いてステップ起動ボタンを押すと、図 1.1 のようなウィンドウが表示される。図 1.1 は、再帰関数を用いて 2 の階乗を計算するプログラムを入力してステップを起動したときの様子である。ウィンドウには左右にそれぞれプログラムが表示されている。左はユーザが入力したプログラムと同じものであり、このプログラムで最初に簡約される式 (`fact 2`) が緑色にハイライトされている。右側のプログラムでは、ハイライトされた部分以外は左側と同じプログラムが表示されており、左側では緑色だった式 (`fact 2`) がその簡約結果に置



図 1.1: DrRacket のステップ



図 1.2: DrRacket のステッパを進めた様子

き換えられ、紫色でハイライトされている。

Step ボタンのうち右のボタンが「実行を進める」ボタンであり、押すと図 1.2 のような表示に切り替わる。最初 (図 1.1) は右側にあったプログラムと同じプログラムが左に表示され、次に簡約される部分式 $(= 2 0)$ が緑色にハイライトされており、右側には同様にその部分が簡約されて紫色になったプログラムが表示されている。当初 $(\text{fact } 2)$ だった式がその値である 2 になるステップまで、ボタンを押すと次々に簡約が行われてプログラムが変形していく様子を視覚的に見ることができる。

このように、プログラムを実行したときに、実行結果の値だけでなく、実行中にプログラムが代数的にどのように書き換えられていくかを見せるツールがステッパである。ステッパの操作は基本的に「前のステップへ」「次のステップへ」のボタンを押すのみであり、プログラミングや CUI での操作に慣れていない初心者でも使いやすい。

しかし、DrRacket のステッパが受け付けるのは Racket 言語のうちの一部の構文で構成された教育用の言語であり、例外処理などの制御オペレータがサポートされていない。初心者にとって理解しにくい言語機能を含むプログラムをステップ実行できるようにするために、本研究ではそういった複雑な言語機能に対応したステッパを実装する方法を示す。

我々は、ステッパの動作を以下の 3 つに分けて処理した。

1. 入力されたプログラムを構文解析して構文木を得る。
2. ステッパ関数に構文木を渡して、ステップを出力しながら入力プログラムを実行する。
3. 出力文字列をユーザの操作に従って 1 つずつ表示する。

この中で最も重要なのは 2 つ目、すなわち入力プログラムを表す構文木を受け取ってステップを表す文字列を出力する関数を作ることである。この関数は基本的には入力プログラムを実行する関数なので、インタプリタ関数の一種である。本論文ではこれ以降、この関数のことを「ステッパ関数」と呼ぶ。

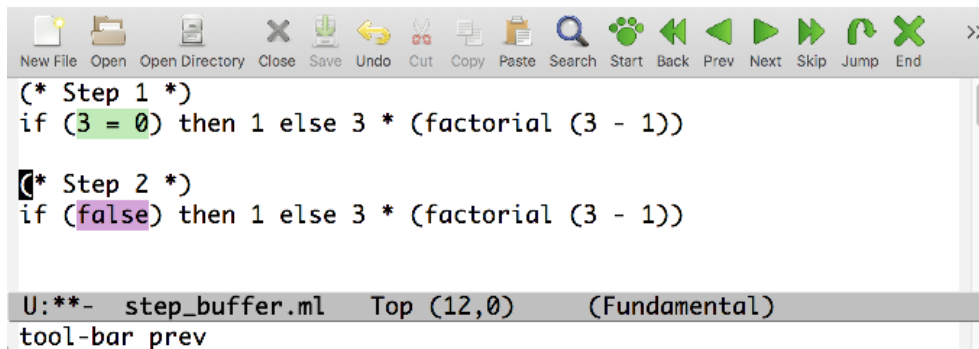


図 1.3: 本研究で実装した OCaml のステッパ

インタプリタ関数に出力機能を加えたものがステッパ関数であるが、プログラムがどのように書き換わっていくかを示すために、どのステップでも簡約している部分の式だけでなくプログラム全体を出力する必要がある。そのため、部分式を再帰的に実行する際にその部分式の外の式の情報 (コンテキスト) を保持する必要がある。

本論文では、まず例外処理の言語機能 `try-with` に対するステッパ関数を実装した。その際には、コンテキストを表現できるデータ型を定義し、それを利用してコンテキストの情報を保持するインタプリタを作ることによって実装することができた。その実装をもとにして、OCaml の `try-with` を含む一部の構文のステッパツール (図 1.3) も実装した。

しかし、コンテキストを表現するデータ型は言語ごとに定義しなければならず、その構造は自明ではない。そこで次に、継続を明示的に扱うことができる制御オペレータの代数的効果 (以下、algebraic effects) に対するステッパ関数を実装する際には、インタプリタ関数を機械的にプログラム変換することによってコンテキストの型とステッパ関数を導出することができた。

そして OCaml ステッパを実際に大学の授業において使用し、学生のプログラム実行ログやアンケートをもとに評価を行った。その結果、さまざまな構文の実行においてステッパの需要があったことや、正しいプログラムを有意に早く書けるようになる場合があったことから、ステッパの学習的効果を示した。

また、実装した OCaml ステッパは構文解析とステップ実行と表示の 3 つの手順を順番に同期的に行っており、大規模なプログラムをステップ実行する際に 2 つ目のステップ実行に長い時間がかかるため表示を始めることができないという問題点があった。その解決のため、1 ステップの実行と表示処理を交互に行うのでステップ実行処理の終了を待たずに利用できる「incremental なステッパ」を実装した。

プログラムを 1 ステップずつ実行することは small-step のインタプリタの動作そのものであるが、本論文では全体を通して big-step のインタプリタを基にしてステッパ関数を実装する。その理由は保守性と、プログラム全体の流れを分かりやすくするためにまとまったステップを省略する機能を実装するのに必要だからである。

以下に本論文の構成を示す。第 2 章では、関連研究について述べる。第 3 章では、try-with を含む言語を対象にしたステッパの実装方法を説明する。第 4 章では、algebraic effects 含む言語を対象にしたステッパ関数のプログラム変換による実装方法を説明する。第 5 章では、1 ステップずつ実行する incremental なステッパの実装方法を紹介する。第 6 章では、プログラミング学習におけるステッパの効果を考察する。そして第 7 章で、結論を述べる。

本論文の第 3 章と第 6 章の内容は、我々が 2019 年に発表した論文 [7] をもとにしている。

第2章 関連研究

2.1 ステッパの実装

ステッパはもともと Clements ら [2] によって Racket 言語の教育用に制限された構文に対して作られた。Clements らはステッパを作るために (i) 全ての簡約を正しい順番で表示する (ii) プログラム全体を再構成するための情報にアクセスすることが必要だと指摘した。この考えに基づいて、Clements らは以下の 3 つの関数を用いてステッパ関数を定義した。

- ブレークポイント挿入関数：簡約が行われる全ての部分にブレークポイントを設定する
- 注釈関数：適切なコンテキストを得られるようにプログラムに注釈を加える
- 再構成関数：スタックに蓄積されたコンテキストからプログラム全体を再構成する

Clements らがブレークポイントを挿入した箇所は我々がステップを出力する関数 `memo` を挿入する箇所と対応する。注釈関数について、Clements らは Racket 組み込みの関数

`w-c-m` ("`with-continuation-mark`") と `c-c-m` ("`current-continuation-marks`") を使ってコンテキストを操作している。`w-c-m` は我々がコンテキストを 1 層深く広げる操作に対応していて、`c-c-m` はスタックのコンテキストフレームを全て取得するために使われる。再構成関数は、我々が関数 `memo` 内で使用する関数 `plug` と同じ働きをする。

Cong と Asai [3] は、コンテキストの記録およびそこから再構成したプログラムの出力を限定継続のオペレータ `shift/reset` を用いて行い、ステッパを実装した。

Whittington と Ridge [13] は `small-step` のインタプリタを直接書くことで OCaml に対するステッパを実装した。

2.2 ステッパと似たツール

PLT Redex [5] は操作的意味論の形式化のための言語で、文法と簡約規則を定義できるようになっており、DrRacket のステッパを継承している。さらに、一画面の中に各ステップでのプログラムを配置し、あるプログラムが 1 ステップ簡約されて別のプログラムになることを矢印で表したグラフを表示する。これはより視覚的に簡約の様子を表すことができるほか、各矢印にそのステップの簡約規則が添えられているのでよりステップを辿りやすい。しかし、複数のステップの

プログラムが同じ画面に表示されている上に簡約が起こる部分式が強調されていないので、長いプログラムをステップ実行すると見づらくなってしまう。

根岸ら [14] の関数型言語 Haskell のデバッガフロントエンドは、一般的なデバッガの実行方法が通用しない遅延評価型言語のグラフィカルなユーザインタフェースでのステップ実行を含むデバッガ操作を可能にした。デバッガがブラウザ上で利用できるようになっており、DrRacket や本研究のステップと同様に各ステップでのプログラムを評価中の式をハイライトしながら表示する。実行するファイルや、ブレイクポイントをどの関数に設定するか、ステップ実行するか次のブレイクポイントまで実行するかなどといった設定を、ブラウザ上のボタンなどをクリックすることで行うことができる。デバッガのインタフェースでの本研究との違いは、根岸ら [14] のデバッガではブレイクポイントをユーザが簡単に設定できるのに対して、本研究ではブレイクポイントは自動的に全ての簡約基に設定され、ユーザは詳細に実行のしかたを決められない代わりに「次のステップ」「前のステップ」などのボタンを押すのみのより簡単な操作のみでステップ実行をすることができることである。

2.3 ステップ実行によるプログラミング学習

Tunnel Wilson ら [12] は、ステップの出力のような内容を学生に手書きで書かせることによって、学生がプログラムの実行のされかたをどのように理解しているか、どのような構文のステップの書き下しができないかといった傾向を分析した。

2.4 algebraic effects

本論文では algebraic effects に対する意味論を、big-step で書かれた、ハンドラ内の実行について CPS になっているインタプリタで定義する。Kammar ら [10] は small-step で意味論を与えた。Hillerström ら [9] は CPS による意味論を与えたが、入力言語を A-正規形に制限しているのに加え、継続がフレームのリストで与えられており、通常の CPS インタプリタにはなっていない。

上記以外の algebraic effects に関する研究としては、ハンドラの挙動が異なる shallow ハンドラの研究 [8] や algebraic effects を含むプログラムに関する論理関係を定義する研究 [1] などがあげられる。本論文で扱っているハンドラは従来の deep ハンドラである。shallow ハンドラにも対応できると考えているが、これは今後の課題である。

第3章 try-with のステップ実行

3.1 はじめに

この章では、try-with 構文を含む言語を対象としたステップ関数を OCaml で実装する方法を示す。ここでは簡単のため型無し λ 計算と try-with 構文のみから成る言語を対象として説明する。また、同様の方法で実装することができた OCaml の try-with を含む一部の構文を対象にしたステップ (以下、OCaml ステップ) について紹介する。

ステップ関数の実装は対象言語の通常のインタプリタ関数を拡張することによって行う。この章ではまず 3.2 節で言語とそのインタプリタを定義し、3.3 節でインタプリタを拡張してステップ関数を実装する。そして 3.4 節で OCaml ステップの対象言語や機能およびその実装について述べ、3.5 節でまとめる。

3.2 対象言語とインタプリタの定義

型無し λ 計算と try-with から成る言語の OCaml による定義を図 3.1 に示す。そしてこの言語に対する代入ベースで call-by-value かつ right-to-left のインタプリタは図 3.2 のように定義できる。

例えば関数適用 $e_1\ e_2$ を実行するときには、まず引数部分 e_2 を実行して、次に関数部分 e_1 を実行する。両方が値になったら β 簡約を行い、簡約後の式を実行する。引数 $expr$ が変数 Var だった場合はエラー終了するように書かれているが、これは入力されたプログラム全体が実行可能な閉じた式だった場合に関数 `eval` の引数に変数がくることはありえないからである。なぜありえないかというと、関数の本体の実行は必ず、引数を受け取って仮引数を実引数に置き換えた後に行うからである。

```
type e_t = Var of string          (* x *)
      | Fun of string * e_t      (* fun x -> e *)
      | App of e_t * e_t        (* e e *)
      | Try of e_t * string * e_t (* try e with x -> e *)
      | Raise of e_t            (* raise e *)
```

図 3.1: 対象言語の定義


```

(* 入力プログラムの例外の値を引数に持つ例外 *)
exception Error of e_t

(* 式を実行する *)
(* eval : e_t -> e_t *)
let rec eval expr = match expr with
| Var (x) -> failwith ("unbound variable: " ^ x) (* ここには来ない *)
| Fun (x, e) -> Fun (x, e)
| App (e1, e2) ->
  begin
    let v2 = eval e2 in
    let v1 = eval e1 in
    match v1 with
    | Fun (x, e) ->
      let e' = subst e x v2 in (* e[v2/x] *)
      let v = eval e' in
      v
    | _ -> failwith "not a function"
  end
| Try (e1, x, e2) ->
  begin
    try
      let v1 = eval e1 in
      v1
    with Error (v) ->
      let e2' = subst e2 x v in (* e2[v/x] *)
      eval e2'
  end
| Raise (e) ->
  let v = eval e in
  raise (Error (v))

(* 実行を開始する *)
(* start : e_t -> e_t *)
let start e =
  try
    eval e
  with
    Error v -> Raise v

```

図 3.2: big-step インタプリタ

入力プログラムの例外処理は、インタプリタで実際に (OCaml の機能の) `try` と `raise` を使って実行する。具体的には、式 `raise e` を実行する場合、まず `e` を実行してその値 `v` を得る。そしてメタレベルの (OCaml の) 例外 `Error v` を起こす。

式 `try e1 with x -> e2` の `e1` を実行している間に例外 `Error v` が発生した場合は、関数 `eval` は `e1` の中の残りの計算を無視して、`e2` の中の変数 `x` に値 `v` を代入した式 `e2[v/x]` の実行に移る。このように、OCaml の `try-with` 構文を利用して OCaml の `try-with` 構文と同じような動作を実現することができる。`try` 節の式のことを、プログラム内では `tryee` という変数名で表していることがある。これは、`try` ハンドラによって「`try` されている」ということを意図している。

メイン関数である `start` が `eval` を呼び出すとき、その呼び出しは `try-with` で囲まれている。`Error v -> raise v` とあるように、対応する `try` 節が無い `raise e` の実行結果は、`e` の実行結果を `v` として `raise v` とする。例えば、`2 + 3 + (raise 4) + 5` の実行結果は `raise 4` である。

3.3 ステッパ関数の実装

ステッパ関数は各簡約ステップでのプログラムの全体を表示する必要がある。例えば単純な算術式 `(1 + 2 * 3) + 4` をステップ実行するとき、我々が期待するステップ列は以下である。

$$\begin{aligned} & (1 + 2 * 3) + 4 \\ \rightarrow & (1 + 6) + 4 \\ \rightarrow & 7 + 4 \\ \rightarrow & 11 \end{aligned}$$

しかし、図 3.2 のプログラムに出力機能を加えても、直ちにこのようなステップ出力は得られない。関数 `eval` が部分式 `2 * 3` を実行している時のことを考えてみよう。式を出力する関数を用意すれば実行中の部分式 `2 * 3` は表示できるが、プログラム全体を再構成するには情報が足りていない。ここで欠けているのは `2 * 3` を囲んでいるコンテキストすなわち `(1 + [.]) + 4` である。(コンテキストの穴を `[.]` と表記する。) したがって、ステッパ関数を実装するためには、それまでの評価文脈を追い続ける必要がある。

そのために、コンテキストを表すデータ型を定義する必要がある。コンテキストはコンテキストフレームの列で表されるので、OCaml ではフレームを表すコンストラクタのリストとして定義することができる。しかし例外処理を含むプログラムの実行では、コンテキストをハンドラで区切って操作するため、より複雑なデータ構造として定義した方が良いことが分かった。3.3.1 節でリストとしてコンテキストを定義した場合について説明し、3.3.2 節で最終的な実装を示す。

```

(* コンテキストフレーム *)
type frame_t =
  | CAppR of e_t                (* e [.] *)
  | CAppL of e_t                (* [.] v *)
  | CTry of string * e_t        (* try [.] with x -> e *)
  | CRaise                     (* raise [.] *)

(* コンテキスト *)
type c_t = frame_t list

(* 式全体を再構成する *)
(* plug : e_t -> c_t -> e_t *)
let rec plug expr ctxt = match ctxt with
| [] -> expr
| CAppR (e1) :: rest -> plug (App (e1, expr)) rest
| CAppL (e2) :: rest -> plug (App (expr, e2)) rest
| CTry (x, e2) :: rest -> plug (Try (expr, x, e2)) rest
| CRaise :: rest -> plug (Raise expr) rest

```

図 3.3: コンテキストと再構成関数 (試作版)

3.3.1 一次元的なコンテキストでの実装

図 3.3 で、`frame_t` 型の代数的データとしてコンテキストフレームを定義している。各フレームがなんらかの部分式の実行を表しており、例えば `CAppR (e1)` は、関数部分が `e1` である関数適用の引数部分を実行していることを表す。評価文脈はこのフレームのリストで定義される。また、式 `expr` をコンテキストフレーム列 `ctxt` で囲んでプログラムを再構成する関数 `plug` も定義する。

これで、関数 `eval` がコンテキストを表す追加の引数を受け取るようにすれば、 $(1 + 2 * 3) + 4$ のような式のステップ表示ができるようになるはずである。例えば、部分式 $2 * 3$ を実行している時、その新しい引数の値は $[(1 + [.]); ([.] + 4)]$ であり、これを関数 `plug` に渡せばプログラム全体を得ることができる。

最終的に得られるステップ関数は、式を `control string`、評価文脈を継続とみなすと、本質的に CK 機械 [6] であるといえる。big-step インタプリタと small-step インタプリタの実行の対応を維持するため、抽象機械を直接実装するのではなく big-step インタプリタの拡張によって実装した。big-step インタプリタを基にすることで、ユーザが指定した関数適用をスキップすることができる (3.4.3 節で触れる)。

しかし、この実装は例外処理について正しく機能しない。例えば、`try (2 + 3 * (raise 4) + 5) with x -> x` について考える。この式をステップ実行する時、我々が期待するステップは以下である。

```

(* Step 0 *) try (2 + (3 * (raise 4)) + 5) with x -> x

```

```
(* Step 1 *) try (raise 4) with x -> x
(* Step 1 *) try (raise 4) with x -> x
(* Step 2 *) 4
```

最初の簡約はステップ関数に `raise 4` が渡されたときに起こる。しかし、最初のステップでハイライトされている式はそれよりも大きい式 $(2 + 3 * (\text{raise } 4) + 5)$ である。これは `raise 4` の実行で例外を起こす際に、この式の外側にある、`try` 節の内部のコンテキストを捨てるからである。コンテキストフレームは1つのリストに入っているので、この時点での関数 `eval` の第2引数は $[(3 * [.]); (2 + [.]); ([.] + 5); (\text{try } [.] \text{ with } x \rightarrow x)]$ であり、すなわち `try` 節の外側のコンテキストを含んでいる。例外処理を扱う場合には、`try` 節の内側と外側のコンテキストを区別する必要があるということである。コンテキストフレームのリストを単純に検索すれば最も内側のハンドラを見つけることができるが、ここでは構造的に区別できるようにコンテキストの型定義を変更していく。

3.3.2 二次元的なコンテキストでの実装

図 3.4 に、改良したコンテキスト定義を示す。新しい `frame_t` 型の定義は、`CTry` を含まないようになっている。`frame_t` 型のフレームは、`try-with` 構文を使うプログラムを実行において `try` 節の内部に限定されたコンテキストを表すために使う。次に、別の型 `ctry_t` が定義されており、これはメタコンテキストを表す型である。そしてコンテキストは `try` 節内部に限定されたコンテキストとメタコンテキストの2つ組として定義される。例として、以下のプログラムの中の `raise 4` を実行しているとき、

```
0 + (try 1 + 2 * (try (3 + raise 4) - 5 with x -> x + 6) with y -> y)
```

現在のコンテキストは以下のようになっている。

```
(([(3 + [.]); ([.] - 5)],
  CTry ("x", x + 6,
    ([2 * [.]; 1 + [.]],
      CTry ("y", y,
        ([0 + [.]], CHole))))))
```

このようにコンテキスト定義を改良すると、まず `frame_t` 型のコンテキストを使って `try` 節の式を再構成し、その後 `ctry_t` 型のコンテキストを使ってプログラム全体を作ることができる。上の例でいうと、ステップはまず $(3 + \text{raise } 4) - 5$ を再構成してハイライトしてから、プログラム全体を再構成している。

```

(* コンテキストフレーム *)
type frame_t =
  | CAppR of e_t          (* e [.] *)
  | CAppL of e_t          (* [.] v *)
  | CRaise                (* raise [.] *)

(* try フレーム *)
type ctry_t =
  | CHole                (* [.] *)
  | CTry of string * e_t * c_t (* try [.] with x -> e *)

(* コンテキスト *)
and c_t = frame_t list * ctry_t

(* try 節の式を再構成する *)
(* plug_in_try : e_t -> frame_t list -> e_t *)
let rec plug_in_try expr ctxt = match ctxt with
| [] -> expr
| first :: rest -> match first with
| CAppR (e1) -> plug_in_try (App (e1, expr)) rest
| CAppL (e2) -> plug_in_try (App (expr, e2)) rest
| CRaise -> plug_in_try (Raise (expr)) rest

(* プログラム全体を再構成する *)
(* plug : e_t -> c_t -> e_t *)
let rec plug expr (clist, tries) =
  let tryee = plug_in_try expr clist in
  match tries with
  | CHole -> tryee
  | CTry (x, e2, outer) -> plug (Try (tryee, x, e2)) outer

```

図 3.4: コンテキストと再構成関数 (最終版)

コンテキストフレームが蓄積されていく様子を具体的に紹介するため、例外処理を含んだ式の実行で関数 `eval` がどのような順で再帰呼び出しされるかを以下に示す。

```
eval (2 * (try 3 + (raise 4) - 5 with x -> x + 6)) ([], CHole)
eval (try 3 + (raise 4) - 5 with x -> x + 6) ([2 * []], CHole)
eval (3 + (raise 4) - 5) ([], CTry ("x", x + 6, ([2 * []], CHole)))
eval 5 ([3 + (raise 4) - []], CTry ("x", x + 6, ([2 * []], CHole)))
eval (3 + (raise 4)) ([[] - 5], CTry ("x", x + 6, ([2 * []], CHole)))
eval (raise 4) ([3 + []; [] - 5], CTry ("x", x + 6, ([2 * []], CHole)))
eval 4 ([raise []; 3 + []; [] - 5], CTry ("x", x + 6, ([2 * []], CHole)))
eval (4 + 6) ([2 * []], CHole)
```

最後のステップにおいて、`try` 節の内部のコンテキストすなわち `3 + (raise []) - 5` が捨てられている。

以上を踏まえ、ステップ関数を図 3.5 および図 3.6 に示す。この関数は通常の big-step インタプリタを以下の 2 点において拡張したものである (図中の灰色の部分)。(i) コンテキストを表す引数を受け取るようにする。(ii) 簡約が行われる全ての箇所での簡約前後のプログラムをそれぞれ出力する。

まず関数適用 `e1 e2` のケースの動作を見ると、以下のようにになっている。通常のインタプリタと同じように、最初に `e2` を実行してその後に `e1` を実行する。`e1` の実行結果が関数であれば、関数適用式は β 簡約基になっている。通常のインタプリタでは、そこで代入 `subst e x v2` をしてその結果を実行する。それに対しステップ関数では、図 3.6 で定義された関数 `memo` の呼び出しが追加されている。この関数は 3 つの引数を受け取る。見つかった簡約基と、その簡約結果と、現時点のコンテキストである。関数 `memo` はこれらの引数を受け取ったら、関数 `plug` と `print_exp` を使って簡約前と後のプログラムをそれぞれ再構成して出力する¹。プログラムを出力したら、普通の実行を再開する。

関数 `eval` では、これ以外にあと 3 箇所関数 `memo` が呼び出されている。それらはそれぞれ以下の簡約規則を表している。

- `try v with x -> e \rightsquigarrow v`
- `try raise v with x -> e2 \rightsquigarrow subst e2 x v`
- `... (raise v) ... \rightsquigarrow raise v`

¹ 実際の実装においては、簡約基と簡約後の式の範囲を表す情報をつけるのに OCaml の *attributes* という機能を利用している。ここでは `green expr1` は `expr1[@stepper.redex]` と出力される式を表し、`purple` についても同様である。(5.2.1 節で詳しく述べる。) ステップを表示する際に、Emacs Lisp のプログラムがその情報から適切に式をハイライトする。

```

(* ステップ関数 *)
(* eval : e_t -> c_t -> e_t *)
let rec eval expr ctxt = match expr with      (* コンテキストのための引数を増やす *)
| Var (x) -> failwith ("unbound variable: " ^ x)
| Lam (x, e) -> Lam (x, e)
| App (e1, e2) ->
  begin
    let v2 = eval e2 (add ctxt (CAppR e1)) in      (* コンテキスト情報を足す *)
    let v1 = eval e1 (add ctxt (CAppL v2)) in      (* コンテキスト情報を足す *)
    match v1 with
    | Lam (x, e) ->
      let e' = subst e x v2 in
      memo (App (v1, v2)) e' ctxt;                  (* 出力 *)
      let v = eval e' ctxt in                        (* コンテキスト情報を足す *)
      v
    | _ -> failwith "not a function"
  end
| Try (e1, x, e2) ->
  begin
    try
      let v1 = eval e1 (add_try ctxt x e2) in      (* コンテキスト情報を足す *)
      memo (Try (v1, x, e2)) v1 ctxt;              (* 出力 *)
      v1
    with Error (v) ->
      let e2' = subst e2 x v in
      memo (Try (Raise v, x, e2)) e2' ctxt;        (* 出力 *)
      eval e2' ctxt                                (* コンテキスト情報を足す *)
    end
| Raise (e0) ->
  let v = eval e0 (add ctxt CRaise) in              (* コンテキスト情報を足す *)
  begin match ctxt with
  | ([], _) -> ()
  | (clist, tries) ->
    memo (plug_in_try (Raise v) clist)              (* 出力 *)
      (Raise v)
      ([], tries)
  end;
  raise (Error (v))

```

図 3.5: ステップ関数

```

(* 簡約前の式と簡約後の式とコンテキストを受け取ってステップを出力する *)
(* memo : e_t -> e_t -> c_t -> unit *)
let memo expr1 expr2 ctxt =
  print_exp (plug (green expr1) ctxt);
  print_exp (plug (purple expr2) ctxt)

(* ステップ実行を始める *)
(* start : e_t -> e_t *)
let start e =
  try
    eval e ([], CHole)      (* 空のコンテキスト *)
  with
    Error (v) -> (Raise v)

```

図 3.6: 関数 memo とメイン関数

2 回目の簡約は必ず 3 回目の簡約の直後に起こるが、それにもかかわらずこれらを別の規則として扱っていることに注目してほしい。その理由は、例えば $3 + (\text{raise } 4) - 5 \rightsquigarrow \text{raise } 4$ のように対応する try 節が無い raise 式の簡約に 3 回目の規則が必要だからである。また、これらの規則を別にしておくのは教育上の意義もある。それは、例外処理が「コンテキストを捨てること」と「例外の値を代入すること」の 2 つの動作で成り立っていることが明確にステップに表れることである。

3.4 実際の OCaml ステッパ

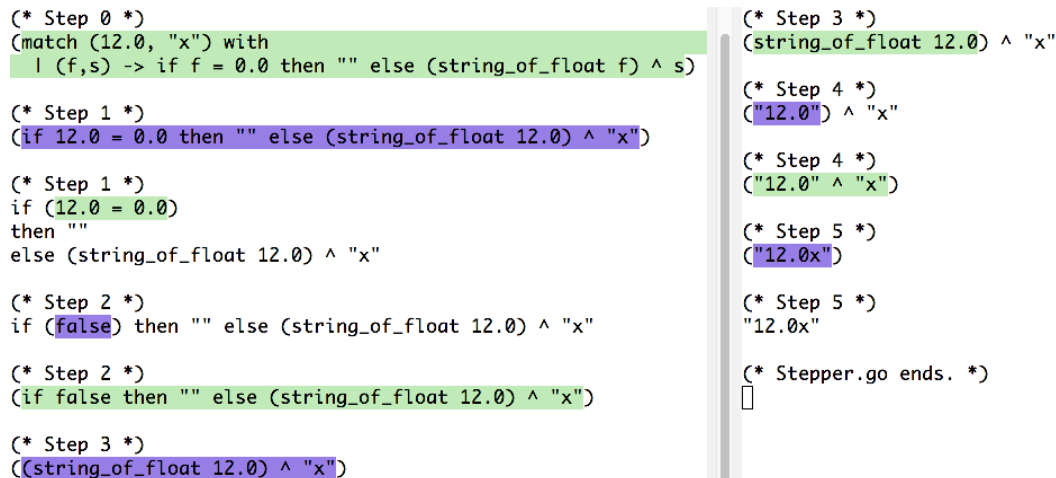
我々が実装した OCaml ステッパは、3.3 節で示した try-with と型無し λ 計算に対するステッパと比べて以下のような特徴がある。

- 型エラー等を想定しない。
- 授業「関数型言語」で使用する構文のほぼ全てに対応している。
- 関数適用が β 簡約されるステップからその式が値になるステップまで飛ばす機能がある。

この節ではそれぞれについて述べる。

3.4.1 型エラーについて

OCaml ステッパでは、まず入力プログラムを OCaml のパーザを利用して構文木にし、型チェックをする。未定義変数エラーを含むシンタックスエラーや型エラーになった場合はステッパ本体のプログラムを起動せず、OCaml が示すエラーメッセージを表示する。よって、コンパイルエラーがあるプログラムの構文木がステッパ関数に渡されることは無いという仮定の上で実装されている。



```

(* Step 0 *)
(match (12.0, "x") with
 | (f,s) -> if f = 0.0 then "" else (string_of_float f) ^ s)

(* Step 1 *)
(if 12.0 = 0.0 then "" else (string_of_float 12.0) ^ "x")

(* Step 1 *)
if (12.0 = 0.0)
then ""
else (string_of_float 12.0) ^ "x"

(* Step 2 *)
if (false) then "" else (string_of_float 12.0) ^ "x"

(* Step 2 *)
(if false then "" else (string_of_float 12.0) ^ "x")

(* Step 3 *)
((string_of_float 12.0) ^ "x")

(* Step 3 *)
(string_of_float 12.0) ^ "x"

(* Step 4 *)
("12.0") ^ "x"

(* Step 4 *)
("12.0" ^ "x")

(* Step 5 *)
("12.0x")

(* Step 5 *)
"12.0x"

(* Stepper.go ends. *)

```

図 3.7: 実際のステップでプログラムを実行する様子

る。ゼロ除算等の実行時エラーに関しては OCaml では例外の発生として処理されるので、そのようにステップ実行処理を続行する。ただし、例外が捕捉されないままその文の実行が終わってしまった場合はそこでステップ実行を終了する。

3.4.2 対象構文

このステップは以下の構文に対応している。

- 整数、実数、真偽値、文字、文字列型
- リスト、組、レコード
- ユーザ定義型
- 条件分岐、変数定義、再帰関数定義、パターンマッチ
- List モジュール、ユーザ定義モジュール
- 例外処理
- 標準出力関数、逐次実行
- 書き換え可能な変数、配列

標準出力や書き換え可能な変数を含むプログラムでは、標準出力された文字列や変数に格納された値などの「状態」をインタプリタが保持する必要がある。状態の情報はステッププログラム内の書き換え可能なグローバル変数の中に格納することで実装した。

OCaml ステップは他に授業で利用する一部の演算子などに対応している。図 3.7 に実際のステップ列の例を示す。

<pre>(* Step 0 *) let fact3 = (factorial 3) (* Step 1 *) let fact3 = (if 3 = 0 then 1 else 3 * (factorial (3 - 1)))</pre>	<pre>(* Step 0 *) let fact3 = (factorial 3) (* Step 18 *) let fact3 = (6)</pre>
--	--

図 3.8: 階乗関数をスキップする様子

3.4.3 関数適用のスキップ

プログラムのステップ数が多くなると、デバッグの目的でステップ実行をするときに膨大な計算ステップを1つ1つ追って見るのは効率が悪く、次第に実用に耐えるものではなくなってしまふ。そこで本研究では、「関数適用」を基準としてステップを飛ばす機能を追加した。

図 3.8 に、階乗関数のスキップの様子を示す。左の状態で「skip」ボタンを押すと、関数の中身 `if 3 = 0 then 1 else 3 * (factorial (3 - 1))` が6になるまでの実行ステップを見ることができ、右の画面に移ることができる。これによって見たいステップだけに注目することができ、実行の全体の流れを把握しやすくなる。

このスキップ機能は、3章で try-with に対応したステッパ関数(図 3.5) の eval 関数を図 3.9 のように変更することで実装できる。本研究では、飛ばすステップ列を (`* Application n start *`) と (`* Application n end *`) という2つの文字列で挟む方法をとった。`n` は関数適用の実行が始まるステップの番号である。あるステップで実行が始まる関数適用は必ず1つ以下なので、このステップ番号によって関数適用を一意に特定することができる。関数 `apply_start : unit -> int` が前者を出力する関数、関数 `apply_end : int -> unit` が後者を出力する関数である。表示処理をする Emacs Lisp プログラムがこれらの文字列を検索し、間のステップを隠す処理をする。スキップ機能を追加したステッパ関数の出力は例えば図 3.10 のようになる。

3.5 この章のまとめ

この章では、例外処理の構文 try-with に対応したステッパ関数を、インタプリタを拡張することによって実装する方法を示した。

まず、try-with 構文に対する通常の big-step インタプリタは実際の OCaml の try-with を用いて実装することができた。

ステッパ関数への拡張においては、各ステップでのプログラム全体を表示するために、実行中の部分式のコンテキストの情報が必要である。例外発生によって一度に捨てられる範囲のコンテキスト、すなわち try 節の内側のコンテキストフレームをひとまとまりにして、そのリストを要素に持つリストのような構造としてコンテキストを定義した。それをインタプリタの再帰の構造にしたがって要素を足しながら引数に渡すようにすると実行中の部分式のコンテキストの情報が得られるようになり、プログラム全体が出力できるステッパ関数が得られた。

```

let rec eval expr ctxt = match expr with
  ...
| App (e1, e2) ->
  begin
    let v2 = eval e2 (add ctxt (CAppR e1)) in
    let v1 = eval e1 (add ctxt (CAppL v2)) in
    match v1 with
    | Lam (x, e) ->
      let e' = subst e x v2 in
      let apply_num = apply_start () in
      memo (App (v1, v2)) e' ctxt;
      let v = eval e' ctxt in
      apply_end apply_num;
      v
    | _ -> failwith "not a function"
  end
| ...

```

(* 開始マークを出力 *)

(* 終了マークを出力 *)

図 3.9: 関数適用をスキップするためのステップ関数

```

(* Step 0 *) (f 4) + 10 * 100
(* Step 1 *) (f 4) + 1000
(* Application 1 start *)
(* Step 1 *) f 4 + 1000
(* Step 2 *) (4 * 2) - 1 + 1000
(* Step 2 *) (4 * 2 - 1) + 1000
(* Step 3 *) (8 - 1) + 1000
(* Step 3 *) 8 - 1 + 1000
(* Step 4 *) 7 + 1000
(* Application 1 end *)
(* Step 4 *) 7 + 1000
(* Step 5 *) 1007

```

図 3.10: 関数適用をスキップするためのステップ関数の出力

第4章 algebraic effects のステップ実行

4.1 はじめに

プログラムのある時点での残りの計算のことを「継続」という。たとえばプログラム $1 + (2 + 4)$ の $2 + 4$ を計算している時点での継続は「今の計算の結果を 1 に足す」という計算である。継続のうち一部を切り取った計算を限定継続という。3章で扱った例外処理機能 `try-with` は、「例外が起きたらその時点での継続のうち `try` までの限定継続を捨てる」ものだといえる。一方、`shift/reset` [4] や algebraic effects [11] といった言語機能では、継続を関数のようなものとして変数に束縛し、値に適用することができる。このような機能を含むプログラムの制御の移り変わりは複雑なので、これを対象にしたステップ関数の実装は `try-with` の場合よりも困難である。

3章で示したステップ関数の実装方法では、プログラム出力に必要なコンテキスト情報の構造を自分で定義し、インタプリタに追加して引数で適切にフレームを足していく必要があった。単純な言語に対するステップであれば、手動でコンテキストの型を定義するのは簡単だが、言語が複雑になってくると必ずしもこれは自明ではない。実際、3章でのコンテキストは `try-with` 構文で区切る必要があったため構造が一次元的でなく、リストのリストになった。algebraic effects などが入った場合、どのようなコンテキストを使えば良いのかは別途、考慮する必要がある。そこで、本研究ではインタプリタを機械的にプログラム変換することでステップおよび必要なコンテキストの構造を導出した。本章では algebraic effects を含む言語についてこの導出の過程を説明する。

まず 4.2 節で algebraic effects を含む言語およびそのインタプリタを定義し、4.3 節でインタプリタを変換してステップを得る過程を説明する。4.4 節では他のいくつかの言語に対するステップ関数を同様の変換によって得ることについて議論し、4.5 節でまとめる。

またこの章でのもう 1 つの貢献として、algebraic effects の big-step インタプリタを定義した (4.2.3 節)。これは、3章でのステップ実装と同様に big-step のインタプリタを元にしてステップ関数を作成するというアプローチをとったため、元のインタプリタが必要になったからである。

4.2 algebraic effects とインタプリタの定義

この節では、algebraic effects を導入した後、型無し λ 計算と algebraic effects からなる言語を示し、そのインタプリタを定義する。

4.2.1 algebraic effects

algebraic effects は、例外や状態などの副作用を表現するための一般的な枠組で、副作用を起こす部分（オペレーション呼び出し）と処理する部分（ハンドラ）からなる [11]。特徴は、副作用の意味がそれを処理するハンドラ部分で決まるところである。例えば、以下のプログラムを考える。

```
with {return x -> x;
      op(x; k) -> k (x + 1)}
handle 10 + op(3)
```

`with h handle e` は、`h` というハンドラのもとで式 `e` を実行するという意味である。`e` の部分を見ると `10 + op(3)` とあるので加算を行おうとするが、そこで `op(3)` というオペレーション呼び出しが起こる。オペレーション呼び出しというのは副作用を起こす命令で、直感的にはここで例外 `op` を引数 3 で起こすのに近い。使えるオペレーションはあらかじめ宣言するのが普通だが、本論文では使用するオペレーションは全て定義されていると仮定する。

オペレーション呼び出しが起こると、プログラムの制御はハンドラ部分に移る。ハンドラは正常終了を処理する部分 `return x -> ...` とオペレーション呼び出しを処理する部分に分かれている。正常終了する部分は `with h handle e` の `e` 部分の実行が終了した場合に実行され、`x` に実行結果が入る。上の例なら、その `x` がそのまま返されて、これがプログラム全体の結果となる。

一方、`e` の実行中にオペレーション呼び出しがあった場合は、オペレーション呼び出しの処理が行われる。まず、呼び出されたオペレーションが処理するオペレーションと同じものかがチェックされる。異なる場合は、そのオペレーションはここでは処理されず、さらに外側の `with handle` 文で処理されることになる。（最後まで処理されなかったら、未処理のオペレーションが報告されてプログラムは終了する。）一方、ここで処理すべきオペレーションと分かった場合には、矢印の右側の処理に移る。ここで、`x` の部分にはオペレーションの引数が入り、`k` の部分には「オペレーション呼び出しから、この `with handle` 文までの限定継続」が入る。`k` に限定継続が入るところが例外とは異なる部分である。上の例では、矢印の右側が `k (x + 1)` となっているので、`x` の値である 3 に 1 が加わった後、もとの計算である `10 + [...]` が再開され、全体として 14 が返ることになる。

algebraic effects の特徴は、オペレーション呼び出しの意味がハンドラで決まる部分にある。`op(3)` とした時点ではこの処理の内容は未定だが、ハンドラ部分に `k (x + 1)` と書かれているため、結果として `op` は 1 を加えるような作用だったことになる。

4.2.2 構文の定義

型無し λ 計算と algebraic effects からなる対象言語を図 4.1 の `e` と定義する。`h` 型のハンドラのエレメント節に出てくる `op` たちは互いに全て異ならなくてはならない。

<code>v</code>	<code>:=</code>	(値)
	<code>x</code>	変数
	<code> fun x -> e</code>	関数
<code>e</code>	<code>:=</code>	(式)
	<code>v</code>	値
	<code> e e</code>	関数適用
	<code> op e</code>	オペレーション呼び出し
	<code> with h handle e</code>	ハンドル
<code>h</code>	<code>:=</code>	(ハンドラ)
	<code>{return x -> e;</code>	<code>return</code> 節
	<code>op(x; k) -> e; ...; op(x; k) -> e}</code>	オペレーション節 (0 個以上)

図 4.1: 対象言語の構文

4.2.3 CPS インタプリタによる意味論

この節では、algebraic effects を含む言語に対する意味論を与える。オペレーション呼び出しにより非局所的に制御が移るので、意味論は CPS インタプリタを定義することで与える。対象言語の OCaml による定義を図 4.2 に示す。ここで `k` は各ハンドラ内部の限定継続を表す。また、`a` は `handle` 節内の式の実行が正常終了したのかオペレーション呼び出しだったのかを示す型である。

図 4.2 の言語に対する call-by-value かつ right-to-left の代入ベースのインタプリタを図 4.3 に定義する。ただし、関数 `subst : e -> (string * v) list -> e` は代入のための関数であり、`subst e [(x, v); (k, cont_value)]` は `e` の中の変数 `x` と変数 `k` に同時にそれぞれ値 `v` と値 `cont_value` を代入した式を返す。関数 `search_op` はハンドラ内のオペレーションを検索する関数で、例えば `{return x -> x; op1(y; k) -> k y}` を表すデータを `h` とすると `search_op "op2" h` は `None` を返し `search_op "op1" h` は `Some ("y", "k", App (Var "k", Var "y"))` を返す。

このインタプリタは、`handle` 節内の実行については普通の CPS になっており、継続である `k` は「直近のハンドラまでの継続」である。関数 `eval` の下から 2 行目で `with handle` 文を実行する際、再帰呼び出しの継続として `(fun x -> Return x)` を渡しており、これによって `handle` 節の実行に入るたびに渡す継続を初期化している。

`handle` 節内を実行した結果を表すのが `a` 型である。`handle` 節内の実行は、オペレーション呼び出しが行われない限りは通常の CPS インタプリタによって進むが、オペレーション呼び出しが行われた場合 (`eval` の下から 4 行目) は引数 `e` を実行後、結果を継続 `k` に渡すことなく `OpCall` を返している。これが `handle` 節の結果となり、`eval` の最下行で `apply_handler` に渡される。一方、`handle` 節内の実行が正常終了した場合は、初期継続 (`fun x -> Return x`) に結果が返り、それが `apply_handler` に渡される。

```

(* 値 *)
type v = Var of string      (* x *)
      | Fun of string * e   (* fun x -> e *)
      | Cont of (k -> k)    (* 継続 fun x => ... *)
(* ハンドラ *)
and h = {return : string * e;      (* {return x -> e;      *}
        ops : (string * string * string * e) list} (* op(x; k) -> e; ...} *)
(* 式 *)
and e = Val of v              (* v *)
      | App of e * e         (* e e *)
      | Op of string * e     (* op e *)
      | With of h * e        (* with h handle e *)
(* handle 内の継続 *)
and k = v -> a
(* handle 内の実行結果 *)
and a = Return of v          (* 値になった *)
      | OpCall of string * v * k (* オペレーションが呼び出された *)

```

図 4.2: 対象言語の定義

ここで、オペレーション呼び出しで返される `OpCall` の第 3 引数が `k` ではなく `fun v -> k v` のように η -expand されているのに注意しよう。このようにしているのは、`k` が「直近のハンドラまでの継続」を表しているのに対し、`OpCall` の第 3 引数はより広い継続を指すことがあり両者を区別したいためである。これについては、次節で非関数化を施す際に詳しく述べる。

`apply_handler` は、そのときの継続 `k`、処理すべきハンドラ `h`、そして `handle` 節内の実行結果 `a` を受け取ってハンドラの処理をする。関数 `apply_handler` の動作は `handle` 節の実行結果とハンドラの内容によって 3 種類ある。

1. `handle` 節が値 `v` になった場合：ハンドラの `return` 節 `return x -> e` を参照して、`e[v/x]` を実行
2. `handle` 節がオペレーション呼び出し `OpCall (name, v, k')` になった場合で、そのオペレーション `name` がハンドラ内で定義されていなかった場合：さらに外側の `with handle` 文に処理を移すため、`handle` 節内の限定継続 `k'` に、1 つ外側の `handle` までの限定継続を合成した継続 `fun v -> ...` を作り、それを `OpCall (name, v, (fun v -> ...))` と返す。この `OpCall` の第 3 引数は「直近のハンドラまでの継続」ではなく、より広い継続となっている。
3. `handle` 節がオペレーション呼び出し `OpCall (name, v, k')` になった場合で、そのオペレーション `name` がハンドラ内で定義されていた場合：そのハンドラの定義 `name (x; y)`

```

(* CPS インタプリタ *)
let rec eval (exp : e) (k : k) : a = match exp with
| Val (v) -> k v    (* 継続に値を渡す *)
| App (e1, e2) ->
  eval e2 (fun v2 -> (* FApp2 に変換される関数 *)
    eval e1 (fun v1 -> match v1 with (* FApp1 に変換される関数 *)
      | Fun (x, e) ->
        let reduct = subst e [(x, v2)] in (* e[v2/x] *)
        eval reduct k
      | Cont (cont_value) -> (cont_value k) v2
      (* 現在の継続と継続値が保持するメタ継続を合成して値を渡す *)
      | _ -> failwith "type error"))
| Op (name, e) ->
  eval e (fun v -> OpCall (name, v, fun v -> k v)) (* FOp に変換される関数 *)
| With (h, e) ->
  let a = eval e (fun v -> Return v) in (* FId に変換される関数、空の継続 *)
  apply_handler k h a (* handle 節内の実行結果をハンドラで処理 *)

(* handle 節内の実行結果をハンドラで処理する関数 *)
and apply_handler (k : k) (h : h) (a : a) : a = match a with
| Return v ->
  (* handle 節内が値 v を返したとき *)
  (match h with {return = (x, e)} -> (* handler {return x -> e; ...} として *)
    let reduct = subst e [(x, v)] in (* e[v/x] に簡約される *)
    eval reduct k) (* e[v/x] を実行 *)
| OpCall (name, v, m) ->
  (* オペレーション呼び出しがあったとき *)
  (match search_op name h with
  | None ->
    (* ハンドラで定義されていない場合、 *)
    OpCall (name, v, (fun v -> (* OpCall の継続の後に現在の継続を合成 *)
      let a' = m v in
      apply_handler k h a'))
  | Some (x, y, e) ->
    (* ハンドラで定義されている場合、 *)
    let cont_value =
      Cont (fun k'' -> fun v -> (* 適用時にその後の継続を受け取って合成 *)
        let a' = m v in
        apply_handler k'' h a') in
    let reduct = subst e [(x, v); (y, cont_value)] in
    eval reduct k)

(* 初期継続を渡して実行を始める *)
let interpreter (e : e) : a = eval e (fun v -> Return v) (* FId に変換される関数 *)

```

図 4.3: 継続渡し形式で書かれたインタプリタ

-> e を参照し、 $e[v/x, \text{cont_value}/y]$ を実行する。(cont_value については、以下の説明を参照。)

オペレーション呼び出しを処理する際に k に束縛する限定継続 cont_value は、「オペレーション呼び出し時の限定継続 k' 」に「現在のハンドラ h 」と「cont_value が呼び出された時の継続 k'' 」を合成したものである。

このようにして作られた限定継続が呼び出されるのは eval の App の Cont のケースである。cont_value は、この継続が呼び出された時点での限定継続が必要なので、それを cont_value k のように渡してから値 v_2 を渡している。

これまで、algebraic effects の意味論は small-step のもの [10, 11] 以外には CPS で書かれた big-step のもの [9] が提示されてきたが、この意味論はすでに部分式に名前が与えられている (A-正規形になっている) ことを仮定している上に、毎回、捕捉する継続を計算しているなど実装には必ずしも合ったものとは言えなかった。ここで示した CPS インタプリタは単純で、ハンドラの意味を的確に捉えており、algebraic effects の定義を与えるインタプリタ (definitional interpreter) と捉えて良いのではないかと考えている。

4.3 インタプリタの変換

本節では、4.2 節で定義したインタプリタ (図 4.3) に対して、正当性の保証された 2 種類のプログラム変換 (非関数化と CPS 変換) をかけることで、コンテキストを明示的に保持するインタプリタを得て、そこからステップ関数を作成する方法を示す。

4.3.1 非関数化

まず、4.2 節において示したインタプリタで引数として渡されている継続を非関数化する。

非関数化というのは、高階関数を 1 階のデータ構造で表現する方法である。高階関数は全てその自由変数を引数に持つような 1 階のデータ構造となり、高階関数を呼び出していた部分は apply 関数の呼び出しとなる。この apply 関数は、高階関数が呼び出されていたら行ったであろう処理を行うように別途、定義されるものである。この変換は機械的に行うことができる。

具体的に図 4.3 のプログラムの継続 k 型の λ 式を非関数化するには次のようにする。結果は図 4.4 と図 4.5 のようになる。

1. 継続を表す λ 式をコンストラクタに置き換える。その際、 λ 式内の自由変数はコンストラクタの引数にする。その結果、得られるデータ構造は図 4.4 のようになる。図 4.3 の中には、コメントとしてどの関数がどのコンストラクタに置き換わったのかが書かれている。

```

(* handle 内の継続 *)
type k = FId                (* [.] *)
      | FApp2 of e * k      (* [e [.]] *)
      | FApp1 of v * k      (* [[.] v] *)
      | FOp of string * k   (* [op [.]] *)

```

図 4.4: 非関数化後の継続の型

2. 関数を表すコンストラクタと引数を受け取って中身を実行するような `apply` 関数を定義する。これは、図 4.5 では `apply_in` と呼ばれている。
3. λ 式を呼び出す部分を、`apply` 関数にコンストラクタと引数を渡すように変更する。

非関数化した後の継続の型を見ると、ラムダ計算の通常の評価文脈に加えてオペレーション呼び出しの引数を実行するフレーム `FOp` が加わっていることがわかる。これが、ハンドラ内の実行のコンテキスト情報である。

ここで、`OpCall` の第3引数は非関数化されていないことに注意しよう。この部分はハンドラ内の継続とは限らないので、ここでは非関数化せずにもとのままとしている。ここを非関数化することも可能ではあるが、そうすると最終的に得られるコンテキスト情報がきれいなリストのリストの形にはならなくなってしまう。

ハンドラ内の評価文脈を表すデータ構造は非関数化により導くことができたが、図 4.5 のインタプリタはオペレーション呼び出しなどの実装で継続を非末尾の位置で使っており純粋な CPS 形式にはなっていないため、全体のコンテキストは得られていない。そのため、このコンテキストを使ってステッパを構成してもプログラム全体を再構成することはできない。プログラム全体のコンテキストを得るためには、このインタプリタに対してもう一度 CPS 変換と非関数化を施し、純粋な CPS 形式にする必要がある。

4.3.2 CPS 変換

図 4.5 では、末尾再帰でない再帰呼び出しの際に継続が初期化されてしまうせいでコンテキスト全体に対応する情報が継続に含まれていなかった。ここでは、全てのコンテキスト情報を明示化するため、さらに CPS 変換を施す。この変換によって現れる継続は `a -> a` 型である。この型 `a -> a` の名前を `k2` とする。変換したプログラムは図 4.6 に示す。

このプログラムは、図 4.5 のプログラムを機械的に CPS 変換すれば得られるもので、`OpCall` の第3引数も CPS 変換される点にさえ注意すれば、特に説明を必要とする箇所はない。プログラム中には、次節で非関数化する部分にその旨、コメントが付してある。この変換により、すべての (serious な) 関数呼び出しが末尾呼び出しとなり、コンテキスト情報はふたつの継続ですべて表現される。

```

(* CPS インタプリタを非関数化した関数 *)
let rec eval (exp : e) (k : k) : a = match exp with
| Val (v) -> apply_in k v  (* 継続適用関数に継続と値を渡す *)
| App (e1, e2) -> eval e2 (FApp2 (e1, k))
| Op (name, e) -> eval e (FOp (name, k))
| With (h, e) -> let a = eval e FId in  (* 空の継続を渡す *)
  apply_handler k h a  (* handle 節内の実行結果をハンドラで処理 *)

(* handle 節内の継続を適用する関数 *)
and apply_in (k : k) (v : v) : a = match k with
| FId -> Return v  (* 空の継続、そのまま値を返す *)
| FApp2 (e1, k) -> let v2 = v in eval e1 (FApp1 (v2, k))
| FApp1 (v2, k) -> let v1 = v in
  (match v1 with
  | Fun (x, e) ->
    let reduct = subst e [(x, v2)] in
    eval reduct k
  | Cont (cont_value) -> (cont_value k) v2
  | _ -> failwith "type error")
| FOp (name, k) ->
  OpCall (name, v, (fun v -> apply_in k v))  (* Op 呼び出しの情報を返す *)

(* handle 節内の実行結果をハンドラで処理する関数 *)
and apply_handler (k : k) (h : h) (a : a) : a = ...  (* 非関数化前と同じ *)

(* 初期継続を渡して実行を始める *)
let interpreter (e : e) : a = eval e FId  (* 空の継続を渡す *)

```

図 4.5: CPS インタプリタを非関数化したプログラム

```

(* CPS インタプリタを非関数化して CPS 変換した関数 *)
let rec eval (exp : e) (k : k) (k2 : k2) : a = match exp with
| Val (v) -> apply_in k v k2
| App (e1, e2) -> eval e2 (FApp2 (e1, k)) k2
| Op (name, e) -> eval e (FOp (name, k)) k2
| With (h, e) ->
    eval e FId (fun a -> apply_handler k h a k2) (* GHandle に変換される *)

(* handle 節内の継続を適用する関数 *)
and apply_in (k : k) (v : v) (k2 : k2) : a = match k with
| FId -> k2 (Return v) (* 継続適用 *)
| FApp2 (e1, k) -> let v2 = v in eval e1 (FApp1 (v2, k)) k2
| FApp1 (v2, k) -> let v1 = v in
    (match v1 with
    | Fun (x, e) ->
        let reduct = subst e [(x, v2)] in
        eval reduct k k2
    | Cont (cont_value) ->
        (cont_value k) v2 k2
    | _ -> failwith "type error")
| FOp (name, k) ->
    k2 (OpCall (name, v, (fun v -> fun k2' -> apply_in k v k2')))) (* 継続適用 *)

(* handle 節内の実行結果をハンドラで処理する関数 *)
and apply_handler (k : k) (h : h) (a : a) (k2 : k2) : a = match a with
| Return v ->
    (match h with {return = (x, e)} ->
    let reduct = subst e [(x, v)] in
    eval reduct k k2)
| OpCall (name, v, m) ->
    (match search_op name h with
    | None ->
        k2 (OpCall (name, v, (fun v -> fun k2' -> (* 継続適用 *)
            m v (fun a' -> apply_handler k h a' k2'))))) (* GHandle に変換 *)
    | Some (x, y, e) ->
        let cont_value =
            Cont (fun k'' -> fun v -> fun k2 ->
                m v (fun a' -> apply_handler k'' h a' k2)) in (* GHandle に変換 *)
        let reduct = subst e [(x, v); (y, cont_value)] in
        eval reduct k k2)

(* 初期継続を渡して実行を始める *)
let interpreter (e : e) : a = eval e FId (fun a -> a) (* GId に変換される *)

```

図 4.6: CPS インタプリタを非関数化して CPS 変換したプログラム

```
(* 全体のメタ継続 *)
type k2 = GId
        | GHandle of h * k * k2
```

図 4.7: 2 回目の非関数化後の継続の型

4.3.3 非関数化

CPS 変換ですべてのコンテキスト情報がふたつの継続に集約された。ここでは、CPS 変換したことにより新たに現れた $a \rightarrow a$ 型の関数を非関数化してデータ構造に変換する。非関数化によって型 $k2$ の定義は図 4.7 に、インタプリタは図 4.8 に変換される。

この非関数化によって、引数 k と引数 $k2$ からコンテキスト全体の情報が得られるようになった。ここで、得られたコンテキストの情報を整理しておこう。 k はハンドラ内のコンテキストを示している。FId 以外はいずれの構成子も k を引数にとっているので、これは FId を空リストととらえれば評価文脈のリストと考えることができる。 $k2$ も同様に h と k が連なったリストと考えることができる。全体として「ハンドラに囲まれた評価文脈のリスト」のリストになっており、直感に合ったハンドラによって区切られたコンテキストが得られていることがわかる。

得られたコンテキストはごく自然なものだが、ハンドラの入る位置などは必ずしも自明ではない。プログラム変換を使うことで、algebraic effects の入った体系に沿ったコンテキストのデータ型が機械的に得られたことには一定の価値があると考えられる。

4.3.4 出力

4.3.3 節までの変換によって、コンテキストの情報を引数に保持するインタプリタ関数を得ることができた。この情報を用いて簡約前後のプログラムを出力するようにするとステップ関数が得られる。具体的には、簡約が起こる部分でプログラム全体を再構成し表示するようにする。図 4.9 が表示を行う関数 `memo` を足した後の関数 `apply_in` と `apply_handler` である。(他の関数は簡約している部分が無いので図 4.8 と同じになる。)

ここで関数 `memo : e -> e -> (k * k2) -> unit` は、簡約基とその簡約後の式と簡約時のコンテキストを受け取って、簡約前のプログラムと簡約後のプログラムをそれぞれ再構成して出力する。

図 4.9 を見ると `apply_in` では普通の関数呼び出しと継続呼び出しが `memo` されている。また、`apply_handler` ではハンドラが正常終了した場合とオペレーション呼び出しが起きた場合にそれぞれ `memo` 関数が挿入されている。また、オペレーション呼び出しが処理されず外側の `with handle` 文に制御を移す際には、図 4.10 に示される関数を使ってコンテキストの結合を行なっている。

```

(* CPS インタプリタを非関数化して CPS 変換して非関数化した関数 *)
let rec eval (exp : e) (k : k) (k2 : k2) : a = match exp with
| Val (v) -> apply_in k v k2
| App (e1, e2) -> eval e2 (FApp2 (e1, k)) k2
| Op (name, e) -> eval e (FOp (name, k)) k2
| With (h, e) -> eval e FId (GHandle (h, k, k2))

(* handle 節内の継続を適用する関数 *)
and apply_in (k : k) (v : v) (k2 : k2) : a = match k with
| FId -> apply_out k2 (Return v)
| FApp2 (e1, k) -> let v2 = v in eval e1 (FApp1 (v2, k)) k2
| FApp1 (v2, k) -> let v1 = v in (match v1 with
| Fun (x, e) ->
    let reduct = subst e [(x, v2)] in
    eval reduct k k2
| Cont (cont_value) ->
    (cont_value k) v2 k2
| _ -> failwith "type error")
| FOp (name, k) ->
    apply_out k2 (OpCall1 (name, v, (fun v -> fun k2' -> apply_in k v k2'))))

(* 全体の継続を適用する関数 *)
and apply_out (k2 : k2) (a : a) : a = match k2 with
| GId -> a
| GHandle (h, k, k2) -> apply_handler k h a k2

(* handle 節内の実行結果をハンドラで処理する関数 *)
and apply_handler (k : k) (h : h) (a : a) (k2 : k2) : a = match a with
| Return v -> (match h with {return = (x, e)} ->
    let reduct = subst e [(x, v)] in eval reduct k k2)
| OpCall1 (name, v, va) ->
    (match search_op name h with
    | None ->
        apply_out k2 (OpCall1 (name, v,
            (fun v -> fun k2' -> m v (GHandle (h, k, k2'))))))
    | Some (x, y, e) ->
        let cont_value =
            Cont (fun k'' -> fun v -> fun k2 -> m v (GHandle (h, k'', k2))) in
        let reduct = subst e [(x, v); (y, cont_value)] in
        eval reduct k k2)

(* 初期継続を渡して実行を始める *)
let interpreter (e : e) : a = eval e FId GId

```

図 4.8: CPS インタプリタを非関数化して CPS 変換して非関数化したプログラム

```

(* handle 節内の継続を適用する関数 *)
and apply_in (k : k) (v : v) (k2 : k2) : a = match k with
| FId -> apply_out k2 (Return v)
| FApp2 (e1, k) -> let v2 = v in eval e1 (FApp1 (v2, k)) k2
| FApp1 (v2, k) -> let v1 = v in (match v1 with
| Fun (x, e) ->
    let redex = App (Val v1, Val v2) in (* (fun x -> e) v2 *)
    let reduct = subst e [(x, v2)] in (* e[v2/x] *)
    memo redex reduct (k, k2); eval reduct k k2
| Cont (x, (k', k2'), cont_value) ->
    let redex = App (Val v1, Val v2) in (* (fun x => k2'[k'[x]]) v2 *)
    let reduct = plug_all (Val v2) (k', k2') in (* k2'[k'[v2]] *)
    memo redex reduct (k, k2); (cont_value k) v2 k2
| _ -> failwith "type error")
| FOp (name, k) ->
    apply_out k2 (OpCall (name, v, (k, GId),
    (fun v -> fun k2' -> apply_in k v k2'))))

(* handle 節内の実行結果をハンドラで処理する関数 *)
and apply_handler (k : k) (h : h) (a : a) (k2 : k2) : a = match a with
| Return v ->
    (match h with {return = (x, e)} ->
    let redex = With (h, Val v) in (* with {return x -> e} handle v *)
    let reduct = subst e [(x, v)] in (* e[v/x] *)
    memo redex reduct (k, k2); eval reduct k k2)
| OpCall (name, v, (k', k2'), m) -> (match search_op name h with
| None ->
    apply_out k2 (OpCall (name, v, (k', compose_k2 k2' h (k, GId)),
    (fun v -> fun k2' -> m v (GHandle (h, k, k2')))))
| Some (x, y, e) ->
    (* with {name(x; y) -> e} handle k2'[k'[name v]] *)
    let redex = With (h, plug_all (Op (name, Val v)) (k', k2')) in
    let cont_value =
        Cont (gen_var_name (), (k', compose_k2 k2' h (FId, GId)),
        (fun k'' -> fun v -> fun k2 -> m v (GHandle (h, k'', k2)))) in
    (* e[v/x, (fun n => with {name(x; y) -> e} handle k2'[k'[n]]) /y *)
    let reduct = subst e [(x, v); (y, cont_value)] in
    memo redex reduct (k, k2);
    eval reduct k k2)

```

図 4.9: 変換の後、出力関数を足して得られるステップ

```

(* コンテキスト k2_in の外側にフレーム GHandle (h, k_out, k2_out) を付加する *)
let rec compose_k2 (k2_in : k2) (h : h) ((k_out, k2_out) : k * k2) : k2 =
  match k2_in with
  | GId -> GHandle (h, k_out, k2_out)
  | GHandle (h', k', k2') ->
    GHandle (h', k', compose_k2 k2' h (k_out, k2_out))

```

図 4.10: 継続を外側に拡張する関数

```

(* 値 *)
type v = ...
  | Cont of string * (c * c2) * ((c * k) -> k) (* 継続 *)
(* handle 内の実行結果 *)
and a = Return of v (* 値になった *)
  | OpCall of string * v * (c * c2) * k (* オペレーションが呼び出された *)
(* handle 内のメタ継続 *)
and k = v -> c2 -> a
(* handle 内のコンテキスト *)
and c = FId (* [.] *)
  | FApp2 of e * c (* [e [.]] *)
  | FApp1 of v * c (* [[.] v] *)
  | FOp of string * c (* [op [.]] *)
(* 全体のコンテキスト *)
and c2 = GId
  | GHandle of h * c * c2

```

図 4.11: 継続の情報を保持するための言語やコンテキストの定義

4.3.5 CPS インタプリタに基づいたステッパ

前節で algebraic effects を持つ言語に対するステッパ関数を作ることができた。しかし、前節で作ったステッパ関数ではコンテキストの情報が非関数化されていた。また、CPS 変換されているためふたつの継続を扱っており、もともとの CPS インタプリタとは形がかなり異なったものとなっている。しかし、一度、前節までで必要なコンテキストの情報がどのようなものかが判明すると、それを直接、もとの CPS インタプリタに加えてステッパ関数を作ることができる。

もとの CPS インタプリタの型定義に必要なコンテキストの情報を加えた定義が図 4.11 になる。ここで、 c と $c2$ がそれぞれハンドラ内、全体のコンテキストの情報で、前節までの非関数化によって得られたものである。一方、 k はもともとからある高階の継続の型である。継続 k は、簡約ごとにプログラム全体を表示するので、必要なコンテキストの情報を新たに引数に取るようになっている。よって k の定義は $v \rightarrow c2 \rightarrow a$ となる。

このデータ定義を使って、もとの CPS インタプリタをステッパ関数に変換したのが図 4.12 である。このインタプリタは、もとの CPS インタプリタにコンテキストの情報として引数 c と $c2$ を加え、簡約ごとにプログラムを再構成し、ステップ表示するようにしたものである。一度、必


```

(* CPS ステップ *)
let rec eval (exp : e) ((c, k) : c * k) (c2 : c2) : a = match exp with
| Val (v) -> k v c2
| App (e1, e2) -> eval e2 (FApp2 (e1, c), (fun v2 c2 ->
  eval e1 (FApp1 (v2, c), (fun v1 c2 -> match v1 with
    | Fun (x, e) ->
      let redex = App (Val v1, Val v2) in (* (fun x -> e) v2 *)
      let reduct = subst e [(x, v2)] in (* e[v2/x] *)
      memo redex reduct (c, c2); eval reduct (c, k) c2
    | Cont (x, (c', c2'), cont_value) ->
      let redex = App (Val v1, Val v2) in (* (fun x => c2[c[x]]) v2 *)
      let reduct = plug_all (Val v2) (c', c2') in (* c2[c[v2]] *)
      memo redex reduct (c, c2); (cont_value (c, k)) v2 c2
    | _ -> failwith "type error"))) c2)) c2
| Op (name, e) -> eval e (FOp (name, c), (fun v c2 ->
  OpCall (name, v, (c, GId), (fun v c2' -> k v c2')))) c2
| With (h, e) ->
  let a = eval e (FId, (fun v c2 -> Return v)) (GHandle (h, c, c2)) in
  apply_handler (c, k) h a c2

(* handle 節内の実行結果をハンドラで処理する関数 *)
and apply_handler ((c, k) : c * k) (h : h) (a : a) (c2 : c2) : a = match a with
| Return v -> (match h with {return = (x, e)} ->
  let redex = With (h, Val v) in (* with {return x -> e} handle v *)
  let reduct = subst e [(x, v)] in (* e[v/x] *)
  memo redex reduct (c, c2); eval reduct (c, k) c2)
| OpCall (name, v, (c', c2'), k') ->
  (match search_op name h with
  | None -> OpCall (name, v, (c', compose_c2 c2' h (c, GId)),
    (fun v' c2'' -> let a' = k' v' (GHandle (h, c, c2'')) in
    apply_handler (c, k) h a' c2''))
  | Some (x, y, e) ->
    (* with {name(x; y) -> e} handle c2'[c'[name v]] *)
    let redex = With (h, plug_all (Op (name, Val v)) (c', c2')) in
    let cont_value = Cont (gen_var_name (),
      (c', compose_c2 c2' h (FId, GId)), (fun (c'', k'') v' c2'' ->
        let a' = k' v' (GHandle (h, c'', c2'')) in
        apply_handler (c'', k'') h a' c2)) in
    (* e[v/x, (fun n => with {name(x; y) -> e} handle c2'[c'[y]])/y *)
    let reduct = subst e [(x, v); (y, cont_value)] in
    memo redex reduct (c, c2); eval reduct (c, k) c2)

let stepper (e : e) : a = eval e (FId, (fun v c2 -> Return v)) GId

```

図 4.12: 変換の結果得られた、CPS インタプリタを基にしたステップ関数

要なコンテキストの情報が特定されると、algebraic effects のように非自明な言語構文が入っていても、直接、ステップ関数を作ることができるようになる。

4.4 他の言語への対応

4.2 節で示した algebraic effects を含む言語の CPS インタプリタをステップ関数にするには、非関数化、CPS 変換、非関数化が必要だったが、他のいくつかの言語についても同様にインタプリタを変換することでステップ関数を導出することを試みた。それぞれの言語のステップ関数導出について説明する。

4.4.1 型無し λ 計算

型無し λ 計算のダイレクトスタイルインタプリタは、CPS 変換して非関数化したら全てのコンテキストを引数に保持するインタプリタになり、出力関数を入れるのみでステップ関数を作ることができた。これは、継続を区切って一部を捨てたり束縛したりするという操作が無いためである。

4.4.2 try-with

try-with は、algebraic effects が限定継続を変数に束縛するのと違って、例外が起こされたときに限定継続を捨てるという機能である。よって継続を表す値は現れないので、インタプリタを CPS で書く必要は無い。ダイレクトスタイルでインタプリタを書いた場合、最初に CPS 変換することで、algebraic effects の CPS インタプリタと同様の変換によってステップが導出できた。最初から CPS インタプリタを書いていたれば 4.3 節と同様の手順になる。

4.4.3 shift/reset

shift/reset は algebraic effects と同様に限定継続を変数に束縛して利用することができる機能である。4.3 節で行ったのと全く同様に、CPS インタプリタを非関数化、CPS 変換、非関数化したらコンテキストが表れ、ステップ関数が得られた。

4.4.4 Multicore OCaml

Multicore OCaml は、OCaml の構文に algebraic effects を追加した構文を持つ。我々は 4.3 節で得られたステップ関数をもとにして、Multicore OCaml の algebraic effects を含む一部の構文を対象にしたステップの実装を目指している。Multicore OCaml の「エフェクト」は 4.2 節で定義した言語の algebraic effects のオペレーションとほとんど同じものであり、継続が one-shot で

あることを除いて簡約のされかたは 4.2 節で定めた言語のインタプリタと同様なので、インタプリタ関数を用意できれば変換によってステップ関数が導出できると考えられる。

4.5 この章のまとめ

ステップ関数を実装するためには、コンテキストの情報を保持しながら部分式を再帰的に実行するインタプリタを作ればよい。3 章で行った方法では言語ごとにコンテキストを表すデータ型を考えた上でインタプリタに実行の流れに従った新しい引数を付け足す作業が必要だったが、本研究では通常のインタプリタを CPS 変換および非関数化するという機械的な操作でコンテキストの型およびコンテキストの情報を保持するインタプリタ関数を導出した。

その方法で、継続を明示的に扱える algebraic effects を含む言語に対するステップ関数を実装し、他の例外処理機能である `try-with` や `shift/reset` を含む言語についても同様の変換ができることを確認した。

第5章 incremental なステップの実装

5.1 ステップの動作

本章で説明する incremental なステップは、表面上は 3 章のステップと同じ動作をするが、内部での処理方法および速さが大きく異なる。本節では、既存のステップと本章の incremental なステップのそれぞれの動作について説明する。

いずれのステップも、実行可能な 1 つのプログラムを対象としている。

5.1.1 DrRacket のステップ

DrRacket のステップ [2] は、ユーザが入力したプログラムを全ステップの情報を生成するプログラムへ変換し、それを実行してステップの情報を蓄えながら、ユーザの操作に従って 1 つのステップを表示する。実行開始に少し遅れて、表示のための処理を並列して行うことになる。

5.1.2 incremental でない OCaml ステップ

3 章で実装した OCaml ステップ (以下、incremental でない OCaml ステップ) では、では、インタプリタにステップ出力機能を足したものに入力プログラムを渡し、全ステップの文字列を生成する。プログラムを全て実行するかステップ数の上限に達すると実行を終了し、最初のステップを表示し、ユーザの操作に従って表示するステップを変える。

インタプリタは新しく我々が作った OCaml の関数であり、通常のインタプリタよりも実行速度が遅い。そこにさらに出力機能を足したインタプリタによるプログラム実行が終わるまで表示が始まらないため、実行に時間がかかるプログラムのステップ実行をするには長い時間待つ必要がある。

5.1.3 提案するステップ

本章で提案するステップでは、インタプリタにステップ出力機能を足したものに入力プログラムを渡し、1 ステップの簡約を計算したらただちにそのステップを出力する。出力後は続きの実行は行わずにプロセスを終了する。ユーザから次のステップを表示するなどの命令がされ次第、前回の出力の一部を新しい入力として受け取って次の 1 ステップ実行をする。

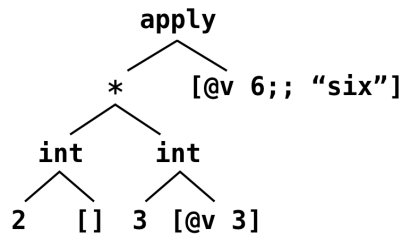


図 5.1: attribute を含む構文木の例

例えば、 $2 * 3 + 5 * 7$ というプログラムを入力されると、incremental でないステップ関数は $2 * 3 + 5 * 7 \rightsquigarrow 2 * 3 + 35 \rightsquigarrow 6 + 35 \rightsquigarrow 41$ という 3 ステップを出力するのに対して、incremental なステップ関数は $2 * 3 + 5 * 7 \rightsquigarrow 2 * 3 + 35$ の 1 ステップを出力する。次のステップを表示する命令がされたときに、外部のプログラムがそこから $2 * 3 + 35$ という部分を抜き出して再度ステップパに入力することでその次のステップを得る。

incremental なステップパには、メモリに膨大なステップの情報を保存する必要がない、ユーザが見ないステップは計算されないという特徴がある。

5.2 OCaml の attribute

OCaml 4.02 以降では、OCaml の構文木中に attribute という情報を付加することができる。attribute はそれぞれ名前と、OCaml のプログラムやシグネチャなどの引数を持つ。incremental でない OCaml ステップパ (3.4 節) および本章で提案するステップパでは attribute を利用している。本節では、本章のステップパの実装で利用する種類の attribute を紹介する。

5.2.1 式の attribute

OCaml のプログラムでは、任意の部分式に attribute をつけることができる。式 e に OCaml プログラム P を引数に持つ **name** という attribute を付けたものは $e[@name\ P]$ と書く。例えば $(2 * 3[@v\ 3])[@v\ 6;;\ "six"]$ という式は大まかには図 5.1 のように構文解析され、ステップ関数のように構文木を扱うプログラムの中で attribute の内容を利用することができる。通常の OCaml コンパイラは attribute を無視するので、シンタックスエラー等を起こしてしまう場合を除いて、attribute が付いた式と付いていない式で意味は変わらない。

incremental でない OCaml ステップパでは、各ステップで簡約が起こっている部分の式をハイライトして示すために attribute を利用している。例えば $(2 * 3) + (5 * 7)$ というプログラムに対する incremental でないステップ関数の出力は図 5.2 の左の文字列である。インタフェースを受け持つプログラムはこの文字列を受け取り、図 5.2 の右側のように、attribute が付いた式をハイライトし、さらに attribute を表す文字列を削除した上で表示している。

(* Step 0 *)	(* Step 0 *)
(2 * 3) + (5 * 7)[@stepper.redex]	(2 * 3) + (5 * 7)
(* Step 1 *)	(* Step 1 *)
(2 * 3) + 35[@stepper.reduct]	(2 * 3) + 35
(* Step 1 *)	(* Step 1 *)
(2 * 3)[@stepper.redex] + 35	(2 * 3) + 35
(* Step 2 *)	(* Step 2 *)
6[@stepper.reduct] + 35	6 + 35
(* Step 2 *)	(* Step 2 *)
(6 + 35)[@stepper.redex]	(6 + 35)
(* Step 3 *)	(* Step 3 *)
41[@stepper.reduct]	41

図 5.2: ハイライトのための attribute の利用

本章の incremental なステップでも、同様の方法で簡約部分のハイライトを行う。本論文では、ステップ関数の出力文字列中のハイライトのための attribute を省略したり、緑色および紫色のハイライトで表すことがある。

5.2.2 プログラムの attribute

attribute はプログラム自体にも付けることができる。

OCaml のプログラムは structure item の列であり、structure item には式、変数定義 (`let` 変数名 = 式)、型定義、モジュール定義、attribute などの種類がある。structure item の間には `;;` を書くことで明示的に structure item の境を示すことができる。

attribute の structure item はプログラム中に何度でも書くことができ、名前の重複などに関する制約も無い。また式に付ける attribute と同じく、標準のコンパイラなどはこれを無視するのでプログラムの意味に影響を与えない。

`name` という名前で OCaml プログラム `P` を含む attribute の structure item を `[@@@name P]` と書く。例えば `let a = 1 [@@@name1 1;; 2 + 3] let b = 2 [@@@name2]` というプログラムは、(1)a の定義、(2)1 と `2 + 3` を引数に持つ `name1` という attribute、(3)b の定義、(4) 引数なしの `name2` という attribute、の4つの structure item のリストとして処理され、それぞれの attribute の内容はステップが参照することができる。

これを用いると、プログラムに自由に情報を付加することができる。incremental なステップの実装においては、副作用によって変化する「状態」や現在のステップ番号を記録するために使用する。

5.3 生じる問題と解決方法

incremental でない OCaml ステップと全く同じようにステップ出力を行うと、incremental なステップでは様々な問題が起きた。本節ではその問題と、回避するために行った出力内容の変更について説明する。

5.3.1 情報の消失

問題点

5.1.1 節および5.1.2 節で紹介した既存のステップは、ステップ実行をした結果を蓄えておき、ユーザが表示ステップを切り替える際にその中から表示するステップの情報を検索して表示するので、ステップ番号を指定すれば任意のステップをすぐに表示することができる。それに対して本研究で提案するステップは、表示を切り替える命令が入力されるたびに1ステップ先や1ステップ前を計算することになる。その際のステップ関数の入力は前回出力したプログラムの一部である。すると、1ステップ前を計算する時に問題が起きる。

例えば、 $2 * 3 + 5 * 7$ というプログラムをステップ実行するとき、最初に表示される1ステップ目は $2 * 3 + 5 * 7 \rightsquigarrow 2 * 3 + 35$ であり、次ステップ表示命令が入力されると今度は $2 * 3 + 35$ を入力として $2 * 3 + 35 \rightsquigarrow 6 + 35$ を出力する。ここで前ステップ表示命令が入力された時に、 $2 * 3 + 35$ や $6 + 35$ という情報から $2 * 3 + 5 * 7 \rightsquigarrow 2 * 3 + 35$ を導き出すことは不可能である。これは計算が不可逆的であるという性質によるものである。すなわち式 $5 * 7$ と 35 のどちらからその値が 35 だという情報は得られるが、式 35 からそれがかつて $5 * 7$ だったという情報は得られないのである。

解決方法

incremental なステップでは、簡約後の式に簡約前の式の情報を、簡約された式を表す attribute `[@stepper.reduct]` の引数として付加することで、簡約によって失われた情報を復元可能にする。「式2が簡約された結果の式1」を式1`[@stepper.reduct 式2]`と表して、 $2 * 3 + 5 * 7 \rightsquigarrow 2 * 3 + 35$ の代わりに $2 * 3 + 5 * 7 \rightsquigarrow 2 * 3 + 35[@stepper.reduct 5 * 7]$ 、また $2 * 3 + 35 \rightsquigarrow 6 + 35$ の代わりに $2 * 3 + 35[@stepper.reduct 5 * 7] \rightsquigarrow 6[@stepper.reduct 2 * 3] + 35[@stepper.reduct 5 * 7]$ を出力すると、どのステップの文字列からでもオリジナルのプログラムまで情報を復元することができる。

さらに、直前のステップで簡約された式が明示的に分かるように、簡約された時のステップ番号も attribute に含める。例えば $6[@stepper.reduct 2 * 3] + 35[@stepper.reduct 5 * 7]$ に $2 * 3$ や $5 * 7$ のそれぞれの簡約が行われた当時のステップ番号を追加して、 $6[@stepper.reduct$

$(2, 2 * 3)] + 35[@stepper.reduct (1, 5 * 7)]$ と出力する。こうすると、ここから前のステップを求めるときに、最後に簡約されたのは 6 だとステップ番号から分かり、6 をその attribute に記録された簡約前の式 $2 * 3$ に置き換えることで前のステップ $2 * 3 + 35[@stepper.reduct (1, 5 * 7)] \rightsquigarrow 6[@stepper.reduct (2, 2 * 3)] + 35[@stepper.reduct (1, 5 * 7)]$ が得られる。

5.3.2 表示の崩れ

問題点

OCaml プログラムは、ライブラリで用意された関数を使うと文字列として出力することができ、本研究のステップではその関数を利用する。その関数は、適当に改行やインデントを入れてプログラムを出力する。しかし、図 5.2 のように attribute が付いたプログラムを出力させて後から外部で attribute を消すと、プログラムの体裁が崩れてしまう可能性がある。

特に、5.3.1 節で示したように、簡約後の式に簡約前の式の情報を含む attribute を付けるようにしてステップ実行を進めると、attribute 付きの式の出力が複数行にわたってしまうことがある。すると、例えば

```
((2 * 3)
  [@reduct 長い式 ]) +
(5 * 7)
```

といった式の途中に改行が入った状態になり、外部のインタフェース用プログラムがこの文字列を受け取って

```
(2 * 3) + (5 * 7)
```

と改行やスペースを調整して表示するのは難しい。

解決方法

我々は、実行に必要な情報がすべて含まれた式「処理用の式」と、表示するための整った式「表示用の式」をそれぞれ出力することでこれを解決した。

具体的には、 $2 * 3 + 5 * 7$ の最後のステップの出力が以下のようになるようにした。

```
(* Step 2 *)
[@@@stepper.process
  6[@stepper.reduct (2, 2 * 3) ] + 35[@stepper.reduct (1, 5 * 7) ] ]
(6 + 35)[@x ]
```



```
(* Step 3 *)
[[@stepper.process 41[@stepper.reduct (3,
  6[@stepper.reduct (2, 2 * 3) ] + 35[@stepper.reduct (1, 5 * 7) ] )] ]
41[@t ]
```

すなわち、

1. 処理用の簡約前の式（`[@stepper.reduct]` を含む）を attribute に入れたもの
2. 表示用の簡約前の式（`[@x]` を含む）
3. 処理用の簡約後の式（`[@stepper.reduct]` を含む）を attribute に入れたもの
4. 表示用の簡約後の式（`[@t]` を含む）

の 4 つのプログラムを出力する。

「処理用の式」は前後のステップを計算するための情報を持つ式であり、そこまでの全ての簡約の情報を attribute に持つ。ユーザが見る画面には表示させないようにインタフェース側で処理をする。「表示用の式」はユーザに見せるための式であり、ハイライトをする式にのみ短い attribute が付いている。

余計な改行の原因である「簡約されている式をハイライトするための attribute」は、インタフェース側のプログラムが表示に利用するので完全に省略することはできない。そこで、上の例のように最も文字数が少ない attribute `[@x]` や `[@t]` を簡約される式に付加することでハイライトする部分を示す。`x` と `t` はそれぞれ簡約基と簡約されたものを表す “redex” と “reduct” の略である。名前が 1 文字で他の情報を含まない attribute は文字数が少なくほとんどの場合改行を引き起こさないため、単純に attribute を文字列から削除してもユーザから見て不自然にプログラムの体裁が崩れることは少ないと考えられる。

次や前のステップを実行するには、インタフェース側のプログラムが attribute である structure item `[[@stepper.process ...]` の内容をステップに渡すようにする。すると、ステップには簡約の情報が全て含まれたプログラムが入力され、前のステップにも戻ることができる。前のステップの実行には処理用の簡約前の式、次のステップの実行には処理用の簡約後の式をステップに渡す。

5.4 入計算に対する実装

本節では、incremental でない OCaml ステップ関数をもとに incremental な OCaml ステップ関数を実装する。対象言語は型無しの入式で、さらに実際の OCaml を模して任意の部分式に複

```
(* 式の種類の定義 *)
type expression_desc = Var of string          (* x *)
                      | Fun of string * expression (* λ x. e *)
                      | App of expression * expression (* e1 e2 *)

and expression = {desc : expression_desc; (* 式の内容 *)
                  attr : attribute list} (* attribute [@name ... ] *)

and attribute = (string * payload)          (* 名前と内容のペア *)
and payload = (int * expression) option    (* ステップ番号と簡約前の式 *)
```

図 5.3: 対象言語の定義

数の attribute を付けられるものとする。各 attribute の第一引数は attribute の名前とし、第二引数には 5.3.1 節で定めた簡約の情報を簡単に表すために `Some` (整数, 式) または情報を持たないことを示す `None` のどちらか (payload 型) をとる¹。これらの型を図 5.3 のように定義する。式は `expression` 型であり、式本体の内容を表す `desc` と任意の個数の attribute を表す `attr` の 2 つの要素を持つ。

5.4.1 incremental でないステップ関数

incremental でないステップ関数は big-step インタプリタ関数にステップ出力のための作用を追加することで構築されている。incremental でないステップ関数の実装は図 5.4 のようになる。これは、3 章で示したコンテキストをリストとして定義した場合の try-with と型無し λ 計算のステップ (図 3.5) の、λ 計算を実行する部分と同じ動作をするプログラムである。関数 `eval` は OCaml の call-by-value かつ right-to-left の評価戦略に従った代入ベースの λ 計算のインタプリタにステップ実行のための作用を追加したステップ関数である。背景に灰色が付いた部分がステップ実行のための作用であり、白い部分のみを読むと単なるインタプリタとして見ることができる。ただし関数 `subst` は代入の関数であり、`subst f x arg_value` は式 `f` の中の変数 `x` を式 `arg_value` に置換した式を返す。このステップ関数の出力は、例えば入力プログラム `2 * 3 + 5 * 7` に対して、図 5.2 の左側の文字列 (にステップ番号の表示を足したもの) である。

ステップ関数は実行可能なプログラムのみを受け付けるため、ステップ関数に渡される式の中に自由変数および型エラーは存在しない。さらにこのステップ関数の基となる代入ベースのインタプリタでは、関数の内部の式は必ず関数適用の簡約 (すなわち実引数の代入) の後に実行するので、常に変数はその実行の前に束縛を解決されており、変数がインタプリタ関数の引数として実行されることはない。よって、関数 `eval` 中の `failwith` の呼び出しは起こり得ない。

関数 `eval` の下から 3 行目の関数 `memo` は、簡約前のプログラムを出力し、`counter` の値を 1 増やし、簡約後のプログラムを出力する関数である。その実装は図 5.5 に示す。ただし、関数

¹[`@x`] は ("`x`", `None`)、[`@stepper.reduct (1, 5 * 7)`] は ("`stepper.reduct`", `Some (1, 5 * 7)`) である。

```

(* コンテキストのフレームの定義 *)
type frame = AppR of expr (* e [...] *)
           | AppL of expr (* [...] v *)

(* ステップ番号を格納する変数 *)
let counter : int ref = ref 0

(* 式とその周りのコンテキストを受け取って式を評価する *)
let rec eval (expr : expression) (context : frame list) : expression =
  match expr.desc with
  | Var (x) -> failwith "error: Unbound variable"
  | Fun (x, f) -> expr
  | App (e1, e2) ->
    let arg_value = eval e2 (AppR e1 :: context) in (* 引数部分を評価 *)
    let fun_value = eval e1 (AppL arg_value :: context) in (* 関数部分を評価 *)
    match fun_value.desc with
    | Fun (x, f) ->
      let redex = {desc = App (fun_value, arg_value); (* 簡約前の式 *)
                  attr = expr.attr} in
      let reduct = {desc = (subst f x arg_value).desc; (* 簡約後の式 *)
                  attr = expr.attr} in
      memo reduct reduct context; (* ステップ出力 *)
      eval reduct context (* 簡約後の式を評価 *)
    | _ -> failwith "error: not a function"

(* 空のコンテキストで 式の評価を始める *)
let start (expr : expression) : expression = eval expr []

```

図 5.4: incremental でないステップ関数の実装

print_counter : unit -> unit はコメントとしてステップ番号 (* Step n *) を標準出力する関数、関数 print : expr -> unit は式を標準出力する関数、関数 plug : expression -> frame list -> expression は計算している途中の部分式とコンテキストを受け取って式を再構成する関数であり、実装は省略する。変数 counter には、現在のステップ番号が格納されている。式の評価中に関数適用の簡約を行うたびに 1 ずつ増加させることで、式全体の通しステップ番号を出力できる。

5.4.2 incremental なステップ関数

incremental なステップ関数では、たとえば入力 $2 * 3 + 35$ [@stepper.reduct (1, $5 * 7$)] に対して、以下の 3 種類の処理を実装する。

- 全ステップ出力 $2 * 3 + 35 \rightsquigarrow 6 + 35, 6 + 35 \rightsquigarrow 41$

```

(* 簡約前後の式とコンテキストを受け取って、そのステップを出力する *)
let memo (redex : expression) (reduct : expression) (context : frame list)
: unit =
  let marked_redex =
    (* 簡約前の式に attribute 追加 *)
    {redex with attr = Some ("stepper.redex", None)} in
  let marked_reduct =
    (* 簡約後の式に attribute 追加 *)
    {reduct with attr = Some ("stepper.reduct", None)} in
  print_counter ();
  print (plug redex current_context);
  counter := !counter + 1;
  print_counter ();
  print (plug reduct current_context)
  (* 簡約前ステップ番号を出力 *)
  (* 簡約前のプログラムを出力 *)
  (* ステップ番号を 1 増やす *)
  (* 簡約後ステップ番号を出力 *)
  (* 簡約後のプログラムを出力 *)

```

図 5.5: ステップ出力関数

```

(* 実行の種類 *)
type mode = All | Next | Prev
let mode: mode =
  try (match Sys.getenv "STEPPER_MODE" with
    | "next" -> Next | "prev" -> Prev | _ -> All)
  with Not_found -> All

(* ステップ番号を格納する変数 *)
let counter: int ref =
  try int_of_string (Sys.getenv "STEPPER_COUNT")
  with Not_found -> match mode with All -> ref 0
    | _ -> failwith "no step number"

```

図 5.6: 実行の種類とステップ番号の定義

- 次ステップ出力 $2 * 3 + 35 \rightsquigarrow 6 + 35$
- 前ステップ出力 $2 * 3 + 5 * 7 \rightsquigarrow 2 * 3 + 35$

このうちどの処理を行うかは、図 5.6 のように、ステップ関数のプログラムの実行ごとに環境変数で定めて、mode というグローバル変数に格納する。

次ステップ出力の内容は全ステップ出力の冒頭の 1 ステップであるので、もし前ステップ出力機能を実装しないならば、例えば関数 memo (図 5.5) の最後に

```
;if mode <> All then exit 0
```

と書けばよい。すると mode が Next のときは最初のステップしか出力されず実行が終了し、All のときは全ステップが出力される。

しかし、5.3.1 節で述べたように、「前ステップ出力」処理のためには出力内容を増やしたり、その新しい出力を処理できるようにしなければならない。それぞれの処理において以下のような実装をする必要がある。

```

(* 簡約前後の式とコンテキストを受け取って、そのステップを出力する *)
let memo (redex : expression) (reduct : expression) (context : frame list)
: unit =
  let marked_redex =                                     (* 簡約前の式に attribute 付加 *)
    {redex with attr = ("stepper.redex", None) :: redex.attr} in
  let marked_reduct =
    {reduct with
      attr = ("stepper.reduct",
              Some (!counter + 1,                                (* 簡約後の式にステップ番号と、 *)
              reduct))                                           (* 簡約前の式の情報を追加 *)
      :: reduct.attr} in
  if mode = Prev                                             (* 前ステップ出力のとき、 *)
  then counter := !counter - 1;                             (* ステップ番号を1戻す *)
  print_counter ();                                         (* 簡約前のステップ番号出力 *)
  print_as_attribute (plug redex context);                 (* 処理用の簡約前のプログラム *)
  print (reduct_mapper
    (plug marked_redex context));                          (* 表示用の簡約前のプログラム *)
  counter := !counter + 1;                                  (* ステップ番号を1進める *)
  print_counter ();                                         (* 簡約後のステップ番号出力 *)
  let latter_program = plug marked_reduct context in
  print_as_attribute (latter_program);                     (* 処理用の簡約後のプログラム *)
  print (reduct_mapper latter_program);                   (* 表示用の簡約後のプログラム *)
  if mode <> All then exit 0                                (* 全ステップ実行でなければプロセス終了 *)

```

図 5.7: incremental なステップのための出力関数

- 全ステップ出力：それを新しい入力として前ステップ出力は行わないので変更無し
- 次ステップ出力：そのステップの簡約によって失われる情報を前ステップ出力で復元できるように attribute を付ける
- 前ステップ出力：次ステップ出力時に attribute に書かれた情報から前ステップを導いて出力する

本研究では、incremental でないステップにこれらの作用を更に付け足すことで、incremental なステップを実装する。

情報の付加

incremental でないステップ関数では、例えば $(2 * 3) + (5 * 7)$ に対して、2 ステップ目を

```

(* Step 1 *)
(2 * 3)[@stepper.redex ] + 35

```

```

(* Step 2 *)

```

```
6[@stepper.reduct ] + 35
```

と出力するが、incremental なステップでは 5.3 節で述べたように

```
(* Step 1 *)
[[[@stepper.process (2 * 3) + 35[@stepper.reduct (1, 5 * 7) ] ]
(2 * 3)[@x ] + 35

(* Step 2 *)
[[[@stepper.process 6[@stepper.reduct (2, 2 * 3) ]
+ 35[@stepper.reduct (1, 5 * 7) ] ]
6[@t ] + 35
```

を出力したいので、出力をする関数 `memo` を書き換える必要がある。

図 5.5 にある関数 `memo` の引数は簡約基、それが簡約されたもの、その簡約時のコンテキストの 3 つであったが、新しい出力の内容もこれらの情報から構成することができる。その実装を図 5.7 に示す。関数 `print_as_attribute : expression -> unit` は、`[[[@stepper.process ...]` にプログラムを入れて出力する関数である。

ここで、5.3.2 節で述べたように、表示用のプログラムではハイライトに最低限必要な attribute だけを出力したい。しかし、今簡約している式の外、すなわちコンテキストに含まれる式の中には attribute `[@stepper.reduct]` が含まれうる。上の出力例で処理用の 35 に付いている `[@stepper.reduct (1, 5 * 7)]` がそれである。表示用のプログラムを出力するためには、今のステップで簡約される式以外の attribute を消したものを得る必要がある。

そのために、本研究では OCaml のモジュール `Ast_mapper` を利用した。`Ast_mapper` は OCaml プログラムの構文木を簡単に部分的に変換するためのモジュールである。ここでは詳しい紹介は省くが、`Ast_mapper` を利用して attribute のみを変換する関数 `redex_mapper : expression -> expression` と関数 `reduct_mapper : expression -> expression` を実装した。`redex_mapper` は `[@x]` 以外の attribute を無くす変換をする関数であり、`reduct_mapper` は `[@stepper.reduct (今のステップ, 簡約前の式)]` を `[@t]` にし、それ以外の attribute を無くす変換をする関数である。

以上によって、5.3 節で定めたように、十分な情報を持つプログラムが出力できた。

情報の利用

簡約後の式に attribute が付加できたので、その情報を利用して「前ステップ出力」をする。前ステップ出力には、

1. 今のステップ番号を持つ attribute を探す

2. その attribute が付いた式を attribute 内の式に置換したプログラムを出力する

という操作が必要となる。「今のステップ番号を持つ [`@reduct ...`]」は1つの式だけに付いており、これを探すには式を全て探索する必要がある。そのために、「前ステップ出力」時にだけ利用する新しいインタプリタ関数を作ってもよいが、検索の順序はステッパ関数、すなわち図5.4の `eval` と同じなので、本研究ではこの関数にさらに作用を足すことで前ステップ出力を行う。

前ステップ出力の処理は、関数 `memo` を図5.7のように定義した上で図5.8のように実装することができる。灰色の部分が `incremental` になったことで加わった部分である。灰色の部分は、`if mode = Prev then begin ... end;` であることから分かるように、前ステップ出力モードの時にしか実行されない。

前ステップ出力モードの時の関数 `eval` の実行は、まず今評価している式に [`@stepper.reduct` (今のステップ番号, 簡約前の式)] が付いているかを調べることから始まる。そこで見つければ、その「簡約前の式」を利用して前ステップの出力をしてプロセスを終了する。見つからなければ、`eval` の本体である `match` 文へ進む。今評価している式をパターンマッチして、関数だったらそれ以上の簡約はできず、そのままその関数を返す。関数適用だったら、引数部分の式の実行に移る。するとまたその引数部分の式に [`@stepper.reduct` (今のステップ番号, 簡約前の式)] が付いているかを調べる。あれば出力して終了、無ければ同様に式を普通に評価する。

このように実行を進めると、1以上の正しいステップ番号を変数 `counter` に持っている限り、必ずどこかに [`@stepper.reduct` (今のステップ番号, 簡約前の式)] が付いた式が見つかる。そして、今のステップに至るまで行ってきた「次ステップの出力」と同じ順序で式を探索してきたため、見つかるまでに評価した式は全て簡約済みであり、その式を見つけるまでに簡約処理、すなわち `match fun_value.desc with ...` を行うことは無い。よって、灰色の部分を書き足すことで前ステップ出力が可能になる。

以上のように、`incremental` でないステッパ関数（図5.4, 5.5）を図5.6, 5.7, 5.8のように書き換えることで、`incremental` なステッパ関数を実装できる。

5.4.3 実際のステッパ

実際に我々が実装するステッパは、対象を OCaml の一部としており、以下の構文に対応している（一部は開発中である）。

- 整数、実数、真偽値、文字、文字列、リスト、組、レコード、ユーザ定義型
- 条件分岐、変数定義、再帰関数定義、パターンマッチ、例外処理
- List モジュール、ユーザ定義モジュール
- 配列、逐次実行、標準出力関数（開発中）

```

(* 今のステップ番号の [@stepper.reduct] を探してその中の式を返す *)
let rec find_last_reduct (attrs : attribute list) : expression =
  match attrs with
  | [] -> raise Not_found      (* リストの最後まで見つからなければ例外 Not_found *)
  | ("stepper.reduct", Some (n, redex)) :: _      (* 簡約後のマークがあって、 *)
    when n = !counter -> redex (* その番号が今のステップ番号だったらその式を返す *)
  | _ :: rest -> find_last_reduct rest

(* ステップ実行インタプリタ *)
let rec eval (expr : expression) (context : frame list) : expression =
  if mode = Prev
  then begin try
    let marked_redex = find_last_reduct expr.attr in
    memo marked_redex expr      (* 出力して最後に exit 0 する *)
    with Not_found -> ()      (* expr が探している式でなければ何もせず、 *)
  end;      (* 以下の match 文へ進む *)
  match expr.desc with
  | Var (x) -> failwith "error: Unbound variable"
  | Fun (x, f) -> expr
  | App (e1, e2) ->
    let arg_value = eval e2 (AppR e1 :: context) in
    let fun_value = eval e1 (AppL arg_value :: context) in
    match fun_value.desc with
    | Fun (x, f) ->
      let redex = {expr with desc = App (fun_value, arg_value)} in
      let reduct = {(subst f x arg_value)
                    with attr = expr.attr} in
      memo redex reduct context;
      eval reduct context
    | _ -> failwith "error: not a function"

```

図 5.8: incremental なステッパ関数

<pre>(* Step 4 *) 3 * (fact 2) (* Step 5 *) 3 * (if 2 = 0 then 1 else 2 * (fact (2 - 1)))</pre>	<pre>(* Step 4 *) 3 * (fact 2) (* Step 17 *) 3 * (2)</pre>
--	---

図 5.9: ステップのスキップ機能

副作用に関わる構文について、incremental でない OCaml ステップではステッププロセスが書き換え可能な変数の値や標準出力された文字列の情報を保持することができたが、incremental にすることでそれが不可能になるので、プログラムの attribute (5.2.2 節) を用いてそのような情報もステップ出力に含めるようにすることで実装することを目指している。

また、incremental でない OCaml ステップは、プログラムの流れを理解する助けやステップの利便性の向上のため、関数適用式が値に計算されるまでが1ステップであるかのように進める機能を有していた。図 5.9 の左のように関数適用が簡約されるステップで Skip ボタンを押すと、図 5.9 右のように、下のプログラムが「関数適用式が値に簡約されたステップ」のプログラムに変わる。この状態で Next ボタンを押すとその続き、図 5.9 の場合には $3 * 2 \leadsto 6$ のステップが表示される。これを本研究の incremental なステップでも1ステップとして扱い、1度の実行で関数適用が値になるステップまでを計算し、その最後のステップを出力するように実装を進めている。そのためには、図 5.6 の mode 型にスキップのためのコンストラクタを追加し、そのモードの場合には「実行している関数適用式が値になるステップまでは出力・終了を行わない」ようにするだけで良い。

5.4.4 ツールの実装

ここまで、ステップ関数としてのステッププログラムの実装を紹介した。ユーザが incremental なステップツールを使用するには、ユーザの入力を受けてステッププログラムを呼び出しステップを表示する外部のプログラムが必要になる。

DrRacket のステップ [2] や incremental でない OCaml ステップでは、外部のプログラムは「ステップを起動して、出力を蓄えて、ユーザの操作に従って表示」をしていたが、本研究のステップでは、「ユーザの操作に従ってステップを呼び出して、出力されたものを装飾して表示」をする。すると、外部のプログラムではステップ番号と前回の出力のみを保持することで実装が可能になる。

5.5 予想される問題点とその経過

本節では、incremental な OCaml ステップを実際に利用するときに発生しうる問題点とその解決方法を挙げ、大学の授業 (6.1 節で触れる) で利用してそれらが実際に問題になったかどうかを述べる。

5.5.1 文字数の爆発

問題点

本章のステップでは、任意のステップの「処理用の出力」に入力プログラムからそこまでの簡約の過程が記されているため、1 ステップ進むごとに文字数が増加する。具体的には、簡約基 `e1` が式 `e2` に簡約されるステップでは、その時点のコンテキストを `E`、ステップ番号を `n` とすると、簡約前のプログラムが `E[e1]`、簡約後のプログラムが `E[e2[@stepper.reduct n;; e1]]` となる。`E` の文字数は変わらないので、`e2[@stepper.reduct n;; と]` の分の文字数が増加することになる。これを続けていくと、1 ステップあたり少なくとも 22 文字は増加することになり、仮に百万ステップの簡約をするとプログラムは数千万文字になる。すると、毎ステップの入出力や通信に時間がかかる可能性がある。

解決方法

解決策としては、古いステップの `attribute` は削除してしまうという方法が考えられる。たくさんのステップを見てから最初の方のステップまで戻るユーザは少ないと仮定すれば、ある程度前のステップについての `attribute` があったら、関数 `memo` で出力するプログラムを再構成する際に消去すれば、さほど実際の使用に影響なく出力する文字数を減らすことができる。

使用した結果

実際の incremental な OCaml ステップ (5.4.3 節) では、Emacs Lisp プログラムによってステップ関数の実行や表示を制御する。これを用いた授業では、文字数の多さが原因の不具合は報告されず、1 ステップの入出力にかかる時間も利用に支障が出ない程度だった。

5.5.2 実行時間

問題点

ステップ数が膨大になると、後ろの方までステップ実行をするのは困難である。incremental なステップ関数での実行速度は通常の OCaml 処理系に決して及ばないので、1 ステップずつ進める場合でも、スキップ機能 (5.4.3 節) を使う場合でも、実行を多く進めるには長い時間が掛かってしまう。

解決方法

少しでも急ぐ為には、一般的なデバッガのように、ステップ実行したい式の付近にユーザがブレークポイントを設定して、そこからステップ実行を始めるという方法が考えられる。そのためには、ブレークポイントまでを部分的に native code にコンパイルして実行するなどの方法を取らざるを得ない。しかしいずれにしても、通常の OCaml コンパイラを用いても長時間かかるプログラムの実行を早く終わらせることは不可能である。

使用した結果

1 ステップずつ進める場合は、プログラムの実行を後ろの方まで進めることは困難であるが、ステップごとの実行時間が問題になることはなかった。スキップ機能を使用する場合は、やはり通常の OCaml で実行するよりも長い時間を要した。具体的には、東京メトロ全線の駅についてダイクストラ法で最短路問題を解くプログラムの実行に、通常の OCaml では 0.327 秒、incremental な OCaml ステップではスキップ機能を使用して約 54 秒かった。

5.5.3 関数適用評価スキップ後の前ステップ出力

問題点

関数適用をスキップ（5.4.3 節）した後に前のステップに戻ろうとすると、スキップで飛ばされたステップには戻ることができない。たとえば図 5.9 のように 2 の階乗の計算をスキップしたとすると、図 5.9 の右の状態から、incremental でないステップでは

- スキップをする前の関数適用式が簡約されるステップ 5（図 5.9 左）
- 関数適用式が最終的な値になるステップ 17 ($3 * (2 * 1) \rightsquigarrow 3 * 2$)

のどちらにも戻ることができたが、incremental なステップでは前者にしか戻ることができない。

その原因は、incremental でないステップでは 5.5.2 節で述べたように文字列検索によってステップ表示を切り替えていたので任意のステップに移ることができたのに対して、incremental なステップではステップ関数が関数適用式が値になるまでを 1 ステップとして出力し、その間の簡約についての情報は出力しないからである。

解決方法

これを解決するには、スキップする部分の計算をしている間の簡約についても attribute に情報を蓄え、スキップ後のステップで全ての簡約の情報が入った長いプログラムを出力する必要がある。しかしそのようにすると 5.5.1 節と 5.5.2 節の問題がより深刻になる可能性がある。

使用した結果

ステッパを利用した学生に答えてもらったアンケートに「ステッパの不便なところ、欲しい機能などがあれば教えてください。」という質問を含めたが、この点に関する要望は出なかった。

5.6 この章のまとめ

本研究では、OCaml の一部の構文に対するステッパを、1 度の実行で 1 つの簡約のみを行うように変更し、ステッパの起動時間を短縮した。そのためには任意のステップのプログラムからそれ以前のステップのプログラムを計算できるようにする必要性が生まれたので、それまでの簡約の内容を全て記録したプログラムを出力することで簡約過程の情報が失われないようにした。

第6章 評価

2016 年度から 2018 年度の間、我々は 3.4 節で述べた OCaml ステッパをお茶の水女子大学で毎年前期に開講されている授業「関数型言語」において利用してきた。この章では、授業で実際に学生に OCaml ステッパを使ってもらったことで得られた実行ログやアンケート結果からステッパの教育的効果について考察する。

まず 6.1 節でステッパを使用した授業の内容や進め方について説明する。そして 6.2 節でステッパの効果について考察する。

6.1 授業の内容

「関数型言語」は OCaml を用いて関数型プログラミングを学ぶ授業であり、再帰、データ型、副作用、モジュールなどの基本的な言語機能を扱う。学期全体を通して、学生はダイクストラ法に基づいて最短路問題を解くプログラムを作成する。授業は週に 1 回、90 分間で、全 15 回で構成されている。（授業終了後も教室に残って作業を続ける学生も多い。）受講者は情報科学を専攻する学部 2 年生で、毎年 40 人程度である。全ての学生がこの授業より前に C 言語によるプログラミングを経験している。

この授業では「反転授業」を行なっている。反転授業とは、学生が各自内容を予習し、授業時間は講義を行わず演習などに活用する授業形態である。この授業では、学生は毎週の授業の前に、この授業に向けた教科書 [15] と教員が作成した動画による予習をし、インターネット上で数問の簡単な問題に解答する。授業時間内には、学生はその回に新しく取り上げた内容に関する演習問題に取り組み、教員および 5～8 人のティーチングアシスタントに質問することができる。

演習問題はほとんどが要求を満たす関数を実装するというものであり、「練習問題」と「レポート課題」に分かれている。練習問題は学生がどの程度その回の内容を理解しているか確認するための簡単な問題で、授業時間内に解くことを期待している。レポート課題は評価に利用する問題で、締め切りが 1 週間後に設定されている。

授業をする演習室では常に OCaml プログラム実行のログをとっている。学生がプログラムを実行する時には必ず、ステッパによる実行と通常の実行のどちらも、個人情報 that 特定されない形でプログラム全文と実行ログが記録される。

解答プログラムが合っているかどうかを確認する方法の 1 つとして、「チェックシステム」を用意してある。チェックシステムは教員が用意した単体テストに通るかどうかを確認できるプログラ

ムで、問題ごとに用意されており、12 週目までのほとんどの問題を対象にしている。各学生が各問題のチェックシステムに初めて通った時には学生の ID が記録されるようになっており、締め切りまでに 1 度以上チェックシステムに通ることが、レポート課題の各問で点を得る条件に含まれている。

6.2 結果

ステップアの学習的効果を評価するため、我々は以下の項目について調査した。

- ステップアがどれほど使われていたか
- ステップアをよく使った年と使わなかった年で学生の解答の速さに違いがあったか
- 学生がステップアについてどう感じたか

ステップアがどれほど使われたかについては、OCaml 実行ログからステップアによる実行とそうでない実行の数を数えた。解答の速さの差は、チェックシステムに記録されている時刻、すなわち学生が初めて正解のプログラムをチェックシステムに通した時刻から統計的に分析した。最後の項目については、学生にアンケートを取った。

6.2.1 ステップアの使用状況

我々が授業「関数型言語」にステップアを導入したのは 2016 年度である。ステップアのインタフェースは 2016 年度から 2018 年度まで概ね変わらない (図 1.3) が、以下の点で各年度のステップアの機能は異なる。

2016 年度 関数定義、条件分岐、レコード、リスト、再帰のような非常に基本的な式と一部の演算子のみが実行できるステップアを利用した。また、教員がステップアを学生に紹介する際に、ステップアの使用を強くは奨励しなかった。2016 年度はステップアによるプログラム実行のログを採取していなかったためステップアがどれくらい利用されていたのかは不明であるが、最初の数週にわずかに利用されたのみだと我々は考えている。

2017 年度 6 週目までに学習する構文要素のほとんどに対応したステップアを利用した。教員は最初の授業で事前にステップアを紹介し、その後の授業でも様々な例でステップアを利用して見せた。

2018 年度 1.3 節で紹介した、モジュール、例外処理、逐次実行、配列などを実行できるステップアを利用した。これは授業範囲のほとんどの構文要素を含んでいる。また関数適用のスキップ機能も追加した。教員は、まるでステップアでしかプログラムが実行できないかのように学生に紹介することで、学生にステップアの利用を促した。(数週の間には学生は普通のインタプリタで実行ができることにだんだん気付いていった。)

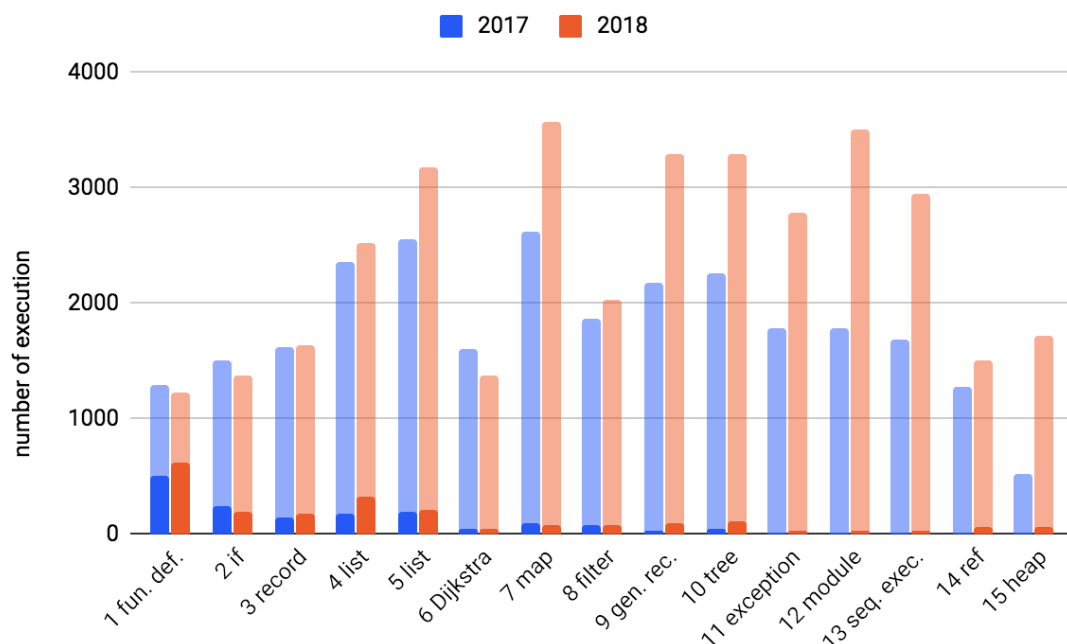


図 6.1: 2017 年度と 2018 年度の各週の、通常の実行 (薄い色) とステップ実行 (濃い色) の回数。2017 年度は終盤では全く使われなくなったが、2018 年度にはある程度使われ続けている。

図 6.1 は、全受講生の全実行 (エラー終了した実行を含む) のうち何回がステップパでの実行だったかを示すグラフである。2017 年度と 2018 年度の両方で、5 週目まではステップパが頻繁に利用されている。この一因は、バグの発見や再帰の理解のためにステップパを使うように教員やティーチングアシスタントが学生に勧めたことである。6 週目以降は、学生がインタプリタに気付き始めたことと実行するプログラムが大きくなったことにより使用回数が減少した。

2017 年度には、ステップパの使用回数は最後の授業に近づくにつれて減少している。それに対して 2018 年度は、例外処理、モジュール、逐次実行、書き換え可能な変数に対応したので、ある程度の回数の使用が見られる。図 6.2 は、2018 年度のステップ実行のうちこの年に新しく対応した構文を利用したプログラムの実行の回数のグラフである。どの構文もいくらかステップ実行されており、例外処理やモジュールのような高度な構文のステップ実行の需要があることが分かる。

図 6.1 と図 6.2 のグラフの正確な数値は付録の表 A.1 に掲載する。

6.2.2 ステップパの効果

ステップパのようなツールの教育効果を測ることは簡単ではない。例えばコンパイラのエラーメッセージ改善の評価ならば、エラーを分類して改善されたエラーメッセージがそのうちどれほどに対応しているかを見ることができだろう。ステップパにおいては、そういった定量的なデータをどのように示せばよいかが不明瞭である。

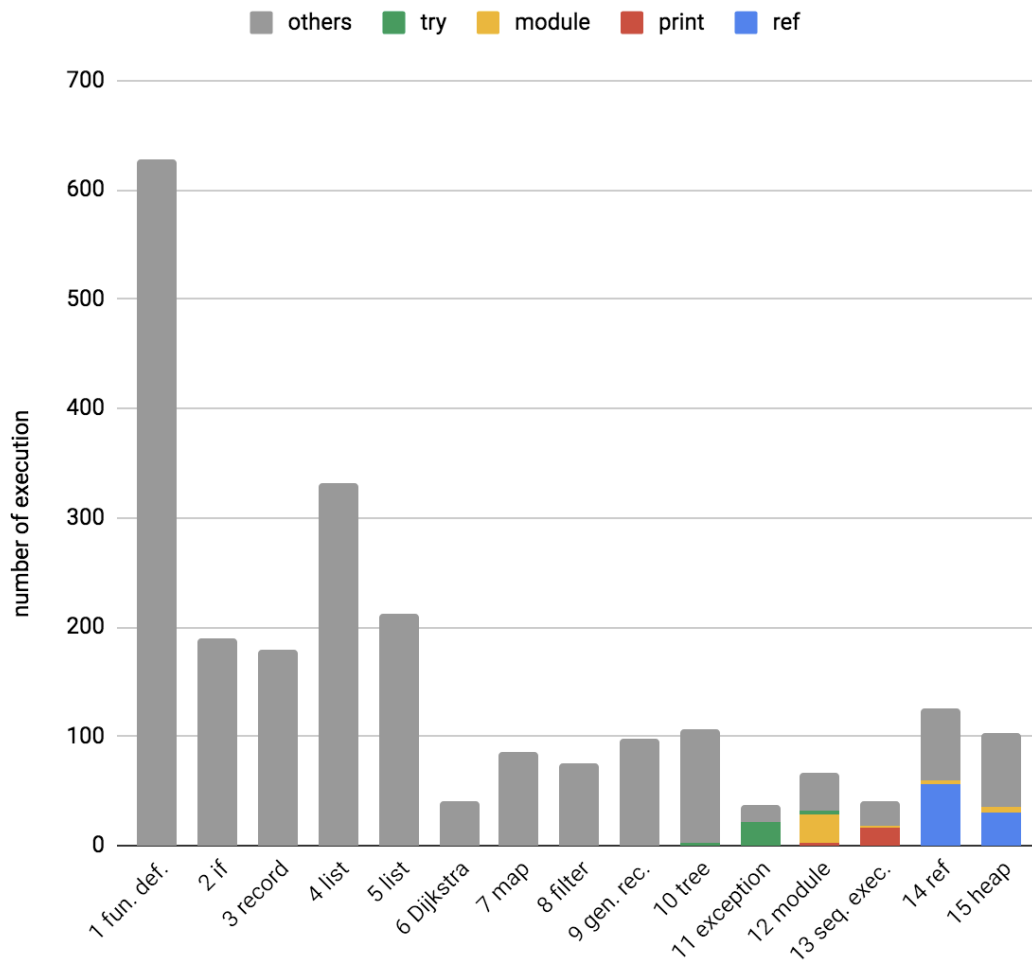


図 6.2: try、module、print、ref を含むプログラムのステップによる実行の回数

ステップの効果を測定するための試みとして、我々は学生がチェックシステムに正しい解答を提出するまでにかかった時間を調査した。全ての問題のチェックシステムに記録されている「正答が初めて提出された時刻」の中から授業開始から 100 分以内の時刻を収集し、2016 年、2017 年、2018 年の間で比較した。

図 6.3 は 2017 年度と 2018 年度それぞれの、2016 年度よりも学生が有意に短い時間で正解を提出した問題の数を示している。このデータは $p < 0.05$ の片側 t 検定に基づいており、詳しい数値は付録の表 A.2 にある。ただし、2018 年度のみ初回の授業に特別なテストを行ったので、第 1 週はこの調査の対象から外した。

図から、実質的なステップの導入の後、提出までの時間が改善されていることが分かる。序盤の問題で特に改善が見られる。しかし 1 つ例外があり、第 6 週の 1 つの問題のみ、2018 年度の正解提出が有意に 2016 年度よりも遅かった。該当する問題は、リストを受け取ったらそのリストの全要素に 1 を足したリストを返す再帰関数を書くという単純な問題である。この問題の正答が

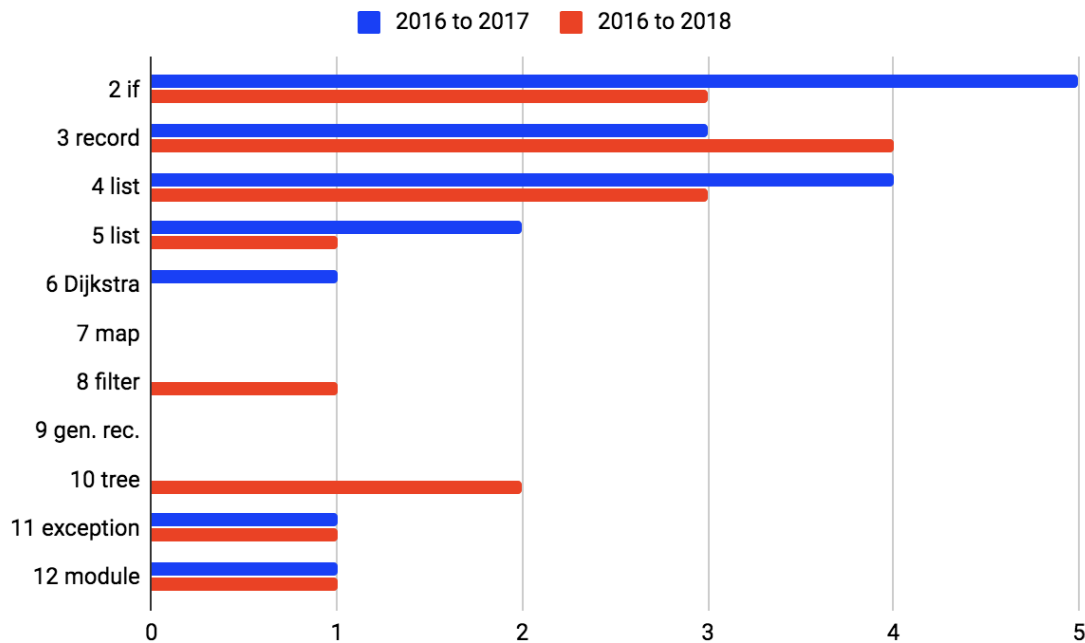


図 6.3: 2017 年度と 2018 年度に、学生が 2016 年度より有意に早く正答を提出した問題の数

2018 年度に遅くなった理由は分からない。全問題の正答時間をまとめて t 検定した結果は、2017 年度が $t(1778) = 2.819$ ($p=0.002$)、2018 年度が $t(2111) = 2.592$ ($p=0.005$) だった。

2017 年度と 2018 年度の平均時間も比較した。第 5 週までの早い週には 2017 年度の方が有意に早く、それより後は 2 問のみ 2018 年度の方が早く、あとは有意差は見られなかった。こちらも全問題の時間をまとめて t 検定すると、 $t(1953)=0.455$ ($p=0.324$) となり、有意な差はなかった。

有効性に対する脅威 この調査の結果は、各年の学生の影響を受けた可能性がある (同じ学生が 2 度以上履修したことはない)。また、3 年間を通して毎回の授業時間の最初に教員が前回の課題や今回の内容などについて話す時間をとっており、その長さは毎回同じではなかった。毎年ほとんど同じようなコメントをしてはいるが完全に同じではないので、結果に影響した可能性がある。

6.2.3 学生による評価

2018 年度前期の最後に、学生にステップパについてのアンケートを実施した。42 人の受講者のうち 38 人が回答した。その中で、ステップパがどれくらい便利だったかを 0 から 4 点で評価してもらった。結果を表 6.1 に示す。この表からはステップパはいつでも役立つ万能なツールではないことが分かる。しかし、多くの学生がステップパを使って時々問題を解決することができている。「ステップパを使えばほぼ必ずプログラムの動きや間違いの原因が分かった」を選んだ学生もいた。

他の質問で、プログラムの間違いの発見や理解のためにステップパを活用できた場面を (あれば)

	点数	人数
ステップを使えばほぼ必ずプログラムの動きや間違いの原因が分かった	4	3
ステップを使って問題を解決できることが多くあった	3	8
たまにステップを使って問題を解決できることがあった	2	25
ステップを使うことで新たに何かが分かることはほとんど無かった	1	2
全く役に立たなかった、または使わなかった	0	0

表 6.1: 2018 年度の学生による点数評価。42 人中 38 人が回答。平均は 2.3 だった。

答えてもらった。その答えを 2 つに分類して紹介する。

プログラムの挙動の理解 7 人の学生が、プログラムの挙動についての理解を深めることができたと答えた。特にその内 5 人が関数がどのように再帰的なデータを消費するのかを把握するのに役立ったと書いている。再帰関数定義が展開されて関数適用の入れ子になっていく様子を見ることができることは再帰の理解の助けになると考えられる。

他の学生は、プログラム全般の動きを観察できたと答えた。その学生たちは、引数が関数呼び出しの前に 1 つずつ実行されることを発見した。1 人は OCaml プログラムが right-to-left で実行されていることに気付いた。以前は、このような細かい動作について特に強調せず授業を進めていた。

デバッグ 多くの学生がデバッグにステップを利用できたと回答した。16 人の学生が、テストを通らなかった時にどこが間違っているのかを見つけることができたという。実行の各ステップを観察することで、期待と違った振る舞いをする箇所を特定することができたという。これは一般的なデバッグをする上で重要な過程である。この授業では出力などの副作用を終盤で学習するので、学生のデバッグの方法は単体テストで関数全体が正しい値を返すかどうかを見るしかなかった。しかしステップがあれば、学生は実行中のどこで間違いが起きているのかを単純に観察することができる。

また、3 人の学生がプログラムが停止しない時にその原因をステップで突き止めることができた。出力機能を使わずに無限ループの原因を特定することにもステップは利用できる。

第7章 結論

本研究では、より多くの言語機能への対応と 1 ステップずつの可逆的な実行によってステップパをさまざまな言語や環境で利用できる可能性を示し、また例外処理やモジュールなどの機能に対応したステップパが関数型言語の学習に及ぼす効果を観察した。

まず、try-with や algebraic effects を例に制御オペレータに対応したステップパ関数を実装した。ステップパとはプログラムの実行を簡約ステップの列として見せるツールであり、全ての簡約の前後で、簡約による書き換えの前後のプログラム全体を出力することが求められる。プログラム全体を出力できるステップパ関数を実装するためには、コンテキストの情報を保持しながら実行を進めるインタプリタを用意すればよい。本論文では全体を通して、通常の big-step インタプリタ関数にコンテキスト情報を保持するための引数を追加し、簡約の前後で実行中の部分式とそのコンテキスト情報からプログラム全体を再構成して出力するようにすることでステップパ関数を実装した。

インタプリタ関数にコンテキスト情報を保持する引数を追加する方法は、3 章と 4 章で別の方法をとっている。例外処理の構文 try-with を含む言語に対するステップパ関数を実装する際には、コンテキストを try 節ごとに区切った 2 層のリスト構造として定義し、それをインタプリタの再帰の構造に従ってフレームを付け足しながら渡すようにした。algebraic effects を含む言語のステップパ関数では、オペレーション呼び出しによって限定される範囲の継続を引数に持ち部分的に CPS であるインタプリタを定義し、そのインタプリタに非関数化、CPS 変換、非関数化という 3 回のプログラム変換を施すことでメタ継続を 1 次データとして取得した。そこからコンテキスト情報を得てステップパを実装した。

次に、incremental なステップパを実装した。incremental なステップパのステップパ関数は、1 ステップを簡約したら実行を終え、次のステップの実行を始めるには前回の出力を入力として受け取るという特徴がある。特に、前のステップに戻るという実行をする際にも前回の出力を入力とするため、実行を逆向きに進める必要がある。簡約は基本的に式の情報を失わせる書き換えであり、簡約後の式から簡約前の式を導出することは不可能である。そこで簡約後の式にユーザから見えなように簡約前の式の情報を付加することで戻る操作を実現した。

そして、6 章ではステップパがプログラミング学習に及ぼす効果を調べた。同じ関数型言語の授業において、ステップパをあまり使わなかった年とステップパを多く使った年を比較すると、ステップパを多く使った年の方が学生が正答するまでにかかる時間が優位に短かった問題が複数あった。よって、ステップパが利用できる方が正しいプログラムを早く書けるようになる可能性があるといえる。

今後の課題として、以下を考えている。

- algebraic effects 構文を持つ OCaml である Multicore OCaml の一部の構文に対応したステップパを実装する。
- incremental なステップパを利用して、クライアントサーバ方式でステップパツールを実装する。
- 他の制御演算子を含む言語のステップパ関数の導出方法を考える。

これらによって、様々なプログラミングの場面でステップパが使えるようになることを期待する。

謝辞

本研究にあたり、指導教官の浅井健一准教授にはとても丁寧に辛抱強くご指導頂きました。心より感謝申し上げます。また、多くのご助言を頂いた副査の戸次大介准教授に深く感謝いたします。OCaml ステッパの開発者である東京工業大学情報理工学院の叢悠悠先輩には様々な面で支えていただき、本当にありがとうございました。そして幾度となく力を貸してくれた石尾千晶さんと北川舞さんをはじめとした浅井研究室の皆様、ステッパを利用していただいた後輩の皆様、私の学生生活を支えてくれた両親に感謝いたします。

関連図書

- [1] Dariusz Biernacki, Maciej Piròg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: Relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.*, Vol. 2, No. POPL, December 2017.
- [2] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *European symposium on programming*, pp. 320–334. Springer, 2001.
- [3] Youyou Cong and Kenichi Asai. Implementing a stepper using delimited continuations. *7th International Symposium on Symbolic Computation in Software Science*, Vol. 39, pp. 42–54, 2016.
- [4] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, LFP '90, pp. 151–160, 1990.
- [5] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. MIT Press, 2009.
- [6] Matthias Felleisen and Daniel P. Friedman. Control operators, the secd-machine, and the λ -calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pp. 193–219. Elsevier, 1986.
- [7] Tsukino Furukawa, Youyou Cong, and Kenichi Asai. Stepping OCaml. In Proceedings Seventh International Workshop on *Trends in Functional Programming in Education*, Chalmers University, Gothenburg, Sweden, 14th June 2018, Vol. 295 of *Electronic Proceedings in Theoretical Computer Science*, pp. 17–34, 2019.
- [8] Daniel Hillerström and Sam Lindley. Shallow effect handlers. In Sukyoung Ryu, editor, *Programming Languages and Systems*, pp. 415–435, Cham, 2018. Springer International Publishing.
- [9] Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. Continuation passing style for effect handlers. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*, Leibniz International Proceedings in Informatics (LIPIcs), pp. 18:1–18:19, 9 2017.

- [10] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pp. 145–158, New York, NY, USA, 2013. Association for Computing Machinery.
- [11] Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, Vol. 319, pp. 19–35, 2015. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).
- [12] Preston Tunnell Wilson, Kathi Fisler, and Shriram Krishnamurthi. Evaluating the tracing of recursion in the substitution notional machine. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pp. 1023–1028. ACM, 2018.
- [13] John Whittington and Tom Ridge. Direct interpretation of functional programs for debugging. In Sam Lindley and Gabriel Scherer, editors, *Proceedings ML Family / OCaml Users and Developers workshops*, Oxford, UK, 7th September 2017, Vol. 294 of *Electronic Proceedings in Theoretical Computer Science*, pp. 41–73, 2019.
- [14] 根岸純一, 岩崎英哉. Haskell プログラムの開発を支援する GHCi デバッガフロントエンド. 情報処理学会論文誌プログラミング, Vol. 2, No. 3, pp. 48–56, July 2009.
- [15] 浅井健一. プログラミングの基礎. サイエンス社, February 2007.

付 録 A 学生の実行ログから得られたデータ

week	2017						2018						contents
	all	step.	try	mod.	print	ref	all	step.	try	mod.	print	ref	
1	1293	504	0	0	0	0	1233	627	0	0	0	0	fun. def.
2	1511	235	0	0	0	0	1375	189	0	0	0	0	if
3	1618	144	0	0	0	0	1641	179	0	0	0	0	record
4	2364	169	0	0	0	0	2517	332	0	0	0	0	list
5	2556	193	0	0	0	0	3173	213	0	0	0	0	list 2
6	1596	43	0	0	0	0	1369	41	0	0	0	0	Dijkstra
7	2621	92	0	0	0	0	3570	86	0	0	0	0	map
8	1874	81	0	0	0	0	2028	75	0	0	0	0	filter
9	2184	34	0	0	0	0	3300	98	0	0	0	0	gen. rec.
10	2254	48	0	0	0	0	3298	106	3	0	0	0	tree
11	1783	20	10	0	0	0	2790	37	22	0	0	0	exception
12	1785	12	8	4	0	0	3501	37	3	26	3	0	module
13	1678	10	0	7	2	0	2943	22	0	3	16	0	seq. exec.
14	1280	11	0	0	0	1	1511	65	0	4	0	56	ref
15	517	6	0	0	0	0	1717	68	0	5	0	30	heap

表 A.1: 2017 年度と 2018 年度の全ての実行 (all) のうちステップの使用回数 (step.)。try、mod.、print、ref の列はそれぞれ例外処理、モジュール、標準出力 (と逐次実行)、書き換え可能な変数 (配列を含む) を表している。右端の列はその週に取り上げた内容である。

week. problem	2016 to 2017			2016 to 2018			contents
	t	p	+/-	t	p	+/-	
2.r1	t(55)=2.098	p= 0.020	dec	t(60)=0.635	p=0.264	dec	if
2.r2	t(56)=2.364	p= 0.011	dec	t(57)=1.831	p= 0.036	dec	
2.r3	t(54)=1.896	p= 0.032	dec	t(59)=0.751	p=0.228	dec	
2.1	t(66)=3.006	p= 0.002	dec	t(74)=3.372	p= 0.001	dec	
2.2	t(56)=3.672	p= 0.000	dec	t(62)=3.036	p= 0.002	dec	
3.r1	t(52)=3.222	p= 0.001	dec	t(61)=2.936	p= 0.002	dec	record
3.r2	t(42)=2.339	p= 0.012	dec	t(56)=3.467	p= 0.001	dec	
3.r3	t(41)=1.373	p=0.089	dec	t(51)=2.688	p= 0.005	dec	
3.1	t(28)=5.610	p= 0.000	dec	t(38)=2.753	p= 0.004	dec	
3.2	t(17)=1.655	p=0.058	dec	t(27)=0.105	p=0.459	dec	
3.3	t(16)=1.546	p=0.071	dec	t(13)=0.603	p=0.279	dec	list
4.r1	t(47)=2.088	p= 0.021	dec	t(61)=2.446	p= 0.009	dec	
4.r2	t(48)=1.909	p= 0.031	dec	t(60)=2.267	p= 0.014	dec	
4.1	t(51)=2.134	p= 0.019	dec	t(60)=2.473	p= 0.008	dec	
4.2	t(18)=3.033	p= 0.004	dec	t(20)=0.489	p=0.315	dec	
5.r1	t(42)=1.037	p=0.153	dec	t(55)=0.257	p=0.399	inc	list 2
5.1	t(49)=1.592	p=0.059	dec	t(61)=0.904	p=0.185	dec	
5.2	t(55)=4.138	p= 0.000	dec	t(62)=1.631	p=0.054	dec	
5.3	t(47)=3.305	p= 0.001	dec	t(50)=1.940	p= 0.029	dec	
6.r1	t(30)=0.322	p=0.375	inc	t(51)=2.011	p= 0.025	inc	Dijkstra's algorithm
6.1	t(41)=1.678	p=0.050	dec	t(61)=1.155	p=0.126	dec	
6.2	t(45)=1.415	p=0.082	dec	t(62)=0.976	p=0.166	dec	
6.3	t(34)=2.296	p= 0.014	dec	t(42)=0.548	p=0.293	dec	
7.r1	t(42)=0.462	p=0.323	inc	t(56)=0.314	p=0.377	dec	map
7.r2	t(41)=0.286	p=0.388	inc	t(54)=1.181	p=0.121	dec	
7.r3	t(40)=0.677	p=0.251	inc	t(51)=1.492	p=0.071	dec	
7.1	t(21)=0.965	p=0.173	dec	t(20)=0.372	p=0.357	dec	
7.2	t(12)=0.380	p=0.355	inc	t(7)=0.686	p=0.258	dec	
8.r1	t(46)=1.162	p=0.126	inc	t(58)=2.694	p= 0.005	dec	filter
8.1	t(16)=0.844	p=0.205	dec	t(22)=0.841	p=0.205	dec	
9.r1	t(44)=1.294	p=0.101	dec	t(55)=0.678	p=0.250	dec	general recursion
10.r1	t(48)=0.312	p=0.378	dec	t(51)=1.308	p=0.098	dec	tree
10.r2	t(48)=0.457	p=0.325	dec	t(50)=1.760	p= 0.042	dec	
10.1	t(33)=0.976	p=0.168	dec	t(38)=0.845	p=0.202	dec	
10.2	t(19)=1.498	p=0.075	dec	t(19)=0.871	p=0.197	dec	
10.3	t(15)=1.538	p=0.072	dec	t(12)=2.240	p= 0.022	dec	
11.r1	t(43)=0.272	p=0.393	dec	t(53)=1.555	p=0.063	dec	exception
11.r2	t(37)=0.454	p=0.326	dec	t(44)=1.906	p= 0.032	dec	
11.r3	t(31)=0.050	p=0.480	inc	t(40)=1.208	p=0.117	dec	
11.1	t(34)=1.567	p=0.063	dec	t(46)=1.518	p=0.068	dec	
11.2	t(28)=2.204	p= 0.018	dec	t(36)=1.563	p=0.063	dec	
11.3	t(17)=0.604	p=0.277	dec	t(21)=0.384	p=0.352	dec	module
12.r1	t(39)=2.229	p= 0.016	dec	t(52)=4.009	p= 0.000	dec	
all	t(1778)=2.819	p= 0.002	dec	t(2111)=2.592	p= 0.005	dec	

表 A.2: 授業開始から学生が正解を提出するまでにかかった時間を比較した片側 t 検定の結果と p 値。列 +/- は平均値が増加したか減少したかを表している。0.05 未満の p 値に、平均時間が減少した問題は 赤色、増加した問題は 水色 を付けた。