OCaml ステッパの機能拡張

古川つきの　(指導教員：浅井 健一)

1　はじめに

本研究で扱う OCaml ステッパとは、叢ら [1] が提案、実装した、OCaml プログラムの計算過程を一つ進むごとに出力するプログラムである。ステッパは、プログラミングを学ぶ者がプログラムの実行のされかたを理解する目的や、デバッグのために利用されることを想定している。実際に、2016 年度から「プログラミングの基礎 [3]」に基づいたお茶の水女子大学の授業「関数型言語」において、OCaml の初学者が多い学生にステッパの利用を推奨し、学習に役立てている。

本研究では、ステッパを OCaml の例外処理に対応させ（3 節）、利便性を損なっていた点を解消し（4 節）、出力に関する機能を追加する（5 節）ことで、ステッパをより実用的なプログラムに改良した。

2　ステッパプログラム

本研究の OCaml ステッパは一つのプログラムとして実装されており、引数に OCaml ソースファイルのファイルパスを受け取り、ステップ実行の様子を表した文字列を返すプログラムである。また、ステップが進むときに、どの部分が実行されているのかを示す符号を OCaml の attribute として付けており、ステップ実行の文字列を表示する際にその部分をマークできるようにしている。例えば、

```
let f x = x + 1;;  100 + f 5
```

というプログラムを入力すると、出力は以下のようになる。

```
(* Step 0 *) 100 + f 5
(* Step 1 *) 100 + (5 + 1)
(* Step 1 *) 100 + (5 + 1)
(* Step 2 *) 100 + 6
(* Step 2 *) 100 + 6
(* Step 3 *) 106
(* Step 3 *) 106
```

3　例外処理

OCaml には、C++ や Java などと同じような例外処理をする構文がある。先行研究 [1] では例外処理が含まれたプログラムを正しく処理できなかったので、本研究では例外処理に対してもステップ表示を定め、実装した。

3.1　例外処理のステップの定義

まず、例外処理の適切なステップ実行のしかたを定めた。たとえば以下のようなステップになる。

```
try 1 + 6 / 3 / 0 + 5
with Division_by_zero -> -1

try 1 + 2 / 0 + 5
with Division_by_zero -> -1

try 1 + (raise Division_by_zero) + 5
with Division_by_zero -> -1

try (raise Division_by_zero)
with Division_by_zero -> -1

-1
```

ただし、OCaml においてゼロ除算エラーの発生は `raise Division_by_zero` が実行されることと等しいこととする。

また、例外が捕捉されないままその式が終わると OCaml ではプログラムの実行が終了されるが、ステッパでも同様に式全体が `raise Division_by_zero` のような「例外発生」のみになったら、そこを最終ステップとして続きの実行は行わないようにした。

3.2　ステップの出力と例外

本研究のステッパでは、

```
1 + 6 / 3 / 0 + 5
```

という入力を与えると、1 ステップ目に

```
1 + 2 / 0 + 5
```

という式を出力する。実際にステッパの中で `6 / 3` の計算をするとき、全体の式のどこに `6 / 3` があるのかという情報がなければ、`1 + 2 / 0 + 5` を出力することができない。そこで、`6 / 3` を計算しているときには、

```
(1 + ([ ] / 0)) + 5
```

という、「その式の周りにある式」が保存されている。この情報を保存するために、Kiselyov ら [2] の delimcc ライブラリを使用している。

しかし、ステップ実行の中に例外発生があると、「周りにある式」の情報を突然変更する必要が出てきてしまう。前節の例で、`raise Division_by_zero` が起こると、そのとき周りの式の一部（1 を足して 5 を足すという計算）をやらずに、例外処理をする `try` 式である `try [ ] with ...` の実行に移る。つまり `2 / 0` が `raise Division_by_zero` に計算されるときに、その周りにある式 `try 1 + [ ] + 5 with Division_by_zero` が、`try [ ] with Division_by_zero` に変化してしまう。

3.3　実装

例外が起こったあとにどこの式の実行に移るかを決定することは、入力プログラムに書いてある制御そのものであって、ステッパ内でも例外を起こすことでそれを再現できる。そこで、例外発生 `raise e` を実行する場合に、ステッパの内部では `raise (Error (e, con))` というような例外を発生させる。この `con` は、最も内側にある `try` 式より内側の「周りの式」である。ステッパの内部で起こした例外は、`try` 式の制御部分で捕捉される。例外が捕捉されてからステップ出力を行うことで例外が起こらない場合の動作には影響を及ぼさず、またエラーの情報に「周りの式」が捨てられる前の状態の式を含めることで例外が発生した時点での全体の式を出力することができた。

4 関数適用のスキップ

大きなプログラムのデバッグのために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う。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。

4.1 スキップ機能の必要性

たとえば、以下のようなプログラムをステップ実行することを考える。

```
let rec sum n =
  if n < 1 then 0 else n + sum (n - 1) in
let rec sumlist lst = match lst with
 [] -> []
 | first :: rest ->
   sum first :: sumlist rest in
sumlist [1; 2; 3; 4]
```

関数 sum は 1 から引数までの自然数の総和を再帰的に求める関数である。関数 sumlist は、引数のリストのすべての要素に関数 sum を適用させたリストを返す関数である。このプログラムのステップ数は 72 に及ぶ。その中の 10 ステップ目が、以下のような式になる。

```
[sum 1; sum 2; sum 3; sum 4]
```

すなわち、最初の 10 ステップで関数 sumlist の主要な計算は終わり、残りの 62 ステップはすべて sum を適用させる計算である。全てのステップを見ないと最終結果 [1; 3; 6; 10] にたどり着けないことはプログラミングの理解やデバッグという目的に合わず、ステッパの大きな欠点だった。

4.2 スキップ機能の動作

そこで、関数適用が始まるステップと、その関数適用の結果が 1 つの値になるステップにそれぞれ目印を出力することで、ステッパの UI を提供するアプリケーションで関数適用の計算過程を飛ばして表示させることができるようにした。

```
[sum 1; sum 2; sum 3; sum 4]
```

の次のステップ

```
[sum 1; sum 2; sum 3;
 if 4 < 1 then 0 else 4 + sum (4 - 1)]
```

はそのまま見られるが、関数適用を飛ばすという選択をすると

```
[sum 1; sum 2; sum 3; 10]
```

という表示が得られる。

4.3 実装

先行研究 [1] での、関数適用のステップを出力するための処理は、

1. 関数適用前の式を緑色で出力
2. 関数の中身を展開し、実際の引数を代入（β 簡約）
3. 代入後の式を紫色で出力
4. 代入後の式を実行

という順で行われていた。この順序で処理をしていると、関数適用の結果が 1 つの値になるステップを知ることができず印がつけられない。そこで、

1. 関数適用前の式を緑色で出力
2. 関数の中身を展開し、実際の引数を代入（β 簡約）
3. その時点でのステップ番号を保持し、ステップ番号と関数適用が始まる旨を出力
4. 代入後の式を紫色で出力
5. 代入後の式を実行
6. 関数適用終了時に、関数適用開始時のステップ番号と関数適用が終わった旨を出力

という順に変更することで、関数適用が終わるステップの直後に関数適用が始まったステップの番号を出力する。それを使ってスキップ機能を実現した。

5 標準出力

OCaml では、簡単な標準出力関数が用意されていて、「プログラミングの基礎 [3]」でも扱われている。本研究では標準出力された文字列をステップ出力の中に挿入する実装を行なった。

1 ステップの実行、または前節の 1 つの関数適用の実行で、標準出力内容がどのように変化したかをステップ実行の一部として見られるように、各ステップの式の下に、それまでに標準出力された文字列を表示させる機能を追加した。

対象とする関数は print_string, print_int, print_float, print_endline, print_newline の 5 つとした。これらの関数の実行では、出力された内容はその後消えたり書き換えられたりすることはなく、また新しく出力をする際にはそれまでの出力文字列の直後に出力されるので、「これまでに出力された文字列」を表す変数をステッパプログラムのグローバル変数として保持しておき、出力の関数を実行するときに文字列の末尾に新しく出力される文字列を追加し、毎ステップの式の出力に続けてその時点での文字列を出力することで実装することができた。

6 まとめと今後の課題

OCaml ステッパをより多くの重要な構文に対応させ、また冗長な計算内容の表示を飛ばす機能や標準出力欄を追加したことで、ステッパの利便性と信頼性を高めた。

今後は、モジュールの定義など他の構文への対応と、より理解しやすいステップ表示を目指していきたい。

参考文献

[1] Y. Cong and K. Asai. Implementing a stepper using delimited continuations. *EPiC Series in Computing*, Vol. 39, pp. 42–54, 2016.

[2] O. Kiselyov, C.-c. Shan, and A. Sabry. Delimited dynamic binding. *ACM SIGPLAN Notices - Proceedings of the 2006 ICFP conference*, Vol. 41, No. 9, pp. 26–37, 2006.

[3] 浅井健一. プログラミングの基礎. サイエンス社, 2007.