Incremental な OCaml ステッパの開発

古川 つきの, 浅井 健一

お茶の水女子大学

furukawa.tsukino@is.ocha.ac.jp, asai@is.ocha.ac.jp

概 要　ステッパはプログラムの実行過程を見せるツールである。ステッパはデバッグに利用できるほか、複雑なコマンド操作などを必要としないので、プログラミング初学者がプログラムの動作を理解する助けになりうる。しかし先行研究のステッパでは、実行時間が大きいプログラムを入力すると長い間処理が終わらず表示や操作ができないという問題があった。本研究では 1 度のステップ実行で 1 つの簡約の内容を出力する incremental なステッパを開発し、それに伴って「簡約されたプログラム」から元のプログラムを復元するための仕組みを提案し実装することによって、実行に時間がかかるプログラムのデバッグや動作の確認を可能にする。

1　はじめに

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、ステッパを使うことだと考える。ステッパは 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 [1] である。ユーザがエディタにプログラムを書いてステッパ起動ボタンを押すと、図 1 のようなウインドウが表示される。図 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パを起動

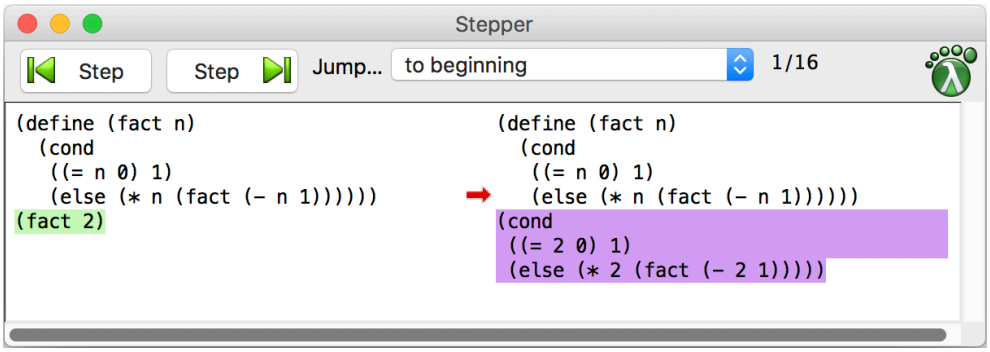

図 1. DrRacket のステッパ

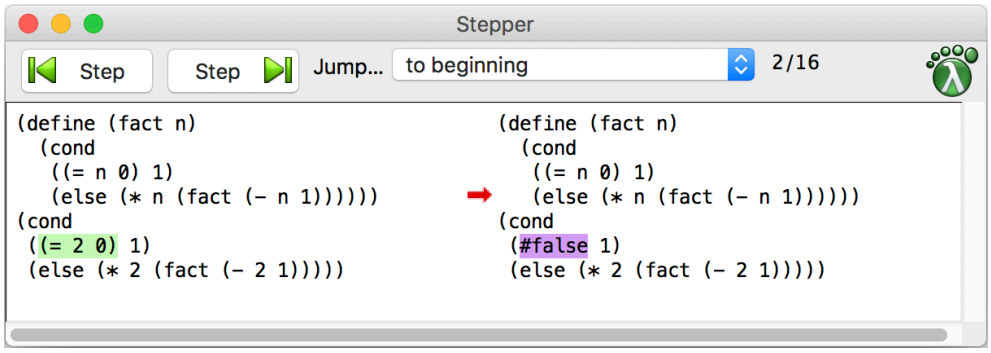

図 **2.** DrRacket のステッパを進めた様子

したときの様子である。ウイ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 ボタンのうち右の実行を進めるボタンを押すと図 2 のような表示に切り替わる。最初（図 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 `(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、例外処理などがサポートされていない。初心者にとって理解しにくい例外処理をステップ実行できるようにするため、著者らは関数型言語 OCaml の、例外処理を含む基礎的な構文に対応したステッパを実装し評価した [2]。簡約される式のハイライトなどの表示の仕方は、概ね DrRacket のステッパ [1] と同様にした。

そのステッパを実際に OCaml の初学者に利用してもらった [2] ところ、ステッパが再帰関数などの理解に役立ったことが分かったが、「プログラムの実行が終わってからインタフェースの動作が始まるので、実行に時間がかかるプログラムを実行しようとすると最初のステップが長時間表示されず利用できない」という問題があった。

本研究では、1 度のステッパプログラムの実行で 1 ステップを実行するように変更し、プログラム全体の実行時間にかかわらずステップ実行ができるステッパを構築する。それによって実行に時間のかかる例外処理を含むプログラムのステップ実行ができるほか、ステッパをクライアントサーバ方式で実装することが可能になる。

2　継続渡し形式のインタプリタ

この節では、ステッパの対象言語とそのインタプリタを定義する。

```
(* 値 *)
type v = Var of string                (* x *)
       | Fun of string * e            (* fun x -> e *)
       | Cont of string * (k -> k)    (* 継続 fun x => ... *)
(* ハンドラ *)
and h = {
  return : string * e;                          (* handler {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 *)
(* 継続 *)
and k = v -> a
(* 実行結果 *)
and a = Return of v
      | OpCall of string * v * k
```

図 **3.** 対象言語の定義

2.1 algebraic effects

2.2 対象言語の構文

対象言語の OCaml による定義を図 3 に示す。型 `v` に含まれる `Cont` は継続を表すコンストラクタであり、入力プログラムに含まれることはないが、ステップ実行の過程で現れる。`Cont` の第一引数の文字列は関数のように表示する際の仮引数名であり表示の為に用いる。

3 CPS インタプリタ

この節では、型無し λ 計算と algebraic effects から成る言語について、インタプリタ関数の機械的なプログラム変換によってステッパ関数を導出する過程を説明する。図 3 の言語に対する、call-by-value かつ right-to-left のインタプリタを図 4 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 `handler {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。

4 インタプリタの変換

本節では、3 節で定義したインタプリタ (図 4) を変換することで、コンテキストの情報を保持するインタプリタを得る方法を示す。用いるプログラム変換は非関数化と CPS 変換の 2 種類である。

```
(* インタプリタ *)
let rec eval (exp : e) (k : k) : a = match exp with
  | Val (v) -> k v
  | App (e1, e2) ->
    eval e2 (fun v2 ->
        eval e1 (fun v1 -> match v1 with
            | Fun (x, e) ->
              let reduct = subst e [(x, v2)] in
              eval reduct k
            | Cont (x, k') ->
              (k' k) v2
            | _ -> failwith "type error"))
  | Op (name, e) ->
    eval e (fun v -> OpCall (name, v, k))
  | With (h, e) ->
    let a = eval e (fun v -> Return v) in
    apply_handler k h a

(* ハンドラを処理する関数 *)
and apply_handler (cont_last : k) (h : h) (a : a) : a =
  match a with
  | Return v ->
    (match h with {return = (x, e)} ->
        let reduct = subst e [(x, v)] in
        eval reduct cont_last)
  | OpCall (name, v, k') ->
    (match search_op name h with
      | None ->
        OpCall (name, v, (fun v ->
            let a' = k' v in
            apply_handler cont_last h a'))
      | Some (x, k, e) ->
        let new_var = gen_var_name () in
        let cont_value =
          Cont (new_var,
                fun cont_last -> fun v ->
                  let a' = k' v in
                  apply_handler cont_last h a') in
        let reduct = subst e [(x, v); (k, cont_value)] in
        eval reduct cont_last)

(* 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)
```

図 4. 継続渡し形式で書かれたインタプリタ

```
and k = FId
      | FApp2 of e * k
      | FApp1 of v * k
      | FOp of string * k
      | FCall of k * h * k
```

図 5. 非関数化後の継続の型

これらの変換はプログラムの動作を変えないので、変換の結果得られるインタプリタと図 4 のインタプリタは、同じ引数 e に対して同じ値を返す。

4.1 非関数化

まず、図 4 のプログラムを非関数化する。具体的には以下を施す。

1. k 型すなわち v -> a 型の匿名関数全てを、関数内に現れる全ての自由変数を引数に持つコンストラクタに分類し、継続の型 k をそれらのコンストラクタから構成されるヴァリアント型に変更する (図 5 のようになる)。
2. v -> a 型の匿名関数全てを、該当するコンストラクタに置き換える
3. 継続に引数を渡している部分 k v を apply_k k v に置き換え、意味が変わらないように関数 apply_k を定義する

変換後のプログラムを図 6 に示す。

非関数化したことで継続 k がコンストラクタとして表されるようになったので、継続の構造を参照することや、継続を部分的に書き換えることが可能になった。具体的な k の構造の例を示す。図 6 の関数 stepper に入力プログラム ((fun a -> a) (with handler {return x -> x, a(x; k) -> x} handle ((fun b -> b) (a (fun c -> c))) (fun d -> d))) を表す構文木を渡して実行を始めた場合、(a (fun c -> c)) を関数 eval に渡して実行を始める際の継続は FApp2 (Fun ("b", Var "b"), FApp1 (Fun ("d", Var "d"))) である。これは式 (a (fun c -> c)) のコンテキスト ((fun a -> a) (with handler {return x -> x, a(x; k) -> x} handle ((fun b -> b) [.]) (fun d -> d))) のうち、handle の内側に対応している。handle から外側が継続に含まれないのは、関数 eval で with h handle e の e の実行の再帰呼び出し時に初期継続を表す FId を渡しているためである。コンテキスト全体に対応した継続を得るために、この後の変換をさらに施す。

4.2 CPS 変換

図 6 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なかったので、全ての継続を引数に持つようにするため、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が図 7 である。

4.3 非関数化

CPS 変換したことにより新たに現れた a -> a 型の匿名関数を非関数化する。非関数化によって型 k2 の定義は図 8 に、ステッパ関数は図 9 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。**??** 節で示した例について比較する。stepper に入力プログラム ((fun a -> a) (with handler {return x -> x, a(x; k) -> x} handle ((fun b -> b) (a (fun c -> c))) (fun d -> d)))

```
let rec eval (exp : e) (k : k) : a = match exp with
  | Val (v) -> apply_in k v
  | 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
    apply_handler k h a

and apply_in (k : k) (v : v) : a = match k with
  | FId -> Return v
  | 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 (x, k') ->
       apply_in (k' k) v2
     | _ -> failwith "type error"
    )
  | FOp (name, k) -> OpCall (name, v, k)
  | FCall (k_last, h, k') ->
    let a = apply_in k' v in
    apply_handler k_last h a

and apply_handler (k_last : k) (h : h) (a : a) : a = match a with
  | Return v ->
    (match h with {return = (x, e)} ->
       let reduct = subst e [(x, v)] in
       eval reduct k_last)
  | OpCall (name, v, k') ->
    (match search_op name h with
     | None ->
       OpCall (name, v, FCall (k_last, h, k'))
     | Some (x, k, e) ->
       let new_var = gen_var_name () in
       let cont_value =
         Cont (new_var, fun k_last -> FCall (k_last, h, k')) in
       let reduct = subst e [(x, v); (k, cont_value)] in
       eval reduct k_last)

let stepper (e : e) : a = eval e FId
```

図 6. 非関数化した CPS インタプリタ

```
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)

and apply_in (k : k) (v : v) (k2 : k2) : a = match k with
  | FId -> 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 (x, k') ->
       apply_in (k' k) v2 k2
     | _ -> failwith "type error")
  | FOp (name, k) -> k2 (OpCall (name, v, k))
  | FCall (k_last, h, k') ->
    apply_in k' v (fun a -> apply_handler k_last h a k2)

and apply_handler (k_last : 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_last k2)
  | OpCall (name, v, k') ->
    (match search_op name h with
     | None ->
       k2 (OpCall (name, v, FCall (k_last, h, k')))
     | Some (x, k, e) ->
       let new_var = gen_var_name () in
       let cont_value =
         Cont (new_var, fun k_last -> FCall (k_last, h, k')) in
       let reduct = subst e [(x, v); (k, cont_value)] in
       eval reduct k_last k2)

let stepper (e : e) : a = eval e FId (fun a -> a)
```

図 **7.** CPS 変換した非関数化した CPS インタプリタ

```
type k2 = GId
        | GHandle of k * h * k2
```

図 **8.** 2 回目の非関数化後の継続の型

```
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 (k, h, k2))

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 (x, k') ->
       apply_in (k' k) v2 k2
     | _ -> failwith "type error")
  | FOp (name, k) -> apply_out k2 (OpCall (name, v, k))
  | FCall (k_last, h, k') ->
    apply_in k' v (GHandle (k_last, h, k2))

and apply_out (k2 : k2) (a : a) : a = match k2 with
  | GId -> a
  | GHandle (k, h, k2) -> apply_handler k h a k2

and apply_handler (k_last : 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_last k2)
  | OpCall (name, v, k') ->
    (match search_op name h with
     | None ->
       apply_out k2 (OpCall (name, v, FCall (k_last, h, k')))
     | Some (x, k, e) ->
       let new_var = gen_var_name () in
       let cont_value =
         Cont (new_var, fun k_last -> FCall (k_last, h, k')) in
       let reduct = subst e [(x, v); (k, cont_value)] in
       eval reduct k_last k2)

let stepper (e : e) : a = eval e FId GId
```

図 **9.** 非関数化して CPS 変換して非関数化した CPS インタプリタ

を表す構文木を渡して実行を始めた場合、(a (fun c -> c)) を表す構文木を関数 eval に渡して実行を始める際の継続 k は ?? 節と同様に FApp2 (Fun ("b", Var "b"), FApp1 (Fun ("d", Var "d"))) である。そして継続 k2 は GHandle () 具体的な k の構造の例を示す。図 6 の関数 stepper に入力プログラム ((fun a -> a) (with handler {return x -> x, a(x; k) -> x} handle ((fun b -> b) (a (fun c -> c))) (fun d -> d))) を表す構文木を渡して実行を始めた場合、(a (fun c -> c)) を関数 eval に渡して実行を始める際の継続は FApp2 (Fun ("b", Var "b"), FApp1 (Fun ("d", Var "d"))) である。これは式 (a (fun c -> c)) のコンテキスト ((fun a -> a) (with handler {return x -> x, a(x; k) -> x} handle ((fun b -> b) [.]) (fun d -> d))) のうち、handle の内側に対応している。handle から外側が継続に含まれないのは、関数 eval で with h handle e の e の実行の再帰呼び出し時に初期継続を表す FId を渡しているためである。コンテキスト全体に対応した継続を得るために、この後の変換をさらに施す。

参考文献

[1] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *European symposium on programming*, pp. 320–334. Springer, 2001.

[2] Tsukino Furukawa, Youyou Cong, and Kenichi Asai. Stepping OCaml. In *Proceedings Sixth Workshop on Trends in Functional Programming in Education*, submitted for publication. 査読用に http://pllab.is.ocha.ac.jp/~asai/tmp/paper.pdf から取得可能.