EPiC Series in Computing
Volume 39, 2016, Pages 42–54
SCSS 2016. 7th International Symposium on Symbolic Computation in Software Science

Implementing a stepper using delimited continuations

Youyou Cong and Kenichi Asai

Ochanomizu University, Tokyo, Japan
{so.yuyu,asai}@is.ocha.ac.jp

Abstract

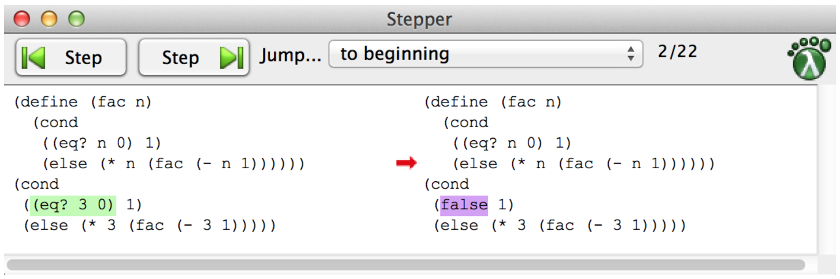
A stepper is a tool that displays all the steps of a program's execution. To implement a stepper, we need to reconstruct each intermediate program from the current redex and the evaluation context. We regard evaluation contexts as delimited continuations and capture them using the control operators shift and reset. This enables us to implement a stepper concisely by writing an evaluator that is close to a standard big-step interpreter. Our implementation is a non-trivial application of shift and reset.

1 Introduction

A stepper is a tool that displays all the intermediate programs that appear during the execution of a given program. It is useful for both educational and debugging purposes. Displaying steps helps the user understand how a program is computed and what is wrong with the program, while saving the effort to write down the steps by hand. Although the widely used debugging facility gdb (Stallman et al. [9]) also allows stepping execution, it only shows part of the program that is being executed, rather than the whole program at the current step. Compared to gdb, a stepper is more user-friendly, and thus is considered to be suited to novice programmers.

DrRacket (Findler et al. [5]; originally called DrScheme), which is a programming environment designed for students among others, provides an algebraic stepper shown in Figure 1. The program on the left-hand side is the current execution state. The expression `(eq? 3 0)` is highlighted in green. This is a redex, i.e., a reducible expression. The highlighting means that this redex is going to be reduced at the current step. On the right-hand side, we find the execution state at the next step. In this program, the redex is replaced by the reduced expression `false`, which is highlighted in purple. Using the buttons on the tool bar, the programmer can step forward and backward through the computation, and thus see every step of reduction.

Since a stepper can be understood as executing a program in small-step style, it is natural to suppose that a stepper can be implemented by writing a small-step interpreter. A small-step interpreter executes a given program by iterating the following three procedures: (i) decompose the program into a redex and an evaluation context; (ii) contract the redex; and (iii) plug the result into the context (Danvy and Nielsen [3]). What a stepper displays are the intermediate programs represented as e_i's in Figure 2. To construct them, we need the redex that is being reduced (the r_i's), and the context surrounding it (the C_i's). The former can be found by recursively calling the interpreter. The latter can be easily obtained in a small-step interpreter, because the context is explicitly passed around as an additional argument.



```
(define (fac n)                          (define (fac n)
  (cond                                    (cond
   ((eq? n 0) 1)                            ((eq? n 0) 1)
   (else (* n (fac (- n 1))))))             (else (* n (fac (- n 1))))))
(cond                                    (cond
 ((eq? 3 0) 1)                            (false 1)
 (else (* 3 (fac (- 3 1)))))              (else (* 3 (fac (- 3 1)))))
```

Figure 1: Stepping a program in DrRacket.

$$\begin{tikzcd}
e_0 \arrow[dddd, dashed, "\text{reduce}"'] \arrow[dr, "\text{decompose}"] & \\
& C_0[r_0] \arrow[d, "\text{contract}"] \\
& C_0[r'_0] \arrow[ddl, "\text{plug}"] \\
& \\
e_1 \arrow[dddd, dashed, "\text{reduce}"'] \arrow[dr, "\text{decompose}"] & \\
& C_1[r_1] \arrow[d, "\text{contract}"] \\
& C_1[r'_1] \arrow[ddl, "\text{plug}"] \\
& \\
e_2 \arrow[d, dash, dashed] \arrow[dr, "\text{decompose}"] & \\
\vdots & \vdots
\end{tikzcd}$$

Figure 2: Executing a program in a small-step interpreter.

In a big-step interpreter, however, the context surrounding a redex is not available as an argument but is embedded implicitly in the control structure of the interpreter. Thus, it is not immediately clear how to obtain such a context. To reconstruct intermediate programs in a big-step interpreter, we make two observations. Firstly, we can use the control operators `shift` and `reset` (Danvy & Filinski [2]) to capture the control structure of the interpreter. Secondly, there is a one-to-one correspondence between the control structure of the interpreter and the evaluation context. With these observations, we can construct a stepper by first capturing the current continuation and then using it both to reconstruct the intermediate program and to continue execution.

A big-step stepper would shed a new light to the stepper that is otherwise considered as a concept based solely on small-step semantics. The correctness proof of the stepper shown in this paper naturally follows the structure of the big-step semantics. It also enables us to use an efficient big-step interpreter during stepping interpretation, to skip a part of steps that the user is not interested in, for example.

This paper presents an implementation of a stepper using `shift` and `reset`. Our contributions are as follows:

- We formulate an implementation of a stepper based on a big-step semantics and using delimited continuations.
- Based on the formulation, we implement a stepper that supports a subset of OCaml (Leroy [7]) including recursive functions. The subset is the minimal one for teaching novices the basic concept of recursion.
- We thus give a non-trivial application of `shift` and `reset`.

The rest of the paper is organized as follows. Section 2 introduces continuations and the control operators. Section 3 presents a stepper for the lambda calculus, showing how intermediate programs are obtained using the control operators. It also provides the correctness proof of the stepper. In Section 4, we extend the stepper with more expressions, and demonstrate how our stepper works with an example. We review previous work in Section 5, and conclude in Section 6.

2 Continuations and control operators

Continuations are a concept from the theory of programming languages, which represents the rest of the computation. For example, if the expression $2 * 3$ in the computation $1 + (2 * 3) - 4$ is being evaluated, the current continuation is the computation where $2 * 3$ is replaced with a hole, that is, the computation "given the value of $2 * 3$, add 1 to it and subtract 4 from the result. Put differently, this is the function $\lambda x. (1 + x - 4)$.

Delimited continuations are continuations with a limited extent. Among various control operators for handling delimited continuations, we use the operators `shift` and `reset`. The `shift` operator captures the continuation surrounding itself, and the `reset` operator delimits the continuation captured by the `shift` operator.

OchaCaml (Masuko and Asai [8]) is an extension of Caml Light (Leroy [6]) with `shift` and `reset`. The program below is written in the OchaCaml syntax.

```
# 1 + reset (fun () -> shift (fun k -> k (k (2 * 3))) - 4) ;;
- : int = -1
```

The expression `shift (fun k -> M)` removes the current continuation delimited by the closest `reset` operator, binds the variable `k` to the continuation, and computes `M`. In the program above, `k` is the function `fun x -> (reset (fun () -> x - 4))`. The computation of adding 1 is not included because it is outside the `reset` clause. The body of the `shift` clause, namely `k (k (2 * 3))`, evaluates to -2, then the 1 outside the `reset` clause is added, yielding -1.

3 A stepper for the lambda calculus

In this section, we show an implementation of a stepper for the lambda calculus. Here we adopt the call-by-value, left-to-right evaluation strategy. The syntax, evaluation contexts, and the reduction rule of the object language are defined in Figure 3.

$$\begin{array}{rcl} e & ::= & v \mid e_1\ e_2 \\ v & ::= & x \mid \lambda x.\, e \\ E[\,] & ::= & [\,] \mid E[[\,]\ e] \mid E[v\ [\,]] \\ E[(\lambda x.\, e)\, v] & \mapsto & E[e[v/x]] \end{array}$$

Figure 3: Syntax, evaluation contexts, and reduction rule.

To see what is essential in a stepper, let us first think about what a standard big-step interpreter does. Given a program, an interpreter traverses it in a certain order. When a redex is found, it reduces the redex to a value, and continues traversing the tree to search for the next redex. The latter task is equivalent to plugging the value into the context surrounding the redex. Then, what about a stepper? A stepper traverses the given program, just like an ordinary interpreter does. When a redex is found, it first reconstructs the current execution state, and makes it visible in one way or another. Then, it resumes ordinary reduction, that is, it reduces the redex and continues the search for a new redex. Note that reconstructing the program corresponds to plugging the redex r_i into the evaluation context C_i in Figure 2. We then find that a stepper requires two kinds of context: one that yields a value and the other one that yields an expression. The former corresponds to the rest of the computation in a standard big-step interpreter, whose hole will be filled with a value. The latter corresponds to the evaluation context in a small-step interpreter, whose hole will be filled with a redex, which is a non-value. What is needed to implement a big-step stepper is a context that behaves in two different ways depending on the plugged expression: If the plugged expression is a non-value, it reconstructs the program, and otherwise it continues the traversal of the program. Such a context can be formed by making a few modifications to a standard big-step interpreter (shown in Section 3.2).

After modifying the interpreter, we capture the context using the `shift` and `reset` operators. These operators also allow us to simulate a state, which is needed for making the intermediate programs visible to the user. Here we maintain a "trace list", in which we store the intermediate programs reconstructed so far. We update the trace list at each reduction step by adding the current program. When reduction is finished, we return the trace list containing all the intermediate programs that appear during the execution of the input program. Using the technique of Filinski [4], we simulate the state with `shift` and `reset`. Normally, a state can be represented using a mutable cell. However, a mutable cell alone is not sufficient for obtaining the trace list, as we spell out in Section 3.3.

In the following subsections, we illustrate three components of our stepper:

- `go`: a function that starts the step-by-step execution
- `stepper`: an interpreter that executes the program step by step
- `memo`: a function that records the current execution step

3.1 Starting the execution: `go`

The function `go` starts the step-by-step execution of a given program. It calls the stepping interpreter `stepper` and stores the result in the state (the trace list) maintained outside the current delimited context. Since `stepper` returns an intermediate program every time it reduces a redex, we can obtain the entire trace list by executing `stepper` as many times as the number of steps required for reducing the program. The following program is the OchaCaml implementation of the function go.

```
let go e = reset (fun () ->
                  let res = stepper e in
                  fun lst -> res :: lst)  (* the context is higher-order *)
                 []  (* initial trace list *)
```

The reset operator delimits the context surrounding stepper into the form "add the result res returned by stepper to the given trace list lst". Notice that the delimited context is higher-order. This enables us to maintain the trace list. As we will see in Section 3.3, the computation stepper e calls the shift operator to capture the evaluation contexts around the redex to be reduced. When shift is called in this context, the captured computation will be a function that first receives an expression (the current redex), which is going to be plugged into the hole. Furthermore, the captured computation receives an additional list, namely the trace list, because the context is higher-order. The trace list is initially an empty list (the one being passed outside the reset clause in go), and will be extended each time stepper returns an intermediate program.

3.2 Stepping the program: stepper

The stepping interpreter stepper is structurally similar to a typical big-step interpreter. Their difference is that stepper has the following additional behaviors:

- When a redex is found, add the current program to the trace list.
- When a non-value expression is returned, reconstruct the current evaluation context.

The program below is the actual implementation of stepper.

```
let rec stepper e = match e with
  Value (Var (x)) -> e
| Value (Lam (x, body)) -> e
| App (e1, e2) ->
    let a1 = stepper e1 in
    begin match a1 with
      Value (Lam (x1, body1)) ->
        let a2 = stepper e2 in
        begin match a2 with
          Value (Lam (x2, body2)) ->
            memo (App (a1, a2)) (fun () ->  (* record the current program *)
            stepper (subst body1 x1 a2))
        | _ -> App (a1, a2)  (* reconstruction *)
        end
    | _ -> App (a1, e2)  (* reconstruction *)
    end
```

We assume the input expression to be a closed term. If the given expression is a value, it is simply returned. Otherwise stepper searches for the left-most redex, following the evaluation contexts defined in Figure 3. When both the function and the argument have been evaluated to a lambda abstraction, it calls the function memo, instead of directly applying the reduction rule. The function memo (to be described in Section 3.3) receives the current redex as its first argument. The redex is used to reconstruct the current program. More specifically, it is plugged into the current evaluation context, and the reconstructed program is added to the trace list. Then the second argument, which is the standard computation for reducing the redex, is evaluated. The reduced expression is plugged into the same evaluation context, and then `stepper` searches for a new redex. What enables the reconstruction of the entire program, and what is missing in a standard big-step interpreter, is the code in the cases where a non-value expression (redex) is returned by `stepper`. For example, if the result `a2` of evaluating the argument `e2` is not a value, it means that a redex is returned for reconstruction of the current program. In that case, we want to reconstruct the application of the function `a1`, namely `App (a1, a2)`. Similarly, if the function `e1` is evaluated to a non-value expression `a1`, we want to reconstruct `App (a1, e2)`. The last two cases in the match expressions in `stepper` allow us to use the context for reconstructing the program in the function `memo`, which is detailed in the next subsection.

The program in the previous page is a substitution-based interpreter. One could also write an environment-based stepper as follows, where `lookup` and `extend` are functions for looking up the value associated with a variable and for extending an environment with a variable-value pair:

```
let rec stepper e env = match e with
  Var (x) -> lookup x env
| Lam (x, body) -> Value (Closure (x, body, env))
| App (e1, e2) ->
    let a1 = stepper e1 env in
    begin match a1 with
      Value (Closure (x1, body1, env1)) ->
        let a2 = stepper e2 env in
        begin match a2 with
          Value (Closure (x2, body2, env2)) ->
            memo (App (a1, a2)) (fun () ->  (* record the current program *)
            stepper body1 (extend env1 x1 a2))
        | _ -> App (a1, a2)  (* reconstruction *)
        end
    | _ -> App (a1, e2)  (* reconstruction *)
    end
```

The reader might think that this stepper is closer to an ordinary big-step interpreter. While the substitution-based stepper traverses the function body to substitute a value each time reduction takes place, the environment-based one only requires on-demand environment lookups, and thus allows faster evaluation. However, we need substitution for free variables in closures when displaying the steps. Here the choice between the two styles is not important; we continue using the substitution-based one.

3.3 Recording steps: `memo`

The function `memo` is used to record each intermediate program. See the definition below.

```
let memo e f = shift (fun k -> fun lst ->
                 (reset (fun () -> k (f ()))) (k e lst))
```

The function `memo` receives two arguments: the current redex `e`, and a thunk `f` that continues the execution. The redex `e` is used to reconstruct the entire program at the current step. To reconstruct the program, we capture the context surrounding `e` using the `shift` operator. Notice that the captured continuation `k` is used twice. The first use (in the call-by-value setting), `k e lst`, is for reconstruction. As we will prove later, when a non-value `e` is passed to `k`, `stepper` correctly reconstructs the current program. Since `k` accepts an additional parameter for the trace list, the application `k e lst` as a whole will reconstruct the current program and add it to the trace list. The resulting trace list is the new state, which is passed to the delimited context (the `reset` clause) for subsequent traces. After updating the trace list, the ordinary reduction is resumed by the computation `f ()`. Having obtained its result, we make the second call to `k` for continuing evaluation.

Note that in the `shift/reset`-based implementation of ordinary state monad, the captured continuation is used only for continuing the computation. In our stepper, the captured continuation is further used for updating the state, because we would like to store intermediate programs, which are reconstructed using the continuation. Thus, even if we used a mutable cell to implement a state, the use of `shift` and `reset` is necessary for the latter purpose.

3.4 Example

Now let us illustrate how our stepper evaluates the expression $((\lambda x.\, x)(\lambda y.\, y))((\lambda z.\, z)(\lambda w.\, w))$. Following the evaluation contexts, `stepper` first evaluates $(\lambda x.\, x)(\lambda y.\, y)$. Since both the function and the argument of this application have the form of lambda abstraction, i.e., a redex is found, the function `memo` is called. Here, the first argument of `memo` is the current redex $(\lambda x.\, x)(\lambda y.\, y)$, and the second argument is the computation for reducing the redex and continuing the rest of the computation.

```
memo (App (λx.x, λy.y)) (fun () -> stepper (subst x x λy.y))
```

The continuation captured by the `shift` operator in `memo` is the following computation, where [ ] represents the hole:

```
reset (fun () ->
        let res =
          let a1 = [ ] in
          begin match a1 with
            Value (Lam (x1, body1)) -> ...
          | _ -> App (a1, (λz.z)(λw.w))
          end
        in fun lst -> res :: lst)
```

We find a hole in the position of `stepper f` in the definition of `stepper`, because the `shift` operator is called while evaluating the first argument of the outermost `App` constructor. By passing the current redex $(\lambda x.\, x)(\lambda y.\, y)$ to `memo`, the hole is filled with that redex. Since the plugged redex $(\lambda x.\, x)(\lambda y.\, y)$ is not yet reduced and thus is not a value, the program $((\lambda x.\, x)(\lambda y.\, y))((\lambda z.\, z)(\lambda w.\, w))$ is reconstructed in the sixth line. The variable `lst` in this case is the empty list passed in the function `go`. Therefore, the trace list is updated to the list $[((\lambda x.\, x)(\lambda y.\, y))((\lambda z.\, z)(\lambda w.\, w))]$. After updating the state, the second argument of `memo` is evaluated, and the result $\lambda y.\, y$ is filled in the hole, resuming the standard evaluation. Since $\lambda y.\, y$ is already a lambda abstraction, `stepper` evaluates the argument expression of the outermost application, namely $(\lambda z.\, z)(\lambda w.\, w)$. This is a redex, so we call `memo` again.

```
memo (App (λz.z, λw.w)) (fun () -> stepper (subst z z λw.w))
```

The `shift` operator will capture the continuation

```
reset (fun () ->
        let res =
          let a2 = [ ] in
          begin match a2 with
            Value (Lam (x2, body2)) -> ...
          | _ -> App (λy.y, a2)
          end
        in fun lst -> res :: lst).
```

This time, `memo` is called while computing the argument expression, so we find a hole in the position of `stepper arg`. The hole is filled with the current redex $(\lambda z.\, z)(\lambda w.\, w)$, and the entire program $(\lambda y.\, y)((\lambda z.\, z)(\lambda w.\, w))$ is reconstructed in the sixth line. The trace list passed to the function `fun lst -> ...` is the list updated at the previous step. By adding the program reconstructed at the current step, we obtain $[(\lambda y.\, y)((\lambda z.\, z)(\lambda w.\, w));\ ((\lambda x.\, x)(\lambda y.\, y))((\lambda z.\, z)(\lambda w.\, w))]$. After evaluating the second argument of `memo`, the current redex will be reduced to $\lambda w.\, w$. By plugging this contractum into the hole, we obtain a new redex: $(\lambda y.\, y)(\lambda w.\, w)$. We make the third call to `memo`, which captures the following continuation:

```
reset (fun () ->
        let res = [ ] in
        fun lst -> res :: lst)
```

As before, we fill the hole with the current redex $(\lambda y.\, y)(\lambda w.\, w)$. Since the evaluation context is empty, i.e., no reconstruction is needed, the redex is simply added to the trace list, yielding $[(\lambda y.\, y)(\lambda w.\, w);\ (\lambda y.\, y)((\lambda z.\, z)(\lambda w.\, w));\ ((\lambda x.\, x)(\lambda y.\, y))((\lambda z.\, z)(\lambda w.\, w))]$. Having reduced the redex to $\lambda w.\, w$, we add it to the trace list as the final result. Thus the computation

`go` $((\lambda x.\, x)(\lambda y.\, y))((\lambda z.\, z)(\lambda w.\, w))$

evaluates to the list

$[\lambda w.\, w;\ (\lambda y.\, y)(\lambda w.\, w);\ (\lambda y.\, y)((\lambda z.\, z)(\lambda w.\, w));\ ((\lambda x.\, x)(\lambda y.\, y))((\lambda z.\, z)(\lambda w.\, w))]$,

which is a list having all the reduction steps of the input program in the reversed order.

3.5 Correctness of the stepper

Here we show that our stepper works in the intended manner: it produces a list containing all the intermediate programs in the correct order. The first step is to show the correctness of reconstruction. We define three contexts as follows (the variable after the subscripts represents the free variable contained in the context):

- $C_0[\ ]$: an empty context
- $C_{1,\mathtt{e2}}[\ ]$:

```
let a1 = [ ] in
begin match a1 with
  Value (Lam (x1, body1)) ->
    let a2 = stepper e2 in
    begin match a2 with
      Value (Lam (x2, body2)) ->
        memo (App (a1, a2)) (fun () ->
        stepper (subst body1 x1 a2))
    | _ -> App (a1, a2)
```

```
          end
      | _ -> App (a1, e2)
      end
```

- $C_{2,\mathtt{a1}}[\,]$:
```
let a2 = [ ] in
begin match a2 with
  Value (Lam (x2, body2)) ->
    memo (App (a1, a2)) (fun () ->
    stepper (subst body1 x1 a2))
| _ -> App (a1, a2)
end
```

Using these contexts, we define the context $C[\,]$, which represents the control structure of the stepping interpreter.

$$C[\,] \quad ::= \quad C_0[\,] \mid C[C_{1,\mathtt{e2}}[\,]] \mid C[C_{2,\mathtt{a1}}[\,]]$$

Next, we define a mapping from $C[\,]$ to the evaluation context $E[\,]$ defined in Figure 3, repeated below:

$$\begin{array}{rcl} E[\,] & ::= & [\,] \mid E[[\,]\ e] \mid E[v\ [\,]] \\ \widetilde{C_0}[\,] & = & [\,] \\ \widetilde{C[C_{1,\mathtt{e2}}[\,]]} & = & \widetilde{C}[[\,]\ \mathtt{e2}] \\ \widetilde{C[C_{2,\mathtt{a1}}[\,]]} & = & \widetilde{C}[\mathtt{a1}\ [\,]] \end{array}$$

Now we prove that plugging a non-value expression `e` into the context $C[\,]$ yields the same expression as plugging `e` into the corresponding evaluation context $\widetilde{C}[\mathtt{e}]$.

Proposition 1. *For any non-value* `e`*,* $C[\mathtt{e}] = \widetilde{C}[\mathtt{e}]$*.*

Proof. The proof is by induction on the structure of the context $C[\,]$. If $C[\,]$ is an empty context, both $C[\mathtt{e}]$ and $\widetilde{C}[\mathtt{e}]$ are trivially equal to `e`. Now suppose $C[\,]$ can be decomposed into $C'[C_{1,\mathtt{e2}}[\,]]$. Our goal is to show $C'[C_{1,\mathtt{e2}}[\mathtt{e}]] = \widetilde{C'[C_{1,\mathtt{e2}}[\mathtt{e}]]}$. On the left-hand side, the context $C_{1,\mathtt{e2}}[\,]$ contains a `shift` operator that would be called in the function `memo`. However, since `e` is a non-value, the `shift` operator will not be executed after plugging `e` into $C_{1,\mathtt{e2}}[\,]$. Rather, plugging `e` into the context yields `e e2`. Therefore, $C'[C_{1,\mathtt{e2}}[\mathtt{e}]]$ is equal to $C'[\mathtt{e\ e2}]$. The right-hand side, $\widetilde{C'[C_{1,\mathtt{e2}}[\mathtt{e}]]}$, is equal to $\widetilde{C'}[\mathtt{e\ e2}]$ by the definition of the mapping $\sim$. By the induction hypothesis, we further have $C'[\mathtt{e\ e2}] = \widetilde{C'}[\mathtt{e\ e2}]$. Hence the proposition holds. The remaining case can be shown in a similar way. $\square$

Using this proposition, we prove the following theorem.

Theorem 1. *Let* $C_I[\,]$ *be the initial context* `let res = [ ] in fun lst -> res :: lst`*, and* `l` *be an arbitrary trace list. For any value* `v`, `reset (fun () ->` $C_I[$`stepper v`$]$`) l` $=$ `v :: l`*. For any non-value* `e` *and context* $C[\,]$, `reset (fun () ->` $C_I[C[$`stepper e`$]]$`) l` $=$ `reset (fun () ->` $C_I[C[$`stepper e'`$]]$`)` `(`$\widetilde{C}[\mathtt{e}]$ `:: l)`*, where* `e'` *is the expression obtained by applying one-step reduction to* `e`*.*

Proof. The proof is by induction on the structure of the term. The statement of the value case immediately follows. In the non-value case, `e` should be an application. More precisely, `e` is either `v1 v2` or `v1 e2` or `e1 e2`.

Suppose `e` = `v1 v2`. Since this is a redex, `stepper` calls `memo`:

$$\begin{aligned} & \texttt{reset (fun () -> } C_I[C[\texttt{stepper e}]]\texttt{) l} \\ = \; & \texttt{reset (fun () -> } C_I[C[\texttt{memo e (fun () -> stepper e}'\texttt{)}]]\texttt{) l} \end{aligned}$$

The `shift` operator in `memo` captures the continuation $C_I[C[\,]]$, and the program reduces in the following way:

$$\begin{aligned} & \texttt{reset (fun () -> } C_I[C[\texttt{stepper e}']]\texttt{) (}C_I[C[\texttt{e}]]\texttt{ l)} \\ = \; & \texttt{reset (fun () -> } C_I[C[\texttt{stepper e}']]\texttt{) (}C[\texttt{e}]\texttt{ :: l)} \end{aligned}$$

By Proposition 1, the program above is equivalent to

$$\texttt{reset (fun () -> } C_I[C[\texttt{stepper e}']]\texttt{) (}\widetilde{C}[\texttt{e}]\texttt{ :: l),}$$

which proves the theorem for this case.

Now suppose `e` = `v1 e2`, and `e2` reduces to `e2`′ in one step. Since `e2` is a non-value, we evaluate `e2` in the context $C_I[C[C_{2,\texttt{v1}}[\,]]]$.

$$\begin{aligned} & \texttt{reset (fun () -> } C_I[C[\texttt{stepper e}]]\texttt{) l} \\ = \; & \texttt{reset (fun () -> } C_I[C[C_{2,\texttt{v1}}[\texttt{stepper e2}]]]\texttt{) l} \end{aligned}$$

By the induction hypothesis, we have

$$\begin{aligned} & \texttt{reset (fun () -> } C_I[C[C_{2,\texttt{v1}}[\texttt{stepper e2}]]]\texttt{) l} \\ = \; & \texttt{reset (fun () -> } C_I[C[C_{2,\texttt{v1}}[\texttt{stepper e2}']]]\texttt{) (}\widetilde{C[C_{2,\texttt{v1}}}[\texttt{e2}]]\texttt{ :: l)} \end{aligned}$$

By the definition of the context $C[\,]$ and the mapping $\sim$, this is equivalent to

$$\texttt{reset (fun () -> } C_I[C[\texttt{stepper (v1 e2}'\texttt{)}]]\texttt{) (}\widetilde{C}[\texttt{v1 e2}]\texttt{ :: l).}$$

Therefore the theorem holds. The `e1 e2` case is similar. □

Using Theorem 1, we can show that `go e0` produces a list of all the intermediate programs of `e0` in reverse order.

Theorem 2. *For any program* `e0` *whose reduction sequence is* `e0` ↦ `e1` ↦ ... ↦ `en` *where* `en` *is a value,* `go e0` *returns the list* `[en; ...; e1; e0]`.

Proof. By induction on the length of reduction steps. If `e0` is a value, the theorem immediately follows from the value case of Theorem 1. If `e0` reduces to `e1`, we have

$$\texttt{e0} = \widetilde{C}[\texttt{r}] \mapsto \widetilde{C}[\texttt{r}'] = \texttt{e1}$$

for some $C[\,]$, where `r` is the first redex to be reduced and `r` ↦ `r`′. In this case, `go e0` reduces to

$$\texttt{reset (fun () -> } C_I[C[\texttt{stepper r}]]\texttt{) [].}$$

By Theorem 1, this program is equal to

$$\begin{aligned} & \texttt{reset (fun () -> } C_I[C[\texttt{stepper r}']]\texttt{) [}\widetilde{C}[\texttt{r}]\texttt{]} \\ = \; & \texttt{reset (fun () -> } C_I[\texttt{stepper e1}]\texttt{) [e0].} \end{aligned}$$

The program above is equivalent to appending the list generated by `go e1` to the list `[e0]`, i.e., `(go e1) @ [e0]` (where `@` is a primitive operator for appending two lists). By the induction hypothesis, `go e1` = `[en; ...; e2; e1]`. Therefore, `go e0` = `(go e1) @ [e0]` = `[en; ...; e1; e0]`. □

4 The extended stepper

In this section, we extend the object language with integers, booleans, conditionals, let-bindings, recursive functions, lists, and pattern matchings. These expressions are chosen from the ones that are taught in the first four weeks of the CS1 course in OCaml at Ochanomizu University. In the extended language, one can write recursive functions on integers and lists.

The extended stepper is implemented in the same strategy as the one for the lambda calculus:

- Search for the redex following the evaluation contexts.
- When a redex is found, add the current program to the trace list by calling `memo`.
- If a non-value expression is returned by `stepper`, reconstruct the current context.

Using the extended stepper, a program that computes the factorial of 3 is stepped as shown in Figure 4. One will find that the function definition `let rec fac ...` is included in every step. This phrase is printed in order to keep the function name `fac` in the `else` clause bound by the definition. Reading the list from the bottom to the top, we see the complete reduction sequence of the input program.

One would wish to make the steps easier to read by, for example, highlighting the redex being focused at each step. This can be realized in the following way. First, we add a new constructor `Redex` to terms, and modify `memo` by wrapping the given redex in the `Redex` constructor. Next, we make a function that converts a term into a string while adding an attribute to highlight each redex. Thus we obtain a more readable trace list, as shown in Figure 5.

The extended stepper is available at `https://github.com/YouyouCong/stepper_ochacaml`.

5 Related work

Clements et al. [1] illustrate the implementation of DrRacket's stepper, which is based on a continuation mark mechanism. A continuation mark is a mark attached to the stack. Each continuation mark has a single value conveying the information that would be used in later computation. In a stepper, what we need is the information for reconstructing the whole program at each reduction step. This can be obtained by placing continuation marks whose value represents the current evaluation context. For example, when evaluating 1 in the program (+ 1 2), we attach a mark-value indicating that we are evaluating the first argument of an addition, and that the second argument is 2. With all the traversed contexts marked on the stack, we can correctly reconstruct the entire program when we find a redex.

In DrRacket's stepper, continuation marks are manipulated using two primitives: **with-continuation-mark** and **current-continuation-marks**. The former is a function that attaches a continuation mark to the stack before executing a computation. The latter is a function that retrieves every mark-value on the stack.

When evaluating a DrRacket program using the stepper, the program is elaborated in two steps. First, we insert a breakpoint to every place where reduction takes place. Next, we translate the breakpoint language into a low-level language that manipulates continuation marks. In the low-level language, all the non-value expressions (including breakpoints) are wrapped

```
["let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 6";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * 2)";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * 1))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (1 * 1)))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (1 * (if true then 1 else (0 * (fac (0 - 1)))))))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (1 * (if (0 = 0) then 1 else (0 * (fac (0 - 1)))))))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (1 * (fac 0))))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (1 * (fac (1 - 1)))))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (if false then 1 else (1 * (fac (1 - 1))))))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (if (1 = 0) then 1 else (1 * (fac (1 - 1))))))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (fac 1)))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (fac (2 - 1))))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (if false then 1 else (2 * (fac (2 - 1)))))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (if (2 = 0) then 1 else (2 * (fac (2 - 1)))))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (fac 2))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (fac (3 - 1)))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (if false then 1 else (3 * (fac (3 - 1))))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (if (3 = 0) then 1 else (3 * (fac (3 - 1))))";
 "let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (fac 3)"]
```

Figure 4: Stepping `fac 3`.

```
let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
(if false then 1 else (3 * (fac (3 - 1))));
let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
(if (3 = 0) then 1 else (3 * (fac (3 - 1))));
let rec fac = (fun n -> (if (n = 0) then 1 else (n * (fac (n - 1))))) in
(fac 3)]
```

Figure 5: Highlighting redexes.

in **with-continuation-mark**, so that the information of every traversed evaluation context is attached to the stack. In particular, breakpoints are translated into a computation that produces a stream of the mark-values on the stack using **current-continuation-marks**. The mark-values are then mapped back to the corresponding evaluation contexts, which are used to reconstruct and display the current program.

The basic idea of their implementation has much in common with ours. When a redex is found, they use the breakpoint operation to stop the execution. In our stepper, we call the function `memo` to record the current step. To reconstruct the program, they use the **current-continuation-marks** operation to retrieve the continuation marks on the stack. In our stepper, we capture the evaluation context using the `shift` operator. To have enough information for reconstruction, they place a continuation mark on the stack every time the interpreter enters an evaluation context. In our stepper, we add a non-value case to each pattern matching and write the code for reconstruction there. The difference is that our stepper manipulates continuations using `shift` and `reset`, rather than defining a low-level language.

6 Conclusion and future work

We have implemented a stepper using the control operators `shift` and `reset`. The stepping interpreter reconstructs the current evaluation context in the cases where a non-value expression is returned. When a redex is found, the context is captured by the `shift` operator and is used to reconstruct the current program.

The difference between our stepper and a standard interpreter tells us exactly what is essential in implementing a stepper: use the context for both reconstructing the current program and continuing evaluation. While DrRacket's stepper introduces a continuation mark mechanism for retrieving the context, our stepper captures the context using a single `shift` operator.

In this paper, we only presented a stepper supporting a small subset of OCaml. Extending the object language with expressions such as tuples and records is trivial. However, there are also extensions that are less straightforward. Two such examples are modules and exceptions. We leave these non-trivial extensions to future work.

Acknowledgments: We are grateful to anonymous reviewers and Jacques Garrigue for helpful comments and suggestions. This work was partly supported by the Leading Graduate School Promotion Center, Ochanomizu University and JSPS Grant-in-Aid for Scientific Research (C) 15K00090.

References

[1] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *Programming Languages and Systems*, number 2028 in Lecture Notes in Computer Science, pages 320–334. Springer, 2001.

[2] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160. ACM, 1990.

[3] Olivier Danvy and Lasse R Nielsen. *Refocusing in reduction semantics*. BRICS, Department of Computer Science, Univ., 2004. A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[4] Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 446–457. ACM, 1994.

[5] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of functional programming*, 12(02):159–182, 2002.

[6] Xavier Leroy. The Caml Light system release 0.74. 1997. http://caml.inria.fr.

[7] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique*, 2014. http://caml.inria.fr.

[8] Moe Masuko and Kenichi Asai. Caml Light+ shift/reset= Caml Shift. *Theory and Practice of Delimited Continuations (TPDC 2011)*, pages 33–46, 2011.

[9] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 51:02110–1301, 2002.