Algebraic Stepper for Simple Modules

Kenichi Asai and Hinano Akiyama

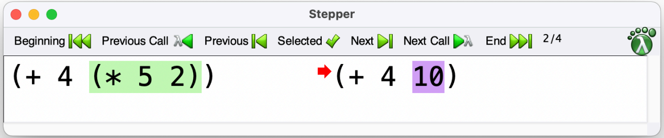
Ochanomizu University, Japan

January 21, 2025

http://pllab.is.ocha.ac.jp/~asai/Stepper/demo/

Algebraic Stepper

A tool to show all the intermediate steps of program execution, like a small-step semantics or algebraic calculation in math.



$$\begin{aligned} & 4 + 5 \times (3 - 1) \\ = \; & 4 + 5 \times 2 \\ = \; & 4 + 10 \\ = \; & 14 \end{aligned}$$

$$\text{fac 5} \rightarrow^* \text{5 * fac 4} \rightarrow^* \text{5 * (4 * fac 3)} \rightarrow^* \text{5 * (4 * (3 * fac 2))} \rightarrow^* \cdots$$

$$(\lambda x.\, \lambda y.\, x + y)\, 3\, 4 \rightarrow (\lambda y.\, 3 + y)\, 4 \rightarrow 3 + 4 \rightarrow 7$$

OCaml Stepper

We implemented a stepper for OCaml and use it in a functional programming course in our university.

- Supports most of the basic constructs of OCaml (including recursion, records, lists, exceptions, output, references).
- Among the topics covered in the course, modules were the only unsupported feature.
- Demo page or in Emacs (VS code support in progress).

Stepper ≠ Small-Step Semantics

Delayed substitution of variables

In a stepper, a variable is substituted to its value using one step when it is used, not when it is declared.

Program:

```
let a = 10
let f x = a + x
let _ = f 100
```

Step execution:

```
  f 100
→ a + 100
→ 10 + 100
→ 110
```

Small-step semantics:

```
let a = 10
let f x = 10 + x
let _ = (\x.10 + x) 100
    (\x.10 + x) 100
→   10 + 100
→   110
```

Variables vs. Functions

We want a variable to be replaced with its value, but not a function.

Program:

```
let a = 10
let f x = a + x
let _ = f 100
```

We want:

```
  f 100
→ a + 100
→ 10 + 100
→ 110
```

But not:

```
  f 100
→ (\x.a + x) 100
→ a + 100
→ 10 + 100
→ 110
```

A constant variable is a redex, a function variable is not.

Variable Annotations

Once declared, variables are annotated with their levels and values.

Users see:

```
let a = 10
let f x = a + x
let _ = f 100

    f 100
  → a + 100
  → 10 + 100
  → 110
```

Internal representation:

```
let a = 10
let f x = a [@0] [@10] + x
let _ = f [@0] [@\x.a[@0][@10] + x] 100

    f [@0] [@\x.a[@0][@10] + x] 100
  → a [@0] [@10] + 100
  → 10 + 100
  → 110
```

Design Choice: Allow Apparent Name Clashes

Users see:

```
let a = 10
let f x = a + x
let a = 20
let _ = f 100

    f 100
→  a + 100
→  10 + 100
→  110
```

Internal representation:

```
let a = 10
let f x = a [@0] [@10] + x
let a = 20
let _ = f [@0] [@\x.a[@0][@10] + x] 100

    f [@0] [@\x.a[@0][@10] + x] 100
→  a [@0] [@10] + 100
→  10 + 100
→  110
```

OCaml Modules

```
let a = 10
let f x = a + x
let _ = f 100

module X = struct
  let a = 20
  let g x = f x + a
  let _ = g 200
end

let _ = X.g 300
```

A program: a tree of static modules

- A module can contain type, variable, and module declarations.
- They are evaluated once in the order of appearance.

Variable reference:

- A variable in the parent module can be accessed directly.
- Access to a variable in a child module requires a module path.

Propagating Values of Variables into Modules

```
let a = 10
let f x = a + x
let _ = f 100

module X = struct
  let a = 20
  let g x = f x + a
  let _ = g 200
end
```

- Values of variables are annotated in the rest of the program.

Propagating Values of Variables into Modules

```
let a = 10
let f x = a [@0] [@10] + x
let _ = f 100

module X = struct
  let a = 20
  let g x = f x + a
  let _ = g 200
end
```

- Values of variables are annotated in the rest of the program.
- When a variables is shadowed, substitution stops.

Propagating Values of Variables into Modules

```
let a = 10
let f x = a [@0] [@10] + x
let _ = f [@0] [@\x. a[@0][@10] + x] 100

module X = struct
  let a = 20
  let g x = f [@1] [@\x. a[@1][@10] + x] x + a
  let _ = g 200
end
```

- Values of functions are also annotated in the rest of the prog.
- Levels increase by 1 when entering a module.

Propagating Values of Variables into Modules

```
let a = 10
let f x = a [@0] [@10] + x
let _ = f [@0] [@\x. a[@0][@10] + x] 100

module X = struct
  let a = 20
  let g x = f [@1] [@\x. a[@1][@10] + x] x + a [@0] [@20]
  let _ = g 200
end
```

- Variables in attributes are not affected.

Propagating Values of Variables into Modules

```
let a = 10
let f x = a [@0] [@10] + x
let _ = f [@0] [@\x. a[@0][@10] + x] 100

module X = struct
  let a = 20
  let g x = f [@1] [@\x. a[@1][@10] + x] x + a [@0] [@20]
  let _ = g [@0] [@\x. f[@1][@...] x + a[@0][@20]] 200
end
```

```
→ f[@1][@\x. a[@1][@10] + x] 200 + a[@0][@20]
→ f[@1][@\x. a[@1][@10] + x] 200 + 20
→ (a[@1][@10] + 200) + 20
→ (10 + 200) + 20
```

Substitution of Modules

```
let a = 10
let f x = a [@0] [@10] + x

module X = struct
  let a = 20
  let g x = f [@1] [@\x. a[@1][@10] + x] x + a [@0] [@20]
end

let _ = X.g 300
```

- When a module is evaluated, its information is propagated to the rest of the program.

Substitution of Modules

```
let a = 10
let f x = a [@0] [@10] + x

module X = struct
  let a = 20
  let g x = f [@1] [@\x. a[@1][@10] + x] x + a [@0] [@20]
end

let _ = X.g [@0] [@\x. f[@0][@\x. a[@0][@10] + x] x
                    + X.a[@0][@20]] 300
```

- Levels decrease by 1.
- When levels are already 0, module path is attached.

Formalization (Stepper and Small-Step Semantics)

Syntax:

module path	p	$::=$	$\epsilon \mid X.p$
value	v	$::=$	$c \mid p.x \mid \lambda z.\, e \mid p.g[@n][@\lambda z.\, e]$
expression	e	$::=$	$v \mid e_0\, e_1 \mid p.x[@n][@c]$
structure item	i	$::=$	let $x = e$ | module $X =$ struct s end
structure	s	$::=$	$[\,] \mid i :: s$

Stepper Only

Reduction Rules (for Expressions)

$$(\lambda z.\, e)\, v \;\rightarrow\; e[v/z]_{\mathrm{e}}$$

$$p.x[@n][@c] \;\rightarrow\; c$$
$$p.g[@n][@\lambda z.\, e]\, v \;\rightarrow\; e[v/z]_{\mathrm{e}}$$

$$\frac{e_1 \rightarrow e_2}{E[e_1] \rightarrow E[e_2]}$$

Stepper Only

```
let a = 10
let f x = a + x
let _ = f 100
```

```
  f 100
→ a + 100
→ 10 + 100
→ 110
```

Evaluation Rules (for Modules)

$$\frac{s_1 \rightsquigarrow s_2}{S[s_1] \rightsquigarrow S[s_2]} \qquad \frac{e_1 \to e_2}{(\mathsf{let}\ x = e_1) :: s \rightsquigarrow (\mathsf{let}\ x = e_2) :: s}$$

$$\begin{array}{llll} & (\mathsf{let}\ x = v) & :: & s \\ \rightsquigarrow & \{\mathsf{let}\ x = v\} & :: & s[x[@0][@v]/x]_{\mathsf{s}} \quad s[v/x]_{\mathsf{s}} \\ & (\mathsf{module}\ X = \mathsf{struct}\ r\ \mathsf{end}) & :: & s \\ \rightsquigarrow & \{\mathsf{module}\ X = \mathsf{struct}\ r\ \mathsf{end}\} & :: & s[\mathsf{lift}_{\mathsf{s}}(X, r)/X]^0_{\mathsf{s}} \quad s[r/X]^0_{\mathsf{s}} \end{array}$$

Stepper Only | Small-Step Semantics Only

Property

Define the erasure $\lfloor e \rfloor$ of e by replacing all the annotated variables with their values, i.e., applying $\lfloor p.x[@n][@c] \rfloor = c$ to all the subexpressions.

Theorem

1. If $e_1 \rightarrow e_2$ in the stepper semantics, $\lfloor e_1 \rfloor \rightarrow^* \lfloor e_2 \rfloor$ in the standard semantics.
2. If $s_1 \rightsquigarrow s_2$ in the stepper semantics, $\lfloor s_1 \rfloor \rightsquigarrow^* \lfloor s_2 \rfloor$ in the standard semantics.

Note: since $\lfloor e \rfloor$ removes variable names, the theorem says nothing about whether the used variable names are correct.

Related Work

Stepper:

- Clements et al. 2001 (Scheme)
- Whitington, Ridge 2017 (OCaml)

OCaml stepper from our group:

- Cong and Asai 2016 (original design)
- Furukawa, Cong, and Asai 2018 (exception)
- Akiyama and Asai 2023 (references)

Modules:

- Many papers on typing for advanced features
- A few small-step semantics, e.g., Crary 2019

Current Status and Summary

- Implemented for OCaml 4.14.2 (last version before OCaml 5).
- Used in a functional programming course in our university.

Stepper $\neq$ small-step semantics

... because of the delayed substitution of variables

Future work:

- Algebraic effects for OCaml 5?
- Functors?
- Signature sealing...? — not likely.