Collapsing Heterogeneous Towers of Interpreters

Michael Buch – mb2244

University of Cambridge

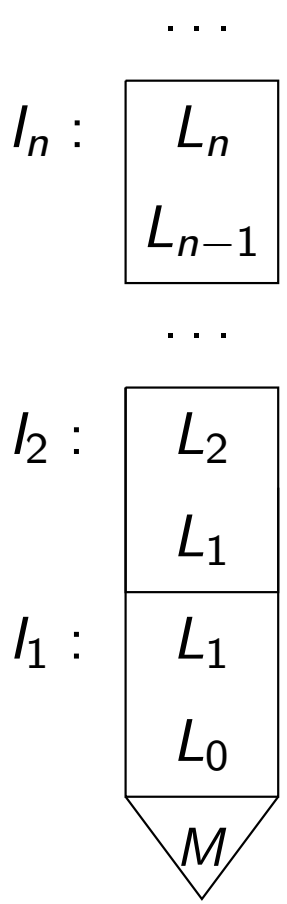
June 20, 2019

Overview

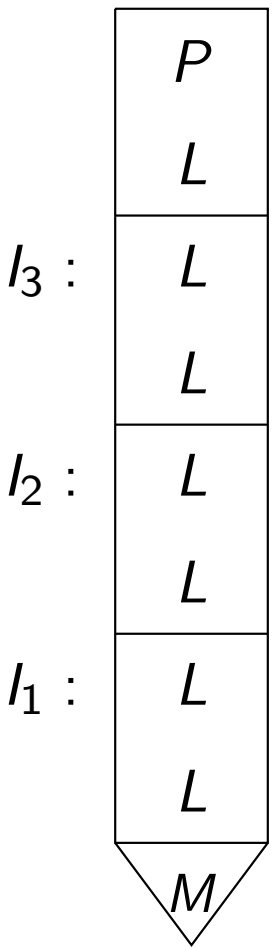
Table of Contents

What are they?

- Reflective Towers
- Model for reflection
- Used for reflective languages, e.g., 3-LISP [1], Brown [2], Blond [3]
- **n-fold interpretative overhead**

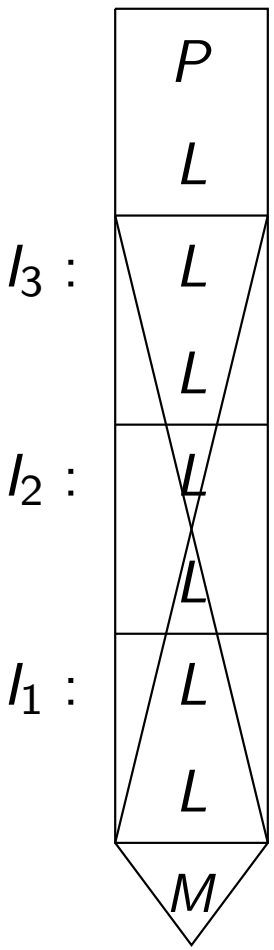


Removing Interpretative Overhead



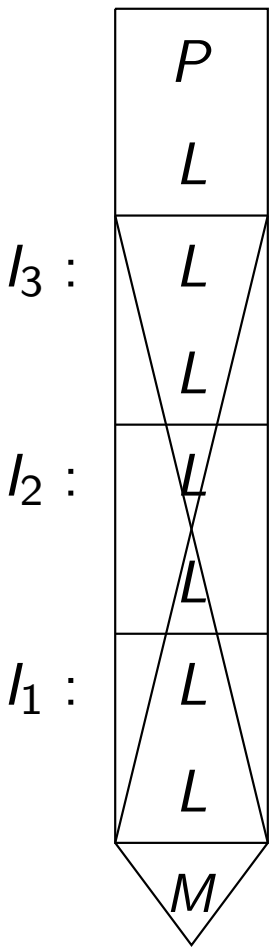
- All interpreters meta-circular $\Longrightarrow$ can remove any combination of I_1, I_2, I_3
  - Best case: simply run P on M

Removing Interpretative Overhead



- All interpreters meta-circular $\Longrightarrow$ can remove any combination of I_1, I_2, I_3
  - Best case: simply run P on M
- Idea: generate new tower without I_1 to I_3

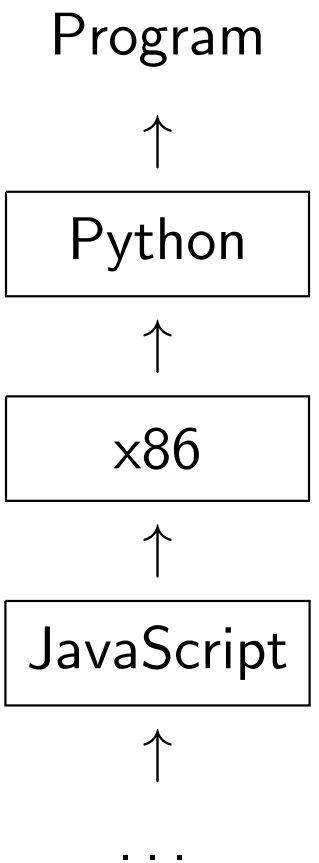
Removing Interpretative Overhead



- All interpreters meta-circular $\Longrightarrow$ can remove any combination of I_1, I_2, I_3
  - Best case: simply run P on M
- Idea: generate new tower without I_1 to I_3
  - Possible solution: compile the tower

Motivating Example

- Python on x86 JavaScript emulator
- Compared to reflective towers:
  - *Different language* at each level
  - *Different internal structure*, e.g., translation to bytecode



Motivating Example

- Python on x86 JavaScript emulator
- Compared to reflective towers:
  - *Different language* at each level
  - *Different internal structure*, e.g., translation to bytecode
    - we refer to this generalization as **heterogeneous towers**

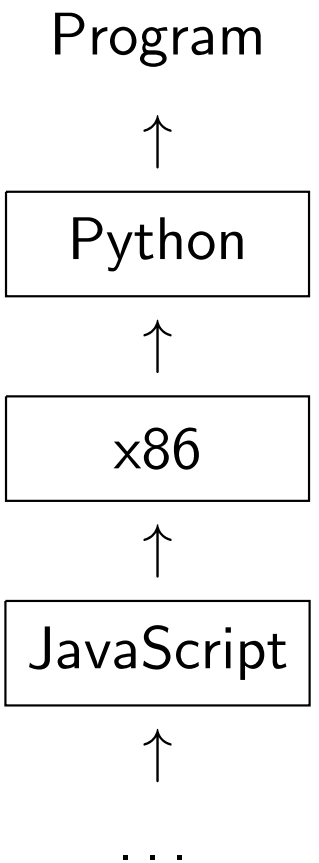


Table of Contents

Staging

- Staging: Split interpreter's execution into multiple stages, e.g., translation to some annotated source and then execution

Staging

- Staging: Split interpreter's execution into multiple stages, e.g., translation to some annotated source and then execution
- staged interpreter $\simeq$ compiler

Staging

- Staging: Split interpreter's execution into multiple stages, e.g., translation to some annotated source and then execution
- staged interpreter $\simeq$ compiler
- 2nd Futamura Projection [4] shows we can use partial evaluator, *mix*, for compilation:

$$\begin{aligned} \mathit{target} &= [\![\mathit{mix}]\!]\,(\mathit{interpreter}, \mathit{source}) \\ &= [\![\mathit{compiler}]\!]\,(\mathit{source}) \end{aligned}$$

- Can use staging to compile the program our tower executes

$\lambda_{\uparrow\downarrow}$

- Language Pink [5]
- Partial Evaluator
- Dynamic expressions wrapped in `Code(...)` constructor
- *lift* operator wraps expressions which we want to keep in the residual program

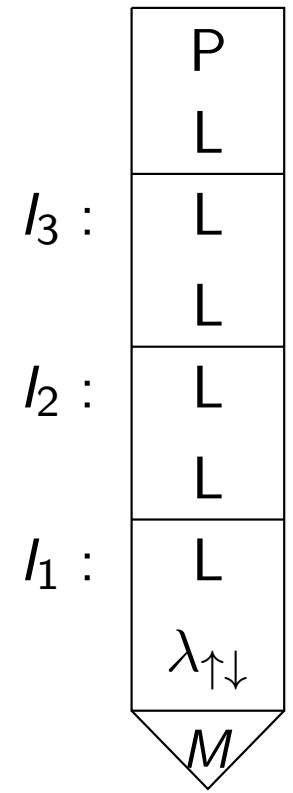
Example Staged Interpreter

```
(lambda _ eval (lambda _ exp (lambda _ env
  (if (num?                exp)     (lift exp)
  (if (sym?                exp)     (env exp)
  (if (sym?           (car exp))
    (if (eq?  '+      (car exp))    (+ ((eval (cadr exp)) env)
                                        ((eval (caddr exp)) env))
    ...
    (if (eq?  'lambda (car exp))    (lift (lambda ...
    (if (eq?  'lift   (car exp))    (lift ((eval (cadr exp)) env))
    (if (eq?  'cons   (car exp))    (lift (cons ...
    (if (eq?  'quote  (car exp))    (lift (cadr exp))
    ...
```

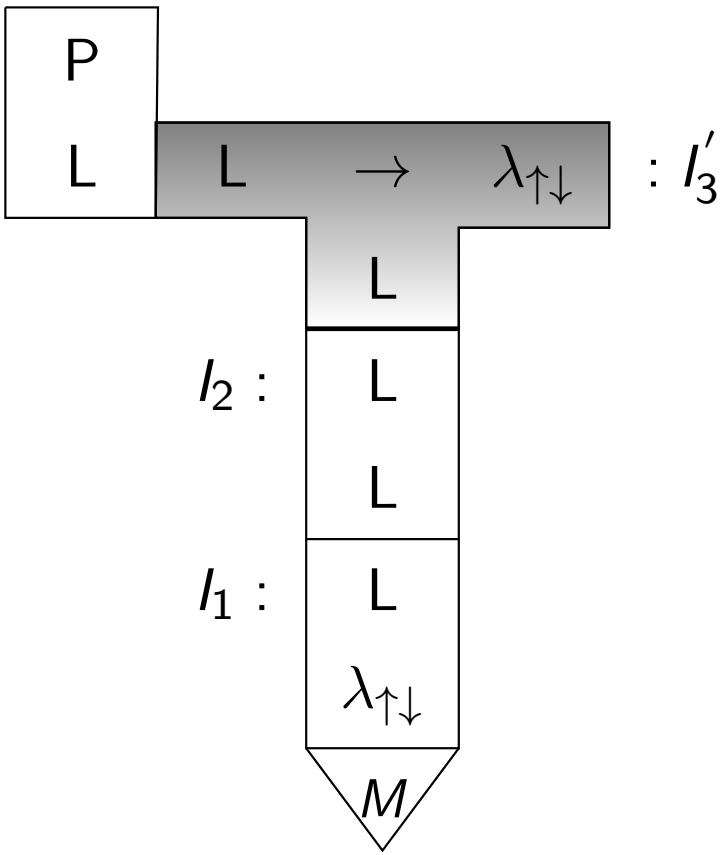
Table of Contents

Ingredients

- Multi-level Language
- Stage-polymorphic base: single evaluator can execute or residualize an expression
- TDPE-style Lift



Stage a Level



Residualize

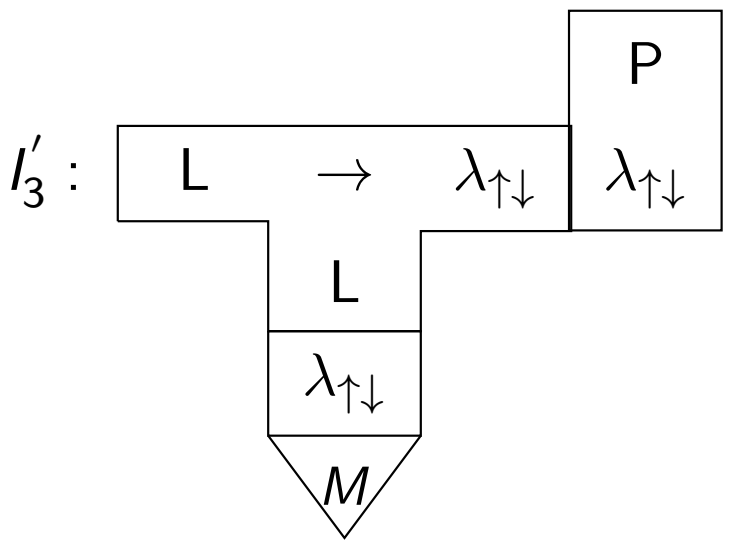
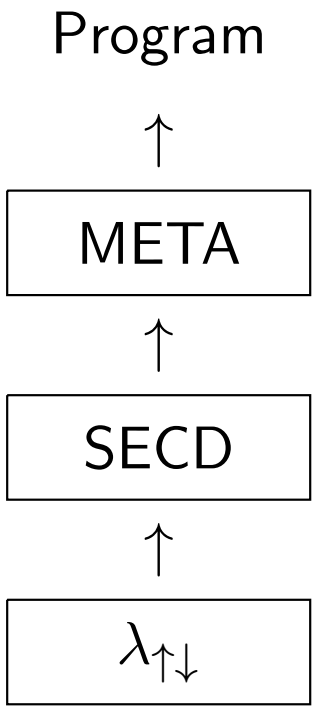


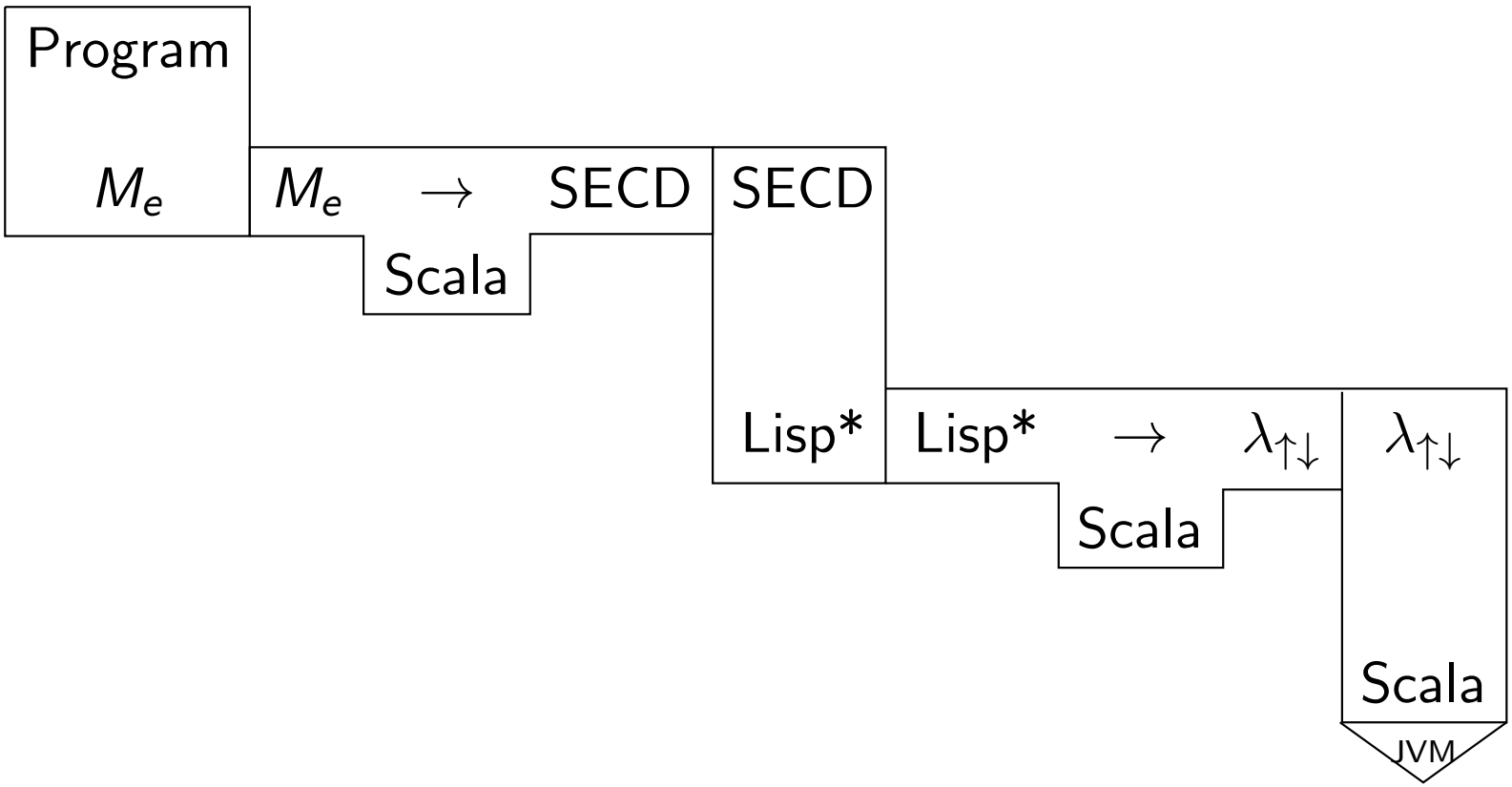
Table of Contents

Methodology

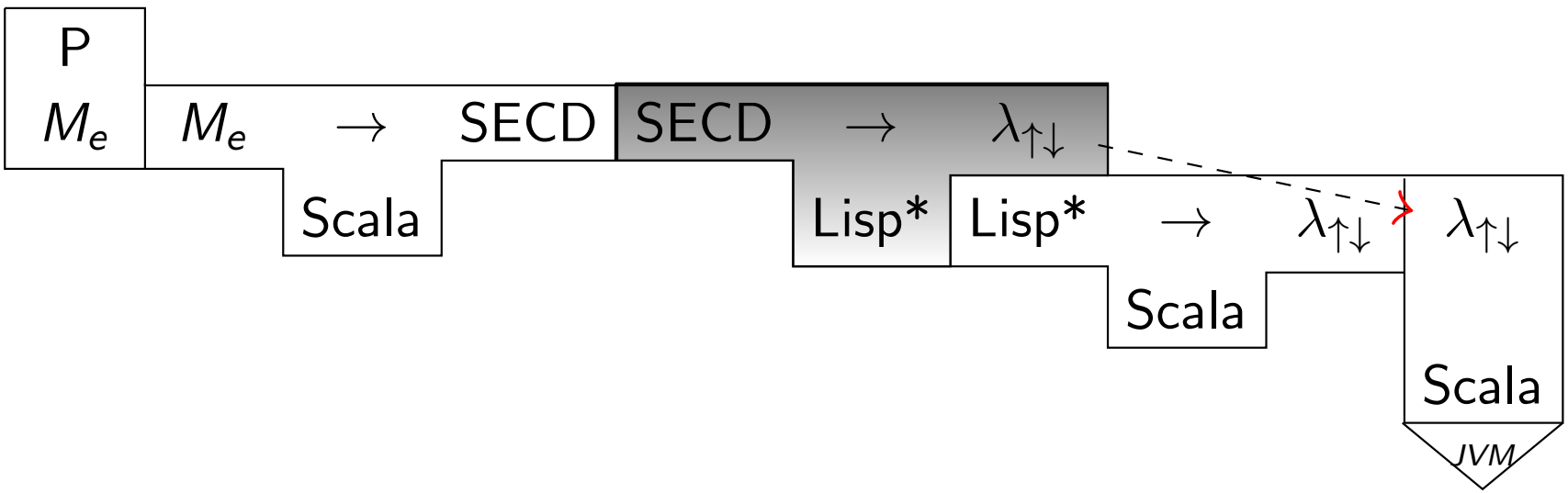
- Construct a tower resembling Python-x86-JavaScript
- Collapse it (while staging at various heights)



Experimental Tower



Staging SECD



Staging SECD

```
((lambda (fun)                    ;Y-combinator
          ((lambda (F)
             (F F))
           (lambda (F)
             (fun (lambda (x) ((F F) x))))))

        (lambda (factorial)   ;Factorial
          (lambda (n)
            (if (eq? n 0)
                1
                (* n (factorial (- n 1)))))))
```

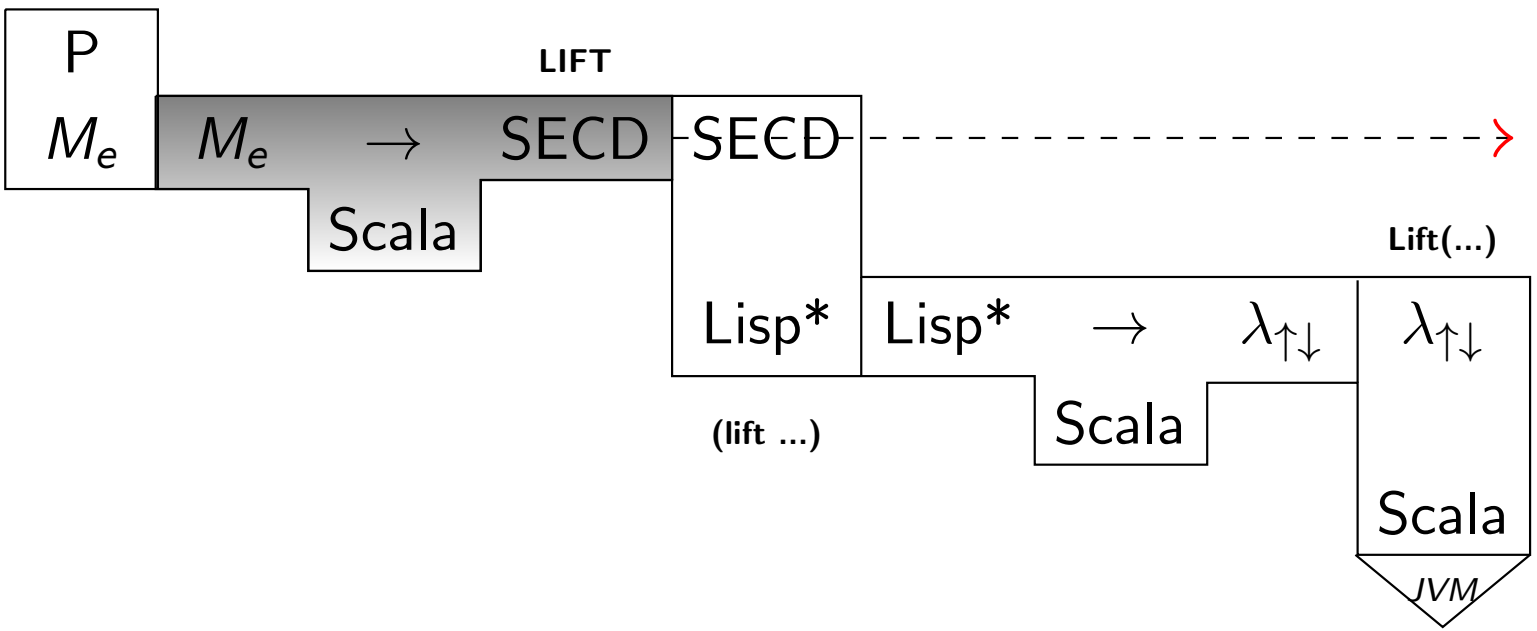
Staging SECD

```
DUM NIL LDF
(LD (1 1) SYM? SEL
   (NIL LD (1 1) CONS LD (1 2) AP JOIN) (LD (1 1) NUM? SEL
(NIL LD (1 1) CONS LDF
   (LD (1 1) RTN) AP JOIN) (LDC + LD (1 1) CAR EQ SEL
(NIL LD (1 2) CONS LD (1 1) CADDR CONS LDR (1 1) AP NIL
   LD (1 2) CONS LD (1 1) CADR CONS LDR (1 1) AP ADD JOIN)
   (LDC - LD (1 1) CAR EQ SEL
   ...
   LDC 1 CONS LDC n CONS LDC - CONS CONS LDC factorial
   CONS CONS LDC n CONS LDC * CONS CONS LDC 1 CONS
   LDC . LDC 0 CONS LDC n CONS LDC eq? CONS CONS LDC if
   CONS CONS LDC . LDC n CONS CONS LDC lambda
   ...
   LDR (1 1) AP RTN ) RAP STOP
```

Staging SECD

```
(let x0
  (lambda f0 x1
    (let x2
      (lambda f2 x3
        (let x4 (car x3)
        (let x5 (car x4)
        (let x6 (sym? x5)
        (if x6
          ...
          (let x8 (car x7)
          (let x9 (num? x8)
          (if x9
            ...
            (let x12 (car x11)
            (let x13 (eq? x12 '+)
            ...
```

Staging M_e


P
M_e
LIFT
M_e → SECD
Scala
SECD
Lisp*
(lift ...)
Lisp* → $\lambda_{\uparrow\downarrow}$
Scala
Lift(...)
$\lambda_{\uparrow\downarrow}$
Scala
JVM

Staging M_e

```
DUM NIL LDF
    (LD (1 1) SYM? SEL ;M_e dispatch Logic
         (NIL LD (1 1) CONS LD (1 2) AP JOIN )
    (LD (1 1) NUM? SEL
        (LD (1 1) LIFT JOIN ) ;Lift literals
  ...
 (LDC letrec LD (1 1) CAR EQ SEL
     (NIL NIL LDF
         (LD (2 1) CADR CAR LD (1 1) EQ SEL
             (LD (12 1) LIFT JOIN) ;Lift recursive lambdas
  ...
  (LDC lambda LD (1 1) CAR EQ SEL
     ...
    CONS LDR (1 1) AP RTN) LIFT JOIN) ;Lift lambdas
  ...
```

Staging M_e

```
(lambda f0 x1
  (let x2
    (lambda f2 x3
      (let x4
        (lambda f4 x5              ;Factorial
          (let x6 (eq? x3 0)
          (let x7
            (if f4 1
              (let x7 (- x3 1)
              (let x8 (x1 x5)
              (let x9 (* x3 x6) x7)))) x5))) f2))
  (let x3
    (lambda f3 x4                  ;Y-combinator
      (let x5
        (lambda f5 x6
          (let x7
        ...
```

Table of Contents

Conclusions

- Heterogeneity raises issues: likely to include translation layers and binding-time info not propagatable
- *lift* is able to eliminate interpretative overhead even in the presence of translation layers in a tower
- We propagate binding times by implementing *lift* in each interpreter (but it requires reverse engineering and transformation of program constructs between interpreter boundaries)

Future Work

- Bringing methodology even closer to practice
- Does our methodology extend to side-effects?
- Interpreter of different paradigms? E.g., WAM for logic programming
- Less-intrusive collapse

References

[1] B. C. Smith, "Reflection and semantics in Lisp," in *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1984, pp. 23–35.

[2] M. Wand and D. P. Friedman, "The mystery of the tower revealed: A nonreflective description of the reflective tower," *Lisp and Symbolic Computation*, vol. 1, no. 1, pp. 11–38, 1988.

[3] O. Danvy and K. Malmkjaer, "Intensions and extensions in a reflective tower," in *Proceedings of the 1988 ACM conference on LISP and functional programming*. ACM, 1988, pp. 327–341.

[4] Y. Futamura, "Partial evaluation of computation process–an approach to a compiler-compiler," *Higher-Order and Symbolic Computation*, vol. 12, no. 4, pp. 381–391, 1999.

[5] N. Amin and T. Rompf, "Collapsing towers of interpreters," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 52, 2017.