Collapsing Towers of Interpreters

Nada Amin
University of Cambridge, UK — nada.amin@cl.cam.ac.uk

joint work with Tiark Rompf
Purdue University, USA — tiark@purdue.edu

The Challenge

Collapse

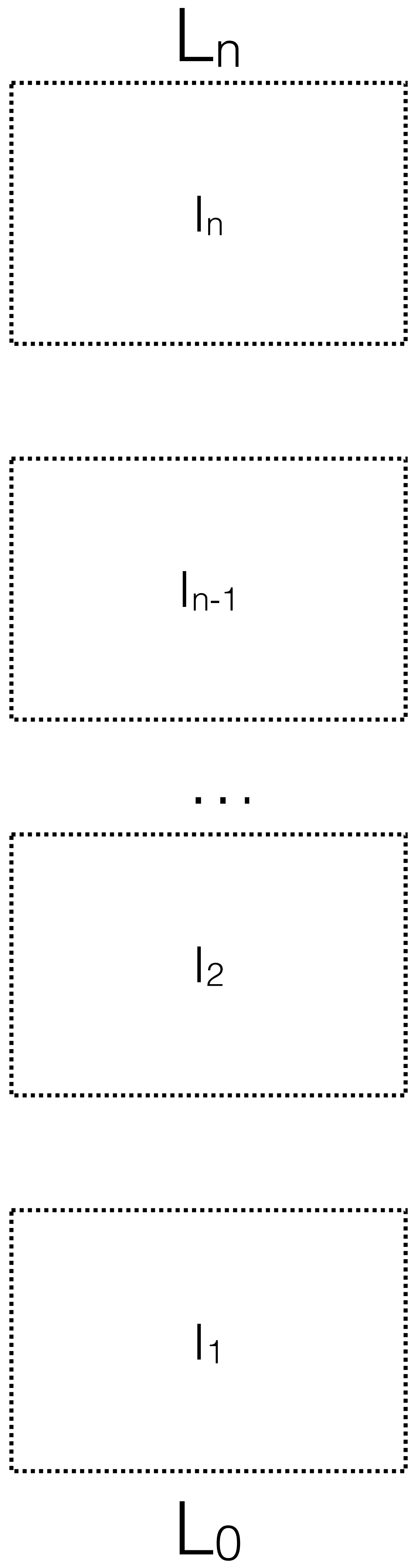
a tower of interpreters

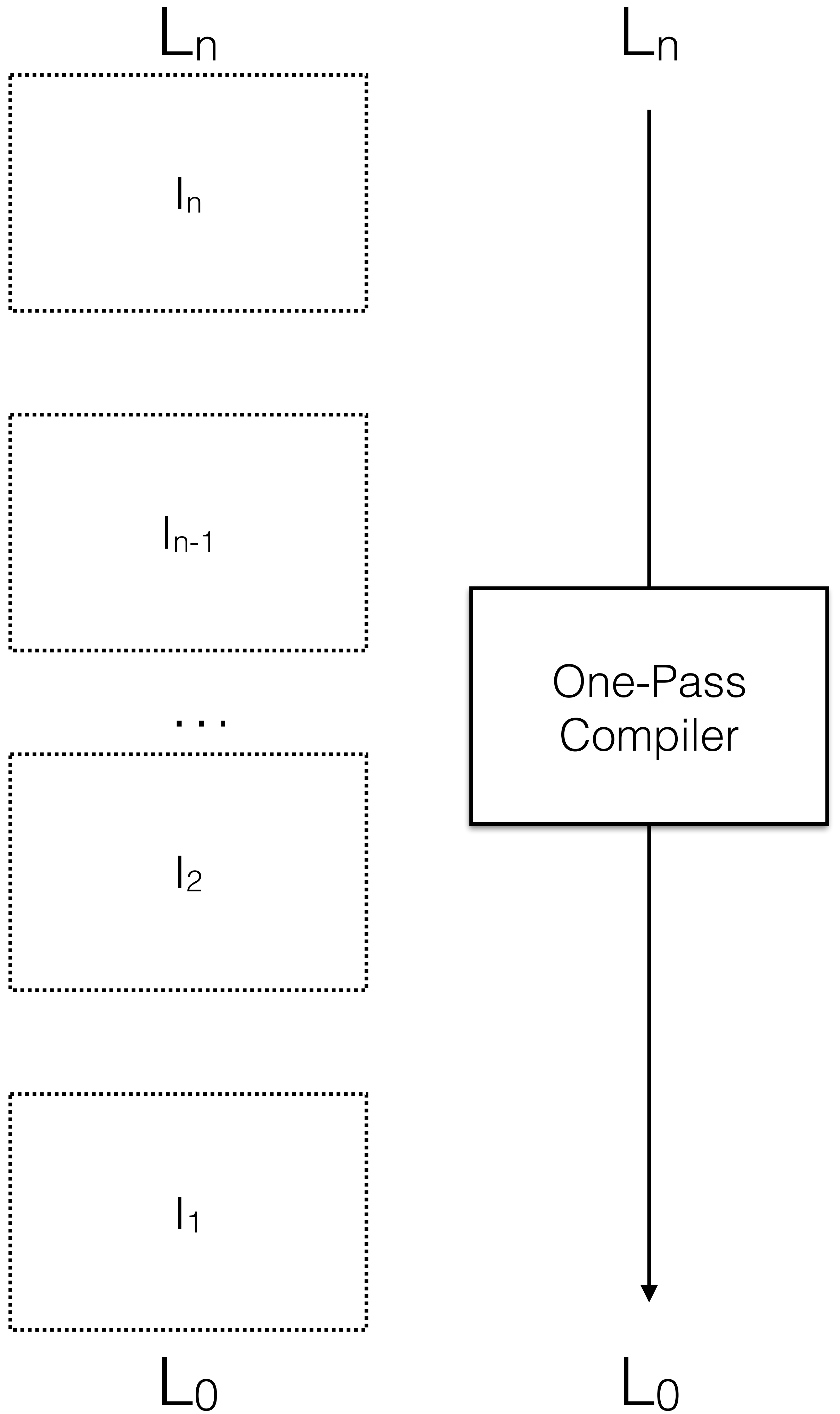
(languages L_0, …, L_n & interpreters for L_{i+1} written in L_i)

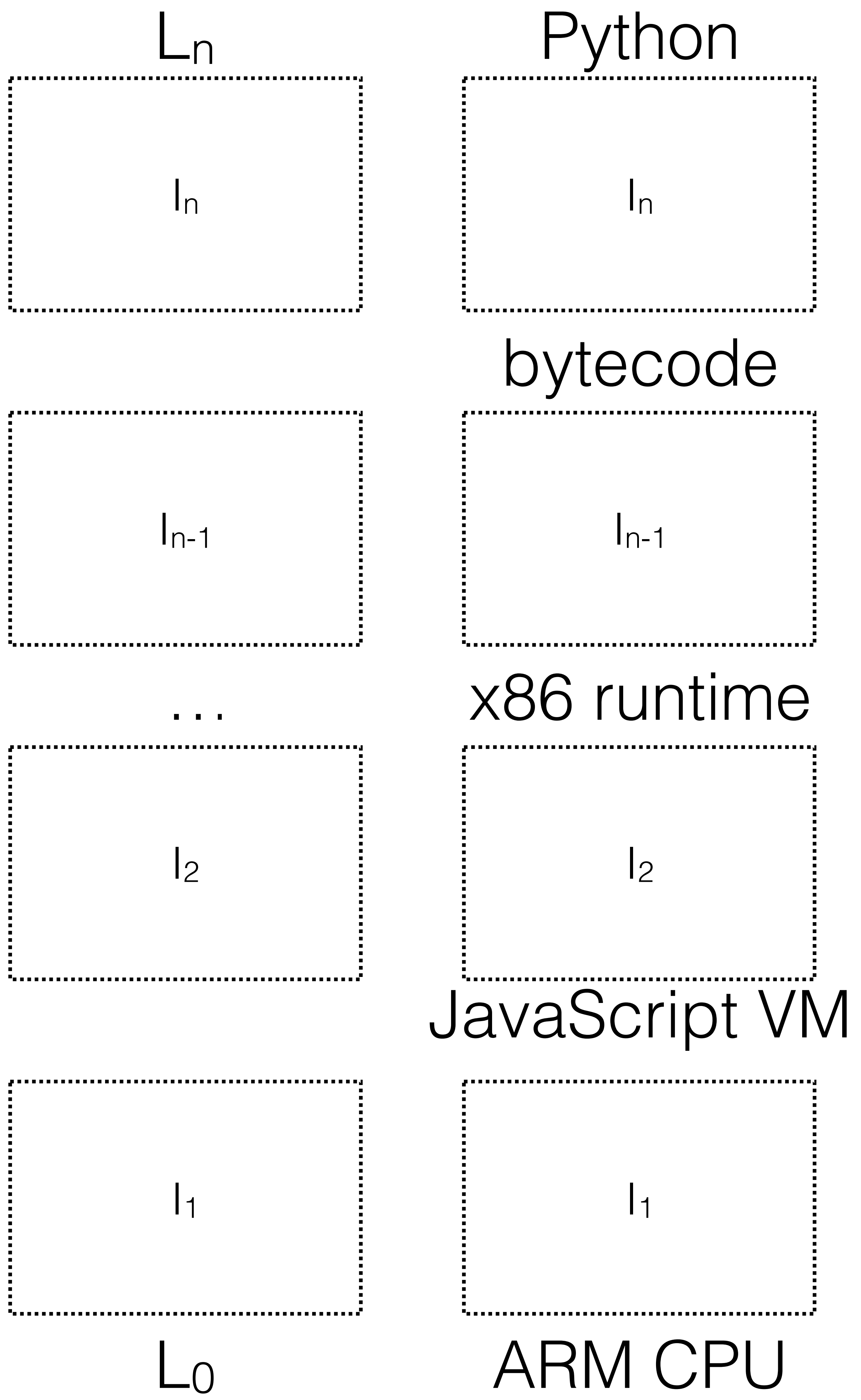
into

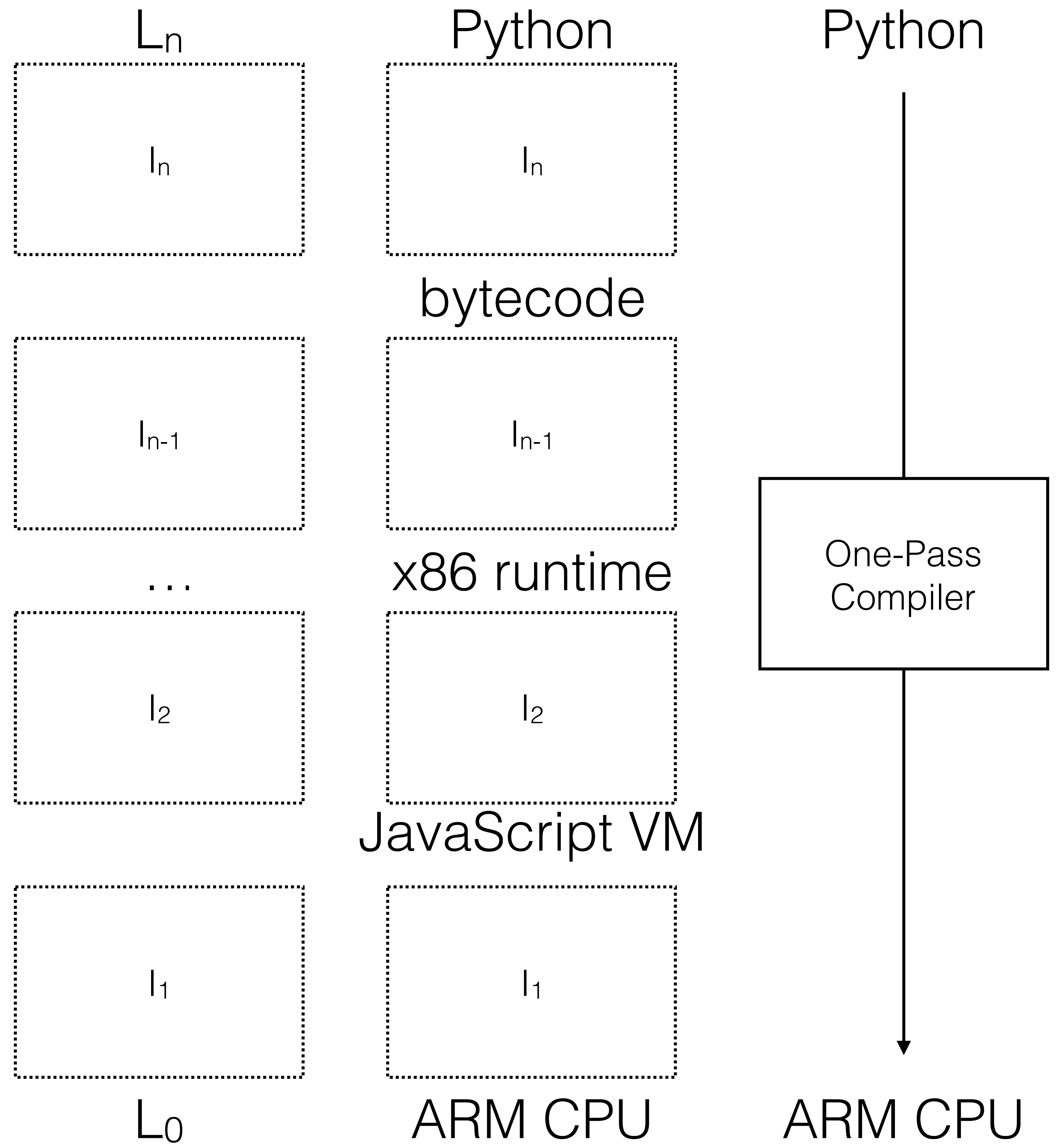
a one-pass compiler from L_n to L_0

removing all interpretive overhead

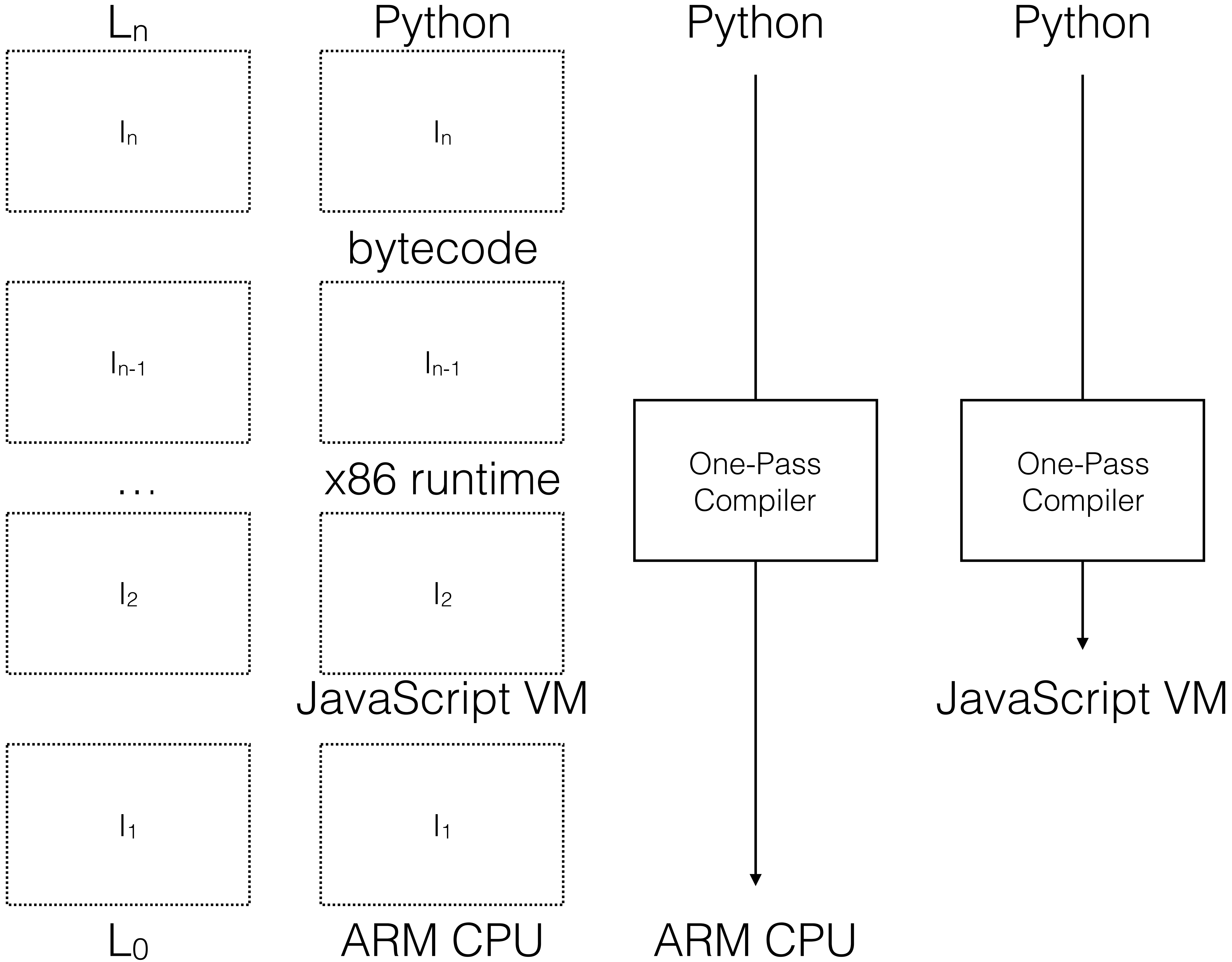

L_n
I_n
I_{n-1}
...
I_2
I_1
L_0

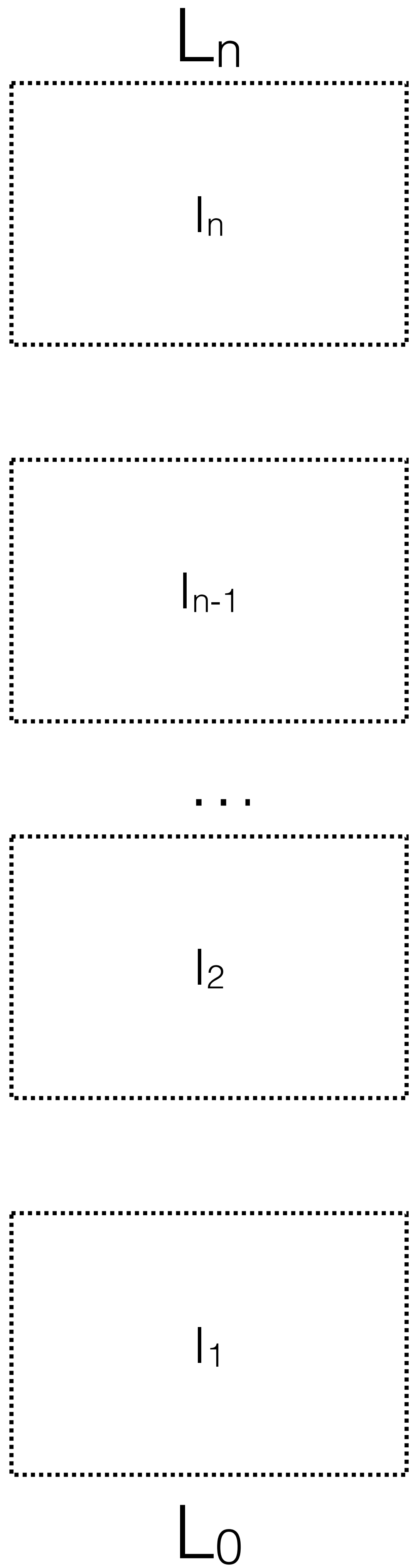

L_n
I_n
I_{n-1}
...
I_2
I_1
L_0
L_n
One-Pass Compiler
L_0


L_n
Python
I_n
I_n
bytecode
I_{n-1}
I_{n-1}
…
x86 runtime
I_2
I_2
JavaScript VM
I_1
I_1
L_0
ARM CPU

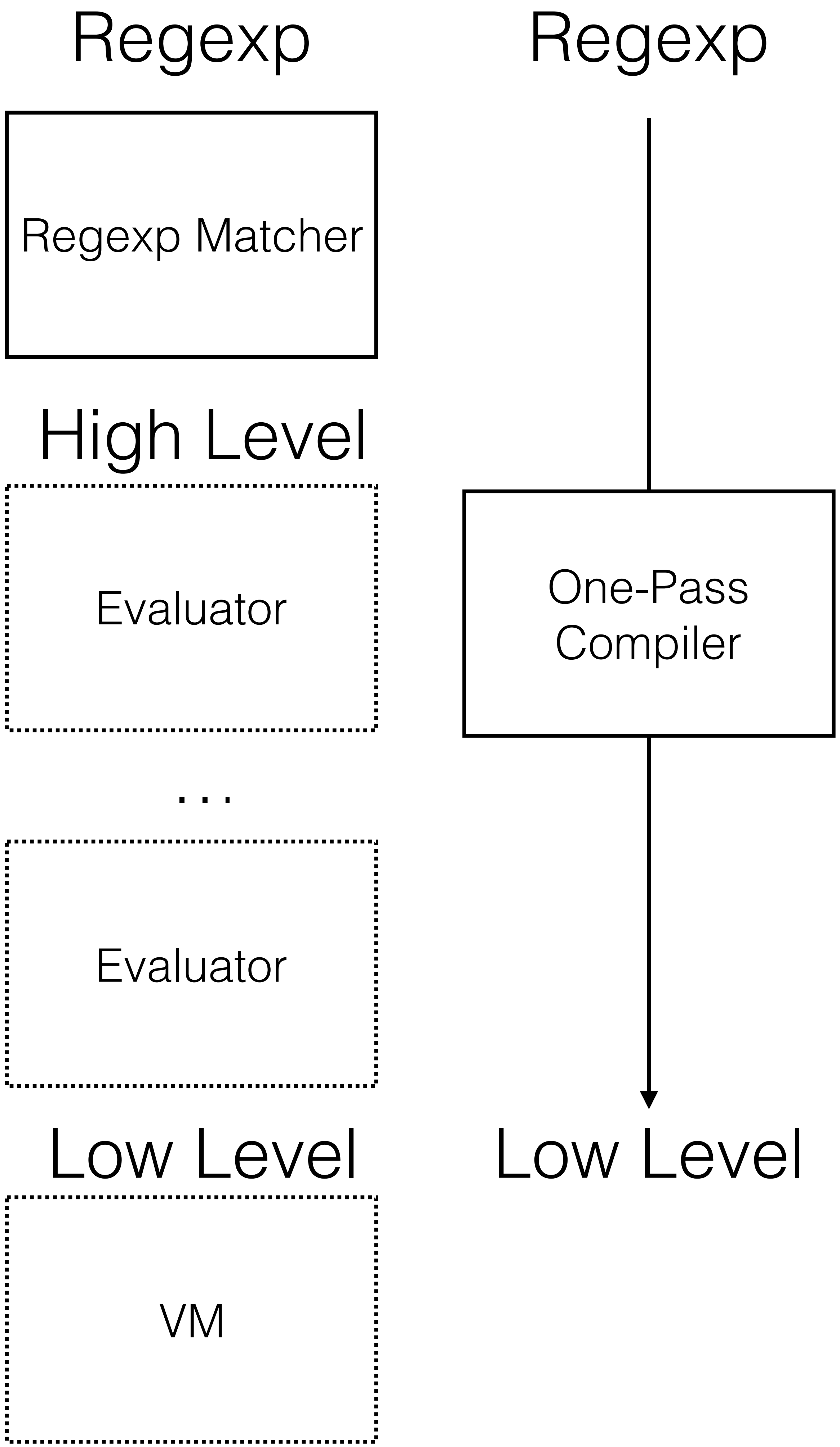


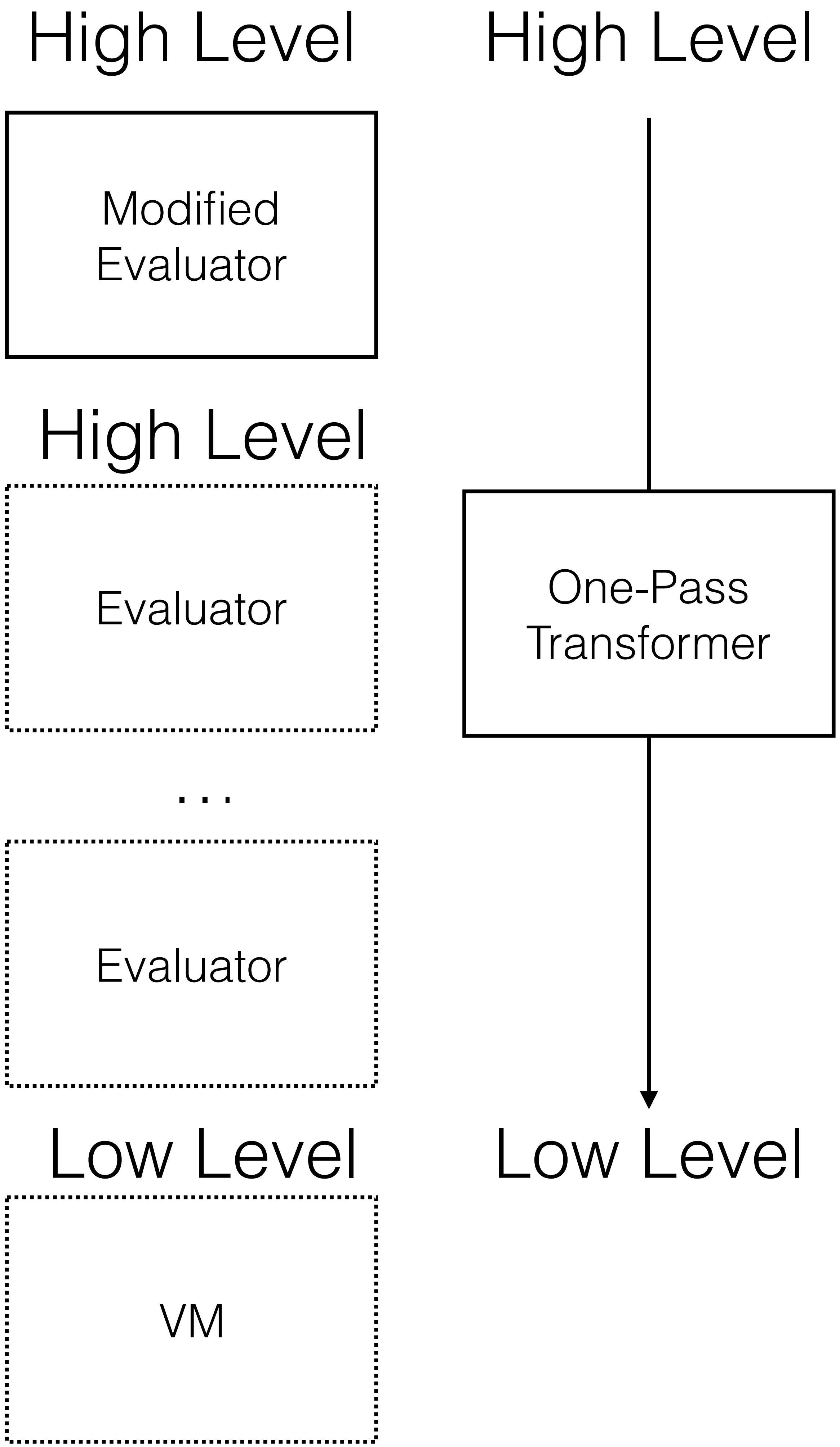
L_n
I_n
I_{n-1}
…
I_2
I_1
L_0
Python
I_n
bytecode
I_{n-1}
x86 runtime
I_2
JavaScript VM
I_1
ARM CPU
Python
One-Pass Compiler
ARM CPU

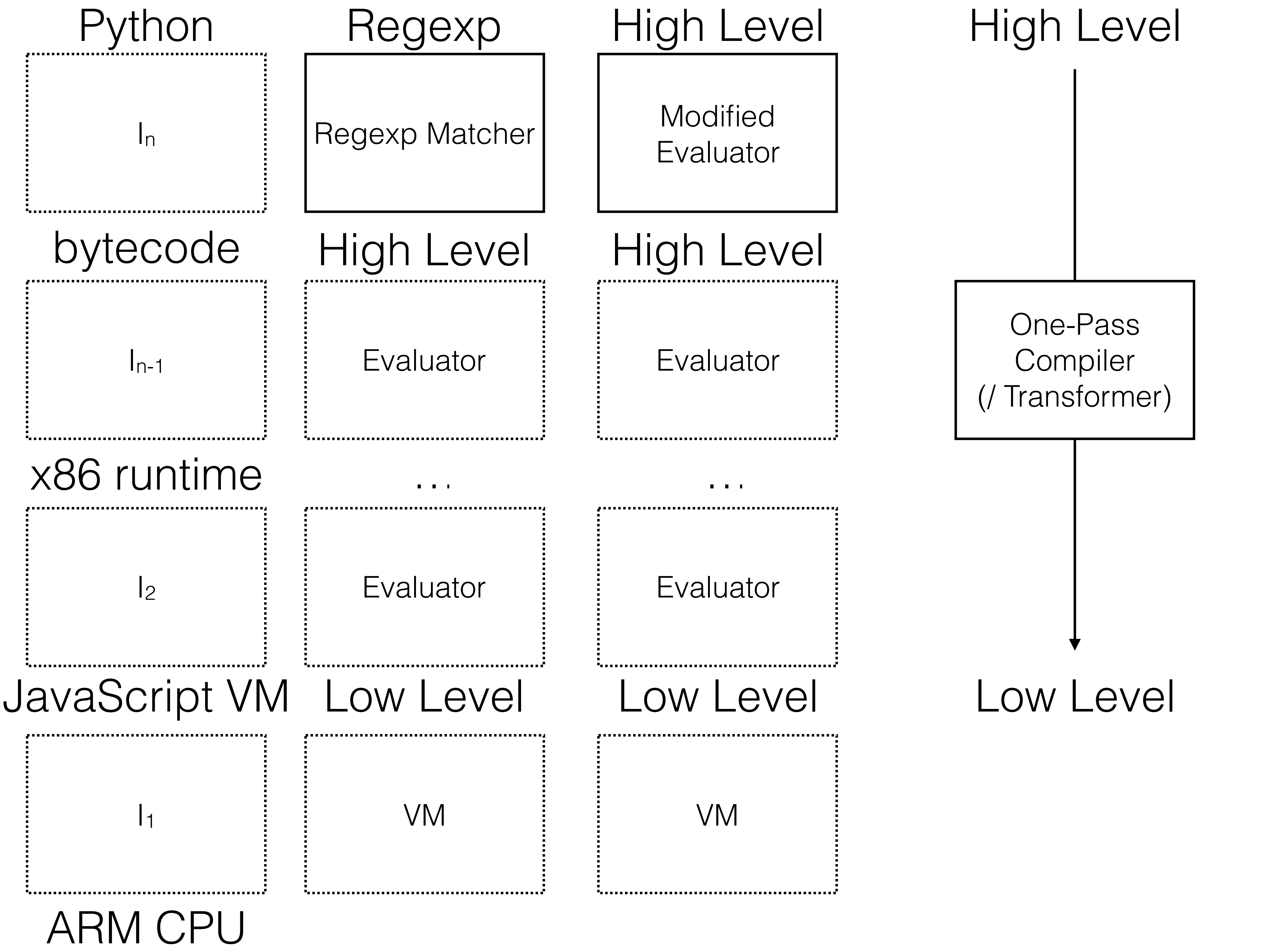

Lₙ
Iₙ
Iₙ₋₁
...
I₂
I₁
L₀
Python
Iₙ
bytecode
Iₙ₋₁
x86 runtime
I₂
JavaScript VM
I₁
ARM CPU
Python
One-Pass Compiler
ARM CPU
Python
One-Pass Compiler
JavaScript VM

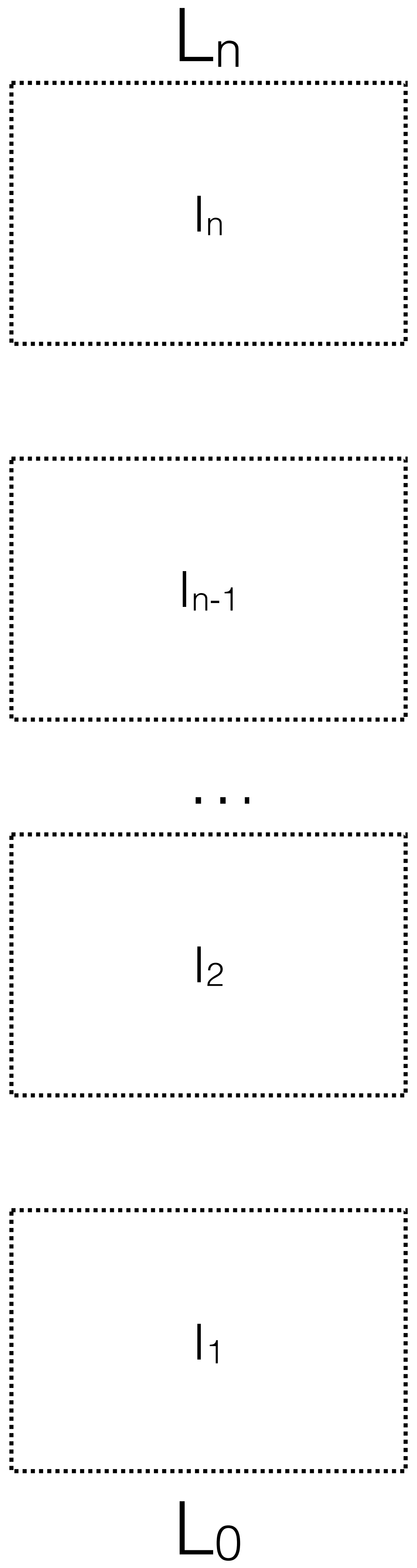


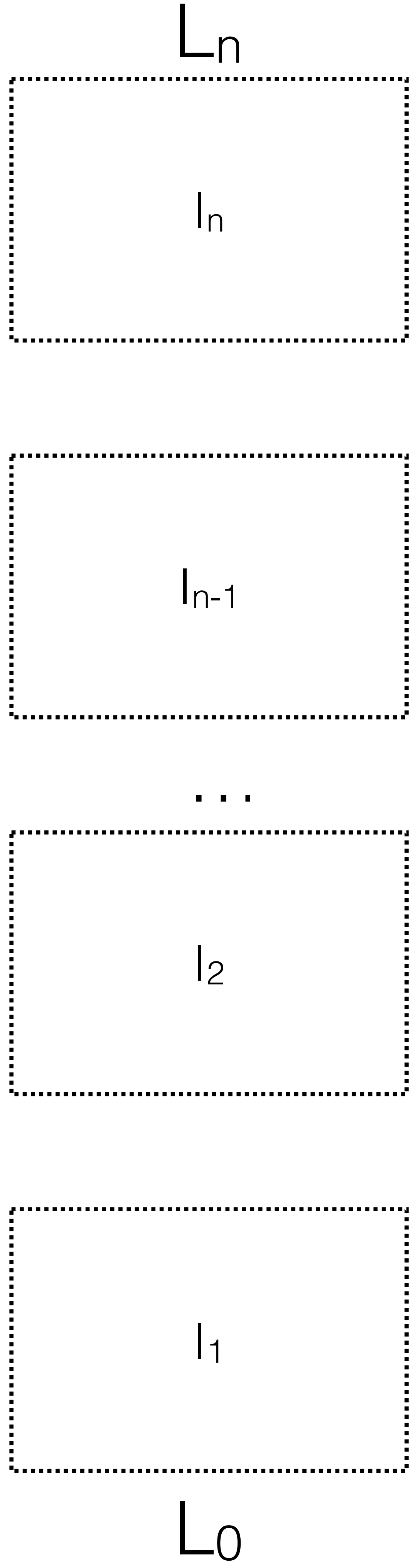
L_n
l_n
l_{n-1}
...
l_2
l_1
L_0


Regexp
Regexp Matcher
High Level
Evaluator
. . .
Evaluator
Low Level
VM
Regexp
One-Pass Compiler
Low Level


High Level
Modified Evaluator
High Level
Evaluator
...
Evaluator
Low Level
VM
High Level
One-Pass Transformer
Low Level


Python
I_n
bytecode
I_{n-1}
x86 runtime
I_2
JavaScript VM
I_1
ARM CPU
Regexp
Regexp Matcher
High Level
Evaluator
...
Evaluator
Low Level
VM
High Level
Modified Evaluator
High Level
Evaluator
...
Evaluator
Low Level
VM
High Level
One-Pass Compiler (/ Transformer)
Low Level

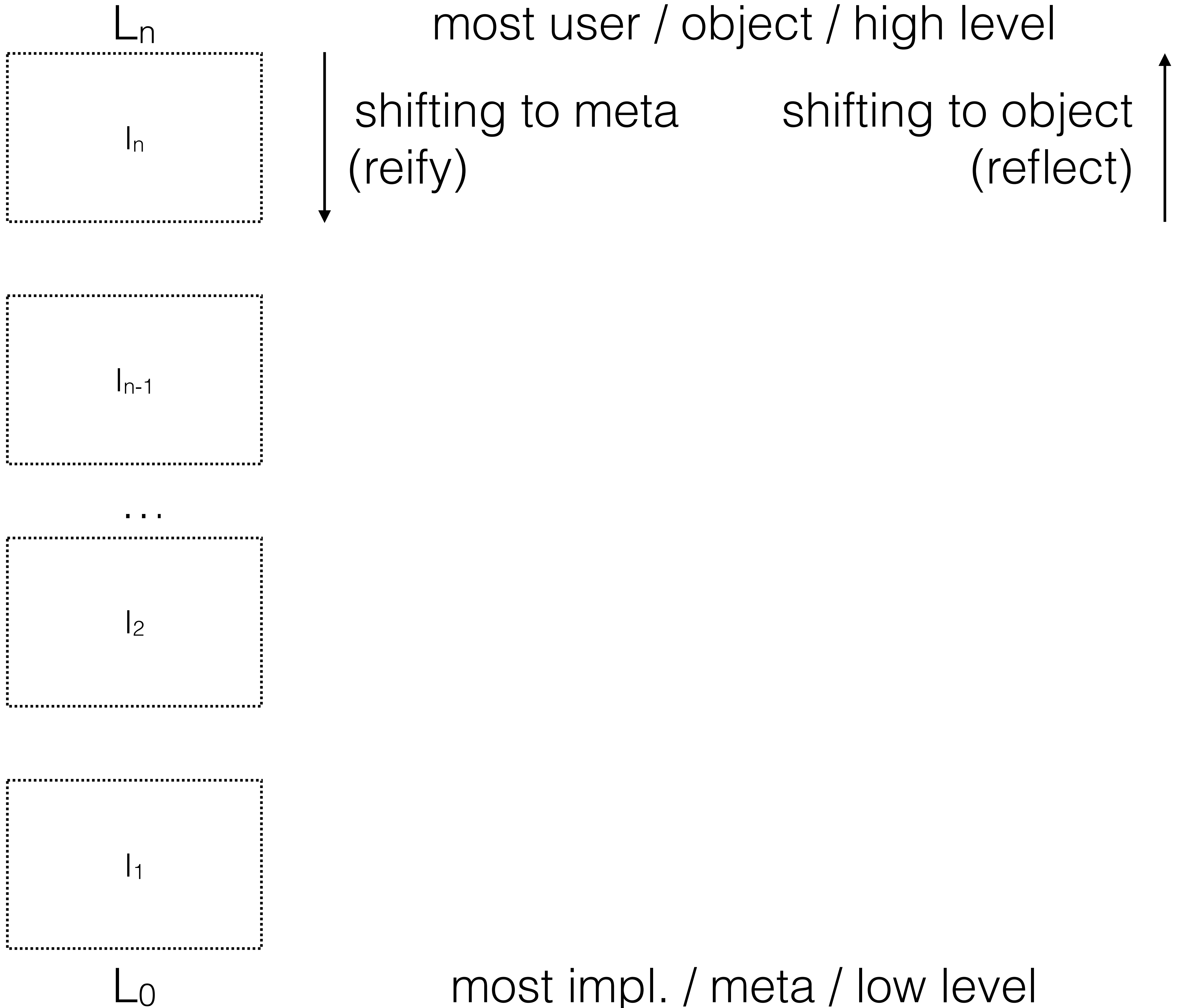

L_n
l_n
l_{n-1}
...
l_2
l_1
L_0

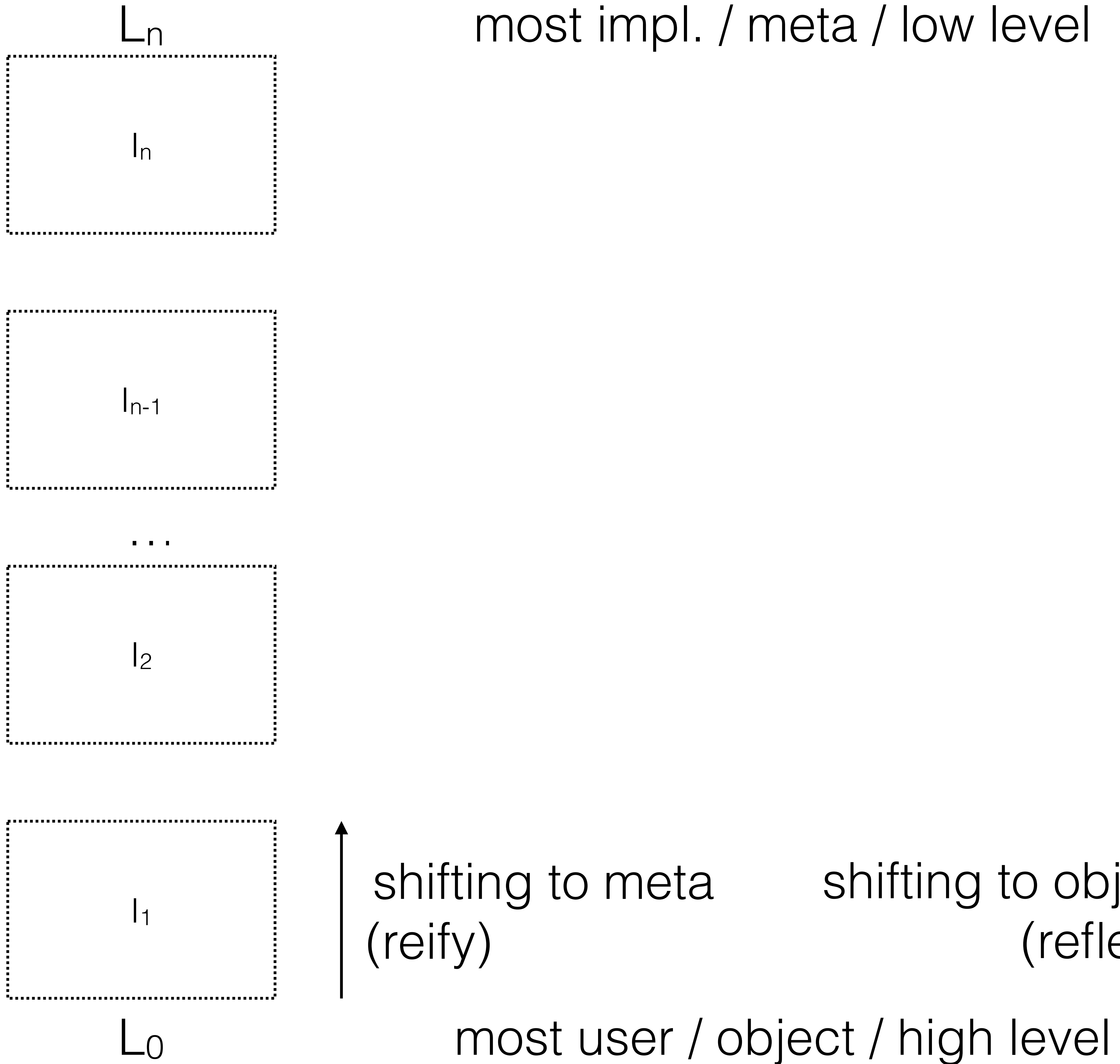


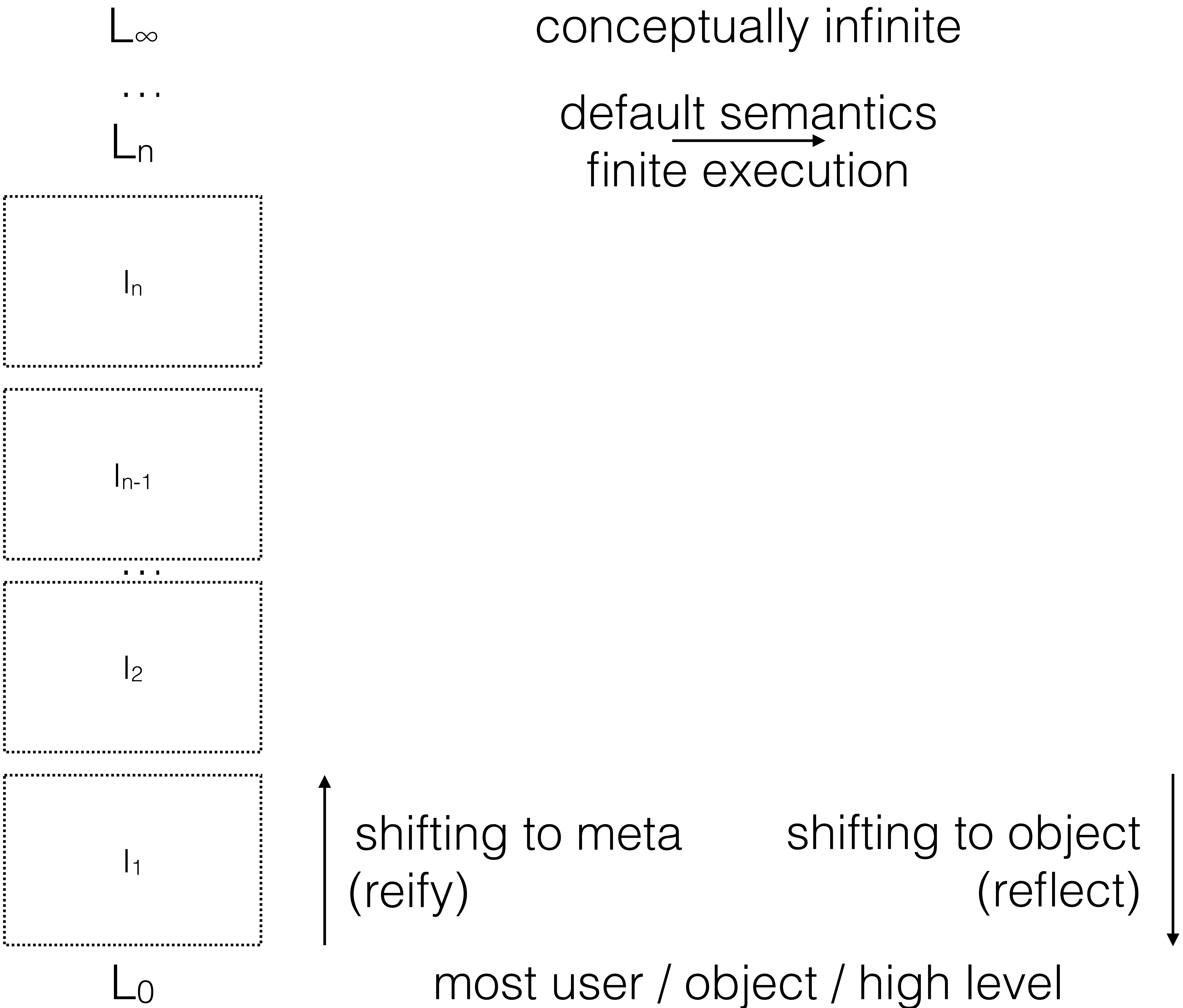
~ conceptually **infinite**

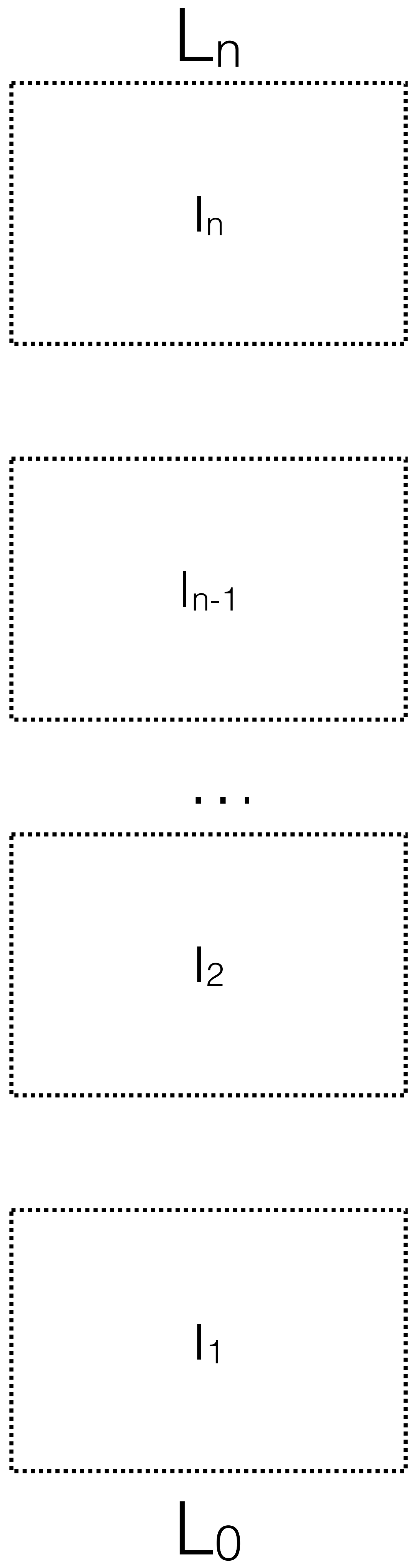
~ **reflective**
can be inspected and modified
at runtime

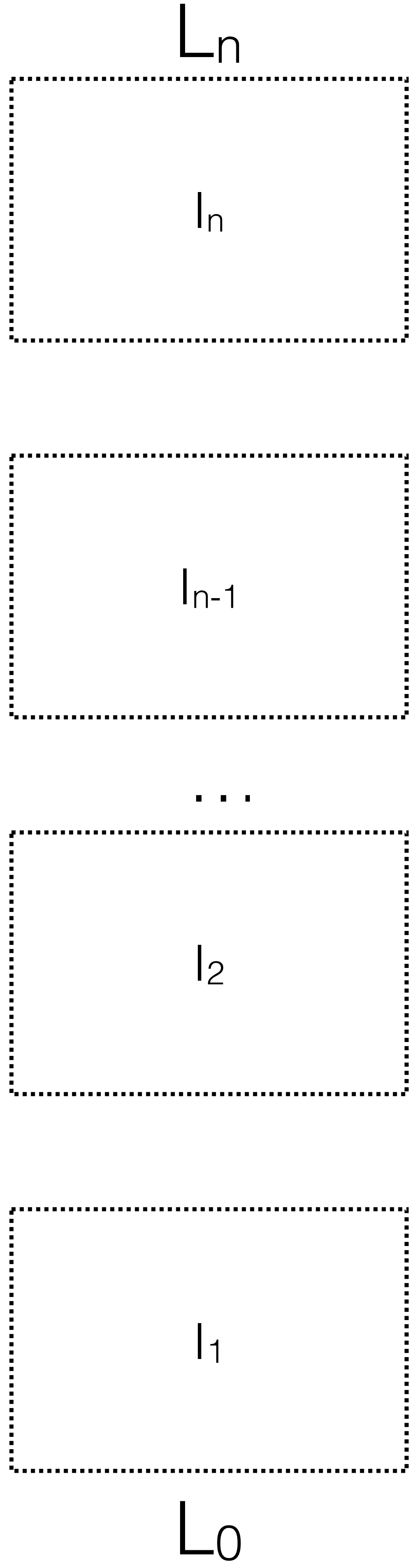
base L_0 = variant of **λ-calculus**


L_n
most user / object / high level
shifting to meta
(reify)
shifting to object
(reflect)
I_n
I_{n-1}
…
I_2
I_1
L_0
most impl. / meta / low level


L_n
most impl. / meta / low level
I_n
I_{n-1}
…
I_2
I_1
shifting to meta (reify)
shifting to object (reflect)
L_0
most user / object / high level


L∞
conceptually infinite
…
Ln
default semantics
finite execution
In
In-1
…
I2
I1
shifting to meta
(reify)
shifting to object
(reflect)
L0
most user / object / high level

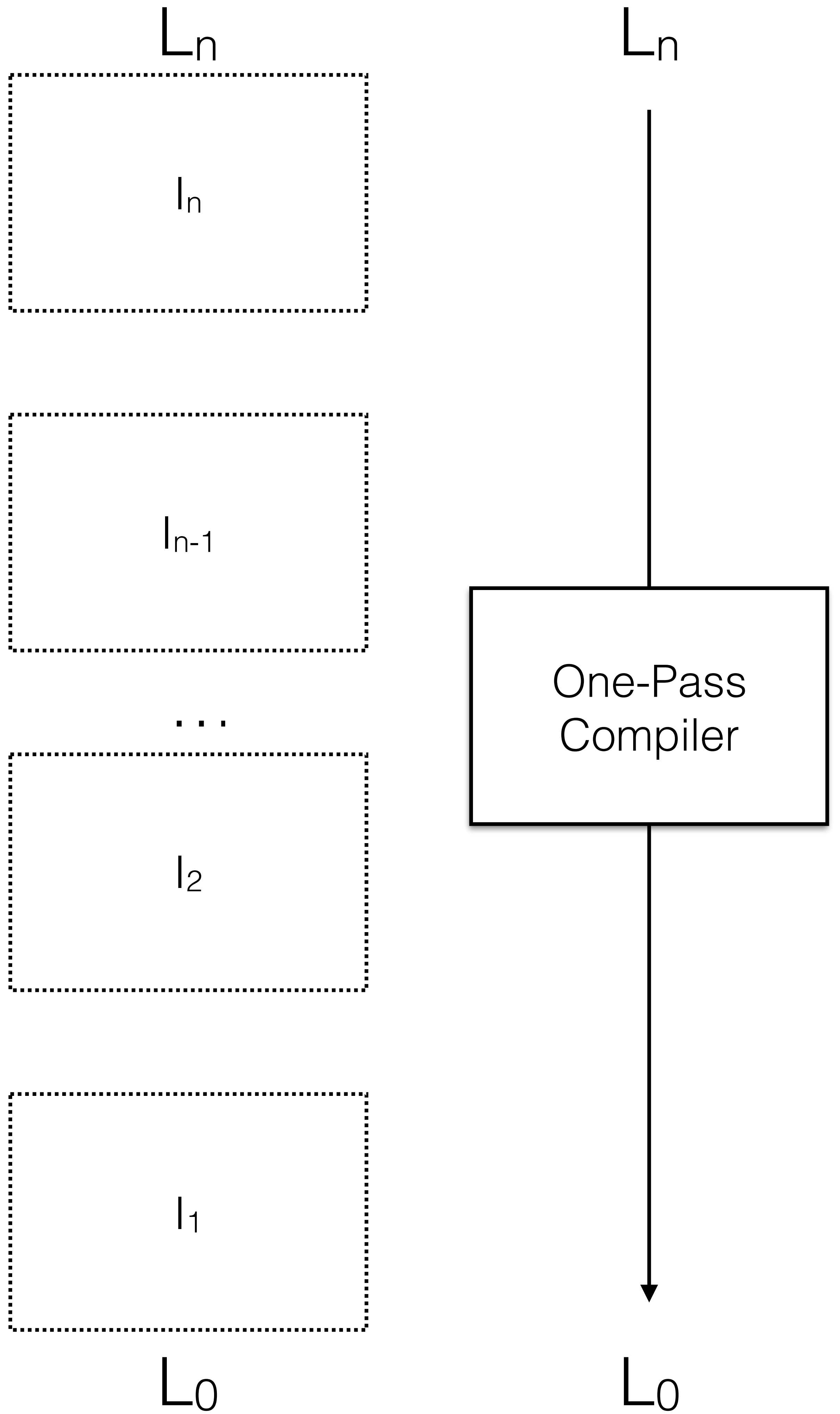

L_n
I_n
I_{n-1}
...
I_2
I_1
L_0



~ conceptually **infinite**

~ **reflective**
can be inspected and modified
at runtime

base L_0 = variant of **λ-calculus**


L_n
I_n
I_{n-1}
...
I_2
I_1
L_0
L_n
One-Pass Compiler
L_0

Example in Purple/Black

```
(define fib (lambda (n)
  (if (< n 2) n
    (+ (fib (- n 1)) (fib (- n 2))))))

> (fib 7) ;; => 13
```

Reflection in Purple/Black

```
(EM (begin ;; EM = Execute-at-Metalevel
(define counter 0)
(define old-eval-var eval-var)
(set! eval-var (lambda (e r k)
  (if (eq? e ’n)
    (set! counter (+ counter 1))
  (old-eval-var e r k)))))


> (fib 7) ;; => 13
> (EM counter) ;; => 102
```

Compilation in Purple

```
(set! fib (clambda (n) ;; c = compiled
  (if (< n 2) n
    (+ (fib (- n 1)) (fib (- n 2))))))
```

```
> (EM (set! counter 0))
> (fib 7) ;; => 13
> (EM counter) ;; => 102
```

Compilation in Purple

```
(EM (set! eval-var old-eval-var))
> (EM (set! counter 0))
> (fib 7) ;; => 13
> (EM counter) ;; => 102
```

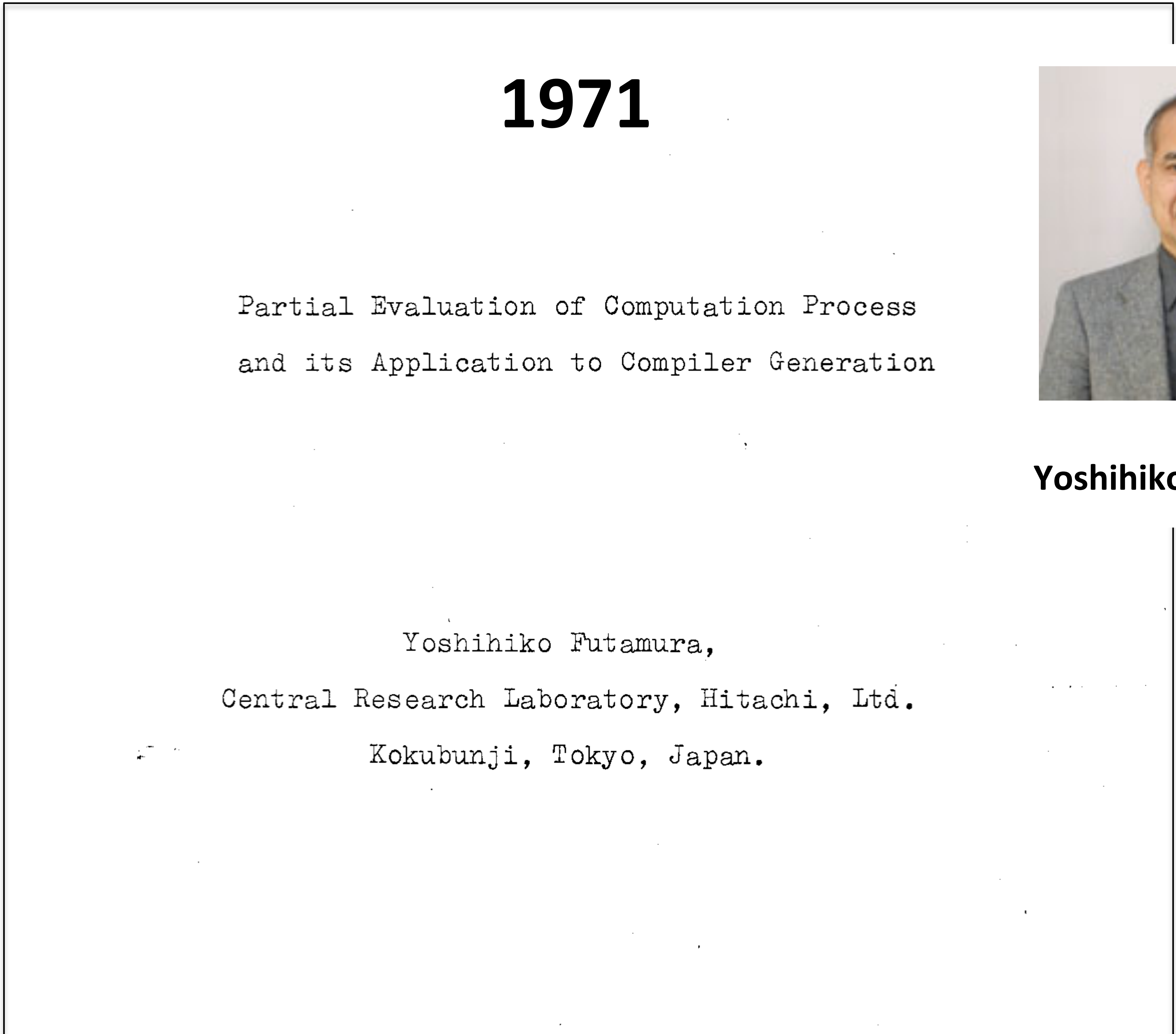
```
(set! fib (clambda (n) …)
> (EM (set! counter 0))
> (fib 7) ;; => 13
> (EM counter) ;; => 0
```

Collapse in Purple

```
{(k, xs) => _app(‘+, _cons(_cell_read(<cell counter>), ‘(1)),
_cont{c_1 => _cell_set(<cell counter>, c_1) _app(‘<, _cons(_car(xs),
‘(2)), _cont{v_1 => _if(_true(v_1),
_app(‘+, _cons(_cell_read(<cell counter>), ‘(1)), _cont{c_2 =>
_cell_set(<cell counter>, c_2)
_app(k, _cons(_car(xs), ‘()), _cont{v_2 => v_2})}),
_app(‘+, _cons(_cell_read(<cell counter>), ‘(1)), _cont{c_3 =>
_cell_set(<cell counter>, c_3)
_app(‘-, _cons(_car(xs), ‘(1)), _cont{v_3 => _app(_cell_read(<cell
fib>), _cons(v_3, ‘()), _cont{v_4 =>
_app(‘+, _cons(_cell_read(<cell counter>), ‘(1)), _cont{c_4 =>
_cell_set(<cell counter>, c_4)
_app(‘-, _cons(_car(xs), ‘(2)), _cont{v_5 => _app(_cell_read(<cell
fib>), _cons(v_5, ‘()), _cont{v_6 =>
_app(‘+, _cons(v_4, _cons(v_6, ‘())), _cont{v_7 => _app(k,
_cons(v_7, ‘()), _cont{v_8 => v_8})})})})})})})})})})})})}
```

Solving the Challenge

1971



Partial Evaluation of Computation Process

and its Application to Compiler Generation

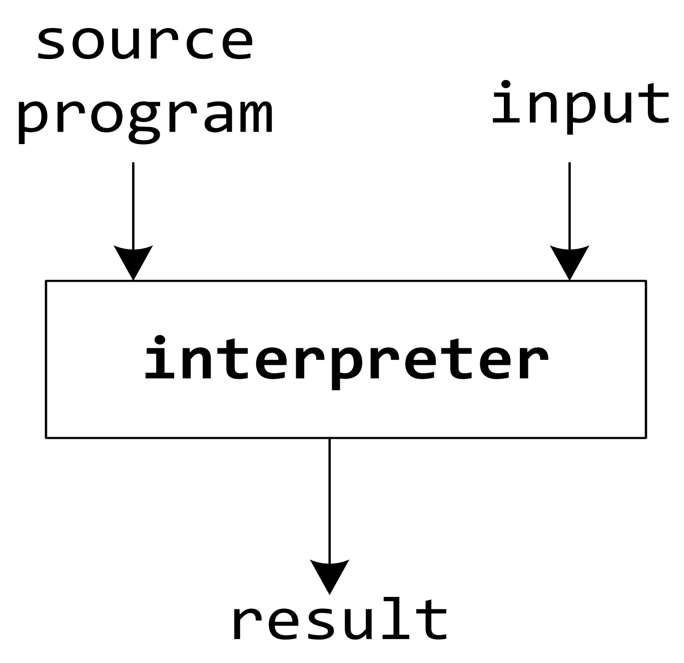
Yoshihiko Futamura,

Central Research Laboratory, Hitachi, Ltd.

Kokubunji, Tokyo, Japan.

Yoshihiko Futamura

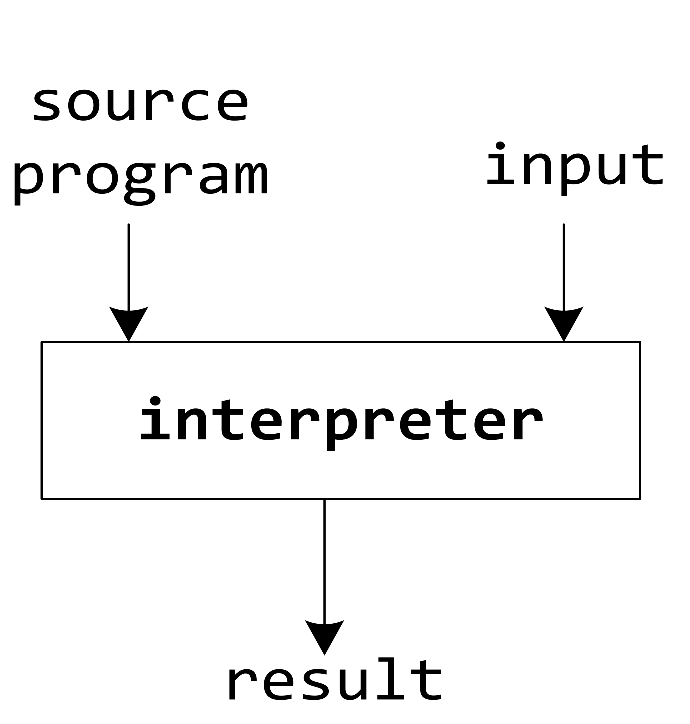
The 1st Futamura Projection



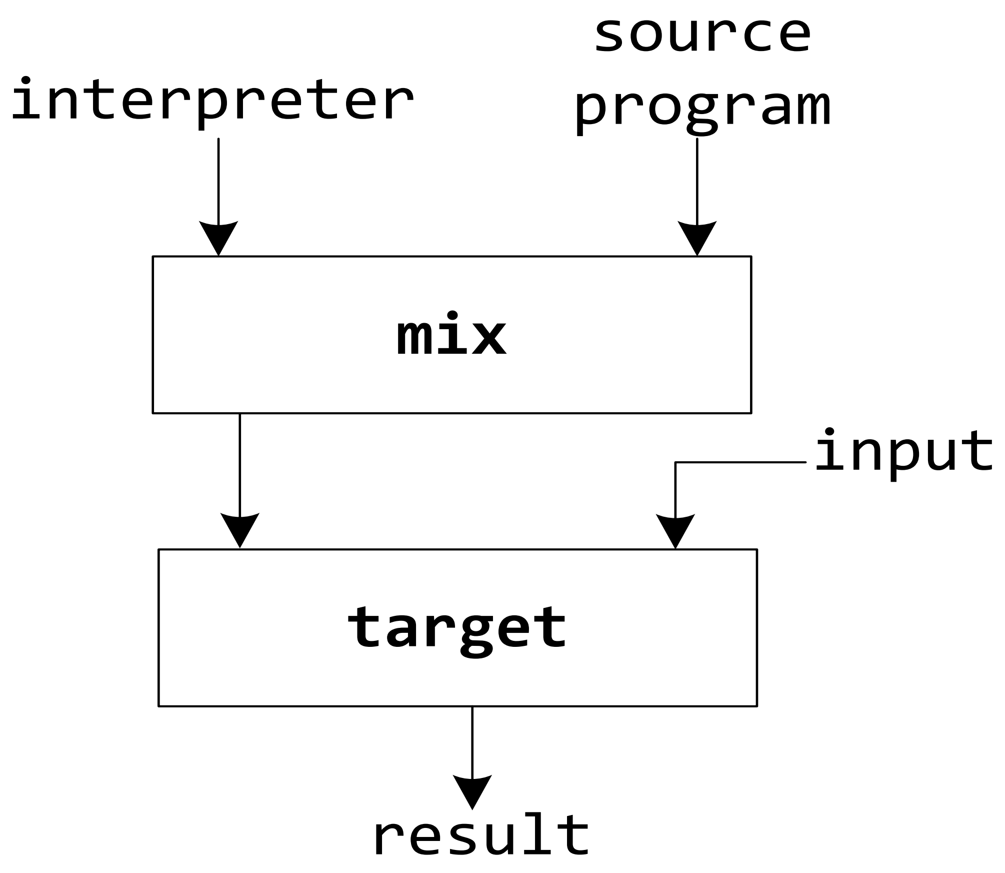
(a)

image credit: Ruby Tahboub (Purdue)

The 1st Futamura Projection



(a)



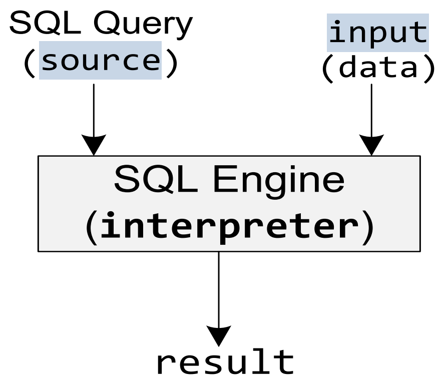
(b)

Specializing an **interpreter** with respect to a program produces a **compiled** version of that program.

image credit: Ruby Tahboub (Purdue)

Practical Realization

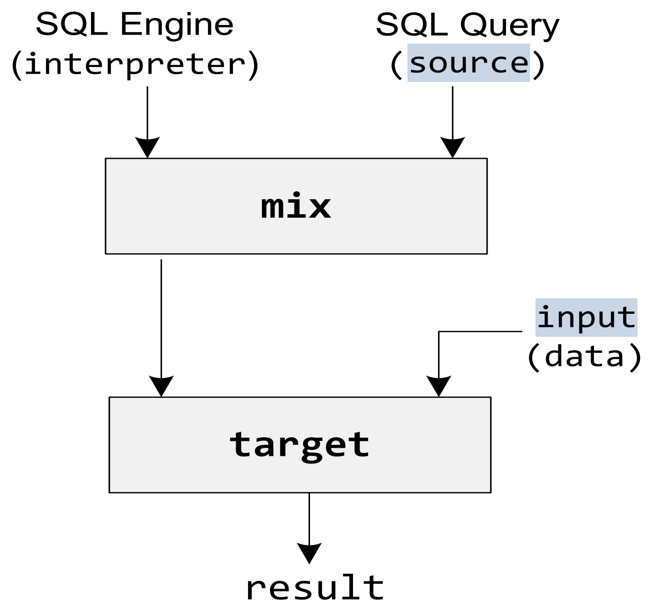
```
result = interpreter(source, input)
```



(a)

Query Interpreter

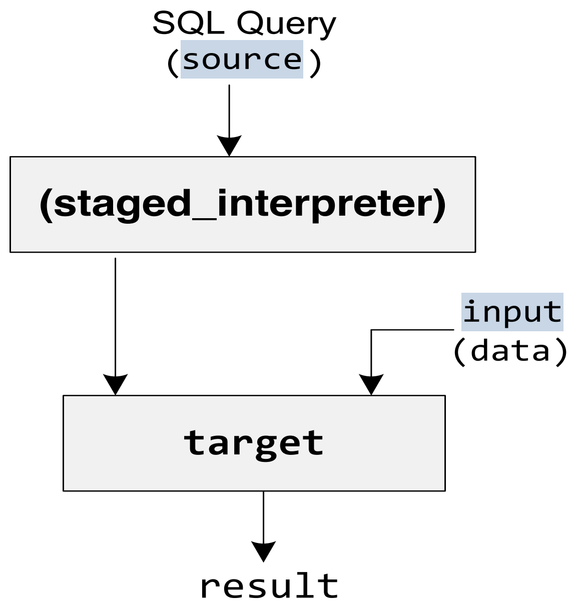
```
target = mix(interpreter, source)
result = target(input)
```



(b)

The first Futamura Projection

```
target = staged_interpreter(source)
result = target(input)
```



(c)

The first Futamura Projection realization through specialization

Automatic partial evaluation is a hard problem due to binding-time analysis (BTA).

Solution: start with a binding-time annotated (staged) program, in a multi-level language.

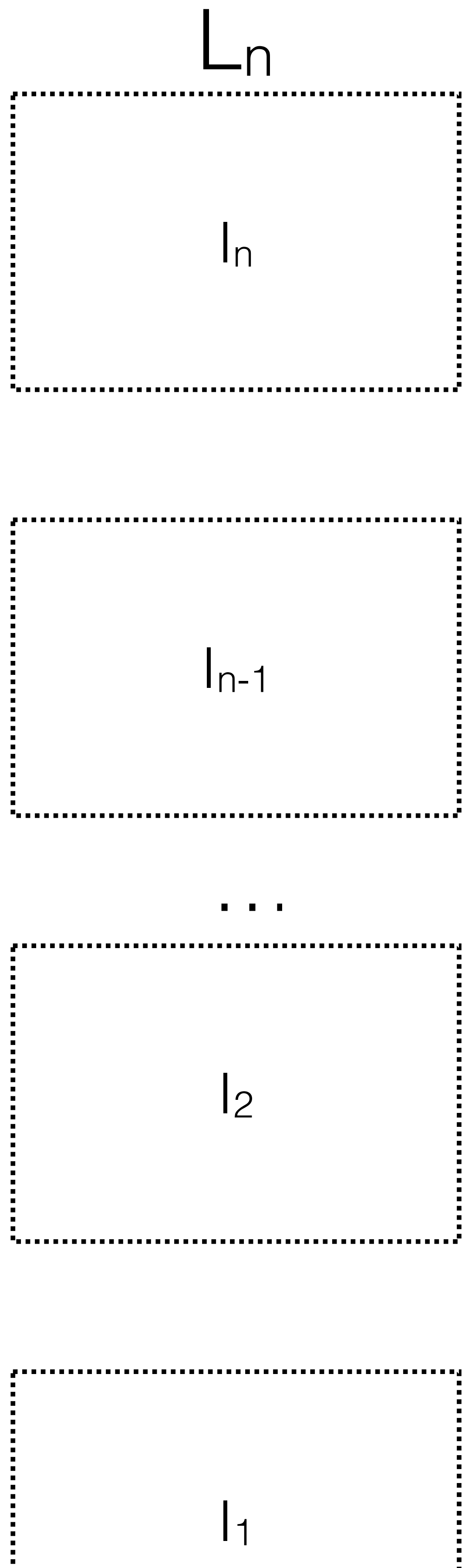
image credit: Ruby Tahboub (Purdue)

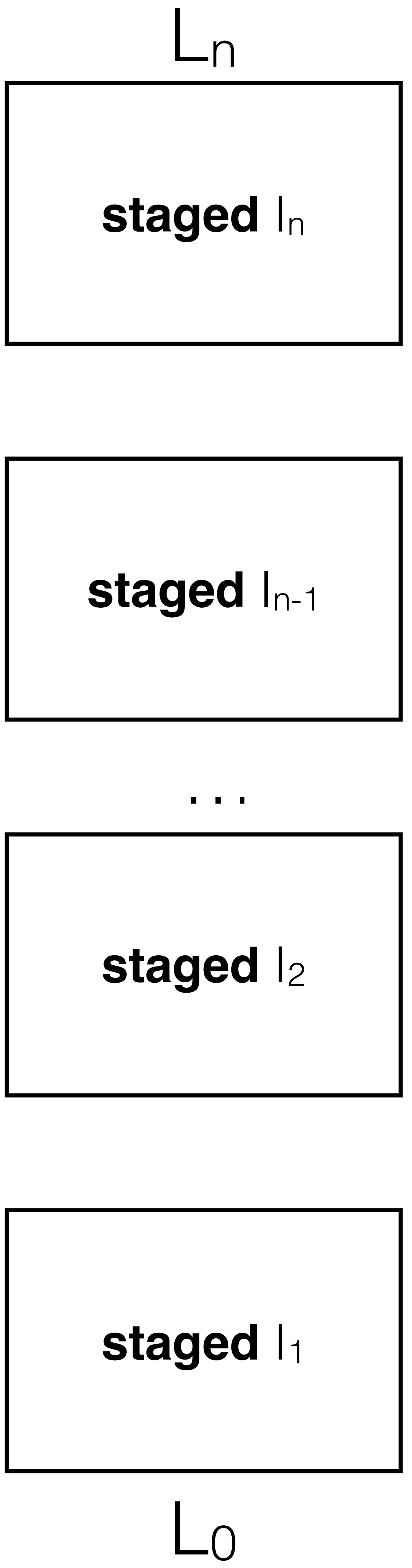
A staged interpreter yields a compiler.

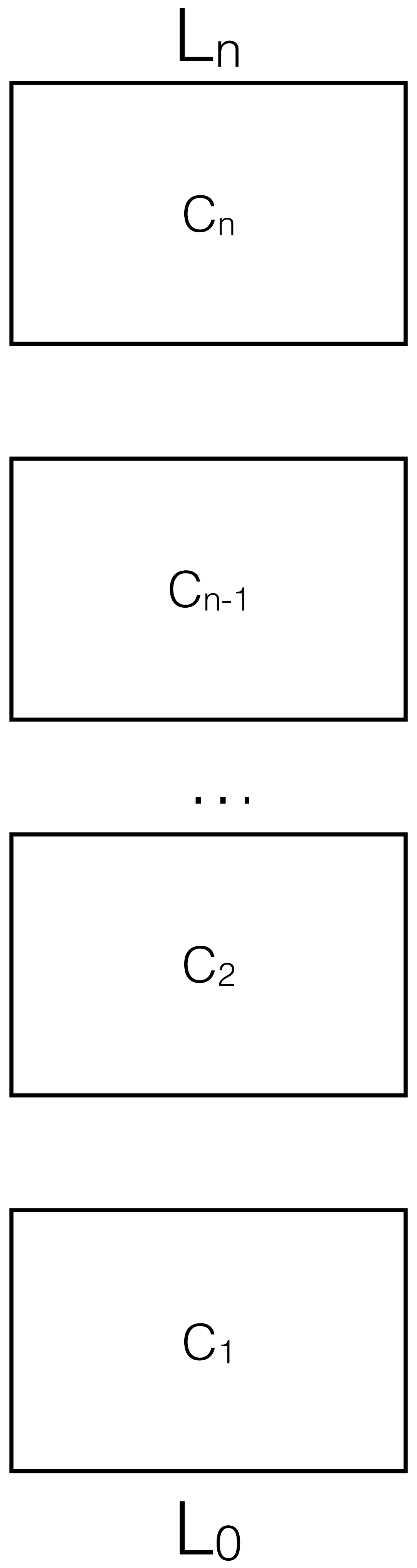
Staging

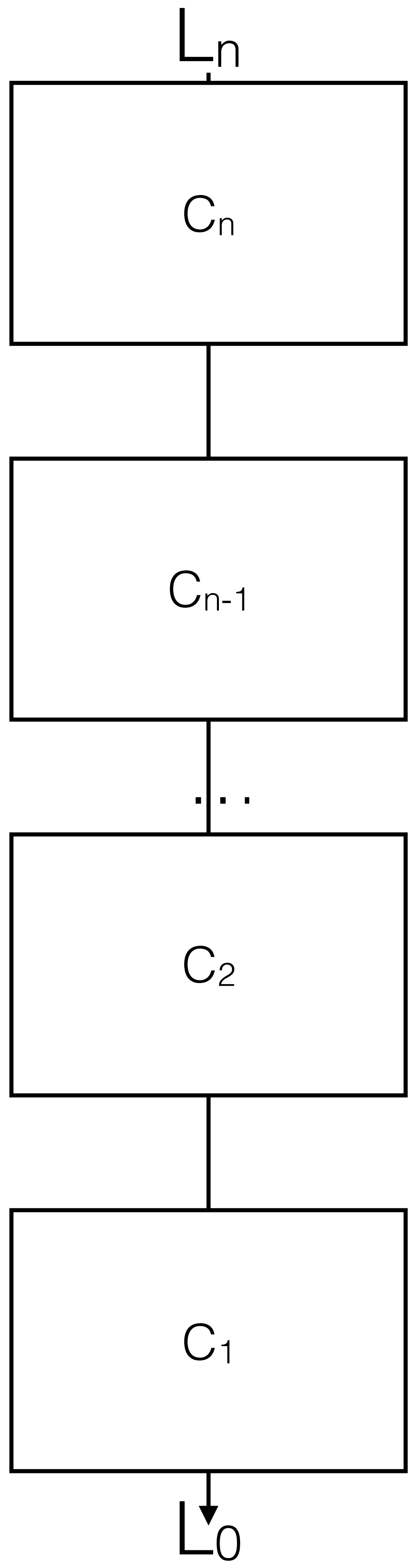
- multi-level language
  `n | x | e @`^{b}` e | λ`^{b}`x.e | …`
- MetaML
  `n | x | e e | λx.e | <e> | ~e | run e`
- Lightweight Modular Staging (LMS) in Scala
  driven by types: `T` vs `Rep[T]`

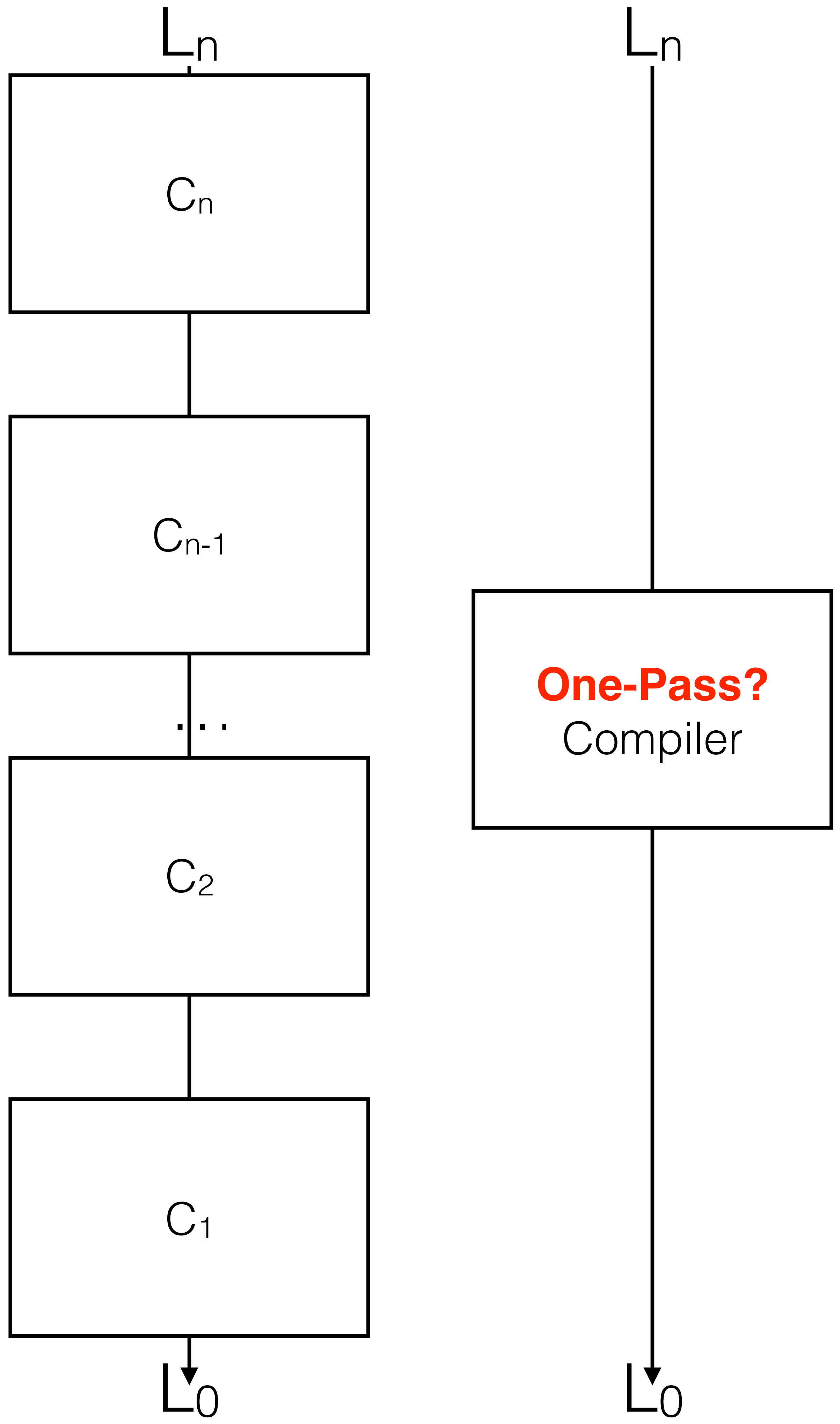
A staged interpreter yields a compiler.

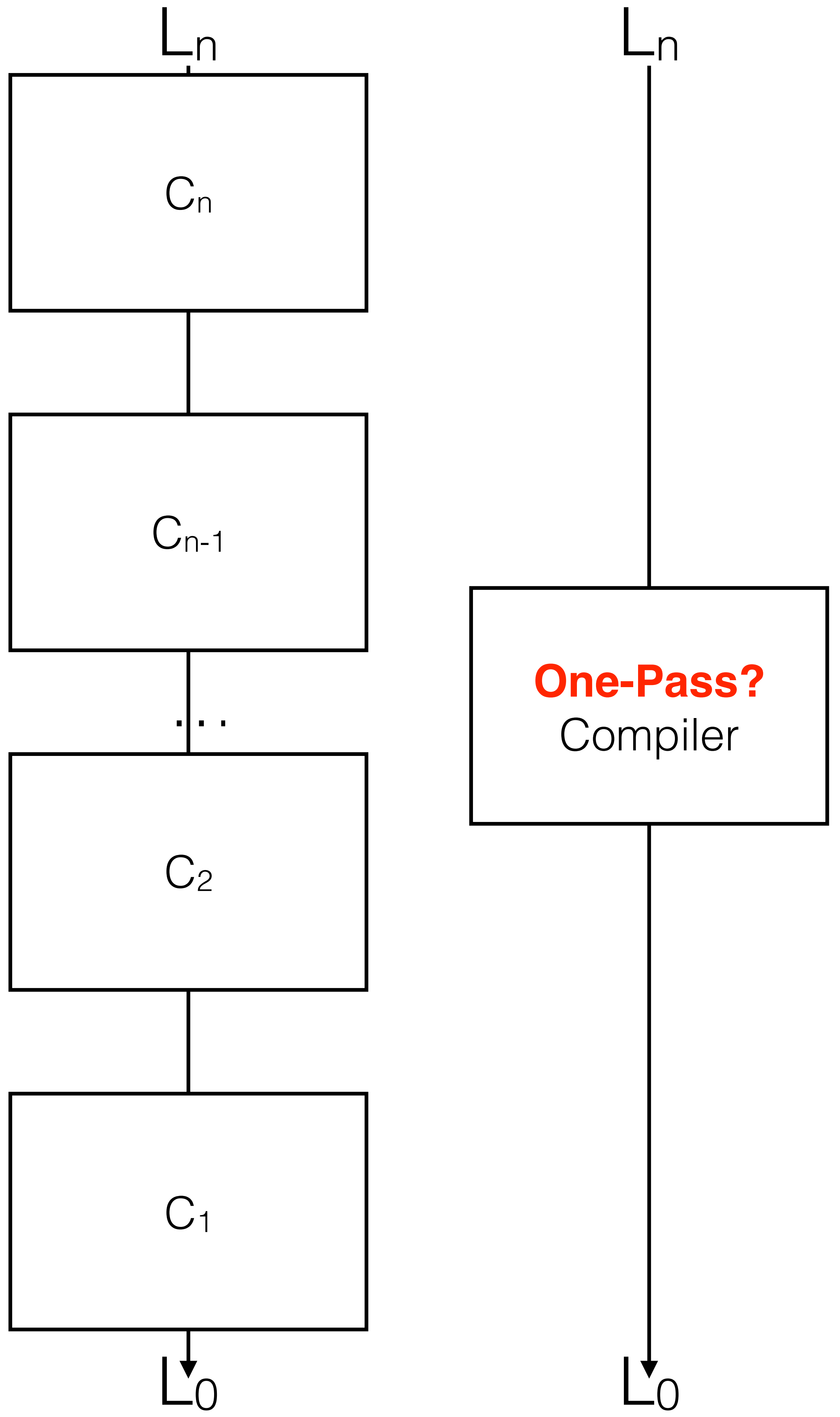

L_n
l_n
l_{n-1}
...
l_2
l_1
L_0


L_n
staged l_n
staged l_{n-1}
...
staged l_2
staged l_1
L_0

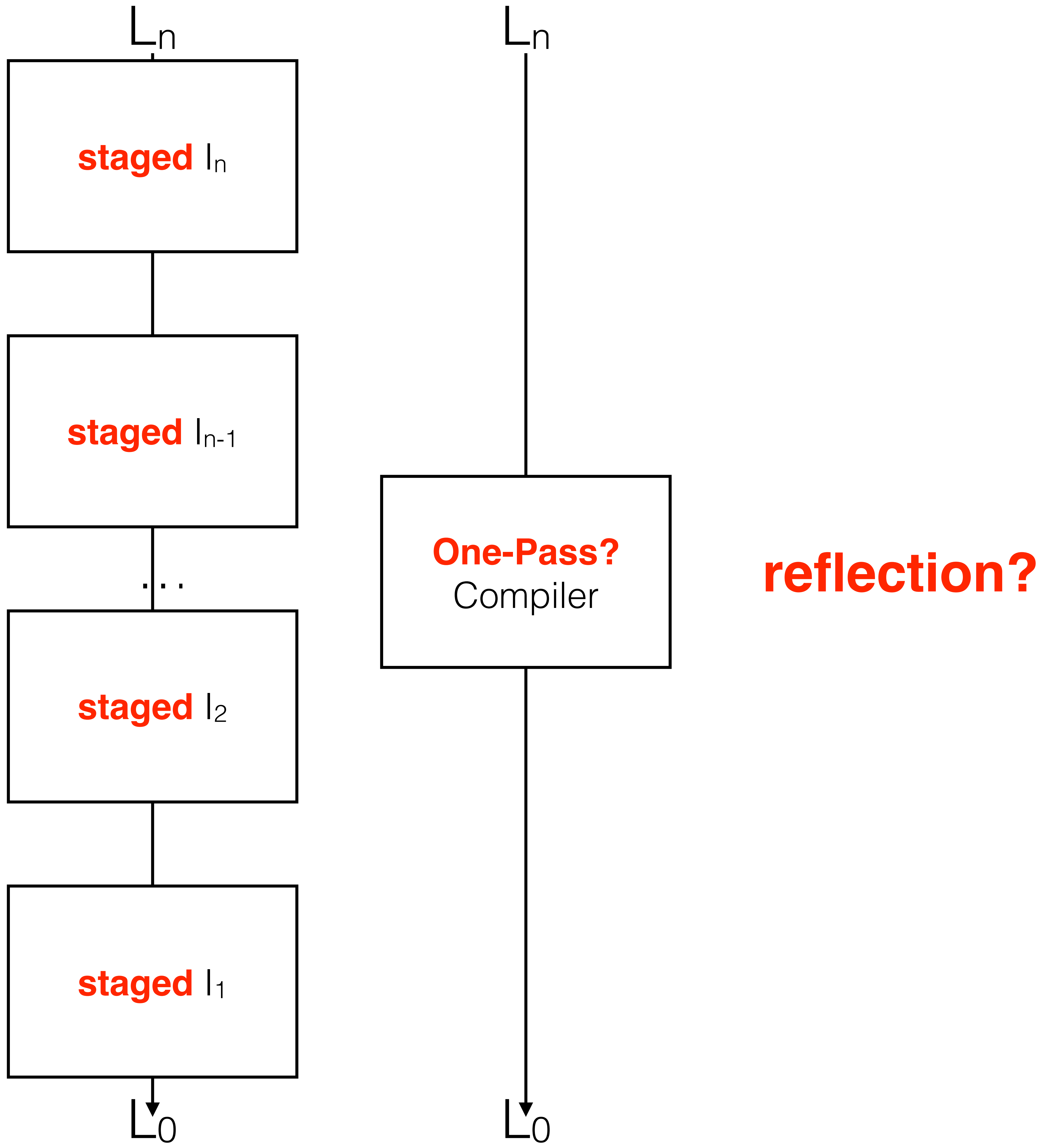

L_n
C_n
C_{n-1}
…
C_2
C_1
L_0


L_n
C_n
C_{n-1}
...
C_2
C_1
L_0

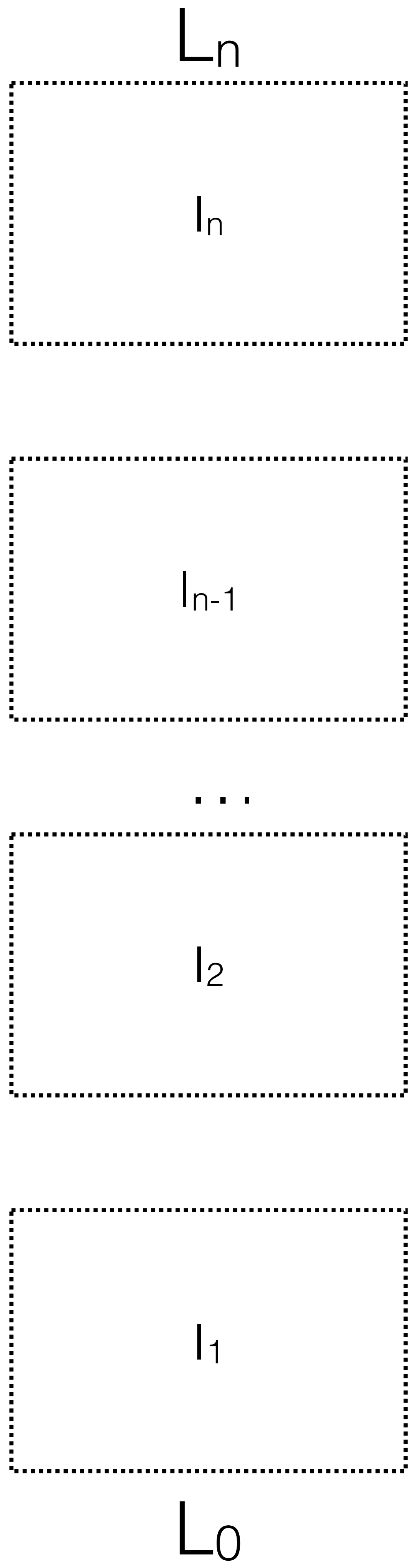

L_n
C_n
C_{n-1}
...
C_2
C_1
L_0
L_n
One-Pass?
Compiler
L_0

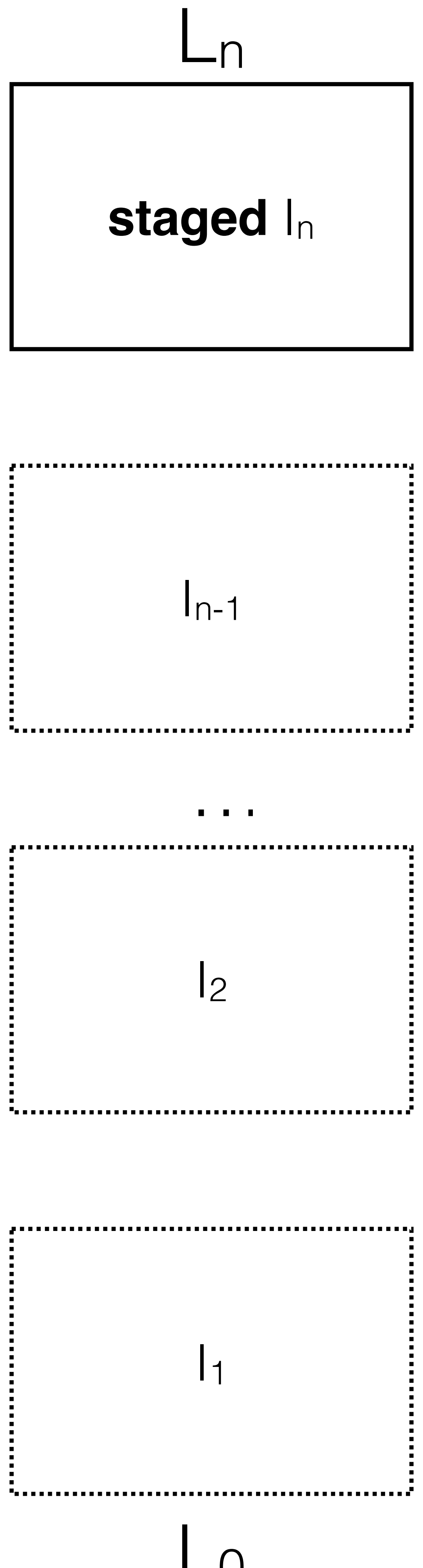


reflection?

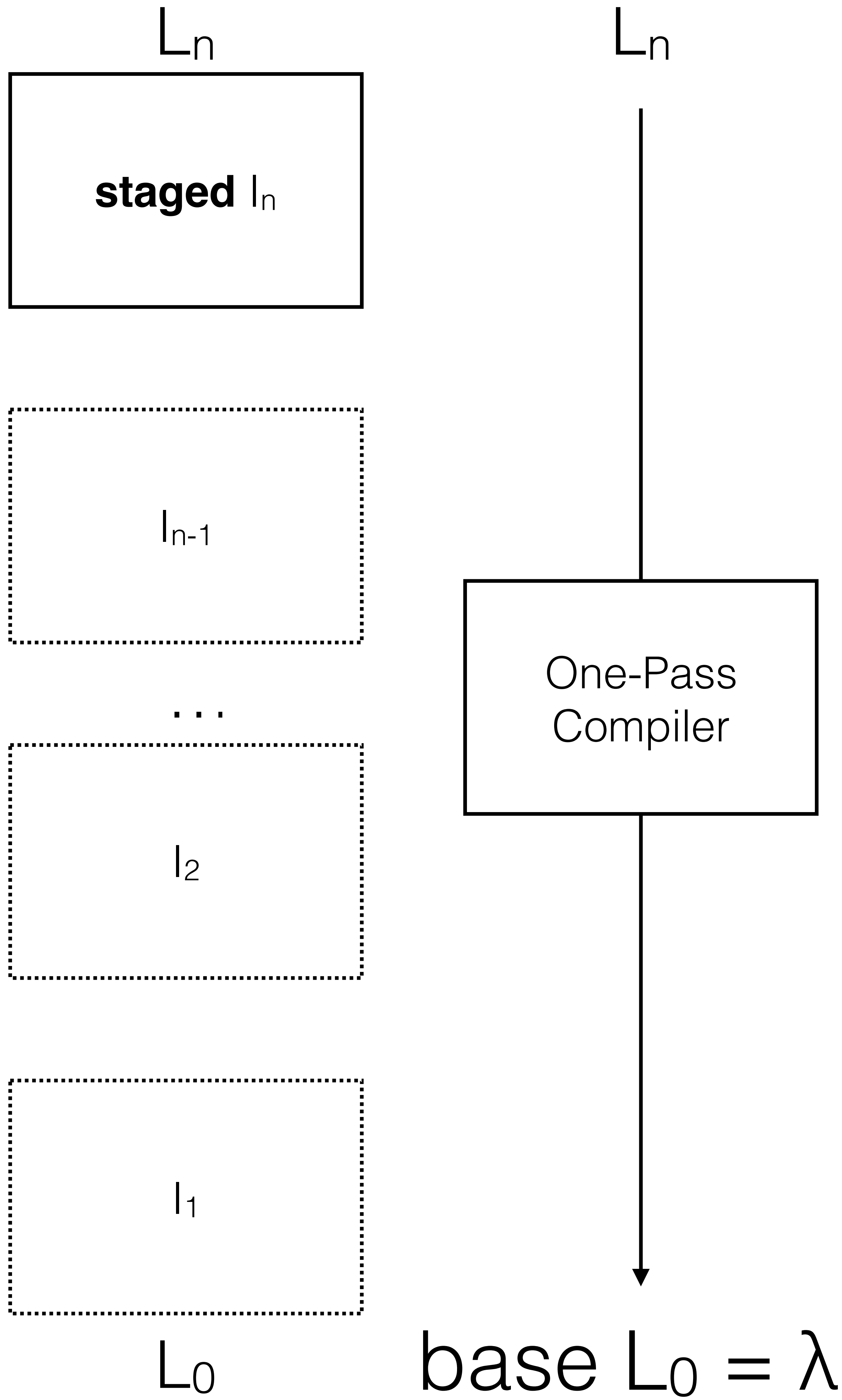

L_n
staged I_n
staged I_{n-1}
. . .
staged I_2
staged I_1
L_0
L_n
One-Pass?
Compiler
L_0
reflection?

Stage Polymorphism

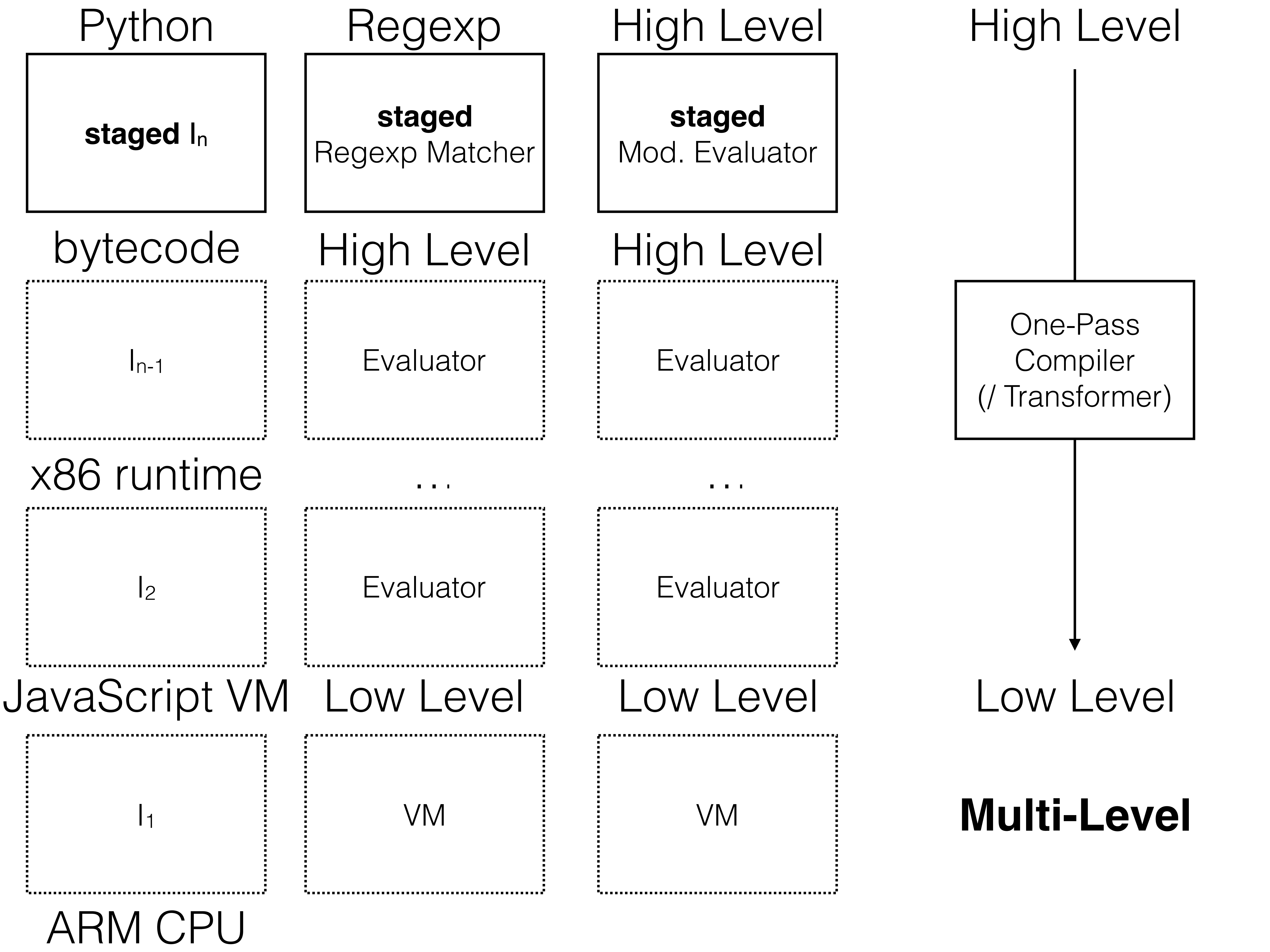

L_n
l_n
l_{n-1}
...
l_2
l_1
L_0

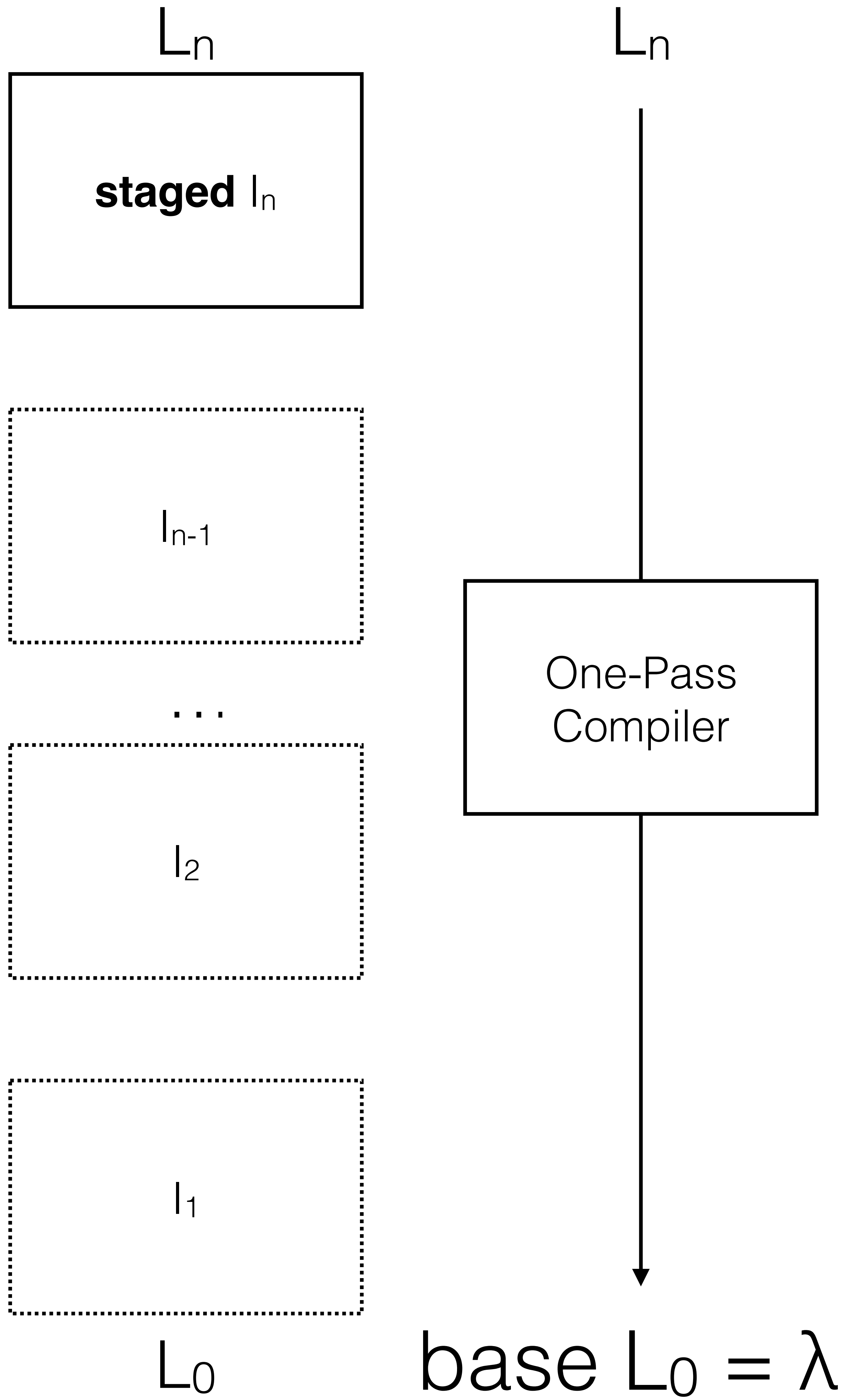


base L_0 = $\lambda_{\uparrow\downarrow}$ = **multi-level** λ-calculus



base $L_0 = \lambda_{\uparrow\downarrow}$ = **multi-level** λ-calculus


Python
staged I_n
bytecode
I_{n-1}
x86 runtime
I_2
JavaScript VM
I_1
ARM CPU
Regexp
staged Regexp Matcher
High Level
Evaluator
...
Evaluator
Low Level
VM
High Level
staged Mod. Evaluator
High Level
Evaluator
...
Evaluator
Low Level
VM
High Level
One-Pass Compiler (/ Transformer)
Low Level
Multi-Level



base $L_0 = \lambda_{\uparrow\downarrow}$ = **multi-level** λ-calculus

Pink & Purple Towers of Interpreters

- abstract over interpretation & compilation: stage-polymorphic multi-level lambda-calculus $\lambda_{\uparrow\downarrow}$ or LMS & polytypic programming via type classes
- control collapse: explicit lifting or compilation unit
- tower size: finite or conceptually infinite
- reflection, mutation, …

$\lambda_{\uparrow\downarrow}$

- Multi-level λ-calculus
- `Lift` operator
- `Let`-insertion
- Stage polymorphism
- Akin to manual *online* partial evaluation

Definitional Interpreter in Scala (or Scheme)

// Multi-stage evaluation

```
def evalms(env: Env, e: Exp): Val = e match {
  case Lit(n)        => Cst(n)
  case Var(n)        => env(n)
  case Cons(e1,e2)   => Tup(evalms(env,e1),evalms(env,e2))
  case Lam(e)        => Clo(env,e)
  case Let(e1,e2)    => val v1 = evalms(env,e1); evalms(env:+v1,e2)
  case App(e1,e2)    => (evalms(env,e1), evalms(env,e2)) match {
    case (Clo(env3,e3), v2)   => evalms(env3:+Clo(env3,e3):+v2,e3)
    case (Code(s1), Code(s2)) => reflectc(App(s1,s2)) }
  case If(c,a,b)     => evalms(env,c) match {
    case Cst(n)        => if (n != 0) evalms(env,a) else evalms(env,b)
    case Code(c1)      => reflectc(If(c1, reifyc(evalms(env,a)), reifyc(evalms(env,b)))) }
  case IsNum(e1)     => evalms(env,e1) match {
    case Code(s1)      => reflectc(IsNum(s1))
    case Cst(n)        => Cst(1)
    case v             => Cst(0) }
  case Plus(e1,e2)   => (evalms(env,e1), evalms(env,e2)) match {
    case (Cst(n1), Cst(n2))  => Cst(n1+n2)
    case (Code(s1),Code(s2)) => reflectc(Plus(s1,s2)) }
  …
  case Lift(e)       => liftc(evalms(env,e))
  case Run(b,e)      => evalms(env,b) match {
    case Code(b1)      => reflectc(Run(b1, reifyc(evalms(env,e))))
    case _             => evalmsg(env, reifyc({ stFresh = env.length; evalms(env, e) })) }
```

Stage Polymorphism

// Multi-stage evaluation

```
def evalms(env: Env, e: Exp): Val = e match {
  case Lit(n)        => Cst(n)
  case Var(n)        => env(n)
  case Cons(e1,e2)   => Tup(evalms(env,e1),evalms(env,e2))
  case Lam(e)        => Clo(env,e)
  case Let(e1,e2)    => val v1 = evalms(env,e1); evalms(env:+v1,e2)
  case App(e1,e2)    => (evalms(env,e1), evalms(env,e2)) match {
    case (Clo(env3,e3), v2)   => evalms(env3:+Clo(env3,e3):+v2,e3)
    case (Code(s1), Code(s2)) => reflectc(App(s1,s2)) }
  case If(c,a,b)     => evalms(env,c) match {
    case Cst(n)        => if (n != 0) evalms(env,a) else evalms(env,b)
    case Code(c1)      => reflectc(If(c1, reifyc(evalms(env,a)), reifyc(evalms(env,b)))) }
  case IsNum(e1)     => evalms(env,e1) match {
    case Code(s1)      => reflectc(IsNum(s1))
    case Cst(n)        => Cst(1)
    case v             => Cst(0) }
  case Plus(e1,e2)   => (evalms(env,e1), evalms(env,e2)) match {
    case (Cst(n1), Cst(n2))  => Cst(n1+n2)
    case (Code(s1),Code(s2)) => reflectc(Plus(s1,s2)) }
  …
  case Lift(e)       => liftc(evalms(env,e))
  case Run(b,e)      => evalms(env,b) match {
    case Code(b1)      => reflectc(Run(b1, reifyc(evalms(env,e))))
    case _             => evalmsg(env, reifyc({ stFresh = env.length; evalms(env, e) })) }
```

// Multi-stage evaluation

```
def evalms(env: Env, e: Exp): Val = e match {
  case Lit(n)        => Cst(n)
  case Var(n)        => env(n)
  case Cons(e1,e2)   => Tup(evalms(env,e1),evalms(env,e2))
  case Lam(e)        => Clo(env,e)
  case Let(e1,e2)    => val v1 = evalms(env,e1); evalms(env:+v1,e2)
  case App(e1,e2)    => (evalms(env,e1), evalms(env,e2)) match {
    case (Clo(env3,e3), v2)   => evalms(env3:+Clo(env3,e3):+v2,e3)
    case (Code(s1), Code(s2)) => reflectc(App(s1,s2)) }
  case If(c,a,b)     => evalms(env,c) match {
    case Cst(n)        => if (n != 0) evalms(env,a) else evalms(env,b)
    case Code(c1)      => reflectc(If(c1, reifyc(evalms(env,a)), reifyc(evalms(env,b)))) }
  case IsNum(e1)     => evalms(env,e1) match {
    case Code(s1)      => reflectc(IsNum(s1))
    case Cst(n)        => Cst(1)
    case v             => Cst(0) }
  case Plus(e1,e2)   => (evalms(env,e1), evalms(env,e2)) match {
    case (Cst(n1), Cst(n2))  => Cst(n1+n2)
    case (Code(s1),Code(s2)) => reflectc(Plus(s1,s2)) }
  …
  case Lift(e)       => liftc(evalms(env,e))
  case Run(b,e)      => evalms(env,b) match {
    case Code(b1)      => reflectc(Run(b1, reifyc(evalms(env,e))))
    case _             => evalmsg(env, reifyc({ stFresh = env.length; evalms(env, e) })) }
```

// Multi-stage evaluation

```
def evalms(env: Env, e: Exp): Val = e match {
  case Lit(n)       => Cst(n)
  case Var(n)       => env(n)
  case Cons(e1,e2)  => Tup(evalms(env,e1),evalms(env,e2))
  case Lam(e)       => Clo(env,e)
  case Let(e1,e2)   => val v1 = evalms(env,e1); evalms(env:+v1,e2)
  case App(e1,e2)   => (evalms(env,e1), evalms(env,e2)) match {
    case (Clo(env3,e3), v2)   => evalms(env3:+Clo(env3,e3):+v2,e3)
    case (Code(s1), Code(s2)) => reflectc(App(s1,s2)) }
  case If(c,a,b)    => evalms(env,c) match {
    case Cst(n)       => if (n != 0) evalms(env,a) else evalms(env,b)
    case Code(c1)     => reflectc(If(c1, reifyc(evalms(env,a)), reifyc(evalms(env,b)))) }
  case IsNum(e1)    => evalms(env,e1) match {
    case Code(s1)     => reflectc(IsNum(s1))
    case Cst(n)       => Cst(1)
    case v            => Cst(0) }
  case Plus(e1,e2)  => (evalms(env,e1), evalms(env,e2)) match {
    case (Cst(n1), Cst(n2))  => Cst(n1+n2)
    case (Code(s1),Code(s2)) => reflectc(Plus(s1,s2)) }
  …
  case Lift(e)      => liftc(evalms(env,e))
  case Run(b,e)     => evalms(env,b) match {
    case Code(b1)     => reflectc(Run(b1, reifyc(evalms(env,e))))
    case _            => evalmsg(env, reifyc({ stFresh = env.length; evalms(env, e) })) }
```

Lift

```
def lift(v: Val): Exp = v match {
  case Cst(n)        => Lit(n)
  case Tup(a,b)      => val (Code(u),Code(v))=(a,b);
    reflect(Cons(u,v))
  case Clo(env2,e2) => reflect(Lam(reifyc(evalms(
    env2:+Code(fresh()):+Code(fresh()),e2))))
  case Code(e)       => reflect(Lift(e)) }

def liftc(v: Val) = Code(lift(v))
```

Lift

```
def lift(v: Val): Exp = v match {
  case Cst(n)        => Lit(n)
  case Tup(a,b)      => val (Code(u),Code(v))=(a,b);
    reflect(Cons(u,v))
  case Clo(env2,e2) => reflect(Lam(reifyc(evalms(
    env2:+Code(fresh()):+Code(fresh()),e2))))
  case Code(e)       => reflect(Lift(e)) }

def liftc(v: Val) = Code(lift(v))
```

Lift

```
def lift(v: Val): Exp = v match {
  case Cst(n)         => Lit(n)
  case Tup(a,b)       => val (Code(u),Code(v))=(a,b);
    reflect(Cons(u,v))
  case Clo(env2,e2) => reflect(Lam(reifyc(evalms(
    env2:+Code(fresh()):+Code(fresh()),e2))))
  case Code(e)        => reflect(Lift(e)) }

def liftc(v: Val) = Code(lift(v))
```

Lift

```
def lift(v: Val): Exp = v match {
  case Cst(n)         => Lit(n)
  case Tup(a,b)       => val (Code(u),Code(v))=(a,b);
    reflect(Cons(u,v))
  case Clo(env2,e2)  => reflect(Lam(reifyc(evalms(
    env2:+Code(fresh()):+Code(fresh()),e2))))
  case Code(e)        => reflect(Lift(e)) }

def liftc(v: Val) = Code(lift(v))
```

API for Let-Insertion

```
var stFresh: Int       = 0
var stBlock: List[Exp] = Nil
def fresh()            = {stFresh += 1; Var(stFresh-1)}
def run[A](f: => A): A = {val sF = stFresh; val sB = stBlock; try f finally {stFresh = sF; stBlock = sB}}
```

```
def reify(f: => Exp) = run{stBlock = Nil; val last = f;
  (stBlock foldRight last)(Let)}
```

```
def reflect(s:Exp)   = {stBlock :+= s; fresh()}
```

```
def reifyc(f: => Val)  = reify{val Code(e) = f; e}
def reflectc(s: Exp)   = Code(reflect(s))
def reifyv(f: => Val)  = run{stBlock = Nil; val res = f; if (stBlock == Nil) res else {
  val Code(last) = res; Code((stBlock foldRight last)(Let))}}
```

API for Let-Insertion

```
var stFresh: Int       = 0
var stBlock: List[Exp] = Nil
def fresh()            = {stFresh += 1; Var(stFresh-1)}
def run[A](f: => A): A = {val sF = stFresh; val sB = stBlock; try f finally {stFresh = sF; stBlock = sB}}
```

```
def reify(f: => Exp) = run{stBlock = Nil; val last = f;
  (stBlock foldRight last)(Let)}
```

```
def reflect(s:Exp)   = {stBlock :+= s; fresh()}
```

```
def reifyc(f: => Val)  = reify{val Code(e) = f; e}
def reflectc(s: Exp)   = Code(reflect(s))
def reifyv(f: => Val)  = run{stBlock = Nil; val res = f; if (stBlock == Nil) res else {
  val Code(last) = res; Code((stBlock foldRight last)(Let))}}
```

API for Let-Insertion

```
var stFresh: Int       = 0
var stBlock: List[Exp] = Nil
def fresh()            = {stFresh += 1; Var(stFresh-1)}
def run[A](f: => A): A = {val sF = stFresh; val sB = stBlock; try f finally {stFresh = sF; stBlock = sB}}
```

```
def reify(f: => Exp) = run{stBlock = Nil; val last = f;
  (stBlock foldRight last)(Let)}
```

```
def reflect(s:Exp)   = {stBlock :+= s; fresh()}
```

```
def reifyc(f: => Val)  = reify{val Code(e) = f; e}
def reflectc(s: Exp)   = Code(reflect(s))
def reifyv(f: => Val)  = run{stBlock = Nil; val res = f; if (stBlock == Nil) res else {
  val Code(last) = res; Code((stBlock foldRight last)(Let))}}
```

$\lambda_{\uparrow\downarrow}$

- Multi-level λ-calculus
- `Lift` operator
- `Let`-insertion
- Stage polymorphism

Pink: Stage-Polymorphic Meta-Circular Evaluator

```
;; Stage-Polymorphic Meta-Circular Evaluator for Pink

(lambda _ maybe-lift (lambda _ eval (lambda _ exp (lambda _ env
 (if (num?                exp)  (maybe-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?  '-      (car exp))  (-   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '*      (car exp))  (*   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'eq?    (car exp))  (eq? ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'if     (car exp))  (if  ((eval (cadr exp)) env) ((eval (caddr exp)) env)
                                                             ((eval (cadddr exp)) env))
  (if (eq?  'lambda (car exp))  (maybe-lift (lambda f x ((eval (cadddr exp))
   (lambda _ y (if (eq? y (cadr exp)) f (if (eq? y (caddr exp)) x (env y)))))))
  (if (eq?  'let    (car exp))  (let x ((eval (caddr exp)) env) ((eval (cadddr exp))
   (lambda _ y (if (eq?  y (cadr exp)) x (env y)))))
  (if (eq?  'lift   (car exp))  (lift ((eval (cadr exp)) env))
  (if (eq?  'run    (car exp))  (run  ((eval (cadr exp)) env) ((eval (caddr exp))env))
  (if (eq?  'num?   (car exp))  (num? ((eval (cadr exp)) env))
  (if (eq?  'sym?   (car exp))  (sym? ((eval (cadr exp)) env))
  (if (eq?  'car    (car exp))  (car  ((eval (cadr exp)) env))
  (if (eq?  'cdr    (car exp))  (cdr  ((eval (cadr exp)) env))
  (if (eq?  'cons   (car exp))  (maybe-lift (cons ((eval (cadr exp)) env)
                                                  ((eval (caddr exp)) env)))
  (if (eq?  'quote  (car exp))  (maybe-lift (cadr exp))
  ((env (car exp)) ((eval (cadr exp)) env)))))))))))))))))
 (((eval (car exp)) env) ((eval (cadr exp)) env)))))))))
```

;; Stage-Polymorphic Meta-Circular Evaluator for Pink

```
(lambda _ maybe-lift (lambda _ eval (lambda _ exp (lambda _ env
 (if (num?                exp)  (maybe-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?  '-      (car exp))  (-   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '*      (car exp))  (*   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'eq?    (car exp))  (eq? ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'if     (car exp))  (if  ((eval (cadr exp)) env) ((eval (caddr exp)) env)
                                                             ((eval (cadddr exp)) env))
  (if (eq?  'lambda (car exp))  (maybe-lift (lambda f x ((eval (cadddr exp))
   (lambda _ y (if (eq? y (cadr exp)) f (if (eq? y (caddr exp)) x (env y)))))))
  (if (eq?  'let    (car exp))  (let x ((eval (caddr exp)) env) ((eval (cadddr exp))
   (lambda _ y (if (eq?  y (cadr exp)) x (env y)))))
  (if (eq?  'lift   (car exp))  (lift ((eval (cadr exp)) env))
  (if (eq?  'run    (car exp))  (run  ((eval (cadr exp)) env) ((eval (caddr exp))env))
  (if (eq?  'num?   (car exp))  (num? ((eval (cadr exp)) env))
  (if (eq?  'sym?   (car exp))  (sym? ((eval (cadr exp)) env))
  (if (eq?  'car    (car exp))  (car  ((eval (cadr exp)) env))
  (if (eq?  'cdr    (car exp))  (cdr  ((eval (cadr exp)) env))
  (if (eq?  'cons   (car exp))  (maybe-lift (cons ((eval (cadr exp)) env)
                                                  ((eval (caddr exp)) env)))
  (if (eq?  'quote  (car exp))  (maybe-lift (cadr exp))
  ((env (car exp)) ((eval (cadr exp)) env)))))))))))))))))
 (((eval (car exp)) env) ((eval (cadr exp)) env)))))))))
```

;; Stage-Polymorphic Meta-Circular Evaluator for Pink

```
(lambda _ maybe-lift (lambda _ eval (lambda _ exp (lambda _ env
 (if (num?                exp)  (maybe-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?  '-      (car exp))  (-   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '*      (car exp))  (*   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'eq?    (car exp))  (eq? ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'if     (car exp))  (if  ((eval (cadr exp)) env) ((eval (caddr exp)) env)
                                                             ((eval (cadddr exp)) env))
  (if (eq?  'lambda (car exp))  (maybe-lift (lambda f x ((eval (cadddr exp))
   (lambda _ y (if (eq? y (cadr exp)) f (if (eq? y (caddr exp)) x (env y)))))))
  (if (eq?  'let    (car exp))  (let x ((eval (caddr exp)) env) ((eval (cadddr exp))
   (lambda _ y (if (eq?  y (cadr exp)) x (env y)))))
  (if (eq?  'lift   (car exp))  (lift ((eval (cadr exp)) env))
  (if (eq?  'run    (car exp))  (run  ((eval (cadr exp)) env) ((eval (caddr exp))env))
  (if (eq?  'num?   (car exp))  (num? ((eval (cadr exp)) env))
  (if (eq?  'sym?   (car exp))  (sym? ((eval (cadr exp)) env))
  (if (eq?  'car    (car exp))  (car  ((eval (cadr exp)) env))
  (if (eq?  'cdr    (car exp))  (cdr  ((eval (cadr exp)) env))
  (if (eq?  'cons   (car exp))  (maybe-lift (cons ((eval (cadr exp)) env)
                                                  ((eval (caddr exp)) env)))
  (if (eq?  'quote  (car exp))  (maybe-lift (cadr exp))
  ((env (car exp)) ((eval (cadr exp)) env)))))))))))))))
 (((eval (car exp)) env) ((eval (cadr exp)) env)))))))))
```

;; Stage-Polymorphic Meta-Circular Evaluator for Pink

```
(lambda _ maybe-lift (lambda _ eval (lambda _ exp (lambda _ env
 (if (num?                exp)  (maybe-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?  '-      (car exp))  (-   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '*      (car exp))  (*   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'eq?    (car exp))  (eq? ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'if     (car exp))  (if  ((eval (cadr exp)) env) ((eval (caddr exp)) env)
                                                             ((eval (cadddr exp)) env))
  (if (eq?  'lambda (car exp))  (maybe-lift (lambda f x ((eval (cadddr exp))
   (lambda _ y (if (eq? y (cadr exp)) f (if (eq? y (caddr exp)) x (env y)))))))
  (if (eq?  'let    (car exp))  (let x ((eval (caddr exp)) env) ((eval (cadddr exp))
   (lambda _ y (if (eq?  y (cadr exp)) x (env y)))))
  (if (eq?  'lift   (car exp))  (lift ((eval (cadr exp)) env))
  (if (eq?  'run    (car exp))  (run  ((eval (cadr exp)) env) ((eval (caddr exp))env))
  (if (eq?  'num?   (car exp))  (num? ((eval (cadr exp)) env))
  (if (eq?  'sym?   (car exp))  (sym? ((eval (cadr exp)) env))
  (if (eq?  'car    (car exp))  (car  ((eval (cadr exp)) env))
  (if (eq?  'cdr    (car exp))  (cdr  ((eval (cadr exp)) env))
  (if (eq?  'cons   (car exp))  (maybe-lift (cons ((eval (cadr exp)) env)
                                                  ((eval (caddr exp)) env)))
  (if (eq?  'quote  (car exp))  (maybe-lift (cadr exp))
  ((env (car exp)) ((eval (cadr exp)) env)))))))))))))))))
 (((eval (car exp)) env) ((eval (cadr exp)) env)))))))))
```

;; Stage-Polymorphic Meta-Circular Evaluator for Pink

```
(lambda _ maybe-lift (lambda _ eval (lambda _ exp (lambda _ env
 (if (num?                exp)  (maybe-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?  '-      (car exp))  (-   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '*      (car exp))  (*   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'eq?    (car exp))  (eq? ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'if     (car exp))  (if  ((eval (cadr exp)) env) ((eval (caddr exp)) env)
                                                             ((eval (cadddr exp)) env))
  (if (eq?  'lambda (car exp))  (maybe-lift (lambda f x ((eval (cadddr exp))
   (lambda _ y (if (eq? y (cadr exp)) f (if (eq? y (caddr exp)) x (env y)))))))
  (if (eq?  'let    (car exp))  (let x ((eval (caddr exp)) env) ((eval (cadddr exp))
   (lambda _ y (if (eq?  y (cadr exp)) x (env y)))))
  (if (eq?  'lift   (car exp))  (lift ((eval (cadr exp)) env))
  (if (eq?  'run    (car exp))  (run  ((eval (cadr exp)) env) ((eval (caddr exp))env))
  (if (eq?  'num?   (car exp))  (num? ((eval (cadr exp)) env))
  (if (eq?  'sym?   (car exp))  (sym? ((eval (cadr exp)) env))
  (if (eq?  'car    (car exp))  (car  ((eval (cadr exp)) env))
  (if (eq?  'cdr    (car exp))  (cdr  ((eval (cadr exp)) env))
  (if (eq?  'cons   (car exp))  (maybe-lift (cons ((eval (cadr exp)) env)
                                                  ((eval (caddr exp)) env)))
  (if (eq?  'quote  (car exp))  (maybe-lift (cadr exp))
  ((env (car exp)) ((eval (cadr exp)) env)))))))))))))))))
 (((eval (car exp)) env) ((eval (cadr exp)) env)))))))))
```

Pink Interpretation

```
(define eval ((lambda ev e
  (((eval-poly (lambda _ e e)) ev) e))
  #nil)))

(define fac-src (quote (lambda f n
  (if (eq? n 0) 1 (* n (f (- n 1))))))))

> ((eval fac-src) 4) ;=> 24
```

Double & Triple Pink Interpretation

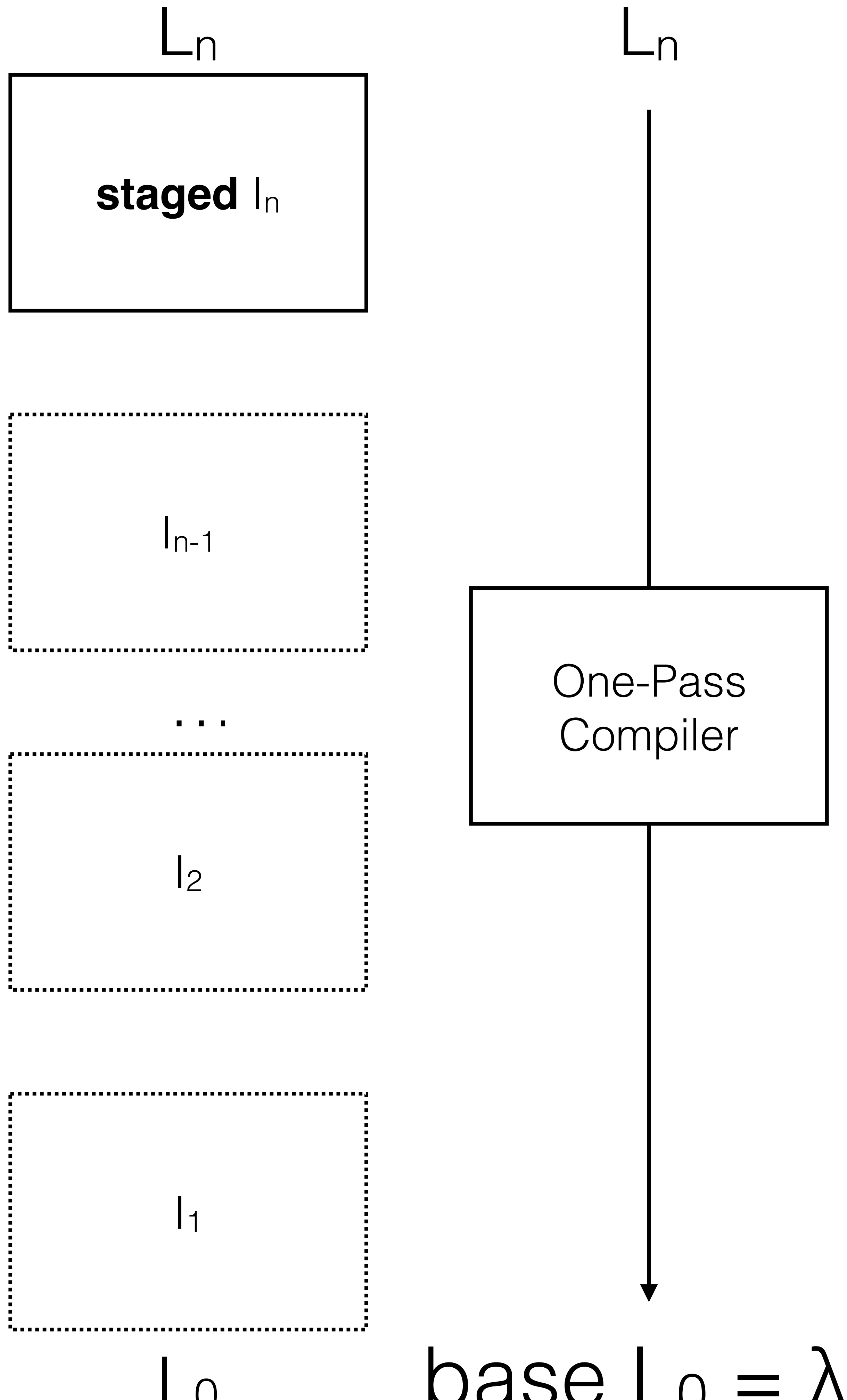
```
> ((eval fac-src) 4)

> (((eval eval-src) fac-src) 4)

> ((((eval eval-src) eval-src) fac-src) 4)

;=>24
```

Pink Compilation

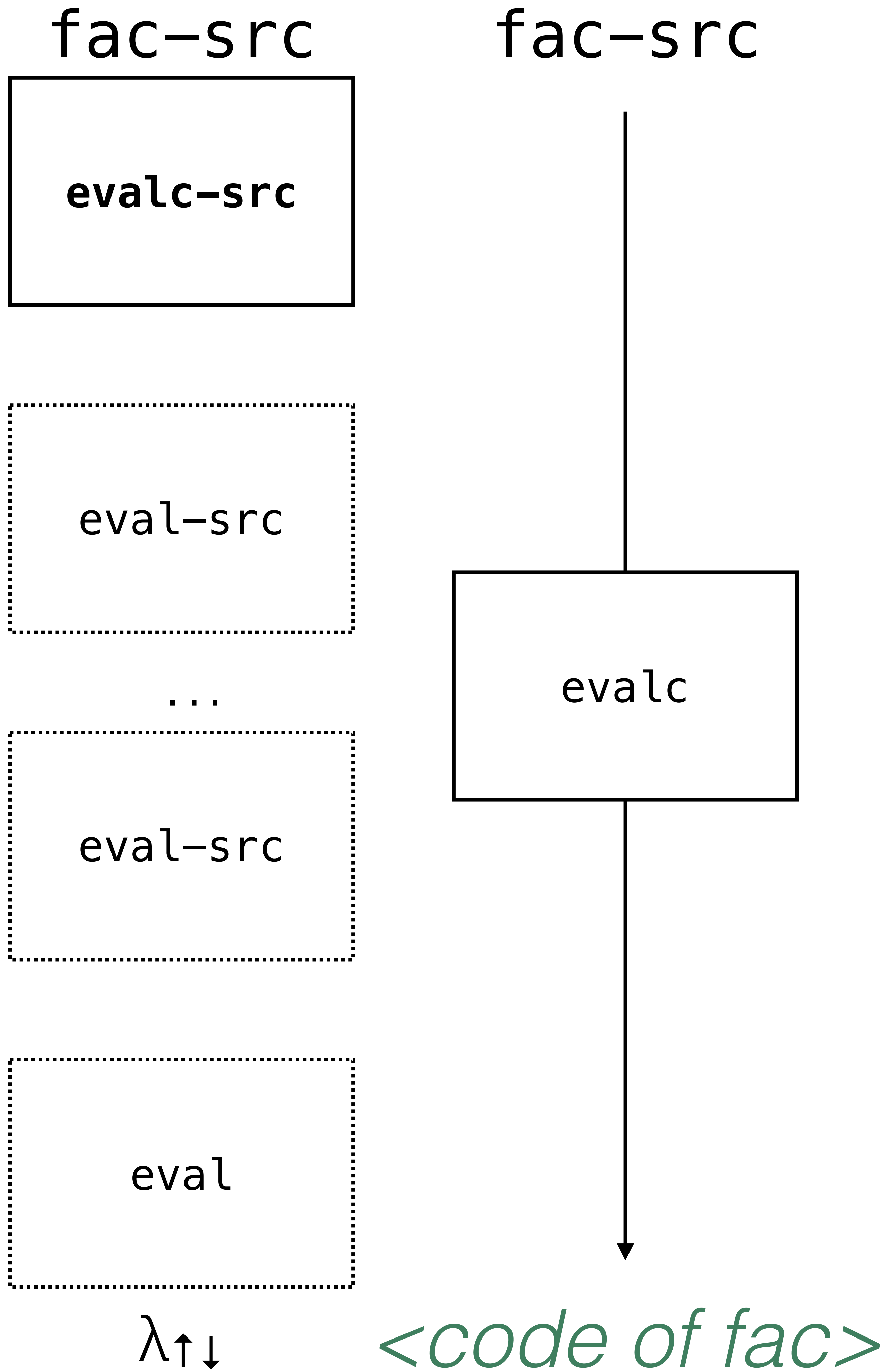
```
(define evalc ((lambda ev e
  (((eval-poly (lambda _ e (lift e)))
  ev) e)) #nil)))

(define fac-src (quote (lambda f n
  (if (eq? n 0) 1 (* n (f (- n 1)))))))

> (evalc fac-src) ;=> <code of fac>
```

Pink Collapsing

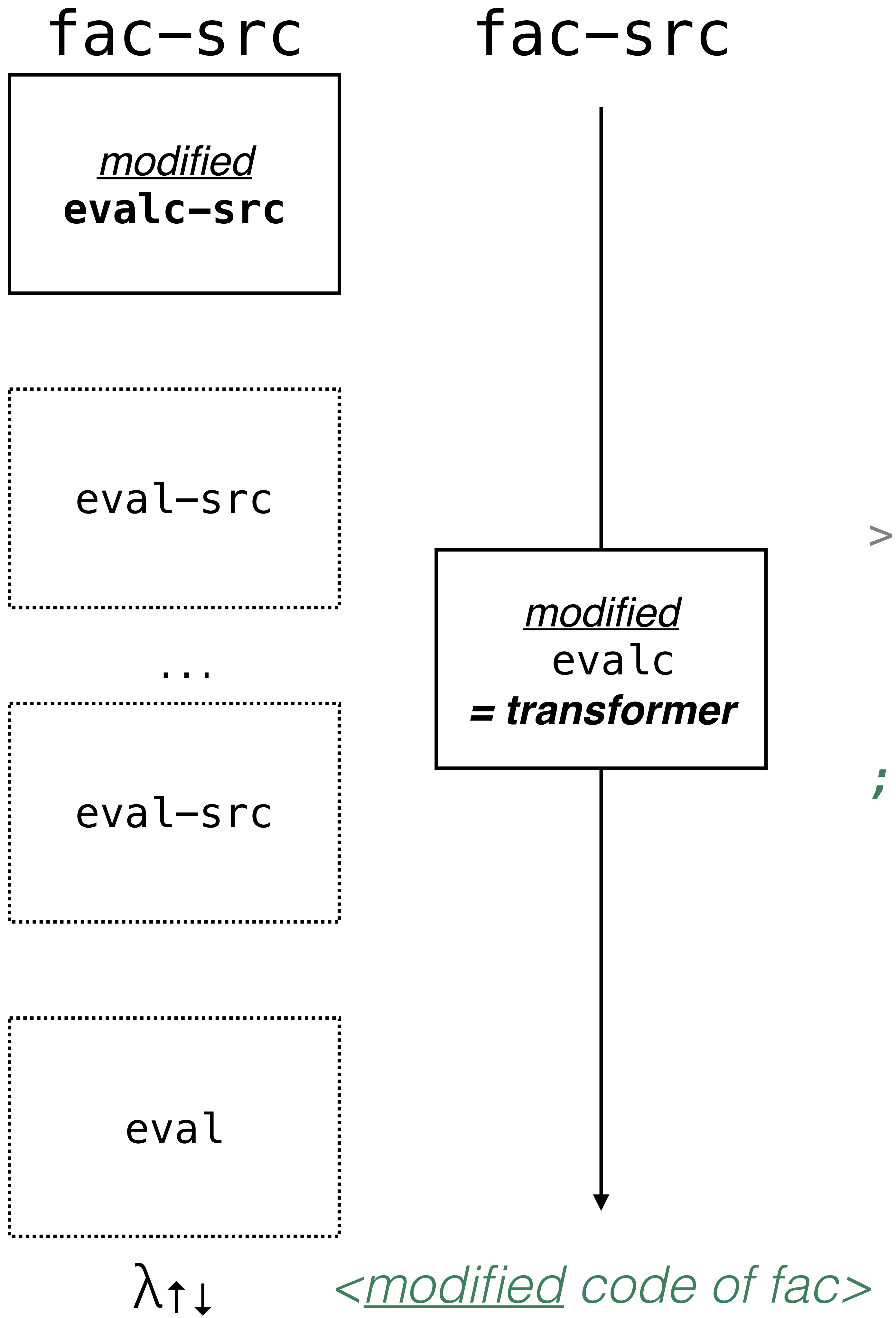
```
> (evalc fac-src)

> ((eval evalc-src) fac-src)

> ((eval eval-src) evalc-src) fac-src)

;=> <code of fac>
```



base L_0 = $\lambda_{\uparrow\downarrow}$ = **multi-level** λ-calculus



```
> (((eval eval-src) … eval-src)
   evalc-src) fac-src)
;=> <code of fac>
```



```
> (((eval eval-src) … eval-src)
   modified-evalc-src) fac-src)
;=> <code of fac>
```

Pink Transformers

```
> (evalc fac-src) ;; =>
(lambda f0 x1
  (let x2 (eq? x1 0)
  (if x2 1
    (let x3 (- x1 1)
    (let x4 (f0 x3)
    (* x1 x4))))))
```

```
> (trace-n-evalc fac-src) ;; =>
(lambda f0 x1
  (let x2 (log 0 x1)
  (let x3 (eq? x2 0)
  (if x3 1
    (let x4 (log 0 x1)
    (let x5 (log 0 x1)
    (let x6 (- x5 1)
    (let x7 (f0 x6)
    (* x4 x7)))))))))
```

```
> (cps-evalc fac-src) ;; =>
(lambda f0 x1 (lambda f2 x3
  (let x4 (eq? x1 0)
  (if x4 (x3 1)
    (let x5 (- x1 1)
    (let x6 (f0 x5)
    (let x7 (lambda f7 x8
      (let x9 (* x1 x8) (x3 x9)))
    (x6 x7))))))))
```

Purple

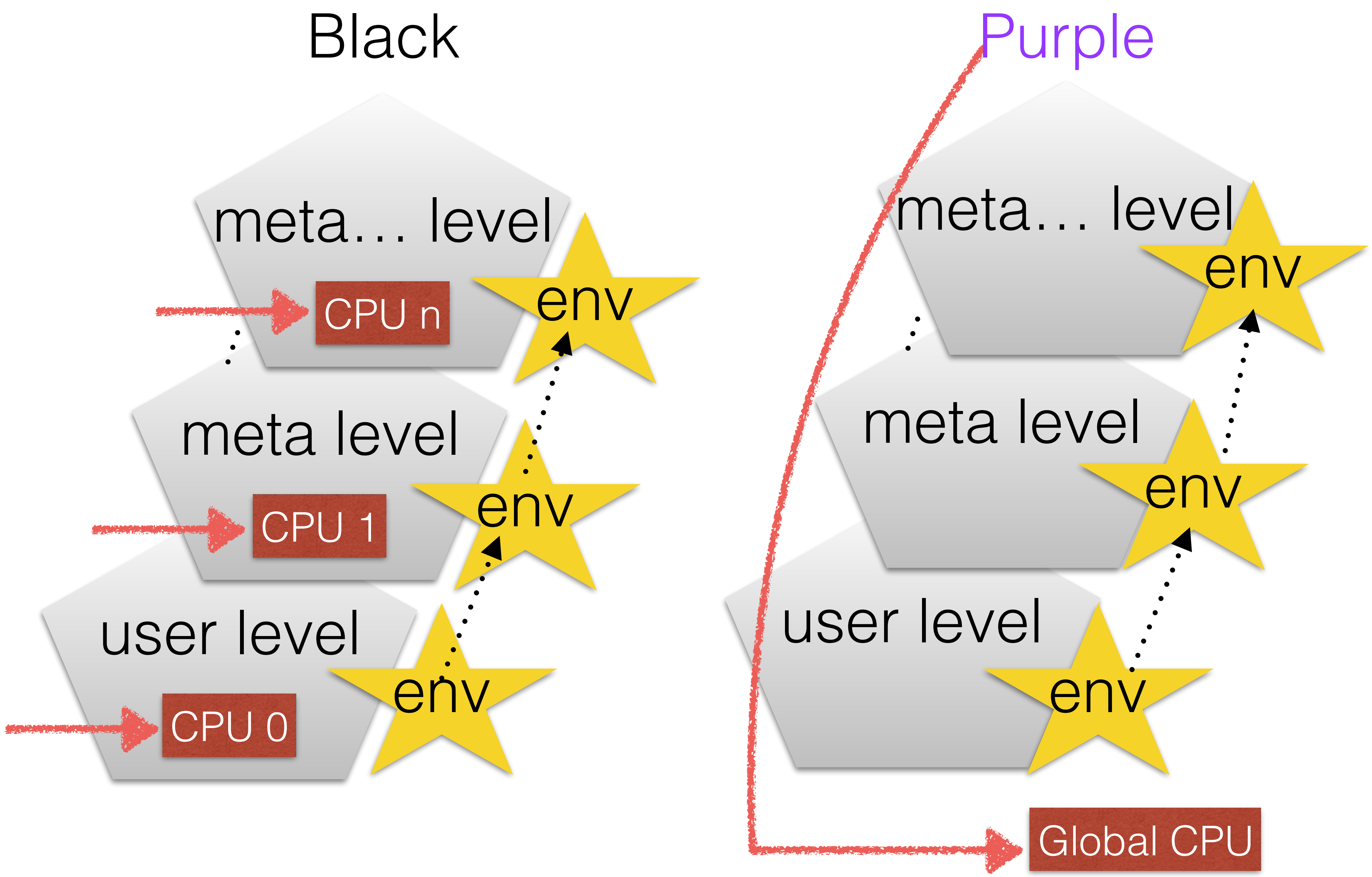
- unit of compilation: **`lambda`** becomes **`clambda`**
- conceptually infinite reflective tower based on Black
- Lightweight Modular Staging (LMS)
  - roughly akin to manual *offline* partial evaluation
  - staged polymorphism through type classes: `Rep[T]` vs `NoRep[T]` (= T) abstracted as `R[T]`

Recall: Collapse in Purple

```
{(k, xs) => _app(‘+, _cons(_cell_read(<cell counter>), ‘(1)),
_cont{c_1 => _cell_set(<cell counter>, c_1) _app(‘<, _cons(_car(xs),
‘(2)), _cont{v_1 => _if(_true(v_1),
_app(‘+, _cons(_cell_read(<cell counter>), ‘(1)), _cont{c_2 =>
_cell_set(<cell counter>, c_2)
_app(k, _cons(_car(xs), ‘()), _cont{v_2 => v_2})})}),
_app(‘+, _cons(_cell_read(<cell counter>), ‘(1)), _cont{c_3 =>
_cell_set(<cell counter>, c_3)
_app(‘-, _cons(_car(xs), ‘(1)), _cont{v_3 => _app(_cell_read(<cell
fib>), _cons(v_3, ‘()), _cont{v_4 =>
_app(‘+, _cons(_cell_read(<cell counter>), ‘(1)), _cont{c_4 =>
_cell_set(<cell counter>, c_4)
_app(‘-, _cons(_car(xs), ‘(2)), _cont{v_5 => _app(_cell_read(<cell
fib>), _cons(v_5, ‘()), _cont{v_6 =>
_app(‘+, _cons(v_4, _cons(v_6, ‘())), _cont{v_7 => _app(k,
_cons(v_7, ‘()), _cont{v_8 => v_8})})})})})})})})})})})})}
```

```scala
trait Ops[R[_]] {
  implicit def _lift(v: Value): R[Value]
  def _liftb(b: Boolean): R[Boolean]

  def _app(fun: R[Value], args: R[Value], cont: Value): R[Value]
  def _fun(f: Fun[R]): R[Value]
  def _cont(f: FunC[R]): Value

  def _true(v: R[Value]): R[Boolean]
  def _if(c: R[Boolean], a: =>R[Value], b: =>R[Value]): R[Value]

  def _cons(car:R[Value],cdr:R[Value]): R[Value]
  def _car(p: R[Value]): R[Value]
  def _cdr(p: R[Value]): R[Value]

  def _cell_new(v: R[Value], memo: String): R[Value]
  def _cell_set(c: R[Value], v: R[Value]): R[Value]
  def _cell_read(c: R[Value]): R[Value]

  def inRep: Boolean
}
```

“Classic” vs Rigid Structures for Reflective Towers



Purple: Modified Semantics

-
  ```
  (EM (begin (define counter 0)
    (define old-eval-var eval-var)
    (set! eval-var (clambda (e r k)
      (if (eq? e 'n)
        (set! counter (+ counter 1)))
      (old-eval-var e r k)))))
  ```

-
  ```
  > (fib 7)          ;=> 13
  > (EM counter) ;=> 102
  ```

-
  ```
  (set! fib (clambda (n) …))
  (EM (set! counter 0))
  > (fib 7)          ;=> 13
  > (EM counter) ;=> 102
  ```

Purple: User-Level "Interpreter"

-
  ```
  (define matches
   (clambda (r) (clambda (s)
    (if (null? r) #t (if (null? s) #f
    (if (eq? (car r) (car s))
     ((matches (cdr r)) (cdr s)) #f))))))
  ```
- `> ((matches '(a b)) '(a c))   ;=> #f`
- `> ((matches '(a b)) '(a b))   ;=> #t`
- `> ((matches '(a b)) '(a b c)) ;=> #t`

Purple: User-Level Collapse

-
  ```
  (define start_ab
    ((clambda () (matches '(a b)))))
  ```

-
  ```
  (define start_ab (clambda (s)
    (if (null? s) #f
    (if (eq? 'a (car s)) ((clambda (s)
    (if (null? s) #f
    (if (eq? 'b (car s)) ((clambda (s)
    #t) (cdr s)) #f))) (cdr s))))))
  ```

Summary

- collapse towers of interpreters
  - using a stage-polymorphic multi-level lambda-calculus
  - using LMS and polytypic programming via type classes
- design: how to expose compilation/collapse?
  - explicit stage lifting
  - **clambda**
  - JIT? …

Potential Applications

- *Towers in the Wild*: e.g. Python on top of x86 runtime on top of JavaScript VM
- *Modified semantics*: e.g. instrumentation/tracing for debugging, sandboxing for security, virtualization, transactions
- *Non-standard interpretations*: program analysis, verification, synthesis, e.g. Racket interpreter on top of miniKanren, Abstracting abstract machines

Thank you!

Code for **Pink** & **Purple**: popl18.namin.net

Reflective towers: … **Brown**, **Blond**, **Black**, …

Staging & LMS: scala-lms.github.io

Stage polymorphism: github.com/GeorgOfenbeck/SpaceTime