Let It Be Optimized
(Functional Pearl)

Guannan Wei, Jun Tan, Dinghong Zhong

ICFP 2026

Tufts
UNIVERSITY

Multi-Stage Programming

Multi-stage programming promises that we can write high-level code that generates low-level efficient programs.

Examples:

- *Language extensions:*
  MetaML, MetaOCaml, Template Haskell, etc.

- *Libraries/frameworks:*
  Lightweight Modular Staging (Scala), Squid (Scala), BuildIt (C++), etc.

Multi-Stage Programming

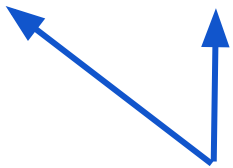
Classical example: staged “power” function to compute x^n

```
def power(n: Int, x: Code[Int]): Code[Int] =
  if n = 0 then lift(1)
  else x * power(n-1, x)
```

Multi-Stage Programming

Classical example: staged “power” function to compute x^n

```
def power(n: Int, x: Code[Int]): Code[Int] =
  if n = 0 then lift(1)
  else x * power(n-1, x)
```



next-stage code fragments

Multi-Stage Programming

Classical example: staged “power” function to compute x^n

```
def power(n: Int, x: Code[Int]): Code[Int] =
  if n = 0 then lift(1)
  else x * power(n-1, x)
```

Multi-Stage Programming

Classical example: staged “power” function to compute x^n

```
def power(n: Int, x: Code[Int]): Code[Int] =
  if n = 0 then lift(1)
  else x * power(n-1, x)
```

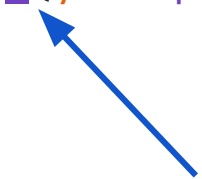
```
_ * _: Code[Int] → Code[Int] → Code[Int]
```

Multi-Stage Programming

Classical example: staged “power” function to compute x^n

```
def power(n: Int, x: Code[Int]): Code[Int] =
  if n = 0 then lift(1)
  else x * power(n-1, x)

// specialize power with n = 4 and unknown next-stage value y
λ(y ⇒ power(4, y))
```



```
λ: (Code[A] → Code[B]) → Code[A → B]
```

Multi-Stage Programming

Classical example: staged “power” function to compute x^n

```
def power(n: Int, x: Code[Int]): Code[Int] =
  if n = 0 then lift(1)
  else x * power(n-1, x)


// specialize power with n = 4 and unknown next-stage value y
λ(y ⇒ power(4, y))
```

The generated code is equivalent to `power(4, y)`:

```
y ⇒ y * y * y * y * 1
```

Multi-Stage Programming

Classical example: staged “power” function to compute x^n

```
def power(n: Int, x: Code[Int]): Code[Int] =
  if n = 0 then lift(1)
  else x * power(n-1, x)


// specialize power with n = 4 and unknown next-stage value y
λ(y ⇒ power(4, y))
```

The generated code is equivalent to `power(4, y)`:

```
y ⇒ y * y * y * y * 1
```

```
y ⇒ y * y * y * y // optimize with multiplicative identity
```

Multi-Stage Programming

Classical example: staged “power” function to compute x^n

```
def power(n: Int, x: Code[Int]): Code[Int] =
  if n = 0 then lift(1)
  else x * power(n-1, x)


// specialize power with n = 4 and unknown next-stage value y
λ(y ⇒ power(4, y))
```

The generated code is equivalent to `power(4, y)`:

```
y ⇒ y * y * y * y * 1

y ⇒ y * y * y * y // optimize with multiplicative identity

y ⇒ let x1 = y * y in let x2 = x1 * x1 in x2 // even better
```

This Functional Pearl

How to build interpreters for such multi-stage languages

This Functional Pearl

How to build interpreters for such multi-stage languages with automatic let-insertion

This Functional Pearl

How to build interpreters for such multi-stage languages
with automatic let-insertion
and semantics-preserving optimizations

This Functional Pearl

How to build interpreters for such multi-stage languages
with automatic let-insertion
and semantics-preserving optimizations
in a modular fashion.

Staging with Let-Insertion

Example:

```
(λx. x + x) lift(42)
```

What should the generated code look like?

Staging with Let-Insertion

Example:

```
(λx. x + x) lift(42)
```

What should the generated code look like?

Syntactic manipulation: splice the argument twice

```
42 + 42
```

Staging with Let-Insertion

Example:

```
(λx. x + x) f()
```

but what about an effectful next-stage term?

Staging with Let-Insertion

Example:

```
(λx. x + x) f()
```

Syntactic manipulation: splice the application twice ✗

```
f() + f()
```

duplicate effects, inconsistent with *unstaged* CBV evaluation `(λx. x + x) f()`

Staging with Let-Insertion

Example:

```
(λx. x + x) f()
```

Syntactic manipulation: splice the application twice ✗

```
f() + f()
```

duplicate effects, inconsistent with *unstaged* CBV evaluation `(λx. x + x) f()`

Let-insertion: use let-bindings for each intermediate term ✓

```
let x1 = f() in
let x2 = x1 + x1 in x2
```

preserve sharing and evaluation order

Staging with Let-Insertion

Example:

```
(λx. x + x) f()
```

Goal: implement automatic let-insertion staging as an evaluator

Syntactic manipulation: splice the application twice ✗

```
f() + f()
```

Let-insertion: use let-bindings for each intermediate term ✓

```
let x1 = f() in
let x2 = x1 + x1 in x2
```

Staging with Let-Insertion

The starting point: an environment-passing evaluator

```
def eval(e: Expr, ρ: Env): Value = e match
  case EVar(x) ⇒ ρ(x)
  case ELam(x, e) ⇒ VClo(x, e, ρ)
  case EApp1(e1, e2) ⇒
    val VClo(x, body, ρf) = eval(e1, ρ)
    eval(body, ρf + (x ↦ eval(e2, ρ)))
```

Staging with Let-Insertion

The starting point: an environment-passing evaluator

```
def eval(e: Expr, ρ: Env): Value = e match
  case EVar(x) ⇒ ρ(x)
  case ELam(x, e) ⇒ VClo(x, e, ρ)
  case EApp1(e1, e2) ⇒
    val VClo(x, body, ρf) = eval(e1, ρ)
    eval(body, ρf + (x ↦ eval(e2, ρ)))
  case EApp2(e1, e2) ⇒ ...
  case EPlus2(e1, e2) ⇒ ...
  case ELift(e) ⇒ ...
  case ERun(e) ⇒ ...
  ...
```

staging constructs

`EPlus2(e1, e2)` corresponds to `e1 + e2`, etc.

Let-Insertion with Mutable State

The starting point: an environment-passing evaluator *with a mutable list to construct let-bindings for the next stage* [Rompf '16]

```
var stBlock: List[(String, Expr)]
```

Let-Insertion with Mutable State

The starting point: an environment-passing evaluator *with a mutable list to construct let-bindings for the next stage* [Rompf '16]

```
var stBlock: List[(String, Expr)]
def reflect(e) =
  stBlock += (x, e) // record x ↦ e in stBlock for some fresh x
  VCode(x)          // return object variable x
def reify(f) = ...  // delimit a scope and assemble its bindings
```

Let-Insertion with Mutable State

The starting point: an environment-passing evaluator *with a mutable list to construct let-bindings for the next stage* [Rompf '16]

```
def eval(e: Expr, ρ: Env): Value = e match
  case EPlus2(e1, e2) ⇒
    val VCode(x1) = eval(e1, ρ)
    val VCode(x2) = eval(e2, ρ)
    reflect(EPlus1(x1, x2))
  ...
```

adds x3 ↦ EPlus1(x1, x2) to stBlock,
returns VCode(x3)

Let-Insertion with Mutable State

The starting point: an environment-passing evaluator *with a mutable list to construct let-bindings for the next stage* [Rompf '16]

```
var stBlock: List[(String, Expr)]
```

Example: `(λx. x + x) f()`

```
reflect(f())      // stBlock = [x1 ↦ f()]
```

Let-Insertion with Mutable State

The starting point: an environment-passing evaluator *with a mutable list to construct let-bindings for the next stage* [Rompf '16]

```
var stBlock: List[(String, Expr)]
```

Example: `(λx. x + x) f()`

```
reflect(f())       // stBlock = [x1 ↦ f()]
reflect(x1 + x1)   // stBlock = [x1 ↦ f(), x2 ↦ x1 + x1]
```

Let-Insertion with Mutable State

The starting point: an environment-passing evaluator *with a mutable list to construct let-bindings for the next stage* [Rompf '16]

```
var stBlock: List[(String, Expr)]
```

Example: `(λx. x + x) f()`

```
reflect(f())       // stBlock = [x1 ↦ f()]
reflect(x1 + x1)   // stBlock = [x1 ↦ f(), x2 ↦ x1 + x1]
reify(...)         // let x1 = f() in let x2 = x1 + x1 in x2
```

Let-Insertion with Mutable State

The starting point: an environment-passing evaluator *with a mutable list to construct let-bindings for the next stage* [Rompf '16]

```
var stBlock: List[(String, Expr)]
```

Example: `(λx. x + x) f()`

```
reflect(f())       // stBlock = [x1 ↦ f()]
reflect(x1 + x1)   // stBlock = [x1 ↦ f(), x2 ↦ x1 + x1]
reify(...)         // let x1 = f() in let x2 = x1 + x1 in x2
```

Can we eliminate the use of mutable state `stBlock`?

Let-Insertion with Mutable State

The starting point: an environment-passing evaluator *with a mutable list to construct let-bindings for the next stage* [Rompf '16]

```
var stBlock: List[(String, Expr)]
```

Example: `(λx. x + x) f()`

```
reflect(f())       // stBlock = [x1 ↦ f()]
reflect(x1 + x1)   // stBlock = [x1 ↦ f(), x2 ↦ x1 + x1]
reify(...)         // let x1 = f() in let x2 = x1 + x1 in x2
```

Can we eliminate the use of mutable state `stBlock`?

When `reflect` records `x ↦ e` who constructs the body of that `let`?

Let-Insertion with Continuations

Instead of returning an object variable representing that bound expression, `reflect` takes a continuation to complete the `let`-body:

```
def reflect(e, k) =
  val x = fresh()
  val body = k(x)
  VCode(ELet(x, e, body))
```

Let-Insertion with Continuations

Instead of returning an object variable representing that bound expression, `reflect` takes a continuation to complete the `let`-body:

```
def reflect(e, k) =
  val x = fresh()
  val body = k(x)
  VCode(ELet(x, e, body))
```

But wait a moment, is that “continuation” `k` truly a continuation?

Let-Insertion with Continuations

Instead of returning an object variable representing that bound expression, `reflect` takes a continuation to complete the `let`-body:

```
def reflect(e, k) =
  val x = fresh()
  val body = k(x)
  VCode(ELet(x, e, body))
```

But wait a moment, is that “continuation” `k` truly a continuation?

Hmm, assembling the whole `let`-expr still happens after that continuation returns.

Let-Insertion with Continuations

Instead of returning an object variable representing that bound expression, `reflect` takes a continuation to complete the `let`-body:

```
def reflect(e, k) =
  val x = fresh()
  val body = k(x)
  VCode(ELet(x, e, body))
```

This is the continuation of k, let's call it meta-continuation!

But wait a moment, is that "continuation" `k` truly a continuation?

Hmm, assembling the whole `let`-expr still happens after that continuation returns.

Let-Insertion with 2 Continuations

Let's make another continuation-passing style transformation:

```
def reflect(e, k, γ) =
  val x = fresh()
  k(x, body ⇒
    γ(VCode(ELet(x, e, body))
```

- cont. `k` builds the `let`-body and takes its own (meta-)continuation.
- meta-cont. `γ` receives the completed `let` and forward it outside.

Let-Insertion with 2 Continuations

Let's make another continuation-passing style transformation:

```
def reflect(e, k, γ) =
  val x = fresh()
  k(x, body ⇒
    γ(VCode(ELet(x, e, body))
```

Why this style is preferred?

- It eliminates the use of mutable states.

Let-Insertion with 2 Continuations

Let's make another continuation-passing style transformation:

```
def reflect(e, k, γ) =
  val x = fresh()
  k(x, body ⇒
    γ(VCode(ELet(x, e, body))
```

Why this style is preferred?

- It eliminates the use of mutable states.
- Stack-safe w/ TCO: all control becomes explicit and tail-positioned.

Let-Insertion with 2 Continuations

Let's make another continuation-passing style transformation:

```
def reflect(e, k, γ) =
  val x = fresh()
  k(x, body ⇒
    γ(VCode(ELet(x, e, body))
```

Why this style is preferred?

- It eliminates the use of mutable states.
- Stack-safe w/ TCO: all control becomes explicit and tail-positioned.
- Definitional: independence of the host's evaluation strategy.

Let-Insertion with 2 Continuations

Let's make another continuation-passing style transformation:

```
def reflect(e, k, γ) =
  val x = fresh()
  k(x, body ⇒
    γ(VCode(ELet(x, e, body))
```

Why this style is preferred?

- It eliminates the use of mutable states.
- Stack-safe w/ TCO: all control becomes explicit and tail-positioned.
- Definitional: independence of the host's evaluation strategy.
- It also makes the scope of optimizations explicit!

The Many Essences of Let-Inserting Staging

Let-insertion is fundamental about control. The pearl revisits and relates existing techniques applied to our small language with staged evaluation:

- mutable state [Sumii and Kobayashi, ‘01; Rompf, ‘16; Amin and Rompf, ‘18]
- 1 continuation [Flanagan, ‘93]
- 2 continuations/extended CPS [Danvy, ‘90]

The Many Essences of Let-Inserting Staging

Let-insertion is fundamental about control. The pearl revisits and relates existing techniques applied to our small language with staged evaluation:

- mutable state [Sumii and Kobayashi, ‘01; Rompf, ‘16; Amin and Rompf, ‘18]
- 1 continuation [Flanagan, ‘93]
- 2 continuations/extended CPS [Danvy, ‘90]
- delimited control operators (shift/reset)
  [Danvy ‘96; Kameyama, ‘07; `genlet` in BER MetaOCaml]

```
def reflect(e): =
  val x = fresh()
  shift: k ⇒
    for case VCode(body) ← k(x)
    yield VCode(ELet(x, e, body))
```

The Many Essences of Let-Inserting Staging

Let-insertion is fundamental about control. The pearl revisits and relates existing techniques applied to our small language with staged evaluation:

- mutable state [Sumii and Kobayashi, ‘01; Rompf, ‘16; Amin and Rompf, ‘18]
- 1 continuation [Flanagan, ‘93]
- 2 continuations/extended CPS [Danvy, ‘90]
- delimited control operators (shift/reset)
  [Danvy ‘96; Kameyama, ‘07; `genlet` in BER MetaOCaml]
- abstract machine with defunctionalized continuations

$$\langle\langle \lambda x.e,\ \rho\rangle,\ \mathbf{lift}(\kappa),\ \gamma\rangle_{cont1} \longmapsto_{\text{CEKM}} \langle e,\ \rho[x \mapsto \text{code } y],\ \square,\ \gamma \circ \boldsymbol{\lambda}\mathbf{c}(y,\kappa)\rangle_{eval} \quad \textit{where } y \textit{ is fresh}$$

$$\langle \text{code } e,\ \boldsymbol{\lambda}\mathbf{c}(y,\kappa),\ \gamma\rangle_{cont1} \longmapsto_{\text{CEKM}} \langle \text{code } x,\ \kappa,\ \gamma \circ \mathbf{letc}(x, \lambda y.e)\rangle_{cont1} \quad \textit{where } x \textit{ is fresh}$$

The Many Essences of Let-Inserting Staging

Let-insertion is fundamental about control. The pearl revisits and relates existing techniques applied to our small language with staged evaluation:

- mutable state [Sumii and Kobayashi, ‘01; Rompf, ‘16; Amin and Rompf, ‘18]
- 1 continuation [Flanagan, ‘93]
- **2 continuations/extended CPS** [Danvy, ‘90]
- delimited control operators (shift/reset)
  [Danvy ‘96; Kameyama et al., ‘09; `genlet` in BER MetaOCaml]
- abstract machine with defunctionalized continuations

Artifact provides complete code, each in about 100 LoC.

Semantics-Preserving Optimizations

We want to optimize code on-the-fly and directly generate optimized code.

- Intensional analysis with code/AST pattern matching
- Semantics-preserving optimizations intrinsic to the staged evaluation

Semantics-Preserving Optimizations

We want to optimize code on-the-fly and directly generate optimized code.

- Intensional analysis with code/AST pattern matching
- Semantics-preserving optimizations intrinsic to the staged evaluation
- *Conflict*: we cannot have both as intrinsic optimizations may change the representation inspected by intensional analysis, thus change the behavior of the program. [Taha, ‘99]

Semantics-Preserving Optimizations

We want to optimize code on-the-fly and directly generate optimized code.

- Intensional analysis with code/AST pattern matching
- Semantics-preserving optimizations intrinsic to the staged evaluation
- *Conflict*: we cannot have both as intrinsic optimizations may change the representation inspected by intensional analysis, thus change the behavior of the program. [Taha, ‘99]

Semantics-Preserving Optimizations

Example: Constant propagation

```
let x1 = 1 in
let x2 = 2 in
let x3 = x1 + x2 in x3
```

```
let x3 = 1 + 2 in x3
```

Semantics-Preserving Optimizations

Example: Constant propagation

```
let x1 = 1 in
let x2 = 2 in
let x3 = x1 + x2 in x3
```

```
let x3 = 1 + 2 in x3
```

```
def reflect(e, k, γ) = e match
  case ENum(i) ⇒ k(VCode(ENum(i)), γ)
  case _ ⇒ k(x, body ⇒ ...)
```

Inspect the expr to be reflected:

If e is a constant, directly pass it to the continuation!

Semantics-Preserving Optimizations

Example: Constant propagation + constant folding

```
let x1 = 1 in
let x2 = 2 in
let x3 = x1 + x2 in x3
```



```
let x3 = 1 + 2 in x3
```



```
3
```

```
def reflect(e, k, γ) = e match
  case EPlus(Const(x), Const(y)) ⇒
    k(VCode(ENum(x+y)), γ)
  // ...
```

Inspect the expr to be reflected:

If e has constant operands, directly fold it and pass to cont.

Semantics-Preserving Optimizations

The continuations make scope-induced optimizations straightforward.

Example: Dead-code elimination

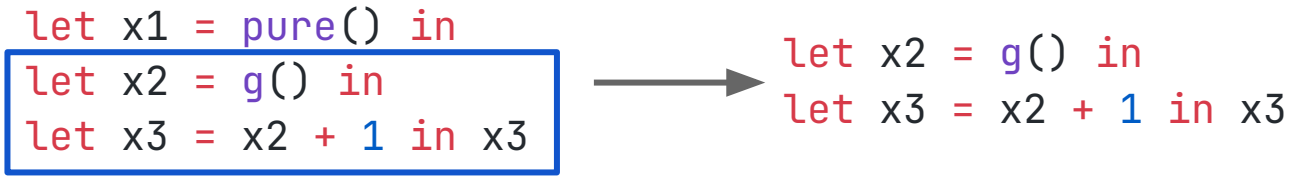
```
let x1 = pure() in
let x2 = g() in
let x3 = x2 + 1 in x3
```

```
let x2 = g() in
let x3 = x2 + 1 in x3
```

Semantics-Preserving Optimizations

The continuations make scope-induced optimizations straightforward.

Example: Dead-code elimination




```
def reflect(e, k, γ) = k(x, body ⇒ ...)
```

Idea: with continuation, we have a chance to inspect the downstream body before committing to the let-binding.

Semantics-Preserving Optimizations

The continuations make scope-induced optimizations straightforward.

Example: Dead-code elimination

```
let x1 = pure() in
let x2 = g() in
let x3 = x2 + 1 in x3
```

```
let x2 = g() in
let x3 = x2 + 1 in x3
```

```
def reflect(e, k, γ) =
  k(x, body ⇒
    if !body.dependsOn(x)
    then γ(VCode(body))
    else γ(VCode(ELet(x, e, body))))
```

No need to backtrack a mutable state or rely post-hoc whole-program dependency analysis.

More Optimizations in the Paper

- Common subexpression elimination
- Algebraic simplification/partially static data

  `1+x+2` → `3+x`
- Function inlining with ANF renormalization

  replay reflects with call-site's cont/meta-cont.
- Code motion

  moving let-bindings across boundaries captured by cont/meta-cont.

All these optimizations are modularly defined with a combinator-style architecture in about 500 LoC.

Let It Be Optimized

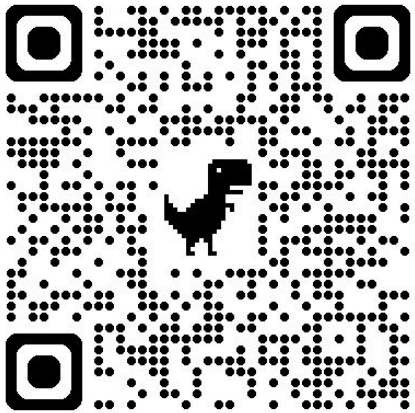
A distilled tutorial about building interpreters for multi-stage languages with automatic let-insertion and semantics-preserving optimizations in a modular fashion.

Insert `let` with

- mutable state
- 1 continuation
- 2 continuations
- shift/reset
- abstract machine

Optimize code with

- algebraic simplification
- common subexpr elim.
- dead-code elimination
- function inlining
- code motion



[QR code for artifact]