Let It Be Optimized: Building Multi-stage Evaluators with Let-Insertion and Optimizations in Small Pieces (Functional Pearl)

GUANNAN WEI, Tufts University, USA
JUN TAN, Independent, China
DINGHONG ZHONG, Tufts University, USA

Multi-stage programming lets programmers write meta-programs that generate efficient code. Staging is typically realized either as a language primitive with quotations and splices (*e.g.*, MetaML and its descendants), or as a library embedded in a host language (*e.g.*, Lightweight Modular Staging). Unlike quotation-based approaches, practical library-based systems combine staged evaluation with automatic let-insertion to preserve evaluation order, along with optimizations that improve residual code. Despite their popularity and practical importance, this combination has received little semantic treatment, making it difficult to reason about correctness or to compare systematically with other staging paradigms.

Using functional programming techniques, this pearl illuminates the operational aspects of staged evaluation with automatic let-insertion and optimizations as found in library-based staging systems. For a core two-stage language, we develop a series of definitional interpreters that concisely describe staged evaluation generating optimized, let-inserted residual programs. The interpreters are written in the extended continuation-passing style, where two continuations naturally account for let-insertion. With minor refactoring, we showcase a suite of optimizations, ranging from simple constant propagation/folding, common subexpression elimination, and dead-code elimination to more involved optimizations such as β-inlining, partially-static data, and code motion. Each optimization is presented as a small, modular extension integrated into staged evaluation, requiring neither additional effort from the meta-programmer, nor complex post-hoc compiler infrastructure.

CCS Concepts: • **Theory of computation → Program semantics**; • **Software and its engineering → Interpreters**; **Source code generation**.

Additional Key Words and Phrases: staging, let-insertion, optimizations, partial evaluation, continuation

ACM Reference Format:
Guannan Wei, Jun Tan, and Dinghong Zhong. 2026. Let It Be Optimized: Building Multi-stage Evaluators with Let-Insertion and Optimizations in Small Pieces (Functional Pearl). *Proc. ACM Program. Lang.* 10, ICFP, Article 278 (August 2026), 31 pages. https://doi.org/10.1145/3828676

1 Introduction

Multi-stage programming (MSP) [87, 88] provides a principled way to write code generators that reuse the abstraction mechanisms of a host language while enabling programmer-controlled optimizations. By controlling which parts of a program are evaluated at compile time and which are deferred to runtime, programmers can construct meta-programs that produce new programs

Authors' Contact Information: Guannan Wei, Tufts University, Medford, USA, guannan.wei@tufts.edu; Jun Tan, Independent, Shanghai, China, tjapzjj@gmail.com; Dinghong Zhong, Tufts University, Medford, USA, dinghong.zhong@tufts.edu.

ACM 2475-1421/2026/8-ART278
https://doi.org/10.1145/3828676

specialized to known inputs [26, 44]. MSP has been widely used to implement domain-specific languages (DSLs) or compilers [28, 57] with type-safe high-level interfaces and low-level performance (*e.g.*, [49, 65, 73, 89, 95, 96]).

Staging Interfaces and Implementations. At the very surface level, MSP systems differ along two closely related axes: how staging intent is expressed in programs, and whether staging is provided as a primitive language feature or as an embedded library.

In language-based systems, staging is typically built into the host language through primitive constructs such as quotation, splicing, and `run`, as exemplified by MetaML [87, 88] and its descendants, such as MetaOCaml [13, 50, 51], (typed) Template Haskell [59, 78, 100], Scala 3 [80, 81], and recent research calculi [15, 101]. This design seamlessly integrates staging directly with the language, making code values and staged computation first-class concepts. However, with quotation/splicing, staging intent is expressed syntactically and explicitly: programmers intersperse quotations and splices throughout the program, and must reason carefully about effects and evaluation order across stages. In particular, quotation/splicing may delay or duplicate effects [40, 41], and preserving the intended evaluation order often necessitates explicit let-insertion [52, 90, 102].

Library-based systems take the practical route: instead of introducing new syntax, staging is embedded in the host language by reusing its abstraction and binding mechanisms (*e.g.*, HOAS [14, 67]). Running a host-language program with the library performs the static computation and constructs the representation of the generated code. This approach is exemplified by Lightweight Modular Staging (LMS) [74, 75] and adopted in systems such as Delite [82], Spatial/Argon [53], Squid [65, 66], BuildIt [10], and various embedded DSL libraries and compilers [3, 39, 45, 60]. Library-based approaches often provide automatic let-insertion to preserve the intended order of effects [19, 33, 46, 55, 65, 70], relieving programmers from manually managing let-bindings.

A limitation of library-based approaches is that staging may not integrate seamlessly with all host language features and can introduce repetitive boilerplate in the meta-program. Moreover, libraries are often tightly tied to specific host language implementations (*e.g.*, virtualization [61, 72] or macros), which complicate maintenance and evolution as the host language itself evolves.

By reusing host language abstractions, library-based approaches can flexibly implement a variety of surface interfaces to express staging intent. In LMS, for example, staging is largely expressed through type-based annotations, often requiring only minimal changes to the original program. On the other hand, quotation and splicing are also possible via macros [65, 66]. However, different surface staging interfaces are less fundamental in the internal implementations than they appear from the surface. Both LMS-style type-based annotations and MetaML-style quotation/splicing can be elaborated into two-level terms that explicitly encode binding-time information [10, 51, 70, 74], tracing back to the foundations of two-level functional languages [63].

Optimizing Object Code. Orthogonal to the choice of staging interface, a crucial aspect in MSP systems is generating optimized object code. These optimizations can be generic or domain-specific, and are essential for achieving competitive performance. A simple example is constant folding, such as simplifying `<1 + 1>` to `<2>` (with angle brackets denoting object code), or β-inlining an object-level function application.

There are several approaches to achieving this: (1) The host language or library can apply programmer-invisible optimizations over object code, *e.g.*, by lifting existing optimizations to the generated code. We call these *intrinsic* optimizations because they are built into the staging system and need not be invoked explicitly. (2) The language/library can provide intensional analysis [15, 81, 93], allowing meta-programmers to inspect and transform object code to implement

domain-specific optimizers. (3) Orthogonally, the meta-program itself can be constructed to improve binding times [17, 24, 103]. This approach often requires careful meta-program structuring to shift more computation to earlier stages, leaving object-code generation as the final step.

In the first approach, language-based staging systems (*e.g.*, MetaOCaml) typically do not directly optimize code objects during construction. Instead, optimizations can be applied *offline*, delegated to the downstream host compiler after object code is generated. In contrast, library-based frameworks (*e.g.*, LMS) use richer internal representations of object code and perform *online* optimizations during staging, in addition to any downstream optimizations. These optimizations may be generic, such as constant folding, common-subexpression elimination, and dead-code elimination, or domain-specific, implemented by meta-programmers over the internal code representation. Several recent staging systems [15, 66, 81] favor the second approach, exposing analytical capabilities to inspect and transform object code.

However, as observed by Taha [85], the first and second approaches are in subtle tension. If intensional analysis is supported *within* the same language, combining it with intrinsic optimizations can compromise soundness: optimizations built into the system may change the code being inspected or transformed by the meta-programmer, thus producing different evaluation results and rendering the combination unsound. Although one can reasonably "associate domain-specific optimization to specific instances of the code datatype" [84, p. 152], a general combination of object-code optimizations and intensional analysis is dangerous.

A Missing Piece. Table 1 classifies several representative MSP approaches along three dimensions: (1) the programmer-facing interface for expressing staging intent; (2) whether staging is implemented as a language feature or as a library; and (3) the mechanisms used to optimize generated object code. It still seems premature to declare a clear winner: each approach strikes a different trade-off between integration, expressiveness, and control over optimization. A natural question then arises: how can these approaches be formally compared, and to what extent can they be reconciled? Answering this question first requires a precise semantic understanding of each approach.

Despite extensive work on the semantics of staged languages, precise semantic accounts of popular library-based approaches with aggressive intrinsic optimizations remain understudied (Table 1). These systems are widely used in practice [11, 89, 94, 96], yet they have received comparatively little semantic treatment. One challenge is that they rely on host-language features, such as type-based operator overloading or macros, to express and resolve staging intent, making it difficult to describe their staging semantics independently. Another is that library-based approaches adopt internal representations of object code and sophisticated staged evaluation mechanisms, such as automatic let-insertion and intrinsic optimizations over generated code. A clean semantic account must therefore disentangle the staging mechanisms from host-language features, while abstracting away from intricate implementation details often shaped by ad-hoc design choices.

This Pearl. The goal of this pearl is to reconstruct the optimizing staged evaluation found in library-based approaches as a proper language with executable semantic artifacts. We illuminate the operational aspects of staged evaluation *equipped with automatic let-insertion and object-code optimizations* using functional programming techniques, specifically by developing a series of definitional interpreters and abstract machines for a core two-stage language through the lens of continuations. Table 1 shows where this pearl fits in the landscape of different MSP approaches.

This pearl is inspired by prior work that attempted to explain LMS-style staging using stateful evaluators [4, 70] or reduction semantics [4]. However, these accounts are either imperative in nature (although somewhat close to the actual implementation) or stop short of modeling the optimizations performed on object code. As a result, it remains difficult to see their functional character and to understand how optimizations can be integrated into the staged evaluation.

Table 1. Comparing several representative approaches to multi-stage programming.

Interface: ○ - quotation + splice ● - 2-level terms ◐ - types		**Staging as primitives**	**Staging as libraries**
Intrinsic optimizations	**Offline**[1]	MetaML [87, 88] ○ MetaOCaml [13, 51] ○ MacoCaml [101] ○ (Typed) Template Haskell [59, 78, 100] ○ Mint [98] ○ $\lambda_{\uparrow\downarrow}$ [4] ●	-
	Online[2]	*This pearl* ●	LMS [74, 75] ◐ BuildIt [10] ◐
Intensional analysis		Scala 3 and $\lambda^{\blacktriangle}$ [81] ○ Mœbius [42] ○ $\lambda^{\circ\triangleright}_{\text{pat}}$ [15] ○ $\lambda^{\{\}}$ [66] ○	Squid [65, 66] ○

[1] "Offline" optimizations are applied once the object code is generated, typically by the downstream host compiler. [2] "Online" optimizations are applied during object-code construction, in addition to downstream offline optimizations.

Figure 1 shows the structure of this pearl. We begin with the prior stateful evaluator for LMS-style staging [70] (Section 2) and systematically derive a functional evaluator by applying the CPS transformation *twice* [23]. The resulting artifact (Section 3) is a definitional interpreter [69] for a core two-stage language written in 2-CPS, which expresses automatic let-insertion to preserve evaluation order (without object-code optimizations for the moment). The interpreter is *definitional* in the sense that, by virtue of CPS, it provides an executable semantics independent of the host language evaluation strategy [69].

Along the way, the development illuminates the essence of LMS-style staged evaluation from multiple perspectives and connects it to established techniques, including Flanagan et al. [33]'s original ANF transformation, the CPS hierarchy and delimited control operators [22], type-directed partial evaluation [19] (a.k.a. normalization by evaluation [6]), and monadic reflection [31]. Additionally, following defunctionalization [25, 69], we present an abstract machine for the same staged evaluation strategy (Section 4), which to our knowledge has not previously been described.

Then, on top of the 2-CPS definitional interpreter, we demonstrate various object-code optimizations *during* staged evaluation. The key insight is that with some minor bookkeeping, the two-continuation structure can be leveraged to concisely implement a wide range of optimizations, starting from simple constant propagation/folding (Sections 5 and 7), common subexpression elimination (Section 6), and dead-code elimination (Section 8), to non-trivial optimizations such as β-inlining (Section 9) with ANF-renormalization [76], partially-static data (Section 10), and code motion (Section 11). Production compilers typically apply such optimizations heuristically; here, we abstract from heuristics and focus instead on the mechanisms that realize them within staging.

Although these optimizations are generally well-known, this pearl demonstrates that they can be integrated directly into evaluation rather than applied post-hoc. As we will see later, several of them are enabled naturally by the 2-CPS structure, such as dead-code elimination and code motion. We focus on generic, domain-independent optimizations, which are transparent and free for the programmer. Nevertheless, the mechanisms demonstrated in the pearl also provide a foundation for incorporating complementary domain-specific ones.

In the context of optimization, we also explain why introducing intensional analyses to inspect or transform object code can conflict with these optimizations (Section 9), following Taha [85].

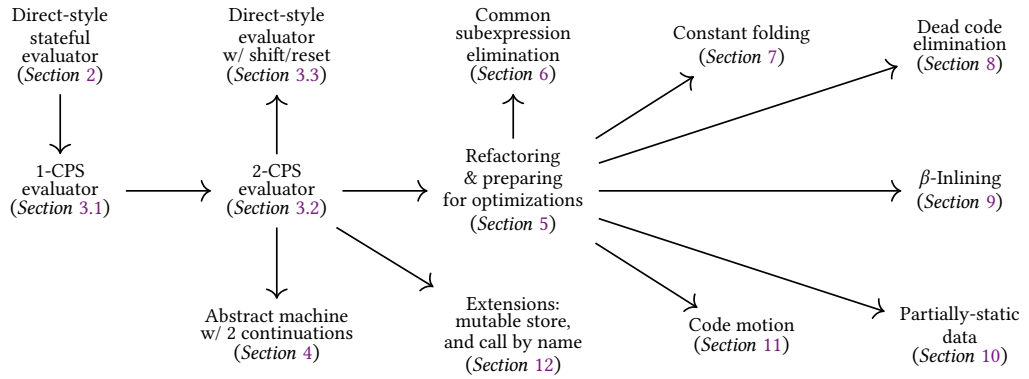

Fig. 1. Overview of this paper.

Finally, Section 12 extends the language with mutable references and recursive functions, and demonstrates how quasi-quotations can be simulated by introducing by-name application.

Contributions and Organization. This pearl explains the essence of staged evaluation featuring automatic let-insertion and various non-trivial object-code optimizations. Both automatic let-insertion and built-in generic optimizations are crucial for practical multi-stage programming: the former preserves the intended evaluation order of side effects, while the latter relieves the programmer from writing their own general-purpose optimizers.

This pearl demonstrates how these features can be elegantly integrated into the staged evaluation itself, without resorting to either meta-programmer's effort, or complex post-hoc compiler infrastructures. Our approach is pragmatic yet rooted in the semantics, and relates different semantic artifacts of the same staged evaluation through the lens of continuations. Our presentation is pedagogical and modular: both the interpreters and optimizations are broken down into small, independent pieces, making them accessible and amenable to further adoption.

Figure 1 shows an overview of the paper. We discuss more related work in Section 13 and conclude the paper in Section 14.

2 Revisiting the WadlerFest Stateful Evaluator

The starting point of our exposition is the stateful evaluator for LMS-style staging presented by Rompf [70] at WadlerFest [58]. We first introduce a core two-stage language that will be shared throughout the paper. Then, starting from a reconstruction of the WadlerFest stateful evaluator [70], we progressively transform it to a functional one with two continuations, eliminating the use of mutable state in constructing the evaluator.

2.1 A Two-Stage Core Language

Figure 2 shows the abstract syntax for a core two-stage language λ_2, defined using Scala 3 algebraic data types. Throughout the paper, we use Scala 3 as the meta-language, but all constructions can be readily translated to other functional languages.

The syntax is mostly standard, and we highlight a few key features here. λ_2 has stage-annotated constructs: applications (`EApp`), conditionals (`ECond`), and binary operations (`EBinOp`) are annotated with a stage indicator, which can be either `Sta` (static, or $\mathbb{1}$) or `Dyn` (dynamic, or $\mathbb{2}$). We do not have explicit second-stage let-bindings, as they will be automatically inserted during the evaluation. To communicate between stages, programmers can explicitly lift the value of an expression from the

```
enum Stage:
  case Sta, Dyn

enum Expr:
  case ENum(i: Int)
  case EBool(b: Boolean)
  case EVar(name: String)
  case ELam(x: String, body: Expr)
  case EApp(e1: Expr, e2: Expr, st: Stage)
  case ELift(e: Expr)
  case ERun(e: Expr)
  case ELet(x: String, rhs: Expr, body: Expr)
  case ECond(e1: Expr, e2: Expr, e3: Expr, st: Stage)
  case EBinOp(op: String, e1: Expr, e2: Expr, st: Stage)
```

Fig. 2. The abstract syntax for the two-stage language λ_2.

current stage to the next stage (`ELift`), and evaluate code generated at runtime (`ERun`). Values such as numbers, booleans, and λ-terms are stage-agnostic, and can be lifted to the next stage.

For example, to generate an addition function specialized to a statically known argument `1`, we can write the source term in the surface syntax or as an AST:

```
// pseudo surface syntax: (λx. lift(λy. lift(x) +² y)) 1
EApp(ELam("x", ELift(ELam("y", EBinOp("+", ELift(EVar("x")), EVar("y"), Dyn)))),
     ENum(1),
     Sta)
```

The staged evaluation generates code in A-normal form (ANF) [8, 33] with explicit let-bindings for intermediate results:

```
let x0 = (λy. let x1 = 1 + y in x1) in x0
```

At this point, we keep the language minimal to explain the core evaluation mechanism. Later, Section 12 extends the language with mutable references and recursive functions, and shows how to simulate quasi-quotations by adding by-name applications.

2.2 The WadlerFest Stateful Evaluator

The WadlerFest evaluator produces a value (defined in Figure 3) given an expression and an environment. Other than standard values, we have `VCode(e)`, which represents a piece of generated code `e` as an `Expr`. The generated code is represented in ANF [33], where intermediate results are named by let-bindings. To generate well-scoped code, the WadlerFest evaluator uses an additional mutable state to accumulate let-bindings that need to be inserted into certain scopes. Figure 3 also shows the auxiliary mutable state `stBlock`, and key auxiliary functions used by the stateful evaluator (Figure 4).

The first-stage evaluation (*e.g.*, expressions with `Sta` binding-time) is entirely standard. The second-stage evaluation (*e.g.*, expressions with `Dyn` binding-time) deserves more attention, as it is the core of LMS-style staging. The WadlerFest stateful evaluator [70] relies on this critical mutable state `stBlock`, and three key functions (Figure 3): function `reify` takes a computation producing a `Value` and generates its representation `Expr`; `reflect` takes an `Expr` and returns a `Value` that represents the generated code; and `lift` takes a current-stage value and lifts it to the next stage, which may involve reification and reflection.

For example, to evaluate a second-stage application `EApp(e1,e2,Dyn)` (Figure 4) where e_1 and e_2 are both dynamic expressions, we first evaluate e_1 and e_2 to get two pieces of generated code x_1 and

```
type Env = Map[String, Value]
var stBlock: List[(String, Expr)] = List()

enum Value:
  case VNum(i: Int)
  case VBool(b: Boolean)
  case VClo(x: String, body: Expr, ρ: Env)
  case VCode(e: Expr)

def lift(v: Value): Value = v match
  case VNum(i) ⇒ reflect(ENum(i))
  case VBool(b) ⇒ reflect(EBool(b))
  case VClo(x, body, ρ) ⇒
    val y = fresh()
    val t = reify:
      eval(body, ρ + (x ↦ VCode(EVar(y))))
    reflect(ELam(y, t))

def reflect(e: Expr): Value =
  val z = fresh()
  stBlock = stBlock :+ (z, e)
  VCode(EVar(z))

def reify(f: ⇒ Value): Expr =
  val prev = stBlock
  stBlock = List()
  val VCode(last) = f
  val e = stBlock.foldRight(last):
    case ((x, s), acc) ⇒ ELet(x, s, acc)
  stBlock = prev
  e
```

Fig. 3. Value domains, auxiliary mutable state, and key functions used by the stateful evaluator.

x_2. The key invariant here is that x_1/x_2 are *object code variables* referring to let-bound expressions in `stBlock`, accumulated in the order of left-to-right call-by-value. Then we use `reflect` to construct the residual expression `EApp1(`x_1`,`x_2`)`,[3] and `reflect` will insert a new let-binding for `EApp1(`x_1`,`x_2`)` into `stBlock`, and return its object-code variable. In the following, we explain how this is achieved.

Reflection, Imperatively. The mutable state `stBlock` has type `List[(String, Expr)]`, which represents the accumulated let-bindings to be inserted in a certain scope. Given an expression `e`, function `reflect` directly appends a new element `(z, e)` to the end of `stBlock`, where `z` is a fresh variable name, and then returns value `VCode(EVar(z))`, an object-code variable referring to the newly inserted let-binding. In the evaluator (Figure 4), whenever we need to generate code, we always call `reflect` to ensure that the generated code is bound to an object-code variable, which can be used to construct larger expressions later.

Reification, Imperatively. Function `reify` takes a computation producing `Value` and converts it to an `Expr` representing the generated code. The way `reify` uses `stBlock` corresponds to the scope of reification. This is more apparent when lifting a λ-term or evaluating 2nd-stage branches, where we generally cannot move let-bindings out of their scopes. Therefore, in function `reify`, we first save the current `stBlock` to `prev`, and reset `stBlock` to empty, so that any let-bindings accumulated during the evaluation of `f` are only local to the scope of `f`. Then we force `f`, expecting `VCode(last)` which represents the body expression (*i.e.*, an object-code variable) of the inserted let-bindings. Now we have all the raw materials to reconstruct the object code by folding over the accumulated `stBlock`. Finally, we restore `stBlock` to `prev`, leaving the outer scope unchanged, and we return the reconstructed expression `e`.

Lifting, Imperatively. Given an expression `e`, the semantics of `ELift(e)` is to evaluate `e` to get a value `v`, and then convert the value to the next stage. If the value is atomic, then we can directly reflect its expression correspondence. However, if the value is a closure `VClo(x, body, ρ)`, we need to `reify` the closure body first and reconstruct a λ-term with the reified body to be `reflected`. The

[3] `EApp1(x1,x2)` is the shorthand for `EApp(x1,x2,Sta)`; similarly `ECond1(...)` for `ECond(...,Sta)`.

```
def eval(e: Expr, ρ: Env): Value = e match
  case ENum(i) ⇒  VNum(i)
  case EBool(b) ⇒ VBool(b)
  case EVar(x) ⇒ ρ(x)
  case ELam(x, e) ⇒ VClo(x, e, ρ)
  case EApp(e₁, e₂, Sta) ⇒
    val VClo(x, body, ρf) = eval(e₁, ρ)
    eval(body, ρf + (x ↦ eval(e₂, ρ)))
  case EApp(e₁, e₂, Dyn) ⇒
    val VCode(x₁) = eval(e₁, ρ)
    val VCode(x₂) = eval(e₂, ρ)
    reflect(EApp1(x₁, x₂))
  case ELift(e) ⇒ lift(eval(e, ρ))
  case ERun(e) ⇒ eval(reify { eval(e, ρ) }, Map())
  case ELet(x, rhs, e) ⇒ eval(e, ρ + (x ↦ eval(rhs, ρ)))
  case ECond(e₁, e₂, e₃, Sta) ⇒
    val VBool(b) = eval(e₁, ρ)
    if b then eval(e₂, ρ) else eval(e₃, ρ)
  case ECond(e₁, e₂, e₃, Dyn) ⇒
    val VCode(x₁) = eval(e₁, ρ)
    val x₂ = reify { eval(e₂, ρ) }
    val x₃ = reify { eval(e₃, ρ) }
    reflect(ECond1(x₁, x₂, x₃))

def evalTop(e: Expr): Value =
  stBlock = List(); val v = eval(e, Map())
  if stBlock.isEmpty then v
  else
    val VCode(r) = v
    VCode(stBlock.foldRight(r) { case ((x, s), acc) ⇒ ELet(x, s, acc) })
```

Fig. 4. The Stateful Evaluator for λ_2. Cases for `EBinOp` are similar to applications, thus omitted.

reification of the body proceeds under the environment ρ with a mapping from `x` to a fresh variable `VCode(EVar(y))` (in other words, normalizing the function body with a neutral argument term as in normalization by evaluation [6]). We return by reflecting a new λ-term with the reified body.

Top-Level Evaluation. Because `eval` is stateful, the staged evaluation that produces the final code value returns only an object-code variable (*i.e.*, by `reflect`); the actual generated code itself remains stored in the mutable state `stBlock`.

The top-level evaluation function (Figure 4) reconstructs the code value if necessary. If `stBlock` is still empty after the evaluation, no code has been generated and the result can be returned directly. Otherwise, the result is expected to be a `VCode` value that may reference object-code variables recorded in `stBlock`. A sequence of let-bindings is then reconstructed from `stBlock`, with the expression contained in the `VCode` serving as the innermost body.

Remarks. There are several ways to view the stateful evaluator. It can be understood as an online partial evaluator [43] for an input language that is already binding-time annotated. The staged evaluation closely mimics the process of normalization by evaluation [6] or type-directed partial evaluation (TDPE) [19]. For example, when lifting a closure value to a next-stage λ-term, we "normalize" its body with a neutral argument term. The auxiliary functions `reflect` and `reify` perform an on-the-fly ANF transformation, and are an instance of monadic reflection [31].

The mutable state `stBlock` serves as an internal representation of the residual program, analogous to the graph-based IR used in LMS [9], while the folding step in `reify` reconstructs residual code from this internal structure. The state-based technique for let-insertion was also described by Danvy [20, Section 2.7], who attributes it to Sumii via personal communication; Sumii and Kobayashi [83] later published a formal account.

3 Deriving Definitional Interpreters with Continuations for Staged Evaluation

While the WadlerFest evaluator is a concise direct-style implementation for LMS-style staging, its construction is not entirely satisfactory and leaves some questions unanswered: First, the use of mutable state to accumulate inserted let-bindings makes it hard to see its functional alternatives or implement in other purely functional host languages. Second, it is not clear whether any optimizations to object code can be straightforwardly integrated into the stateful evaluator, which is one of the crucial aspects of LMS-style staging.

Of course, one could always use a state monad (or explicit state passing) to thread the inserted let-binding store, but this introduces substantial plumbing and obscures the structure of the evaluator. We instead pursue a different route that exploits the hidden control structure of the stateful evaluator, gradually making it explicit by transforming the evaluator into continuation-passing style.

In this section, we first transform the stateful WadlerFest evaluator into one with a single delimited continuation, and then one with two continuations (*i.e.*, a 2-CPS evaluator), eventually eliminating the use of mutable state in constructing the evaluator. The 2-CPS formulation connects naturally to delimited control and the CPS hierarchy [22]. We also present its direct-style counterpart expressed using delimited control operators [22], which have been used to implement type-directed partial evaluators with let-insertion [21, 32].

3.1 Transforming to 1-CPS

Our derivation starts from a key observation: let-insertion in `reflect` is deliberately incomplete – the function only records the binding and returns the fresh variable, yet does not construct a full let-expression on its own; instead, the caller of `reflect` finishes constructing the body of the let-binding, which corresponds to the continuation of the current object-code generation scope. The actual let-expression is assembled only after reifying an entire scope, such as the body of a lifted λ-term or a branch of a second-stage conditional.

This observation naturally suggests transforming the evaluator into CPS by making the continuation explicit in `reflect`. The continuation receives the freshly generated code variable and produces the residual code for the enclosing scope.

Reflection with 1 Continuation. Concretely, instead of returning a fresh variable `VCode(EVar(...))` directly, we pass it to the continuation `k` of type `Value ⇒ Value`, which completes the generation of the let body:

```
def reflect(e: Expr, k: Value ⇒ Value): Value =
  val z = fresh()
  val VCode(t) = k(VCode(EVar(z)))
  VCode(ELet(z, e, t))
```

Introducing continuations shifts the job for constructing complete let-expressions from `reify` to `reflect`. Because `reflect` now expects an explicit continuation to finish evaluation, the evaluator itself must also be transformed into CPS. The following snippet illustrates the dynamic application case in `eval`: we first evaluate the function and the argument, and then pass the generated code to `reflect`, which uses the current continuation to receive the object-code variable for the application `EApp1(`x_1`, `x_2`)` in the residual program.

```
def eval(e: Expr, ρ: Env, k: Value ⇒ Value): Value = e match
  case EApp(e1, e2, Dyn) ⇒
    eval(e1, ρ, { case VCode(x1) ⇒
      eval(e2, ρ, { case VCode(x2) ⇒
        reflect(EApp1(x1, x2), k) }) })
  // other cases omitted ...
```

Reification without Continuation. Then, what about reification? In the stateful version, `reify` takes a thunk, evaluates it, and manages the surrounding mutable state that delimits the scope of let-insertion. With a single continuation, this auxiliary state is no longer needed. If we look at how `reify` is used in the stateful evaluator, we can see that it is always called to evaluate a term. With continuations introduced, we now specialize `reify` to this behavior under the identity continuation, which itself delimits the evaluation scope, *i.e.*, the extent of code generation when evaluating `e`:

```
def reify(e: Expr, ρ: Env): Expr =
  val VCode(t) = eval(e, ρ, v ⇒ v)
  t
```

Now, `reify` becomes much simpler and no longer relies on mutable state to control the scope of let-insertion. In the following, we show the case for 2nd-stage conditionals, which involves reifying branches to code values:

```
def eval(e: Expr, ρ: Env, k: Value ⇒ Value): Value = e match
  case ECond(e1, e2, e3, Dyn) ⇒
    eval(e1, ρ, { case VCode(x1) ⇒
      val x2 = reify(e2, ρ)
      val x3 = reify(e3, ρ)
      reflect(ECond1(x1, x2, x3), k)
    })
  // other cases omitted ...
```

Lifting with 1 Continuation. The lifting function also translates naturally to CPS: when reflecting newly generated code, we simply forward the continuation to `reflect`. Because the current scope is not yet complete, reflecting new object code reuses the evaluation continuation rather than closing the scope. When lifting a λ-term, we use `reify` to evaluate the function body under an extended environment. In the main evaluator, we call `lift` with the current continuation.

```
def lift(v: Value, k: Value ⇒ Value): Value = v match
  case VNum(i) ⇒ reflect(ENum(i), k)
  case VBool(b) ⇒ reflect(EBool(b), k)
  case VClo(x, body, ρ) ⇒
    val y = fresh()
    val e = reify(body, ρ + (x ↦ VCode(EVar(y))))
    reflect(ELam(y, e), k)
def eval(e: Expr, ρ: Env, k: Value ⇒ Value): Value = e match
  case ELift(e) ⇒ eval(e, ρ, v ⇒ lift(v, k))
  // other cases omitted ...
```

Remarks. This transformation eliminates the mutable state from the evaluator, making let-insertion functional via continuations. Using a single continuation effectively reconstructs the classical ANF transformation of Flanagan et al. [33]. In particular, our `reflect` function corresponds closely to their `normalize-name` function [33, Appendix A and Figure 9], which introduces a fresh name and passes it to the continuation that builds the body of the let-binding.

3.2 Further Transforming to 2-CPS

The evaluator derived in the previous section implements the same staged semantics as the stateful version, but it is not yet fully in CPS. In particular, the continuation call inside `reflect` is not in tail position, since the complete let-expression must still be assembled afterward. Likewise, the calls to `reify` in `eval` and `lift` are not tail calls.

This observation leads to a further CPS transformation, introducing an additional meta-continuation. After this transformation, all control flow is explicit and every call occurs in tail position, yielding *a definitional interpreter in the extended continuation-passing style* (*i.e.*, 2-CPS). A key benefit of this fully CPS-transformed interpreter is that it faithfully enforces our call-by-value staged evaluation regardless of the evaluation strategy of the host language [69]. This property is particularly important when the host language is non-strict, such as Haskell.

Reflection with 2 Continuations. We start from the `reflect` function, which after the transformation takes a continuation of type `Cont` and an additional meta-continuation of type `MCont`. We also define the answer type `Ans` to be `Value`.

```
type Ans   = Value
type MCont = Value ⇒ Ans
type Cont  = (Value, MCont) ⇒ Ans
```

The continuation `k` still receives the freshly generated code variable, and produces the body of the let-binding to the meta-continuation. The meta-continuation completes the let-expression and forwards it to the outer continuation γ, making all control flow explicit and tail-positioned.

```
def reflect(e: Expr, k: Cont, γ: MCont): Ans =
  val x = fresh()
  k(VCode(EVar(x)), { case VCode(t) ⇒
    γ(VCode(ELet(x, e, t)))
  })
```

Reification with Only the Meta-Continuation. The job of reification is merely evaluating an expression, whose result is expected to be an object-code value, and it does not need to manage the scope of let-insertion anymore. As in the 1-CPS version, we use the identity continuation to delimit the scope, but here it forwards the resulting value to the meta-continuation.

```
def reify(e: Expr, ρ: Env, γ: MCont): Ans = eval(e, ρ, (x, γ₁) ⇒ γ₁(x), γ)
```

However, we still need to provide a meta-continuation to `reify`, which determines how to use the code result of reification to construct larger expressions, as we will see below.

Lifting with 2 Continuations. For atomic cases, we simply call `reflect` with the current continuation and meta-continuation. For a closure value, we first reify the function body under an extended environment, and provide the meta-continuation receiving the reified function body `VCode(e)`. The meta-continuation then constructs the residual λ-term to be `reflected` with the current continuation and meta-continuation in the evaluation.

```
def lift(v: Value, k: Cont, γ: MCont): Ans = v match
  case VNum(i) ⇒ reflect(ENum(i), k, γ)
  case VBool(b) ⇒ reflect(EBool(b), k, γ)
  case VClo(x, body, ρ) ⇒
    val y = fresh()
    reify(body, ρ + (x ↦ VCode(EVar(y))), { case VCode(e) ⇒
      reflect(ELam(y, e), k, γ)
    })
```

We can further refactor `lift` to curry the continuation, so that `lift(k)` of type `Cont` can be conveniently used in the main evaluator as a continuation.

```
def lift(k: Cont)(v: Value, γ: MCont): Ans = ... // same body as above
```

In the following, we demonstrate the main evaluator in 2-CPS with a few representative cases, such as lifting, dynamic application, and dynamic conditionals:

```
def eval(e: Expr, ρ: Env, k: Cont, γ: MCont): Ans = e match
  case ELift(e) ⇒ eval(e, ρ, lift(k), γ)
  case EApp(e1, e2, Dyn) ⇒
    eval(e1, ρ, { case (VCode(x1), γ1) ⇒
      eval(e2, ρ, { case (VCode(x2), γ2) ⇒
        reflect(EApp1(x1, x2), k, γ2)
      }, γ1) }, γ)
  case ECond(e1, e2, e3, Dyn) ⇒
    eval(e1, ρ, { case (VCode(x1), γ1) ⇒
      reify(e2, ρ, { case VCode(x2) ⇒
        reify(e3, ρ, { case VCode(x3) ⇒
          reflect(ECond1(x1, x2, x3), k, γ1)
        }) }) }, γ)
  // other cases omitted ...
```

This interpreter can also serve as a denotational semantics for staged programs, mapping a multi-stage expression to a function with continuations: ⟦e⟧`: Env × Cont × MCont ⇒ Value`.

3.3 Back to Direct-Style with Delimited Control Operators

We have now fully transformed the stateful evaluator into a definitional interpreter in 2-CPS. In the 1-CPS version, the continuation is already delimited, *i.e.*, it is not a full continuation but a function that eventually returns; while the 2-CPS evaluator highlights its connection to extended CPS style. We could further obtain a direct-style staged evaluator with let-insertion using `shift`/`reset`, whose underlying semantics exhibits the same 2-CPS structure [22].

Unfortunately, our host language Scala 3 does not have native support for delimited control operators. However, we can simulate `shift`/`reset` in monadic style [31] and use them to implement `reify` and `reflect` as shown below. Given a return type `R`, `Cont[R, _]` is an instance of monad, and our implementation specializes both `A` and `R` to `Value`. The full implementation is omitted for brevity and is available in our artifact [97].

```
type Cont[R, A] = (A ⇒ R) ⇒ R
def reify(e: Expr, ρ: Env): Cont[Value, Value] = reset(eval(e, ρ))
def reflect(e: Expr): Cont[Value, Value] =
  val x = fresh()
  shift: k ⇒
    for case VCode(t) ← k(VCode(EVar(x)))
    yield VCode(ELet(x, e, t))
```

In fact, earlier type-directed partial evaluators with let-insertion are implemented using delimited control operators [21, Section 5.2.7] [18–20]. More recent BER MetaOCaml also uses delimited control operators [48] to support manual let-insertion [51]. Additionally, one could implement the let-inserting staged evaluation using effect handlers [34], which are now supported in mainstream languages such as OCaml 5 [79].

4 Intermezzo: An Abstract Machine for Let-Inserting Staged Evaluation

All interpreters presented so far are "big-step", and some are higher-order: they are recursively defined functions that evaluate a source term directly to a value. Readers may wonder whether

$$
\begin{array}{rrcl}
 & x, y & \in & \mathsf{Var} \\
\text{Stage} & s & \in & \mathbb{1} \mid \mathbb{2} \\
\text{Expr} & e & ::= & x \mid \lambda x.e \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{lift}\ e \mid \mathsf{run}\ e \mid (e_1\ e_2)^s \mid \cdots \\
\text{Value} & v & ::= & \mathsf{code}\ e \mid \langle \lambda x.e,\ \rho \rangle \mid \cdots \\
\text{Environment} & \rho & = & \mathsf{Var} \to_{\mathrm{fin}} \mathsf{Value} \\
\text{Continuation} & \kappa & ::= & \square \mid \mathbf{arg}^s(e, \rho, \kappa) \mid \mathbf{fun}^s(v, \kappa) \mid \mathbf{lift}(\kappa) \mid \mathbf{run}(\kappa) \mid \mathbf{\lambda c}(x, \kappa) \mid \mathbf{letc}(x, e) \\
\text{Meta Cont.} & \gamma & ::= & \bullet \mid \gamma \circ \kappa \\
\text{State} & \varsigma & ::= & \langle e,\ \rho,\ \kappa,\ \gamma \rangle_{eval} \mid \langle v,\ \kappa,\ \gamma \rangle_{cont1} \mid \langle v,\ \gamma \rangle_{cont2}
\end{array}
$$

Fig. 5. Syntax of the CEKM machine.

the same staged evaluation can be given a more explicit, mechanical semantics, which is easier to implement in a low-level language even without higher-order functions. This is indeed possible, and in this section, we briefly detour to present an abstract machine semantics for the same staged evaluation, which we call the CEKM machine.

It is well-known that abstract machines can be systematically derived from big-step evaluators via the process of CPS transformation and defunctionalization [1, 2, 25]. The CEKM machine (Figure 5) is one such defunctionalized abstract machine for the 2-CPS evaluator presented in Section 3.2: instead of representing continuations and meta-continuations as higher-order functions, the abstract machine represents them as first-order data structures (*i.e.*, stacks), and the transition rules explicitly inspect these data structures to determine the next step of evaluation.

Syntax and State Space. We use a subset of the syntax defined in Section 3 for the abstract machine. The base language is an untyped two-stage λ-calculus extended with let-bindings, lift, and run. Applications $(e_1\ e_2)^s$ are annotated with a stage indicator s, which can be either static $\mathbb{1}$ or dynamic $\mathbb{2}$. We may abbreviate $(e_1\ e_2)^{\mathbb{1}}$ as $e_1\ e_2$. Values include, but are not limited to, code values $\mathsf{code}\ e$ or closures $\langle \lambda x.e,\ \rho \rangle$. The environment ρ is a mapping from variables to values during evaluation.

Evaluation contexts are represented inside-out as continuations κ. They include the empty context (the hole $\square$), application frames $\mathbf{arg}^s(e, \rho, \kappa)$ and $\mathbf{fun}^s(v, \kappa)$ indexed by stage indicators, the staging operators $\mathbf{lift}(\kappa)$ and $\mathbf{run}(\kappa)$ for communication between stages, $\mathbf{\lambda c}(x, \kappa)$ for generating second-stage functions, and $\mathbf{letc}(x, e)$, which marks a pending let-insertion.

Meta-continuations γ are stacks of continuations, which are either empty ($\bullet$) or formed by pushing a continuation onto an existing stack, written $\gamma \circ \kappa$. Notably, $\mathbf{letc}(x, e)$ does not itself capture a continuation; instead, it is pushed onto the meta-continuation immediately when introduced.

The abstract machine semantics is defined as a single-step transition relation $\longmapsto_{\mathrm{CEKM}}$ on machine states ς, which can be one of three forms: $\langle e,\ \rho,\ \kappa,\ \gamma \rangle_{eval}$ represents evaluating a redex, $\langle v,\ \kappa,\ \gamma \rangle_{cont1}$ represents applying a continuation to a value, and $\langle v,\ \gamma \rangle_{cont2}$ represents applying a meta-continuation. The multi-step evaluation relation $\longmapsto^*_{\mathrm{CEKM}}$ is the reflexive-transitive closure of $\longmapsto_{\mathrm{CEKM}}$.

Transition Rules. To form an initial state, we inject an expression e into state $\langle e,\ \varnothing,\ \square,\ \bullet \rangle_{eval}$ with the empty environment, halt continuation, and empty meta-continuation. By repeatedly applying the transition rules, we would reach an answer state $\langle v,\ \bullet \rangle_{cont2}$.

The first-stage evaluation rules are mostly standard, similar to the CEK machine for the call-by-value λ-calculus. In the following, we discuss several interesting transition rules and explain the correspondence with the 2-CPS evaluator. Rules (APP) and (ARG-K) handle left-to-right evaluation of stage-indexed applications, pushing the continuations with their appropriate stage indicators. For first-stage application, they will eventually be handled by (FUN1-K) when the function value

(λ-VAL)	$\langle \lambda x.e,\ \rho,\ \kappa,\ \gamma \rangle_{eval}$	$\longmapsto_{\text{CEKM}}$	$\langle \langle \lambda x.e,\ \rho \rangle,\ \kappa,\ \gamma \rangle_{cont1}$
(VAR)	$\langle x,\ \rho,\ \kappa,\ \gamma \rangle_{eval}$	$\longmapsto_{\text{CEKM}}$	$\langle \rho(x),\ \kappa,\ \gamma \rangle_{cont1}$
(APP)	$\langle (e_1\ e_2)^s,\ \rho,\ \kappa,\ \gamma \rangle_{eval}$	$\longmapsto_{\text{CEKM}}$	$\langle e_1,\ \rho,\ \mathbf{arg}^s(e_2, \rho, \kappa),\ \gamma \rangle_{eval}$
(LIFT)	$\langle \mathsf{lift}\ e,\ \rho,\ \kappa,\ \gamma \rangle_{eval}$	$\longmapsto_{\text{CEKM}}$	$\langle e,\ \rho,\ \mathbf{lift}(\kappa),\ \gamma \rangle_{eval}$
(RUN)	$\langle \mathsf{run}\ e,\ \rho,\ \kappa,\ \gamma \rangle_{eval}$	$\longmapsto_{\text{CEKM}}$	$\langle e,\ \rho,\ \square,\ \gamma \circ \mathbf{run}(\kappa) \rangle_{eval}$
(LET)	$\langle \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2,\ \rho,\ \kappa,\ \gamma \rangle_{eval}$	$\longmapsto_{\text{CEKM}}$	$\langle (\lambda x.e_2)\ e_1,\ \rho,\ \kappa,\ \gamma \rangle_{eval}$
(ARG-K)	$\langle v,\ \mathbf{arg}^s(e, \rho, \kappa),\ \gamma \rangle_{cont1}$	$\longmapsto_{\text{CEKM}}$	$\langle e,\ \rho,\ \mathbf{fun}^s(v, \kappa),\ \gamma \rangle_{eval}$
(FUN1-K)	$\langle v,\ \mathbf{fun}^{\mathbb{1}}(\langle \lambda x.e,\ \rho \rangle, \kappa),\ \gamma \rangle_{cont1}$	$\longmapsto_{\text{CEKM}}$	$\langle e,\ \rho[x \mapsto v],\ \kappa,\ \gamma \rangle_{eval}$
(FUN2-K)	$\langle \mathsf{code}\ e_2,\ \mathbf{fun}^{\mathbb{2}}(\mathsf{code}\ e_1, \kappa),\ \gamma \rangle_{cont1}$	$\longmapsto_{\text{CEKM}}$	$\langle \mathsf{code}\ x,\ \kappa,\ \gamma \circ \mathbf{letc}(x, e_1\ e_2) \rangle_{cont1}$ *where x is fresh*
(LIFT-K)	$\langle \langle \lambda x.e,\ \rho \rangle,\ \mathbf{lift}(\kappa),\ \gamma \rangle_{cont1}$	$\longmapsto_{\text{CEKM}}$	$\langle e,\ \rho[x \mapsto \mathsf{code}\ y],\ \square,\ \gamma \circ \mathbf{\lambda c}(y, \kappa) \rangle_{eval}$ *where y is fresh*
(λC-K)	$\langle \mathsf{code}\ e,\ \mathbf{\lambda c}(y, \kappa),\ \gamma \rangle_{cont1}$	$\longmapsto_{\text{CEKM}}$	$\langle \mathsf{code}\ x,\ \kappa,\ \gamma \circ \mathbf{letc}(x, \lambda y.e) \rangle_{cont1}$ *where x is fresh*
(RUN-K)	$\langle \mathsf{code}\ e,\ \mathbf{run}(\kappa),\ \gamma \rangle_{cont1}$	$\longmapsto_{\text{CEKM}}$	$\langle e,\ \emptyset,\ \kappa,\ \gamma \rangle_{eval}$
(LETC-K)	$\langle \mathsf{code}\ e_2,\ \mathbf{letc}(x, e_1),\ \gamma \rangle_{cont1}$	$\longmapsto_{\text{CEKM}}$	$\langle \mathsf{code}\ (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2),\ \gamma \rangle_{cont2}$
(APPLY-MK)	$\langle v,\ \square,\ \gamma \rangle_{cont1}$	$\longmapsto_{\text{CEKM}}$	$\langle v,\ \gamma \rangle_{cont2}$
(POP-MK)	$\langle v,\ \gamma \circ \kappa \rangle_{cont2}$	$\longmapsto_{\text{CEKM}}$	$\langle v,\ \kappa,\ \gamma \rangle_{cont1}$

Fig. 6. Selected transition rules of the CEKM machine.

is a closure, which transitions to evaluating the function body in the extended environment. For second-stage application, eventually both the function and argument values are code fragments (FUN2-K), which are subject to let-inserting a second-stage application term (explained later).

Rules (RUN) and (LIFT-K) reify a term by evaluation, while resetting the current continuation as they delimit the scope of let-insertion. In the case of (LIFT-K), we evaluate under the λ-binder. These two rules step to a state with the empty continuation $\square$ (recall that the `reify` in the 2-CPS evaluator does morally the same) and push the current continuation onto the meta-continuation stack. The elimination rule (RUN-K) for run applies when a value code e is provided to the continuation $\mathbf{run}(\kappa)$. We step to evaluating e in the empty environment assuming it is a closed program, with the continuation κ and the same meta-continuation.

Rules (FUN2-K) and (λC-K) handle the cases where a code value is subject to let-insertion. They pop the current continuation frame and push a pending let-insertion marker $\mathbf{letc}(x, e)$ onto the meta-continuation stack where x is fresh. The remaining continuation κ is unchanged, since it forms the body of the eventual let-binding, which is not yet generated until the $\mathbf{letc}(x, e)$ is popped from the meta-continuation stack via (POP-MK). The whole let-expression is formed in (LETC-K) when a code value for the let-body is provided to continuation $\mathbf{letc}(x, e)$.

Remarks. The CEKM machine resembles many aspects of the environment-based abstract machine for the first level of the CPS hierarchy [7, Figure 7], which characterizes delimited control operators such as `shift` and `reset`. Both machines represent meta-continuations as stacks of continuations and iteratively pop them in the same way (APPLY-MK, POP-MK). In our machine, rules (RUN) and (LIFT-K) effectively bake the semantics of reset $\langle e \rangle$ into the transition system. Although we do not have an explicit `shift` operator, rules (FUN2-K) and (λC-K) correspond to applying the captured delimited continuation to a fresh variable.

For brevity, we omit the mechanical steps of the defunctionalization derivation that transforms the 2-CPS evaluator into the CEKM machine; interested readers can refer to [1, 7, 25] for the details.

5 Preparing for Optimizations: Refactoring and Bookkeeping

We have presented several semantic artifacts of the same let-inserting TDPE-style staged evaluation, none of which yet performs object-code optimizations. In this section, we present two minor refactorings to the 2-CPS definitional interpreter in preparation for the optimizations.

- First, we restructure `reflect` as a composition of small combinators, each implementing a specific optimization. This yields a modular and extensible architecture: new optimizations can be added by defining additional combinators and composing them with the existing ones.
- Second, we introduce a `Cache` data structure that tracks code generated in the current scope. This enables optimizations to inspect and reuse residual code, which is essential for transformations such as common subexpression elimination (CSE).

Although these refactorings can be implemented on top of either the 2-CPS evaluator or the abstract machine, we present them in the 2-CPS evaluator for clarity and modularity.

5.1 Refactoring to Reflect Combinators

Appetizer: Constant Propagation. We begin by motivating the refactoring with a simple optimization: constant propagation, a common optimization that is found in almost all compilers. In the current design (Section 3.2), `reflect` inserts a let-binding for *every* expression, including trivial and atomic ones such as numeric literals. For example, evaluating the source term

```
EBinOp("+", ELift(ENum(1)), ELift(ENum(2)), Dyn)
```

produces the following pretty-printed residual code:

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

However, a more desirable result is to propagate constant values directly and generate

```
let x₃ = 1 + 2 in x₃.
```

This performs constant propagation during code generation, yielding more compact residual code that is easier to optimize further, *e.g.*, by constant folding (Section 7).

We can achieve this by modifying `reflect` so that it does not introduce let-bindings for atomic expressions such as numbers. Instead of creating a fresh variable and inserting a let-binding in the meta-continuation, we pass the atomic expression directly to the continuation while reusing the same meta-continuation. This leads to the following revised definition of `reflect`:

```
def reflect(e: Expr, k: Cont, γ: MCont): Ans = e match
  case ENum(i) ⇒ k(VCode(ENum(i)), γ)
  case EBool(b) ⇒ k(VCode(EBool(b)), γ)
  case e ⇒
    val x = fresh()
    k(VCode(EVar(x)), { case VCode(body) ⇒ γ(VCode(ELet(x, e, body))) })
```

Refactoring for Modularity and Extensibility. Reflection is one of the key interfaces for controlling *when* to insert bindings of *what* expressions, and we will see many optimizations naturally occur at this stage. However, if we anticipate more optimizations, we may end up with a monolithic and complex `reflect` function that handles many special cases, which can be difficult to maintain and extend. For better modularity and extensibility, we refactor to separate the logic for different optimizations into smaller, composable units. Each optimization is expressed as a partial function that handles only the expressions that will be optimized in a specific way. For readability, we write the infix type operator `A ⇀ B` for `PartialFunction[A, B]` from Scala's standard library.

```
type ⇀[A, B] = PartialFunction[A, B]
```

For example, given a continuation and meta-continuation, `reflectAtom(k,γ)` is a partial function that only matches atomic expressions and performs constant propagation as described above.

```
def reflectAtom(k: Cont, γ: MCont): Expr ⇀ Ans =
  case ENum(i) ⇒ k(VCode(ENum(i)), γ)
  case EBool(b) ⇒ k(VCode(EBool(b)), γ)
```

The function `reflectDefault` implements the fallback behavior when no optimization applies. It is equivalent to the original `reflect`, but packaged as a partial function that matches all expressions.

```
def reflectDefault(k: Cont, γ: MCont): Expr ⇀ Ans =
  case e ⇒ ... // let-insertion as before
```

The final `reflect` function is obtained by folding over a list of such reflectors, using `reflectDefault` as the base case and trying each optimization in sequence until one matches the input expression.

```
def reflect(e: Expr, k: Cont, γ: MCont): Ans =
  val opts = List(reflectAtom, ... /* optimizations extended here */)
  opts.foldRight(reflectDefault(k, γ)) { (f, acc) ⇒ f(k, γ).orElse(acc) }(e)
```

It is worth noting that small reflectors can recursively call `reflect` to re-reflect a transformed expression. Later, we will see that this is useful for allowing β-inlining (Section 9) and code motion (Section 11) to trigger more optimizations.

5.2 Tracking Inserted Code

The second refactoring introduces a `Cache` data structure that tracks code inserted in the current scope. Conceptually, `Cache` is a bidirectional map between expressions and variable names: it supports (1) looking up the variable associated with a let-inserted expression and (2) retrieving the expression bound to a given variable. The following snippet defines `Cache` as an instance of `BiMap`, which maintains two maps for forward and backward lookups, and an `inv` method to invert the direction of the mapping:

```
case class BiMap[A, B](forward: Map[A, B], backward: Map[B, A]):
  def +(ab: (A, B)): BiMap[A, B] = BiMap(forward + ab, backward + ab.swap)
  def contains(a: A): Boolean = forward.contains(a)
  def apply(a: A): B = forward(a)
  def inv: BiMap[B, A] = BiMap(backward, forward)
type Cache = BiMap[Expr, String]
```

We refactor `eval`, `reflect`, `reify`, and the continuations with a contextual parameter `Cache` using Scala 3's `using` clause [64]. Contextual parameters allow the cache to be synthesized automatically, avoiding the need to pass it at every call site and requiring only signature changes in most cases. The cache is updated explicitly in `reflectDefault`, where we record the mapping from a newly inserted expression to its fresh variable. The following shows the updated body of `reflectDefault`:

```
def reflectDefault(k: Cont, γ: MCont)(using c: Cache): Expr ⇀ Ans =
  case e ⇒
    val x = fresh()
    k(VCode(EVar(x)), { case VCode(body) ⇒
      γ(VCode(ELet(x, e, body)))
    })(using c + (e ↦ x)) // explicitly pass the updated cache for x and e
```

Notably, the updated cache is explicitly passed to the continuation `k`, even though the let-expression has not yet been assembled in the meta-continuation. This allows the evaluation of the body to access and assume the inserted expression immediately, as if it were already in scope. The definitions of other reflectors do not need to update the cache, as they do not insert new let-bindings;

the only change is the signatures:

```
def reflectAtom(k: Cont, γ: MCont)(using Cache): Expr ⇀ Ans = ... // same as before
def reflect(e: Expr, k: Cont, γ: MCont)(using Cache): Ans = ...    // same as before
```

The continuation type is changed accordingly to indicate that `Cache` can be passed implicitly. The meta-continuation type remains unchanged, since it does not require access to the cache:

```
type Cont = (Value, MCont) ⇒ Cache ?⇒ Ans // ?⇒ indicates Cache is contextual
type MCont = Value ⇒ Ans
```

6 Optimization: Common Subexpression Elimination

With `Cache` tracking generated code, we can readily implement common subexpression elimination (CSE) by simply looking up the cache before generating new code for an expression. For example, without CSE, it is possible to generate the code that duplicates the same expression:

```
let x0 = ... in
let x1 = x0 + 1 in
let x2 = x0 + 1 in
let x3 = x1 * x2 in ...
```

and CSE allows reusing the previously generated code for `x0 + 1` and generating more efficient code by eliminating the redundant computation:

```
let x0 = ... in
let x1 = x0 + 1 in
let x3 = x1 * x1 in ...
```

To achieve this, we first define an extractor object [29] `CachedExpr` that checks whether an expression is pure and already exists in the scope. If so, it returns the let-bound object-code variable (`EVar`); otherwise, it returns `None`.

```
object CachedExpr:
  def unapply(e: Expr)(using c: Cache): Option[EVar] =
    if (e.isPure && c.contains(e)) Some(EVar(c(e)))
    else None
```

Then we can pattern match on the expression using `CachedExpr` in the following `reflectCse` combinator. If the pattern matches, we know `x` is an object-code variable that has been generated in the current scope for the same pure expression before, and we directly continue with `VCode(x)` without generating new let-bindings.

```
def reflectCse(k: Cont, γ: MCont)(using c: Cache): Expr ⇀ Ans =
  case CachedExpr(x) if Opt("cse") ⇒ k(VCode(x), γ)
```

Note that `reflectCse` only needs to handle the case guarded by this pattern, and the general `reflect` function (Section 5.1) will call other reflectors if the pattern does not match. We also use an auxiliary flag `Opt("opt-name"): Boolean` to check whether the optimization is enabled, controlled by the programmer.

CSE is a generic optimization and we do not restrict the syntactic form of the eligible common subexpressions. However, we assume that there is a heuristic or dependency analysis to determine whether an expression is pure, thus avoiding the unsound omission of side effects. A conservative one would be to only consider arithmetic expressions as pure, which is common in practice.

7 Optimization: Constant Folding

Like CSE, constant folding can be implemented as a small reflector that pattern-matches expressions with constant operands. A typical example is folding second-stage arithmetic whose operands are known constants. We begin with an extractor `StaticConst` that recognizes literal constant expressions, as well as variables that are let-bound to literals:

```
object StaticConst:
  def unapply(e: Expr)(using c: Cache): Option[Value] = e match
    case ENum(i) ⇒ Some(VNum(i))
    case EBool(b) ⇒ Some(VBool(b))
    case EVar(x) if c.inv.contains(x) ⇒ StaticConst.unapply(c.inv(x))
    case _ ⇒ None
```

Using the code cache (note that `c.inv` is a map from object variables to expressions), we can recursively look up the cache to check whether a variable is let-bound to a constant expression. This makes the extractor robust even in the absence of explicit constant propagation.

Using this extractor object, we can detect binary operations whose operands are static and evaluate them during reflection. The primitive operation is computed at the first stage using auxiliary function `evalPrim`, and its result is re-reflected as second-stage code:

```
def evalPrim(op: String, v1: Value, v2: Value): Value = ...
def reflectFolding(k: Cont, γ: MCont)(using c: Cache): Expr ⇀ Ans =
  case EBinOp(op, StaticConst(v1), StaticConst(v2), _) if Opt("cst-folding") ⇒
    evalPrim(op, v1, v2) match
      case VNum(i) ⇒ reflect(ENum(i), k, γ)
      case VBool(b) ⇒ reflect(EBool(b), k, γ)
```

This optimization applies only when all operands are static. In Section 10, we generalize the approach to partially-static data.

8 Optimization: Dead-Code Elimination

Dead-code elimination (DCE) is a useful optimization for reducing code size. Our staged evaluator currently introduces let-bindings to name intermediate results, but some of them may turn out to be unused. Eliminating such dead bindings would reduce code size and expose further optimization opportunities. We will show that DCE can be implemented in a surprisingly simple way using 2-CPS, which would be significantly more cumbersome in the WadlerFest evaluator (Section 2).

Unlike the stateful evaluator (Section 2), which unconditionally inserts a let-binding and returns its fresh second-stage variable, the 2-CPS interpreter gives us a chance to inspect the generated body before committing to the binding. To implement DCE, we as usual first provide the continuation with a fresh variable `VCode(EVar(x))`. The meta-continuation then receives the constructed body and can check whether `x` actually occurs. As with CSE, we assume a purity analysis to identify whether an expression is eligible for DCE. Since we eliminate only pure dead code and all variables are freshly generated, it suffices to examine data dependencies via free variables. If `x` does not occur in the body, the binding can be skipped; otherwise, we proceed with let-insertion as usual:

```
def reflectDce(k: Cont, γ: MCont)(using c: Cache): Expr ⇀ Ans =
  case e if Opt("dce") && e.isPure ⇒
    val x = fresh()
    k(VCode(EVar(x)), { case VCode(body) ⇒
      if (!body.freeVar(x)) then γ(VCode(body))
      else γ(VCode(ELet(x, e, body)))
    })(using c + (e ↦ x))
```

This works precisely because the continuation `k` is delimited: it corresponds exactly to the scope of `x`, and we always delimit the boundaries such as second-stage functions or conditional branches.

Without 2-CPS, implementing DCE would be significantly more cumbersome. In the stateful evaluator (Section 2), removing an unused binding would require non-trivial backtracking over the mutable state. Or alternatively, one would need a separate post-hoc DCE pass that analyzes

dependencies over the entire residual program. In contrast, the 2-CPS structure supports dead-code elimination locally and compositionally during evaluation.

9 Optimization: Inlining with Renormalization

Function inlining is a central optimization for functional programs: it replaces a call with the body of the callee, potentially exposing further simplifications. However, different from previous optimizations focused on a single expression, inlining involves code blocks consisting of multiple let-bindings. In the following, we explain how this can be implemented with only a handful of lines of code using continuations.

Non-Solution: Naive β-Inlining. Given our infrastructure, it appears to be straightforward to implement β-inlining optimization. We first define an extractor object `StaticLam` that recognizes static λ-terms, either syntactically or retrieved from the cache.

```
object StaticLam:
  def unapply(e: Expr)(using c: Cache): Option[ELam] = e match
    case e@ELam(_, _) ⇒ Some(e)
    case EVar(f) if c.inv.contains(f) ⇒ StaticLam.unapply(c.inv(f))
    case _ ⇒ None
```

We then pattern-match applications whose function part is static, and if so we invoke the current continuation with the function body substituted with the argument:

```
def reflectNaiveInline(k: Cont, y: MCont)(using c: Cache): Expr ⇀ Ans =
  case EApp(StaticLam(ELam(x, body)), arg, _) if Opt("fun-inline") ⇒
    k(body.subst(x, arg), k, y)
```

Here directly substituting the argument is safe, as it is either a variable or an atomic value.

Unfortunately, naive β-inlining breaks the ANF of generated code. ANF is not closed under the usual β-reduction [47, 76]: the right-hand side of a let-binding cannot be another nested let-binding. Consider the following example obtained by directly inlining the function `f` at the call-site:

```
// before the inlining optimization:
let f = λx. { let y = x + 1 in y } in
let r = f(2) in
let z = r * 3 in z
// naive inlining f at the call-site f(2) leads to:
let f = λx. { let y = x + 1 in y } in
let r = let y = 2 + 1 in y in
let z = r * 3 in z
```

The resulting term is no longer in ANF. One option is to switch to monadic normal form [37], which relaxes this restriction. Or alternatively, we can restore ANF by renormalizing the inlined code. We showcase the latter approach that can be elegantly implemented with continuations.

β-Inlining with Renormalization. The so-called renormalization essentially "re-reflects" every let-binding of the inlined body at the call-site. The function `renormalize` assumes its input `Expr` is already in ANF and recursively replays reflection. In the continuation of reflecting the right-hand side of a let-binding, we obtain the fresh variable `y` referring to the re-reflected expression, and before renormalizing the body, we need to substitute the old variable `x` with `y`:

```
def renormalize(e: Expr, k: Cont, y: MCont)(c: Cache): Ans = e match
  case ELet(x, rhs, body) ⇒
    reflect(rhs, { case (VCode(y), y₁) ⇒ renormalize(body.subst(x, y), k, y₁) }, y)
  case _ ⇒ k(VCode(e), y)
```

Because `renormalize` delegates to the top-level `reflect`, all existing optimizations are automatically applied during reconstruction. Inlining with renormalization is therefore just a substitution followed by renormalization:

```
def reflectInline(k: Cont, γ: MCont)(using c: Cache): Expr ⇀ Ans =
  case EApp(StaticLam(ELam(x, body)), arg) if Opt("fun-inline") ⇒
    renormalize(body.subst(x, arg), k, γ)
```

With renormalization, the earlier example is now optimized as follows. The downstream use of receiving variable `r` is substituted with the inner-most body of function `f`.

```
let f = λx. { let y = x + 1 in y } in
let y = 2 + 1 in
let z = y * 3 in z
```

Further constant folding and dead-code elimination will simplify it to object code `9`. Renormalization also applies to simplification of conditionals. When a branch can be chosen statically, we need to renormalize the selected branch:

```
def reflectCondSimp(k: Cont, γ: MCont)(using c: Cache): Expr ⇀ Ans =
  case ECond(StaticConst(VBool(b)), e₂, e₃, _) if Opt("simp-cond") ⇒
    if b then renormalize(e₂, k, γ) else renormalize(e₃, k, γ)
```

Detour: Conflict with Intensional Analysis. So far, we have used some analyses over the reflected code to guide optimizations, but these analyses are not exposed at the source level. In some situations, it would be desirable to let programmers perform such intensional analysis themselves and inspect the structure of generated code in the same language. However, exposing object-code structure conflicts with our built-in optimizations: a program could observe differences between optimized and unoptimized code and change its behavior accordingly, leading to unsoundness. We illustrate the issue following Taha [85].

Suppose we add a construct `EIsApp(e)` that tests if `e` evaluates to a second-stage application. Using our existing infrastructure, this construct can be implemented by looking up the code cache:

```
def isAppCont(k: Cont)(v: Value, γ: MCont)(using c: Cache): Ans = v match
  case VCode(EVar(x)) ⇒ c.inv(x) match
    case EApp(_, _, _) ⇒ k(VBool(true), γ)
    case _ ⇒ k(VBool(false), γ)
  case VCode(_) ⇒ k(VBool(false), γ)
```

We define an auxiliary function `isAppCont` that inspects the evaluated value of `e`. In `isAppCont`, because applications must be let-bound, any non-variable code cannot be an application, so we immediately return `false`. Otherwise, we look up the current cache to check whether the object-code variable binds to an application. Note that given a continuation `k`, `isAppCont(k)` has type `Cont`, therefore the evaluation case for `EIsApp(e)` can be implemented by evaluating `e` with `isAppCont(k)` as the continuation.

```
def eval(e: Expr, ρ: Env, k: Cont, γ: MCont)(using Cache): Ans = e match
  case EIsApp(e) ⇒ eval(e, ρ, isAppCont(k), γ)
  // other cases unchanged and omitted ...
```

Now consider the following program written in the pseudo surface syntax which constructs a second-stage application (both the function and argument are lifted, and `@²` denotes second-stage application), and inspects it with `isApp`.

```
lift(isApp(lift(λx. x) @² lift(42)))
```

Our staged evaluator (*i.e.*, compiler) may legitimately choose to inline the application *or* keep it as an application. Thus, depending on the optimization decisions made by the compiler, the above

program may yield both `true` and `false`! In other words, optimization could change the result of evaluation, rendering the transformation unsound.

As Taha [84] noted, the tension between general intensional analysis and intrinsic optimizations does not rule out domain-specific optimizations for particular code datatypes. Exploring how to support such domain-specific optimizations within our framework is left for future work.

10 Optimization: Simplifying Partially-Static Data

Constant folding described in Section 7 works well when all operands are statically known, for example `2 + 3`. However, expressions such as `(2 + x) + 3` contain both *static* components, such as `2` and `3`, and *dynamic* components whose values are not known at the current stage, such as `x`. Such expressions are instances of partially-static data and still admit further optimization. In this example, `(2 + x) + 3` can be simplified to `x + 5`, or equivalently `5 + x`, by exploiting the associativity and commutativity of addition as a commutative monoid. That said, so far our optimizations have not exploited any algebraic laws of the operations, which we will improve in this section.

To integrate such an algebraic simplification into our staged evaluation, the optimization needs to operate at reflection time, interact well with let-insertion, and produce valid object code. It is not entirely straightforward, since at reflection time, the reflected expression may use intermediate results introduced by previous let-insertions, and we must choose a suitable algebraic structure for optimizing the reflected expression.

A common approach is to convert partially static expressions into a canonical representation, simplify it in that representation by exploiting algebraic laws, and then residualize it back to the normal form of code. Yallop et al. [103] provided a generic framework for realizing this idea in Haskell. We adapt the core representation from this framework to our 2-CPS staged evaluator and use it to implement partially-static data optimizations.

Representing Partially-Static Data. Yallop et al. [103] proposed to model partially-static data as a free extension, namely the coproduct of an algebra of static values and its corresponding free algebra generated by dynamic computation. This coproduct gives a uniform representation for constructing partially-static structures and generating code from them, avoiding ad hoc data structures. We use it to represent partially static data via the following type classes:

```
trait Coproduct[Alg[_], A: Alg, B: Alg]:
  type Coprod // the carrier
  given ac: Alg[Coprod]
  def inl(a: A): Coprod
  def inr(b: B): Coprod
  def eva[C: Alg](fa: A ⇒ C, fb: B ⇒ C,
                  co: Coprod): C
```

```
trait Free[Alg[_], T]:
  type FreeA
  given af: Alg[FreeA]
  def pvar(x: T): FreeA
  def pbind[C: Alg](freeA: FreeA,
                    f: T ⇒ C): C
```

The trait `Coproduct` requires the carrier type `Coprod`, two injectors `inl`/`inr`, and the eliminator `eva` from the coproduct into another algebra `C`, which is typically the generated code. The trait `Free` is the generic free algebra interface, which requires the underlying carrier `FreeA`, function `pvar` to inject a dynamic variable, and `pbind` to interpret free terms in any algebra.

Instantiation. With such a generic interface, we can specialize the coproduct to our framework: injecting into the left lifts static data (*e.g.*, `ENum`) into the algebra, while injecting to the right lifts dynamic components (*e.g.*, `EVar`) into the corresponding free algebra. The `eva` function can serve as the residualization procedure that converts the coproduct representation back to `Expr`.

With this representation in place, it is straightforward to instantiate the framework for a specific algebraic structure, such as an additive commutative monoid or a ring. The instantiation also

decides the concrete canonical representation of the coproduct, for example, a multinomial for a ring. This representation is informed by the admitted algebraic laws of the underlying algebra.

For our language, we instantiate the static part of the coproduct as the integer ring. The free algebra for the dynamic component is instantiated as `Free[Ring, EVar]`, where `EVar` ranges over the variables introduced by let-insertion. The carrier type `Coprod` is multinomials, serving as the canonical representation of partially-static ring structures.

```
given [A, B](using r: Ring[A], f: FreeRing[B]): Coproduct[Ring, A, f.FreeA] =
  new Coproduct[Ring, A, f.FreeA] { type Coprod = Multinomial[B, A] ... }
```

We elide the full definition of the coproduct as well as the multinomial representation, which can be found in the artifact [97].

Reflection-Time Optimization for Rings. To optimize an `Expr` using the ring algebra, we define a partially-static data evaluation function `evalRingPS` that re-interprets the original expression as the ring-algebraic operation over the corresponding coproduct, rather than computing a concrete value as `evalPrim` does in constant folding.

```
def evalRingPS(expr: Expr)(using c: Cache): RingNF = expr match
  case EBinOp("+", e₁, e₂, _) ⇒ evalRingPS(e₁) ⊕ evalRingPS(e₂)
  case EBinOp("*", e₁, e₂, _) ⇒ evalRingPS(e₁) ⊗ evalRingPS(e₂)
  case EVar(x) if c.inv.contains(x) && c.inv(x).isPure ⇒ evalRingPS(c.inv(x))
  case ENum(i) ⇒ sta(i)
  case e ⇒ dyn(e)
```

The type `RingNF` abbreviates the carrier of the coproduct instance for the ring algebra, whose underlying representation is a multinomial. Static values are injected into the left of `RingNF` using `sta`, whereas non-algebraic operations and impure expressions are treated as dynamic values and injected into the right side using `dyn`.

Finally, we define a reflector `reflectPS` that applies this optimization during reflection. To achieve this, we define an extractor object `OptimizableRingExpr` that recognizes ring expressions that are not already in the normal form, and then returns the expression optimized through the canonical representation computed by `evalRingPS`.

```
def reflectPS(k: Cont, γ: MCont)(using c: Cache): Expr ⇀ Ans =
  case OptimizableRingExpr(optExpr) if Opt("ps") ⇒ renormalize(optExpr, k, γ)
```

After obtaining the optimized expression `optExpr`, we use `renormalize` (introduced in Section 9) to reflect it into the residual program conforming to ANF.

Supporting Mixed Algebraic Structures. In a simple language like ours, it may suffice to focus on a single algebra, such as rings, and treat all other expressions as dynamic terms. However, the object language may involve operators from different algebraic structures, and it would be desirable to optimize them all.

To optimize partially static expressions over mixed algebras, we could extend `evalRingPS` with additional cases for their operations. However, when multiple algebras are involved, different subexpressions may belong to different structures and satisfy different equational laws, requiring distinct canonical representations. In some cases, such as the integer ring and monoid, the subsuming algebra and its canonical representation can be used uniformly for these subexpressions. In general, however, combining different canonical representations into a single one is not straightforward. An alternative is to use equality graphs (e-graphs) [62, 91, 99] as a general representation. E-graphs would also make it possible to customize the algebraic laws and corresponding optimizations simply by adding rewrite rules. We leave the integration of e-graph-based optimization into 2-CPS staging and its interaction with let-insertion to future work.

```
def lift(k: Cont)(v: Value, γ: MCont)(using c: Cache): Ans = v match
  // other cases unchanged and omitted ...
  case VClo(x, body, ρ) ⇒
    val y = fresh()
    reify(body, ρ + (x ↦ VCode(EVar(y))), { case VCode(e) ⇒
      if (Opt("lam-hoist"))
        val c₁ = BiMap((EVar(y) ↦ y))
        hoist(e, { case (VCode(e₁), γ1) ⇒ reflect(ELam(y, e₁), idK, γ1) },
                 { case VCode(e₂) ⇒ renormalize(e₂, k, γ) })(using c₁)
      else
        reflect(ELam(y, e), k, γ) // same as before if hoisting is disabled
    })
```

Fig. 7. Lifting values with λ-hoisting.

11 Optimization: Code Motion

So far our optimizations under ANF have been mostly local: they decide *when* to insert bindings (*e.g.*, constant propagation) or *what* to insert (*e.g.*, constant folding and β-inlining). Another class of optimizations concerns *where* code is placed. These code motion optimizations affect the scheduling of computation and can enable further improvements such as CSE.

Code motion comes in many forms, such as code hoisting, sinking, or loop-invariant code motion. Their common theme is moving code across scope boundaries. In this section, we use λ-hoisting, *i.e.*, lifting computations out of function bodies, as a running example to demonstrate how code motion can be realized in our framework. The key idea, again, is to leverage the two continuations to separate let-bindings into different contexts.

λ-Hoisting. If a computation inside a function does not depend on the function argument and is effect-free, it can be beneficial to lift it outside the function, avoiding redundant computation for each call. For example, consider the following two snippets contrasting the effect of λ-hoisting:

```
// object code without hoisting:
let f = λx.
  let t = 1000 * 1000 in
  let z = x + t in z in
let v = f(1) in
let w = f(2) in ...
// optimized code after hoisting:
let t = 1000 * 1000 in
let f = λx. let z = x + t in z in
let v = f(1) in
let w = f(2) in ...
```

The optimal code only computes `1000 * 1000` once, while the non-optimal code computes it once *per call* to `f`.

To implement λ-hoisting, we modify `lift` when converting a closure into an object λ-term (Figure 7). In the continuation of reification, we have obtained the generated function body `e`. If hoisting is enabled, a helper function `hoist` is applied to transform `e`; otherwise, the body is reflected unchanged. The role of `hoist` is to partition let-bindings, based on a dependency analysis, into those that should remain inside the function body and those that can be moved outside the function. It takes two continuations: (1) The first receives the residual function body $\texttt{e}_1$ and a meta-continuation γ_1, then reflects an `ELam` term with `idK` (*i.e.*, $\texttt{(v,}\gamma\texttt{)} \Rightarrow \gamma\texttt{(v)}$) and γ_1 which then inserts

the lifted bindings, and (2) the second (*i.e.*, the meta-continuation) receives $\texttt{e}_2$, then `renormalizes` the entire function term that is now surrounded by the lifted bindings. The renormalization is necessary to restore ANF after hoisting, since the lifted term $\texttt{e}_2$ may not be a single expression, but a block of let-bindings. When calling `hoist`, we also construct a new cache $\texttt{c}_1$ that only tracks the function parameter `y`, since we now enter a new scope and the argument is the root of dependency.

The implementation of `hoist` relies on an analysis `hoistable` to decide whether a let-binding can be hoisted. In either case, we recurse on the body, but *accumulate bindings in different continuations.* Hoistable bindings are accumulated in the meta-continuation, which is the continuation after the function body is reflected (*i.e.*, the meta-continuation received by the continuation passed to `hoist`), thus placed outside the function; the rest are accumulated in the ordinary continuation and remain inside the function body, eventually passed to the caller's continuation.

```
def hoist(e: Expr, k: Cont, γ: MCont)(using c: Cache): Ans = e match
  case ELet(x, rhs, body) if hoistable(rhs) ⇒
    hoist(body, k, { case VCode(lam) ⇒
      γ(VCode(ELet(x, rhs, lam)))
    })(using c + (rhs ↦ x))
  case ELet(x, rhs, body) ⇒
    hoist(body, { case (VCode(body1), γ1) ⇒
      k(VCode(ELet(x, rhs, body1)), γ1)
    }, γ)(using c + (rhs ↦ x))
  case _ ⇒ k(VCode(e), γ)
```

One simple heuristic for determining whether a let-binding is hoistable is to require that the expression is pure and independent of variables already scheduled in the function body, which we can check via the cache:

```
def hoistable(e: Expr)(using c: Cache): Boolean =
  e.isPure && e.freeVar.intersect(c.allVals).isEmpty
```

Discussion. We have now reached the end of our optimization tour. Our goal has not been to develop efficient implementations with production-ready heuristics, but to show *how* the 2-CPS structure naturally accommodates these optimizations as modular extensions, using only simple data structures. Still, practical optimizers must make additional choices of when to apply these optimizations.

One classic distinction is between shrinking and non-shrinking optimizations [5]. Function inlining, for example, is non-shrinking: it may improve performance, but must be controlled by heuristics such as function size and call frequency to avoid excessive code growth. Hoisting raises a different trade-off. In the example above, a binding is lifted at the cost of introducing a free variable for the function. In this particular case, no closure allocation is needed, since the call site can pass the free variable directly rather than packaging a closure [77] as the call site and function share the same scope. In general, however, hoisting may introduce runtime closure allocation.

Both DCE (Section 8) and λ-hoisting rely on a dependency analysis, and are dual in spirit. DCE removes bindings that are unused by computations *below*; hoisting lifts bindings that are independent of computations *above* within a scope. Both are driven by dependency analysis, and both fit elegantly into our framework through the two continuations. Our dependency analysis tracks only data dependencies; however, for languages with references or other effects, eliminating or moving effectful code would require a sophisticated analysis that also tracks effect dependencies. We leave this extension to future work.

12 Extensions

We have presented our let-inserting staged evaluation semantics from various perspectives, together with a spectrum of object-code optimizations. Now, we continue the journey by discussing a few orthogonal extensions to the core calculus, including mutable state, recursion, and the simulation of MetaML-style quotations via by-name evaluation.

Mutable References and Recursive Functions. It is possible to extend our evaluators and abstract machines to support recursive functions and ML-style mutable references using standard techniques. The extension adds a store to the semantics, which maps locations to values. Environments now map variables to locations instead of values. Values are allocated on the store, and the store is threaded through the evaluation as an additional component of the answer type. The following definitions show the type signature of the extended evaluator:

```
type Env = Map[String, Loc]
type Store = Map[Loc, Value]
type Ans = (Value, Store)
type Cont = (Value, Store, MCont) ⇒ Cache ?⇒ Ans
type MCont = (Value, Store) ⇒ Ans
def eval(e: Expr, ρ: Env, σ: Store, k: Cont, γ: MCont)(using Cache): Ans = ...
```

Constructs for ML-style mutable references (*i.e.*, `ref`, read, and write) and recursive functions (*e.g.*, `fix`) can be straightforwardly added to the syntax. Their semantics follows standard techniques [30, 68]. Once we have first-class mutable references across stages, their interaction can be subtle: for example, lifting a reference from the first stage to the second stage would require cross-stage persistence [36], and storing open code values in references can lead to scope extrusion issues. This pearl does not address these issues, which remain active areas of research [12, 56, 86].

By-Name Staging. Readers may notice that the evaluators and the CEKM machine uniformly enforce call-by-value for both static and dynamic computation, with call-by-value for the latter enforced via let-insertion. In contrast, MetaML-style quotation delays evaluation: quoting `<e>` does not evaluate `e`, but produces a *syntactic* code value that can be copied.

This difference is visible with the following example where a first-stage function constructs second-stage addition. The argument to the function is a quoted application `<f()>`. In a MetaML-style system with quotation and splicing (written ~ below), the application `f()` is duplicated in the generated code, potentially duplicating effects if `f()` has any:

```
// MetaML-style
(λx. < ~x + ~x >) <f()> ⟶ f() + f()
```

A direct translation of the MetaML-style program into our calculus is as follows: instead of using explicit quotation and splicing, addition and application (written `@`) are annotated with superscript ^{2}, indicating that they are dynamic.

```
// Our calculus
(λx. x +² x) f@²()      ⟶ let x = f() in let y = x + x in y
```

The let-inserting staged evaluation ensures the application is evaluated once and shared.

Just as MetaML-style staging could enforce call-by-value evaluation through let-insertion [51], a uniform let-inserting semantics can also support by-name staging via explicit thunking and forcing. This is also the pattern used in practice: LMS exploits Scala by-name parameters to express control flow (*e.g.*, conditionals) that delays evaluation into specific scopes.

To model this behavior, we extend the language with by-name functions (or dually, by-name applications), which evaluate to closures `VBNClo`. When such a closure is applied, the argument term is packaged as a thunk `VThunk` and forced only when the corresponding variable is looked up in the environment. This semantics is implemented by the following change to the evaluator:

```
def eval(e: Expr, ρ: Env, k: Cont, γ: MCont)(using Cache): Ans = e match
  case EVar(x) ⇒ ρ(x) match
    case VThunk(e, ρ) ⇒ eval(e, ρ, k, γ)          // forcing
    case v ⇒ k(v, γ)
  case EApp(e₁, e₂, Sta) ⇒
    eval(e₁, ρ, {
      case (VBNClo(x, body, ρf), γ1) ⇒            // by-name application (thunking)
        eval(body, ρf + (x ↦ VThunk(e₂, ρ)), k, γ1)
      case (VClo(x, body, ρf), γ1) ⇒              // by-value application
        eval(e₂, ρ, { case (v₂, γ2) ⇒ eval(body, ρf + (x ↦ v₂), k, γ2) }, γ1)
    }, γ)
  ...                                             // other cases same as before
```

With this extension, MetaML-style quotations and syntactic duplication can be simulated by writing by-name functions. For example, the previous program, rewritten using a by-name abstraction, generates the same duplicated applications as MetaML-style systems:

```
// Our calculus extended with by-name functions
(λbx. x +² x) f@²() ⟶ let x = f() in let y = f() in let z = x + y in z
```

13 Related Work

In the Introduction and throughout the paper, we have already discussed the connections between our work and a wide range of related work. Here, we discuss a few additional connections and related work that we did not have the chance to discuss in the main text.

Semantics. Our reconstruction in Section 2 closely follows the stateful evaluator for LMS-style staging presented by Rompf [71] and Amin and Rompf [4], with a few presentational differences. First, we use named variables in both the evaluator and the representation of accumulated let bindings, whereas Rompf [71] uses de Bruijn levels. Second, we extend the core language with recursion along with mutable references, while Amin and Rompf [4] support recursion via self references. The calculus of Amin and Rompf [4] also generalizes the evaluator to multiple stages; the same extension could be adopted here, but we present the two-stage setting for clarity.

In this pearl, we have focused on building executable semantic artifacts for LMS-style staged evaluation. For let-inserting staged evaluation of this kind, Amin and Rompf [4] developed the $\lambda_{\uparrow\downarrow}$-calculus and its reduction semantics, and Tan and Wei [90] further developed type systems and examined its meta-theoretic semantics-preservation property, *i.e.*, the generated code is contextually equivalent to the original annotation-erased program.

For MetaML-style languages, Taha [84] presents both big-step and reduction semantics. Ge and Garcia [35] develop an environmental abstract machine semantics for MetaML. These accounts do not address manual or automatic let-insertion. In the presence of effects, delimited control and the CPS hierarchy are established techniques for let-insertion, both in partial evaluation [19, 27, 38, 55] and in various multi-stage programming systems [46, 51, 52, 74]. In this pearl, the same ideas are made explicit through a 2-CPS definitional interpreter and its defunctionalized abstract machine.

Internal Representations and Optimizations. Our development uses ANF as a textual intermediate representation for object code. The Squid [65, 66] staging library also implements an ANF internal representation. In contrast, LMS [74] represents code using a graph-based intermediate representation, also known as sea-of-nodes IR [9, 16]. Graph-based representations permit flexible code motion and many optimizations efficiently. For example, common subexpression elimination can be realized via hash consing, β-inlining via graph substitution/rewiring, and dead-code elimination via a reachability analysis.

Although our object code is represented as ANF, it can also be read as a dependency graph: each let-binding corresponds to a node, and free variable occurrences induce edges to earlier bindings. It would be interesting to investigate a functional graph-based representation that preserves the modularity of our evaluator while enabling more efficient analyses.

The code motion optimization is found in other staging systems too. For example, MetaOCaml uses `genlet` to lift bindings as far outward as permitted by data dependencies [51]. For partially static data, we adopt the free extension of algebras proposed by Yallop et al. [103]. Thiemann [92] proposed the notion of partially-static operations, exploiting algebraic laws in partial evaluation.

Partial Evaluation and Normalization by Evaluation. LMS draws inspiration from type-directed partial evaluation [19] and normalization-by-evaluation [6], and so do our staged evaluators. For instance, lifting a closure to an object λ-term corresponds to two-level η-expansion in TDPE. More broadly, the staged evaluator can be viewed as an online partial evaluator for a binding-time annotated language [63]. Recent work has continued to explore evaluation-based accounts of staging; for example, Kovács [54] proposed a "staging-by-evaluation" algorithm, analogous to normalization by evaluation. Our setting differs in that it preserves the order of effects with automatic let-insertion and integrates optimizations directly into staged evaluation.

14 Conclusion

This pearl begins by reimagining what a standalone language for LMS-style staging and optimizations would look like. To that end, we developed a sequence of definitional interpreters and abstract machines for let-inserting staged evaluation through the lens of continuations, together with a spectrum of object-code optimizations using functional programming techniques. Many of these optimizations admit simple and elegant implementations as small extensions to the 2-CPS staged evaluator. We also toured a few exotic aspects of staging, such as the conflict between intensional analysis and optimization.

By providing an operational account of LMS-style staged evaluation equipped with automatic let-insertion and object-code optimizations, this pearl bridges the gap between practical staging framework implementations and their semantic artifacts. We hope this pearl serves both as a clean description and a practical reference for implementing LMS-style staged evaluation in other languages, or just as a standalone language in its own right.

Data Availability Statement

Our artifact includes the full implementation of the evaluators, abstract machines, optimizations and a test suite. The artifact is available at https://doi.org/10.5281/zenodo.20373714 [97] and https://github.com/instar-lang/instar-lang/tree/icfp26.

Acknowledgments

We would like to thank the ICFP anonymous reviewers for their insightful comments and suggestions, which have improved the quality of this paper. We also thank Tiark Rompf and Cameron Wong for comments and discussions.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Uppsala, Sweden) *(PPDP '03)*. Association for Computing Machinery, New York, NY, USA, 8–19. doi:10.1145/888251.888254

[2] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. 2004. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Inform. Process. Lett.* 90, 5 (2004), 223–232. doi:10.1016/j.ipl.2004.02.012

[3] Nada Amin, William E. Byrd, and Tiark Rompf. 2019. Lightweight Functional Logic Meta-Programming. In *APLAS (Lecture Notes in Computer Science, Vol. 11893)*. Springer, 225–243.

[4] Nada Amin and Tiark Rompf. 2018. Collapsing towers of interpreters. *Proc. ACM Program. Lang.* 2, POPL (2018), 52:1–52:33.

[5] Andrew W. Appel and Trevor Jim. 1997. Shrinking lambda Expressions in Linear Time. *J. Funct. Program.* 7, 5 (1997), 515–540.

[6] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. 1998. Normalisation by Evaluation. In *Prospects for Hardware Foundations (Lecture Notes in Computer Science, Vol. 1546)*. Springer, 117–137.

[7] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. 2005. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. *Log. Methods Comput. Sci.* 1 (2005). https://api.semanticscholar.org/CorpusID:5693133

[8] William J. Bowman. 2024. A Low-Level Look at A-Normal Form. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 165–191.

[9] Oliver Bracevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for Impure Higher-Order Languages: Making Aggressive Optimizations Affordable with Precise Effect Dependencies. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 400–430.

[10] Ajay Brahmakshatriya and Saman P. Amarasinghe. 2021. BuildIt: A Type-Based Multi-stage Programming Framework for Code Generation in C++. In *CGO*. IEEE, 39–51.

[11] Ajay Brahmakshatriya, Christopher Rinard, Manya Ghobadi, and Saman P. Amarasinghe. 2024. NetBlocks: Staging Layouts for High-Performance Custom Host Network Stacks. *Proc. ACM Program. Lang.* 8, PLDI (2024), 467–491.

[12] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. 2000. Closed Types as a Simple Approach to Safe Imperative Multi-stage Programming. In *ICALP (Lecture Notes in Computer Science)*. Springer, 25–36.

[13] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *GPCE (Lecture Notes in Computer Science, Vol. 2830)*. Springer, 57–76.

[14] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543. doi:10.1017/S0956796809007205

[15] Tsung-Ju Chiang and Ningning Xie. 2025. Multi-stage Programming with Splice Variables. *Proc. ACM Program. Lang.* 9, ICFP (2025), 400–434.

[16] Cliff Click and Michael Paleczny. 1995. A Simple Graph-Based Intermediate Representation. In *Intermediate Representations Workshop*. ACM, 35–49.

[17] Charles Consel and Olivier Danvy. 1991. For a Better Support of Static Data Flow. In *FPCA (Lecture Notes in Computer Science, Vol. 523)*. Springer, 496–519.

[18] Olivier Danvy. 1996. Pragmatics of Type-Directed Partial Evaluation. In *Dagstuhl Seminar on Partial Evaluation (Lecture Notes in Computer Science, Vol. 1110)*. Springer, 73–94.

[19] Olivier Danvy. 1996. Type-Directed Partial Evaluation. In *POPL*. ACM Press, 242–257.

[20] Olivier Danvy. 1999. Type-Directed Partial Evaluation. In *Partial Evaluation*, John Hatcliff, Torben Æ Mogensen, and Peter Thiemann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 367–411.

[21] Olivier Danvy. 2006. An analytical approach to program as data objects.

[22] Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *LISP and Functional Programming*. ACM, 151–160.

[23] Olivier Danvy and Andrzej Filinski. 1992. Representing Control: A Study of the CPS Transformation. *Math. Struct. Comput. Sci.* 2, 4 (1992), 361–391.

[24] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. 1996. Eta-Expansion Does The Trick. *ACM Trans. Program. Lang. Syst.* 18, 6 (1996), 730–751.

[25] Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at Work. In *PPDP*. ACM, 162–174.

[26] Rowan Davies and Frank Pfenning. 1996. A Modal Analysis of Staged Computation. In *POPL*. ACM Press, 258–270.

[27] Dirk Dussart and Peter Thiemann. 1997. Partial evaluation for higher-order languages with state. *Berichte des Wilhelm-Schickard-Instituts WSI-97-XX, Universit at T ubingen* (1997).

[28] Conal Elliott, Sigbjørn Finne, and Oege de Moor. 2003. Compiling embedded languages. *J. Funct. Program.* 13, 3 (2003), 455–481.

[29] Burak Emir, Martin Odersky, and John Williams. 2007. Matching Objects with Patterns. In *ECOOP (Lecture Notes in Computer Science, Vol. 4609)*. Springer, 273–298.

[30] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press.

[31] Andrzej Filinski. 1994. Representing Monads. In *POPL*. ACM Press, 446–457.

[32] Andrzej Filinski. 2001. Normalization by Evaluation for the Computational Lambda-Calculus. In *TLCA (Lecture Notes in Computer Science, Vol. 2044)*. Springer, 151–165.

[33] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *PLDI*. ACM, 237–247.

[34] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.* 1, ICFP (2017), 13:1–13:29.
[35] Rui Ge and Ronald Garcia. 2017. Refining semantics for multi-stage programming. In *GPCE*. ACM, 2–14.
[36] Yuichiro Hanada and Atsushi Igarashi. 2014. On Cross-Stage Persistence in Multi-Stage Programming. In *FLOPS (Lecture Notes in Computer Science)*. Springer, 103–118.
[37] John Hatcliff and Olivier Danvy. 1994. A Generic Account of Continuation-Passing Styles. In *POPL*. ACM Press, 458–471.
[38] John Hatcliff and Olivier Danvy. 1997. A Computational Formalization for Partial Evaluation. *Math. Struct. Comput. Sci.* 7, 5 (1997), 507–541.
[39] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic embedding of DSLs. In *GPCE*. ACM, 137–148.
[40] Jun Inoue and Walid Taha. 2012. Reasoning about Multi-stage Programs. In *ESOP (Lecture Notes in Computer Science, Vol. 7211)*. Springer, 357–376.
[41] Jun Inoue and Walid Taha. 2016. Reasoning about multi-stage programs. *J. Funct. Program.* 26 (2016), e22.
[42] Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2022. Mœbius: metaprogramming using contextual types: the stage where system f can pattern match on itself. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–27.
[43] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Prentice Hall.
[44] Ulrik Jørring and William L. Scherlis. 1986. Compilers and Staging Transformations. In *POPL*. ACM Press, 86–96.
[45] Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. 2014. Yin-yang: concealing the deep embedding of DSLs. In *GPCE*. ACM, 73–82.
[46] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2011. Shifting the stage - Staging with delimited control. *J. Funct. Program.* 21, 6 (2011), 617–662.
[47] Andrew Kennedy. 2007. Compiling with continuations, continued. In *ICFP*. ACM, 177–190.
[48] Oleg Kiselyov. 2012. Delimited control in OCaml, abstractly and concretely. *Theor. Comput. Sci.* 435 (2012), 56–76.
[49] Oleg Kiselyov. 2018. Reconciling Abstraction with High Performance: A MetaOCaml approach. *Found. Trends Program. Lang.* 5, 1 (2018), 1–101.
[50] Oleg Kiselyov. 2023. MetaOCaml Theory and Implementation. *CoRR* abs/2309.08207 (2023).
[51] Oleg Kiselyov. 2026. MetaOCaml: ten years later - System description. *Sci. Comput. Program.* 250 (2026), 103397.
[52] Oleg Kiselyov and Jeremy Yallop. 2022. let (rec) insertion without Effects, Lights or Magic. *CoRR* abs/2201.00495 (2022).
[53] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: a language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 296–311. doi:10.1145/3192366.3192379
[54] András Kovács. 2022. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.* 6, ICFP (2022), 540–569.
[55] Julia L. Lawall and Peter Thiemann. 1997. Sound Specialization in the Presence of Computational Effects. In *TACS (Lecture Notes in Computer Science, Vol. 1281)*. Springer, 165–190.
[56] Michael Lee, Ningning Xie, Oleg Kiselyov, and Jeremy Yallop. 2026. Handling Scope Checks: A Comparative Framework for Dynamic Scope Extrusion Checks. *Proc. ACM Program. Lang.* 10, POPL (2026), 1123–1152.
[57] Daan Leijen and Erik Meijer. 1999. Domain specific embedded compilers. In *DSL*. ACM, 109–122.
[58] Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). 2016. *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Lecture Notes in Computer Science, Vol. 9600. Springer.
[59] Geoffrey Mainland. 2007. Why it's nice to be quoted: quasiquoting for haskell. In *Haskell*. ACM, 73–82.
[60] Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. 2019. AutoGraph: Imperative-style Coding with Graph-based Performance. In *SysML*. mlsys.org.
[61] Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. 2012. Scala-virtualized. In *PEPM*. ACM, 117–120.
[62] Greg Nelson and Derek C. Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *J. ACM* 27, 2 (1980), 356–364.
[63] Flemming Nielson and Hanne Riis Nielson. 1992. *Two-level functional languages*. Cambridge tracts in theoretical computer science, Vol. 34. Cambridge University Press.

[64] Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2018. Simplicitly: foundations and applications of implicit function types. *Proc. ACM Program. Lang.* 2, POPL (2018), 42:1–42:29.

[65] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. Quoted staged rewriting: a practical approach to library-defined optimizations. In *GPCE*. ACM, 131–145.

[66] Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2018. Unifying analytic and statically-typed quasiquotes. *Proc. ACM Program. Lang.* 2, POPL (2018), 13:1–13:33.

[67] Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *PLDI*. ACM, 199–208.

[68] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

[69] John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference (2)*. ACM, 717–740.

[70] Tiark Rompf. 2016. The Essence of Multi-stage Evaluation in LMS. In *A List of Successes That Can Change the World (Lecture Notes in Computer Science, Vol. 9600)*. Springer, 318–335.

[71] Tiark Rompf. 2016. Reflections on LMS: exploring front-end alternatives. In *SCALA@SPLASH*. ACM, 41–50.

[72] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized: linguistic reuse for deep embeddings. *High. Order Symb. Comput.* 25, 1 (2012), 165–207.

[73] Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Arvind K. Sujeeth, Manohar Jonnalagedda, Nada Amin, Georg Ofenbeck, Alen Stojanov, Yannis Klonatos, Mohammad Dashti, Christoph Koch, Markus Püschel, and Kunle Olukotun. 2015. Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems. In *SNAPL (LIPIcs, Vol. 32)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 238–261.

[74] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*. ACM, 127–136.

[75] Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* 55, 6 (2012), 121–130.

[76] Amr Sabry and Philip Wadler. 1997. A Reflection on Call-by-Value. *ACM Trans. Program. Lang. Syst.* 19, 6 (1997), 916–941.

[77] Zhong Shao and Andrew W. Appel. 2000. Efficient and safe-for-space closure conversion. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 129–161.

[78] Tim Sheard and Simon L. Peyton Jones. 2002. Template meta-programming for Haskell. *ACM SIGPLAN Notices* 37, 12 (2002), 60–75.

[79] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI*. ACM, 206–221.

[80] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A practical unification of multi-stage programming and macros. In *GPCE*. ACM, 14–27.

[81] Nicolas Stucki, Jonathan Immanuel Brachthäuser, and Martin Odersky. 2021. Multi-stage programming with generative and analytical macros. In *GPCE*. ACM, 110–122.

[82] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s (2014), 134:1–134:25.

[83] Eijiro Sumii and Naoki Kobayashi. 2001. A Hybrid Approach to Online and Offline Partial Evaluation. *High. Order Symb. Comput.* 14, 2-3 (2001), 101–142.

[84] Walid Taha. 1999. *Multi-stage programming: its theory and applications*. Oregon Graduate Institute of Science and Technology.

[85] Walid Taha. 2000. A Sound Reduction Semantics for Untyped CBN Multi-stage Computation. Or, the Theory of MetaML is Non-trivial (Extended Abstract). In *PEPM*. ACM, 34–43.

[86] Walid Taha and Michael Florentin Nielsen. 2003. Environment classifiers. In *POPL*. ACM, 26–37.

[87] Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. In *PEPM*. ACM, 203–217.

[88] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242.

[89] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD Conference*. ACM, 307–322.

[90] Jun Tan and Guannan Wei. 2026. When Do Staging Annotations Preserve Semantics? Mechanizing Typed Semantics-Preserving Multi-Stage Programming with Let-Insertion (Extended Version). arXiv:2606.30854 [cs.PL] https://arxiv.org/abs/2606.30854

[91] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *POPL*. ACM, 264–276.

[92] Peter Thiemann. 2013. Partially static operations. In *PEPM*. ACM, 75–76.

[93] Marcos Viera and Alberto Pardo. 2006. A multi-stage language with intensional analysis. In *GPCE*. ACM, 11–20.

[94] Fei Wang, Daniel Zheng, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2019. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.* 3, ICFP (2019), 96:1–96:31.
[95] Guannan Wei, Oliver Bracevac, Shangyin Tan, and Tiark Rompf. 2020. Compiling symbolic execution with staging and algebraic effects. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 164:1–164:33.
[96] Guannan Wei, Songlin Jia, Ruiqi Gao, Haotian Deng, Shangyin Tan, Oliver Bracevac, and Tiark Rompf. 2023. Compiling Parallel Symbolic Execution with Continuations. In *ICSE*. IEEE, 1316–1328.
[97] Guannan Wei, Jun Tan, and Dinghong Zhong. 2026. *Artifact for Let It Be Optimized: Building Multi-Stage Evaluators with Let-Insertion and Optimizations in Small Pieces (Functional Pearl)*. doi:10.5281/zenodo.20534869
[98] Edwin M. Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. 2010. Mint: Java multi-stage programming using weak separability. (2010), 400–411.
[99] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29.
[100] Ningning Xie, Matthew Pickering, Andres Löh, Nicolas Wu, Jeremy Yallop, and Meng Wang. 2022. Staging with class: a specification for typed template Haskell. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30.
[101] Ningning Xie, Leo White, Olivier Nicole, and Jeremy Yallop. 2023. MacoCaml: Staging Composable and Compilable Macros. *Proc. ACM Program. Lang.* 7, ICFP (2023), 604–648.
[102] Jeremy Yallop and Oleg Kiselyov. 2019. Generating mutually recursive definitions. In *PEPM@POPL*. ACM, 75–81.
[103] Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. 2018. Partially-static data as free extension of algebras. *Proc. ACM Program. Lang.* 2, ICFP (2018), 100:1–100:30.

Received 2026-02-19; accepted 2026-05-13