Data Representation in Functional Languages

Christophe Gyurgyik

Stanford University

1 Introduction

Functional programming languages often exhibit inferior runtime performance compared to their imperative counterparts, which more directly encode the von Neumann architecture. A principal culprit is data representation: functional languages typically box values behind pointers, incurring indirection that obfuscates compiler analysis and heap allocation overhead that modern hardware penalizes severely.

These functional languages can be partitioned into two camps based on their default representation strategy. Languages that are *boxed-by-default*, e.g., Haskell and OCaml, represent values uniformly as pointers, simplifying polymorphism and enabling separate compilation but requiring explicit mechanisms to recover performance. Languages that are *unboxed-by-default*, e.g., Morphic and MLton, represent values directly, achieving better performance at the cost of whole-program analysis and restricted polymorphism. These two camps face fundamentally different constraints and consequently adopt very different approaches.

We begin by examining two boxed-by-default languages. First, we discuss Haskell's approach to introducing unboxed values as first-class [22], and extensions to this work to reduce constraints [7]. Second, we turn to OCaml's modal memory management, which employs type qualifiers tracking locality, uniqueness, and affinity to enable safe stack allocation and in-place mutation while respecting the language's existing design constraints [18]. We then examine Morphic, an unboxed-by-default language that uses *lambda set specialization* to defunctionalize higher-order programs, achieving the benefits of specialization while maintaining the expressiveness of first-class functions [3]. Finally, we conclude by distilling lessons learned from these three approaches into a desideratum for the ideal, performance-oriented functional language.

2 The Representation Problem

Boxing values by default simplifies the compilation of functional languages considerably. A uniform representation means a single calling convention suffices for all functions, which in turn yields straightforward implementations of parametric polymorphism and existential types. Separate compilation becomes tractable because module boundaries need not expose representation details. Advanced type system features such as polymorphic recursion and higher-rank polymorphism, which can generate infinitely many type instantiations at compile time, remain viable because all instantiations share a unified calling convention.

However, these benefits come at a cost. When every value is boxed, even simple arithmetic incurs pointer indirection and heap allocation. To recover performance, boxed-by-default languages must introduce mechanisms for selective unboxing. This proves surprisingly burdensome: the very uniformity that made compilation easy now stands in the way. Parametric polymorphism, for instance, assumes that type variables can be instantiated with any type, but now types may have different bit widths and calling conventions.

Unboxed-by-default languages sidestep these difficulties by relaxing the constraints that boxed-by-default languages abide by. They enforce their own requirements:

1. **Strict evaluation.** While not universal in box-by-default, lazy evaluation imposes an additional axis of representation complexity, whereas strict evaluation permits values to be passed directly.
2. **Finite types.** Only a finite number of types may exist at compile time. This rules out polymorphic recursion and certain uses of higher-rank polymorphism.
3. **Whole-program analysis.** Separate compilation becomes impossible, but the compiler gains visibility into all type instantiations and can specialize accordingly.

Under these assumptions, compile-time monomorphization can essentially guarantee unboxed representations for all non-recursive types. However, whole-program analysis demands longer compilation times, increased memory footprint during building, and inflated binary sizes [28]. Nevertheless, for performance-critical applications, these costs are often acceptable [25, 26].

3 First-Class Unboxed Values

Haskell is a non-strict, purely functional language that historically represented all values as boxed and lifted. Non-strictness introduces an additional dimension to the representation problem: a value of lifted type may be undefined ($\perp$), representing a non-terminating computation. This leads to the invariant that *lifted types must be boxed*, since the runtime must be able to represent a thunk that has not yet been evaluated. An unlifted type, by contrast, may be boxed or unboxed.

Peyton Jones and Launchbury [22] introduce unboxed values as first-class citizens in Haskell, enabling programmers to opt into unboxed representations where performance demands it. We examine their approach and its evolution into modern GHC's treatment of representation polymorphism.

Motivation

Consider the following Haskell function:

```
f :: Int → Int → Int
f x y = x + (x - y)
```

For simplicity, we assume that operands are evaluated left to right, though this is not required since no ordering is imposed by the non-strict semantics of the language. We give the following operational interpretation:

1. unbox `x` (lhs)
2. evaluate `x` to `x'`
3. unbox `x` (rhs)
4. unbox `y`
5. evaluate `y` to `y'`
6. compute `r' = x' - y'`
7. box `r'` as `u'`
8. unbox `u'`
9. compute `r'' = x' + u'`
10. box `r''`

This sequence involves four indirect memory accesses and a redundant intermediate boxing operation. In a strict language with unboxed integers, the same computation would reduce to two machine instructions.

Unboxing Constraints

Three constraints govern when unboxing is possible in Haskell. First, since unboxed values cannot represent $\perp$, the compiler must guarantee that an unboxed value is fully evaluated. This can be established via a strictness analysis or, more recently, through explicit strictness annotations such as bang patterns (`!`) and `seq`.

Second, polymorphic functions cannot directly manipulate unboxed values, since different unboxed types may have different sizes and calling conventions. Three approaches do exist to overcome this limitation:

1. *Runtime tagging*: Attach a discriminating tag indicating the type's representation, as in the ML family [17]. This requires all types to share a uniform width and incurs runtime overhead.
2. *Monomorphization*: Specialize each polymorphic function for every type instantiation. This conflicts with Haskell's support for separate compilation and polymorphic recursion.
3. *Type passing*: Explicitly provide type information at each call site of a polymorphic function.

Peyton Jones and Launchbury [22] declined all three, instead restricting polymorphic type variables to range only over boxed types. Finally, recursive types must remain boxed, since their size cannot be determined statically.

Introducing Unboxed Values

The key insight of Peyton Jones and Launchbury [22] is to make boxing and unboxing explicit in Haskell's intermediate language Core. Evaluation order is encoded via `case` expressions: the construct `case n of I# n# -> ...` forces evaluation of `n` and extracts its unboxed contents into `n#`. Boxedness is reflected in the type system by distinguishing `Int` (boxed, lifted) from `Int#` (unboxed, unlifted). The motivating example translates to:

```
case x of Int x'# → case (
  case x of Int x''# →
    case y of Int y'# →
      case (x''# -# y'#) of Int r'#
  ) of Int u'# →
    case (x'# +# u'#) of Int r''#
```

With boxing and unboxing made explicit, standard compiler transformations can eliminate redundant operations. After applying common subexpression elimination, case-of-case, and case fusion, the optimized code becomes:

```
-- 1. eliminate common scrutinisations [4.1]
-- 2. case-of-case transformation [4.2, P1]
-- 3. case-fusion [4.2, P2]
case x of Int x'# →
  case y of Int y'# →
    case (x'# -# y'#) of r'# →
      case (#x' + #r') of #r'' →
        Int #r''
```

The function still unboxes its arguments and reboxes the result, since its type signature demands boxed integers. However, the intermediate boxing has been eliminated. Finally, changing the signature to `Int#→Int#→Int#` would eliminate even the boundary conversions.

Type System Integration

Enforcing the unboxing constraints requires only modest changes to the type system. The rule for type instantiation restricts polymorphic type variables to boxed types π:

$$\textsc{Spec}\ \frac{A \vdash e : \forall \alpha.\, \sigma}{A \vdash e : \sigma[\pi/\alpha]}$$

Function application splits into two rules:

$$\textsc{App}\ \frac{A \vdash e_1 : \pi \to \tau \qquad A \vdash e_2 : \pi}{A \vdash e_1\ e_2 : \tau}$$

$$\textsc{App\#}\ \frac{A \vdash e_1 : \nu \to \tau \qquad A \vdash h : \nu}{A \vdash e_1\ h : \tau}$$

The first handles boxed arguments; the second requires the argument h to be in head-normal form, ensuring that unboxed values are never passed unevaluated. Similar adjustments apply to `let` and `letrec`. Together with a syntactic check rejecting recursive unboxed types, these rules suffice to ensure well-formedness.

Levity Polymorphism

Unboxed types remain part of modern Haskell, but the original restrictions proved overly draconian. Eisenberg and Peyton Jones [7] introduced *levity polymorphism*, allowing limited polymorphism over representations. The key observation is that some functions genuinely do not care about representation: they never move or store their representation-polymorphic arguments in ways that depend on layout.

The kind system tracks representation via the `TYPE` constructor, e.g., `Int :: TYPE LiftedRep` and `Int# :: TYPE IntRep`. Here, we say that `Int` has a lifted representation, while `Int#` has an integer representation. A function can then abstract over representation:

```
($) :: ∀
  (r :: Rep)
  (a :: TYPE LiftedRep)
  (b :: TYPE r). (a → b) → a → b
f ($) x = f x
```

Here the return type `b` is representation-polymorphic, but the input `a` must be lifted. This is well-formed because the function merely returns `b` without storing or binding it (assuming tail call optimization). Conversely, the following program is rejected:

```
id :: ∀ (r :: Rep) (a :: TYPE r). a → a
id x = x
```

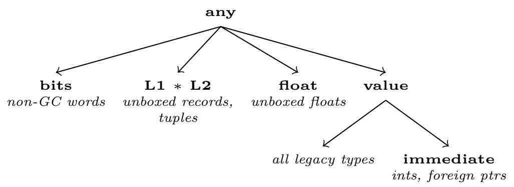
The calling conventions would require knowing the representation of `a` to allocate appropriate storage on the stack. The restriction is precisely characterized: a function may be representation-polymorphic only if its representation-polymorphic variables appear in positions where the generated code is independent of layout. This remains quite restrictive, but it carves out useful territory that the original work forbade entirely.

Beyond Compile-Time Monomorphization

An alternative approach, exemplified by C#/.NET, defers monomorphization to runtime via a JIT compiler [13]. This enables unrestricted polymorphism: even types unknown at compile time (due to separate compilation or first-class polymorphism) can be specialized on demand. The cost is carrying type information at runtime, but the expressiveness gain is substantial. For languages unwilling to sacrifice separate compilation or non-standard typing features, runtime specialization may be the only path to full unboxing.

4 Modal Memory Management

OCaml is a strict, call-by-value language supporting both functional and imperative paradigms. Like Haskell, it permits infinitely many types at compile time and supports separate compilation, precluding wholesale monomorphization. OCaml does support unboxed types through a layout system similar in spirit to Haskell's levity polymorphism, albeit less general[1]: a hierarchy of *sub-layouts* patches specific unboxing cases rather than providing a unified treatment [5, 20]. A subset of this hierarchy appears below:



The root `any` represents layouts unknown at compile time; code operating at this layout is rejected, since the compiler cannot generate appropriate instructions without knowing the representation. Below it, the hierarchy branches into `value` and several unboxed layouts. The `value` layout encompasses all types safe for garbage collection: pointers to heap-allocated data and *immediates*, which are 63-bit integers distinguished from pointers by a low tag bit. The garbage collector inspects this bit to avoid following immediates as if they were pointers. Unboxed layouts such as `float64` and `bits64` fall outside `value` and require careful handling: an unboxed float, for instance, might coincidentally resemble a valid pointer, so the collector must never encounter it in a context expecting a `value`. The distinction between floating-point and bit layouts reflects architectural constraints, since some processors mandate different registers for each.

Given the lack of white papers available on unboxed types, we focus instead on a complementary approach: OCaml's *modal* type system, which reduces heap allocations by tracking properties of values that enable stack allocation and in-place mutation [18].

[1]Remarkably, the original RFC [20] points out that storing two 32-bit integers (typically 8B) in a record resulted in 80B of overhead.

Motivation

Idiomatic functional programming incurs heap allocations that imperative code would avoid. Consider a function that shifts the color of each pixel in a list:

```
type Number { int, int }

let shift_color offset pixels =
  List.map (fun {r; g; b} → Color.clamp {
    r = r + offset.r;
    g = g + offset.g;
    b = b + offset.b;
  }) pixels
```

Three sources of allocation lurk here: each new `Color` record, the spine of the result list, and the closure capturing `offset`. This can be fixed by using the imperative paradigms of OCaml, but the goal of modal memory management [18] is to recover efficiency without abandoning functional style code.

The approach introduces three *modes* as type qualifiers: locality, uniqueness, and affinity. Each mode forms a lattice with subtyping, and together they enable the compiler to prove that certain values can be stack-allocated or updated in place. The benefits closely resemble those of unboxing.

Locality

Locality governs the lexical region in which a value may be referenced, and thereby determines whether it can be stack-allocated. A value qualified as `@local` is guaranteed not to escape its enclosing region, thus the compiler may place it on the stack. A value qualified as `@global` may be referenced anywhere and must be heap-allocated. The subtyping relation `global <: local` holds: a global value can always be used where a local one is expected, but not vice versa. Locality is a *deep* property: if a value is local, so are all its constituents.

A third mode, introduced by the keyword `exclave`, permits values to escape their immediate region when in tail position. This reconciles stack allocation with tail-call optimization: without it, a tail-recursive function returning a locally-allocated value would be forced to copy that value to the heap before returning.

This design echoes region-based memory management as in Cyclone [8], though with less generality. Cyclone supports arbitrarily nested regions with explicit lifetime parameters. OCaml offers only the binary distinction between local and global with a special case for tail recursion.

Uniqueness

Uniqueness tracks whether a value has at most one reference pointing to it, enabling safe in-place mutation. A value qualified as `@unique` is guaranteed to have no aliases; the compiler may overwrite it without copying. A value qualified as `@aliased` may have multiple references and must be treated immutably. The subtyping relation `unique <: aliased` holds, and uniqueness is deep.

Children may override their parent's uniqueness: the spine of a list might be unique while its elements are aliased. The converse is forbidden: an aliased container cannot guarantee unique elements, since multiple paths to the container imply multiple paths to its contents.

OCaml permits *conditional ownership*, where a value's fate depends on runtime control flow:

```
val consume : 'a @ unique → unit
val observe : 'a @ aliased → unit

let f v b = if b
            then consume v
            else (observe v; observe v)
```

Here `v` is either consumed uniquely or aliased twice, depending on `b`. This flexibility stems from OCaml's garbage collector, which handles deallocation regardless of ownership. In a language with explicit memory management like Rust, such a program would be ill-formed: ownership, a linear property, must be statically determined to know when to free.

Affinity

Affinity constrains how many times a value may be used in the future. A value qualified as `@many` may be used arbitrarily often, while one qualified as `@once` may be used at most once. The subtyping relation `many <: once` holds. This mode prevents accidental aliasing of unique values through closures:

```
let xs @ unique : int list = [1;2;3] in
let f = fun v → v :: xs in
let ys = f 4 in
let zs = f 5 (* error if f is marked @once *)
```

Without an affinity constraint, calling `f` twice creates two references to the unique list `xs`. Marking the closure as `@once` catches the error statically.

Uniqueness versus Linearity

A brief digression on semantics: uniqueness and linearity are often conflated, but they differ in subtle ways when unrestricted values coexist in the system [19]. Both restrict contraction (prohibiting duplication), but they make guarantees in opposite temporal directions. Uniqueness asserts something about the past: no other references to this value currently exist. Linearity asserts something about the future: this value will be consumed exactly once.

The distinction manifests in what weakenings are permitted. A unique value can be weakened to an aliased value (we simply forget the uniqueness guarantee), but a linear value cannot be weakened to an unrestricted one (doing so would permit the value to go unconsumed). Conversely, an unrestricted value can be strengthened to a linear one (we promise to consume it), but not to a unique one (we cannot retroactively guarantee no aliases exist).

OCaml's design reflects this: uniqueness and affinity are orthogonal modes, and their combination handles cases that either alone would misclassify. Modes form a lattice over three axes (affinity, uniqueness, locality), and closures require a *dagger operation* to mediate between them:

$$\mathsf{once}^\dagger := \mathsf{unique} \qquad \mathsf{many}^\dagger := \mathsf{aliased}$$

The dagger connects affinity (how many times a closure may be invoked) to uniqueness (how many references exist to a value): a closure callable many times will access its captures many times, effectively aliasing them.

This reasoning is formalized in the *lock* judgment $\Gamma, \langle\mu\rangle$, which transforms a context Γ when variables are captured by a closure of mode $\mu = (a, u, l)$. For each captured binding $x : \tau \ @ \ (a', u', l')$, the lock coerces uniqueness to $u' \vee a^\dagger$. If the closure has affinity many, then $a^\dagger = \mathsf{aliased}$, forcing any captured variable to become aliased regardless of its original uniqueness.

Mode Inference and Polymorphism

The compiler infers modes via constraint solving, defaulting to the legacy triple `(many, aliased, global)` when annotations are absent. This ensures backward compatibility, i.e., unannotated code behaves as before. Programmers may opt into stricter modes where performance demands, and the type system guarantees these modes are well-formed.

True mode polymorphism remains future work. The current workaround is a template mechanism that generates multiple monomorphic copies [21]:

```
let%template[@mode m = (global, local)]
  id : 'a. 'a @ m → 'a @ m = fun x → x
```

This expands to two definitions, one for each mode in the list. The approach is ad hoc but pragmatic, providing mode-polymorphic behavior.

Impact

The empirical results, while narrow in scope, demonstrate the potential. In a benchmark saturating the garbage collector, modal management yielded a 9% runtime improvement. The implementation effort is substantial, and remained incomplete at the time of publication. Mode polymorphism, which the authors identify as critical for practical adoption, was deferred to future work. Moreover, 85 functions in their codebase were duplicated across different locality modes, a direct consequence of this limitation. In general, the system achieves its goals, but the interaction of three mode axes, partial context joins, dagger-mediated locks, and a nine-element semiring with coaliased grades represents a considerable departure from Hindley-Milner.

5 Lambda Set Specialization

Morphic [3] takes the opposite stance from Haskell and OCaml: rather than boxing by default and selectively unboxing, it strictly unboxes everything except recursive data types. This is feasible because Morphic accepts the constraints outlined in Section 2, namely strict evaluation, a finite number of types at compile time, and whole-program analysis. The result is a language where the default representation matches that of imperative languages.

A challenge for any unboxed-by-default language is higher-order functions, a key ingredient in a functional language. A closure is traditionally represented as a pointer to a heap-allocated structure containing the function code and its captured environment. Morphic eliminates this indirection through *defunctionalization* [23], a transformation that converts higher-order programs into first-order ones. The novelty of Morphic lies in *lambda set specialization*, which combines defunctionalization with specialization to avoid the performance degradation that naive defunctionalization would introduce.

Defunctionalization

Consider a higher-order function that applies its argument twice:

```
twice(f: Int → Int, x: Int): Int = f(f(x))

let add: Int → Int = \x → x + 2 in
let mul: Int → Int = \x → x * 2 in
(twice(add, 5), twice(mul, 3))
```

Classical *monovariant* defunctionalization replaces function values with tags and introduces a dispatch function:

```
type AddOrMul { 'add | 'mul }

twice(tag: AddOrMul, x: Int): Int =
  match tag {
    'add → (x + 2) + 2,
    'mul → (x * 2) * 2,
  }

(twice('add, 5), twice('mul, 5))
```

The program is now first-order, but we have introduced a runtime dispatch that the original code did not require.

Each call site of `twice` passes a statically known function, yet the defunctionalized version must branch on the tag. Specialization recovers the lost efficiency:

```
twice$_{add}$(x: Int): Int = (x + 2) + 2
twice$_{mul}$(x: Int): Int = (x * 2) * 2

(twice$_{add}$(5), twice$_{mul}$(3))
```

This is exactly what a monomorphizing compiler like C++ would produce. Now, consider a variant where the closure is chosen at runtime:

```
let add: Int → Int = \x → x + 2 in
let mul: Int → Int = \x → x * 2 in
let f: Int → Int =
  if b { add } else { mul } in
twice(f, 5)
```

Here specialization is impossible: `f` could be either `add` or `mul`, so the dispatch is genuinely needed. The challenge is to specialize where possible and fall back to dispatch only where necessary.

Lambda Set Annotations

Morphic's solution is to track, at the type level, which lambdas a function value might contain. Each function type carries a *lambda set annotation* denoting the set of possible closures:

```
twice$\langle\alpha\rangle$(f: Int $\xrightarrow{\alpha}$ Int, x: Int): Int
  = (f as $\alpha$)((f as $\alpha$)(x))

let add: Int $\xrightarrow{\{\lambda x \to x+2\}}$ Int = \x → x + 2 in
let mul: Int $\xrightarrow{\{\lambda x \to x*2\}}$ Int = \x → x * 2 in
(
  twice$\langle\{\lambda x \to x+2\}\rangle$(add, 5),
  twice$\langle\{\lambda x \to x*2\}\rangle$(mul, 3)
)
```

The function `twice` is now polymorphic over lambda sets via the parameter α. At each call site, α is instantiated with a concrete set. When that set is a singleton, the compiler specializes: no dispatch is needed. When the set contains multiple lambdas, as in the runtime-conditional example, the compiler generates a sum type and dispatch code:

```
let f: Int $\xrightarrow{\{\lambda x \to x+2, \lambda x \to x*2\}}$ Int =
  if b { add } else { mul } in
twice$\langle\{\lambda x \to x+2, \lambda x \to x*2\}\rangle$(f, 5)
```

Crucially, even in this case the closure and its environment remain unboxed; only the control flow requires a runtime decision and the decision is local. This approach generalizes naturally. Lambda set annotations flow through the program via type inference, and monomorphization instantiates each polymorphic definition at every lambda set it is used with. The result is specialization wherever the analysis can prove it safe, with dispatch reserved for genuinely dynamic cases.

An interesting observation made by Brandon et al. [3] is that the unit of specialization should be strongly connected components (SCC) rather than individual functions. Type inference requires that the definition dependency graph admit a topological ordering, since inferring the type of a definition depends on the signatures of its dependencies. Mutual recursion violates this requirement by introducing cycles. The solution is to collapse each SCC into a single definition, with each mutually recursive function becoming a let-bound lambda in that definition's body. This restores the necessary topological structure while allowing the specialization machinery to proceed unchanged.

Evaluation

Lambda set specialization yields substantial improvements over monovariant defunctionalization. Compared to MLton, another unboxed-by-default compiler that defunctionalizes without specializing, Morphic achieves speedups ranging from 0.91× to 6.85× across benchmarks. Binary sizes sometimes *decrease* despite the additional specialization: when specialized functions are small enough to inline, LLVM eliminates the original function entirely. If there exists only a single call, this results in a smaller binary.

The Cost of Uniformity

Unboxed-by-default languages sacrifice representation choice. In Morphic, everything is unboxed except recursive types and the built-in `Array` type. This uniformity can be suboptimal. Consider mutually recursive types:

```
type Even { Even(Int, Odd) }
type Odd  { Odd(Int, Even) }
```

Morphic boxes both `Even` and `Odd` to break the cycle, but only one indirection is necessary for finite representation. A smarter analysis could box only one, but even this choice can be difficult to make optimally. More significantly, boxing sometimes improves performance. Consider an AST with debugging metadata:

```
type Span { Span(Int, Array Byte, ...) }
type Expr {
  Const(Int, Span),
  Var(Int, Span),
  Add(Expr, Expr, Span),
  ...
}
```

If `Span` is rarely accessed, inlining it into every node degrades cache locality. Boxing `Span` would keep the hot fields local. This is standard practice in optimizing compilers [15], but Morphic cannot express it. Conversely, MLton can express this, but doesn't always guarantee unboxing where one might expect it. For example, it will box a pair of fixed-precision integers `(r1, r2)` if `r1` and `r2` are not statically known values. In general, determining where a value will live in MLton is up to the compiler, thus the choice is still illusive. We provide concrete examples of this in Appendix A.

Boxing also enables sharing. The classic example is string interning, where identical strings share a single allocation. Morphic supports this for string literals but cannot generalize to user-defined types. Unboxing is not universally beneficial. The optimal representation depends on access patterns, data sizes, and cache behavior. A language that forces a single choice foregoes optimizations that require the other. To motivate these claims, we provide empirical evidence in Appendix B.

6 Toward Parity Performance

We have surveyed three approaches to improving the performance of functional languages: Haskell's first-class unboxed types with levity polymorphism, OCaml's modal memory management, and Morphic's lambda set specialization. Each makes different tradeoffs between language expressiveness and performance. We now ask, what would a functional language designed primarily for runtime performance look like?

The Case for Unboxing by Default

From a purely performance-oriented perspective, the Morphic approach is the most promising path toward parity with imperative languages. Several observations support this claim. First, strict evaluation is essential. Non-strictness introduces an unavoidable level of indirection: every value must be represented as a thunk that may or may not have been evaluated. Haskell programmers routinely annotate their code with bang patterns, `seq`, and strictness pragmas to recover the performance that strictness would have provided by default [12]. Therefore, a language prioritizing runtime performance should be strict.

A(nother) brief digression: Lennart Augustsson has a blog post defending the merits of non-strictness in a functional programming language [1]. This was in response to Bob Harper griping about laziness; both contain entertaining perspectives. His biggest complaint was the lack of function reuse in strict languages. Consider the following example:

```
any :: (a → Bool) → [a] → Bool
any p xs = or . map p xs
```

In a strict language, this call will first scan the list entirely with `map`, then perform the outer reduction using `or`. Thus, to perform the optimal choice, i.e., exit early, it must be rewritten entirely:

```
any p [] = False
any p (x:xs) = x || any p xs
```

Perhaps one solution to this inconvenience is recent work on mechanizing such transformations in the more general case of trees [24]. Specifically, `or` is associative and has an annihilator, thus a compiler can (and should) infer it is possible to exit early.

Second, monomorphization eliminates the overhead of uniform representation. When the compiler specializes each polymorphic function for its concrete type instantiations, it can use the natural machine representation for each type: integers in registers, floats in vector units, structs laid out contiguously. The cost is compile time and binary size, but these are acceptable tradeoffs for performance-critical applications.

Third, defunctionalization with specialization, as in Morphic's lambda set specialization approach, eliminates the overhead of closures. Higher-order functions compile to first-order code with static dispatch wherever possible, and to unboxed sum types with minimal dispatch overhead where runtime choice is unavoidable.

Together, these yield code that a C programmer would recognize: values stored directly, functions called without indirection, and fewer hidden allocations. This is the baseline from which a performance-oriented functional language should start.

The Need for Choice in Data Representation

Yet, Morphic's uniformity is also its limitation. In unboxing everything by default, it foregoes optimizations that require boxing. As we state in Section 5, there are scenarios where boxing yields better performance, e.g., cold metadata that degrades cache locality when inlined, shared immutable structures that reduce memory usage, and mutually recursive types that require fewer indirections than Morphic's conservative analysis provides.

This observation is not novel. Past work on efficient functional programming language compilation noted that aggressive unboxing can be counterproductive, e.g., when register spilling is excessive [16, 17]. The tradeoff depends on how values flow through the program, a property that varies with the algorithm and input data.

The takeaway is that no single representation policy is universally optimal. The choice of data representation depends on characteristics of the input data, the algorithm, and the underlying architecture [9]. A language that forces uniformity, whether boxing everything or unboxing everything, will be suboptimal for some evaluation context. What we need is not a different default, but the ability to choose.

Representation Parametricity

Before continuing, we must clarify what it means for code to be independent of representation. The term *representation polymorphism* conflates two distinct notions. In the Haskell literature [6, 7], representation polymorphism is an *operational*[2] guarantee: a single compiled function works correctly across multiple representations at runtime. The type system verifies that representation-polymorphic code performs no operations requiring knowledge of layout, ensuring that one piece of machine code suffices for all instantiations. This is a strong property, but it is also extremely restrictive, i.e., many useful functions cannot satisfy it.

In monomorphizing languages, this operational goal is abandoned by design. Every polymorphic function is compiled once per instantiation and as a result no code sharing occurs. Therefore, only a *denotational* semantics remains meaningful. Following Wadler's characterization of parametric polymorphism as information hiding [27], we can define representation parametricity as the guarantee that source code cannot observe representation. A function is representation-parametric if its semantics is identical regardless of how its arguments are laid out in memory or derived. This semantic definition is the one that matters for optimization. If the algorithm written in the core language cannot observe representation, then the compiler or performance engineer is free to choose any layout that preserves semantics.

Fortunately, representation-parametric code is natural in functional languages. Pure functional programs operate on *values*, not *locations*, e.g., providing immutable bindings rather than mutable cells and algebraic data types rather than pointers. The operations that violate representation parametricity are precisely the operations that functional languages typically omit. Imperative code is necessarily steeped in explicit data representation choices; functional code often, by default, is not.

[2]I may be perverting the terms operational and denotational here. However, I have not seen anyone else make this distinction before so was compelled to choose my own nomenclature.

Separating Algorithm from Representation

This observation suggests a design principle. Namely, if functional code is naturally representation-parametric, then representation choices can be made *separately* from algorithmic logic. This separation echoes an early insight of Hoare's: recursive data structures should be understood abstractly, independent of their concrete representation in memory [11]. Namely, programs should only analyze such structures through case analysis and recursion. Consequently, the algorithm describes *what* to compute over inductively defined structure; the representation describes *how* that structure is represented.

The precedent for such separation exists in other domains. For example, Legion separates logical data regions from physical layouts [2] in distributed computing and achieves performance competitive with hand-tuned code while preserving high-level abstractions. We propose an analogous separation for data structures in functional languages, where semantics-preserving layout transformations are specified independent of the algorithm. The source program remains representation-parametric; the compiler exploits this freedom.

Notably, this principle of *data independence* is well-established in domain-specific languages [4, 9, 14], but has yet to permeate general-purpose functional languages. Consider the following example, where an algorithm is written in the core language and a separate "data representation specification" is provided:

```
type Span {
  Span (
    source_begin : Int, source_end : Int,
    file : String, module : String,
  )
}
type Expr {
  Const(Int, Span),
  Var(String, Span),
  Add(Expr, Expr, Span),
}
simplify : Expr → Expr
simplify expr = ...
```

```
layout Expr(
  Const(_, s), Var(_, s), Add(_, _, s)
) ← apply Box(s)
```

The schedule specifies that `Span` fields within `Expr` should be boxed, improving cache locality for passes that traverse expressions without inspecting spans. The algorithm is unchanged; only the physical layout differs. More ambitious transformations should be expressible in the same framework:

```
type Point {
  Point(x: Float, y: Float, z: Float)
}

type Points { Points(Array Point) }

f(ps: Points) → Bool
f ps = ...
```

```
layout Points(
  Array Point(x,y,z)
) ← split (x,y,z)

// Points(
//   Array x:Float,
//   Array y:Float,
//   Array z:Float
// )
```

This schedule transforms an array of points into three separate arrays of coordinates, improving vectorization for algorithms that access fields independently. This is a semantics-preserving transformation that current functional languages either cannot express without changing the core language or *always* provide by default [10] thus again eliminates choice. We assume the compiler will generate the necessary memory offsets, e.g., `p[i].x` to `p.x[i]`.

Recursive algebraic data types present another fundamental representational challenge that all functional languages must confront. Unlike primitive values or simple records, a recursive type such as a linked list or binary tree cannot be fully unboxed: doing so would require infinite space at compile time. Consequently, every language we have surveyed mandates that at least one component within any recursive cycle be boxed [3, 22]. This constraint foregoes a suite of optimizations that imperative languages exploit with impunity. Consider the canonical example of a list:

```
type List a { Nil, Cons(a, List a) }
```

In Haskell, OCaml, and Morphic alike, the recursive reference to `List a` within the `Cons` constructor is represented as a pointer to heap-allocated memory. Each node lives at an arbitrary address determined by the allocator, and traversal requires chasing pointers across potentially disparate memory regions. The performance implications are twofold: indirection latency and cache locality degradation. Imperative languages, by contrast, routinely circumvent these penalties through techniques that exploit a crucial observation: at runtime, recursive data structures are often *finitely bounded*. By allocating nodes contiguously in some arena, 64-bit pointers may be replaced by more compact indices. Moreover some offsets can be inferred from structural information and thus require no additional space, e.g., a child placed directly next to their parent in memory. Both of these data representation optimizations are demonstrated in Appendix B.

When no schedule is provided, the compiler applies sensible defaults: unboxed for non-recursive types, boxed for recursive types, standard discriminated unions for sum types. Programmers opt into more sophisticated layouts only where justified. Finally this provides an additional benefit, namely the algorithm must only be defined once. This is crucial for optimizing an algorithm for a particular evaluation context, where data characteristics and architectural limitations will influence the choice of layout.

Externalizing Compilation

Representation is not the only axis where uniformity may be suboptimal. Compilation strategy itself admits choice. Monomorphization yields the fastest code, but at the cost of compilation time and binary size. For some functions, the cost may exceed the benefit. A library function instantiated at dozens of types might unacceptably bloat the binary, especially when it lies on a cold path. Moreover, optimizing compilers are expensive, and whole-program analysis doubly so. The Pareto principle applies; in some cases, a small fraction of code dominates runtime. The programmer, equipped with domain knowledge and profiling tools, can demarcate these hot regions more precisely. Expensive analyses should focus there, while cold code compiles with cheaper strategies.

Just as Haskell and OCaml allow selective specialization within a boxed-by-default world, a monomorphizing language could allow selective uniformity. The compiler could default to specialization but accept annotations indicating where shared code is acceptable, or where whole-program visibility is unnecessary. The broader principle is that compilation strategy, like representation, should be under programmer control. Externalizing the compiler exposes the decisions that affect performance as first-class configuration, rather than burying them in heuristics that cannot anticipate every evaluation context. The specification of data representation, discussed above, is one particularly consequential form of this externalization.

Toward Performant Functional Languages

The preceding discussion suggests design principles for a functional language oriented toward runtime performance parity with imperative code. Many questions remain open:

- What is the specification language for data representation transformations? In domain-specific languages, a restricted domain delimits the space of valid (and useful) transformations, e.g., The Tensor Algebra Compiler operates over tensors, and SQL over relations. A general-purpose functional language, however, enjoys no such luxury. User-defined algebraic data types can encode arbitrary structure. Too restrictive, and the language isn't very useful; too expressive, and Rice's theorem foregoes any hope of guaranteeing semantic preservation.
- How should memory management interact with representation choice? Certain layout transformations assume particular allocation disciplines. Unboxing recursive structures into contiguous arenas, for instance, requires lifetimes amenable to bulk deallocation. A coherent design must ensure that representation specifications compose with the memory management strategy rather than conflicting with it.
- Where should the boundary lie between compiler inference and programmer control? Ideally, a gradual workflow exists: naive code compiles with sensible defaults, and the programmer introduces algorithm-agnostic representation optimizations as performance demands arise.

While essential, these design principles do not exhaust the performance-critical aspects of functional language design, e.g., garbage collection strategies. Nevertheless, we contend that a language adhering to these principles would represent meaningful progress toward closing the performance gap between functional and imperative code, and, more importantly, doing so without forfeiting the abstractions that make functional programming worthwhile.

References

[1] Lennart Augustsson. More points for lazy evaluation. https://augustss.blogspot.com/2011/05/more-points-for-lazy-evaluation-in.html, May 2 2011. Blog post on lazy evaluation in Haskell.

[2] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[3] William Brandon, Benjamin Driscoll, Frank Dai, Wilson Berkow, and Mae Milano. Better defunctionalization through lambda set specialization. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi: 10.1145/3591260. URL https://doi.org/10.1145/3591260.

[4] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13 (6):377–387, 1970.

[5] Stephen Dolan. Unboxed types for ocaml. Jane Street Tech Talk, 2019. https://www.janestreet.com/tech-talks/unboxed-types-for-ocaml/.

[6] Paul Downen, Zena M. Ariola, Simon Peyton Jones, and Richard A. Eisenberg. Kinds are calling conventions. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi: 10.1145/3408986. URL https://doi.org/10.1145/3408986.

[7] Richard Eisenberg and Simon Peyton Jones. Levity polymorphism. In *ACM Conference on Programming Language Design and Implementation (PLDI'17)*, pages 525–539. ACM, June 2017. URL https://www.microsoft.com/en-us/research/publication/levity-polymorphism/.

[8] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. *SIGPLAN Not.*, 37(5):282–293, May 2002. ISSN 0362-1340. doi: 10.1145/543552.512563. URL https://doi.org/10.1145/543552.512563.

[9] Christophe Gyurgyik, Alexander J Root, and Fredrik Kjolstad. Data layout polymorphism for bounding volume hierarchies, 2025. URL https://arxiv.org/abs/2511.15028.

[10] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 556–571, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. doi: 10.1145/3062341.3062354. URL https://doi.org/10.1145/3062341.3062354.

[11] C.A.R. Hoare. *Recursive data structures*. Prentice-Hall, Inc., USA, 1989. ISBN 0132840278.

[12] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*,

HOPL III, page 12–1–12–55, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937667. doi: 10.1145/1238844.1238856. URL https://doi.org/10.1145/1238844.1238856.

[13] Andrew Kennedy and Don Syme. Design and implementation of generics for the .net common language runtime. *SIGPLAN Not.*, 36(5):1–12, May 2001. ISSN 0362-1340. doi: 10.1145/381694.378797. URL https://doi.org/10.1145/381694.378797.

[14] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[15] Chris Leary and Tianqi Wang. XLA: Tensorflow, compiled. In *TensorFlow Dev Summit*, pages 2–3, 2017.

[16] Xavier Leroy. Efficient data representation in polymorphic languages. In *Proceedings of the 2nd International Workshop on Programming Language Implementation and Logic Programming*, PLILP '90, page 255–276, Berlin, Heidelberg, 1990. Springer-Verlag. ISBN 354053010X.

[17] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, page 177–188, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897914538. doi: 10.1145/143165.143205. URL https://doi.org/10.1145/143165.143205.

[18] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. Oxidizing ocaml with modal memory management. *Proc. ACM Program. Lang.*, 8(ICFP), August 2024. doi: 10.1145/3674642. URL https://doi.org/10.1145/3674642.

[19] Danielle Marshall, Michael Vollmer, and Dominic Orchard. Linearity and uniqueness: An entente cordiale. In *European Symposium on Programming*, pages 346–375. Springer International Publishing Cham, 2022.

[20] OCaml RFCs. Unboxed types in ocaml. https://github.com/ocaml/RFCs/blob/881b220adc1f358ab15f7743d5cd764222ab7d30/rfcs/unboxed-types.md, 2020. URL https://github.com/ocaml/RFCs/blob/881b220adc1f358ab15f7743d5cd764222ab7d30/rfcs/unboxed-types.md. Git commit 881b220adc1f358ab15f7743d5cd764222ab7d30; OCaml RFCs repository.

[21] OxCaml Developers. Ocaml with templates: Intro. https://oxcaml.org/documentation/templates/01-intro/, 2025. Accessed: 2026-01-14.

[22] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, page 636–666, Berlin, Heidelberg, 1991. Springer-Verlag. ISBN 0387543961.

[23] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, page 717–740, New York, NY, USA, 1972. Association for Computing Machinery. ISBN 9781450374927. doi: 10.1145/800194.805852. URL https://doi.org/10.1145/800194.805852.

[24] Alexander J Root, Christophe Gyurgyik, Purvi Goel, Kayvon Fatahalian, Jonathan Ragan-Kelley, Andrew Adams, and Fredrik Kjolstad. Compiling set queries into work-efficient tree traversals, 2025. URL https://arxiv.org/abs/2511.15000.

[25] Rupanshu Soi, Rohan Yadav, Fredrik Kjolstad, Alex Aiken, Maryam Mehri Dehnavi, Michael Garland, and Michael Bauer. Optimal software pipelining and warp specialization for tensor core gpus, 2025. URL https://arxiv.org/abs/2512.18134.

[26] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, 2022.

[27] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, 1989.

[28] Stephen Weeks. Whole-program compilation in mlton. In *Proceedings of the 2006 Workshop on ML*, ML '06, page 1, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934839. doi: 10.1145/1159876.1159877. URL https://doi.org/10.1145/1159876.1159877.

Appendix

A MLton

While playing with larger programs in MLton v20241230, we encountered several mystifying allocation decisions that affected performance. We distill these into the minimal cases presented below:

```
structure Main = struct
  val seed = case CommandLine.arguments () of
    []        => 42
    | s::_    => getOpt (Int.fromString s, 42)

  fun mkVal i = seed + i

  fun check name v =
      let
        (* heap bytes allocated *)
        val size = MLton.size v
        val loc  = if size > 0
            then "heap-allocated"
            else "stack-allocated"
      in
        print (name ^ " : " ^ loc ^ "\n")
      end

  val () = check "1" (mkVal 1)
  val () = check "2" (ref (mkVal 2))
  val () = check "3" (ref (ref (mkVal 3)))
  val () = check "4" (hd [ref (mkVal 4)])

  val L = Array.fromList [ref (mkVal 5)]
  val () = check "5" (Array.sub (L, 0))

  (* single-element tuple [1] *)
  type 'a one = {1 : 'a}
  val () = check "6" {1 = mkVal 6}

  (* multi-element tuple *)
  val () = check "7" (7, 77)
  val () = check "8" (mkVal 8, mkVal 88)
end

(*
(mkVal 1)               : stack-allocated
(ref (mkVal 2))         : heap-allocated
(ref (ref (mkVal 3)))   : stack-allocated
hd [ref (mkVal 4)]      : stack-allocated
Array.sub (L, 0)        : heap-allocated
{1 = mkVal 5}           : stack-allocated
(6, 66)                 : heap-allocated
(mkVal 7, mkVal 77)     : heap-allocated
*)
```

MLton provides a builtin `MLton.size` that reports, at runtime, the number of bytes a value occupies on the heap. A single `ref` is heap-allocated, yet a `ref` of `ref` resides on the stack. More vexing still is the tuple `(mkVal 8, mkVal 88)`: two runtime-computed values that MLton heap-allocates, though Morphic places the equivalent structure on the stack. The behavior of `(7, 77)` exhibits a curious non-locality: removing `(mkVal 8, mkVal 88)` causes this tuple to become stack-allocated. We contacted the MLton mailing list seeking clarification; a response is pending. Regardless of the underlying rationale, this illustrates a broader frustration for performance-conscious programmers: the illusion of control, where seemingly equivalent code yields dramatically different runtime behavior depending on opaque compiler heuristics.

B Morphic

To provide a twinkle of justification for better programmer control over data representation, we present empirical evidence that the *unbox everything* philosophy adopted by Morphic [3] is not universally optimal. Further, the choice between boxed and unboxed representations, or between array-of-structs and struct-of-arrays layouts, can yield order-of-magnitude performance differences depending on access patterns. Many of these optimizations, commonplace in production compilers [15], remain largely inaccessible to functional programmers without greatly obfuscating the algorithm.

To evaluate each representation, we implement a straightforward recursive evaluator over the syntax tree. For the inline and outline representations, the implementation follows the natural pattern-matching style endemic to functional programming. The flat representation, by contrast, abandons the elegance of structural recursion for the sake of cache locality. The implementation of the algorithm for both algebraic data types (inline and outline) and flattened arenas are found in Appendix C.

Data Representation

Inline (Unboxed). This is the Morphic default: all fields are stored contiguously within the parent node. Recursive children are necessarily behind pointers, but metadata such as source spans are inlined directly into each constructor.

```
// Unboxed
type Span {
  Span(
    Int, Int,
    Int, Int,
    Int, Int,
    Int, Int,
    Array Byte, Array Byte
  )
}

type Expr {
  Const(Int, Span),
  Var(Int, Span),
  Add(Expr, Expr, Span),
  Mul(Expr, Expr, Span),
  If(Expr, Expr, Expr, Span),
  Let(Expr, Expr, Span),
}
```

Outline (Boxed). Here we box the infrequently-accessed `Span` metadata. Since Morphic provides no mechanism for explicit boxing, we simulate it by wrapping `Span` in a length-one `Array Span`, which the compiler represents as two 64-bit integers and a pointer[3]. Any native boxing support would likely improve upon these numbers, making our results a conservative lower bound on the benefits of boxing.

```
type _Span {
  _Span(
    Int, Int,
    Int, Int,
    Int, Int,
    Int, Int,
    Array Byte, Array Byte,
  )
}

// "Boxed" with Array of size 1.
type Span { Span(Array _Span) }

type Expr {
  Const(Int, Span),
  Var(Int, Span),
  Add(Expr, Expr, Span),
  Mul(Expr, Expr, Span),
  If(Expr, Expr, Expr, Span),
  Let(Expr, Expr, Span),
}
```

Flat (Arena). We store all expression nodes in contiguous memory following a struct-of-arrays (SoA) layout. This representation segregates expressions from spans. Children of a node are stored immediately after their parent, eliminating the need for explicit child indices (all expressions have a statically known number of children). While semantically equivalent to the algebraic formulation, this representation demands intrusive change to the algorithm.

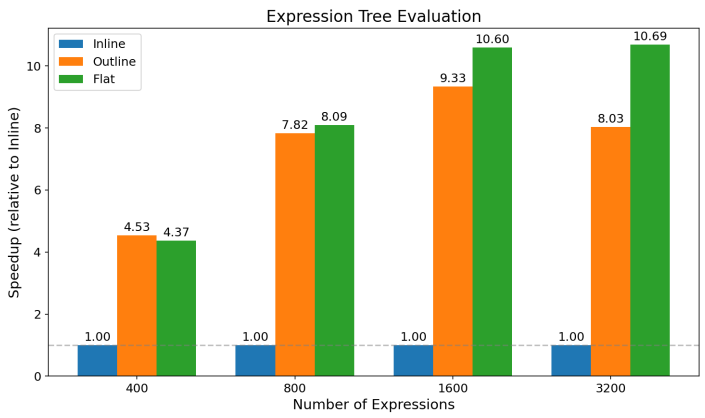
[3]We also experimented with encoding boxing via `Span` augmented with a self-referential field; this approach performed strictly worse.

```
tag_const: Int = 0
tag_var: Int = 1
tag_add: Int = 2
tag_mul: Int = 3
tag_if: Int = 4
tag_let: Int = 5

type Span {
  Span(
    Int, Int,
    Int, Int,
    Int, Int,
    Int, Int,
    Array Byte, Array Byte
  )
}

type ExprArena {
  ExprArena(Array Int, Array Span)
}

get_tag(arena: ExprArena, idx: Int): Int =
  let ExprArena(data, _) = arena in
  Array.get(data, idx)

get_field(
  arena: ExprArena, idx: Int, offset: Int
): Int =
  let ExprArena(data, _) = arena in
  Array.get(data, idx + offset)

get_span(
  arena: ExprArena, idx: Int, offset: Int
): Int =
  let ExprArena(_, spans) = arena in
  Array.get(spans, idx + offset)
```

Methodology & Results

We measure end-to-end performance on a MacBook Pro with an M2 Pro processor and 16GB of unified memory. We use Morphic `3cc45ab` and Clang `16.0.6`. Each benchmark begins with a cold run followed by five timed runs. We discard the minimum and maximum and report the mean of the remaining three.

Figure B presents the results (higher is better). The inline (unboxed) representation, i.e., Morphic's default, serves as the baseline. The outline (boxed) representation achieves speedups ranging from 4.53× at 400 expressions to 9.33× at



1600 expressions. The flat representation performs similarly, reaching 10.69× at 3200 expressions. These gains stem from improved cache locality: by boxing the cold `Span` metadata, the hot expression structure becomes more compact, reducing cache misses during traversal. Moreover, the flattened version uses less data per node (due to implicit indexes), further improving cache locality. Clearly, this benchmark favors contiguous representations; one could just as easily conjure an example where boxing performs better. This is precisely the point of our argument: no single data representation is optimal across the Cartesian product of algorithms, data characteristics, and machine architectures. Rather than committing to a fixed choice, languages should enable systematic exploration of this space.

C AST Evaluation Algorithm

```
eval(e: Expr, env: Array Int): Int =
  match e {
    Const(n, _) -> n,
    Var(i, _) ->
      Array.get(env, i),
    Add(a, b, _) ->
      eval(a, env) + eval(b, env),
    Mul(a, b, _) ->
      eval(a, env) * eval(b, env),
    If(c, t, f, _) ->
      if eval(c, env) = 0 {
        eval(f, env)
      } else {
        eval(t, env)
      },
    Let(binding, body, _) ->
      let v = eval(binding, env) in
      eval(body, Array.push(env, v)),
  }
```

```
eval(
  arena: ExprArena,
  idx: Int,
  env: Array Int
) : Int =
  let tag =
    get_tag(arena, idx)
  in
  if tag = tag_const {
    get_field(arena, idx, 1)
  } else if tag = tag_var {
    Array.get(
      env,
      get_field(arena, idx, 1)
    )
  } else if tag = tag_add {
    eval(
      arena,
      get_field(arena, idx, 1), env
    ) +
    eval(
      arena,
      get_field(arena, idx, 2), env
    )
  } else if tag = tag_mul {
    eval(
      arena,
      get_field(arena, idx, 1), env
    ) *
    eval(
      arena,
      get_field(arena, idx, 2), env
    )
  } else if tag = tag_if {
    let c =
      get_field(arena, idx, 1) in
    let t =
      get_field(arena, idx, 2) in
    let f =
      get_field(arena, idx, 3) in
    if eval(arena, c, env) = 0 {
      eval(arena, f, env)
    } else {
      eval(arena, t, env)
    }
  } else {
    let b =
      get_field(arena, idx, 1) in
    let body =
      get_field(arena, idx, 2) in
    eval(
      arena,
      body,
      Array.push(
        env, eval(arena, b, env)
      )
    )
  }
```