First-Order Laziness

ANTON LORENZEN, University of Edinburgh, United Kingdom
DAAN LEIJEN, Microsoft Research, USA
WOUTER SWIERSTRA, Utrecht University, Netherlands
SAM LINDLEY, University of Edinburgh, United Kingdom

In strict languages, laziness is typically modeled with explicit thunks that defer a computation until needed and memoize the result. Such thunks are implemented using a closure. Implementing *lazy data structures* using thunks thus has several disadvantages: closures cannot be printed or inspected during debugging; allocating closures requires additional memory, sometimes leading to poor performance; reasoning about the performance of such lazy data structures is notoriously subtle. These complications prevent wider adoption of lazy data structures, even in settings where they should shine. In this paper, we introduce *lazy constructors* as a simple first-order alternative to lazy thunks. Lazy constructors enable the thunks of a lazy data structure to be defunctionalized, yielding implementations of lazy data structures that are not only significantly faster but can easily be inspected for debugging.

CCS Concepts: • **Software and its engineering** → **Control structures**; **Recursion**; • **Theory of computation** → **Operational semantics**.

Additional Key Words and Phrases: Laziness, Defunctionalization, Perceus

ACM Reference Format:
Anton Lorenzen, Daan Leijen, Wouter Swierstra, and Sam Lindley. 2025. First-Order Laziness. *Proc. ACM Program. Lang.* 9, ICFP, Article 261 (August 2025), 29 pages. https://doi.org/10.1145/3747530

1 Introduction

Purely functional data structures have several important advantages. Data structures implemented in a purely functional language are persistent, thread safe, and may be verified using elementary methods. *Efficient* purely functional data structures, however, often require laziness to avoid recomputation, even when implemented in a strict language [Okasaki 1998]. In a strict language, like OCaml and Racket, computations may be deferred by creating an explicit thunk. Despite the apparent simplicity of implementing laziness in this fashion, using higher-order functions has its drawbacks: thunked computations cannot be printed or inspected; allocating closures requires additional memory; reasoning about the performance of such arbitrary closures is a subtle affair.

This paper explores how to add a dash of laziness to a strict language, where computations are deferred explicitly using a *first-order data constructor*, defunctionalizing the higher-order closures programmers would otherwise write by hand. As we will show, these techniques suffice to implement purely functional data structures efficiently, reducing the time and space used by traditional implementation techniques. To sketch the main idea, consider the following definition of a stream in Koka [Leijen 2014]:

Authors' Contact Information: Anton Lorenzen, University of Edinburgh, Edinburgh, United Kingdom, anton.lorenzen@ed.ac.uk; Daan Leijen, Microsoft Research, Seattle, USA, daan@microsoft.com; Wouter Swierstra, Utrecht University, Utrecht, Netherlands, w.s.swierstra@uu.nl; Sam Lindley, University of Edinburgh, Edinburgh, United Kingdom, sam.lindley@ed.ac.uk.

ACM 2475-1421/2025/8-ART261
https://doi.org/10.1145/3747530

```
type stream<a>
  SCons(head : a, tail : stream<a>)
  SNil
  lazy SAppend(s1 : stream<a>, s2 : stream<a>) ->
    match s1
      SCons(x,xx) -> SCons(x,SAppend(xx,s2))
      SNil        -> s2
```

Besides the familiar data constructors, `SNil` and `SCons`, there is a third *lazy constructor* `SAppend`. Whenever we append two streams using this constructor, the operation takes constant time.

However, in contrast to the regular constructors, we never match on a lazy constructor. Instead, whenever the run-time encounters an `SAppend` constructor, the associated right-hand side of the data declaration is executed, producing a single `SCons` cell if the first stream is non-empty. Written in this style, the append of the two streams happens *on-demand*, only traversing as much of the first stream as is necessary. To illustrate this point, we define the `take` function on streams:[1]

```
fun stream/take(xs : stream<a>, n : int) : list<b>
  if n <= 0 then Nil
  else match xs
    SCons(x,xx) -> Cons(x, stream/take(xx,n - 1))
    SNil        -> Nil
```

As this definition shows, there is no need to write a case for the `SAppend` constructor. If there are any lazy constructors in the argument stream, these are forced on-demand as the `take` function traverses its input. This is best illustrated with an example:

```
val xs : stream<int> = SCons(0,SAppend(SCons(1,SNil),SCons(2,SNil)))
```

If we call `take(xs,1)` this produces a singleton list with the number `0`, leaving the tail of the stream unchanged. If we call `take(xs,2)`, however, this evaluates the lazy `SAppend` constructor – but only enough to discover that the second element of the resulting list should be `1`. Taking three or more elements forces the entire stream. This process happens entirely under the hood and the program cannot observe that thunks have been evaluated. Laziness preserves referential transparency: a lazy thunk is indistinguishable from the value it computes.

However, for debugging or educational purposes, it would be nice to be able to peek under the hood [Gill 2000]. With lazy constructors, this is possible: the unsafe primitive `debug-show` displays the lazy constructors without forcing any further evaluation. The informal description of the behaviour of `take` is visible in the command line:

```
> take(xs,1); debug-show(xs)
SCons(0,SAppend(SCons(1,SNil),SCons(2,SNil)))
> take(xs,2); debug-show(xs)
SCons(0,SCons(1,SAppend(SNil,SCons(2,SNil))))
> take(xs,3); debug-show(xs)
SCons(0,SCons(1,SCons(2,SNil)))
```

Lazy constructors are limiting: unlike unrestricted laziness as in Haskell or the explicit thunks in strict languages, our example stream only supports a single lazy operation. If we need other lazy operations, we need to add further lazy constructors to the stream data type. As we shall see, however, most implementations of lazy data structures (e.g. as given by Okasaki [1998]) rely only on a handful of lazy operations. By making the laziness first-order, we gain the ability to inspect and optimize thunked computations in new and interesting ways.

For example, the compiler can now statically determine the runtime size of each lazy constructor: the memory location associated with each forced `SAppend` cell can for example always be reused *in-place* for the resulting `SCons` cell, instead of overwriting it with an indirection node as in most implementations of laziness. Moreover, with Perceus reference counting [Lorenzen and Leijen 2022;

[1]In Koka, we can *locally qualify* an identifier, as in `stream/take`. A bare `take` is usually resolved to the right definition based on the type context, but we can always use the fully qualified name as well to distinguish it for example from `list/take`.

Reinking, Xie et al. 2021], if the matched `SCons` of `s1` happens to be unique at runtime, the next `SAppend` can reuse that memory in-place as well.

Just as first-order data types are easier to manipulate and implement efficiently than their Church encoding, the first-order approach to laziness pioneered in this paper is both efficient and effective. This paper demonstrates the applicability of lazy constructors, nails down their semantics, and benchmarks the performance of our implementation in Koka. More specifically, this paper makes the following contributions:

- We illustrate the use of first-order laziness through a series of examples drawn from Okasaki's book on functional on functional data structures [Okasaki 1998], such as the Bankers Queue and Realtime Queue (Section 2). Our implementation using lazy constructors arises naturally from defunctionalizing the thunked closures used in Okasaki's original implementation (Section 3).
- We formalize the behaviour of lazy constructors as a modest extension of Launchbury's natural semantics for lazy evaluation [Launchbury 1993] and prove that this extension preserves type soundness and referential transparency (Section 4).
- We present a small step semantics, which forms the basis of the implementation in Koka. The first-order nature of lazy constructors enables new compiler optimizations that are not possible in general: avoiding indirection nodes entirely; re-using memory; and running in constant stack space. We justify these compiler optimizations using equational reasoning (Section 5).
- We implement lazy constructors in Koka and benchmark all lazy queues and heaps given by Okasaki [1998]. Our benchmarks show that lazy data structures, implemented using lazy constructors, are always faster than the same data structures implemented using traditional thunks, and can come close to their strict implementations even in sequential settings where laziness provides no benefit (Section 6).

The associated technical report [Lorenzen et al. 2025] contains proofs, listings of Okasaki's data structures and further examples of lazy constructors.

2 Programming with First-Order Laziness

To illustrate the importance of laziness, even in a strict language, we revisit the Bankers Queue example by Okasaki [1998]. It is a typical example of a functional data structure that uses laziness to obtain better amortized time complexity bounds in a persistent setting.

2.1 A Strict Bankers Queue Using Lists

To warm up, we first define a *strict* Bankers Queue; the next section will give an alternative lazy implementation using streams. A Bankers Queue consists of a pair of lists, where new elements are appended to the rear list `ys`, and elements are removed from the front list `xs`:

```
struct queue<a>   // queue with elements ‘xs ++ reverse(ys)‘
  xs : list<a>    // front list
  n  : int        // length of the front
  ys : list<a>    // rear list (to be reversed)
  m  : int        // length of the rear
```

The queue maintains the invariants that `length(xs)==n`, `length(ys)==m`, and `n>=m`. As the rear list grows and the front list shrinks, the queue becomes unbalanced. To ensure the desired invariant is maintained, Okasaki defines a `balance` function that sometimes moves the rear list to the front list:

```
fun balance( Queue(xs,n,ys,m) : queue<a> ) : queue<a>
  if n >= m
    then Queue(xs,n,ys,m)
    else Queue(xs ++ reverse(ys), n + m, Nil, 0)
```

The the enqueue and dequeue operations ensure the result queues are always balanced:

```
fun snoc( Queue(xs,n,ys,m) : queue<a>, y : a ) : queue<a>
  balance(Queue(xs, n, Cons(y,ys), m + 1))
fun uncons( Queue(xs,n,ys,m) : queue<a> ) : maybe<(a,queue<a>)>
  match xs
    Cons(x,xx) -> Just((x, balance(Queue(xx, n - 1, ys, m))))
    Nil        -> Nothing
```

However, as noted by Okasaki, this implementation is not always very efficient. A rebalancing step may take time linear in the length of the queue (`xs ++ reverse ys`). In a *persistent* setting, there may be many shared references to a single queue; unless we ensure the rebalancing computation is also shared, each reference may need to redo the rebalancing work. To illustrate this point, consider the following code snippet:

```
val q = Queue(xs,n,xs,n)
for(1,n, fn(i) snoc(q,i))
```

In this example, we construct an 'almost unbalanced' queue `q`. Each `snoc` operation in the loop requires the queue to be rebalanced, where each rebalancing requires $2n$ steps. Consequently, the entire loop takes quadratic time. If, however, the rebalancing work is *shared* between the different calls to `snoc`, then the loop runs in linear time. Even in this strict and persistent setting, there is a clear need for some memoization in order to avoid such recomputation.

2.2 A Lazy Bankers Queue Using Streams

To obtain the optimal amortized time complexity of the Bankers Queue in a persistent setting, we need to ensure that the result of the rebalancing is shared between all copies of the queue. Rather than using lists, we use a variation of the streams from the introduction instead:

```
struct queue<a>
  xs : stream<a>  // the front stream
  n  : int        // length of the front
  ys : stream<a>  // rear stream
  m  : int        // length of the rear
```

However, unlike our streams from the introduction, we not only need an append operation, but also a reverse operation:

```
type stream<a>
  SNil
  SCons(head : a, tail : stream<a>)
  lazy SAppend(s1 : stream<a>, s2 : stream<a>) ->
    match s1
      SCons(x,xx) -> SCons(x,SAppend(xx,s2))
      SNil        -> s2
  lazy SReverse(s : stream<a>, acc : stream<a>) ->  // accumulating reverse
    match s
      SCons(x,xx) -> SReverse( xx, SCons(x,acc) )
      SNil        -> acc
```

The rebalancing function now uses the *lazy constructors* to defer and share rebalancing:

```
fun balance( Queue(xs,n,ys,m) ) : queue<a>
  if n >= m
    then Queue(xs,n,ys,m)
    else Queue(SAppend(xs,SReverse(ys,SNil)), n + m, SNil, 0)
```

Since the *only* operations we need to rebalance the queue are append and reverse, we only need two lazy constructors – `SAppend` and `SReverse`. Moreover, the definitions of `snoc` and `uncons` remain unchanged as we never match on lazy constructors. The `balance` function introduces lazy constructors, but defers the associated work. Consider the loop we saw previously:

```
val q = Queue(xs,n,xs,n)
for(1,1000, fn(i) snoc(q,i))
```

Each call to snoc simply creates a delayed computation for the rebalancing in constant time (as `SAppend(xs,SReverse(ys,SNil))`), which only takes constant time. In contrast, the uncons operation pattern matches on the front stream, which may trigger evaluation of the lazy constructors and can thus take linear time. Still, Okasaki shows that this implementation of the bankers queue has constant amortized time complexity.

2.3 Lazy Match

Unlike traditional implementations of laziness, lazy constructors remain first order. Consequently, they can be printed for the sake of debugging:

```
> val xs = SCons(1,SCons(0,SNil))
> val q0 = Queue(xs,2,xs,2)
> val q  = snoc(q,2)
> debug-show(q)
Queue( SAppend(SCons(1,SCons(0,SNil)),SReverse(SCons(2,SCons(1,SCons(0,SNil)))) ), 5, SNil, 0)
```

Of course, since q is persistent, we can uncons an element and still observe the original queue:

```
> val _ = uncons(q)
> debug-show(q)
Queue( SCons(1, SAppend(SCons(0,SNil),SReverse(SCons(2,SCons(1,SCons(0,SNil)))) ), 5, SNil, 0)
```

The reader may be startled at this point: clearly the queue q has changed! Doesn't this break referential transparency? The answer is no: though the stream has indeed changed, this cannot be observed since any attempt to match on q never yields a lazy constructor. In fact, this is exactly why any thunk can be overwritten with its value without breaking referential transparency.

However, it can be useful for debugging to peek under the hood during evaluation of a lazy data structure: this is what the debug-show function does. This function is implemented using an additional unsafe primitive, `lazy match`, that can observe lazy constructors without forcing evaluation.

The lazy match construct is used extensively to implement first-order laziness. In particular, the Koka compiler inserts an additional eval call whenever a programmer matches on a data type with lazy constructors. The compiler generated eval function evaluates the argument to weak head normal form. The code corresponding to the uncons function becomes:

```
fun uncons( Queue(xs,n,ys,m) )
  match stream/eval(xs)      // compiler inserts an ‘eval‘ automatically
    SCons(x,xx) -> Just((x, balance(Queue(xx, n - 1, ys, m))))
    SNil        -> Nothing
```

The eval function uses the lazy match primitive, inserting the code associated with each lazy constructor in the corresponding branch, roughly like:

```
// compiler generated
fun stream/eval(s : stream<a>)
  lazy match s
    SAppend(s1,s2)  -> match s1
      SCons(x,xx) -> lazy-update(s, SCons(x,SAppend(xx,s2)))
      SNil        -> lazy-update(s, eval(s2))
    SReverse(s1,acc) -> match s1
      SCons(x,xx) -> lazy-update(s, eval(SReverse(xx, SCons(x,acc))))
      SNil        -> lazy-update(s, eval(acc))
    _ -> s
```

where the lazy-update primitive updates the root node with the result. In practice, we generate a more efficient version where we do not use stack space unnecessarily. While this stream/eval function uses the stack in the last three branches (where eval is not a tail-call), we will derive a more efficient version in Section 5.

2.4 The Bankers Queue with Logarithmic Worst-Case Time Complexity

While our Bankers Queue has constant *amortized* time complexity, its worst-case time complexity is still linear in the size of the queue. The reversal is monolithic: once the `SAppend` is fully evaluated, we need to completely reverse the second list to find the last element. How can we ensure that these queues have better worst-case complexity?

Okasaki [1998] reimplements the Bankers Queue using *rotations* that combine appending with reversal. However, our first-order lazy constructors support a simpler solution – albeit one that requires further language support. The main idea is to evaluate at most one lazy constructor instead of recursively evaluating up to weak head normal form. If we do this for the reversed tail each time we evaluate an `SAppend` constructor, then we can ensure that by the time all `SAppend`'s are done, the tail is fully reversed. To evaluate only one lazy constructor, the compiler generates a second evaluation function, called `eval-one`:

```
// compiler generated
fun stream/eval-one(s : stream<a>) : stream<a>
  lazy match s
    ...
    SReverse(s1,acc) -> match s1
      SCons(x,xx) -> lazy-update(s, SReverse(xx, SCons(x,acc)))
      SNil        -> lazy-update(s, acc)
    _ -> s
```

As we can see, it is almost equivalent to the `eval` function. The key difference is that `eval-one` is no longer recursive. For example, if the first list is `Nil`, the `eval-one` function returns the accumulated list – whether it is in weak head normal form or not.

Using this primitive, we can reduce the worst-case time complexity from linear to logarithmic, by only adding a single line to the code associated with the `SAppend` constructor:

```
type stream<a>
  ...
  lazy SAppend( s1 : stream<a>, s2 : stream<a> ) ->
    stream/eval-one(s2)  // for each SAppend, evaluate also one SReverse
    match s1
      SCons(x,xx) -> SCons(x, SAppend(xx,s2))
      SNil        -> s2
```

The key here is that `s2` always contains the `SReverse` constructor. In this fashion, we evaluate one step of the reversal every time we invoke the `SAppend` constructor. As `s2` is only one element longer than `s1` when rebalance, by the time the `SAppend` hits the `SNil` case the `SReverse` is almost done – we pay a little more during each `SAppend` step, but gain more predictable performance overall.

2.5 Avoiding Stack Overflows from Recursive Evaluation

The code associated with the `SAppend` constructor seems entirely innocent. On closer inspection, however, when there are nested `SAppend` constructors, this may trigger recursive evaluation:

```
type stream<a>
  ...
  lazy SAppend( s1 : stream<a>, s2 : stream<a> ) ->
    match s1                                    // may evaluate an SAppend recursively!
      SCons(x,xx) -> SCons(x, SAppend(xx,s2))
      SNil        -> s2
```

As this call is not tail-recursive, it requires stack space. If the `s1` stream is a long sequence of unevaluated `SAppend` constructors, this may ultimately lead to a stack overflow.

This is not a purely theoretical concern. When working on our benchmarks, we discovered that the Physicists Queue, Implicit Queue, and Binomial Heaps as presented by Okasaki [1998] require stack space linear in the size of the queue. This can be a problem in practice: when two

$$\begin{tikzcd}
SAppend(xs_1, ys_1) \arrow[d, "xs_1"] & SAppend(Root, ys_1) & SAppend(Root, ys_1) & SCons(h, t_1) \arrow[d, "t_1"] \\
SAppend(xs_2, ys_2) \arrow[d, "xs_2"] & SAppend(up_1, ys_2) \arrow[u, "up_1"'] & SAppend(up_1, ys_2) \arrow[u, "up_1"'] & SAppend(t_2, ys_1) \arrow[d, "t_2"] \\
SAppend(xs_3, ys_3) \arrow[d, "xs_3"] & SAppend(xs_3, ys_3) \arrow[u, "up_2"'] \arrow[d, "xs_3"] & SCons(h, t_3) \arrow[u, "up_2"'] \arrow[d, "t_3"] & SAppend(t_3, ys_2) \arrow[d, "t_3"] \\
SCons(h, t) & SCons(h, t) & SAppend(t, ys_3) & SAppend(t, ys_3)
\end{tikzcd}$$

Fig. 1. In a Schorr-Waite traversal, we first descend and reverse the pointers until we find a normal constructor. Then we evaluate the lazy constructors from the bottom-up. From left to right: (1) nested lazy constructors: to force the topmost constructor we first have to force the ones below it, (2) the runtime has descended to the bottom-most constructor and reversed the pointers, (3) the bottom-most constructor has been evaluated and we follow the 'up' pointer, (4) the final result once evaluation is complete.

million elements were subsequently inserted into the Physicists Queue, the stack overflowed in both Koka 3.1.3 and OCaml 4.14.2 [2]. We are not aware of any technique that avoids this problem with traditional lazy thunks.

However, as we show in detail in Section 5.5, this problem *can* be resolved with lazy constructors. As we compile a specialized `eval` function for each data type, we can specialize `eval` to evaluate nested thunks tail-recursively using an in-place Schorr-Waite traversal [Leijen and Lorenzen 2023 2025; Lorenzen et al. 2023; Schorr and Waite 1967], illustrated in Figure 1. If a programmers wants to ensure that an argument of a lazy constructor is fully evaluated before the constructor is evaluated, they can put an exclamation mark, e.g. `!xs`, before the name of the argument. This indicates that Koka's runtime should force the argument before it begins to evaluate the lazy constructor[3]:

```
    lazy SAppend( !xs : stream<a>, ys : stream<a> ) -> ...
```

With the `!` annotation, before the `SAppend` constructor is evaluated, the runtime first evaluates the `xs` stream. This does not require any extra stack- or heap space, since `eval` can keep track of all lazy constructors under evaluation using an in-place *zipper* [Huet 1997; Lorenzen et al. 2024] stored in the lazy constructors themselves. In this fashion, we avoid allocating arbitrary stack space, even if `SAppend` constructors are nested.

2.6 The Realtime Queue with Constant Time Complexity

The Bankers Queue operations are amortized constant-time, but still have logarithmic worst-case time complexity. When the queue rebalances, the front stream may start with another `SAppend` constructor. If this is the case, rebalancing creates nested `SAppend` constructors. Fortunately, the number of nested constructors is bounded: since we only rebalance when the front stream is shorter than the rear stream, we know that the number of nested `SAppend` thunks is logarithmic in the total length of the queue. A call to `uncons` may have to perform one step for each of the `SAppend` constructors in the front stream, leading to a logarithmic worst-case time complexity.

To ensure our operations have constant worst-case time complexity, we need the front stream to evaluate one step *every time* we perform an operation on the queue. Okasaki [1998] (Section 7.2) proposes to do this by adding a 'schedule parameter'. This schedule parameter is a suffix of the

[2] As measured on an Apple M1 which has a hard limit for stack size of 65mb.

[3] At the moment this is not yet implemented in Koka but we hope to have it working soon according to the rules in Section 5.5.

front stream; the schedule starts with the *only* `SAppend` constructor in the entire stream. Maintaining this invariant is the key ingredient to obtain constant worst-case time complexity.

```
struct queue<a>
  front : stream<a>
  rear  : list<a>
  sched : stream<a>
```

There are several other modifications to the queue type. Note that the rear queue is now a list rather than a stream; rather than store the lengths of the front and rear stream, the schedule determines when the queue must be rebalanced:

```
fun balance( Queue(front,rear,sched) : queue<a> ) : queue<a>
  match sched
    SCons(_,s) -> Queue(front,rear,s)
    SNil       -> val f = SAppend(front,SReverse(rear,SNil)) in Queue(f,Nil,f)
```

In each rebalancing operation, the schedule is evaluated to weak head normal form. If the schedule is non-empty, its tail becomes the new schedule. Rebalancing happens when the schedule is empty – and hence the `front` stream is fully evaluated. In that case, the new schedule is initialized to `f`, shared with the new front of the queue. As the front stream is fully evaluated, `f` will never contain nested `SAppend` constructors. This ensures the desired constant worst-case time complexity.

The only thing that remains to be done, is implement the `snoc` and `uncons` operations:

```
fun snoc( Queue(front,rear,sched) : queue<a>, x : a) : div queue<a>
  balance( Queue(front, Cons(x,rear), sched) )

fun uncons( Queue(front,rear,sched) : queue<a> ) : div maybe<(a,queue<a>)>
  match front
    SCons(x,xx) -> Just((x, balance(Queue(xx,rear,sched))))
    SNil        -> Nothing
```

As both operation call `balance`, they are guaranteed to advance the schedule, as required. In the previous examples we mostly relied on laziness to defer computation, but in the Realtime Queue we also rely on the computation being *shared* – in this case between the front stream and the schedule.

3 Illuminating First-Order Laziness

To develop a deeper intuition for lazy constructors, we begin by showing how these arise naturally as the defunctionalized version of explicit thunks. To do so, we recreate the stream definition used in the previous section, starting from the more familiar implementation from the literature. We begin by defining the standard stream interface, without using any lazy constructors, but instead deferring computations with explicit thunks:

```
alias stream<a> = thunk<streamcell<a>>
type streamcell<a>
  SCons( head : a, tail : stream<a> )
  SNil
```

A stream is a list where the list cells are separated by lazy thunks (using the style of Wadler et al. [1998] and Okasaki [1998, section 4.2]). This makes it possible to perform operations like appending streams in constant time initially, where the linear-time append is only evaluated once that part of the list is reached. Such delayed computations are encapsulated in thunks, which take a closure that is only evaluated when needed. Thunks have the typical interface:

```
type thunk<a>
fun delay( f : () -> a )  : thunk<a>
fun force( t : thunk<a> ) : div a
```

The `delay` function creates a thunk from a computation and `force` evaluates the thunk. In Koka, we use the `div` effect to indicate that forcing a thunk may diverge. Under the hood, the implementation ensures that the computation is run at most once: after the first call to `force`, its result is memoized and returned in constant time upon every subsequent call. We can use this API to implement the familiar `append` and `reverse` functions on streams:

```
fun sappend( s1 : stream<a>, s2 : stream<a> ) : div streamcell<a>
  match s1.force
    SCons(x,xx) -> SCons(x, append(xx,s2))
    SNil        -> s2.force
fun append( s1 : stream<a>, s2 : stream<a> ) : div stream<a>
  delay{ sappend(s1,s2) }
fun sreverse( s1 : stream<a>, s2 : stream<a> ) : div streamcell<a>
  match s1.force
    SCons(x,xx) -> sreverse(xx, delay{ SCons(x,s2) })
    SNil        -> s2.force
fun reverse( s1 : stream<a> ) : div stream<a>
  delay{ sreverse(s1, delay{ SNil }) }
```

3.1 Lazy Constructors for Thunks

As the first step towards recreating the streams used in the previous section, we begin by implementing the `thunk<a>` interface using lazy constructors:

```
type thunk<a>
  Memo( v : a )
  lazy Lazy( f : () -> a ) ->
    Memo( f() )
```

This type has two constructors. The `Memo` constructor stores a value of type `a`. The `Lazy` constructor is a *lazy constructor* that can be used to construct a values of type `thunk<a>`. Remember, the lazy constructor is never observable in a match statement; any match on a value of type `thunk<a>` will force evaluation of the corresponding expression, `Memo( f() )`. Using this definition, we can easily simulate the typical thunk interface from strict languages:

```
fun delay( f : () -> a ) : thunk<a>
  Lazy(f)

fun force( t : thunk<a> ) : a
  match t
    Memo(v) -> v
```

The power of lazy constructors is that they allow us to *specialize* thunks to the computations they contain. By analyzing the possible closures `f` that may be stored in `Lazy`, we can specialize the higher-order `thunk<a>` type for our program. We define the first-order `stream<a>` type as:

```
type stream<a> =
  Memo( v : streamcell<a> )
  lazy SAppend( xs : stream<a>, ys : stream<a> ) ->
    Memo( sappend(xs, ys) )
  lazy SReverse( xs : stream<a>, acc : stream<a> ) ->
    Memo( sreverse(xs, acc) )
```

We keep the `Memo` constructor from the definition of `thunk<a>` but specialize `Lazy(f)` to the two computations that are used. This type now has two separate lazy constructors for these two computations, but we can still define `force` as before. We do need to update the corresponding code to append and reverse streams. The definition for `sappend`, for example, now reads:

```
fun sappend( s1 : stream<a>, s2 : stream<a> ) : div streamcell<a>
  match s1.force
    SCons(x,xx) -> SCons(x,SAppend(xx,s2))
    SNil        -> s2.force
```

Here we still need to force each stream to a `streamcell`. Where the previous definition deferred the recursive call to `sappend`, we now simply use the lazy `SAppend` constructor to the same effect.

3.2 The Cost of Laziness

Based on the last section, one might think that lazy constructors are just the defunctionalization of explicit thunks. But, in fact, they are a bit more powerful than that and allow us to fix a performance problem that arises when using explicit thunks. Let us consider a fully evaluated stream, such as:

```
> val nums = Memo(SCons(1, Memo(SCons(2, Memo(SCons(3, Memo(SNil)))))))
```

There is an indirection node between every pair of adjacent elements in the stream! This is a consequence of the encoding of streams, where the elements of `streamcell<a>` and those of `stream<a>` alternate. This means that traversing even a fully-evaluated stream will require twice as many pointer lookups as traversing a list: all `SCons` and `Memo` nodes live in different cells linked by pointers. In practice, languages like OCaml can mitigate this problem since they distinguish indirection nodes from all other values, which makes it possible to omit indirection nodes when the stream is fully evaluated on creation.

However, if a stream is the result of a lazy computation, the indirection nodes are unavoidable. For example, if we append a stream to `nums`, the `SAppend` constructor has to be rewritten as a `Memo` constructor to ensure that all references to `app` can share the memoized result:

```
> val nums' = SAppend(nums, Memo(SNil))
> debug-show(force(nums'))
Memo(SCons(1, SAppend(Memo(SCons(2, Memo(SCons(3, Memo(SNil))))), Memo(SNil))))
```

This implies that once we fully evaluate the append, all `Memo` nodes in the stream are necessary:

```
> debug-show(forceall(nums'))
Memo(SCons(1, Memo(SCons(2, Memo(SCons(3, Memo(SNil)))))))
```

These indirections are typical for the traditional approach to laziness, but they may introduce a significant performance cost compared to a strict program. Appending to a list of length n involves only n allocations, but appending to a stream of length n requires $2n$ allocations to also create all the indirection nodes in between. In fact, without defunctionalization, this is usually even more expensive since each thunk involves another allocation for a closure and so $3n$ allocations can be necessary.

Furthermore, the indirection nodes stay in the `stream` even once the thunks are fully evaluated, where they introduce additional pointer lookups. Garbage collected languages like OCaml may remove this indirection during GC runs, but this optimization is not available in reference counted languages such as Koka [Leijen 2014] or Lean [Moura and Ullrich 2021].

This is one of the reasons why lazy data structures are often less efficient than their strict counterparts. As we show in our benchmarks, the classic lazy data structures of Okasaki [1998] are a factor of 2-3x less efficient than their strict counterparts when implemented using explicit thunks.

3.3 Fusing Streams and Stream Cells

In contrast to traditional approaches to laziness, first-order lazy constructors allow us to remove most of these indirections. To obtain a better version of `stream<a>` with fewer indirections, we inline the definition of `streamcell<a>` in the type of streams:

```
type stream<a>
  SNil
  SCons( x : a, xx : stream<a> )
  lazy SAppend( xs : stream<a>, ys : stream<a> ) ->
    sappend(xs, ys)
  lazy SReverse( xs : stream<a>, acc : stream<a> ) ->
    sreverse(ys, acc)
```

Compared to the previous `stream<a>` declaration, we have replaced the `Memo` constructor with the `SNil` and `SCons` constructors. Furthermore, we do not return `Memo` from `SAppend` and `SReverse` and instead return the remaining stream immediately. This makes the structure of the type quite different: where the previous definition would alternate `SCons` cells with thunks, we can now mix normal and lazy constructors arbitrarily. For example, the fully evaluated stream `SCons(1, SCons(2, SNil))` is a valid inhabitant of this stream type, as is the stream containing lazy constructors `SAppend(SAppend(SNil, SNil), SNil)`.

To complete this definition, however, we need to update our `sappend` and `sreverse` functions. These no longer need to match on `Memo` constructors, but rather manipulate the streams directly. To illustrate this point, we redefine both `sreverse` and `sappend`:

```
fun sreverse( s1 : stream<a>, s2 : stream<a> ) : div stream<a>
  match s1
    SCons(x,xx) -> sreverse( xx, SCons(x,s2) )
    SNil        -> s2

fun sappend( s1 : stream<a>, s2 : stream<a> ) : div stream<a>
  match s1
    SCons(x,xx) -> SCons(x, SAppend(xx,s2) )
    SNil        -> s2
```

Note that, unlike our previous definition, we now build the accumulator of `sreverse` as a sequence of `SCons` cells with no more indirection nodes `Memo` in between.

But where did the indirections go? We still need to memoize the result of evaluating an `SAppend` or `SReverse`. To ensure the results of evaluating these lazy constructors are still shared, Koka inserts an implicit indirection into the lazy constructor above:

```
  lazy SAppend( xs : stream<a>, ys : stream<a> ) ->
    Indirect(sappend(xs, ys))
```

When matching on a stream `s` it can now happen that `s` is an indirection node, pointing to some `s'`. In that case the runtime follows the indirection and keeps matching on `s'`. These indirection nodes, however, are only created when a lazy constructor is evaluated. For example, the accumulator built in the `sreverse` function is completely free from indirections. This reduces the memory overhead that typically arises from sharing lazy computations.

3.4 In-Place Reuse of Lazy Constructors

As an additional optimization Koka avoids allocating an indirection node when it sees a constructor. This is exactly what made our earlier `thunk<a>` type work:

```
type thunk<a>
  Memo( v : a )
  lazy Lazy( f : () -> a ) ->
    Memo( f() )
```

Here, no implicit indirection node is created: instead the memory underlying the `Lazy` constructor is rewritten to contain a `Memo` constructor during evaluation.

By inlining the definitions of `sappend` and `sreverse` into the definition of the `stream` data type, we avoid indirection nodes altogether. The type of the `SAppend` constructor then becomes:

```
  lazy SAppend( xs : stream<a>, ys : stream<a> ) ->
    match xs
      SCons(x,xx) -> SCons(x, SAppend(xx,ys))
      SNil        -> ys
```

This definition makes explicit that the `SAppend` constructor evaluates to an `SCons` constructor if the first branch is taken. Koka can detect this fact and will not create an indirection node in that case: instead the memory cell holding the `SAppend` constructor is overwritten to contain the `SCons` constructor. This is similar to how the original Spineless Tagless G-machine can sometimes perform in-place updates of closures instead of creating indirections [Peyton Jones 1992].

In those branches where the tail position is not a constructor of an appropriate size, we still generate an indirection node. In the `SNil` case above, we reuse the space of the `SAppend` constructor for an indirection to `ys`. To get rid of them, we can inline the `force` and match on the second stream:

```
lazy SAppend( xs : stream<a>, ys : stream<a> ) ->
  ...
    SNil -> match ys
      SCons(y,yy) -> SCons(y,yy)
      SNil        -> SNil        // an indirection is still necessary here
```

This rewrites SAppend into SCons in the first branch and only requires an indirection node in the last case. However, if the old SCons in ys still has a reference it will stay around, which can increase space usage [Peyton Jones 1992].

3.5 In-Place Reuse with Reference Counting

It is important to note that our in-place reuse of *lazy constructors* is quite different to in-place reuse using reference counting [Reinking, Xie et al. 2021; Schulte and Grieskamp 1992; Ullrich and de Moura 2019]. In that setting, memory cells are reused in-place when their reference count is one. In contrast, the memoization of lazy constructors does not require reference count at all. In fact, memoization is only useful if the memory cell is shared among several references!

Nonetheless, the Koka compiler combines both techniques. Internally, the Koka compiler rewrites the SAppend constructor to the following code snippet:

```
lazy SAppend( xs : stream<a>, ys : stream<a> ) as _root ->
  match xs
    SCons( x, xx ) as cell ->
      reuse-always(_root, SCons( x, reuse-if-unique(cell, SAppend(xx,ys)) ))
    SNil -> reuse-always(_root, Indirect(ys))
```

That is, the _root memory cell holding the SAppend constructor is overwritten with the SCons cell in the first branch and the Indirect node in the second branch. This is independent of the reference count of the stream cell. Conversely, if the reference count of the cell of the front stream happens to be one (and only then), its memory location is reused for the new SAppend constructor.

When a programming languages combines lazy constructors with reference counting, this allows programmers to write code that runs with no fresh allocations at all. This is a key advantage of the first-order approach to laziness. Koka's memory re-use based on reference counts is limited to first-order data constructors: it cannot re-use memory locations associated with closures or traditional thunks. Our approach paves the way for adding laziness to the fully in-place calculus [Lorenzen et al. 2023], which promises to enable the first fully in-place lazy data structures.

4 Formalization

In this section we formalize a high-level view on lazy constructors that abstracts from implementation concerns such as in-place updates. First, we consider a model of lazy constructors based on Section 3.1, where forcing does not have to recurse. Our type $A \nrightarrow_F B$ states that a lazy constructor carrying A can be forced using the function F to yield a normal constructor of type B and will be memoized. This model of lazy constructors is quite similar to traditional laziness and we can adapt Launchbury [1993]'s semantic to reason about it.

Similar to how normal data types can be encoded as a sum-of-products, we propose to model lazy data types as a *thunked*-sum-of-products. For example, a lazy data type such as:

```
type example
  A(a1 : A1, ..., ai : Ai)
  B(b1 : B1, ..., bj : Bj)
  lazy C(c1 : C1, ..., ck : Ck) -> e_c
  lazy D(d1 : D1, ..., dl : Dl) -> e_d
```

might be encoded as

$$((C_1 \times \ldots \times C_k) + (D_1 \times \ldots \times D_l)) \nrightarrow_F ((A_1 \times \ldots \times A_i) + (B_1 \times \ldots \times B_j))$$

with: $F(lv) \;=\; \text{case } lv \:\{\; \mathsf{inl}\;(c_1,\;\ldots,\;c_k) \rightarrow e_c;\; \mathsf{inr}\;(d_1,\;\ldots,\;d_l) \rightarrow e_d \;\}$.

However, this encoding only works if e_c and e_d are guaranteed to return one of the constructors A or B. As described in Section 3.3, we want to allow lazy constructors to also return further lazy constructors. To model this, we wrap the lazy type into a recursive type:

$$\mu\alpha.\ ((C_1\ \times \ldots\ \times C_k)\ +\ (D_1\ \times \ldots\ \times D_l))\ \nrightarrow_F\ (((A_1\ \times \ldots\ \times A_i)\ +\ (B_1\ \times \ldots\ \times B_j))\ +\ \alpha)$$

This allows the function F to return either a normal constructor (inl) or another lazy constructor (inr). As we show in this section, we can perform more optimizations if we fuse the recursive type and the lazy constructors into an abstract type of *recursive lazy constructors* $A\ \twoheadrightarrow_F\ B := \mu\alpha.\ A\ \nrightarrow_F\ (B\ +\ \alpha)$. Our final encoding is then:

$$((C_1\ \times \ldots\ \times C_k)\ +\ (D_1\ \times \ldots\ \times D_l))\ \twoheadrightarrow_F\ ((A_1\ \times \ldots\ \times A_i)\ +\ (B_1\ \times \ldots\ \times B_j))$$

4.1 Core Calculus

Figure 2 shows the syntax and typing rules of a calculus with lazy constructors. The calculus is a standard lambda calculus restricted to be first-order. As such, we include units, sums, products and isorecursive types but not closures. We include top-level function declarations $F(x)\ =\ e\ :\ A \longrightarrow B$. The restriction to first-order is not necessary for the soundness of lazy constructors (and indeed, you can store closures in lazy constructors in Koka), but it emphasises our point that laziness can exist in a purely first-order setting.

To model lazy constructors, we add a new type $A\ \nrightarrow_F\ B$. This type represents the lazy computation that arises from applying the top-level function F to an argument of type A, evaluating to a value of type B. We can create a new value of this type by supplying an already computed value $v\ :\ B$ as memo v or an input $v\ :\ A$ to the computation as $\mathsf{lazy}_F\ v$. While these introduction forms are similar to a sum type, the elimination form step v always returns a value of type B, either by evaluating the computation F or by returning the memoized value. The introduction forms $\mathsf{lazy}_F\ v$ and memo v are not part of the syntax for values, since in the semantics they involve the side-effect of allocating a new location in a store which can persist the result of the lazy evaluation.

4.2 Natural Semantics

In Figure 3, we present a big-step semantics for our calculus. The judgement $\Gamma\ :\ e\ \Downarrow \Delta\ :\ v$ means that under store Γ the expression e will evaluate to store Δ and value v. For the standard features of our calculus, the big-step rules are straightforward and they do not modify the store.

Following the Natural Semantics for Lazy Evaluation [Launchbury 1993], we use a store Γ to keep track of thunks. While Launchbury stores expressions e in the store, we store lazy constructors of the form $\mathsf{lazy}_F\ v$. Assuming that it is known in advance what possible expressions can appear, these representations correspond where F abstracts the expression e as $e\ =\ F(v)$ with $v\ =\ \mathsf{fv}(e)$. Launchbury stores a fully evaluated expression as a value w, whereas we use the more explicit memo w.

The LAZY rule then follows Launchbury's Let-rule, where we create a new thunk in the store Γ and return a new reference to it. Similar to how the Let-rule applies both to expressions and values (since values are a subset of expressions), our LAZY rule applies to all lazy values (both $\mathsf{lazy}_F\ v$ and memo v). The STEP rule follows Launchbury's Variable-rule, where we remove x from the store, evaluate the computation and store the result in the new store. In the Variable-rule, the computed value is further transformed to rename all bound variables, but we can omit this step since our values do not contain lambdas and thus no bound variables. If the thunk happens to be evaluated already, we use the RECALL rule to access it.

Types:

$$A, B \;::=\; 1 \mid A + B \mid A \times B \mid \alpha \mid \mu\alpha.\, A \mid A \nrightarrow_F B$$

Values and Expressions:

$$\begin{array}{lcll|lcll}
v & ::= & x, y, z & \text{(variables)} & e & ::= & v \mid lv & \text{((lazy) values)} \\
 & \mid & () & \text{(unit)} & & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e & \text{(let binding)} \\
 & \mid & \mathsf{in}_l\ v \mid \mathsf{in}_r\ v & \text{(sum)} & & \mid & \mathsf{case}\ v\ \{\ \mathsf{in}_l\ x \rightarrow e;\ \mathsf{in}_r\ y \rightarrow e\ \} & \text{(case split)} \\
 & \mid & (v,\ v) & \text{(pair)} & & \mid & \mathsf{split}\ v\ \{\ (x,\ y) \rightarrow e\ \} & \text{(splitting pairs)} \\
 & \mid & \mathsf{fold}\ v & \text{(fold rec. type)} & & \mid & \mathsf{unfold}\ v & \text{(unfold rec. type)} \\
lv & ::= & \mathsf{memo}\ v & \textbf{(memoized value)} & & \mid & F\ v & \text{(application)} \\
 & \mid & \mathsf{lazy}_F\ v & \textbf{(lazy computation)} & & \mid & \mathsf{step}\ v & \textbf{(single-step forcing)}
\end{array}$$

$$\Sigma \;::=\; \varnothing \mid \Sigma, F(x) = e : A \rightarrow B \quad \text{(recursive top-level functions)}$$

$$\frac{}{\Gamma,\ x : A \vdash x : A}\ \textsc{Var} \qquad \frac{\Gamma \vdash e_1 : A \quad \Gamma,\ x : A \vdash e_2 : B}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : B}\ \textsc{Let}$$

$$\frac{\Gamma \vdash v : A_i \quad i \in \{l,\ r\}}{\Gamma \vdash \mathsf{in}_i\ v : A_l + A_r}\ \textsc{Inl/Inr} \qquad \frac{\Gamma \vdash v : A_l + A_r \quad \Gamma,\ x : A_i \vdash e_i : C}{\Gamma \vdash \mathsf{case}\ v\ \{\ \mathsf{in}_l\ x \rightarrow e_l;\ \mathsf{in}_r\ y \rightarrow e_r\ \} : C}\ \textsc{Case}$$

$$\frac{\Gamma \vdash v : A \quad \Gamma \vdash w : B}{\Gamma \vdash (v,\ w) : A \times B}\ \textsc{Pair} \qquad \frac{\Gamma \vdash v : A \times B \quad \Gamma,\ x : A,\ y : B \vdash e : C}{\Gamma \vdash \mathsf{split}\ v\ \{\ (x,\ y) \rightarrow e\ \} : C}\ \textsc{Split}$$

$$\frac{\Gamma \vdash v : A[\mu\alpha.\, A/\alpha]}{\Gamma \vdash \mathsf{fold}\ v : \mu\alpha.\, A}\ \textsc{Fold} \qquad \frac{\Gamma \vdash v : \mu\alpha.\, A}{\Gamma \vdash \mathsf{unfold}\ v : A[\mu\alpha.\, A/\alpha]}\ \textsc{Unfold}$$

$$\frac{}{\Gamma \vdash () : 1}\ \textsc{Unit} \qquad \frac{F : A \rightarrow B \in \Sigma \quad \Gamma \vdash v : A}{\Gamma \vdash F\ v : B}\ \textsc{App}$$

$$\frac{}{\Vdash \varnothing}\ \textsc{DefBase} \qquad \frac{\Vdash \Sigma \quad x : A \vdash e : B}{\Vdash \Sigma, F(x) = e : A \rightarrow B}\ \textsc{DefFun}$$

$$\frac{F : A \rightarrow B \in \Sigma \quad \Gamma \vdash v : A}{\Gamma \vdash \mathsf{lazy}_F\ v : A \nrightarrow_F B} \qquad \frac{F : A \rightarrow B \in \Sigma \quad \Gamma \vdash v : B}{\Gamma \vdash \mathsf{memo}\ v : A \nrightarrow_F B} \qquad \frac{\Gamma \vdash v : A \nrightarrow_F B}{\Gamma \vdash \mathsf{step}\ v : B}$$

Fig. 2. Syntax and Types for a First-order Calculus with Lazy Constructors

4.3 Soundness

We show that our calculus is sound with respect to our semantics using a logical relation. We include a step-indexing parameter k since our calculus includes isorecursive types and write $\Downarrow_k$ for evaluations that can be performed in k steps. A store Δ extends Γ in k steps, written as $\Gamma \sqsubseteq_k \Delta$, if Δ only contains more or more-evaluated thunks than Γ. Concretely, we take the reflexive-transitive closure of the rules:

$$\frac{}{\Gamma : v \Downarrow \Gamma : v}\ \textsc{value} \qquad \frac{\Gamma : e_1 \Downarrow \Delta : v \quad \Delta : e_2[v/x] \Downarrow \Theta : w}{\Gamma : \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow \Theta : w}\ \textsc{let}$$

$$\frac{F(x) = e \in \Sigma \quad \Gamma : e[v/x] \Downarrow \Delta : w}{\Gamma : F\ v \Downarrow \Delta : w}\ \textsc{app} \qquad \frac{\Gamma : e[v_1/x,\ v_2/y] \Downarrow \Delta : w}{\Gamma : \mathsf{split}\ (v_1,\ v_2)\ \{\ (x,\ y) \to e\ \} \Downarrow \Delta : w}\ \textsc{split}$$

$$\frac{z\ \text{fresh}}{\Gamma : lv \Downarrow (\Gamma,\ z \mapsto lv) : z}\ \textsc{lazy} \qquad \frac{}{\Gamma : \mathsf{unfold}\ (\mathsf{fold}\ v) \Downarrow \Gamma : v}\ \textsc{unfold}$$

$$\frac{z \mapsto \mathsf{memo}\ v \in \Gamma}{\Gamma : \mathsf{step}\ z \Downarrow \Gamma : v}\ \textsc{recall} \qquad \frac{\Gamma : e_i[v/x_i] \Downarrow \Delta : w}{\Gamma : \mathsf{case}\ (\mathsf{in}_i\ v)\ \{\ \mathsf{in}_l\ x_l \to e_l;\ \mathsf{in}_r\ x_r \to e_r\ \} \Downarrow \Delta : w}$$

$$\frac{\Gamma : F\ v \Downarrow \Delta : w}{(\Gamma,\ x \mapsto \mathsf{lazy}_F\ v) : \mathsf{step}\ x \Downarrow (\Delta,\ x \mapsto \mathsf{memo}\ w) : w}\ \textsc{step}$$

Fig. 3. Natural Semantics for Lazy Constructors

$$\frac{}{\Gamma \sqsubseteq_1 \Gamma,\ x \mapsto lv}\ \textsc{extend} \qquad \frac{\Gamma : F\ v \Downarrow_k \Delta : w}{\Gamma,\ x \mapsto \mathsf{lazy}_F\ v \sqsubseteq_{k+1} \Delta,\ x \mapsto \mathsf{memo}\ w}\ \textsc{eval}$$

As usual for logical relations we will argue that if an expression can evaluate to a value under a store Γ then it evaluates to the same value for all stores that extend Γ. In this interpretation, the LAZY rule thus encodes the notion of referential transparency: evaluating arbitrary thunks in the heap does not change whether (and to what) an expression evaluates.

Next, we define our meaning of values $\mathcal{V}_k[\![A]\!]$ and expressions $\mathsf{E}_{k,\Delta}[\![A]\!]$. Since we are working with a big-step semantics, we can not distinguish between stuck and diverging programs. However, we will prove that if a well-typed program evaluates to a value, then the value will have the correct type:

$$\mathsf{E}_{k,\,\Delta}[\![A]\!] := \{\ e \mid \forall j < k.\ \forall \Theta,\ v.\ (\Delta : e \Downarrow_j \Theta : v) \Rightarrow \Delta \sqsubseteq_j \Theta \text{ and } (\Theta,\ v) \in \mathcal{V}_{k-j}[\![\ A\ ]\!]\ \}$$

The interpretation of values is straightforward in all cases except for the lazy type. For a lazy value $\mathsf{lazy}_F\ v$, we need both that v is valid for type A and that the thunk can be evaluated to a value of type B at any time in the future:

$$\begin{aligned}
\mathcal{V}_k[\![\ \mu\alpha.\ A\ ]\!] &:= \{\ (\Delta,\ \mathsf{fold}\ v) \mid \forall j < k.\ (\Delta,\ v) \in \mathcal{V}_j[\![\ A[\mu\alpha.\ A/\alpha]\ ]\!]\ \} \\
\mathcal{V}_k[\![\ A \nrightarrow_F B\ ]\!] &:= \{\ ((\Delta,\ z \mapsto \mathsf{lazy}_F\ v),\ z) \mid (\Delta,\ v) \in \mathcal{V}_k[\![\ A\ ]\!],\ \forall j \leqslant k,\ \Theta.\ \Delta \sqsubseteq_j \Theta \Rightarrow \\
&\qquad\qquad F\ v \in \mathsf{E}_{k-j,\,\Theta}[\![B]\!]\ \} \\
&\quad \cup \{\ ((\Delta,\ z \mapsto \mathsf{memo}\ v),\ z) \mid (\Delta,\ v) \in \mathcal{V}_k[\![\ B\ ]\!]\ \}
\end{aligned}$$

With this setup, we can prove:

Lemma 1. (*Store extension preserves types.*)
If $(\Delta,\ v) \in \mathcal{V}_k[\![\ A\ ]\!]$ and $\Delta \sqsubseteq_j \Theta$, then $(\Theta,\ v) \in \mathcal{V}_{k-j}[\![\ A\ ]\!]$.

We connect our semantics to the type system by defining the semantic soundness relation $\Gamma \vDash e : A$ which implies that for all substitutions σ of variables in Γ by values of the correct type that are valid for k more steps, the evaluation of e will yield a value of type A in k steps (if it converges):

$$\Gamma \vDash e : A := \forall k \geqslant 0,\ \Delta,\ \sigma \in \mathcal{G}_{k,\Delta}[\![\Gamma]\!].\ \sigma(e) \in \mathsf{E}_{k,\Delta}[\![A]\!]$$

Then we obtain our type soundness result:

Theorem 1. (*Type Soundness.*)
If $\Gamma \vdash e : A$, then $\Gamma \vDash e : A$.

As an intermediate result from our soundness proof, we also see that evaluating the heap further does not change the final computed value. This shows that the evaluation of lazy constructors in the store in referentially transparent:

Theorem 2. (*Lazy evaluation is referentially transparent.*)
If $\Gamma : e \Downarrow \Delta : v$ and $\Gamma \sqsubseteq \Gamma'$, then $\Gamma' : e \Downarrow \Delta' : v$ with $\Delta \sqsubseteq \Delta'$.

4.4 Recursive Lazy Constructors

To model a type like the `stream`, where `SReverse` can evaluate to another lazy constructor `SReverse`, we need to encode lazy data types using iso-recursive types. In the style of a delay monad [Altenkirch et al. 2017; Capretta 2005; Chapman et al. 2019] or trampoline [Ganz et al. 1999], we can define a type of recursive lazy constructors as:

$$A \twoheadrightarrow_F B := \mu\alpha.\, A \rightarrow_F (B + \alpha)$$

where the `eval` function recursively steps the lazy constructors until we obtain a non-lazy constructor:

$$\mathsf{eval}\ x \ = \ \mathsf{case}\ (\mathsf{step}\ (\mathsf{unfold}\ x))\ \{\ \mathsf{inl}\ y \rightarrow y;\ \mathsf{inr}\ y \rightarrow \mathsf{eval}\ y\ \}$$

This design is only a small extension of our first model of lazy constructors, yet allows us to encode the full power of lazy constructors as in Section 3.3. However, while the definition above is a correct description of recursive lazy constructors, it turns out that our implementation contains another subtlety: it does not allow us to distinguish how many iterations `eval` had to perform.

Consider an inhabitant of this type like $\mathsf{fold}\ (\mathsf{memo}\ (\mathsf{inr}\ (\mathsf{fold}\ (\mathsf{memo}\ (\mathsf{inl}\ v)))))$. At first glance, it might appear that the outer indirection is unnecessary and that the value is in fact equivalent to $\mathsf{fold}\ (\mathsf{memo}\ (\mathsf{inl}\ v))$, since both values yield v when passed to eval. However, in the definition above, there is no restriction that lazy constructors are always deconstructed using eval, which makes it possible to distinguish the two values by simply case-splitting after a call to step. This is a well-known problem for implementations of laziness that attempt to short-cut indirections. For example, consider the following OCaml code:

```
let nested = lazy (lazy (raise Not_found))
let eval l = match l with lazy v -> ()
let () = eval nested; eval nested; ()
```

In this code, we evaluate the outer lazy twice and expect the exception *not* to be thrown. However, after the first evaluation, the lazy is rewritten into an indirection. If the runtime system attempted to short-cut the indirection, this would make `nested` point to the inner lazy value. Then the second evaluation would throw the exception, thus changing the semantics of the program. In practice, OCaml's runtime system still tries to short-cut indirections, but guarantees that this does not affect the semantics of the program; in particular, indirections are not removed if they lead to an unevaluated lazy value [4].

However, our example is a bit different from the OCaml example: while the two lazy values in OCaml belong to different thunks, in our case the two indirections morally belong to the same thunk. This suggests that if we were to make our encoding abstract, we could use this fact to short-cut the indirection.

[4] https://github.com/ocaml/ocaml/blob/4.14/runtime/minor_gc.c#L236

4.5 Short-Cutting Indirections

To be able to short-cut indirections, we thus need to ensure that the encoding of recursive lazy constructors stays abstract and make it a first-class type in our calculus. We introduce a new type $A \twoheadrightarrow_F B$ and define its interpretation by fusing the $A \twoheadrightarrow_F B$ type with the iso-recursive type wrapped around it:

$$\begin{aligned}\mathcal{V}_k[\![A \twoheadrightarrow_F B]\!] := \{ ((\Delta,\ z \mapsto \mathsf{lazy}_F\ v),\ z) \mid (\Delta,\ v) \in \mathcal{V}_k[\![A]\!],\ \forall j < k,\ \Theta.\ \Delta \sqsubseteq \Theta \Rightarrow \\ F\ v \in \mathsf{E}_{k-j,\Theta}[\![B + (A \twoheadrightarrow_F B)]\!] \} \\ \cup \{ ((\Delta,\ z \mapsto \mathsf{memo}\ v),\ z) \mid (\Delta,\ v) \in \mathcal{V}_k[\![B]\!] \} \\ \cup \{ ((\Delta,\ z \mapsto \mathsf{indirect}\ v),\ z) \mid \forall j < k.\ (\Delta,\ v) \in \mathcal{V}_j[\![A \twoheadrightarrow_F B]\!] \}\end{aligned}$$

Compared to earlier, a lazy value now evaluates to $B + (A \twoheadrightarrow_F B)$, but memo still only contains values of type B. Instead, if the lazy value evaluates to $A \twoheadrightarrow_F B$, then we create an indirection node pointing to the folded lazy constructor. The crucial aspect of this formalization is that we can now shortcut indirections. We formalize this using the following two rules:

$$\frac{y \mapsto \mathsf{indirect}\ z \in \Gamma}{\Gamma,\ x \mapsto \mathsf{indirect}\ y \sqsubseteq \Gamma,\ x \mapsto \mathsf{indirect}\ z}\ \textsc{cuti} \qquad \frac{y \mapsto \mathsf{memo}\ v \in \Gamma}{\Gamma,\ x \mapsto \mathsf{indirect}\ y \sqsubseteq \Gamma,\ x \mapsto \mathsf{memo}\ v}\ \textsc{cutm}$$

We do not allow shortcutting an indirection that points directly to a lazy constructor since that could duplicate work: if the lazy constructor would be duplicated into a different location in the store, its evaluation would be independent of the evaluation of the original location. Our referential transparency theorem still holds for this calculus:

Theorem 3. (*Short-cutting indirections is referentially transparent.*)
If $\Gamma : e \Downarrow \Delta : v$ and $\Gamma \sqsubseteq \Gamma'$, then $\Gamma' : e \Downarrow \Delta' : v$ with $\Delta \sqsubseteq \Delta'$.

5 Implementation

While Section 4 gives a high-level overview over lazy constructors, it does not give a direct strategy for implementing lazy constructors efficiently. In this section, we instead take a more low-level view. We propose several primitives that can be used to implement lazy constructors efficiently and derive an efficient recursive evaluation algorithm. Unlike the high-level step function of the previous section, our new low-level primitives do not preserve the referential transparency of lazy evaluation and should thus be exposed only as unsafe or kept hidden in the underbelly of a compiler.

In Section 2.3, we define a simple `stream/eval` function, but already noticed that it used too much stack space. In this section, we will show how to derive a more efficient implementation of `stream/eval`:

```
fun stream/eval( s : stream<a> )
  lazy match s
    SAppend( s1, s2 ) -> lazy-eval-sappend(s, s1, s2)
    SReverse( s1, acc ) -> lazy-eval-sreverse(s, s1, acc)
    Indirect(ind) -> eval(ind)
    _ -> s
fun lazy-eval-sappend( s : stream<a>, s1 : stream<a>, s2 : stream<a> )
  match s1
    SCons(x,xx) -> lazy-update(s, SCons(x, SAppend(xx, s2)))
    SNil        -> lazy-update(s, Indirect(s2)); eval(s2)
fun lazy-eval-sreverse( s : stream<a>, s1 : stream<a>, acc : stream<a> )
  match s1
    SCons(x,xx) -> lazy-eval-sreverse(s, xx, SCons(x, acc))
    SNil        -> lazy-update(s, Indirect(s2)); eval(s2)
```

Expressions:

$$\begin{array}{lll} e ::= & \ldots & \text{(as before)} \\ & \mid \; \mathsf{lazy\ match}\ v\ \{\ \mathsf{lazy}_F\ l\ y \rightarrow e;\ \mathsf{memo}\ y \rightarrow e\ \} & \text{(lazy match and acquire lock)} \\ & \mid \; \mathsf{memoize}\ l\ w & \text{(update cell and release lock)} \end{array}$$

$$\Sigma \ ::= \ \ldots \mid \Sigma, F(l; x) \ = \ e : A \rightarrow B \rightarrow C \quad \text{(top-level functions)}$$

We keep all rules as in the high-level core calculus except STEP. We augment each rule with a linear environment L of locations. For the introduction rules, UNFOLD rule and APP rule L is empty, the LET rule splits L among its antecedents while the CASE and SPLIT rules pass L to their antecedents. Furthermore we add the rules:

$$\frac{\Vdash \Sigma \qquad l : A \mid x : B \vdash e : C}{\Vdash \Sigma, F(l; x) \ = \ e : A \rightarrow B \rightarrow C}\ \text{DEFLAPP} \qquad \frac{F : A \rightarrow B \rightarrow C \in \Sigma \qquad \varnothing \mid \Gamma \vdash v : B}{l : A \mid \Gamma \vdash F\ l\ v : C}\ \text{LAPP}$$

$$\frac{\varnothing \mid \Gamma \vdash w : B}{l : A \nrightarrow_F B \mid \Gamma \vdash \mathsf{memoize}\ l\ w : B}\ \text{MEMOIZE}$$

$$\frac{\varnothing \mid \Gamma \vdash v : A \nrightarrow_F B \qquad L, l : A \nrightarrow_F B \mid x : A \vdash e_1 : C \qquad L \mid \Gamma, \ y : B \vdash e_2 : C}{L \mid \Gamma \vdash \mathsf{lazy\ match}\ v\ \{\ \mathsf{lazy}_F\ l\ x \rightarrow e_1;\ \mathsf{memo}\ y \rightarrow e_2\ \} : C}\ \text{LAZYMATCH}$$

Fig. 4. Low-level core calculus

Compared to our simpler implementation, we can see that we now write an indirection node if a lazy constructor returns another lazy constructor. This allows us to keep evaluating without using stack space, but means that we might have to follow an indirection chain from a previous evaluation. Furthermore, our simpler version generated an `eval(SReverse(xx, SCons(x,acc)))`, where a lazy constructor is created and immediatedly evaluated. In our new implementation, we instead directly jump to the correct evaluation function `lazy-eval-sreverse` which saves a branch and writes to memory.

Unfortunately, it is tricky to show that our final implementation is in fact correct, where a lazy constructor will be updated to the correct value. However, it turns out that we can derive this version by equational reasoning from the simpler implementation, which corresponds more clearly to our high-level calculus.

5.1 Implementation Calculus

Our low-level calculus is a variation of the high-level calculus. We introduce two new primitives: $\mathsf{lazy\ match}\ v\ \{\ \mathsf{lazy}_F\ l\ x \rightarrow e_1;\ \mathsf{memo}\ y \rightarrow e_2\ \}$ allows us to inspect the value of a lazy constructor. In the first branch e_1 we additionally get access to the location l of the lazy constructor. Our second primitive $\mathsf{memoize}\ l\ w$ allows us to overwrite the cell l of a lazy constructor with memo w. A location $l :: A \nrightarrow_F B$ acts as a destination for a value of type B [Allain et al. 2025; Bagrel and Spiwack 2025; Shaikhha et al. 2017], which can filled using memoize. In our semantics, all locations l returned by lazy match are *locked* and thus can not be accessed until the lock is released by memoize.

To ensure the soundness of the low-level calculus, we need to ensure that locked locations are handled linearly. In particular, this guarantees that a locked location is overwritten using memoize exactly once. We achieve this by adding a second environment L to the calculus that contains all locked locations and ensure that no locked location can ever escape into a value held in Γ. The rules

Store and evaluation context:

$$v ::= a \mid \ldots \text{ (heap cells)}$$
$$\varphi ::= \mathsf{memo}\ v \mid \mathsf{lazy}_F\ v \mid \mathsf{locked}$$
$$\mathsf{S} ::= \varnothing \mid \mathsf{S}, a \mapsto \varphi$$
$$\mathsf{E} ::= \square \mid \mathsf{let}\ x = \mathsf{E}\ \mathsf{in}\ e$$

$$\frac{\mathsf{S} \mid e_1 \longrightarrow \mathsf{S}' \mid e_2}{\mathsf{S} \mid \mathsf{E}[e_1] \longmapsto \mathsf{S}' \mid \mathsf{E}[e_2]}\ \textsc{step}$$

Evaluation steps:

(let)	$\mathsf{S} \mid \mathsf{let}\ x = v\ \mathsf{in}\ e$	$\longrightarrow$	$\mathsf{S} \mid e[v/x]$
(app)	$\mathsf{S} \mid F\ v$	$\longrightarrow$	$\mathsf{S} \mid e[v/x]$ where $F(x) = e \in \Sigma$
$(split)$	$\mathsf{S} \mid \mathsf{split}\ (v_1,\ v_2)\ \{\ (x,\ y) \to e\ \}$	$\longrightarrow$	$\mathsf{S} \mid e[v_1/x, v_2/y]$
$(case)$	$\mathsf{S} \mid \mathsf{case}\ (\mathsf{in}_i\ v)\ \{\ \mathsf{in}_l\ x_l \to e_l;\ \mathsf{in}_r\ x_r \to e_r\ \}$	$\longrightarrow$	$\mathsf{S} \mid e_i[v/x_i]$
$(unfold)$	$\mathsf{S} \mid \mathsf{unfold}\ (\mathsf{fold}\ v)$	$\longrightarrow$	$\mathsf{S} \mid v$
$(lazy)$	$\mathsf{S} \mid \mathsf{lazy}_F\ v$	$\longrightarrow$	$\mathsf{S},\ a \mapsto \mathsf{lazy}_F\ v \mid a$ $\quad a$ fresh
$(memo)$	$\mathsf{S} \mid \mathsf{memo}\ v$	$\longrightarrow$	$\mathsf{S},\ a \mapsto \mathsf{memo}\ v \mid a$ $\quad a$ fresh
$(memoize)$	$\mathsf{S},\ a \mapsto \mathsf{locked} \mid \mathsf{memoize}\ a\ w$	$\longrightarrow$	$\mathsf{S},\ a \mapsto \mathsf{memo}\ w \mid w$

$$
\begin{array}{lll}
(\textit{lazy match}) & \mathsf{S},\ a \mapsto v \mid \mathsf{lazy\ match}\ a\ \{\ \mathsf{lazy}_F\ l\ x \to e_1;\ \mathsf{memo}\ y \to e_2\ \} & \\
\qquad\longrightarrow & \mathsf{S},\ a \mapsto \mathsf{locked} \mid e_1[a/l, w/x] & \text{if } v = \mathsf{lazy}_F\ w \\
\qquad\longrightarrow & \mathsf{S},\ a \mapsto \mathsf{memo}\ w \mid e_2[w/y] & \text{if } v = \mathsf{memo}\ w
\end{array}
$$

Fig. 5. Small-step semantics of implementation

of our high-level calculus can be modified to treat the L environment linearly in the usual way.

In further preparation, we add a new type of top-level function $F(l; x)$ which also takes a location. This is necessary, since locked locations may not be stored in values and so we can not represent this by a product $F((l, x))$. Despite taking two arguments a function $F(l; x)$ has to be fully applied.

Given those primitive operations, we can implement the step operation as:

$$
\begin{array}{l}
\mathsf{step}\ x\ = \mathsf{lazy\ match}\ x \\
\qquad\qquad \mathsf{lazy}_F\ l\ v \to \mathsf{memoize}\ l\ (F\ v) \\
\qquad\qquad \mathsf{memo}\ y \to y
\end{array}
$$

It is easy to see that with this implementation of step, the original STEP rule of the high-level calculus becomes derivable. In particular, our low-level calculus strictly extends the high-level calculus:

Lemma 2. (*The low-level calculus implements the high-level calculus*)
If $\Gamma \vdash e : A$, then $\varnothing \mid \Gamma \vdash e : A$.

5.2 Small-Step Semantics

In Figure 5, we describe a small-step semantics for lazy constructors. As in the natural semantics, we only keep lazy constructors in the store. Each memory cell is either a lazy or memo value or locked. The small-semantics of the derived step function corresponds directly to the natural semantics of the primitive step: while the big-step semantics removes the cell x from the heap entirely during evaluation, the small-step semantics keeps it in the heap as locked and thus inaccessible. This allows us to prove that the small-step semantics faithfully implements the big-step semantics:

Lemma 3. (*The small-step semantics implements high-level semantics.*)
If $\Gamma : e \Downarrow \Delta : v$, then $\Gamma \mid e \longmapsto^* \Delta \mid v$.

Would it also be possible to show the reverse direction? In general this is not possible, since the implementation calculus allows us to memoize *any* value of the correct type, while the high-level calculus only allows us to memoize values that are produced by the evaluation of a lazy constructor.

However, for expressions that can be checked in the high-level calculus, the small-step semantics and the big-step semantics are equivalent.

Our description of cells under evaluation as "locked" mirrors our implementation: If Koka detects that a lazy constructor is thread-shared (recorded using reference counts [Ullrich and de Moura 2019]), Koka will use an atomic compare-and-swap to overwrite the tag of the lazy constructor with a special tag that indicates that the cell is being evaluated. The lazy match primitive acquires this lock during matching and the memoize primitive releases it. This ensures that a cell is evaluated at most once, even in the presence of multiple threads. However, as in other implementations of laziness, the evaluation of a lazy constructor may deadlock if it tries to evaluate itself. Haskell and OCaml can detect this case and throw an exception.

5.3 Tail-Recursive Evaluation

Using our new primitives, we can obtain a faster implementation of the lazy evaluation function. As before, we can define the recursive forcing function as:

$$\begin{aligned}
\mathsf{eval}\ x\ &= \mathsf{case}\ (\mathsf{step}\ (\mathsf{unfold}\ x))\ \{\ \mathsf{inl}\ y \to y;\ \mathsf{inr}\ y \to \mathsf{eval}\ y\ \}\\
&= \mathsf{lazy\ match}\ (\mathsf{unfold}\ x)\\
&\qquad\quad \mathsf{lazy}_F\ l\ v \to \mathsf{case}\ (\mathsf{memoize}\ l\ (F\ v))\ \{\ \mathsf{inl}\ y \to y;\ \mathsf{inr}\ y \to \mathsf{eval}\ y\ \}\\
&\qquad\quad \mathsf{memo}\ y \to \mathsf{case}\ y\ \{\ \mathsf{inl}\ y \to y;\ \mathsf{inr}\ y \to \mathsf{eval}\ y\ \}
\end{aligned}$$

In the first branch of the lazy-match we run $F\ v$, memoize its result and then case-split to find out if the result can be returned (inl) or is a lazy constructor that needs to be evaluated further.

It turns out that we can often avoid the case-split if we specialize the eval function to the concrete function F that is evaluated in the lazy constructor. To achieve this we will use equational reasoning in the style of [Leijen and Lorenzen 2023 2025]. First, we define the translation $[\![e]\!]_l$ as:

$$[\![e]\!]_l\ = \mathsf{case}\ (\mathsf{memoize}\ l\ e)\ \{\ \mathsf{inl}\ y \to y;\ \mathsf{inr}\ y \to \mathsf{eval}\ y\ \}$$

For the function F that is evaluated in eval, we define a specialized function F' with the definition:

$$F'(l;\ v)\ =\ [\![F\ v]\!]_l$$

and we can use it in our eval function as:

$$\begin{aligned}
\mathsf{eval}\ x\ &= \mathsf{lazy\ match}\ (\mathsf{unfold}\ x)\\
&\qquad\quad \mathsf{lazy}_F\ l\ v \to F'\ l\ v\\
&\qquad\quad \mathsf{memo}\ v \to \mathsf{case}\ v\ \{\ \mathsf{inl}\ y \to y;\ \mathsf{inr}\ y \to \mathsf{eval}\ y\ \}
\end{aligned}$$

So far, nothing has happened: our new evaluation function corresponds exactly to the previous version. However, we have shifted the position of the case-split from the eval function into the translation. The key insight is now that we can improve the translation $[\![e]\!]_l$ by specializing it to different syntactic constructs. For example, several syntactic constructs permute with both memoize l and case:

$$\begin{aligned}
&[\![\mathsf{let}\ y\ =\ e_1\ \mathsf{in}\ e_2]\!]_l && =\ \mathsf{let}\ y\ =\ e_1\ \mathsf{in}\ [\![e_2]\!]_l\\
&[\![\mathsf{case}\ v\ \{\ \mathsf{inl}\ y \to e_1;\ \mathsf{inr}\ y \to e_2\ \}]\!]_l && =\ \mathsf{case}\ v\ \{\ \mathsf{inl}\ y \to [\![e_1]\!]_l;\ \mathsf{inr}\ y \to [\![e_2]\!]_l\ \}\\
&[\![\mathsf{split}\ v\ \{\ (y,\ z) \to e\ \}]\!]_l && =\ \mathsf{split}\ v\ \{\ (y,\ z) \to [\![e]\!]_l\ \}
\end{aligned}$$

This means that we can push down the memoization and case-split into the return values of the computation F. If the translated expression e is well-typed, there are few possible return values to F. For variables (and unfolds of variables), we have to perform the memoization and case-split. But in some cases we can do better:

$$\begin{aligned}
&[\![\mathsf{inl}\ w]\!]_l = \ \mathsf{memoize}\ l\ (\mathsf{inl}\ w);\ w\\
&[\![\mathsf{inr}\ w]\!]_l = \ \mathsf{memoize}\ l\ (\mathsf{inr}\ w);\ \mathsf{eval}\ w
\end{aligned}$$

If the function F ends in inl w, the evaluation ends at this point. We can thus memoize this result

$$\frac{\varnothing \mid \Gamma \vdash w : B}{l : A \twoheadrightarrow_F B \mid \Gamma \vdash \mathsf{memoize}\ l\ w : B} \qquad \frac{\varnothing \mid \Gamma \vdash w : A \twoheadrightarrow_F B}{l : A \twoheadrightarrow_F B \mid \Gamma \vdash \mathsf{indirect}\ l\ w : A \twoheadrightarrow_F B}$$

$$\frac{\begin{array}{cc}\varnothing \mid \Gamma \vdash v : A \twoheadrightarrow_F B & L, l : A \twoheadrightarrow_F B \mid x : A \vdash e_1 : C \\ L \mid \Gamma,\ y : A \twoheadrightarrow_F B \vdash e_2 : C & L \mid \Gamma,\ z : B \vdash e_3 : C\end{array}}{L \mid \Gamma \vdash \mathsf{lazy\ match}\ v\ \{\ \mathsf{lazy}_F\ l\ x \to e_1;\ \mathsf{indirect}\ y \to e_2;\ \mathsf{memo}\ z \to e_3\ \} : C}\ \text{LAZYMATCH}$$

Evaluation steps:

$$\begin{array}{llll}
(\textit{memoize}) & \mathrm{S},\ a \mapsto \mathsf{locked} \mid \mathsf{memoize}\ a\ w & \longrightarrow & \mathrm{S},\ a \mapsto \mathsf{memo}\ w \mid w \\
(\textit{indirect}) & \mathrm{S},\ a \mapsto \mathsf{locked} \mid \mathsf{indirect}\ a\ w & \longrightarrow & \mathrm{S},\ a \mapsto \mathsf{indirect}\ w \mid w
\end{array}$$

$$\begin{array}{lll}
(\textit{lazy match}) & \mathrm{S},\ a \mapsto v \mid \mathsf{lazy\ match}\ a\ \{\ \mathsf{lazy}_F\ l\ x \to e_1;\ \mathsf{indirect}\ y \to e_2;\ \mathsf{memo}\ z \to e_3\ \} & \\
\longrightarrow & \mathrm{S},\ a \mapsto \mathsf{locked} \mid e_1[a/l, w/x] & \text{if } v = \mathsf{lazy}_F\ w \\
\longrightarrow & \mathrm{S},\ a \mapsto \mathsf{indirect}\ y \mid e_2[w/y] & \text{if } v = \mathsf{indirect}\ w \\
\longrightarrow & \mathrm{S},\ a \mapsto \mathsf{memo}\ w \mid e_3[w/z] & \text{if } v = \mathsf{memo}\ w
\end{array}$$

Fig. 6. Short-cutting indirections

and return w without an extra case-split. If the function F ends in inr w, we also memoize the intermediate result and directly continue evaluating it.

In our implementation, these two cases enable the in-place update of lazy constructors: Since we know the size of the data that is memoized, we can often avoid creating an indirection node memo and instead write the data directly into the lazy cell.

Another interesting special case is if we can already see syntactically what the next lazy thunk will be. This happens for example when a `SReverse` constructor is evaluated to another `SReverse`. In that case, we can specialize the call to eval further:

$$\begin{array}{rl}
[\![\mathsf{inr}\ (\mathsf{fold}\ (\mathsf{lazy}_F\ v))]\!]_l & = \mathsf{case}\ (\mathsf{memoize}\ l\ (\mathsf{inr}\ (\mathsf{fold}\ (\mathsf{lazy}_F\ v))))\ \{\ \mathsf{inl}\ y \to y;\ \mathsf{inr}\ y \to \mathsf{eval}\ y\ \} \\
& = \mathsf{let}\ y\ =\ \mathsf{lazy}_F\ v\ \mathsf{in}\ \mathsf{memoize}\ l\ (\mathsf{inr}\ (\mathsf{fold}\ y));\ \mathsf{eval}\ (\mathsf{fold}\ y) \\
& = \mathsf{let}\ y\ =\ \mathsf{lazy}_F\ v\ \mathsf{in}\ \mathsf{memoize}\ l\ (\mathsf{inr}\ (\mathsf{fold}\ y));\ \mathsf{lazy\ match}\ y \\
& \qquad \mathsf{lazy}_F\ l\ v \to F'\ l\ v \\
& \qquad \mathsf{memo}\ v \to \mathsf{case}\ v\ \{\ \mathsf{inl}\ y \to y;\ \mathsf{inr}\ y \to \mathsf{eval}\ y\ \} \\
& = \mathsf{let}\ y\ =\ \mathsf{lazy}_F\ v\ \mathsf{in}\ \mathsf{memoize}\ l\ (\mathsf{inr}\ (\mathsf{fold}\ y));\ \mathsf{lock}\ y\ \mathsf{in}\ F'\ y\ v
\end{array}$$

In the last line, we now create a new lazy cell, memoize it in l and then immediately jump to F'. Since F' expects y to be locked, we use the macro:

$$\mathsf{lock}\ y\ \mathsf{in}\ e := \mathsf{lazy\ match}\ y\ \{\ \mathsf{lazy}_F\ y\ \_ \to e;\ \mathsf{memo}\ v \to \mathsf{impossible}\ \}$$

However, this might seem slightly wasteful: Why do we write the result into a new cell y and write an indirection into l when we could just write the result into l? This is not quite possible so far, since we do not consider indirections specially and thus run into the problem of Section 4.4. However, by considering indirections as proposed in Section 4.5, we can use an additional reasoning step to reduce the last line to just $F'\ l\ v$.

5.4 Short-Cuts During Evaluation

To be able to short-cut indirections, we change our memoize and lazy match primitives and add a new indirect primitive as in Figure 6. The indirect instruction acts like memoize but puts an indirect into the store that can be safely shortcut as shown in Section 4.5. On the locations l we now have to record the full type of the lazy constructor, where a location of type of type $A \twoheadrightarrow_F B$

Expressions:

$$\begin{array}{lll} e ::= & \ldots & \text{(as before)} \\ \mid & \mathsf{link}\ l\ (l',\ v)\ \mathsf{in}\ e & \text{(link cells and keep lock)} \\ \mid & \mathsf{unlink}\ \{\ \mathsf{inl}\ () \rightarrow e;\ \mathsf{inr}\ (l,\ x) \rightarrow e\ \} & \text{(unlink locked cells)} \end{array}$$

$$\varphi ::= \mathsf{memo}\ v\ \mid \mathsf{lazy}_F\ v\ \mid \mathsf{locked}\ \mid \mathsf{locked}\ (z,\ v)$$

$$\begin{array}{lll} (\mathit{link}) & S,\ a \mapsto \mathsf{locked} \mid \mathsf{link}\ a\ (a',\ v)\ \mathsf{in}\ e & \longrightarrow S,\ a \mapsto \mathsf{locked}\ (a',\ v) \mid e \\ (\mathit{unlink}) & S,\ a \mapsto v \mid \mathsf{unlink}\ a\ \{\ \mathsf{inl}\ () \rightarrow e_1;\ \mathsf{inr}\ (l,\ x) \rightarrow e_2\ \} & \\ \longrightarrow & S,\ a \mapsto \mathsf{locked} \mid e_1 & \text{if } v\ =\ \mathsf{locked} \\ \longrightarrow & S,\ a \mapsto \mathsf{locked} \mid e_2[a'/l, w/x] & \text{if } v\ =\ \mathsf{locked}\ (a',\ w) \end{array}$$

$$\frac{\varnothing \mid \Gamma \vdash v : A \qquad L,\ l : \Box_A\ B \mid \Gamma \vdash e : C}{L,\ l : B,\ l' : \Box_A\ B \mid \Gamma \vdash \mathsf{link}\ l\ (l',\ v)\ \mathsf{in}\ e : C}\ \textsc{link}$$

$$\frac{L \mid \Gamma \vdash e_1 : C \qquad L,\ l : B,\ l' : \Box_A\ B \mid \Gamma,\ x : A \vdash e_2 : C}{L,\ l : \Box_A\ B \mid \Gamma \vdash \mathsf{unlink}\ l\ \{\ \mathsf{inl}\ () \rightarrow e_1;\ \mathsf{inr}\ (l',\ x) \rightarrow e_2\ \} : C}\ \textsc{unlink}$$

Fig. 7. Linking cells

can be filled either with an indirection to another value of type $A \twoheadrightarrow_F B$ or with the memoized result B. Given those primitive operations, we can refine the eval operation from Section 5.3 as:

$$\begin{array}{l} \mathsf{eval}\ x\ = \mathsf{lazy\ match}\ x \\ \qquad\qquad \mathsf{lazy}_F\ l\ v \rightarrow \mathsf{case}\ (F\ v)\ \{\ \mathsf{inl}\ y \rightarrow \mathsf{memoize}\ l\ y;\ \mathsf{inr}\ y \rightarrow \mathsf{eval}\ (\mathsf{indirect}\ l\ y)\ \} \\ \qquad\qquad \mathsf{indirect}\ y \rightarrow \mathsf{eval}\ y \\ \qquad\qquad \mathsf{memo}\ y \rightarrow y \end{array}$$

We can now repeat the calculation from the previous section. The cases for let-bindings, case-statements and split-statements are the same and for inl w and inr w we obtain similar terms. The main difference is in the special inr ($\mathsf{lazy}_F\ v$) case. After calling $F'\ y\ v$, y points to a chain of indirections ending in a memo. The cutm and cuti short-cutting rules give us the laws:

$$\begin{array}{l} \mathsf{indirect}\ l\ y;\ \mathsf{memoize}\ y\ z\ =\ \mathsf{memoize}\ l\ z;\ \mathsf{memoize}\ y\ z \\ \mathsf{indirect}\ l\ y;\ \mathsf{indirect}\ y\ z\ =\ \mathsf{indirect}\ l\ z;\ \mathsf{indirect}\ y\ z \end{array}$$

This allows us to replace the evaluation of $F'\ y\ v$ by $F'\ l\ v$:

$$\begin{array}{lll} [\![\mathsf{inl}\ w]\!]_l & = & \mathsf{memoize}\ l\ w \\ [\![\mathsf{inr}\ w]\!]_l & = & \mathsf{indirect}\ l\ w;\ \mathsf{eval}\ w \\ [\![\mathsf{inr}\ (\mathsf{lazy}_F\ v)]\!]_l & = & \mathsf{let}\ y\ =\ \mathsf{lazy}_F\ v\ \mathsf{in\ indirect}\ l\ y;\ \mathsf{lock}\ y\ \mathsf{in}\ F'\ y\ v \\ & = & \mathsf{let}\ y\ =\ \mathsf{lazy}_F\ v\ \mathsf{in\ lock}\ y\ \mathsf{in\ indirect}\ y\ l;\ F'\ l\ v \\ & = & F'\ l\ v \end{array}$$

In the last case, we are creating an indirection in y that points to l. But then the cell y is unused and we can avoid allocating it altogether.

In practice, we also short-cut indirections in the indirect case of the lazy match construct. This important in practice to avoid long chains of indirections, which can change the time complexity if traversed repeatedly. We discuss in more detail in our tech report [Lorenzen et al. 2025].

5.5 Linking Cells for Schorr-Waite Traversal

Leading up to our discussion of the Schorr-Waite traversal, we make another addition to our calculus. While we have so far represented locked cells by the value locked, our implementation actually only sets a flag in the header to indicate that they are locked. This means that the storage space of the cell remains available even while it is in a locked state. In our calculus, we use a new value $\mathsf{locked}\ (l,\ v)$ to indicate that a cell is in a locked state but contains both another location l and a value v.

We can write to a locked cell l using the $\mathsf{link}\ l\ (l',\ v)\ \mathsf{in}\ e$ construct and we can check whether a locked cell l contains a value using the $\mathsf{unlink}\ l$ construct. Cells that may contain a value have type $\square_A\ B$, which means they contain a linked value of type A and are a destination for type B. To create a linked chain in the first place, we assume that there is a cell $\mathsf{null}\ :\ \square_A\ B$ in the environment (corresponding to a NULL pointer in the implementation).

5.6 Schorr-Waite Evaluation of Lazy Constructors

As we saw in Section 2.5, the evaluation of thunks can lead to stack overflow, if there are recursive calls to eval in F. We can avoid this by further transforming the lazy evaluation function as $[\![e]\!]_{z,l}$ with an additional zipper z that is stored in the lazy cells themselves.

For the i-th call to eval in F, we write E_i to denote its evaluation-context. Let A_i be the product type of the free variables of E_i. We define the zipper as the sum type of the A_i and define an unroll function that for a given zipper extracts the correct evaluation context to continue:

$$\begin{aligned}
&\mathsf{unroll}\ l\ v\ = \mathsf{unlink}\ l\\
&\qquad\qquad\qquad \mathsf{inr}\ (z,\ \alpha) \rightarrow \mathsf{case}\ \alpha\ \{\ \overline{\mathsf{in}_i\ \alpha_i \rightarrow [\![\mathsf{E}_i[v]]\!]_{z,l}}\ \}\\
&\qquad\qquad\qquad \mathsf{inl}\ () \rightarrow v\ \}
\end{aligned}$$

Then we can extend our translation to find such evaluation contexts, construct the zipper and call the correct unroll function:

$$[\![\mathsf{E}_i[\mathsf{eval}\ v]]\!]_{z,l}\ =\ \mathsf{link}\ l\ (z,\ \mathsf{in}_i\ \alpha_i)\ \mathsf{in}\ \mathsf{unroll}\ l\ (\mathsf{eval}\ v)\ \mathsf{where}\ \alpha_i\ =\ \mathsf{fv}(\mathsf{E}_i)$$

Again, we see that nothing has changed: the call to unroll can be inlined to yield the left-hand-side, since the unlink in unroll just extract what we just linked into l. But this setup now provides us with a technique for making the call to eval tail-recursive. We define a new forcing function that includes the unroll:

$$\begin{aligned}
\mathsf{eval}'\ z\ x\ &=\ \mathsf{unroll}\ z\ (\mathsf{eval}\ x)\\
&=\ \mathsf{unroll}\ z\ (\mathsf{case}\ (\mathsf{step}\ (\mathsf{unfold}\ x))\ \{\ \mathsf{inl}\ y \rightarrow y;\ \mathsf{inr}\ y \rightarrow \mathsf{eval}\ y\ \})\\
&=\ \mathsf{case}\ (\mathsf{step}\ (\mathsf{unfold}\ x))\ \{\ \mathsf{inl}\ y \rightarrow \mathsf{unroll}\ z\ y;\ \mathsf{inr}\ y \rightarrow \mathsf{unroll}\ z\ (\mathsf{eval}\ y)\ \}\\
&=\ \mathsf{case}\ (\mathsf{step}\ (\mathsf{unfold}\ x))\ \{\ \mathsf{inl}\ y \rightarrow \mathsf{unroll}\ z\ y;\ \mathsf{inr}\ y \rightarrow \mathsf{eval}'\ z\ y\ \}\\
&=\ \mathsf{lazy\ match}\ (\mathsf{unfold}\ x)\\
&\qquad \mathsf{lazy}_F\ l\ v \rightarrow \mathsf{case}\ (\mathsf{memoize}\ l\ (F\ v)\ )\ \{\ \mathsf{inl}\ y \rightarrow \mathsf{unroll}\ z\ y;\ \mathsf{inr}\ y \rightarrow \mathsf{eval}'\ z\ y\ \}\\
&\qquad \mathsf{memo}\ y \rightarrow \mathsf{case}\ y\ \{\ \mathsf{inl}\ y \rightarrow \mathsf{unroll}\ z\ y;\ \mathsf{inr}\ y \rightarrow \mathsf{eval}'\ z\ y\ \}
\end{aligned}$$

This suggests that we change our interpretation function to:

$$[\![e]\!]_{z,l}\ = \mathsf{case}\ (\mathsf{memoize}\ l\ e\ )\ \{\ \mathsf{inl}\ y \rightarrow \mathsf{unroll}\ z\ y;\ \mathsf{inr}\ y \rightarrow \mathsf{eval}'\ z\ y\ \}$$

which yields our tail-recursive forcing function:

$$\begin{aligned}
\mathsf{eval}'\ z\ x\ &=\ \mathsf{lazy\ match}\ (\mathsf{unfold}\ x)\\
&\qquad \mathsf{lazy}_F\ l\ v \rightarrow F'\ z\ l\ v\\
&\qquad \mathsf{memo}\ y \rightarrow \mathsf{case}\ y\ \{\ \mathsf{inl}\ y \rightarrow \mathsf{unroll}\ z\ y;\ \mathsf{inr}\ y \rightarrow \mathsf{eval}'\ z\ y\ \}\\[1em]
\mathsf{eval}\ x\ &=\ \mathsf{eval}'\ \mathsf{null}\ x
\end{aligned}$$

As before we have:

$$\begin{aligned}
[\![\mathsf{let}\ y\ =\ e_1\ \mathsf{in}\ e_2]\!]_{z,l} &=\ \mathsf{let}\ y\ =\ e_1\ \mathsf{in}\ [\![e_2]\!]_{z,l}\\
[\![\mathsf{case}\ v\ \{\ \mathsf{inl}\ y \rightarrow e_1;\ \mathsf{inr}\ y \rightarrow e_2\ \}]\!]_{z,l} &=\ \mathsf{case}\ v\ \{\ \mathsf{inl}\ y \rightarrow [\![e_1]\!]_{z,l};\ \mathsf{inr}\ y \rightarrow [\![e_2]\!]_{z,l}\ \}\\
[\![\mathsf{split}\ v\ \{\ (y,\ z) \rightarrow e\ \}]\!]_{z,l} &=\ \mathsf{split}\ v\ \{\ (y,\ z) \rightarrow [\![e]\!]_{z,l}\ \}
\end{aligned}$$

But now our bases cases are:

$$\begin{aligned}
[\![\mathsf{inl}\ w]\!]_{z,l} &= \mathsf{memoize}\ l\ (\mathsf{inl}\ w);\ \mathsf{unroll}\ z\ w \\
[\![\mathsf{inr}\ w]\!]_{z,l} &= \mathsf{memoize}\ l\ (\mathsf{inr}\ w);\ \mathsf{eval}'\ z\ w \\
[\![\mathsf{inr}\ (\mathsf{lazy}_F\ v)]\!]_{z,l} &= \mathsf{let}\ y\ =\ \mathsf{lazy}_F\ v\ \mathsf{in}\ \mathsf{memoize}\ l\ (\mathsf{inr}\ (\mathsf{fold}\ y));\ \mathsf{lock}\ y\ \mathsf{in}\ F'\ z\ y\ v
\end{aligned}$$

Compared to our previous calculation, little has changed: we only pass the zipper on to eval′ and call unroll once evaluation is finished. In the remaining indirection in the last case can be short-cut to just $F'\ z\ l\ v$ as discussed in Section 5.4.

6 Benchmarks

To test the runtime performance of lazy constructors, we have implemented all the lazy queues and heaps presented by Okasaki [1998], both using the standard approach with lazy thunks and with our new approach using lazy constructors. We benchmark them in a sequential setting without sharing of the persistent data structures. Since the laziness has no performance benefits in this setting (even in a theoretical or amortized sense), this allows to isolate the performance overhead of laziness itself. We compare the following systems and implementations:

- Koka (lazy): Koka 3.1.3 (`-O2 -no-debug`) using the implementation given by Okasaki with Koka's traditional lazy type.
- Koka (strict): Same as Koka (lazy) but with all laziness removed from the implementations.
- Koka (lazy cons): Same as Koka (lazy) but using our custom implementation with lazy constructors.
- OCaml (lazy): OCaml 4.14.2 (`-O2`, `OCAMLRUNPARAM="s=16M"`) using the implementations as given by Okasaki using the `Lazy.t` type.
- OCaml (strict): Same as OCaml (lazy), but with all laziness removed from the implementations.
- Haskell (lazy): GHC 9.10.1 (`-O2 -threaded -fworker-wrapper-cbv` and `+RTS -N8 -A32M -qb0`) using the implementations given by Okasaki.
- Haskell (strict): Same as Haskell (lazy) but compiled using `-XStrict`.
- Koka (no reuse): As we discuss in the next section, reuse analysis [Lorenzen and Leijen 2022; Reinking, Xie et al. 2021] has a large impact on the performance of the benchmarks in Koka, so we also test the Koka benchmarks with the `-fno-reuse` flag which disables reuse analysis.

For queues, we iterate the following procedure 1000 times: we `snoc` 100 000 integers into a queue, where in the first iteration we generate the integers at random and in the following iterations we `uncons` the elements out of the previous queue. This setup means that at any time the memory contains up to two queues with about 100 000 elements combined. This keeps the RSS of the program stable over the course of a benchmark run which particularly helps garbage collected languages. For heaps, we use the same method where we `deleteMin` from one heap and `insert` into the next, but we only do this for 100 heaps.

The benchmarks results are shown in Figure 8 where we normalize against the run time of the strict versions. The figure shows from top-to-bottom the benchmarks for Koka, Koka with no-reuse, OCaml, and Haskell. The results support our three main claims:

(1) Compared to the strict version, traditional lazy thunks have a significant average performance overhead of 150% in Koka, 110% in OCaml, and 140% in Haskell. In contrast, lazy constructors have a smaller average overhead of just 43% in Koka.
(2) Lazy constructors are always faster than traditional lazy thunks.
(3) In some benchmarks, the performance overhead of lazy constructors is less than 25%, thus yielding lazy data structures that are close in performance to their strict counterparts, while maintaining superior theoretical properties.

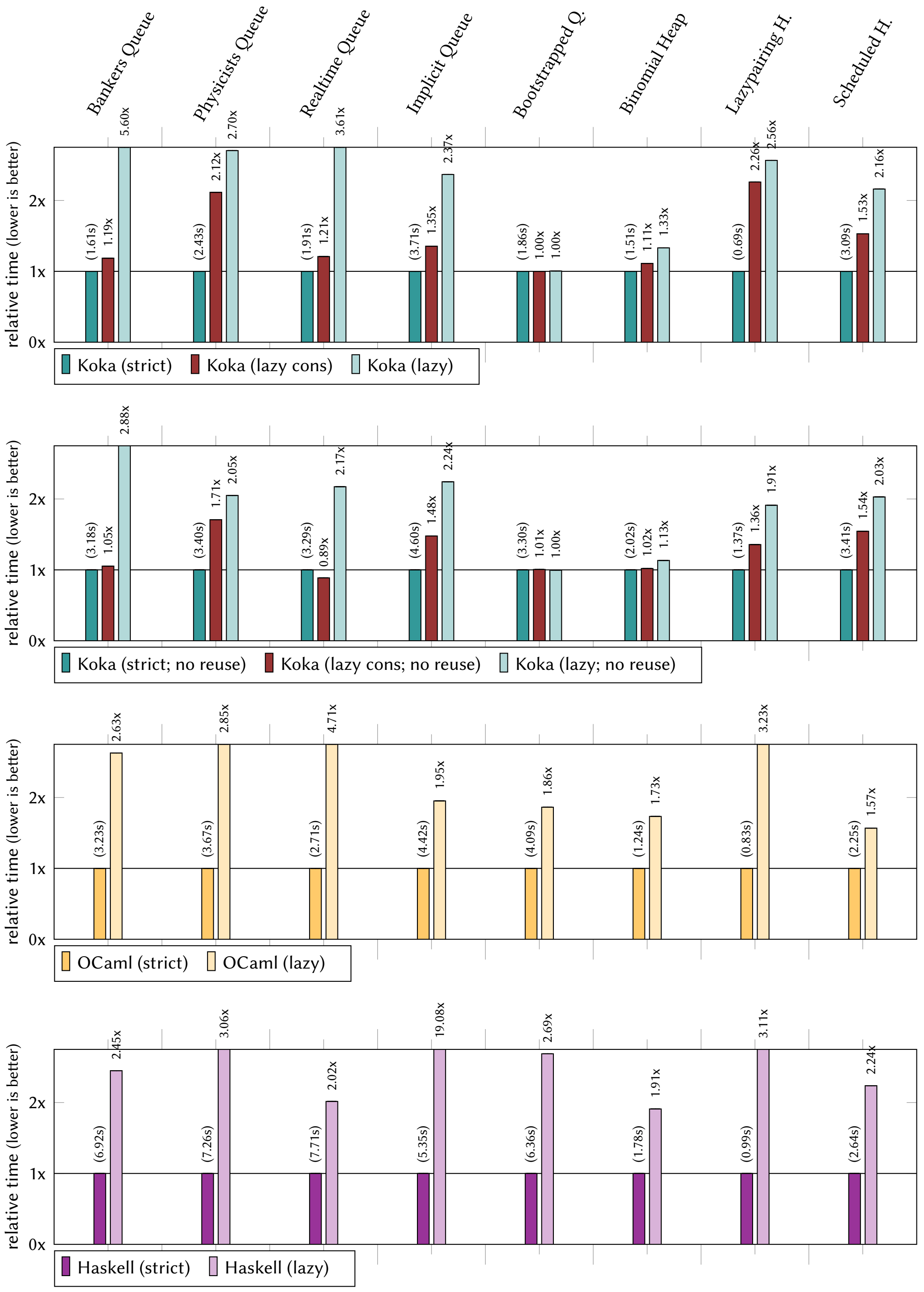

Fig. 8. Benchmarks on Apple M1 for Okasaki's queues and heaps in a sequential setting. Each graph shows results relative to the strict version of the benchmark. From top-to-bottom, the results are for Koka, Koka with no-reuse, OCaml, and Haskell.

6.1 Reuse Analysis and Laziness

We have split our analysis of Koka's performance into two parts: one with reuse analysis enabled and one with it disabled. Reuse analysis significantly improves the performance of strict data structures in Koka. However, it is much less useful in a lazy setting. To see this, consider the realtime queue, where we perform a lazy rotation using a `stream` type:

```
alias stream<a> = thunk<streamcell<a>>
type streamcell<a> =
  SNil
  SCons( x : a, xs : stream<a> )

fun rotate( front : stream<a>, rear : list<a>, sched : stream<a> ) : pure stream<a>
  match (front.eval(), rear)
    (SNil, Cons(y,_))         -> memo(SCons(y,sched ))
    (SCons(x,xs), Cons(y,ys)) -> delay{ SCons(x, rotate(xs,ys,memo(SCons(y,sched))) ) }
```

Here, `memo` creates a thunk from an existing value, while `delay` creates a thunk from a computation. Crucially, while reuse analysis can reuse the space from the `Cons` cell for the `SCons` cell passed to `memo` in the first branch, it does not apply to the constructors in the closure passed to `delay`. This is because reuse analysis never considers reuse opportunities under closures, since closures can be run arbitrarily often (although, in this case, this is a missing optimization since the `delay` function guarantees that the closure is only called once).

In contrast, when laziness is removed, the `stream` becomes a simple list and both the `front` and `rear` list can now be reused for the final result:

```
fun rotate( front : list<a>, rear : list<a>, sched : list<a> ) : pure list<a>
  match (front, rear)
    (Nil, Cons(y,_))          -> Cons(y, sched)
    (Cons(x,xs), Cons(y,ys)) -> Cons(x, rotate(xs,ys,Cons(y,sched)))
```

In practice, reuse analysis can significantly speed-up both the strict version of this data structure and our version with lazy constructors, while the traditional lazy version remains mostly unaffected.

7 Related Work

Quotient Types. Lazy constructors $A \nrightarrow_F B$ can be viewed as a quotient type obtained by quotienting the sum-type $A + B$ by the step function, while recursive lazy constructors $A \ntwoheadrightarrow_F B$ are the sum type quotiented by eval. In general, it is possible to mutate quotient types under the hood, where one element of an equivalence class can be swapped with any other element of the same class without breaking referential transparency. This trick was proposed by Selsam et al. [2020] and was a major inspiration for this work. They show how to implement pointer equality and hash-based memoization under a quotient so that they can be used in pure code, but do not consider the interaction with recursive types.

Semantics of Laziness. Nailing down the semantics of laziness was a longstanding problem. Launchbury [1993] was the first to give a natural semantics for lazy evaluation. The key insight was to require all function arguments to be let-bound; whenever an argument was evaluated, the corresponding heap location was updated. Sestoft [1997] has derived an abstract machine from Launchbury's lazy semantics. Even though the resulting machine is first order, it still needs to handle arbitrary lazy closures. Deriving a similar machine from our semantics for lazy constructors would be interesting further work. More recently, Nakata and Hasegawa [2009] have given an alternative small-step and big-step semantics for laziness, that has been proven to be equivalent to the original natural semantics by Launchbury.

Laziness in OCaml. OCaml short-cuts indirection nodes during GC when the indirection points to a strict value. If the indirection node points to a lazy value or another indirection, it can not be short-cut to preserve the soundness of lazy pattern matching. In recent versions of OCaml, the short-cutting of indirection nodes is disabled when using instrumentation [Dolan 2018] and during major GC [Dolan 2021]. However, in practice, it appears that most lazy values are either forced early and their indirection short-cut during minor GC or not forced at all [Scherer et al. 2021]. OCaml's implementation of laziness currently does not support multicore [Scherer et al. 2021].

Laziness in Haskell. In Haskell, laziness is pervasive: all function arguments are evaluated lazily by default. As such, Haskell compilers use sophisticated techniques to make this efficient [Hartel 1991; Johnsson 1984; Marlow and Peyton Jones 2004 2006; Marlow et al. 2007; Peyton Jones 1992]. Just like our lazy constructors, early GHC implementations based on Spineless Tagless G-Machine [Peyton Jones 1992] used to update closures in-place if the closure was large enough. This approach was later abandoned [Marlow and Peyton Jones 1998,page 12] in favor of a more uniform return convention. As we showed in Section 5.3, we need to be careful to preserve lazy semantics when applying optimizations, and the GHC compiler takes great care to preserve laziness during its many program transformations [Peyton Jones and Lester 1991; Peyton Jones et al. 1996].

8 Limitations and Future Work

The Global Nature of Defunctionalization. One drawback of lazy constructors is that they have to be declared up-front in the data type definition. This means that our approach does not support adding lazy constructors to a data type defined in a different module or library. To enable users of a library to define their own thunks, its author may add special lazy constructors such as `lazy SLazy(f : () -> stream<a>) -> f()` to their types. Users can then use these lazy constructors to define their own thunks, but they will be based on closures and not defunctionalized. To also benefit from defunctionalization, a programming language could combine lazy constructors with open data types [Löh and Hinze 2006]. However, open lazy constructors would have to carry a function pointer to their evaluation function, which makes it harder to see where reuse happens and we expect the performance may be slightly worse due to the indirect call.

GADTs and Effects. As typical for defunctionalized programs [Pottier and Gauthier 2004], some thunks can only be expressed as lazy constructors if the language supports GADTs [Cheney and Hinze 2003; Xi et al. 2003]. Koka currently assumes that for a type like `stream<a>`, all lazy constructors return a `stream<a>` and not, for example, a `stream<int>`. To lift this restriction, we would have to implement GADTs, so that the branches of the `lazy match` construct can make use of this type information. Furthermore, Koka assumes that lazy constructors perform no effects except divergence. We could allow lazy constructors to perform other effects (eg. throw an exception, write to a reference, I/O), but then every function that matches on a lazy data type would have to be annotated by those effects, leading to the expression problem if a new lazy constructor is added later.

Concurrency. Since Koka's reference counting scheme makes it possible to efficiently determine whether a thunk is thread-shared or not [Reinking, Xie et al. 2021; Ullrich and de Moura 2019], our implementation can use a fast-path and does not have to acquire or release locks in single-threaded programs. For thread-shared thunks, we implement blackholing using an atomic compare-and-swap operation, where other threads will busy-wait until the evaluation of the lazy constructor is complete. This is an inefficient strategy, which is likely to be slower than more advanced schemes [Harris et al. 2005]. However, we plan to change the implementation in the future to block threads an a mutex. In particular, any lazy constructor has at least one field (to hold a possible indirection) and while being black-holed, we can use that field to store a mutex on which other threads block.

Acknowledgements

We are grateful to the anonymous reviewers and Gabriel Scherer for many helpful comments and suggestions. We thank Stephen Dolan and András Kovács for previous discussions about benchmarking. Sam Lindley was supported by UKRI Future Leaders Fellowship "Effect Handler Oriented Programming" (MR/T043830/1 and MR/Z000351/1).

References

Allain, Bour, Clément, Pottier, and Scherer. Jan. 2025. Tail Modulo Cons, OCaml, and Relational Separation Logic. *Proc. ACM Program. Lang.* 9 (POPL). Association for Computing Machinery, New York, NY, USA. doi:10.1145/3704915.

Altenkirch, Danielsson, and Kraus. 2017. Partiality, Revisited: The Partiality Monad as a Quotient Inductive-Inductive Type. In *International Conference on Foundations of Software Science and Computation Structures*, 534–549. Springer. doi:10.1007/978-3-662-54458-7_31.

Bagrel, and Spiwack. 2025. Destination Calculus: A Linear Lambda-Calculus for Purely Functional Memory Writes. *Proceedings of the ACM on Programming Languages* 9 (OOPSLA1). ACM New York, NY, USA: 253–279. doi:10.1145/3720423.

Capretta. 2005. General Recursion via Coinductive Types. *Log. Methods Comput. Sci.* 1 (2). doi:10.2168/LMCS-1(2:1)2005.

Chapman, Uustalu, and Veltri. 2019. Quotienting the Delay Monad by Weak Bisimilarity. *Mathematical Structures in Computer Science* 29 (1): 67–92. doi:10.1017/S0960129517000184.

Cheney, and Hinze. 2003. *First-Class Phantom Types*. Cornell University.

Dolan. 2018. AFL Stability Fixes for Objects (MPR#7725) and Lazy Values. `https://github.com/ocaml/ocaml/pull/1754`.

Dolan. 2021. Speed up GC by Prefetching during Marking. `https://github.com/ocaml/ocaml/pull/10195`.

Ganz, Friedman, and Wand. 1999. Trampolined Style. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*, 18–27. doi:10.1145/317636.317779.

Gill. 2000. Debugging Haskell by Observing Intermediate Data Structures. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Haskell, Haskell 2000, Satellite Event of PLI 2000, Montreal, Canada, September 17, 2000*, edited by Graham Hutton, 41:1. Electronic Notes in Theoretical Computer Science 1. Elsevier. doi:10.1016/S1571-0661(05)80538-9.

Harris, Marlow, and Jones. 2005. Haskell on a Shared-Memory Multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, 49–61. doi:10.1145/1088348.1088354.

Hartel. 1991. Performance of Lazy Combinator Graph Reduction. *Software: Practice and Experience* 21 (3). Wiley Online Library: 299–329. doi:10.1002/SPE.4380210306.

Huet. 1997. The Zipper. *Journal of Functional Programming* 7 (5): 549–554. doi:10.1017/S0956796897002864.

Johnsson. 1984. Efficient Compilation of Lazy Evaluation. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, 58–69. doi:10.1145/502874.502880.

Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 144–154. POPL '93. Charleston, South Carolina, USA. doi:10.1145/158511.158618.

Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *MSFP'14, 5th Workshop on Mathematically Structured Functional Programming*. doi:10.4204/EPTCS.153.8.

Leijen, and Lorenzen. Jan. 2023. Tail Recursion Modulo Context: An Equational Approach. *Proc. ACM Program. Lang.* 7 (POPL). doi:10.1145/3571233.

Leijen, and Lorenzen. Jan. 2025. Tail Recursion Modulo Context: An Equational Approach (extended Version). Microsoft Research. `https://www.microsoft.com/en-us/research/publication/tail-recursion-modulo-context-an-equational-approach-extended-version/`. Under submission.

Lorenzen, and Leijen. Sep. 2022. Reference Counting with Frame Limited Reuse. In *Proceedings of the 27th ACM SIGPLAN International Conference on Functional Programming (ICFP'2022)*. ICFP'22. Ljubljana, Slovenia. doi:10.1145/3547634.

Lorenzen, Leijen, and Swierstra. 2023. FP^2: Fully in-Place Functional Programming. In *Proc. ACM Program. Lang.*, 7:275–304. ICFP. doi:10.1145/3607840.

Lorenzen, Leijen, Swierstra, and Lindley. 2024. The Functional Essence of Imperative Binary Search Trees. *Proc. ACM Program. Lang.* 8 (PLDI): 518–542. doi:10.1145/3656398.

Lorenzen, Leijen, Swierstra, and Lindley. Jul. 2025. *First-Order Laziness*. `https://antonlorenzen.de/papers/lazycons.pdf`.

Löh, and Hinze. 2006. Open Data Types and Open Functions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 133–144. doi:10.1145/1140335.1140352.

Marlow, and Peyton Jones. Jan. 1998. The New GHC/Hugs Runtime System. `https://www.microsoft.com/en-us/research/publication/the-new-ghchugs-runtime-system/`.

Marlow, and Peyton Jones. 2004. Making a Fast Curry: Push/enter vs. Eval/apply for Higher-Order Languages. ACM New York, NY, USA, 4–15. doi:10.1145/1016850.1016856.

Marlow, and Peyton Jones. 2006. Making a Fast Curry: Push/enter vs. Eval/apply for Higher-Order Languages. *Journal of Functional Programming* 16 (4-5). Cambridge University Press: 415–449. doi:10.1017/S0956796806005995.

Marlow, Yakushev, and Peyton Jones. 2007. Faster Laziness Using Dynamic Pointer Tagging. ACM, 277–288. doi:10.1145/1291151.1291194.

Moura, and Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, 12699:625–635. Lecture Notes in Computer Science. Springer. doi:10.1007/978-3-030-79876-5_37.

Nakata, and Hasegawa. 2009. Small-Step and Big-Step Semantics for Call-by-Need. *Journal of Functional Programming* 19 (6). Cambridge University Press: 699–722. doi:10.1017/S0956796809990219.

Okasaki. 1998. *Purely Functional Data Structures*. Cambridge University Press.

Peyton Jones. 1992. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. *Journal of Functional Programming* 2 (2). Cambridge University Press: 127–202. doi:10.1017/S0956796800000319.

Peyton Jones, and Lester. 1991. A Modular Fully-Lazy Lambda Lifter in Haskell. *Software: Practice and Experience* 21 (5). Wiley Online Library: 479–506. doi:10.1002/spe.4380210505.

Peyton Jones, Partain, and Santos. 1996. Let-Floating: Moving Bindings to Give Faster Programs. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, 1–12. doi:10.1145/232627.232630.

Pottier, and Gauthier. 2004. Polymorphic Typed Defunctionalization. ACM New York, NY, USA, 89–98. doi:10.1145/964001.964009.

Reinking, Xie, de Moura, and Leijen. 2021. Perceus: Garbage Free Reference Counting with Reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 96–111. PLDI 2021. ACK, New York, NY, USA. doi:10.1145/3453483.3454032.

Scherer, Bury, Leroy, Barnoy, Munch-Maccagnoni, White, Sivaramakrishnan, Bünzli, and Frisch. 2021. Discussing the Design of Lazy under Multicore. `https://github.com/ocaml-multicore/ocaml-multicore/issues/750`.

Schorr, and Waite. 1967. An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures. *Communications of the ACM* 10 (8). ACM New York, NY, USA: 501–506. doi:10.1145/363534.363554.

Schulte, and Grieskamp. 1992. Generating Efficient Portable Code for a Strict Applicative Language. In *Declarative Programming, Sasbachwalden 1991*, 239–252. Springer. doi:10.1007/978-1-4471-3794-8_16.

Selsam, Hudon, and Moura. 2020. Sealing Pointer-Based Optimizations behind Pure Functions. *Proceedings of the ACM on Programming Languages* 4 (ICFP). ACM New York, NY, USA: 1–20. doi:10.1145/3408997.

Sestoft. 1997. Deriving a Lazy Abstract Machine. *Journal of Functional Programming* 7 (3): 231–264. doi:10.1017/S0956796897002712.

Shaikhha, Fitzgibbon, Peyton Jones, and Vytiniotis. 2017. Destination-Passing Style for Efficient Memory Management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, 12–23. doi:10.1145/3122948.3122949.

Ullrich, and de Moura. Sep. 2019. Counting Immutable Beans – Reference Counting Optimized for Purely Functional Programming. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages (IFL'19)*. Singapore. doi:10.1145/3412932.3412935.

Wadler, Taha, and MacQueen. 1998. How to Add Laziness to a Strict Language without Even Being Odd. In *SML'98, The SML Workshop*.

Xi, Chen, and Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 224–235. doi:10.1145/604131.604150.