A Lightweight Approach to Formal Algorithmic Complexity

Functional Pearl

ANONYMOUS AUTHOR(S)

Proof assistants are typically used to verify programs' correctness – yet reasoning about the performance of programs appears to be less straightforward. This paper explores a lightweight technique for reasoning about the algorithmic complexity of purely functional data structures and algorithms: defining a cost function by induction on a program's graph. To illustrate this approach, this paper presents the worst case and amortized analysis of several data structures, including batched queues and real time queues. Using recent work on first order laziness, this paper reasons formally about sharing in a purely functional setting, without introducing an explicit heap or resorting to separation logic. Simple induction suffices.

CCS Concepts: • **Theory of computation → Complexity theory and logic**; **Program specifications**; **Functional constructs**; **Data structures design and analysis**.

1 Introduction

> Q: *Can proof assistants reason about complexity of programs?*
> A: *Complexity theory is just math. And all math can be formalized.*
> – an exchange on the Proof Assistants StackExchange forum[1]

Modern proof assistants are highly effective tools for reasoning about purely functional programs. While there is a large body of work proving functional correctness [see for instance, Appel 2026], this paper ploughs a different furrow: exploring a lightweight approach to reasoning about algorithmic complexity of purely functional algorithms and data structures. The approach presented here have several advantages:

- The key technique is simple enough to explain to an undergraduate student, yet powerful enough to handle a wide range of examples – including algorithms for sorting and searching, and data structures, such as batched queues and real time queues. Unlike existing approaches, this approach does not rely on type annotations of functions [Danielsson 2008] or working in a particular logical framework [Grodin et al. 2024; Niu et al. 2022]. In fact, the study of a function's algorithmic complexity is entirely decoupled from the function's definition, enabling the post-hoc verification of non-functional properties.
- Besides analysing the worst case time complexity, we prove the amortized complexity cost of ephemeral data structures. Using dependent types, the distinction between ephemeral usage and persistent usage is made explicit.
- Using recent work on *first-order laziness* [Lorenzen et al. 2025], we show how to reason about data structures that rely on lazy evaluation and sharing. This is notoriously difficult to do, yet our proposed solution avoids introducing an explicit heap or mutation.

[1] https://proofassistants.stackexchange.com/questions/1976/can-proof-assistants-reason-about-complexity-of-programs

ICFP '26, Indiana University, Indianapolis

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/XXXXXXX.XXXXXXX

This paper aims to dispel the widespread misconception that the *only* properties proof assistants can reason about is their extensional behaviour. Although the development we present is in Agda, the same ideas can be readily ported to any other proof assistant using dependent types, including Lean, Idris or Rocq.

2 Code is data: structural induction on the graph of a function

To illustrate the key ideas, we begin by revisiting a classical example of functional programming: list reversal. The 'fast' algorithm for list reversal uses an accumulating paramater:

```
reverse-acc : (xs acc : List A) → List A
reverse-acc []       acc = acc
reverse-acc (x : xs) acc = reverse-acc xs (x : acc)

fast-reverse : List A → List A
fast-reverse xs = reverse-acc xs []
```

The 'slow' algorithm for list reversal repeatedly calls the append function on lists:

```
slow-reverse : List A → List A
slow-reverse []       = []
slow-reverse (x : xs) = slow-reverse xs ++ [ x ]
```

A simple inductive argument shows that these two algorithms produce the same result. The key property relating these two definitions reads as follows:

```
reverse-correct : (xs ys : List A) → reverse-acc xs ys ≡ reverse xs ++ ys
```

Yet we have no way to argue that one algorithm is faster than the other. In fact, if the ambient type theory supports *function extensionality*, both these functions are *equal*, as they produce the same output for input lists.

What to do? Even in an intensional type theory, there is no way to observe a function's implementation. In this paper, we propose a lightweight approach: evoking the old functional programmer's maxim that 'code is data.' The key idea is to make the definition of a function explicity by reifying the *graph* of the function as an inductively defined *relation*. In general, the graph of a function `f : A → B` is a relation `R : A → B → Set`, such that `R x (f x)` holds. In the case of the reverseAcc function, this corresponds to the following data type:

```
data ReverseAcc : List A → List A → List A → Set where
  base : ReverseAcc [] ys ys
  step : ReverseAcc xs (x : ys) zs → ReverseAcc (x : xs) ys zs
```

You may want to think of this as a data type describing the *call graph* of the reverse-acc function; or alternatively, a Prolog programming for reversing one list onto another. Such relations provide an *intensional* inductive description of the function's behaviour. It is straightforward to prove that this definition is sound and complete with respect to the function's definition:

```
Reverse-acc-sound    : ReverseAcc xs acc ys → reverse-acc xs acc ≡ ys
Reverse-acc-complete : reverse-acc xs acc ≡ ys → ReverseAcc xs acc ys
```

But crucially, reifying the definition of the function in this fashion lets us compute the *size* of the graph and relate this to the size of its inputs. To do so, we define a simple *cost function*, a function that counts the size of the graph of our reversal function:

```
cost-reverse-acc : ReverseAcc xs ys zs → ℕ
cost-reverse-acc base     = 1
cost-reverse-acc (step r) = 1 + cost-reverse-acc r
```

The accumulating reverse function takes a number of steps linear in the size of its first argument. With these definitions, however, we make this more precise by *proving* that reverse-acc xs ys requires one step for each element of xs and one final step in the base case:

```
bound-reverse-acc : (rev : ReverseAcc xs ys zs) → cost-reverse-acc rev ≡ 1 + length xs
```

The proof uses straightforward induction on the graph of reverse-acc. In this fashion, reasoning about the complexity of a function definition is no harder than proving its functional correctness.

2.1 Size and *Ordnung*

Algorithmic complexity relates size of inputs to the number of steps necessary to produce the output. To measure things systematically, we introduce some overloaded notation that we will use throughout this paper. Typically, we are interested in measuring the 'size' of our inputs. We introduce the corresponding record, Size, with a single field:

```
record Size (A : Set) : Set where
  field
    |_| : A → ℕ
```

For example, we typically consider the size of a list to be its length:

```
instance
  size-List : Size (List A)
  size-List = record {|_| = length}
```

We will use Agda's *instance arguments* [Devriese and Piessens 2011] to find the necessary Size record, when needed. Now this notion of Size only works for types in Set. The graph of a function f : A → B, however, will have type A → B → Set. For such families of types, we introduce a separate notion of size for indexed families of types:

```
record Size₂ (P : A → B → Set) : Set where
  field
    |_|₂ : ∀ {x : A} {y : B} → P x y → ℕ
```

Similar definitions exist for relations on more than two arguments. In the remainder of this paper, we will omit the subscripts, as these can readily be inferred.

Typically, we are not interested in the exact number of steps required, but rather want an upper bound. We introduce a (simplified) version of the typical 'big O' notation:

```
_∈O_ : {{ Size A }} → (G : A → B → Set) → {{ Size G }} → (ℕ → ℕ) → Set
G ∈O f = ∃[ c₀ ] ∃[ c ] (g : G x y) → | x | ≡ n → | g | ⩽ c₀ + c * f (n)
```

We say that the relation G is in the complexity class f when we can find constants c_0 and c, such that the cost of the graph G x y is less than c_0 + c * f (n), for any input x of size n. This simplified definition will serve our purposes for this paper, but a much more careful treatment of 'big O notation' in type theory can be found in Guéneau's PhD thesis [2019].

Using this notation, we show that the accumulating reversal function is linear in its first argument:

```
rev-acc-linear : ∀ (ys : List A) → (λ xs → ReverseAcc xs ys) ∈O (λ n → n)
```

2.2 Quadratic complexity

What about our slow reversal function? Can we also prove that it is quadratic? To do so, we once again reify the slow-reverse function, together with the append function it uses, by defining the graphs of both functions:

```
data Append : List A → List A → List A → Set where
  base : Append [] ys ys
  step : Append xs ys zs → Append (x ∷ xs) ys (x ∷ zs)
data Slow-reverse : List A → List A → Set where
  base : Slow-reverse [] []
  step : Slow-reverse xs ys → Append ys [ x ] zs → Slow-reverse (x ∷ xs) zs
```

Next, we define the obvious cost functions. Since slow-reverse calls append, we add the associated cost to that of the recursive call:

```
append-cost : Append xs ys zs → ℕ
slow-reverse-cost : Slow-reverse xs ys → ℕ
slow-reverse-cost base           = 1
slow-reverse-cost (step rev app) = 1 + slow-reverse-cost rev + append-cost app
```

Unsurprisingly, we show that the slow-reverse functions takes a quadratic number of steps:

```
slow-reverse-quadratic : Slow-reverse ∈O (λ n → n * n)
```

The proof proceeds by induction on the graph of the slow-reverse function. The proof requires a few auxiliary properties of addition and multiplication, but is otherwise straightforward.

2.3 Limitations and caveats

Before covering more interesting examples, we want to reflect briefly on the general approach. Firstly, the function's graph and the associated cost are a crucial part of the problem specification. It is all too easy provide 'incorrect' definitions. Here is an alternative definition of the graph of our slow-reverse function:

```
data Slow-reverse-wrong : List A → List A → Set where
  wrong : Slow-reverse-wrong xs (slow-reverse xs)
```

The data type, while it also describes the graph of slow-reverse, is not very useful. Crucially, we expect the graphs to follow the same recursive structure as the functions they represent. Provided the type indices are built from constructors and variables, this will be the case.

Many proof assistants, including Rocq and Lean, provide automation for generating a function's induction principles from its definition [Breitner 2024; Sozeau 2010; Sozeau and Mangin 2019]; generating the corresponding inductively defined graph is very similar. In Agda, we could also use the reflection mechanism to generate these definitions automatically [van der Walt and Swierstra 2012]. There is some manual overhead (and room for error) in writing these definitions by hand – but doing so is instructive at first.

Similarly, the cost functions that calculate the expected number of steps are part of the specification. Consider the following definition:

```
wrong-cost : Slow-reverse xs ys → ℕ
wrong-cost _ = 1
```

The wrong-cost function defines a cost function for our slow-reverse function, albeit not a very useful one! It assumes that every slow reverse takes constant time, which is not typically what one would expect. One could, once again, use metaprogramming or generic programming techniques to generate these cost functions to avoid this issue [Backhouse et al. 1998; Escot and Cockx 2022; Jansson and Jeuring 1997; Ko et al. 2022].

This drawback, however, is not unique to our approach: many textbooks on algorithms leave underspecified what operations actually count towards a program's execution time. Just as any other formally verified proof, the value of the proof inherently depends on the specification. Nonetheless, there are situations where having some flexibility in defining a suitable cost functions is useful – as we will see next.

2.4 Insertion sort

Consider *insertion sort*. The insert function inserts a new element into a sorted list:

```
insert : ℕ → List ℕ → List ℕ
insert y []       = [ y ]
insert y (x ∷ xs) = if y ⩽ᵇ x then y ∷ x ∷ xs else x ∷ insert y xs
```

As this function has two results when the input list is non-empty, there are two constructors in the corresponding graph:

```
data Insert (y : ℕ) : List ℕ → List ℕ → Set where
  base : Insert y [] [ y ]
  stop : y ⩽ x → Insert y (x ∷ xs) (y ∷ x ∷ xs)
  step : y > x → Insert y xs zs → Insert y (x ∷ xs) (x ∷ zs)
```

The obvious cost function is appears equally straightforward:

```
cost-insert : Insert y xs zs → ℕ
cost-insert base          = 1
cost-insert (stop x⩽y)    = 1 + cost-⩽ x⩽y
  where
  cost-⩽ : x ⩽ y → ℕ
  cost-⩽ _ = 1
cost-insert (step y>x rec) = cost-> y>x + succ (cost-insert rec)
  where
  cost-> : y > x → ℕ
  cost-> _ = 1
```

Yet we have maken an important design decision here: we have chosen to consider the cost of comparing two natural numbers to be constant. These seems to be an obvious choice – but if these natural numbers are implemented naively using Peano numerals, this cost is *not* constant: it takes longer to compare bigger numbers. Given that we expect to compile this code to (arbitrary precision) machine integers, this seems a reasonable choice. This illustrates how having some choice in defining the cost function is important and how this forms part of our specification.

With these definitions in place, it is easy to prove that the insert function is linear in the length of its input list:

```
insert-linear : (y : ℕ) → Insert y ∈O (λ n → n)
```

As a result, repeatedly calling insert leads to a quadratic time sorting function.

2.5 Logs come from trees

Our next example explores binary search trees. There are many different ways to enforce the desired invariants of our data structure: intrinsically or extrinsically? Do we require the trees to be balanced? Or are we only interested in the ordering of elements? The purpose of this example is to demonstrate how to incorporate additional invariants extrinsically in the cost analysis. As a result, we choose a simply typed definition of binary trees:

```
data Tree : Set where
  leaf : Tree
  node : Tree → ℕ → Tree → Tree
```

The find function looks up its argument key k in a given tree:

```
find : ℕ → Tree → Bool
find k leaf          = false
find k (node l x r)  = if k ≡ᵇ x then true
                       else (if k <ᵇ x then find k l else find k r)
```

What is the complexity of this find function? To answer this question, we need a notion of size for our trees, the graph of the find function, and the corresponding cost function:

```
size-tree : Tree → ℕ
size-tree leaf          = 1
size-tree (node l x r)  = 1 + size-tree l + size-tree r

data Find (k : ℕ) : Tree → Bool → Set

cost-find : Find k t b → ℕ
```

We have left out the definitions of the graph and cost function, as we hope the pattern is familiar by now. Using these definitions, we prove that the *worst case complexity* is linear in the size of the tree:

```
worst-case : (k : ℕ) → Find k ∈O (λ n → n)
```

Oftentimes, however, we do not search through an arbitrary tree, but rather one that is suitably balanced. We could add indices to our Tree type to ensure that the tree is balanced – but as we mentioned previously, we instead choose to add this assumption to the *graph* of the find function. To simplify our analysis, we consider *perfect* trees, where the left and right subtree have the exact same size:

```
Balanced : Tree → Tree → Set
Balanced l r = size-tree l ≡ size-tree r
```

It is straightforward to relax this constraint, bounding the difference in size between the two subtrees, as is the case for AVL trees [Adelson-Velskii and Landis 1962] and Braun trees [Hoogerwoord 1992a; Rem and Braun 1983], for instance.

We can add this additional assumption to the graph of the find function:

```
data Find-Balanced (k : ℕ) : Tree → Bool → Set where
  base  : Find-Balanced k leaf false
  stop  : Find-Balanced k (node l k r) true
  left  : Balanced l r → k < x → Find-Balanced k l b → Find-Balanced k (node l x r) b
  right : Balanced l r → k > x → Find-Balanced k r b → Find-Balanced k (node l x r) b

cost-find-balanced : Find-Balanced k t b → ℕ
```

Here the two possible recursive calls, corresponding to the left and right constructors, introduce an explicit assumption that the two subtrees are have equal size. Using this assumption, we can prove the expected logarithmic complexity:

```
worst-case-balanced : (k : ℕ) → Find-Balanced k ∈O (λ n → log₂ n)
```

Of course, this analysis hinges on the assumption that the tree is perfectly balanced. To prove that this graph accurately reifies the find function on perfectly balanced trees, we prove the following lemma, establishing that the revised graph of our find function accurately reflects the search through a perfect tree:

```
Perfect : Tree → Set
Perfect leaf          = ⊤
Perfect (node l _ r)  = Balanced l r × Perfect l × Perfect r

find-perfect : (k : ℕ) → (t : Tree) → Perfect t → Find-Balanced k t (find k t)
```

We could have achieved a similar result in a much simpler fashion: defining an inductive family of perfectly balanced trees. This choice would have simplified the analysis enormously – the purpose of the example is to show how to incorporate extrinsic assumptions in our cost analysis. Alternatively, we could also pass the Perfect assumption as an argument to the cost-find function, as we will do towards the end of this paper. As is often the case, there are many different ways to formulate and prove the same result.

3 Batched queues: amortized cost

All our analyses so far establish upper bounds on the *worst case* time complexity of our functions. In this section we explore a classic *amortized* analysis of a queue implementation using two lists. The code in Figure 1 is a variation of the batched queues from Okasaki's classic textbook [1998]. Each queue consists of two lists, the front and the rear. Elements are enqueued onto the rear, but removed from the front. Once the front runs out of elements, the elements in the rear are reversed to form the new front.

We enforce one invariant intrinsically: when the front of the queue is empty, so is the rear. By doing so, we know that when the deq operation encounters an empty front queue, there is no need to inspect (or reverse) the rear. Note that both the enq and deq operations are not recursive. The balance function, however, that restores our invariant, may take non-constant time, if the rear needs reversing.

To say anything about the cost of these queue operations, we define their graphs:

```
data Balance : List A × List A → Q A → Set where
  stop  : Balance (x ∷ xs , ys) (queue (x ∷ xs) ys (λ ()))
  work  : Reverse xs ys → Balance ([] , xs) (queue ys [] (const tt))
data Enq : Q A → Q A → Set where
  enq-q : Balance (front q , x ∷ rear q) q₁ → Enq q q₁
data Deq : Q A → Q A → Set where
  deq-∷ : ∀ {inv} → Balance (xs , ys) q → Deq (queue (x ∷ xs) ys inv) q
```

Note that Balance may call the (linear) list reversal function defined in the previous section. The obvious cost functions count the size of these graphs:

```
record Q (A : Set) : Set where
  constructor queue
  field
    front : List A
    rear  : List A
    inv   : So (null front) → So (null rear)
open Q

empty : Q A
empty = queue [] [] id

balance : List A × List A → Q A
balance ([]          , rear) = queue (reverse rear) [] (const tt)
balance (front @ (_ ∷ _) , rear) = queue front rear λ ()

enq : A → Q A → Q A
enq x q = balance (front q , x ∷ rear q)

deq : Q A → Maybe (A × Q A)
deq (queue [] r i)        = nothing
deq (queue (x ∷ xx) rear i) = just (x , balance (xx , rear))
```

Fig. 1. Implementing queues using two lists

```
cost-enq     : Enq q₀ q₁ → ℕ
cost-deq     : Deq q₀ q₁ → ℕ
cost-balance : Balance (xs,ys) q → ℕ
```

As balance may require reversing the rear list, the worst case time complexity of balance is linear in the length of the rear list; correspondingly, the worst case complexity of both enq and deq is linear:

```
balance-linear : Balance ∈O (λ n → n)
```

As an aside, we've omitted the Size instance for pairs of lists; in this particular example, we define the size of the pair to be the length of the second component, i.e., the length of the rear list.

3.1 Sums of costs

The key idea behind an *amortized* analysis is to find an upper bound on the total cost of a *series of operations*. If the expensive operations are sufficiently rare, these should not have a modest effect on the total cost of the series. We will reproduce the amortized analysis of batched queues in Agda.

Our analysis shifts from the cost of a single operation to series of operations. What are valid series? We restrict ourselves to two queue operations, enqueuing and dequeueing:

```
data Op (q₀ : Q A) : Q A → Set where
  enq-op : Enq q₀ q₁ → Op q₀ q₁
  deq-op : Deq q₀ q₁ → Op q₀ q₁
```

The data type Op q_0 q_1 is either an enq or deq operation, taking the old queue q_0 to a new queue q_1. Even though dequeuing may fail (when the queue is empty), the *graph* of Deq only considers the successful case – hence never have to worry about the possibility of failure, even if the deq *function* may not return a result. Next, we chain together a list of queue operations, one after the other:

```
data Ops (q₀ : Q A) : Q A → Set where
  stop : Ops q₀ q₀
  step : Op q₀ q₁ → Ops q₁ q₂ → Ops q₀ q₂
```

Note that the types ensure that each subsequent operation acts on the result of the previous one.

The total cost of such a sequence of queue operations is the sum of the cost of the individual operations:

```
cost-op : Op q₀ q₁ → ℕ
cost-op (enq-op e) = cost-enq e
cost-op (deq-op d) = cost-deq d

cost-ops : Ops q₀ q₁ → ℕ
cost-ops (step op ops) = cost-op op + cost-ops ops
cost-ops stop          = zero
```

Our aim is to find an upper bound on the total cost of a series of queue operations. It is not immediately obvious how to do so: any upper bound must predict how often rebalancing occurs and what the cost of each expensive rebalancing operation is.

3.2 Amortized cost

The key idea behind amortized complexity is due to Tarjan [1985], although we closely follow the presentation by Hoogerwoord [1992b]. Rather than reasoning about the cost of the sum of operations, we define a notion of credit:

```
credit : Q A → ℕ
```

For now, we leave underspecified what assumptions we make about this credit function. Note that credit is a function on queues – not on operations. We define the amortized cost of a single operation as its cost, together with its change in credit:

```
amortized-cost-op : Op q₀ q₁ → ℕ
amortized-cost-op {q₀} {q₁} op = cost-op op + credit q₁ ∸ credit q₀
```

The amortized cost of a series of operations simply computes the sum of amortized costs:

```
amortized-cost-ops : Ops q₀ q₁ → ℕ
amortized-cost-ops stop          = zero
amortized-cost-ops (step op ops) = amortized-cost-op op + amortized-cost-ops ops
```

Why is this definition useful? When we sum the amortized cost, adjacent credits cancel out! To make this precise, we prove the following equality:

```
amortized-property : (ops : Ops q₀ q₁) →
  credit q₀ + amortized-cost-ops ops ≡ cost-ops ops + credit q₁
```

If we choose the credit of our empty queue to be zero, then we know that the amortized-cost-ops gives an upper bound for total cost. The inductive case of the corresponding proof is in Figure 2,

```
credit q₀ + amortized-cost-ops (step op ops)
  ≡⟨ definition of amortized-cost-ops ⟩
credit q₀ + (amortized-cost-op op + amortized-cost-ops ops)
  ≡⟨ definition of amortized-cost ⟩
credit q₀ + ((cost-op op + credit q₁ ∸ credit q₀) + amortized-cost-ops ops)
  ≡⟨ associativity ⟩
(credit q₀ + (cost-op op + credit q₁ ∸ credit q₀)) + amortized-cost-ops ops
  ≡⟨ monus-addition associativity ⟩
(credit q₀ + (cost-op op + credit q₁) ∸ credit q₀) + amortized-cost-ops ops
  ≡⟨ monus-addition cancellation ⟩
(cost-op op + credit q₁) + amortized-cost-ops ops
  ≡⟨ associativity ⟩
cost-op op + (credit q₁ + amortized-cost-ops ops)
  ≡⟨ induction hypothesis ⟩
cost-op op + (cost-ops ops + credit q₂)
  ≡⟨ associativity ⟩
(cost-op op + cost-ops ops) + credit q₂
  ≡⟨ definition of cost-ops ⟩
cost-ops (step op ops) + credit q₂
```

Fig. 2. An equational proof relating amortized and worst case cost

where op : Op q_0 q_1 and ops : Ops q_1 q_2; the Agda proof terms have been replaced with a more legible description.

This proof does hinge on several lemmas. All costs are positive natural numbers. The definition of amortized cost uses a 'monus' operation, i.e., cut-off subtraction on natural numbers. Reasoning about the monus is rather subtle, as it does not satisfy the familiar properties of subtraction on integers. The proof requires that monus cancels addition and that monus and addition are associative:

```
m+n∸m≡n : ∀ m n → (m + n) ∸ m ≡ n
+-∸-assoc : ∀ m → o ⩽ n → (m + n) ∸ o ≡ m + (n ∸ o)
```

To use the second lemma, we need to ensure the amortized cost both operations is positive:

```
amortized-cost-positive : (op : Op q₀ q₁) → credit q₀ ⩽ cost-op op + credit q₁
```

This side condition guarantees that the amortized cost never 'overspends'. Each operation uses no more credits than are available; credits are never lost.

3.3 Credits and queues

All that remains to be done, is to choose the notion of credit for our batched queues. Okasaki [1998] argues that the *length of the rear* queue gives a suitable notion of credit:

```
credit : Q A → ℕ
credit q = length (rear q)
```

How do our queue operations change the credits available? Each enqueue operation takes constant time, but increases the accumulated credit as it adds an element to the rear. When rebalancing occurs, typically after a dequeue operation, we use the accumulated credits to pay for the expensive list reversal.

Why is amortization important? Instead of reasoning about the entire sequence of operations, we know that the amortized cost of the series gives an upper bound on the total cost. We therefore prove an upper bound on the amortized cost of each single operation:

```
amortized-bound : (op : Op q₀ q₁) → amortized-cost-op op ⩽ 3
```

Why three? Consider a enqueue operation that does not rebalance: 1) it takes a step to enter the body of enq; 2) it calls balance; 3) it allocates one new credit as the resulting rear list is longer. It is easy to construct a similar argument for the other queue operations. The most interesting case is where rebalancing occurs – but by construction, we will have accumulated enough credits to pay for the expensive reversal operation.

If we know that a single operation has a constant amortized cost, the overall cost of any series of operations is bounded accordingly:

```
amortized-bound-ops : (ops : Ops q₀ q₁) → amortized-cost-ops ops ⩽ 3 * | ops |
```

Using the key amortized-property from the figure above, together with the fact that the empty queue has zero credit, we show that the *total costs* are bounded:

```
amortized-constant : (ops : Ops empty q) → cost-ops ops ⩽ 3 * | ops |
```

Hence we conclude that enqueue, dequeue and rebalancing operations are all *amortized constant time operations*.

4 Real time queues: laziness and sharing

There is one important point we have glossed over in the previous section. Recall the definition of a series of operations from the previous section:

```
data Ops (q₀ : Q A) : Q A → Set where
  stop : Ops q₀ q₀
  step : Op q₀ q₁ → Ops q₁ q₂ → Ops q₀ q₂
```

Here the queue operations are applied one after the other; each new operation is applied to the result of the previous one. As a result, the expensive rebalancing operations are rare and do not dominate the overall cost. Unfortunately, not all programs that work with queues satisfy this pattern of usage. Consider the following (admittedly contrived) example:

```
f : List A → List A
f ys = let q = queue [ 0 ] [1000000, ... , 2 , 1 ] λ () in
       map (λ y → ... deq q ...) ys
```

This function constructs a queue, q, that will require more than a million steps for the next deq operation. In this example, this rebalancing happens over and over again as the map function is executed. The amortized analysis from the previous section hinges on rebalancing a queue only once – and cannot be used to limit the cost of the function f above. Here the queue q is said to be used *persistently* rather than *ephemerally*.

To address this, Okasaki gives several implementations of queues with *worst case constant bounds*, including real time queues [Okasaki 1998, §7.2]. Just like batched queues, real time queues are implemented using a pair of lists; the key differences are in their invariants and rebalancing operations. Where batched queues perform one expensive rebalance operation every so often; real

time queues perform a part of the expensive rebalancing after every queue operation: the total amount of work is the same, but the work is scheduled differently. The implementation of real time queues, however, makes essential use of *sharing* and *lazy evaluation*.

So far, we have implicitly assumed a strict evaluation order: the cost functions compute the required time to fully evaluate a function call, assuming the arguments are already fully evaluated. How should we account for thunked computations, laziness and sharing? Recently, Pottier et al. [2024] have shown how to use separation logic and the 'debit style' reasoning to prove amortized bounds for lazy data structures. Yet we will not succumb to the siren's call of separation logic – pure functions and simple induction shall suffice.

4.1 First order laziness

Many *implementations* of laziness use a mutable reference to store thunks on the heap; upon evaluation, a thunk is replaced with its value. Rather than model such mutable state in Agda, however, we employ the recent work on *first order laziness* for the Koka programming language [Lorenzen et al. 2025]. In this work, a fixed number of lazy computations may be suspended, in the form of so-called *lazy constructors*. Although Agda lacks the language support that Koka provides for lazy constructors, we will illustrate their key ideas and how to model these in Agda, before returning to the implementation of real time queues.

Consider the following data type declaration:

```
data Stream (A : Set) : Set where
  []   : Stream A
  _:_  : A → Stream A → Stream A
  app  : Stream A → Stream A → Stream A
  rev  : Stream A → Stream A → Stream A
```

Besides the familiar constructors for lists, we have included two additional constructors: app appends one stream onto another; rev reverses its first argument onto its second. To distinguish between regular and lazy constructors, we need to define which constructors we consider to be in *weak head normal form*:

```
whnf? : Stream A → Set
whnf? []          = ⊤
whnf? (_ : _)     = ⊤
whnf? (app _ _)   = ⊥
whnf? (rev _ _)   = ⊥
```

The familiar list constructors are in weak head normal form. The app and rev constructors, on the other hand, are *lazy constructors* that require further evaluation. To specify how these constructors are evaluated, we define a single reduction step on streams that are not yet in weak head normal form:

```
data _⇝_ : Stream A → Stream A → Set where
  app-⇝        : xs ⇝ xs' → app xs ys ⇝ app xs' ys
  rev-⇝        : xs ⇝ xs' → rev xs ys ⇝ rev xs' ys
  app-⇝-base   : app [] ys ⇝ ys
  app-⇝-step   : app (x : xs) ys ⇝ (x : app xs ys)
  rev-⇝-base   : rev [] ys ⇝ ys
  rev-⇝-step   : rev (x : xs) ys ⇝ rev xs (x : ys)
```

We can also define the function implementing this single step that is the identity on streams in weak head normal form:

```
step : Stream A → Stream A
```

In what should be a familiar pattern, the small step relation is the graph of the step function. Repeatedly executing a single evaluation step, yields a weak head normal form:

```
data _⇝*_ : Stream A → Stream A → Set where
  step-⇝ : xs ⇝ ys → ys ⇝* zs → xs ⇝* zs
  stop-: : (x : xs) ⇝* (x : xs)
  stop-[] : [] ⇝* []
```

We can also define the corresponding function, that we will call seq. To pattern match on the result of seq, we include an implicit proof argument that the result is indeed in weak head normal form:

```
data WHNF (A : Set) : Set where
  ⟨_⟩ : (xs : Stream A) → {whnf? xs} → WHNF A

seq : (xs : Stream A) → WHNF A
```

In this fashion, programming with streams appears quite similar to programming with lists: when pattern matching on the result of a call to seq, we never observe the lazy constructors. For example, to map over the stream (and thereby force any suspended lazy constructors), we write:

```
map : (f : A → B) → Stream A → List B
map f xs with seq xs
... | ⟨ [] ⟩     = []
... | ⟨ x : xs ⟩ = f x : map f xs
```

Here Agda uses the (implicit) proof argument to rule out the cases for app and rev; these will be forced by the call to seq.

After a call to seq, a stream may still contain (possibly nested) lazy constructors. If a stream is fully evaluated, i.e., it does not contain any lazy constructors at all, we shall refer to it as a *list*:

```
isList? : Stream A → Set
```

Finally, we define a simple cost function for our seq function: counting the number of evaluation steps necessary to produce a weak head normal form.

```
cost-steps : xs ⇝* ys → ℕ
cost-steps (step-⇝ _ steps) = 1 + cost-steps steps
cost-steps stop-:           = 1
cost-steps stop-[]          = 1
```

Using this cost function, we prove simple properties about the number of steps required to force evaluation of a lazy constructors to their weak head normal form:

```
app-cost : (steps : app xs ys ⇝* zs) → isList? xs → isList? ys →
  cost-steps steps ⩽ 2
rev-cost : (steps : rev xs ys ⇝* zs) → isList? xs → isList? ys →
  cost-steps steps ⩽ 2 + length xs
```

Unsurprisingly, evaluating an app constructor to weak head normal form requires constant time; evaluating a rev constructor is linear in its first argument (provided the arguments to both constructors are already fully evaluated).

4.2 Real time queues

Now that we have this model of laziness in place, we turn our attention to implementing real time queues. The implementation is given in Figure 3. Just as we saw for batched queues, we maintain a front and a rear stream: new elements are added to the rear; elements are removed from the front. Unlike batched queues, however, we enforce a different invariant, relating the front to a third stream: the *schedule*. What is this schedule? Okasaki writes:

> *We can think of s (the schedule) in two ways, either as a suffix of f [the front] or as a pointer to the first unevaluated suspension in f.*

The invariant that we enforce intrinsically, sched ⊑ front, in our definition of real time queues will make this statement formal: the schedule is a suffix of the front; the prefix of the front that is not in the schedule is fully evaluated. Furthermore, the schedule stream itself is either:

- a fully evaluated list. Initially, for example, the schedule is the empty list.
- or a suspended computation of the form app xs (rev ys zs) for lists xs, ys, and zs, where the length of ys is equal to the length of xs + 1.

If you look ahead to Figure 4, you can see that the rebalancing functions inspect the schedule, forcing its evaluation. By also evaluating one step of the list reversal in the call to the auxiliary function step-rev, we increase the cost of each queue operation – but doing so ensures that by the time we run out of elements in xs, the reversal of ys will be almost finished. As we never have to compute an unbounded linear time reversal operation, the worst case cost of each operation remains constant. Once the suspended app computation is finished, the remainder of the schedule is a (reversed) list that is fully evaluated – as in the first bullet point above.

Specifying these invariant effectively is one of the key challenges when it comes to reasoning about real time queues. We will return to the definition of rebalancing real time queues shortly, but first we formally define the notion of suffix – together with three other invariants that real time queues satisfy.

4.3 Contexts and suffixes

Just as there are many equivalent definitions of the 'less than or equals to' order on natural numbers, there are many ways to specify that one list is a suffix of another. We have chosen a definition that relies on the notion of *one hole contexts*:

```
data Ctx (A : Set) : Set where
  □   : Ctx A
  _:_ : A → Ctx A → Ctx A
```

A one hole context for our data type of streams is simply a list of values, ending in a hole □. We require contexts to be fully evaluated, that is, they do not contain lazy constructors. This definition generalises naturally to other data types, with or without lazy constructors: the derivative of the underlying functor gives rise to the corresponding context [Hinze et al. 2004; McBride 2001].

There are two operations on contexts that we will employ: filling in the hole with a stream; appending one context after another. Both operations are easy to define:

```
_[_] : Ctx A → Stream A → Stream A
□ [ xs ]       = xs
(x : c) [ xs ] = x : (c [ xs ])
```

```
record Q (A : Set) : Set where
  constructor queue
  field
    front  : Stream A
    rear   : Stream A
    sched  : Stream A
    inv    : sched ⊑ front

  empty : Q A
  empty = queue [] [] [] ⊑-refl

  enq : Q A → A → Q A
  enq q y = balance-enq (front q) (y : rear q) (sched q) (inv q)

  deq : Q A → Maybe (A × Q A)
  deq q with seq (front q)
  ... | ⟨ [] ⟩      = nothing
  ... | ⟨ x : xx ⟩  = just (x , balance-deq xx (rear q) (sched q) (inv q))
```

Fig. 3. Real time queues – definition and operations

```
_ + _ : Ctx A → Ctx A → Ctx A
□       + c₂ = c₂
(x : c₁) + c₂ = x : (c₁ + c₂)
```

Minamide [1998] has proposed that these operations on contexts may be made efficient by using an additional pointer that gives immediate access to the end of the context. The Koka programming language implements this idea [Lorenzen et al. 2024] – hidden behind a purely functional interface – but guarantees that both operations are executed by means of constant time pointer manipulation.

Using this definition of context, we define the suffix relation between streams as follows:

```
data _⊑_ (xs : Stream A) : Stream A → Set where
  suffix : (c : Ctx A) → xs ⊑ c [ xs ]
```

In words, xs ⊑ ys holds precisely when we can find a context c such that ys is equal to c [xs]. It is straightforward to prove that this relation is both transitive and reflexive. With this definition, we enforce that the prefix of the front is fully evaluated: the prefix is described by a context without lazy constructors. Furthermore, as we shall see, this definition readily captures our intuition behind the sharing of evaluation: if the context c witnesses xs ⊑ ys and xs ⇝ xs' then c [xs] becomes c [xs'].

4.4 Invariants and rebalancing

To complete our definition of real time queues, we need to define the balance functions that restores the property that the schedule is suffix of the front stream – but also forces enough evaluation to ensure that enq and deq remain constant time. To understand how the balance functions work, however, we need to identify several additional invariants that real time queues satisfy. The code

```
balance-enq : (front rear sched : Stream A) → (sched ⊑ front) → Q A
balance-enq .(c [ sched ]) rear sched (suffix c) with seq sched
... | ⟨ [] ⟩                  = let f = app (c [ [] ]) (rev rear [])
                               in queue f [] f ⊑-refl
... | ⟨ x : xx ⟩               = let sched' = step-rev xx in
                               let c' = c + (x : □) in
                               queue (c' [ sched' ]) rear sched' (suffix c')
step-rev : Stream A → Stream A
step-rev (app xs (rev ys zs)) = app xs (step (rev ys zs))
step-rev xs                   = xs
```

Fig. 4. Balancing real time queues

to rebalance after an enq operation is given in Figure 4. Before studying this further, however, we identify the invariants that we will prove *extrinsically* about our real time queues.

Invariants. We call a property on queues an *invariant* when it holds for the empty queue and is preserved by each queue operation:

```
Invariant : (P : Q A → Set) → Set
Invariant P = P empty × (∀ q q' → Op q q' → P q → P q')
```

The real time queues satisfy three invariants that we now cover one by one.

Our first simple invariant states that the rear of the queue is always a fully evaluated list:

```
RearInv : Q A → Set
RearInv q = isList? (rear q)
```

This is obviously true for the empty queue. The enq operation extends the rear list, but does not introduce lazy constructors; the balancing and dequeue operations do not change the rear list. It is straightforward to prove that this invariant holds.

A second invariant relates the lengths of the streams that constitute our queues. Okasaki suggests the following invariant:

```
LengthInv : Q A → Set
LengthInv q = length (rear q) + length (sched q) ≡ length (front q)
```

In our implementation, however, we are a bit more precise. The intrinsic invariant, stating that sched ⊑ front, already tells us something about the length of the front queue, namely it must be equal to the length of the evaluated prefix plus the length of the schedule. A more precies invariant therefore removes the length of the schedule from both sides of this equality:

```
LengthInv : Q A → Set
LengthInv q = length (rear q) ≡ length-ctx (inv q)

length-ctx : Ctx A → ℕ
length-ctx c = length (c [ [] ])
```

Put differently: the evaluated prefix of the front queue has the same length as the rear. Why does this hold? It is clearly true of the empty queue. After each enq operation, the rear is too long – but evaluating the schedule, as done by the call to seq in the balance-enq function, extends the evaluated prefix. After each successful deq operation, the front is too short – but again, evaluating the schedule extends the evaluated prefix, reestablishing the desired invariant.

Both these properties, however, are only ever used to prove the key invariant that we need to establish the constant time behaviour of real time queues. We previously claimed that the schedule was always fully evaluated or of the form app xs (rev ys zs). To formalise this in Agda, we introduce the following predicate, characterising valid schedules:

```
data Schedule : Stream A → Set where
  done  : (isList? xs) → Schedule xs
  thunk : (xs ys zs : Stream A) → isList? xs → isList? ys → isList? zs) →
          succ (length xs) ≡ length ys → Schedule (app xs (rev ys zs))
```

This property is the key to proving that all queue operations are worst case constant time.

Rebalancing. With the specification of the key invariants out of the way, we finally return to our rebalancing operations. After each deq or enq operation, the length invariant is broken: either the front is too short or the rear is too long. To restore this – and the other invariants – there are two rebalancing functions, one of which is given in Figure 4, that we explain now in more detail.

First of all, note that pattern matching on the proof that sched ⊑ front tells us *exactly* what the *value* of the front stream is, namely c [sched]. The pattern we see for the front queue is *forced*. As the body of the rebalance function evaluates the schedule to weak head normal form, as done by the call seq sched, we need to ensure this evaluation is also shared with the front queue. We distinguish two possible cases, one for each possible weak head normal form of the schedule:

- If the schedule is empty, we create a new suspension appending the evaluated prefix of the front to the reversed rear. As a result of the enqueue operation, the length invariant no longer holds: the rear is one longer than the front – but this is exactly what is necessary to ensure the schedule is well-formed: the thunk constructor reestablishes the schedule invariant. Of course, we still need to check that all the streams involved (c [[]], rear and []) are fully evaluated – but the only interesting case is for the rear stream, which we proved separately in the RearInv above.
- If the schedule is non-empty, its head is accumulated in the evaluated prefix of the front, extending the context – thereby reestablishing the desired length invariant relating this prefix to the length of the rear. Furthermore, if the schedule contains a suspended rev constructor, one step of this reversal is also executed. This ensures that the new schedule continues to satisfy the Schedule predicate above – where the first argument to app is only one shorter than the first argument to rev.

While we not gone into detail how to complete the formal proofs that these invariants hold, these are largely straightforward. Most of the proofs need to distinguish the two different cases for rebalancing, keeping track of where evaluation occurs and is shared – but these require very few auxiliary definitions or lemmata.

What about rebalancing after an deq operation? The function has almost exactly the same behaviour. To understand why there are two different balancing functions, it is important to understand how enq and deq break the suffix invariant in different ways. After a *dequeue* operation, the front stream has become too short: we need to take the tail of *both* the front and schedule. We therefore drop the first element of the evaluated prefix of the front:

```
tail : Ctx A → Ctx A
tail □       = □
tail (x : c) = c
```

Besides this modest change, the rebalancing function is identical; the proofs reestablishing the invariants are no more complicated. The complete code for rebalancing can be found in Appendix A.

Note that the presentation of real time queues is slightly different from the one given by Okasaki [1998]. The original presentation uses one lazy operation, rotate, that takes three arguments – but behaves identically to the combination of append and reverse used here. One advantage of our definition is that it potentially offers more memory reuse [Lorenzen et al. 2023]. As the lazy constructors are evaluated, the corresponding memory locations may be reused, rather than allocating fresh cons cells.

4.5 Cost analysis

To prove the worst case constant time bounds for queue operations, we rely on one key insight: it takes a constant amount of time to force a valid schedule to weak head normal form:

```
schedule-constant : ∃[ c ] (Schedule s → (steps : s ⇝* s') → cost-steps steps ⩽ c)
```

As a result, forcing the front queue (as is done by the deq operation), also takes constant time. Furthermore, both rebalancing functions are not recursive – but do require forcing the schedule.

There is one rather subtle point we want to discuss explicitly: our pure specification restores the invariant, sched ⊑ front, by extending the context after the schedule has been evaluated to weak head normal form. The cost functions consider this operation to have zero cost. We make the assumption that the let binding, introduced when rebalancing encounters an empty schedule, creates a *shared* thunk for the front and schedule. Any subsequent evaluation of the schedule is automatically shared with the front. As a result, the cost of 'updating' the front after evaluating the schedule is zero; this update is usually left implicit – as the front and the schedule share memory locations – in traditional semantics of lazy evaluation with an explicit heap [Launchbury 1993]. For the moment, however, we defer a more thorough discussion about suitable *cost models* to the discussion in the next section.

Using the lemma above, we prove that each operation takes at most constant time:

```
real-time-constant : ∃[ c ] (Schedule (sched q) → (op : Op q q') → cost-op op ⩽ c)
```

The proof itself is rather dull – the challenge is identifying and establishing the invariants that make it so.

This cost analysis does have one interesting aspect: we establish a constant time upper bound on queues *provided the schedule is valid*. Without this assumption, evaluating the schedule to weak head normal form could take an arbitrary amount of time, for instance, when the schedule contains many nested rev and app constructors. We justify this assumption by proving this invariant holds for every queue arises from a sequence of enq and deq operations on the empty queue. Any library should therefore take care to provide only these operations as the queue interface – or this analysis will become unsound.

5 Discussion

Proof assistants. There is a great deal of related work in the intersection of cost analysis, functional languages, and proof assistants. The recent textbook by Nipkow [2025] gives an overview of purely functional data structures and algorithms, together with their implementation, specification and verification using the interactive proof assistant Isabelle. This book not only gives the precise specification and correctness proofs, but also covers the worst case and amortized complexity

analysis. Rather than reify the graph of a function explicitly and define the cost functions inductively, the authors use a metaprogram to generate the cost functions from a function definition directly. Nipkow, however, avoids reasoning about lazy evaluation as 'reasoning about running time under lazy evaluation is nontrivial' [2025, page 248]. Yet Okasaki writes: 'functional languages have the curious property that *all* data structures are automatically persistent' and 'lazy evaluation can mediate the conflict between amortization and persistence'. Laziness, despite its apparent complexity, is crucial for the efficient implementation of many purely functional data structures.

In spirit, our work is most closely related to the Bove-Capretta method [2005]. Bove and Capretta have shown how to reify a function's call graph to decouple its termination proof from its definition; any function can be made structurally recursive in this fashion. In our work, we reify the entire graph of a function to define a suitable cost function. Yet there are some differences. As our analysis of the slow reverse function shows, it is important to have access to *all* auxiliary function calls – such as the calls to append – and not just recursive structure of the function definition under inspection. Furthermore, by including the function's result in the reified data type, we check that the corresponding data type is sound and complete with respect to its function definition. Where both the Bove-Capretta predicates and function graphs can be generated automatically from a function's definition, our cost analyses do not require modification to the function's original definition.

There are several related approaches to incorporating the cost analysis of a function together with its definition in a proof assistant. Danielsson [2008] has shown how track the necessary evaluation steps for lazy programs in a function's type signature in Agda. More recently, the work on cost-aware logical frameworks is also accompanied by an Agda implementation [Grodin et al. 2024; Niu et al. 2022]. Provided users write their function in these libraries, they provide an accurate upper bound on the function's cost. There is a trade off here: using the lightweight techniques suggested in this paper, cost analysis is decoupled from a function's definition and can be performed *post hoc*, just as other forms of verification. Reasoning about complexity is not always straightforward: the analysis of real time queues hinges on a complex invariant (the validity of the schedule) and requires a non-trivial argument establish the constant time bounds – these reasoning steps are hard to infer automatically. On the other hand, the approach taken in this paper requires more manual work (and meta-theoretic justification) that these frameworks provide out of the box. Further afield, there are numerous type and effect systems that track a functional program's cost or resource usage [Hoffmann et al. 2012; Hoffmann and Hofmann 2010; Hoffmann and Jost 2022; Jost et al. 2010; Rajani et al. 2021].

Cost models. What makes a good cost function? We have side stepped the meta-theoretic discussion up until this point – arguing that the graphs and cost functions are part of the specification of the cost analysis. To substantiate any claim, however, these cost functions must respect a *cost model*, relating the cost function to the program's semantics. Fortunately, there is decades of research on this topic [Danner et al. 2015; Gurr 1990; Le Métayer 1988; Rosendahl 1989; Sands 1995; Shultis 1984; Van Stone 2003; Wegbreit 1975]. For the first order data structures and functions that we have covered in the first half of this paper, counting function calls – and thereby the size of the graph of a function – suffices [Sands 1990]. The syntactic restriction, requiring all type indices in the function's graph to be constructors or variables, ensures that all costs of intermediate results are taken into account. Variable access, memory allocation, garbage collection, cache locality, and timing of primitive functions all matter for performance – but do not affect the complexity analysis overall.

What about laziness? Can we substantiate the claim that real time queues can be implemented using sharing in the style suggested here? One way to do so would rewrite the real time queues given here with an explicit heap, tracking thunks and aliasing. There is already existing work on

formalizing constructor contexts in Iris [Jung et al. 2018] by Lorenzen et al. [2024]. Constructing such a cost model is still necessary to justify the validity of the cost function used in our analysis, but for a pearl such as this, however, we appeal to the reader's intuition: intrinsically enforcing sharing guarantees an efficient implementation using aliasing is possible.

Amortized analysis and lazy evaluation. Where we have argued above that laziness is essential for efficient purely functional data structures, we have only considered the *worst case* analyses and not *amortized* analyses. Okasaki [1998] has shown how to use debit based reasoning can be used to establish amortized bounds on persistent data structures. The amortized analyses of lazy data structures is notoriously difficult. The natural semantics of laziness given by Launchbury [1993] use a heap to store thunks. Using separation logic, recent work has shown how to record additional information in the ghost state associated with a program [Mével et al. 2019; Pilkiewicz and Pottier 2011; Pottier et al. 2024], making it possible to formalize the informal arguments given by Okasaki.

Alternative semantics for lazy evaluation may make amortized analyses easier. Bjerner and Holmström [1989] give an untyped demand-driven analysis of first order lazy functional programs. This has formed the basis of recent work, formalized in Rocq, on the cost of evaluating lazy functional programs, including an amortized analysis of lazy queues [Xia et al. 2024]. Modelling lazy evaluation in Rocq, however, requires a careful treatment of approximations and inserting explicit 'ticks' to count evaluation steps. Alternatively, the recent work on clairovoyant semantics [Hackett and Hutton 2019] shows how call by need can be viewed as a form of non-deterministic call by value. This work, in turn, inspired a 'memorist semantics' [Li et al. 2026] that tracks both the cost and usage information by means of an explicit monad.

Lorenzen [2025] has developed a testing library for checking amortized bounds of lazy data structures. Their work suggests using *credit passing style*, where credits are only ever placed on lazy constructors, may still be sound, even for data structures that are used persistently. The key property of amortized analysis in Figure 2 establishes an *equality*; the associated side condition guarantees that operations never overspend. In the persistent setting, however, we may want to adapt this argument: proving an *inequality* instead and allowing credits to be discarded, for instance, when a shared thunk has been evaluated further than you expect. Proving the soundness of such an amortized argument would offer a new avenue towards reasoning about lazy data structures that are used persistently.

Conclusion. In a purely functional language, reasoning relies on elementary methods: functions are defined by structural induction; proofs follow suit. This paper explores how simple induction – albeit on the function's graph rather than its inputs – offers a convenient way to reason about a function's computational complexity in a proof assistant with dependent types. In this style, we have written worst case analyses, amortized analyses of ephemeral data structures, and worst case analyses of persistent data structures, offering a practical approach to machine-checked complexity analyses all within a purely functional setting.

References

M Adelson-Velskii and Evgenii Mikhailovich Landis. 1962. *An algorithm for the organization of information.* Technical Report.

Andrew W. Appel. 2026. *Verified Functional Algorithms.* Software Foundations, Vol. 3. Electronic textbook. Version 1.6.0, http://softwarefoundations.cis.upenn.edu.

Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. 1998. Generic programming. In *International School on Advanced Functional Programming*. Springer, 28–115.

Bror Bjerner and Sören Holmström. 1989. A composition approach to time analysis of first order lazy functional programs. In *Proceedings of the fourth International Conference on Functional Programming Languages and Computer Architecture*. 157–165.

Ana Bove and Venanzio Capretta. 2005. Modelling general recursion in type theory. *Mathematical Structures in Computer Science* 15, 4 (2005), 671–708.

Joachim Breitner. 2024. Functional Induction theorems. (2024). Published online at https://lean-lang.org/blog/2024-5-17-functional-induction.

Nils Anders Danielsson. 2008. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '08)*. 133–144. https://doi.org/10.1145/1328438.1328457

Norman Danner, Daniel R Licata, and Ramyaa Ramyaa. 2015. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 140–151.

Dominique Devriese and Frank Piessens. 2011. On the bright side of type classes: instance arguments in Agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. 143–155. https://doi.org/10.1145/2034773.2034796

Lucas Escot and Jesper Cockx. 2022. Practical generic programming over a universe of native datatypes. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 625–649.

Harrison Grodin, Yue Niu, Jonathan Sterling, and Robert Harper. 2024. Decalf: A directed, effectful cost-aware logical framework. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 273–301.

Armaël Guéneau. 2019. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs*. Ph.D. Dissertation. Université de Paris.

Douglas J Gurr. 1990. *Semantic frameworks for complexity*. Ph.D. Dissertation. The University of Edinburgh.

Jennifer Hackett and Graham Hutton. 2019. Call-by-need is clairvoyant call-by-value. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–23.

Ralf Hinze, Johan Jeuring, and Andres Löh. 2004. Type-indexed data types. *Science of Computer Programming* 51, 1-2 (2004), 117–151.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 34, 3 (2012), 1–62.

Jan Hoffmann and Martin Hofmann. 2010. Amortized resource analysis with polynomial potential: A static inference of polynomial bounds for functional programs. In *European Symposium on Programming*. Springer, 287–306.

Jan Hoffmann and Steffen Jost. 2022. Two decades of automatic amortized resource analysis. *Mathematical Structures in Computer Science* 32, 6 (2022), 729–759.

Rob R Hoogerwoord. 1992a. A logarithmic implementation of flexible arrays. In *International Conference on Mathematics of Program Construction*. Springer, 191–207.

Rob R Hoogerwoord. 1992b. A symmetric set of efficient list operations (Functional pearl). *Journal of Functional Programming* 2, 4 (1992), 505–513.

Patrik Jansson and Johan Jeuring. 1997. PolyP—a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 470–482.

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 223–236.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.

Hsiang-Shang Ko, Liang-Ting Chen, and Tzu-Chi Lin. 2022. Datatype-generic programming meets elaborator reflection. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 225–253.

John Launchbury. 1993. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 144–154.

Daniel Le Métayer. 1988. ACE: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10, 2 (1988), 248–266.

Xing Li, Yao Li, Peter Schachte, and Christine Rizkallah. 2026. The Memorist Tale: Every Thunk Every Cost All At Once. (2026). Accepted for publication at ESOP.

Anton Lorenzen. 2025. Lightweight Testing of Persistent Amortized Time Complexity in the Credit Monad. In *Proceedings of the 18th ACM SIGPLAN International Haskell Symposium (Haskell '25)*. Association for Computing Machinery, New York, NY, USA, 80–93. https://doi.org/10.1145/3759164.3759351

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP^2: Fully in-Place Functional Programming. *Proceedings of the ACM on Programming Languages* 7, ICFP (2023), 275–304.

Anton Lorenzen, Daan Leijen, Wouter Swierstra, and Sam Lindley. 2024. The Functional Essence of Imperative Binary Search Trees. *Proceedings of the ACM on Programming Languages* PLDI (2024). https://doi.org/10.1145/3656398

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

Conor McBride. 2001. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript* (2001), 74–88.

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time credits and time receipts in Iris. In *European Symposium on Programming*. Springer, 3–29.

Yasuhiko Minamide. 1998. A functional representation of data structures with a hole. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 75–84.

Tobias Nipkow. 2025. *Functional Data Structures and Algorithms: A Proof Assistant Approach.* ACM.

Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022. A cost-aware logical framework. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–31.

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

Alexandre Pilkiewicz and François Pottier. 2011. The essence of monotonic state. In *Proceedings of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. 73–86.

François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. 2024. Thunks and Debits in Separation Logic with Time Credits. *Proc. ACM Program. Lang.* 8, POPL (2024). https://doi.org/10.1145/3632892

Vineet Rajani, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2021. A unifying type-theory for higher-order (amortized) cost analysis. *Proc. ACM Program. Lang.* 5, POPL (2021). https://doi.org/10.1145/3434308

Martin Rem and Wim Braun. 1983. *A logarithmic implementation of flexible arrays.* Technical Report Memorandum MR83/4. Eindhoven University of Technology.

Mads Rosendahl. 1989. Automatic complexity analysis. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. 144–156.

David Sands. 1990. *Calculi for time analysis of functional programs.* Ph.D. Dissertation. Imperial College London, UK.

David Sands. 1995. A naive time analysis and its theory of cost equivalence. *Journal of Logic and Computation* 5, 4 (1995), 495–541.

Jon Shultis. 1984. *Complexity of higher-order programs.* Technical Report. Colorado Univ., Boulder (USA). Dept. of Computer Science.

Matthieu Sozeau. 2010. Equations: A dependent pattern-matching compiler. In *International Conference on Interactive Theorem Proving*. Springer, 419–434.

Matthieu Sozeau and Cyprien Mangin. 2019. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–29.

Robert Endre Tarjan. 1985. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods* 6, 2 (1985), 306–318.

Paul van der Walt and Wouter Swierstra. 2012. Engineering Proof by Reflection in Agda. In *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers (Lecture Notes in Computer Science)*, Ralf Hinze (Ed.), Vol. 8241. Springer, 157–173. https://doi.org/10.1007/978-3-642-41582-1_10

Kathryn Van Stone. 2003. *A denotational approach to measuring complexity in functional programs.* Ph.D. Dissertation. Carnegie Mellon.

Ben Wegbreit. 1975. Mechanical program analysis. *Commun. ACM* 18, 9 (1975), 528–539.

Li-yao Xia, Laura Israel, Maite Kramarz, Nicholas Coltharp, Koen Claessen, Stephanie Weirich, and Yao Li. 2024. Story of Your Lazy Function's Life: A Bidirectional Demand Semantics for Mechanized Cost Analysis of Lazy Programs. *Proceedings of the ACM on Programming Languages* 8, ICFP (2024), 30–63.

A Implementation of real time queues

```
record Q (A : Set) : Set where
  constructor queue
  field
    front : Stream A
    rear  : Stream A
    sched : Stream A
    inv   : sched ⊑ front

open Q

empty : Q A
empty = queue [] [] [] ⊑-refl

balance-enq : (front rear sched : Stream A) → (sched ⊑ front) → Q A
balance-enq .(c [ sched ]) rear sched (suffix c) with seq sched
... | ⟨ [] ⟩     = let f = app (c [ [] ]) (rev rear [])
                   in queue f [] f ⊑-refl
... | ⟨ x ∷ xx ⟩ = let sched' = step-rev xx in
                   let c' = c + (x ∷ □) in
                   queue (c' [ sched' ]) rear sched' (suffix c')

tail : Ctx A → Ctx A
tail □      = □
tail (x ∷ c) = c

balance-deq : (xx front rear sched : Stream A) → (sched ⊑ front) → Q A
balance-deq xx .(c [ sched ]) rear sched (suffix c) with seq sched
... | ⟨ [] ⟩     = let f = app xx (rev rear [])
                   in queue f [] f ⊑-refl
... | ⟨ x ∷ xx ⟩ = let sched' = step-rev xx in
                   let c' = tail (c + (x ∷ □)) in
                   queue (c' [ sched' ]) rear sched' (suffix c')

enq : Q A → A → Q A
enq q y = balance-enq (front q) (y ∷ rear q) (sched q) (inv q)

deq : Q A → Maybe (A × Q A)
deq q with seq (front q)
... | ⟨ [] ⟩     = nothing
... | ⟨ x ∷ xx ⟩ = just (x , balance-deq xx (front q) (rear q) (sched q) (inv q))
```