Cost analysis & first order laziness

Wouter Swierstra

Utrecht University

Functional programming 101

```haskell
module Slow where

reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Functional programming 101

```haskell
module Fast

reverseAcc :: [a] -> [a] -> [a]
reverseAcc acc []     = acc
reverseAcc acc (x:xs) = reverseAcc (x:acc) xs

reverse :: [a] -> [a]
reverse xs = reverseAcc []
```

Proofs

How are the two reverse functions related?

Proofs

How are the two reverse functions related?

Using QuickCheck or other property based testing tools:

```
checkReverse :: [Int] -> [Int] -> Bool
checkReverse xs ys = Fast.reverseAcc xs ys == Slow.reverse xs ++ ys
```

Proofs

How are the two reverse functions related?

Using QuickCheck or other property based testing tools:

```
checkReverse :: [Int] -> [Int] -> Bool
checkReverse xs ys = Fast.reverseAcc xs ys == Slow.reverse xs ++ ys
```

Or using pen and paper proofs, we perform structural induction on xs.

Proofs

How are the two reverse functions related?

Using QuickCheck or other property based testing tools:

```
checkReverse :: [Int] -> [Int] -> Bool
checkReverse xs ys = Fast.reverseAcc xs ys == Slow.reverse xs ++ ys
```

Or using pen and paper proofs, we perform structural induction on xs.

Students can mechanise such proofs in Agda, Lean or Rocq – once they learn about proof assistants.

Complexity

But *why* is `Fast.reverse` fast? And why is `Slow.reverse` slow?

How do you *prove* that one is faster than the other?

Complexity

But *why* is `Fast.reverse` fast? And why is `Slow.reverse` slow?

How do you *prove* that one is faster than the other?

And how do you *mechanise* these proofs?

Problem with proof assistants...

If we use a proof assistant to prove that `∀ xs → Slow.reverse xs ≡ Fast.reverse xs`.

Problem with proof assistants...

If we use a proof assistant to prove that `∀ xs → Slow.reverse xs ≡ Fast.reverse xs`.

And our proof assistant is consistent with or supports function extensionality...

Problem with proof assistants…

If we use a proof assistant to prove that `∀ xs → Slow.reverse xs ≡ Fast.reverse xs`.

And our proof assistant is consistent with or supports function extensionality…

Both definitions of reverse are interchangeable – there is no way to distinguish them.

Problem with proof assistants…

If we use a proof assistant to prove that `∀ xs → Slow.reverse xs ≡ Fast.reverse xs`.

And our proof assistant is consistent with or supports function extensionality…

Both definitions of reverse are interchangeable – there is no way to distinguish them.

We are (apparently) stuck.

Induction on what?

We want to inspect the reverse functions' definitions...

But what should we do induction over?

Induction on what?

We want to inspect the reverse functions' definitions...

But what should we do induction over?

The *graph* of the function.

Induction on what?

We want to inspect the reverse functions' definitions...

But what should we do induction over?

The *graph* of the function.

The *graph* of a function $f : A \to B$ is a relation on $A \times B$ given by:

$$\{(x, f(x)) : x \in A\}$$

(Think 'plotting a graph' rather than vertices and edges.)

Code is data

The graph of the accumulating reverse function becomes:

```
data RevAcc : List A → List A → List A → Set where
  base : RevAcc [] acc acc
  step : RevAcc xs (x ∷ acc) ys → RevAcc (x ∷ xs) acc ys
```

You can also read this as:

- a Prolog program for reversing a list;
- or as (the paths through) the call graph of the accumulating reverse function.

Code is data

The graph of the accumulating reverse function becomes:

```
data RevAcc : List A → List A → List A → Set where
  base : RevAcc [] acc acc
  step : RevAcc xs (x ∷ acc) ys → RevAcc (x ∷ xs) acc ys
```

You can also read this as:

- a Prolog program for reversing a list;
- or as (the paths through) the call graph of the accumulating reverse function.

It is easy to prove this definition has at least some good properties:

```
sound    : RevAcc xs acc ys → reverse-acc xs acc ≡ ys
complete : reverse-acc xs acc ≡ ys → RevAcc xs acc ys
```

Cost functions

We define the associated *cost function* by computing the size of the graph:

```
cost-reverse-acc : RevAcc xs acc ys → ℕ
cost-reverse-acc base        = 1
cost-reverse-acc (step rev)  = 1 + cost-reverse-acc rev
```

Cost functions

We define the associated *cost function* by computing the size of the graph:

```
cost-reverse-acc : RevAcc xs acc ys → ℕ
cost-reverse-acc base        = 1
cost-reverse-acc (step rev)  = 1 + cost-reverse-acc rev
```

And then we can prove:

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

Cost models - Dave Sands's PhD thesis

For this language we can build such cost-functions as follows: For each equation of the form

$$f_i(x_1, \ldots, x_{n_i}) = e_i$$

we construct an equation which computes the cost (in terms of the number of non-primitive function calls) of applying f_i to a tuple of values. The cost equation is defined as:

$$cf_i(x_1, \ldots, x_{n_i}) = 1 + \mathcal{T}[\![e_i]\!]$$

where $\mathcal{T}$ is a syntax-directed mapping defined in figure 2.4.

$$\begin{array}{rcl}
\mathcal{T}[\![c]\!] & = & 0 \\
\mathcal{T}[\![x]\!] & = & 0 \\
\mathcal{T}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] & = & \mathcal{T}[\![e_1]\!] + \texttt{if } e_1 \texttt{ then } \mathcal{T}[\![e_2]\!] \texttt{ else } \mathcal{T}[\![e_3]\!] \\
\mathcal{T}[\![p(e_1, \ldots, e_n)]\!] & = & \mathcal{T}[\![e_1]\!] + \cdots + \mathcal{T}[\![e_n]\!] \\
\mathcal{T}[\![f_i(e_1, \ldots, e_n)]\!] & = & cf_i(e_1, \ldots, e_n) + \mathcal{T}[\![e_1]\!] + \cdots + \mathcal{T}[\![e_n]\!]
\end{array}$$

Figure 2.4: Cost-Function Construction Map

Slow reverse

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

cost-append : Append xs ys → ℕ

cost-slow-rev : SlowRev xs ys → ℕ
cost-slow-rev base            = 1
cost-slow-rev (step rev app)  = 1 + cost-slow-rev rev + cost-append app
```

Big O

Can we prove that slow reverse is quadratic?

```
_∈O_ : (G : A → B → Set) → (f : ℕ → ℕ) → Set
```

We say that `G ∈O f` when:

- there are constants c_0 and c such that,
- for any input `x : A` of size `n`
- the cost function of the associated graph `g : G x y` is at most $c_0 + c * f(n)$.

Big O

Can we prove that slow reverse is quadratic?

```
_∈O_ : (G : A → B → Set) → (f : ℕ → ℕ) → Set
```

We say that `G ∈O f` when:

- there are constants c_0 and c such that,
- for any input `x : A` of size `n`
- the cost function of the associated graph `g : G x y` is at most $c_0 + c * f(n)$.

With this notation, we can prove that `SlowRev ∈O (λ n → n * n)` and `FastRev ∈O (λ n → n)`

Caveats

```
data SlowRev : List A → List A → Set where
  oops : SlowRev xs (slow-rev xs)
```

We should take care that the indices are variables and constructors.

Caveats

```
data SlowRev : List A → List A → Set where
  oops : SlowRev xs (slow-rev xs)
```

We should take care that the indices are variables and constructors.

```
cost-slow-rev : SlowRev xs ys → ℕ
cost-slow-rev _ = 0
```

We should (usually) count size of the complete call graph.

But there are exceptions…

Both the generation of graphs and cost functions may be automated with a bit of metaprogramming and/or generic programming.

What else?

Once you have the idea you can show:

- insertion sort is quadratic;
- worst case lookup time in a binary search tree is O(n)…
- …but logarithmic when the tree is balanced.
- (insert your favourite algorithm here)

What else?

Once you have the idea you can show:

- insertion sort is quadratic;
- worst case lookup time in a binary search tree is O(n)...
- ...but logarithmic when the tree is balanced.
- (insert your favourite algorithm here)

But instead - let's look at some more interesting variations:

- *Amortized complexity*
- Worst case complexity of *lazy* data structures

Batched queues

```
record Q (A : Set) : Set where
  constructor queue
  field
    front  : List A
    rear   : List A
    inv    : Empty? front → Empty? rear

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))
```

Balance

```
balance : List A × List A → Q A
balance ([]              , rear)  = queue (reverse rear) []
balance (front @(_ :: _) , rear)  = queue front rear
```

Where `enq` and `deq` themselves do a constant amount of work, but both call `balance`...

... and balancing may be linear in the length of the rear list in the worst case.

Balance

```
balance : List A × List A → Q A
balance ([]              , rear)  = queue (reverse rear) []
balance (front @(_ :: _) , rear)  = queue front rear
```

Where `enq` and `deq` themselves do a constant amount of work, but both call `balance`…

… and balancing may be linear in the length of the rear list in the worst case.

Arguably such expensive operations are rare; ‘usually’ balance takes constant time.

Can we make this more precise?

Towards amortized cost

The idea behind *amortized cost* is to find an upper bound on a *series* of operations.

```
data Ops (q₀ : Q A) : Q A → Set where
  []   : Ops q₀ q₀
  _::_ : Op q₀ q₁ → Ops q₁ q₂ → Ops q₀ q₂
```

where each operation is an enqueue or dequeue (with the associated graphs):

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

Towards amortized cost

The idea behind *amortized cost* is to find an upper bound on a *series* of operations.

```
data Ops (q₀ : Q A) : Q A → Set where
  []   : Ops q₀ q₀
  _::_ : Op q₀ q₁ → Ops q₁ q₂ → Ops q₀ q₂
```

where each operation is an enqueue or dequeue (with the associated graphs):

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

We sum the cost of individual operations:

```
cost-ops : Ops q₀ q₁ → ℕ
```

Upper bound

How can we find an upper bound on `cost-ops`? We would need to predict when rebalancing happens (and at what cost)....

Upper bound

How can we find an upper bound on `cost-ops`? We would need to predict when rebalancing happens (and at what cost)....

Idea: define a function that computes some notion of *credit* for each queue:

```
credit : Q A → ℕ
```

and consider the *amortized* cost as the real cost + change in credit:

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

Why?

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

When we compute the sum of amortized costs *adjacent credits cancel out*!

Example

$$(\text{cost-op } op_0 + \text{credit } q_1 \dot{-} \text{credit } q_0)$$
$$+ (\text{cost-op } op_1 + \text{credit } q_2 \dot{-} \text{credit } q_1)$$
$$\dots$$
$$+ (\text{cost-op } op_n + \text{credit } q_n \dot{-} \text{credit } q_{(n-1)})$$

Example

```
  (cost-op op₀ + credit q₁ ∸ credit q₀)
  + (cost-op op₁ + credit q₂ ∸ credit q₁)
  ...
  + (cost-op opₙ + credit qₙ ∸ credit q₍ₙ₋₁₎)
```

More formally, we prove:

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

Example

```
  (cost-op op₀ + credit q₁ ∸ credit q₀)
  + (cost-op op₁ + credit q₂ ∸ credit q₁)
  ...
  + (cost-op opₙ + credit qₙ ∸ credit q₍ₙ₋₁₎)
```

More formally, we prove:

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

If the credit of the empty queue is zero, the amortized costs form an *upper bound* on the total cost.

Example

```
  (cost-op op₀ + credit q₁ ∸ credit q₀)
  + (cost-op op₁ + credit q₂ ∸ credit q₁)
  ...
  + (cost-op opₙ + credit qₙ ∸ credit q₍ₙ₋₁₎)
```

More formally, we prove:

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

If the credit of the empty queue is zero, the amortized costs form an *upper bound* on the total cost.

Proof?

```
credit q₀ + amortized-cost-ops (op ∷ ops)
  ≡⟨ {- definitions -} ⟩
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-ops (op ∷ ops) + credit q₂
```

Back to queues

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

Back to queues

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

We prove that each operation takes *amortized constant time*:

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

Back to queues

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

We prove that each operation takes *amortized constant time*:

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

Hence the total amortized time required for a series of operations is bounded:

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

But...

Such an amortized argument proves that expensive rebalancing operations are rare...

But...

Such an amortized argument proves that expensive rebalancing operations are rare...

But what about the following example?

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

But...

Such an amortized argument proves that expensive rebalancing operations are rare...

But what about the following example?

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

We require the series of operations to have a certain structure: the operations line up one after another – yet this is not generally the case!

Worst case constant time queues

Okasaki presents several data structures for *worst case constant time* queues.

Idea:

- Implement a queue using two lists;
- Ensure each rebalancing operation does some *constant* amount of work...
- ...but guarantee that rebalancing never requires *unbounded* work.

Worst case constant time queues

Okasaki presents several data structures for *worst case constant time* queues.

Idea:

- Implement a queue using two lists;
- Ensure each rebalancing operation does some *constant* amount of work…
- …but guarantee that rebalancing never requires *unbounded* work.

But the implementation relies on *laziness*, *sharing*, and *aliasing*…

Worst case constant time queues

Okasaki presents several data structures for *worst case constant time* queues.

Idea:

- Implement a queue using two lists;
- Ensure each rebalancing operation does some *constant* amount of work…
- …but guarantee that rebalancing never requires *unbounded* work.

But the implementation relies on *laziness*, *sharing*, and *aliasing*…

How does fit in this framework?

what can you tell me about first order laziness?

First-order laziness usually refers to a specific cognitive bias in thinking and problem-solving:

> The tendency to accept the first explanation, interpretation, or solution that comes to mind without examining deeper or alternative possibilities.

It's not a formal psychological diagnosis, but it's a useful concept in reasoning, philosophy, and decision-making.

First order laziness in Koka

```
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
  lazy SReverse(s1 : stream<a>, s2 : stream<a>) ->
    ...
```

Lazy constructors require constant time to create;

Pattern matching on a `stream<a>` forces any lazy constructors, executing the associated code, until we reach a whnf.

Towards first order laziness in Agda

```
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
```

Together with a function

```
whnf? : Stream A → Set
```

Small steps…

We define a `step` function that performs a single evaluation step.

The corresponding graph:

```
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-[]       : app [] ys ⇝ ys
  app-∷        : app (x ∷ xs) ys ⇝ (x ∷ app xs ys)
  rev-[]       : rev [] ys ⇝ ys
  rev-∷        : rev (x ∷ xs) ys ⇝ rev xs (x ∷ ys)
```

Computing weak head normal forms

The `seq : Stream A → Stream A` function forces to weak head normal form:

The corresponding graph:

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

Seqing Convenience

We introduce a separate data type specifically for weak head normal forms that arise from a call to seq:

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

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

Costs

The cost of each call to `step` is one; the cost of each call to `seq` is 1 + the number of steps.

For instance, we show (provided xs and ys are fully evaluated):

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

Costs

The cost of each call to `step` is one; the cost of each call to `seq` is 1 + the number of steps.

For instance, we show (provided `xs` and `ys` are fully evaluated):

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

If reversal takes a *linear* amount of time, how can we ever write *worst case* constant queue operations?

Spreading the work

Suppose we have a stream of the form `app xs (rev ys zs)`.

We want to repeatedly cal `seq` on this stream, discarding the head.

The first few calls to `seq` will be cheap… until we hit the `rev` constructor.

Spreading the work

Suppose we have a stream of the form `app xs (rev ys zs)`.

We want to repeatedly cal `seq` on this stream, discarding the head.

The first few calls to `seq` will be cheap… until we hit the `rev` constructor.

But what now suppose `xs` and `ys` have (approximately) the same length.

If we also `step` the `rev` constructor after each `seq`, by the time we run out of elements in `xs` the expensive reversal is finished.

Spreading the work

Suppose we have a stream of the form `app xs (rev ys zs)`.

We want to repeatedly cal `seq` on this stream, discarding the head.

The first few calls to `seq` will be cheap… until we hit the `rev` constructor.

But what now suppose `xs` and `ys` have (approximately) the same length.

If we also `step` the `rev` constructor after each `seq`, by the time we run out of elements in `xs` the expensive reversal is finished.

```
steprev : Stream A → Stream A
steprev (app xs (rev ys zs))    = app xs (step (rev ys zs))
steprev xs                      = xs
```

We never need unbounded work – the expensive reverse is ‘guarded’ by the `app` prefix.

Real time queues

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

- New elements added to the `rear`; elements dequeued from the `front`. But how to rebalance?

What is the schedule?

Real time queues

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

- New elements added to the `rear`; elements dequeued from the `front`. But how to rebalance?

What is the schedule?

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.

Defining the suffix relation

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

plug : Ctx A → Stream A → Stream A
_++_ : Ctx A → Ctx A → Ctx A
```

I will sometimes also write `c [ xs ]` for `plug c xs`.

Defining the suffix relation

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

plug : Ctx A → Stream A → Stream A
_++_ : Ctx A → Ctx A → Ctx A
```

I will sometimes also write `c [ xs ]` for `plug c xs`.

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

Generalises nicely to other data types (cf. the derivative of a functor is its one hole context).

Captures intuition about sharing: if the context `c` witnesses `xs ⊑ ys` and `xs ⇝ xs'` then `c [ xs ]` also becomes `c [ xs' ]`.

Invariants

What does rebalancing need to do?

There are three invariants that we will enforce:

- The rear does not contain lazy constructors;
- The length of the evaluated prefix of the front stream is equal to the length of the rear;
- The schedule is valid:

```
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))
```

Rebalancing (after an enqueue)

```
balance : (front rear sched : Stream A) → (sched ⊑ front) → Q A
```

Rebalancing (after an enqueue)

```
balance : (front rear sched : Stream A) → (sched ⊑ front) → Q A

balance .(c [ sched ]) rear sched (suffix c) with seq sched
```

Rebalancing (after an enqueue)

```
balance : (front rear sched : Stream A) → (sched ⊑ front) → Q A

balance .(c [ sched ]) rear sched (suffix c) with seq sched

... | ⟨ [] ⟩      =  let f = app (plug c []) (rev rear []) in
                      queue f [] f ⊑-refl
```

Rebalancing (after an enqueue)

```
balance : (front rear sched : Stream A) → (sched ⊑ front) → Q A

balance .(c [ sched ]) rear sched (suffix c) with seq sched

... | ⟨ [] ⟩      = let f = app (plug c []) (rev rear []) in
                    queue f [] f ⊑-refl

... | ⟨ x ∷ xx ⟩  = let sched' = steprev xx in
                    let c' = c ++ (x ∷ □) in
                    queue (plug sched' c') rear sched' (suffix c')
```

The ‘bookkeeping’ operations – reestablishing the suffix invariant – have zero cost.

Cost analysis

Under the assumption that the schedule is valid, it is easy to prove that rebalancing is worst case constant time.

Key property: evaluating the schedule to weak head normal form always takes constant time:

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

Cost analysis

Under the assumption that the schedule is valid, it is easy to prove that rebalancing is worst case constant time.

Key property: evaluating the schedule to weak head normal form always takes constant time:

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

From this, it is easy to prove that each queue operation takes at most constant time.

Future work

I have carefully avoided reasoning about *amortized complexity in a lazy setting*.

Anton Lorenzen has done recent work about justifying Okasaki's debit style reasoning and credit-based reasoning through explicit heaps – but it is fairly 'operational'.

We can define an ordering on streams xs $\prec$ ys when evaluating (parts of) xs produces ys.

- What properties of `credit` ensure that the amortization argument does not hold for one xs but all ys for which xs $\prec$ ys'?
- Can we produce a simple equational argument justifying lazy amortization?
- Can we use *constructor contexts* to factor out the sharing in real time queues?

Conclusions

- It is (surprisingly?) easy to prove worst case time complexity bounds of purely functional data structures and algorithms in this style;
- Formalising proofs – such as the typical amortization argument – makes the underlying assumptions explicit.
- Reasoning about laziness and sharing – without heaps! – in a purely functional setting.
- Lazy constructors are useful when specifying the key invariants, such as ensuring the schedule is well formed.