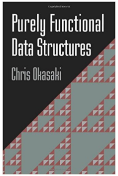
Thunks and Debits in Iris with Time Credits

Pottier	Guéneau	Jourdan	Mével	POPL	2024

```
val create: (unit -> ’a) -> ’a thunk
val force : ’a thunk -> ’a
```

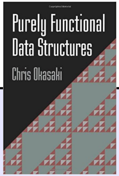
```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
val create: (unit -> 'a) -> 'a thunk
val force : 'a thunk -> 'a
```




```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

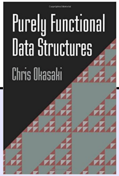
```
val create: (unit -> 'a) -> 'a thunk
val force : 'a thunk -> 'a
```




```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
val create: (unit -> 'a) -> 'a thunk
val force : 'a thunk -> 'a
```

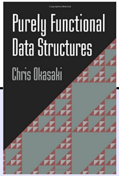



```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
val create: (unit -> 'a) -> 'a thunk
val force : 'a thunk -> 'a
```

informal amortized
time complexity analysis
of purely functional
lazy data structures



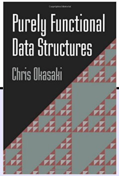

```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
val create: (unit -> 'a) -> 'a thunk
val force : 'a thunk -> 'a
```

informal amortized
time complexity analysis
of purely functional
lazy data structures

credit-based reasoning
about thunks?



NO!

```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

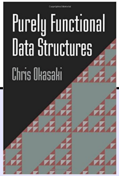
```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
val create: (unit -> 'a) -> 'a thunk
val force : 'a thunk -> 'a
```

informal amortized
time complexity analysis
of purely functional
lazy data structures

UNSOUND!

credit-based reasoning
about thunks?




```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
val create: (unit -> 'a) -> 'a thunk
val force : 'a thunk -> 'a
```

informal amortized
time complexity analysis
of purely functional
lazy data structures

SOUND

debit-based reasoning
about thunks

```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

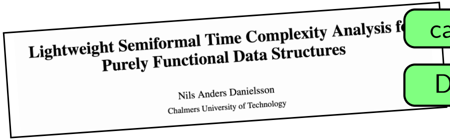
```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
val create: (unit -> 'a) -> 'a thunk
val force : 'a thunk -> 'a
```

informal amortized
time complexity analysis
of purely functional
lazy data structures

debit-based reasoning
about thunks

can be formalized


Lightweight Semiformal Time Complexity Analysis f
Purely Functional Data Structures

Nils Anders Danielsson
Chalmers University of Technology

Danielsson, 2008

```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
let create f = ref (UNEVALUATED f)
let force t  = match !t with ...
```

implement thunks
using mutable state

informal amortized
time complexity analysis
of purely functional
lazy data structures

debit-based reasoning
about thunks

```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
let create f = ref (UNEVALUATED f)
let force t  = match !t with ...
```

informal amortized
time complexity analysis
of purely functional
lazy data structures

debit-based reasoning
about thunks

Iris with time **credits** ✓

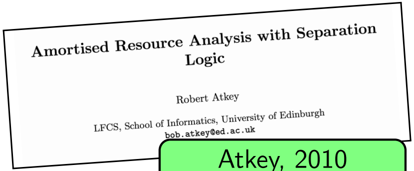
```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
let create f = ref (UNEVALUATED f)
let force t  = match !t with ...
```

informal amortized time complexity analysis of purely functional lazy data structures

debit-based reasoning about thunks


Amortised Resource Analysis with Separation Logic

Robert Atkey

LFCS, School of Informatics, University of Edinburgh
bob.atkey@ed.ac.uk

Atkey, 2010

Iris with time **credits**

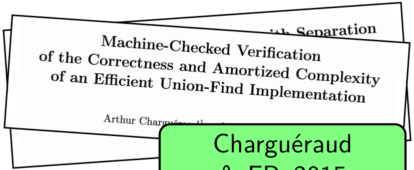
```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
let create f = ref (UNEVALUATED f)
let force t  = match !t with ...
```

informal amortized time complexity analysis of purely functional lazy data structures

debit-based reasoning about thunks



Charguéraud & FP, 2015

Iris with time **credits**

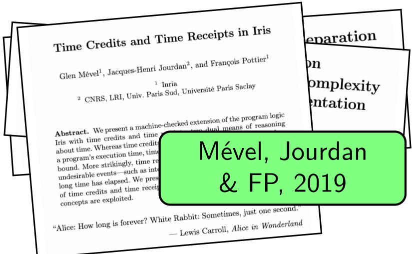
```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

informal amortized time complexity analysis of purely functional lazy data structures

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
let create f = ref (UNEVALUATED f)
let force t  = match !t with ...
```

debit-based reasoning about thunks


Time Credits and Time Receipts in Iris

Glen Mével[1], Jacques-Henri Jourdan[2], and François Pottier[1]

[1] Inria
[2] CNRS, LRI, Univ. Paris Sud, Université Paris Saclay

Abstract. We present a machine-checked extension of the program logic Iris with time credits and time ... about time. Whereas time credits ... a program's execution time, tim... bound. More strikingly, time re... undesirable events—such as int... long time has elapsed. We pres... of time credits and time receip... concepts are exploited.

"Alice: How long is forever? White Rabbit: Sometimes, just one second."
— Lewis Carroll, *Alice in Wonderland*

Mével, Jourdan & FP, 2019

Iris with time **credits**

```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
let create f = ref (UNEVALUATED f)
let force t  = match !t with ...
```

informal amortized time complexity analysis of purely functional lazy data structures

debit-based reasoning about thunks

$\$: \mathbb{N} \to iProp$

$\$(n_1 + n_2) \equiv \$n_1 * \$n_2$

$\{\$1\}\ tick()\ \{True\}$

Iris with time **credits**

```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
let create f = ref (UNEVALUATED f)
let force t  = match !t with ...
```

informal amortized
time complexity analysis
of purely functional
lazy data structures

debit-based reasoning
about thunks

ghost piggy bank API
with **credit/debit** reasoning

Iris with time **credits**

```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
let create f = ref (UNEVALUATED f)
let force t  = match !t with ...
```

informal amortized
time complexity analysis
of purely functional
lazy data structures

debit-based reasoning
about thunks

$$\mathit{PiggyBank}\ P\ Q\ N\ T\ 0 \ast \text{↯}^{\mathcal{F}} \Rrightarrow_{\mathcal{E}} \exists nc. \left(\begin{array}{l} ((\triangleright P\ nc \ast \$nc) \lor \triangleright Q) \ast \\ (\triangleright Q \Rrightarrow_{\mathcal{E}} \text{↯}^{\mathcal{F}}) \end{array} \right)$$

host piggy bank API
credit/debit reasoning

Iris with time **credits**

```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
let create f = ref (UNEVALUATED f)
let force t  = match !t with ...
```

informal amortized
time complexity analysis
of purely functional
lazy data structures

thunk API
with **credit/debit** reasoning ✓

ghost piggy bank API
with **credit/debit** reasoning ✓

Iris with time **credits** ✓

```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

```
let creat
let force
```

$$\mathit{Thunk}\ \mathcal{F}\ t\ n\ R\ \phi * \$k \Rrightarrow_{\mathcal{E}} \mathit{Thunk}\ \mathcal{F}\ t\ (n-k)\ R\ \phi$$

informal amortized
time complexity analysis
of purely functional
lazy data structures

thunk API
credit/debit reasoning

ghost piggy bank API
with **credit/debit** reasoning

Iris with time **credits**

```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

stream API
with **credit/debit** reasoning ✓

```
let create f = ref (UNEVALUATED f)
let force t  = match !t with ...
```

thunk API
with **credit/debit** reasoning ✓

ghost piggy bank API
with **credit/debit** reasoning ✓

Iris with time **credits** ✓

```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

```
type 'a s
 and 'a c
   = Nil
```

$$\frac{(m)\ ds_1 \leq ds_2\ (n)}{Stream\ h\ s\ ds_1\ xs * \$m \Rrightarrow_{\varepsilon} Stream\ h\ s\ ds_2\ xs}$$

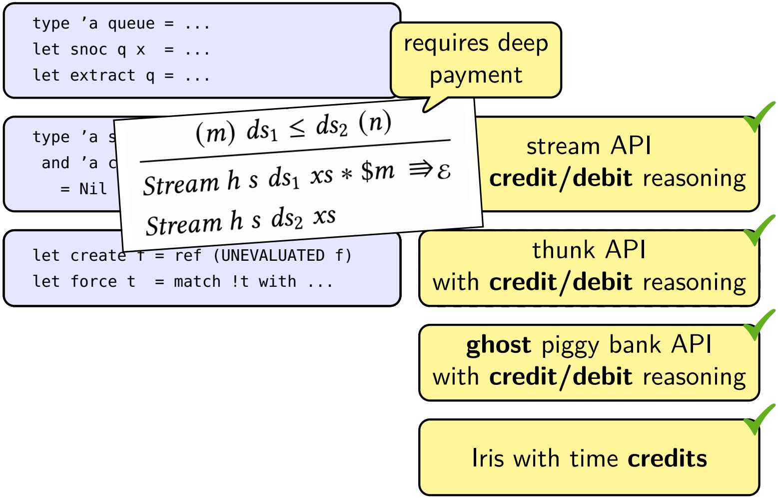
```
let create f = ref (UNEVALUATED f)
let force t  = match !t with ...
```

stream API
credit/debit reasoning

thunk API
with **credit/debit** reasoning

ghost piggy bank API
with **credit/debit** reasoning

Iris with time **credits**



type 'a queue = ...
let snoc q x = ...
let extract q = ...
type 'a s
and 'a c
= Nil
let create f = ref (UNEVALUATED f)
let force t = match !t with ...
requires deep payment
(m) ds1 ≤ ds2 (n)
Stream h s ds1 xs * $m ⇛ε
Stream h s ds2 xs
stream API
credit/debit reasoning
thunk API
with credit/debit reasoning
ghost piggy bank API
with credit/debit reasoning
Iris with time credits

```
type 'a queue = ...
let snoc q x  = ...
let extract q = ...
```

banker's queue API
purely **credit**-based

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

stream API
with **credit/debit** reasoning

```
let create f = ref (UNEVALUATED f)
let force t  = match !t with ...
```

thunk API
with **credit/debit** reasoning

ghost piggy bank API
with **credit/debit** reasoning

Iris with time **credits**

$$\{\$\underline{X} \ast \mathit{BQueue}\ q\ (x :: xs) \ast \lightning^{\infty}\}$$
$$\mathit{extract}\ q$$
$$\{\lambda(x', q').\ \ulcorner x' = x \urcorner \ast \mathit{BQueue}\ q'\ xs \ast \lightning^{\infty}\}$$

banker's queue API
purely **credit**-based

```
type 'a stream = 'a cell thunk
 and 'a cell
   = Nil | Cons of 'a * 'a stream
```

stream API
with **credit/debit** reasoning

```
let create f = ref (UNEVALUATED f)
let force t  = match !t with ...
```

thunk API
with **credit/debit** reasoning

ghost piggy bank API
with **credit/debit** reasoning

Iris with time **credits**

In summary,

following up on Okasaki (1999), Danielsson (2008), MJP (2019),

we use a rich Separation Logic to perform **machine-checked** proofs

of **correctness** and **time complexity**

of a stack of libraries

that marry **imperative** and **functional** programming.

We explain **debits** and **deep payment** in terms of **credits**.

Thunks

Thunks: Implementation

A thunk is a **mutable data structure** that offers a memoization service.

```
type 'a state = UNEVALUATED of (unit -> 'a) | EVALUATED of 'a
type 'a thunk = 'a state ref
let create f  = ref (UNEVALUATED f)
let force t   =
  match !t with
  | UNEVALUATED f -> let v = f() in t := EVALUATED v; v
  |   EVALUATED v -> v
```

An abstract predicate *Thunk* t n ϕ
where t is the thunk, n is its debit, ϕ is its postcondition.

Two runtime operations: **creating and forcing** a thunk,

and several ghost operations, including **sharing** and **paying**.

THUNK-CREATE

$$\{\$\underline{N} * \textbf{once}\,\{\$n\}\,f()\,\{\lambda v.\,\Box\,\phi\,v\}\}$$
$$\textit{create } f$$
$$\{\lambda t.\,\textit{Thunk } t\; n\;\phi\}$$

Creating a thunk costs $O(1)$ credits.

If the suspended computation costs n credits then the thunk has debit n.

- Say Alice wants to suspend a computation whose cost is 10.
- She creates a thunk, whose debit is initially 10.

Thunk API: Ordinary Payment

Thunk-Pay

$Thunk\ t\ n\ \phi * \$k \Rrightarrow Thunk\ t\ (n-k)\ \phi$

Paying consumes credits and reduces a thunk's debit.

- Say Alice pays $2. Then Alice knows the remaining debit is 8.

Paying is permitted at all times.

Thunk API: Sharing

THUNK-PERSISTENT
persistent($Thunk\ t\ n\ \phi$)

Sharing a thunk is permitted.

Each principal has **its own view** of the debit
and can pay independently, so debit is an **over-approximation** of true debt.

- Say Alice tells Bob and Charlie that the debit is 8.
- Say Bob pays \$1. Bob knows the debit is 7.
- Say Charlie pays \$8. Charlie knows the debit is 0.

Thunk API: Forcing

THUNK-FORCE

$$\{Thunk\ t\ 0\ \phi\ *\ \$\underline{F}\ *\ \lightning\}$$
$$force\ t$$
$$\{\lambda v.\ \Box\ \phi\ v\ *\ \lightning\}$$

Whoever knows the debit is 0 can **force the thunk**.

Forcing costs $O(1)$ credits.

A thunk can be forced many times.

Thunk API: Deep Payment

Whereas ordinary payment **consumes** credits and **reduces** a thunk's debit,

THUNK-PAY

$$Thunk\ t\ n\ \phi * \$k \Rrightarrow Thunk\ t\ (n-k)\ \phi$$

Thunk API: Deep Payment

Whereas ordinary payment **consumes** credits and **reduces** a thunk's debit,

THUNK-PAY
$$\mathit{Thunk}\ t\ n\ \phi\ *\ \$k \Rrightarrow \mathit{Thunk}\ t\ (n-k)\ \phi$$

deep payment **increases** a thunk's debit and **produces** credits
for use **in the future**, when this thunk is forced.

Thunk API: Deep Payment

Whereas ordinary payment **consumes** credits and **reduces** a thunk's debit,

THUNK-PAY

$$\mathit{Thunk}\ t\ n\ \phi\ *\ \$k \Rrightarrow \mathit{Thunk}\ t\ (n-k)\ \phi$$

THUNK-CONSEQUENCE

$$\mathit{Thunk}\ t\ n_1\ \phi \mathrel{-\!\!*} (\forall v.\ (\$n_2 * \Box\ \phi\ v) \Rrightarrow \Box\ \psi\ v) \Rrightarrow \mathit{Thunk}\ t\ (n_1 + n_2)\ \psi$$

deep payment **increases** a thunk's debit and **produces** credits for use **in the future**, when this thunk is forced.

Whereas ordinary payment **consumes** credits and **reduces** a thunk's debit,

THUNK-PAY

$$\mathit{Thunk}\ t\ n\ \phi\ *\ \$k \Rrightarrow \mathit{Thunk}\ t\ (n-k)\ \phi$$

THUNK-CONSEQUENCE

THUNK-DEEP-PAY-EXAMPLE

$$\mathit{Thunk}\ t_1\ n_1\ (\lambda t_2. \mathit{Thunk}\ t_2\ n_2\ \phi) \Rrightarrow \mathit{Thunk}\ t_1\ (n_1+n_2)\ (\lambda t_2. \mathit{Thunk}\ t_2\ 0\ \phi)$$

deep payment **increases** a thunk's debit and **produces** credits for use **in the future**, when this thunk is forced.

Deep payment implies that debits can be **shifted towards the left**.

A key rule, whose justification is new in this work and involves **ghost piggy banks**.

Streams

A stream's elements are **computed on demand** and **memoized**.

```
type 'a stream = 'a cell thunk
 and 'a cell   = Nil | Cons of 'a * 'a stream
```

Streams are also known as lazy lists, or just **lists** in Haskell.

An abstract predicate $\mathit{Stream}\ s\ \vec{d}\ \vec{x}$
where s is the stream, $\vec{d}$ is its sequence of debits, $\vec{x}$ is its sequence of elements.

Streams can be **shared**.

Debits can be **shifted towards the left**.

Stream-Persist
persistent($\mathit{Stream}\ s\ \vec{d}\ \vec{x}$)

Stream-Shift-Debit
$\ulcorner \vec{d}_1 \leq \vec{d}_2 \urcorner \mathrel{-\!\!*}$
$\mathit{Stream}\ s\ \vec{d}_1\ \vec{x} \Rrightarrow$
$\mathit{Stream}\ s\ \vec{d}_2\ \vec{x}$

An abstract predicate $\mathit{Stream}\ s\ \vec{d}\ \vec{x}$
where s is the stream, $\vec{d}$ is its sequence of debits, $\vec{x}$ is its sequence of elements.

Streams can be **shared**.

Debits can be **shifted towards the left**.

STREAM-PERSIST
$$\text{persistent}(\mathit{Stream}\ s\ \vec{d}\ \vec{x})$$

STREAM-SHIFT-DEBIT

STREAM-SHIFT-DEBIT-EXAMPLE
$$\mathit{Stream}\ s\ (\underbrace{0, 0, \ldots, 0}_{n \text{ times}}, n)\ \vec{x} \Rrightarrow$$
$$\mathit{Stream}\ s\ (\underbrace{1, 1, \ldots, 1}_{n \text{ times}}, 0)\ \vec{x}$$

The banker's queue

The Banker's Queue: OCaml Code

A FIFO queue (Okasaki, 1999). Every operation has amortized time complexity $O(1)$.

```
type 'a queue =
  { lenf: int; f:  'a stream; lenr: int; r:  'a list }
let empty () =
  { lenf = 0; f = nil(); lenr = 0; r = [] }
let check ({ lenf = lenf ; f = f; lenr = lenr; r = r } as q) =
  if lenf >= lenr then q
  else { lenf = lenf + lenr; f = append f (revl r); lenr = 0; r = [] }
let snoc q x =
  check { q with lenr = q.lenr + 1; r = x :: q.r }
let extract q =
  let x, f = uncons q.f in
  x, check { q with f = f; lenf = q.lenf - 1 }
```

The expression `append f (revl r)` constructs a stream whose debit sequence is (roughly)

$$\underbrace{1, 1, \ldots, 1}_{n \text{ times}}, n, \underbrace{0, 0, \ldots, 0}_{n \text{ times}}$$

By **shifting debits towards the left**, the debit sequence can be smoothened up:

$$\underbrace{2, 2, \ldots, 2}_{n \text{ times}}, 0, \underbrace{0, 0, \ldots, 0}_{n \text{ times}}$$

Thus every debit is $O(1)$, which is why `extract` costs only $O(1)$.

Ghost piggy banks

Piggy Banks: API Overview

An abstraction with four main operations: **creating, paying, sharing, forcing** a bank.

Piggy banks **do not exist** at runtime: all operations are ghost state updates.

The piggy bank API involves both credits and debits.

Paying and **sharing** works in the same way as for thunks.

PiggyBank-Pay

$$\mathit{PiggyBank}_{P,Q}\ n \ast \$k \Rrightarrow \mathit{PiggyBank}_{P,Q}\ (n-k)$$

PiggyBank-Persist

$$\text{persistent}(\mathit{PiggyBank}_{P,Q}\ n)$$

When a piggy bank is created, a **target amount** is fixed, and becomes the initial **debit**.

An **initial property** P and a **target property** Q are also fixed upon creation.

- Say P holds initially.
- Alice creates a piggy bank with initial debit 10.
- Her purpose is to gather \$10 and spend it to execute a transition from P to Q.

PiggyBank-Create

$$P\ n \Rrightarrow PiggyBank_{P,Q}\ n$$

Whoever knows the debit is 0 can **force the bank**.

They get the collected credit and must establish Q.

A bank can be forced several times.

- Say Charlie forces the bank first.
  He gets $\$10$
  and can spend them to run code that establishes Q.
- Say Alice later forces the bank.
  She gets $\$0$
  and learns that Q holds already.

Forcing the bank requires a unique token: this forbids reentrancy/concurrency.

PiggyBank-Break

$$\mathit{PiggyBank}_{P,Q}\ 0 \ast \lightning \Rrightarrow \exists n.\left(\begin{array}{l}((\triangleright P\ n \ast \$n) \vee \triangleright Q) \ast \\ (\triangleright Q \Rrightarrow \lightning)\end{array}\right)$$

Piggy banks **do not support** deep payment, so they are simpler than thunks.

Our construction of thunks can allocate **several piggy banks** per thunk:

- when a new thunk is created,
  a new piggy bank is created for it;
- when a deep payment is made on an existing thunk,
  a **new piggy bank** is created for this thunk,
  so a **new target amount** and a **new target property** can be set.

> This data structure also illustrates a subtle point about nested suspensions—the debits for a nested suspension may be allocated, and even discharged, before the suspension is physically created. For example, consider how

Okasaki (1999)

Conclusion

Debits and deep payment can be explained in terms of credits!

In the paper:

- forbidding **reentrancy** = guaranteeing **productivity**;
  achieved by indexing thunks with **heights**;
- **correctness** and amortized **time complexity** of 3 data structures by Okasaki.

Limitations:

- only 3 data structures verified in this paper;
- making Iris$^{\$}$ more user-friendly would require some engineering work;
- open problem: how to control the time complexity of unbounded waiting loops?

Backup Slides

Reversing a list and converting it to a stream:

```
let rec append (s1 : 'a stream) (s2 : 'a stream) : 'a stream =
  Thunk.create @@ fun () -> match Thunk.force s1 with
  | Nil          -> Thunk.force s2
  | Cons (x, s1) -> Cons (x, append s1 s2)

let rec revl_append (l : 'a list) (c : 'a cell) : 'a cell =
  match l with
```

Concatenating two streams:

```
  | x :: l -> revl_append l (Cons (x, Thunk.create @@ fun () -> c))

let revl (l : 'a list) : 'a stream =
  Thunk.create @@ fun () -> revl_append l Nil
```

The **debit subsumption** judgement

$$\vec{d}_1 \le \vec{d}_2$$

can be defined as follows:

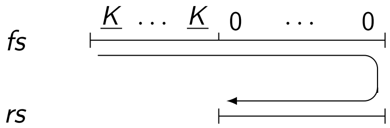
$$\forall i. \quad \sum(\mathit{take}\ i\ \vec{d}_1) \le \sum(\mathit{take}\ i\ \vec{d}_2)$$

This judgement **moves debits towards the left**.

There is a **front stream** *fs* and a **rear list** *rs*. One maintains $|fs| \geq |rs|$.

Every thunk in *fs* carries a certain **debt** or **debit**.

The first $|fs| - |rs|$ thunks have debt $\underline{K}$; the rest have debt 0.



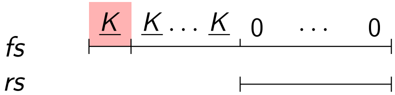
Elements are **inserted** in the rear, **extracted** from the front.

The Banker's Queue: Extraction

If $|fs| > |rs|$, then extraction does not require rebalancing.

Extraction requires **paying** K before the first thunk can be forced.

Including this payment, its time complexity is $O(1)$.

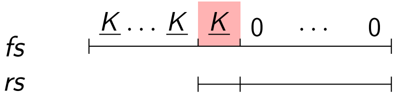


The Banker's Queue: Insertion

If $|fs| > |rs|$, then insertion does not require rebalancing.

Insertion actually consumes O(1) time,

and requires paying K to maintain the invariant.



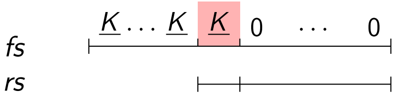
A **deep payment**,

possibly involving a thunk **that does not even exist yet** in memory!

If $|fs| > |rs|$, then insertion does not require rebalancing.

Insertion actually consumes O(1) time,

and requires paying K to maintain the invariant.

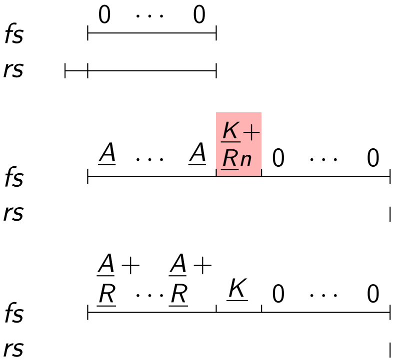


> This data structure also illustrates a subtle point about nested suspensions—the debits for a nested suspension may be allocated, and even discharged, before the suspension is physically created. For example, consider how ++ works.

A **deep payme**

possibly involving a thunk **that does not even exist yet** in memory!

Rebalancing involves *revl*, *append*, and a **redistribution** of debits.



The queue is unbalanced.
$|fs| = n \wedge |rs| = n + 1$

Reverse and append the rear list to the front stream.

Redistribute debits by adding R to the first n debits.

Moving debits towards the left is safe: it requires earlier payments.

The Banker's Queue: Public API

The banker's queue admits a simple **specification** in Iris$^{\$}$.

Banker-Persistent
persistent($BQueue\ q\ \vec{x}$)

Banker-Empty
$\{\$\underline{E}\}\ empty\ ()\ \{\lambda q.\ BQueue\ q\ []\}$

Queues are **persistent**. **Creation** costs $O(1)$.

Insertion and **extraction** cost $O(1)$.

Banker-Snoc

$$\{\$\underline{S} \ast \mathit{BQueue}\ q\ \vec{x}\}\ \mathit{snoc}\ q\ x\ \{\lambda q'.\ \mathit{BQueue}\ q'\ (\vec{x} +\!\!+ [x])\}$$

Banker-Extract

$$\{\$\underline{X} \ast \mathit{BQueue}\ q\ (x :: \vec{x}) \ast \lightning\}$$

$$\mathit{extract}\ q$$

$$\{\lambda(x', q').\ \ulcorner x' = x \urcorner \ast \mathit{BQueue}\ q'\ \vec{x} \ast \lightning\}$$

Extraction requires a token $\lightning$.

Extraction forces a thunk, and **thunks are not thread-safe**.