Thunks and Debits in Iris with Time Credits
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

credit-based reasoning
about thunks?
debit-based reasoning
about thunks

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

Iris with time credits
ghost piggy bank API
with credit/debit reasoning
thunk API with credit/debit reasoning
stream API with credit/debit reasoning
banker's queue API purely credit-based
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
val create: (unit -> 'a) -> 'a thunk
val force : 'a thunk -> 'a

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

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

Iris with time credits

ghost piggy bank API

with credit/debit reasoning

thunk API
with credit/debit reasoning

stream API
with credit/debit reasoning

banker's queue API
purely credit-based

credit-based reasoning about thunks?
debit-based reasoning about thunks

credit-based reasoning

Purely Functional Data Structures
Chris Okasaki
informal amortized time complexity analysis of purely functional lazy data structures
Informal amortized time complexity analysis of purely functional lazy data structures

credit-based reasoning about thunks?
Informal amortized time complexity analysis of purely functional lazy data structures

Credit-based reasoning about thunks?

unsound!
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

Informal amortized time complexity analysis of purely functional lazy data structures

Debit-based reasoning about thunks

Iris with time credits ghost piggy bank API with credit/debit reasoning
Stream API with credit/debit reasoning
Banker's queue API purely credit-based
Informal amortized time complexity analysis of purely functional lazy data structures

deficit-based reasoning about thunks

can be formalized

Danielsson, 2008

Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures

Nils Anders Danielsson
Chalmers University of Technology
**informal** amortized time complexity analysis of purely functional lazy data structures

debit-based reasoning about thunks

implement thunks using mutable state
Informal amortized time complexity analysis of purely functional lazy data structures

debit-based reasoning about thunks

Iris with time credits
Amortised Resource Analysis with Separation Logic

Robert Atkey
LFCS, School of Informatics, University of Edinburgh
bob.atkey@ed.ac.uk
Atkey, 2010

Informal amortized time complexity analysis of purely functional lazy data structures

Debit-based reasoning about thunks

Iris with time credits
let create f = ref (UNEVALUATED f)
let force t = match !t with ...

let snoc q x = ...
let extract q = ...

informal amortized time complexity analysis of purely functional lazy data structures

debit-based reasoning about thunks

Iris with time credits

Charguéraud & FP, 2015
type 'a queue = ...
let snoc q x = ...
let extract q = ...

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
informal amortized time complexity analysis of purely functional lazy data structures

debit-based reasoning about thunks

\[
\begin{align*}
\text{Informal amortized time complexity analysis of purely functional lazy data structures} \\
\text{debit-based reasoning about thunks} \\
\end{align*}
\]
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
informal amortized
time complexity analysis
of purely functional
lazy data structures

credit-based reasoning
about thunks

debit-based reasoning
about thunks

Piggy Bank

\[ P \mathcal{Q} N \mathcal{T} 0 * \frac{F}{\varepsilon} \Rightarrow \varepsilon \]

\[ \exists \text{enc. } \left( \left( (\Rightarrow P \text{ nc } \mathcal{S} \text{ nc}) \vee \Rightarrow Q \right) * \right) \]

Iris with time credits

Ghost piggy bank API

credit/debit reasoning

Thunk API

credit/debit reasoning

Stream API

credit/debit reasoning

Banker's Queue API

purely credit-based
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
informal amortized time complexity analysis of purely functional lazy data structures

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

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

Thunk \( F_t n R \phi \star S_k \Rightarrow E \)
Thunk \( F_t (n - k) R \phi \)

thunk API credit/debit reasoning

ghost piggy bank API with credit/debit reasoning

Iris with time credits
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
type 'a queue = ...
  let snoc q x = ...
  let extract q = ...

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

credit-based reasoning
debit-based reasoning

Iris with time credits

stream API
with credit/debit reasoning

thunk API
with credit/debit reasoning

ghost piggy bank API
with credit/debit reasoning

Iris with time credits
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

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

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

credit-based reasoning about thunks?
debit-based reasoning about thunks

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

Iris with time credits

credit/debit reasoning

thunk API
with credit/debit reasoning

ghost piggy bank API
with credit/debit reasoning

Iris with time credits

credit/debit reasoning

stream API

stream API

purely credit-based banker's queue API

Stream \( h \ s \ ds_1 \ xs \) * \( m \) \( \Rightarrow \varepsilon \)
Stream \( h \ s \ ds_2 \ xs \)

\( ds_1 \leq ds_2 \) (n)
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

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

credit-based reasoning about thunks?

debit-based reasoning about thunks

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

Iris with time credits

ghost piggy bank API with credit/debit reasoning

stream API with credit/debit reasoning

thunk API with credit/debit reasoning

requires deep payment

Stream \( h \) \( s \) \( d_s_1 \) \( x_s \) * \( m \) ⇒ \( \varepsilon \)

\( m \) \( d_s_1 \) \( \leq \) \( d_s_2 \) \( (n) \)

Stream \( h \) \( s \) \( d_s_2 \) \( x_s \)
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

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

Iris with time credits

banker’s queue API
purely credit-based

stream API
with credit/debit reasoning

thunk API
with credit/debit reasoning

ghost piggy bank API
with credit/debit reasoning

Iris with time credits
CREATE: $(\lambda f. (\text{UNEVALUATED } f))$

FORCE: $(\lambda t. \text{match } !t \text{ with } ...)$

TYPE: $(\forall 'a. \text{cell}\thunk)$

STREAM: $(\text{Nil | Cons of } 'a \times 'a\ \text{stream})$

QUEUE: $(\text{...})$

LET: $(\text{let create } f = \text{ref (UNEVALUATED } f) \land \text{let force } t = \text{match } !t \text{ with } ...)$

BANKER’S QUEUE API: purely $\text{credit}$-based

STREAM API: with $\text{credit/debit}$ reasoning

THUNK API: with $\text{credit/debit}$ reasoning

GHOST PIGGY BANK API: with $\text{credit/debit}$ 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.
A thunk is a **mutable data structure** that offers a memoization service.

```ocaml
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 \( Thunk \ t \ n \ \phi \) where \( t \) is the thunk, \( n \) is its debit, \( \phi \) is its postcondition.

Two runtime operations: **creating and forcing** a thunk, and several ghost operations, including **sharing** and **paying**.
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.
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.
**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.
Whoever knows the debit is 0 can force the thunk.

Forcing costs $O(1)$ credits.

A thunk can be forced many times.
Whereas ordinary payment consumes credits and reduces a thunk’s debit,

\[ Thunk \ t \ n \ \phi \ \ast \ \$k \ \Rightarrow \]

\[ Thunk \ t \ (n - k) \ \phi \]
Whereas ordinary payment *consumes* credits and *reduces* a thunk’s debit,

\[
\text{Thunk-Pay} \\
\text{Thunk } t \ n \ \phi \ast \$k \Rightarrow \text{Thunk } t (n - k) \ \phi
\]

deep payment *increases* a thunk’s debit and *produces* credits for use *in the future*, when this thunk is forced.
Whereas ordinary payment consumes credits and reduces a thunk’s debit, deep payment increases a thunk’s debit and produces credits for use in the future, when this thunk is forced.

**Thunk-Pay**

\[
\text{Thunk } t \ n \ \phi \ * \ $k \ \Rightarrow \\
\text{Thunk } t \ (n - k) \ \phi
\]

**Thunk-Consequence**

\[
\text{Thunk } t \ n_1 \ \phi \ \rightarrow \\
(\forall v. \ ($n_2 \ * \ □ \ \phi \ v) \ \Rightarrow \ □ \ \psi \ v) \ \Rightarrow \\
\text{Thunk } t \ (n_1 + n_2) \ \psi
\]
Whereas ordinary payment consumes credits and reduces a thunk’s debit,

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**.

```haskell
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 $Stream\ s \ \vec{d} \ \vec{x}$
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**

$\text{persistent}(Stream\ s \ \vec{d} \ \vec{x})$

**Stream-Shift-Debit**

$\lceil \vec{d}_1 \leq \vec{d}_2 \rfloor \Rightarrow$

$Stream\ s \ \vec{d}_1 \ \vec{x} \ \vec{x} \ \vec{x}$

$Stream\ s \ \vec{d}_2 \ \vec{x}$
An abstract predicate $\text{Stream } s \vec{d} \vec{x}$
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.

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

$$\text{STREAM-SHIFT-DEBIT}$$
$$\text{Stream } s (0, 0, \ldots, 0, n) \vec{x} \Rightarrow$$
$$\text{Stream } s (1, 1, \ldots, 1, 0) \vec{x}$$

$$\text{STREAM-SHIFT-DEBIT-EXAMPLE}$$
$$\text{Stream } s (0, 0, \ldots, 0, n) \vec{x} \Rightarrow$$
$$\text{Stream } s (1, 1, \ldots, 1, 0) \vec{x}$$
The banker’s queue
A FIFO queue (Okasaki, 1999). Every operation has amortized time complexity $O(1)$.

```ocaml
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 \( \text{append } f \ (\text{revl } r) \) constructs a stream whose debit sequence is (roughly) 

\[
\underbrace{1, 1, \ldots, 1}_n, \underbrace{0, 0, \ldots, 0}_n
\]

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

\[
\underbrace{2, 2, \ldots, 2}_n, \underbrace{0, 0, \ldots, 0}_n
\]

Thus every debit is \( O(1) \), which is why extract costs only \( O(1) \).
Ghost piggy banks
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.

\[
\text{PiggyBank-Pay}
\]
\[
PiggyBank_{P,Q} \quad n \ast \quad k \quad \Rightarrow
\]
\[
PiggyBank_{P,Q} \quad (n - k)
\]

\[
\text{PiggyBank-Persist}
\]
\[
persistent(PiggyBank_{P,Q} \quad n)
\]
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 $Q$. 

\[
\text{PiggyBank-CREATE} \quad P \, n \Rightarrow \text{PiggyBank}_{P,Q} \, n
\]
Piggy Banks: Forcing the Bank

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.
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.

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?
Reversing a list and converting it to a stream:

```ocaml
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)
```

```ocaml
let rec revl_append (l : 'a list) (c : 'a cell) : 'a cell = 
  match l with 
  | x :: l -> revl_append l (Cons (x, Thunk.create @@ fun () -> c))
```

```
let rec revl (l : 'a list) : 'a stream = 
  Thunk.create @@ fun () -> revl_append l Nil
```
The **debit subsumption** judgement

\[ \vec{d}_1 \leq \vec{d}_2 \]

can be defined as follows:

\[ \forall i. \quad \sum (\text{take } i \vec{d}_1) \leq \sum (\text{take } i \vec{d}_2) \]

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 $K$; the rest have debt 0.

Elements are **inserted** in the rear, **extracted** from the front.
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 $O(1)$.

```
+-------------------------+-------------------------+
| fs                      | rs                      |
| K          K ... K 0     |                          |
+-------------------------+-------------------------+
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.

A **deep payment**, possibly involving a thunk **that does not even exist yet** in memory!

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

Rebalance involves

\[
\begin{align*}
\text{fs} & \quad 0 \quad \cdots \quad 0 \\
\text{rs} & \\
\end{align*}
\]

The queue is unbalanced.

\[|\text{fs}| = n \land |\text{rs}| = n + 1\]

Reverse and append the rear list to the front stream.

\[
\begin{align*}
\text{fs} & \quad A \quad \cdots \quad A \quad \frac{K+}{Rn} \quad 0 \quad \cdots \quad 0 \\
\text{rs} & \\
\end{align*}
\]

Redistribute debits by adding \(R\) to the first \(n\) debits.

\[
\begin{align*}
\text{fs} & \quad A + \quad A + \\
& \quad \frac{R}{\cdots R} \quad K \quad 0 \quad \cdots \quad 0 \\
\text{rs} & \\
\end{align*}
\]

Moving debits towards the left is safe: it requires earlier payments.
The banker’s queue admits a simple specification in Iris$^\text{\$}$. 

\[
\begin{align*}
\text{Banker-Persistent} & \quad \text{Banker-Empty} \\
\text{persistent}(BQueue \ q \ \vec{x}) & \quad \{\text{\$E}\} \ text{empty} () \ \{\lambda q. \ BQueue \ q \ []\}
\end{align*}
\]

Queues are persistent. Creation costs $O(1)$.
Insertion and extraction cost $O(1)$.

**Banker-Snoc**

\[
\{ \$S \ * \ BQueue \ q \ \vec{x} \} \ \text{snoc} \ q \times \{ \lambda q'. \ BQueue \ q' \ (\vec{x} \ ++ \ [x]) \}
\]

**Banker-Extract**

\[
\{ \$X \ * \ BQueue \ q \ (x :: \vec{x}) \ * \ \ell \}
\]

\[
\text{extract} \ q
\]

\[
\{ \lambda (x', q'). \ \neg x' = x^\top \ * \ BQueue \ q' \ \vec{x} \ * \ \ell \}
\]

Extraction requires a token $\ell$.

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