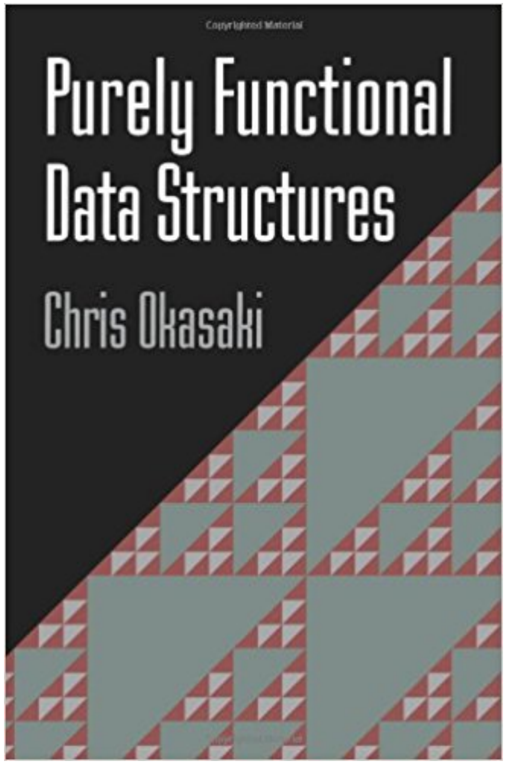
Thunks and Debits in Iris with Time Credits

F. Pottier A. Guéneau J.-H. Jourdan G. Mével

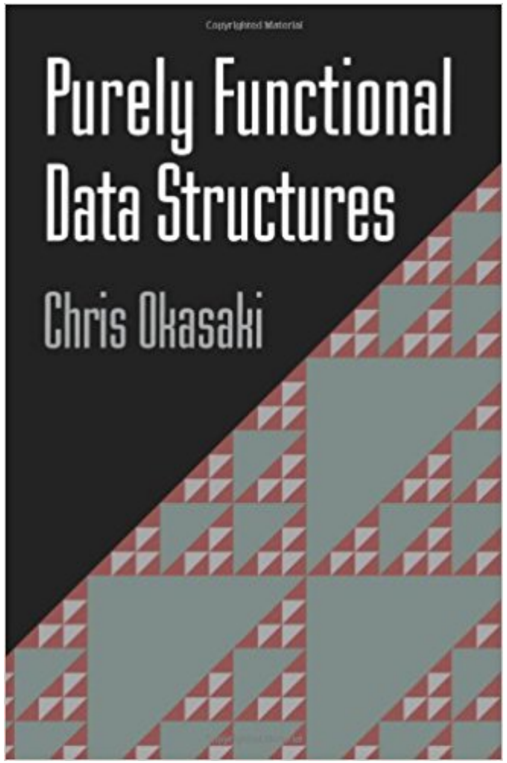
Inria / Laboratoire Méthodes Formelles

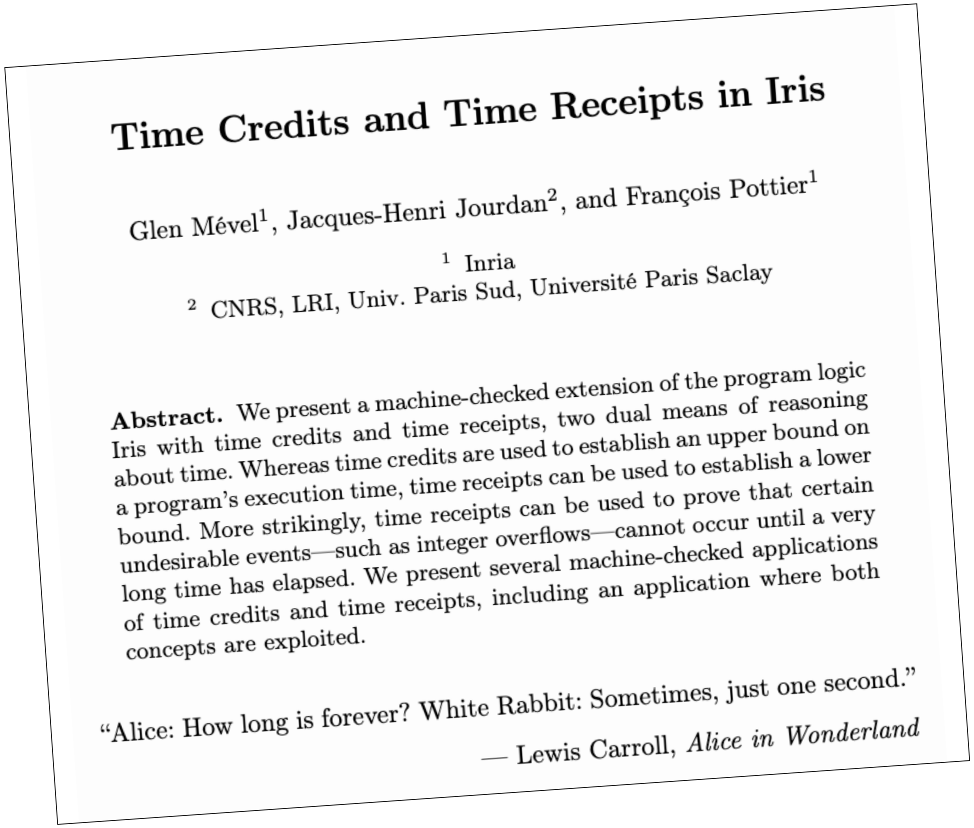
March 16, 2023

From Purely Functional Data Structures to Iris$^{\$}$



From Purely Functional Data Structures to Iris$^{\$}$





Time Credits and Time Receipts in Iris

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

[1] Inria

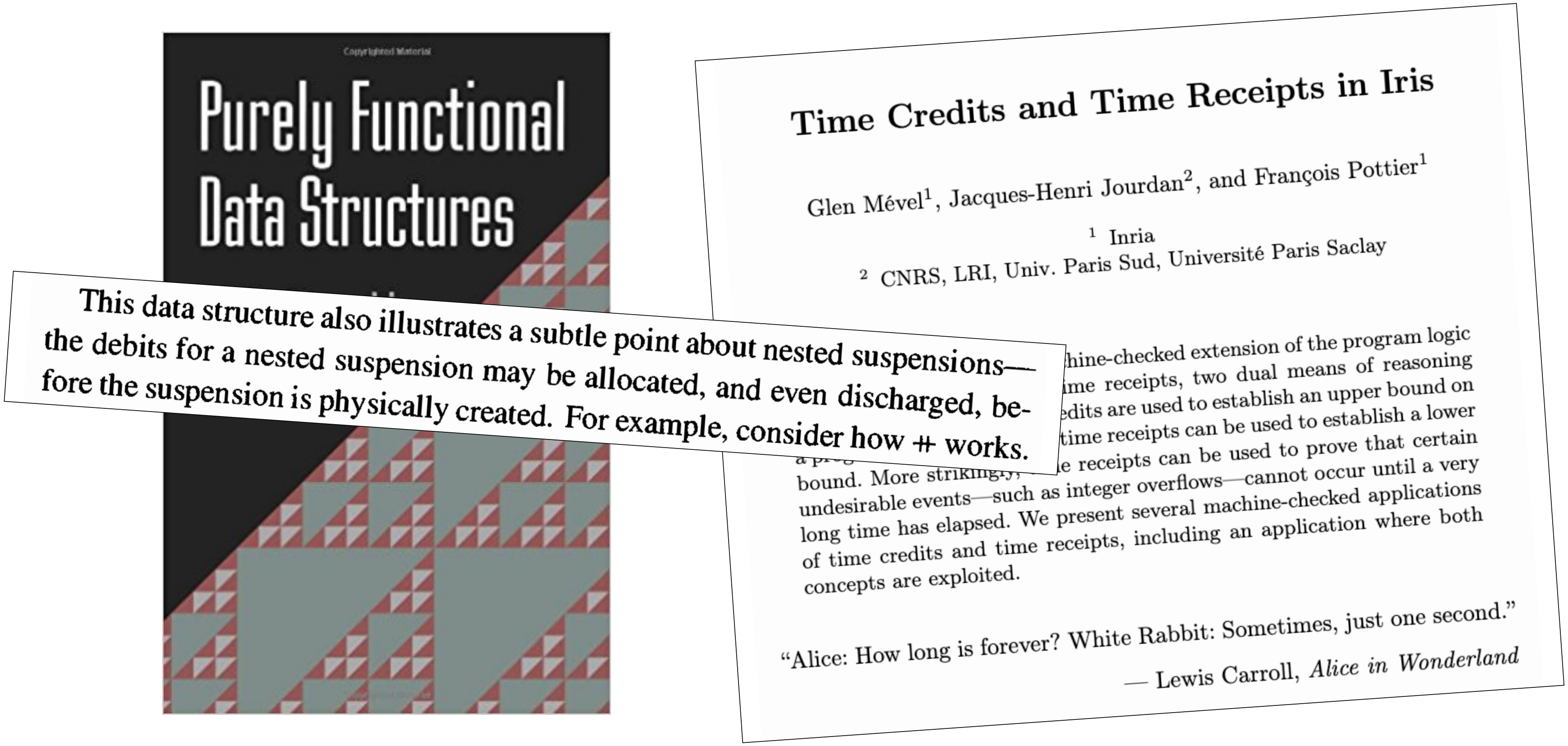
[2] CNRS, LRI, Univ. Paris Sud, Université Paris Saclay

Abstract. We present a machine-checked extension of the program logic Iris with time credits and time receipts, two dual means of reasoning about time. Whereas time credits are used to establish an upper bound on a program's execution time, time receipts can be used to establish a lower bound. More strikingly, time receipts can be used to prove that certain undesirable events—such as integer overflows—cannot occur until a very long time has elapsed. We present several machine-checked applications of time credits and time receipts, including an application where both concepts are exploited.

"Alice: How long is forever? White Rabbit: Sometimes, just one second."

— Lewis Carroll, *Alice in Wonderland*

From Purely Functional Data Structures to Iris$^{\$}$



Copyrighted Material

Purely Functional Data Structures

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.

Time Credits and Time Receipts in Iris

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

[1] Inria

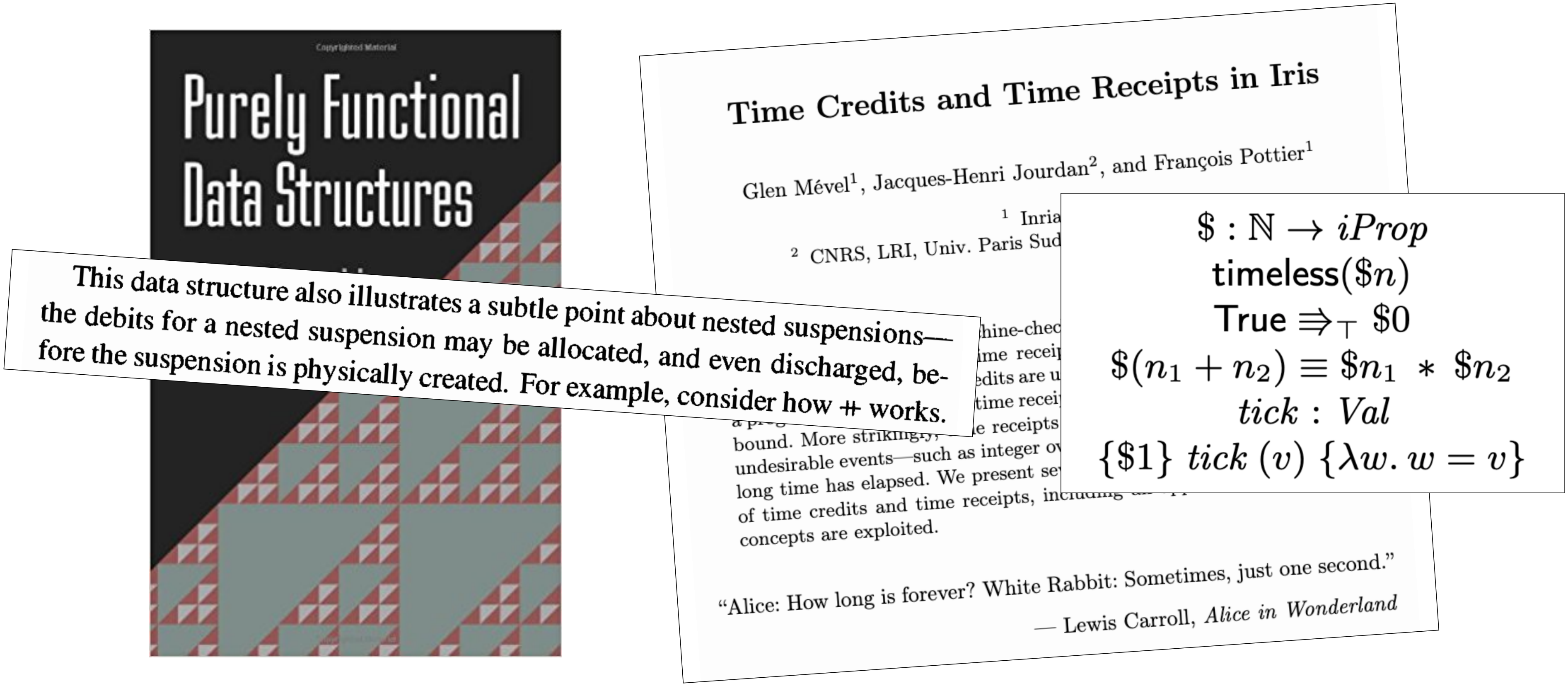
[2] CNRS, LRI, Univ. Paris Sud, Université Paris Saclay

hine-checked extension of the program logic
ime receipts, two dual means of reasoning
edits are used to establish an upper bound on
time receipts can be used to establish a lower
a prog
e receipts can be used to prove that certain
bound. More strikingly,
undesirable events—such as integer overflows—cannot occur until a very long time has elapsed. We present several machine-checked applications of time credits and time receipts, including an application where both concepts are exploited.

"Alice: How long is forever? White Rabbit: Sometimes, just one second."

— Lewis Carroll, *Alice in Wonderland*

From Purely Functional Data Structures to Iris$^\$$



Copyrighted Material

Purely Functional Data Structures

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.

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

hine-chec

ime receip

edits are u

time recei

e receipts

bound. More strikingly,

undesirable events—such as integer ov

long time has elapsed. We present se

of time credits and time receipts, including

concepts are exploited.

"Alice: How long is forever? White Rabbit: Sometimes, just one second."

— Lewis Carroll, *Alice in Wonderland*

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

$$\mathsf{timeless}(\$n)$$

$$\mathsf{True} \Rrightarrow_{\top} \$0$$

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

$$tick : Val$$

$$\{\$1\}\ tick\ (v)\ \{\lambda w.\ w = v\}$$

The Code that We Specify and Verify

Three Layers

The code discussed in this talk is organized in several layers:

- thunks, also known as suspensions;
- purely functional data structures, such as
  - streams, also known as lazy lists;
  - the banker's queue, which use streams.

A very small amount of code with subtle time complexity properties.

Thunks: OCaml Code

Thunks fit in 5 lines of code.

```
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
```

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

It is viewed as a persistent data structure by the user.

No lock. Only two colors. Reentrancy is a programming error.

Streams: OCaml Code

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 a persistent data structure.

Streams: OCaml Code

Reversing a list and converting it to a stream:

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

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

Concatenating two streams:

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

The Banker’s Queue: OCaml Code

The banker’s queue fits in 10 lines of code.

```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 }
```

It is a persistent data structure.

Every operation has (amortized) constant time complexity.

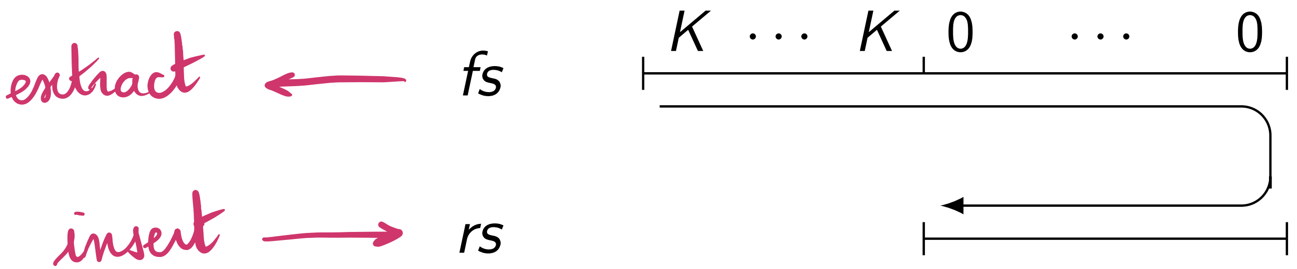
The Banker's Queue: Informal Analysis

The Banker's Queue: Debit Invariant

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. our invariant



extract ← *fs* $K \cdots K \; 0 \cdots 0$

insert → *rs*

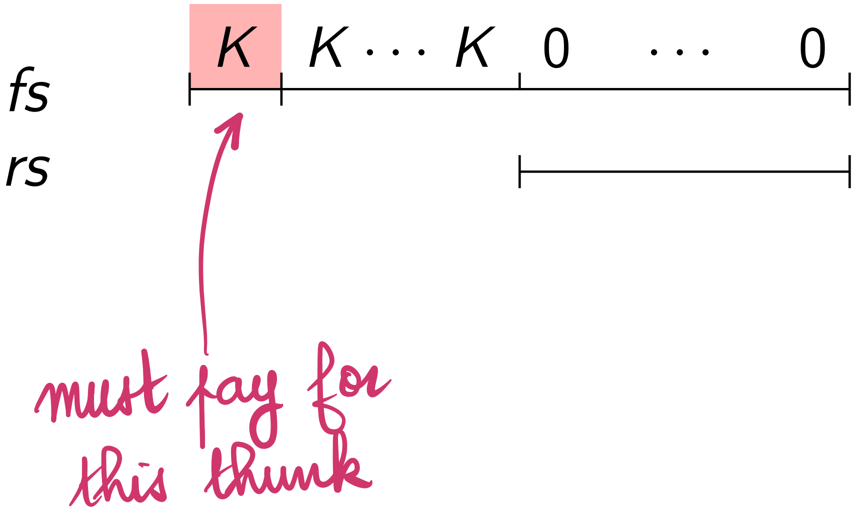
Elements are inserted in the rear, extracted from the front.

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 $O(1)$.

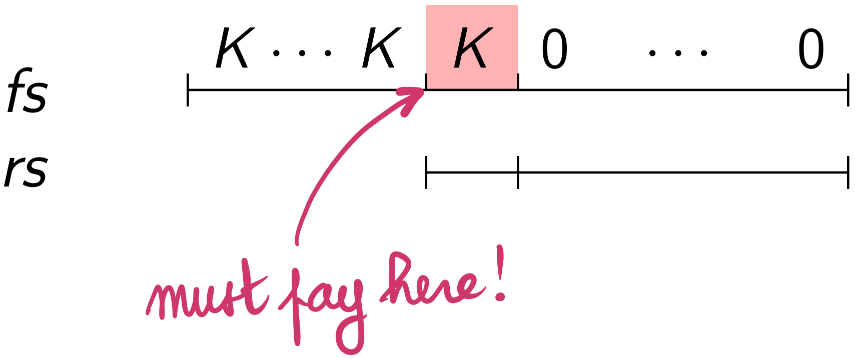


The Banker's Queue: Insertion

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,

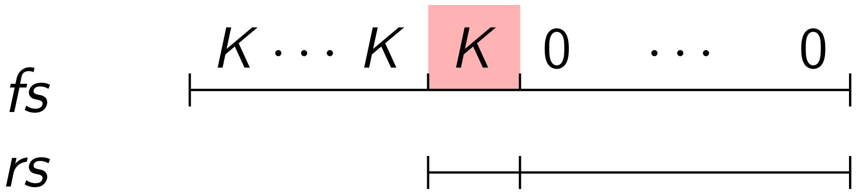
possibly involving a thunk that does not even exist yet in memory!

The Banker's Queue: Insertion

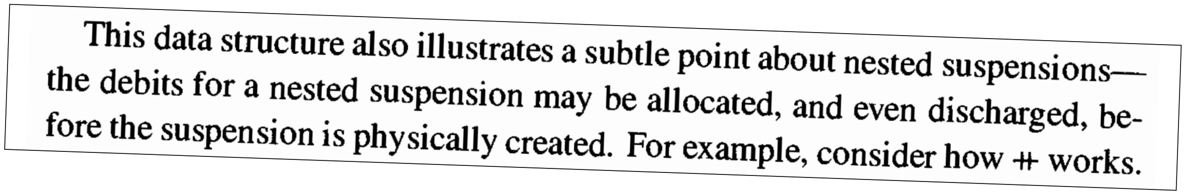
If $|fs| > |rs|$, then insertion does not require rebalancing.

Insertion actually consumes O(1) time,

and requires paying K to maintain the invariant.



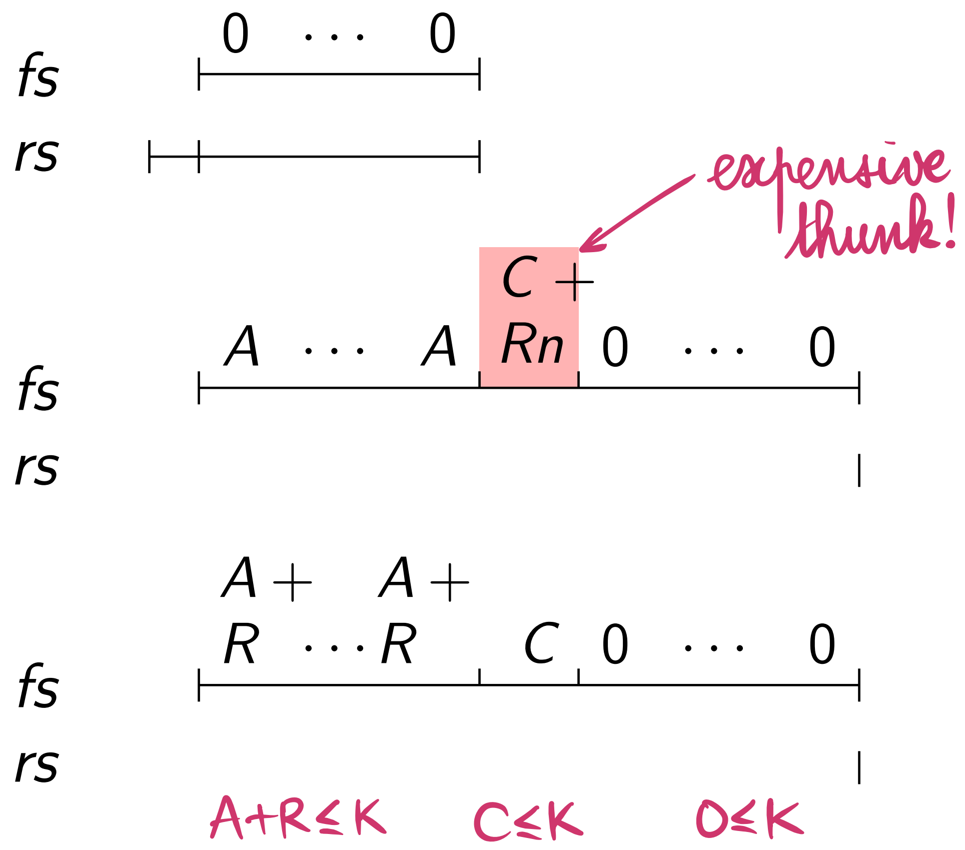
A deep payment, possibly involving a thunk that does not even exist yet in memory!



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.

The Banker's Queue: Rebalancing

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: Formal Analysis

The Banker's Queue: Public Interface (1)

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

$$\frac{\textsc{Banker-Persistent}}{\mathsf{persistent}(\mathit{BQueue}\ p\ q\ \mathit{xs})}$$

$$\textsc{Banker-Empty}$$
$$\{\$13\}\ \mathit{empty}\ ()\ \mathsf{returns}\ (\exists q)\ q\ \{\mathit{BQueue}\ p\ q\ []\}$$

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

The Banker's Queue: Public Interface (2)

Insertion and extraction cost $O(1)$.

$$\begin{array}{l}\textsc{Banker-Snoc}\\ \quad\{\$136 \ast \mathit{BQueue}\ p\ q\ \mathit{xs}\}\\ \quad\quad \mathit{snoc}\ q\ x\\ \text{returns } (\exists q')\ q'\ \{\mathit{BQueue}\ p\ q'\ (\mathit{xs} \mathbin{++} [x])\}\end{array}$$

$$\begin{array}{l}\textsc{Banker-Extract}\\ \quad\{\$165 \ast \mathit{BQueue}\ p\ q\ (x :: \mathit{xs}) \ast \lightning^{\infty}_{p}\}\\ \quad\quad \mathit{extract}\ q\\ \text{returns } (\exists q')\ (x, q')\ \{\mathit{BQueue}\ p\ q'\ \mathit{xs} \ast \lightning^{\infty}_{p}\}\end{array}$$

Extraction requires a token $\lightning^{\infty}_{p}$ where p is a "non-atomic pool".

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

The Banker's Queue: Internal Definition

The proof states the debit invariant

and relies on high-level reasoning rules for streams.

$$
\begin{aligned}
K &\triangleq 60 \\
\mathit{bqueueDebits}\ \mathit{nf}\ \mathit{nr} &\triangleq K^{\mathit{nf}-\mathit{nr}} \mathbin{++} 0^{\min(\mathit{nf},\mathit{nr})+1} \quad \|
\end{aligned}
$$

desired sequence of debits.

$$
\begin{aligned}
\mathit{BQueueRaw}\ p\ q\ \mathit{fs}\ \mathit{rs} &\triangleq \\
&\exists s, h, l. \left\{ \begin{array}{l} \ulcorner q = (|\mathit{fs}|, s, |\mathit{rs}|, l) \urcorner \ * \\ \mathit{Stream}\ p\ h\ s\ \underline{(\mathit{bqueueDebits}\ |\mathit{fs}|\ |\mathit{rs}|)}\ \mathit{fs}\ * \\ \mathit{List}\ l\ \mathit{rs} \end{array} \right.
\end{aligned}
$$

Stream is indexed with a sequence of debits.

$$
\begin{aligned}
\mathit{BQueue}\ p\ q\ \mathit{xs} &\triangleq \\
&\exists \mathit{fs}, \mathit{rs}. \left\{ \begin{array}{l} \mathit{BQueueRaw}\ p\ q\ \mathit{fs}\ \mathit{rs}\ * \\ \ulcorner \mathit{xs} = \mathit{fs} \mathbin{++} \mathit{rev}\ \mathit{rs} \wedge |\mathit{rs}| \le |\mathit{fs}| \urcorner \end{array} \right.
\end{aligned}
$$

Streams: A Key Reasoning Rule

The predicate *Stream* is indexed with a sequence of debits *ds*.

The following ghost update allows:

- paying *m* credits (in depth); and
- moving debits towards the left.

Stream-Forward-Debt

$$\ulcorner (m)\ ds_1 \leq ds_2\ (n) \urcorner \mathrel{-\!\!*}$$
$$\mathit{Stream}\ p\ h\ s\ ds_1\ xs\ *\ \$m \Rrightarrow_{\mathcal{E}}$$
$$\mathit{Stream}\ p\ h\ s\ ds_2\ xs$$

Streams: Debit Subsumption

The debit subsumption judgement (whose definition is not shown)

$$(m)\ ds_1 \leq ds_2\ (n)$$

implies

$$\forall i. \quad \sum(take\ i\ ds_1) \leq m + \sum(take\ i\ ds_2)$$

By paying m now,
one reduces the apparent cost of forcing the stream down to depth i
by at most m,
and this holds for every depth i.

When m is zero, this judgement moves debits towards the left.

Questions

How can one construct these high-level reasoning rules, in Iris$^{\$}$, by starting from first principles?

Let us now descend to the level of thunks.

Let us give formal statements of Okasaki's reasoning rules and see how to construct a predicate *Thunk* that satisfies these rules.

Thunks: a Desired API

Thunks: Abstract Predicate

We would like a persistent assertion $\mathit{Thunk}\ \underset{\text{address of the thunk}}{t}\ \overset{\text{debit}}{n}\ \overset{\text{postcondition}}{\phi}$.

More parameters are needed: $p, \mathcal{E}, R$, but are not discussed in this talk.
Side conditions on namespaces and masks are omitted in the following slides.

Thunks: Creation

Creation costs $O(1)$.

$$\begin{array}{c}\textsc{Thunk-Create}\\ \{\$3 \ast \mathit{isAction}\ f\ \mathbf{n}\ R\ \phi\}\\ \mathit{create}\ f\\ \text{returns } (\exists t)\ t\ \{\mathit{Thunk}\ p\ \mathcal{F}\ t\ \mathbf{n}\ R\ \phi\}\end{array}$$

$\mathit{isAction}\ f\ \ n\ R\ \phi$ denotes the one-shot triple

$$\mathbf{1}\ \{R\ \ast\ \$n\}\ f()\ \text{returns}\ (\exists v)\ v\ \{R\ \ast\ \Box\ \phi\ v\}$$

So, if the suspended computation costs n then the thunk has debit n.

Thunks: Debt Management

One can over-approximate an apparent debt.

$$\begin{array}{l}\textsc{Thunk-Increase-Debt}\\ \ulcorner n_1 \leq n_2 \urcorner \mathrel{-\!\!*} \mathit{Thunk}\ p\ \mathcal{F}\ t\ n_1\ R\ \phi \mathrel{-\!\!*} \\ \qquad\qquad\qquad \mathit{Thunk}\ p\ \mathcal{F}\ t\ n_2\ R\ \phi\end{array}$$

$$\begin{array}{l}\textsc{Thunk-Pay}\\ \mathit{Thunk}\ p\ \mathcal{F}\ t\ n\ R\ \phi \ *\ \$k \Rrightarrow_{\mathcal{E}} \\ \mathit{Thunk}\ p\ \mathcal{F}\ t\ (n-k)\ R\ \phi\end{array}$$

One can pay to reduce an apparent debt.

Thunks: Forcing

Provided the debt is zero, forcing costs $O(1)$.

$$
\begin{array}{l}
\textsc{Thunk-Force} \\
\qquad \{\, \mathit{Thunk}\ p\ \mathcal{F}\ t\ 0\ R\ \phi \ \ast\ \$11 \ \ast\ \lightning^{\mathcal{F}}_{p} \ \ast\ \ R \} \\
\qquad\qquad\qquad\qquad \mathit{force}\ t \\
\text{returns } (\exists v)\ v\ \{\, \mathit{ThunkVal}\ t\ v \ \ast\ \Box\, \phi\ v \ \ast\ \lightning^{\mathcal{F}}_{p} \ \ast\ \ R \}
\end{array}
$$

A token $\lightning^{\mathcal{F}}_{p}$ is required.

A forced-thunk witness is produced.

Thunks: Forcing an Already-Forced Thunk

Forcing a thunk again costs $O(1)$ even if its debt n is nonzero.

$$\textsc{Thunk-Force-Forced}$$

$$\{\mathit{Thunk}\ p\ \mathcal{F}\ t\ n\ R\ \phi\ *\ \mathit{ThunkVal}\ t\ v\ *\ \$11\ *\ \lightning^{\mathcal{F}}_{p}\}$$

$$\mathit{force}\ t$$

$$\text{returns } v\ \{\lightning^{\mathcal{F}}_{p}\}$$

The result v is the value predicted by the *ThunkVal* assertion.

The postcondition $\Box\ \phi\ v$ is not obtained in this case.

Thunks: Consequence Rule

When n_2 is zero, this rule weakens a thunk's postcondition.

When $n_2 \neq 0$, it allows deep payment and strengthens the postcond.

$$\begin{array}{l}
\textsc{Thunk-Consequence} \\
\mathit{Thunk}\ p\ \mathcal{F}_1\ t\ n_1\ R\ \phi \mathrel{-\!\!*} \\
\mathit{isUpdate}\ n_2\ R\ \phi\ \psi\ \Rrightarrow_{\mathcal{E}} \\
\mathit{Thunk}\ p\ \mathcal{F}\ t\ (n_1 + n_2)\ R\ \psi
\end{array}$$

$\mathit{isUpdate}\ n_2\ R\ \phi\ \psi$ denotes the ghost update

$$\forall v.\ (R\ *\ \$n_2\ *\ \Box\,\phi\ v) \Rrightarrow_{\top} (R\ *\ \Box\,\psi\ v)$$

A key rule, missing in our previous work (2019) and difficult to justify.

Construction of Thunks

Construction of Thunks: Overview

We construct the predicate *Thunk* in three stages.

1. basic thunks satisfy all rules except THUNK-CONSEQUENCE;
2. proxy thunks support one application of THUNK-CONSEQUENCE;
3. iterating this construction yields thunks
   that support arbitrarily many applications of this rule.

Piggy banks, a ghost data structure, are used at levels 1 and 2.

Piggy Banks

Piggy Banks: Principles

A piggy bank has two states: pending and forced, described by two Iris assertions $P\ nc$ and Q.

A piggy bank carries an apparent debt n.

Piggy banks involve no code.

Their reasoning rules reflect Okasaki's discipline:

- A target amount nc is fixed upon creation.
- One can pay and reduce the apparent debt in several increments.
- When the debt is 0, breaking the bank yields $\$nc$ which can be used to pay for the transition of $P\ nc$ to Q.
- A piggy bank is persistent.

Piggy Banks: Selected Reasoning Rules

PiggyBank-Create

$$P\ nc \Rrightarrow_{\mathcal{E}} \mathit{PiggyBank}\ nc$$

PiggyBank-Increase-Debt

$$\ulcorner n_1 \leq n_2 \urcorner \mathrel{-\!\!*} \mathit{PiggyBank}\ n_1 \mathrel{-\!\!*} \mathit{PiggyBank}\ n_2$$

PiggyBank-Pay

$$\mathit{PiggyBank}\ n * \$k \Rrightarrow_{\mathcal{E}} \mathit{PiggyBank}\ (n-k)$$

PiggyBank-Break

$$\mathit{PiggyBank}\ 0 * ↯^{\mathcal{F}}_{p} \overset{\text{opening}}{\Rrightarrow_{\mathcal{E}}} \exists nc. \left(\begin{array}{l} ((\triangleright P\ nc * \$nc) \vee \triangleright Q) * \\ (\triangleright Q \underset{\text{closing}}{\Rrightarrow_{\mathcal{E}}} ↯^{\mathcal{F}}_{p}) \end{array} \right)$$

Breaking the bank is a non-atomic process with two distinct steps: opening and closing the bank. A unique token forbids reentrancy.

Paying does not require a token, so is permitted at all times.

Piggy Banks: Internal Definition

Internally, a non-atomic invariant and an atomic invariant are used:

$$\mathit{PiggyBank}\ P\ Q\ A\ p\ N\ n \triangleq$$

$$\exists\, \varphi, \pi, nc.$$

non-atomic

$$\boxed{\begin{array}{l} \exists \mathit{forced}. \\ \quad \boxed{\varphi \mapsto \bullet\, \mathit{forced}} \quad * \\ \quad \text{if } \neg \mathit{forced} \text{ then } P\ nc \text{ else } Q \end{array}}^{N}_{p} \quad *$$

atomic

$$\boxed{\begin{array}{l} \exists \mathit{forced}, ac. \\ \quad \boxed{\varphi \mapsto \circ\, \mathit{forced}} \quad * \quad \boxed{\pi \mapsto \bullet\, ac} \quad * \\ \quad \text{if } \neg \mathit{forced} \text{ then } \$ac \text{ else } \ulcorner nc \leq ac \urcorner \end{array}}^{A} \quad *$$

$$\boxed{\pi \mapsto \circ\, (nc - n)}$$

They agree on the Boolean flag *forced* thanks to a shared ghost cell φ.

Stage 1: Basic Thunks

Basic Thunks: Internal Definition

A basic thunk involves a physical reference t and a piggy bank whose parameters describe the pending and forced states of the thunk.

$$\mathit{BasicThunk}\ p\ \mathcal{F}\ t\ n\ R\ \phi \triangleq$$

$$\exists \delta, N.\ \ulcorner \uparrow N \subseteq \mathcal{F} \urcorner \ *\ t \rightsquigarrow \delta\ *\ \mathit{PiggyBank}$$

pending: $(\lambda nc.\ \exists f.\ \boxed{\delta \mapsto ?}\ *\ t \mapsto \mathit{UNEVALUATED}\ f\ *\ \mathit{isAction}\ f\ nc\ R\ \phi)$

forced: $(\exists v.\ \boxed{\delta \mapsto v}\ *\ t \mapsto \mathit{EVALUATED}\ v\ *\ \Box\ \phi\ v)$

$$\mathit{ThunkPayment}\ p\ N\ n$$

$$\mathit{ThunkVal}\ t\ v \triangleq$$

$$\exists \delta.\quad t \rightsquigarrow \delta\ *\ \boxed{\delta \mapsto v}$$

Basic thunks satisfy the desired rules except THUNK-CONSEQUENCE.

Why Consequence is Tricky

A reminder:

$$\begin{array}{l}\textsc{Thunk-Consequence}\\ \mathit{Thunk}\ p\ \mathcal{F}_1\ t\ n_1\ R\ \phi\ \twoheadrightarrow\\ \mathit{isUpdate}\ n_2\ R\ \phi\ \psi\ \Rrightarrow_{\mathcal{E}}\\ \mathit{Thunk}\ p\ \mathcal{F}\ t\ (n_1+n_2)\ R\ \psi\end{array}$$

Supporting this rule seems tricky, because

- it appears to set a new postcondition and a new target amount,
- yet these are fixed at construction time
  by the invariants of piggy banks and basic thunks;
- furthermore, the old and new views of the thunk must coexist.

Stage 2: Proxy Thunks

An Intuition

Applying Thunk-Consequence to a thunk t
is almost like constructing a new thunk t'
via the expression *create* $(\lambda().\ \mathit{force}\ t)$.

If we actually created a new thunk,
we could set a new target amount and a new postcondition.

We need Thunk-Consequence to be a ghost update
(this is absolutely necessary to allow deep payment)
and to not actually create a new thunk...

but it can create a new piggy bank.

Proxy Thunks: Creation Rule

Based on this idea, we create a variant of the consequence rule that transforms a basic thunk into a proxy thunk:

Proxy-Create

$$\dfrac{\mathcal{F}_1 \uplus \uparrow N \subseteq \mathcal{F}}{\begin{array}{l} \mathit{BasicThunk}\ p\ \mathcal{F}_1\ t\ n_1\ R\ \phi \mathrel{-\!\!*} \\ \mathit{isUpdate}\ n_2\ R\ \phi\ \psi \Rrightarrow_{\mathcal{E}} \\ \mathit{ProxyThunk}\ p\ \mathcal{F}\ t\ (n_1 + n_2)\ R\ \psi \end{array}}$$

Proxy Thunks: Internal Definition

A proxy thunk is just a basic thunk together with a new piggy bank.

$$
\begin{aligned}
& \mathit{ProxyThunk}\ p\ \mathcal{F}\ t\ n\ R\ \phi \triangleq \\
& \quad \exists n_1, n_2, \phi, \mathcal{F}_1, N.\ \ \ulcorner \mathcal{F}_1 \uplus \uparrow N \subseteq \mathcal{F} \urcorner\ * \\
& \qquad \mathit{Thunk}\ p\ \mathcal{F}_1\ t\ n_1\ R\ \phi\ * \\
& \qquad \mathit{PiggyBank} \\
& \qquad\quad (\lambda nc. \ulcorner nc = n_1 + n_2 \urcorner\ *\ \mathit{isUpdate}\ n_2\ R\ \phi\ \psi) \\
& \qquad\quad (\exists v. \mathit{ThunkVal}\ t\ v\ *\ \Box\ \psi\ v) \\
& \qquad\quad \mathit{ThunkPayment}\ p\ N\ n
\end{aligned}
$$

Thunks: a Common API

Proxy thunks enjoy the same reasoning rules as basic thunks.

The “common thunk API”,

- all rules except Thunk-Create and Thunk-Consequence,

is the same for basic thunks and proxy thunks.

Stage 3: Thunks

Iterating the Construction

We have built proxy thunks on top basic thunks.

The construction works on top of an arbitrary flavor of thunks
provided they satisfy the common API,
and produces a new flavor that again satisfy the common API.

Iterating the construction allows building

- basic thunks,
- proxy thunks that wrap basic thunks,
- proxy thunks that wrap proxy thunks that wrap basic thunks, etc.

The fixed point satisfies the common API plus Thunk-Create and Thunk-Consequence, that is, the full desired API.

Thunks: Internal Definition

Here is a possible definition of the greatest fixed point:

$$
\begin{array}{l}
\mathit{Thunk}\ p\ \mathcal{F}\ t\ n\ R\ \phi \triangleq \\
\quad \exists \mathit{Thunk}.\ N,\ d,\ \mathcal{F}'. \\
\qquad \mathit{Thunk}\ \text{is persistent}\ \ * \\
\qquad \mathit{Thunk}\ \text{satisfies the common thunk API}\ \ * \\
\qquad \ulcorner \forall d'.\ d < d' \Rightarrow \mathcal{F}' \mathbin{\#} {\uparrow}(N \cdot d') \urcorner\ \ * \\
\qquad \ulcorner \mathcal{F}' \subseteq {\uparrow}N \subseteq \mathcal{F} \urcorner\ \ * \\
\qquad \mathit{Thunk}\ p\ \mathcal{F}'\ t\ n\ R\ \phi
\end{array}
$$

An inductive definition is also possible.

Conclusion

Conclusion

A new result in a beautiful line of work:

- Okasaki (1999)
- Danielsson (2008)
- Mével et al. (2019)

In the paper:

- forbidding reentrancy by indexing thunks with heights;
- specs for operations on streams; machine-checked proofs; etc.

Future work:

- engineering work required to make Iris$^{\$}$ more user-friendly.

Backup Slides

Piggy Banks

Piggy Banks: the Ghost Cell φ

The cell is owned by exactly two participants.

One token suffices to know the content of the cell.

The two participants always agree on the content:

$$\boxed{\varphi \mapsto \bullet\, \mathit{forced}_1} \ast \boxed{\varphi \mapsto \circ\, \mathit{forced}_2} \vdash \ulcorner \mathit{forced}_1 = \mathit{forced}_2 \urcorner$$

The two participants must cooperate to update the cell:

$$\boxed{\varphi \mapsto \bullet\, \mathit{forced}} \ast \boxed{\varphi \mapsto \circ\, \mathit{forced}} \Rrightarrow \boxed{\varphi \mapsto \bullet\, \mathit{forced}'} \ast \boxed{\varphi \mapsto \circ\, \mathit{forced}'}$$

Piggy Banks: the Ghost Cell π

There is one authoritative view and many fragmentary views of the cell.

A fragment $\boxed{\pi \mapsto \circ\, k}$ is a witness that the true value is at least k:

$$\boxed{\pi \mapsto \bullet\, ac} * \boxed{\pi \mapsto \circ\, k} \vdash \ulcorner k \leq ac \urcorner$$

This is sound because updates must be monotonic:

$$\boxed{\pi \mapsto \bullet\, ac} \Rrightarrow \boxed{\pi \mapsto \bullet\, (ac + k)}$$

An accurate witness can be created at any time:

$$\boxed{\pi \mapsto \bullet\, ac} \vdash \boxed{\pi \mapsto \circ\, ac}$$

Thunks

Thunks: Forced-Thunk Witnesses

ThunkVal-Confront

$$\mathit{ThunkVal}\ t\ v_1 \ast \mathit{ThunkVal}\ t\ v_2 \mathrel{-\!\!\ast} \ulcorner v_1 = v_2 \urcorner$$

Height-Indexed Thunks

Height-Indexed Thunks: Selected Rules

A thunk can force thunks of lesser height only.

HThunk-Create

$$\{\$3 \ast \mathit{isAction}\ f\ n\ (\lightning^{h}_{p})\ \phi\}$$

$$\mathit{create}\ f$$

$$\text{returns } (\exists t)\ t\ \{\mathit{HThunk}\ p\ h\ t\ n\ \phi\}$$

HThunk-Inc-Height-Debt

$$\ulcorner h_1 \leq h_2 \urcorner \mathrel{-\!\!\ast} \ulcorner n_1 \leq n_2 \urcorner \mathrel{-\!\!\ast}$$

$$\mathit{HThunk}\ p\ h_1\ t\ n_1\ \phi \mathrel{-\!\!\ast}$$

$$\mathit{HThunk}\ p\ h_2\ t\ n_2\ \phi$$

HThunk-Force

$$\left\{\mathit{HThunk}\ p\ h\ t\ 0\ \phi \ast \$11 \ast \lightning^{h'}_{p} \ast \ulcorner h < h' \urcorner\right\}$$

$$\mathit{force}\ t$$

$$\text{returns } (\exists v)\ v\ \{\Box\ \phi\ v \ast \mathit{ThunkVal}\ t\ v \ast \lightning^{h'}_{p}\}$$

This height-based discipline is simpler than the mask-based discipline shown earlier.

Streams

Streams: the Predicate *Stream*

Here is the general form of the predicate *Stream*:

$$\mathit{Stream}\ p\ h\ s\ \mathit{ds}\ \mathit{xs}$$

Streams: Definition

The definition is straightforward:

$$
\begin{aligned}
\mathit{Stream}\ p\ h\ s\ []\ \mathit{xs} &\triangleq \text{False} \\
\mathit{Stream}\ p\ h\ s\ (d :: \mathit{ds})\ \mathit{xs} &\triangleq \\
&\quad \mathit{HThunk}\ p\ h\ s\ d\ (\lambda c. \mathit{StreamCell}\ p\ h\ c\ \mathit{ds}\ \mathit{xs}) \\
\mathit{StreamCell}\ p\ h\ c\ \mathit{ds}\ [] &\triangleq \ulcorner c = \mathit{Nil} \urcorner * \ulcorner \mathit{ds} = [] \urcorner \\
\mathit{StreamCell}\ p\ h\ c\ \mathit{ds}\ (x :: \mathit{xs}) &\triangleq \\
&\quad \exists s.\ \ulcorner c = \mathit{Cons}(x, s) \urcorner * \mathit{Stream}\ p\ h\ s\ \mathit{ds}\ \mathit{xs}
\end{aligned}
$$

Streams: Creation

Constructing a stream costs $O(1)$.

$$\begin{array}{l}\textsc{Stream-Create}\\ \qquad \{\$5 \ast \mathit{isCellAction}\ p\ h\ d\ e\ \mathit{ds}\ \mathit{xs}\}\\ \qquad\qquad\qquad \mathit{create}\ (\lambda().e)\\ \text{returns } (\exists s)\ s\ \{\mathit{Stream}\ p\ h\ s\ (d :: \mathit{ds})\ \mathit{xs}\}\end{array}$$

isCellAction p h d e ds xs denotes the one-shot triple

$$\mathbf{1}\ \{\lightning_p^h \ast \$d\}\ e \text{ returns } (\exists c)\ c\ \{\mathit{StreamCell}\ p\ h\ c\ \mathit{ds}\ \mathit{xs} \ast \lightning_p^h\}$$

Streams: Forcing

Provided the head debit is zero, forcing a stream costs $O(1)$.

Stream-Force

$$\left\{\begin{array}{l} \mathit{Stream}\ p\ h\ s\ (0 :: \mathit{ds})\ \mathit{xs} \quad * \\ \$11\ *\ \lightning_p^{h'}\ *\ \ulcorner h < h' \urcorner \end{array}\right\}$$

$$\mathit{force}\ s$$

$$\text{returns } (\exists c)\ c \left\{\begin{array}{l} \mathit{StreamCell}\ p\ h\ c\ \mathit{ds}\ \mathit{xs} \quad * \\ \mathit{ThunkVal}\ s\ c\ *\ \lightning_p^{h'} \end{array}\right\}$$

Streams: Specifications of *revl* and *append*

revl constructs one expensive thunk followed with n cheap thunks.

$$\begin{array}{l}\textsc{Stream-Revl}\\ \quad\{\mathit{List}\ l\ \mathit{xs}\ \ast\ \$13\ \ast\ \ulcorner n = |\mathit{xs}|\urcorner\}\\ \qquad\qquad \mathit{revl}\ l\\ \text{returns } (\exists s)\ s\ \{\mathit{Stream}\ p\ h\ s\ (19n :: 0^n)\ (\mathit{rev}\ \mathit{xs})\}\end{array}$$

$$\begin{array}{l}\textsc{Stream-Append}\\ \quad\{\mathit{Stream}\ p\ h\ s_1\ \mathit{ds}_1\ \mathit{xs}_1\ \ast\ \mathit{Stream}\ p\ h\ s_2\ \mathit{ds}_2\ \mathit{xs}_2\ \ast\ \$8\}\\ \qquad\qquad \mathit{append}\ s_1\ s_2\\ \text{returns } (\exists s)\ s\ \{\mathit{Stream}\ p\ (h+1)\ s\ (\mathit{ds}_1 \bowtie \mathit{ds}_2)\ (\mathit{xs}_1 \mathbin{++} \mathit{xs}_2)\}\end{array}$$

append joins ds_1 and ds_2 using the debit join operator $\bowtie$.

Streams: Definition of Debit Join

Debit join $\bowtie$ can be defined as follows:

$$\begin{aligned}(ds_1 \mathbin{++} [d_1]) \bowtie (d_2 :: ds_2) \quad &\triangleq \\ &map\ (A + \_)\ ds_1 \ \mathbin{++}\ (A + d_1 + B + d_2)\ ::\ ds_2\end{aligned}$$

where $A \triangleq 30$ and $B \triangleq 11$.

Streams: Debit Subsumption: Definition

$$\textsc{Sub-Nil}\quad \frac{n \leq m}{(m)\ [] \leq []\ (n)}$$

$$\textsc{Sub-Cons}\quad \frac{d_1 \leq m + d_2 \qquad (m + d_2 - d_1)\ ds_1 \leq ds_2\ (n)}{(m)\ d_1 :: ds_1 \leq d_2 :: ds_2\ (n)}$$

Streams: Debit Subsumption: Reasoning Rules

$$\textsc{Sub-Variance}\quad \frac{(m)\ ds_1 \leq ds_2\ (n) \qquad m \leq m' \qquad n' \leq n}{(m')\ ds_1 \leq ds_2\ (n')}$$

$$\textsc{Sub-Trans}\quad \frac{(m_1)\ ds_1 \leq ds_2\ (n_1) \qquad (m_2)\ ds_2 \leq ds_3\ (n_2)}{(m_1 + m_2)\ ds_1 \leq ds_3\ (n_1 + n_2)}$$

$$\textsc{Sub-Append}\quad \frac{(m)\ ds_1 \leq ds_2\ (n) \qquad (n)\ ds_1' \leq ds_2'\ (k)}{(m)\ ds_1 \mathbin{++} ds_1' \leq ds_2 \mathbin{++} ds_2'\ (k)}$$

$$\textsc{Sub-Add-Slack}\quad \frac{(m)\ ds_1 \leq ds_2\ (n)}{(m + k)\ ds_1 \leq ds_2\ (n + k)}$$

$$\textsc{Sub-Repeat}\quad \frac{d_1 \leq d_2}{(0)\ d_1^n \leq d_2^n\ (n \times (d_2 - d_1))}$$

$$\textsc{Sub-Refl}\quad (m)\ ds \leq ds\ (m)$$

The Banker's Queue

The Banker's Queue: Specification of *check*

Banker-Check

$$\{\$48 \ * \ \mathit{BQueueRaw}\ p\ q\ \mathit{fs}\ \mathit{rs} \ * \ \ulcorner |\mathit{rs}| \leq |\mathit{fs}| + 1 \urcorner\}$$

$$\mathit{check}\ q$$

$$\text{returns } (\exists q')\ q'\ \{\mathit{BQueue}\ p\ q'\ (\mathit{fs} \mathbin{++} \mathit{rev}\ \ \mathit{rs})\}$$