Persistent Amortised Analysis, Operationally

Anton Lorenzen

University of Edinburgh, UK

Abstract

Amortised analysis is a technique for proving a combined time bound for a batch of operations on a data structure, even if some of those operations are expensive. But the traditional method of amortised analysis yields incorrect time bounds when the data structure is used persistently. Persistence allows operations to be performed on previous versions of the data structure, which prevents us from amortising expensive restructuring work. In his seminal book, Chris Okasaki showed how to extend amortised analysis to persistent usage. His method works by extending the data structure with *thunks* and performing the analysis with *debits* rather than credits. His argument, that credits are unsound for analysing persistent usage, has become folklore.

In this paper, we provide a new perspective on the role of debits in Okasaki's work. First, we set up an operational semantics of call-by-value lambda calculus with thunks, and show formally that traditional amortised analysis does not work in a persistent setting. Then we show that, contrary to the folklore, amortised analysis in a persistent setting can be performed purely in terms of credits without using debits at all. Finally, we provide a formal semantics for Okasaki's original debit-based approach.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Software and its engineering → Functional languages; Theory of computation → Invariants

Keywords and phrases Lazy Data Structures, Amortised Analysis

Digital Object Identifier 10.4230/LIPIcs.ICALP.2026.185

Category Track B: Automata, Logic, Semantics, and Theory of Programming

Related Version *Full Version*: `https://arxiv.org/abs/2605.09411` [21]

Acknowledgements I am grateful to Kengo Hirata, François Pottier, Wouter Swierstra, and Kim Worrall for their helpful feedback, and to the participants of the AUTOSARD Mid-Term Workshop 2025 for inspiring discussions. Any remaining errors are my own.

1 Introduction

Amortised analysis is a technique for proving a combined time bound for a batch of operations on a data structure, even if some of those operations are expensive. But the traditional method of amortised analysis only yields correct time bounds if data structures are used *sequentially* and fails if data structures are used *persistently*. A data structure is said to be used sequentially if each operation is applied only to the most recent version of the data structure, as returned by the operation preceding it. This happens automatically in mutable data structures, where operations overwrite the data structure and previous versions are no longer accessible.

In contrast, persistent data structures [5, 29, 6] enable the programmer to access or update any previous version of the data structure at no additional cost. This creates a more flexible programming model, but breaks amortised analysis. Consider a version of a persistent data structure that is highly unbalanced. In a traditional amortised analysis, this version is assigned *credits*, which may be spent by a future operation to perform expensive restructuring work and rebalance the data structure [32]. But credits may be spent on this work only if the previous version is not used again. If the previous version is still accessible,

each future operation on it will repeat the expensive restructuring work. Since the stored credits can only pay for one of these operations, traditional amortised analysis yields incorrect time bounds for data structures that are used persistently.

Okasaki [25] proposed that *thunks* can be used to achieve amortised time bounds for data structures that are used persistently. A thunk is a memory cell that may only be updated in a special, deterministic fashion: It starts out in a *lazy* state and may be updated to the *memoised* state. Its value in the memoised state must be a deterministic function of its value in the lazy state, and it should be unobservable whether the update has already happened; a program may ask to read the memoised value, but it will not learn whether an update was performed to obtain it. This makes it possible to share restructuring work between different versions of a data structure. If an operation memoises a thunk, it will be memoised in all previous versions of the data structure as well. This does not change the behaviour of the previous versions, since the update is deterministic and unobservable.

In his seminal book [25], Okasaki adds thunks to many well-known data structures and shows that his variations enjoy good amortised time bounds, even when used persistently. In his analyses, he reasons as if thunks were updated to the memoised state immediately after they are created. However, he allows the program to access the memoised value only once the update has been paid for. The cost of the update is recorded as *debits* on the thunk. A debit is a negative credit that costs one time step to reduce by one, and the thunk can only be accessed once the debits have been reduced to zero.

This technique allows Okasaki to reason about persistent usage as if the data structure was only used sequentially. That is, if a data structure observes a particular time bound in a sequential setting according to his reasoning style, then it also observes the same time bound in a persistent setting. He claims that this is due to his use of debits rather than credits: "although savings can only be spent once, it does no harm to pay off debt more than once" [25, page 59]. This claim has become folklore [28], and led to debits becoming the foundation of subsequent work [4, 14, 24, 26].

Okasaki only provides an informal argument for the correctness of his technique, which has motivated subsequent work formalising his approach. Danielsson [4] implements a variant of Okasaki's debit passing style in Agda. He shows that it is *sound*: the time bound proven using debit passing style is an upper bound on the actual time taken by the program. Pottier et al. [28] formalise this variant of debit passing style in Iris and additionally provide a formal foundation for reasoning about the future evaluation of thunks. However, these approaches differ from (and improve upon) Okasaki's approach: they assume that thunks are updated only when the program first requests their value.

With this improvement, it is clearer why one may reason about persistent usage as if the data structure was only used sequentially. A thunk is only updated once, which makes it possible to store credits on thunks and spend them on the update once it is requested [4, 26, 28, 24]. Pilkiewicz and Pottier [26] and Mével et al. [24] prove that thunks are *monotonic*: When reasoning about a persistent data structure, we may assume that a thunk is in the lazy state and requires further credits before it can be updated, but if the thunk is already memoised, our reasoning will still be valid and simply over-approximate the cost. This insight makes it possible to reason about persistent usage using credits rather than debits. Nevertheless, their high-level reasoning principles are still based on debits.

In this paper, we provide a new perspective on the role of debits in Okasaki's work.

In Section 3, we define the soundness and persistence of a reasoning principle in terms of the operational semantics of call-by-value lambda calculus with thunks. By deriving all of our results directly from an operational semantics, we can state exactly when a reasoning principle works in a persistent setting. As an example, we re-derive the fact that the traditional method of amortised analysis is not persistent.

In Section 4, we give an operational semantics for credit passing style. Based on Danielsson's debit passing style, this method stores *credits on lazy thunks* and updates thunks *when they are forced*. We show that it is sound and persistent in our operational model. Contrary to the folklore, this shows that persistent amortised analysis can be performed purely in terms of credits. Additionally, we present an operational semantics for the related method of credit inheritance and show that it is sound and persistent.

In Section 5, we provide an operational semantics of Okasaki's persistent banker's method. Rather than store credits on lazy thunks, this method stores *debits on memoised thunks* and (conceptually) updates thunks *when they are created*. This models Okasaki's original use of debits as a barrier or "layaway plan", which prevents the program from accessing the result of a thunk before it has been paid for. We show that this is sound and persistent in our operational semantics. Additionally, we present an operational semantics for the related method of debit inheritance and show that it is sound and persistent.

The proofs and additional material are included in the full version [21]. Our earlier workshop paper [20] contains an implementation of the credit-based reasoning principles as a Haskell library for automated testing of time complexity bounds. As part of that work, we have re-analysed the time complexity of all lazy data structures in Okasaki's book [25] using credits. Our library is available online via the creditmonad package and on GitHub.

2 Preliminaries

To model operations on persistent data structures, we consider a lambda calculus with algebraic data types. This is convenient, since the lambda calculus does not include any operations that mutate data structures and so all data structures are persistent by default.

Our syntax is split into values, heap values and expressions. Expressions denote computations, which can return a value and store heap values in the heap.

v, w	$::=$	x, y, z	(variables)	e	$::=$	v	(values)
	$\mid$	a, b, c	(pointers)		$\mid$	hv	(heap values)
	$\mid$	$()$	(unit)		$\mid$	$\mathrm{let}\, x = e\, \mathrm{in}\, e$	(let binding)
	$\mid$	$\mathrm{inl}\, v \mid \mathrm{inr}\, v$	(sum)		$\mid$	$\mathrm{case}\, v\, \{\mathrm{inl}\, x \mapsto e; \mathrm{inr}\, y \mapsto e\}$	(case split)
	$\mid$	(v, v)	(pair)		$\mid$	$\mathrm{let}(x, y) = v\, \mathrm{in}\, e$	(destruct a pair)
	$\mid$	$\lambda x.e$	(lambda)		$\mid$	$v\, w$	(lambda application)
hv	$::=$	$\mathrm{fold}\, v$	(allocate)		$\mid$	$\mathrm{unfold}\, v$	(dereference)
$\mathcal{F}$	$::=$	$\emptyset \mid \mathcal{F}, F(x) = e$			$\mid$	$F\, v$	(function call)

We distinguish between variables x, y, z in computations and pointers a, b, c in the heap. Loosely speaking, sums allow us to choose between values (similar to enumerations or tagged unions), while pairs allow us to combine values (similar to structs). We will not need lambdas (which correspond to anonymous functions) in this article, but include them for completeness. We define top-level functions in the $\mathcal{F}$ environment and call them using $F(v)$. We do not include a fixpoint operator or other recursion scheme, and instead assume that top-level functions may be recursive. We use fold to allocate an expression into the heap and unfold to look up a value from the heap.

Our syntax uses fine-grain call-by-value style [18], where most primitives operate on values. This makes it easier to specify semantic rules. However, one can easily desugar a more traditional syntax into this style by introducing let-bindings where necessary.

2.1 Big-Step Semantics

The (worst-case) time complexity of our language is given by a big-step operational semantics. We write $\Gamma : e \Downarrow_k \Delta : w$ to mean "under heap Γ, the expression e evaluates to value w with updated heap Δ in k steps". A heap is defined as a mapping from pointers to heap values:

$$\Gamma ::= \emptyset \mid \Gamma, a \mapsto hv$$

Our big-step semantics is specified using the inference rules below. Each rule specifies that if the premises above the line hold, then the conclusion below the line also holds. We write $[v/x]$ for the capture-avoiding substitution that replaces all free occurrences of x with v. We may assume that each operation is performed on values of the correct shape: for example, a case-split is performed on either an $\mathrm{inl}\, v$ or an $\mathrm{inr}\, v$ value. If the shape of the value is incorrect, no rule applies and we say that the evaluation is stuck. We only write $\Gamma : e \Downarrow_k \Delta : w$ if the evaluation is not stuck.

$$\frac{}{\Gamma : v \Downarrow_0 \Gamma : v}\ (\text{value}) \qquad \frac{\Gamma : e_i[v/x_i] \Downarrow_k \Delta : w}{\Gamma : \mathrm{case}(\mathrm{in}_i\, v)\, \{\mathrm{in}_l\, x_l \mapsto e_l; \mathrm{in}_r\, x_r \mapsto e_r\} \Downarrow_k \Delta : w}\ (\text{case})$$

$$\frac{a \notin \mathrm{dom}(\Gamma)}{\Gamma : \mathrm{fold}\, v \Downarrow_0 \Gamma, a \mapsto \mathrm{fold}\, v : a}\ (\text{fold}) \qquad \frac{\Gamma : e_1 \Downarrow_k \Delta : v \qquad \Delta : e_2[v/x] \Downarrow_l \Theta : w}{\Gamma : \mathrm{let}\, x = e_1\, \mathrm{in}\, e_2 \Downarrow_{k+l} \Theta : w}\ (\text{let})$$

$$\frac{a \mapsto \mathrm{fold}\, v \in \Gamma}{\Gamma : \mathrm{unfold}\, a \Downarrow_0 \Gamma : v}\ (\text{unfold}) \qquad \frac{\Gamma : e[v_1/x, v_2/y] \Downarrow_k \Delta : w}{\Gamma : \mathrm{let}(x, y) = (v_1, v_2)\, \mathrm{in}\, e \Downarrow_k \Delta : w}\ (\text{split})$$

$$\frac{\Gamma : e[v/x] \Downarrow_k \Delta : w}{\Gamma : (\lambda x.e)v \Downarrow_{k+1} \Delta : w}\ (\text{app}) \qquad \frac{F(x) = e \in \mathcal{F} \qquad \Gamma : e[v/x] \Downarrow_k \Delta : w}{\Gamma : F\, v \Downarrow_{k+1} \Delta : w}\ (\text{call})$$

In rules that allocate values into the heap, we require that the allocated pointer a is fresh. To simplify matters, it is convenient to assume that this choice is deterministic. When we compare two evaluations of the same expression, we may assume that they allocate their heap values into the same memory cells. In addition, we assume that the chosen pointer is globally fresh. For example, when we have two evaluations of expressions e_1 and e_2 from the same heap, we may combine them into an evaluation of $\mathrm{let}\, x = e_1\, \mathrm{in}\, e_2$ by assuming that they never choose the same pointer for allocation. It is possible (but laborious) to lift both assumptions through the explicit maintenance of substitutions of pointers.

This operational semantics closely approximates the cost of executing a functional program in practice. Like Danielsson [4], we only count the steps that correspond to lambda application and function calls. It can be shown that, for a fixed expression, call-by-value semantics can be simulated by a random access machine using only constant overhead [3].

2.2 Thunks

To model thunks, we extend our syntax and semantics. We add two new heap values: lazy thunks and memoised thunks. A lazy thunk holds a value and is annotated by the top-level function F that should be applied to update the thunk. A memoised thunk holds the value that was computed by the update. We can force a thunk to obtain its memoised value, updating it if necessary. In practice, the function F is not stored in the thunk itself, but inferred from its type during the force operation [22].

$$\begin{array}{rcll} hv & ::= \ldots \mid & \mathrm{memo}\, v & \text{(memoised value)} \\ & \mid & \mathrm{lazy}_F\, v & \text{(lazy computation)} \end{array} \qquad \begin{array}{rcll} e & ::= \ldots \mid & \mathrm{force}\, v & \text{(force thunk)} \end{array}$$

To extend our semantics with thunks, we use a variant of Launchbury's natural semantics [17], which makes thunks first-order [22]. We allocate a thunk in a fresh memory cell using the (lazy) and (memo) rules. If a thunk is memoised, we retrieve its value using the (recall) rule at no cost. If a thunk is lazy, we have to force it using the (force) rule, where we remove it from the heap, run the computation and then store the memoised value back into the heap. Forcing itself is free, but involves a function call, which costs one step.

$$\frac{F \in \mathcal{F} \qquad a \notin \mathrm{dom}(\Gamma)}{\Gamma : \mathrm{lazy}_F\ v \Downarrow_0 (\Gamma, a \mapsto \mathrm{lazy}_F\ v) : a}\ (\mathrm{lazy})$$

$$\frac{\Gamma : F\ v \Downarrow_k \Delta : w}{(\Gamma, a \mapsto \mathrm{lazy}_F\ v) : \mathrm{force}\ a \Downarrow_k (\Delta, a \mapsto \mathrm{memo}\ w) : w}\ (\mathrm{force})$$

$$\frac{a \notin \mathrm{dom}(\Gamma)}{\Gamma : \mathrm{memo}\ v \Downarrow_0 (\Gamma, a \mapsto \mathrm{memo}\ v) : a}\ (\mathrm{memo}) \qquad \frac{a \mapsto \mathrm{memo}\ v \in \Gamma}{\Gamma : \mathrm{force}\ a \Downarrow_0 \Gamma : v}\ (\mathrm{recall})$$

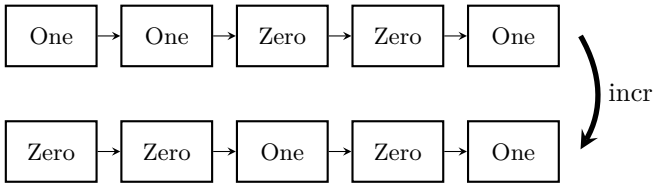
▶ **Example 1.** Because the (force) rule changes thunks from lazy computations to memoised values, repeatedly forcing a thunk is cheap. For example, we can combine several rules to derive:

$$\frac{\Gamma : F\ v \Downarrow_k \Delta : w \qquad a \notin \mathrm{dom}(\Gamma)}{\Gamma : \mathrm{let}\ x = \mathrm{lazy}_F\ v\ \mathrm{in}\ \mathrm{let}\ y = \mathrm{force}\ x\ \mathrm{in}\ \mathrm{force}\ x \Downarrow_k (\Delta, a \mapsto \mathrm{memo}\ w) : w}\ (\ldots)$$

Despite calling force x twice, the total cost is only k steps instead of $2k$ steps.

3 Persistent Amortised Analysis

To illustrate the traditional method of amortised analysis and why it fails in a persistent setting, consider a binary counter implemented as a (little-endian) list of bits. The increment operation traverses the list until it finds a zero bit and flips all bits that it encounters along the way.



A counter representing the number n has at most $\log n$ bits and incrementing the counter takes $O(\log n)$ time in the worst case. However, most increments will be far quicker. In fact, incrementing a binary counter from zero to n takes only $O(n)$ time in total, when the worst-case bound would suggest that it takes $O(n \log n)$ time. This motivates the banker's method of amortised analysis [32], which shows that the *amortised time* per increment is only $O(1)$.

3.1 Banker's Method

The banker's method makes it possible to amortise expensive operations against cheap ones. Cheap operations may spend additional time to save *credits* on the heap. Credits may be spent during an expensive operation to reduce its cost. We model credits by a natural number n that is attached to a heap allocation:

$$\Gamma ::= \emptyset \mid \Gamma, a \mapsto_n hv$$

We add rules for saving credits on these cells and spending them later:

$$\frac{}{\Gamma, a \mapsto_n hv : \mathrm{save}\, m\, a \Downarrow_m \Gamma, a \mapsto_{n+m} hv : a}\ (\text{save})$$

$$\frac{\Gamma, a \mapsto_n hv : e \Downarrow_k \Delta : w \qquad n, m \geq 0}{\Gamma, a \mapsto_{n+m} hv : \mathrm{spend}\, m\, \mathrm{from}\, a\, \mathrm{on}\, e \Downarrow_{\max(k-m,0)} \Delta : w}\ (\text{spend})$$

In the (save) rule, we take m steps to add m credits to a heap allocation. Conversely, in the (spend) rule, we may spend m credits from the heap to reduce the number of steps we have to take. We do not allow negative credits: if we try to spend more credits than we have, this rule does not apply and evaluation is stuck.

3.2 Soundness

To show that the (save) and (spend) rules are correct, we need to compare them to the rules defined earlier. The big-step rules defined in Section 2 give the worst-case number of steps that are needed to evaluate an expression under a given heap. We will call it the real semantics and denote it by $\Downarrow^{\mathrm{R}}_k$. The banker's semantics extends the real semantics with the (save) and (spend) rules, and we denote it by $\Downarrow^{\mathrm{B}}_k$.

We show that the (save) and (spend) rules are correct by comparing the banker's semantics to the real semantics. But this is tricky to do directly, since they use different syntaxes for heaps. We relate the heaps through an erasure function $\ulcorner\cdot\urcorner : \mathrm{Heap}^{\mathrm{B}} \to \mathrm{Heap}^{\mathrm{R}}$ that embeds the banker's heaps with credits into the real heaps without credits.

▶ **Definition 2** (Cost Model). *A cost model* $(\Downarrow, \Phi, \ulcorner\cdot\urcorner)$ *consists of:*

- *a class of heaps* Heap *which may be embedded into real heaps by* $\ulcorner\cdot\urcorner : \mathrm{Heap} \to \mathrm{Heap}^{\mathrm{R}}$
- *a big-step operational semantics* $\Downarrow$: $(\mathrm{Heap} \times \mathrm{Expr}) \rightharpoonup \mathbb{N} \times \mathrm{Heap} \times \mathrm{Value}$
- *a potential function* $\Phi : \mathrm{Heap} \to \mathbb{N}$.

We call a cost model *sound* if it yields the same result as the real semantics and allows us to prove an upper bound on the number of steps taken by the real semantics.

▶ **Definition 3** (Soundness [4]). *A cost model* $(\Downarrow, \Phi, \ulcorner\cdot\urcorner)$ *is sound if for all* $\Gamma : e \Downarrow_n \Delta : v$*:*

- $\ulcorner\Gamma\urcorner : e \Downarrow^{\mathrm{R}}_k \ulcorner\Delta\urcorner : v$
- $k + \Phi(\Delta) - \Phi(\Gamma) \leq n$.

▶ **Corollary 4.** *The real cost model* $(\Downarrow^{\mathrm{R}}, \Phi^{\mathrm{R}}, \ulcorner\cdot\urcorner^{\mathrm{R}})$ *is sound for* $\Phi^{\mathrm{R}}(\Gamma) = 0$ *and* $\ulcorner\Gamma\urcorner^{\mathrm{R}} = \Gamma$.

The banker's method yields a sound cost model. The potential function Φ^{B} counts the total number of credits in the heap and the $\ulcorner\cdot\urcorner^{\mathrm{B}}$ function removes the credit annotations:

$$\Phi^{\mathrm{B}}(\emptyset) = 0 \qquad \Phi^{\mathrm{B}}(\Gamma, a \mapsto_n hv) = \Phi^{\mathrm{B}}(\Gamma) + n \qquad \ulcorner\emptyset\urcorner^{\mathrm{B}} = \emptyset \qquad \ulcorner\Gamma, a \mapsto_n hv\urcorner^{\mathrm{B}} = \ulcorner\Gamma\urcorner^{\mathrm{B}}, a \mapsto hv$$

Additionally, we need to interpret the "save $m\, a$" and "spend m from a on e" expressions in the real semantics. We treat them as no-ops, where "save $m\, a$" returns a at no cost and "spend m from a on e" evaluates e without adjusting its cost.

▶ **Lemma 5** ([32]). *The banker's cost model* $(\Downarrow^{\mathrm{B}}, \Phi^{\mathrm{B}}, \ulcorner\cdot\urcorner^{\mathrm{B}})$ *is sound.*

Proof. By induction on the derivation of $\Gamma : e \Downarrow^{\mathrm{B}}_n \Delta : v$. The (save) and (spend) rules do not change the final result. (save) adds m credits to the heap but also costs m steps. (spend) reduces the number of steps by up to m but also removes m credits from the heap. ◀

3.3 Amortised Analysis of Binary Counters

To analyse the binary counter using the banker's method, we define the heaps of binary counters inductively. The $\mathrm{Counter}(\Gamma : v)$ predicate defines a counter as a list of zeros and ones, where we store a credit for each one-bit.

$$\texttt{End} = \mathrm{inl}() \quad \texttt{Cons}(n, a) = \mathrm{inr}(n, a) \quad \texttt{Zero} = \mathrm{inl}() \quad \texttt{One} = \mathrm{inr}()$$

$$\mathrm{Counter}(a \mapsto_0 \mathrm{fold}\,\texttt{End} : a)$$

$$\mathrm{Counter}(\Gamma : a) \implies \mathrm{Counter}(\Gamma, b \mapsto_0 \mathrm{fold}\,\texttt{Cons}(\texttt{Zero}, a) : b) \qquad (b \notin \mathrm{dom}(\Gamma))$$

$$\mathrm{Counter}(\Gamma : a) \implies \mathrm{Counter}(\Gamma, b \mapsto_1 \mathrm{fold}\,\texttt{Cons}(\texttt{One}, a) : b) \qquad (b \notin \mathrm{dom}(\Gamma))$$

We define the increment operation as follows. We must save a credit for each one-bit we create, but may spend a credit for each one-bit that we flip to a zero-bit.

$\mathrm{incr}(c) = \mathrm{case}(\mathrm{unfold}\, c)\{$
  $\texttt{End} \mapsto \mathrm{save}\, 1\, (\mathrm{fold}\,\texttt{Cons}(\texttt{One}, c))$
  $\texttt{Cons}(n, a) \mapsto \mathrm{case}\, n\{$
    $\texttt{Zero} \mapsto \mathrm{save}\, 1\, (\mathrm{fold}\,\texttt{Cons}(\texttt{One}, a))$
    $\texttt{One} \mapsto \mathrm{spend}\, 1\, \mathrm{from}\, c\, \mathrm{on}\, \mathrm{fold}\,\texttt{Cons}(\texttt{Zero}, \mathrm{incr}(a))\}\}$

▶ **Lemma 6.** *If* $\mathrm{Counter}(\Gamma : c)$*, then* $\Gamma : \mathrm{incr}(c) \Downarrow_2^{\mathrm{B}} \Delta : c'$ *with* $\mathrm{Counter}(\Delta : c')$.

Proof. By induction on the Counter predicate. In each case, the function call takes one step.

- **(End)**: From $\mathrm{Counter}(\Gamma : c)$, we obtain $\mathrm{Counter}(\Gamma, c' \mapsto_1 \mathrm{fold}\,\texttt{Cons}(\texttt{One}, c) : c')$. We save one credit on the new one-bit, for a total cost of 2.
- **(Zero)**: From $\mathrm{Counter}(\Gamma, c \mapsto_0 \mathrm{fold}\,\texttt{Cons}(\texttt{Zero}, a) : c)$, we deduce $\mathrm{Counter}(\Gamma : a)$ and obtain $\mathrm{Counter}(\Gamma, c' \mapsto_1 \mathrm{fold}\,\texttt{Cons}(\texttt{One}, a) : c')$. We save one credit on the new one-bit, for a total cost of 2.
- **(One)**: From $\mathrm{Counter}(\Gamma, c \mapsto_1 \mathrm{fold}\,\texttt{Cons}(\texttt{One}, a) : c)$, we deduce $\mathrm{Counter}(\Gamma : a)$. Thus, by the induction hypothesis, $\Gamma : \mathrm{incr}(a) \Downarrow_2^{\mathrm{B}} \Delta : c'$ with $\mathrm{Counter}(\Delta : c')$. We obtain $\mathrm{Counter}(\Delta, c'' \mapsto_0 \mathrm{fold}\,\texttt{Cons}(\texttt{Zero}, c') : c'')$. This yields a total cost of 3, but we can reduce it to 2 by spending the credit from the one-bit. ◀

3.4 Persistent Usage

This proof is standard for imperative languages, where each increment operation modifies the counter in place. However, this proof does not apply if the counter is used persistently. In functional languages it is possible to increment the same version of the counter multiple times:

```
val ones = One(One(One(End))) // saves three credits
val c1 = incr(ones)           // spends three credits and saves one
val c2 = incr(ones)           // spends three credits and saves one
```

In the program above, we create a counter with three one-bits, which gives us three credits. Then we perform two increments on *the same counter*. Each increment flips three one-bits to zero-bits and creates another one-bit. In total, the program spends six credits but only saves five credits. Clearly this is wrong: we cannot spend more credits than we have.

The problem is that Lemma 6 does not apply to the second increment. Given $\mathrm{Counter}(\Gamma : \mathrm{ones})$, we can perform the first increment, yielding $\mathrm{Counter}(\Delta : c_1)$. But there is no guarantee that $\mathrm{Counter}(\Delta : \mathrm{ones})$ holds, which prevents us from applying the lemma again.

In fact, increments do not take $O(1)$ amortised time in a persistent setting:

▶ **Proposition 7.** *Starting from an empty counter, n persistent increments may take $\Omega(n \log n)$ time.*

Proof. Use up to $n/2$ increments to obtain a binary counter corresponding to the number $2^{\lfloor \log(n/2) \rfloor} - 1$. Then perform $n/2$ increments on this counter, persistently. Each increment takes $\Omega(\log n)$ time, since it has to flip all $\lfloor \log(n/2) \rfloor$ bits again. ◀

This is not just of purely theoretical concern: many classic data structures do not enjoy good amortised time bounds in a persistent setting. For example, Binomial Heaps [33] extend binary counters to a priority queue by associating each one-bit with a tree. However, their amortised bound does not hold in a persistent setting [25].

3.5 Persistence

To describe when an analysis holds in a persistent setting, we need to formalise how the heap may change during evaluation. Intuitively, an operational semantics is persistent if the heap only gets "better" during evaluation. First, we describe how the heap changes during evaluation using a preorder $\sqsubseteq$ on heaps, which we call the *accessibility* relation [1].

▶ **Definition 8** (Accessibility). *An accessible cost model is a cost model $(\Downarrow, \Phi, \ulcorner \cdot \urcorner)$ equipped with a preorder $\sqsubseteq$ on heaps such that $\Gamma \sqsubseteq \Delta$ iff $\Gamma : e \Downarrow_n \Delta : v$ for some e, n, v.*

Note that $\sqsubseteq$ is fully determined by $\Downarrow$. However, it is convenient to have a succinct definition of $\sqsubseteq$ that hides the concrete expression e. For this reason, we will define the $\sqsubseteq$ relation explicitly using inference rules, and prove that our explicit rules yield accessible cost models. For the real cost model, the heap only changes by adding new allocations and forcing thunks:

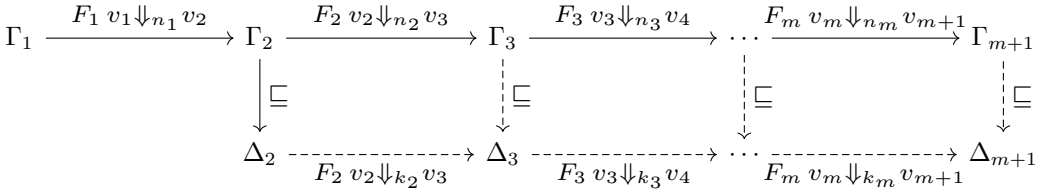
$$\frac{}{\Gamma \sqsubseteq \Gamma}\ [\text{refl}] \qquad \frac{\Gamma \sqsubseteq \Delta \qquad \Delta \sqsubseteq \Theta}{\Gamma \sqsubseteq \Theta}\ [\text{trans}] \qquad \frac{a \notin \operatorname{dom}(\Gamma)}{\Gamma \sqsubseteq \Gamma, a \mapsto hv}\ [\text{alloc}]$$

$$\frac{\Gamma \sqsubseteq \Delta \qquad \Gamma : F\, v \Downarrow_k \Delta : w}{(\Gamma, a \mapsto \operatorname{lazy}_F v) \sqsubseteq (\Delta, a \mapsto \operatorname{memo} w)}\ [\text{force}]$$

Let us say that an analysis describes a sequence of operations $F_1, F_2, \ldots, F_m$, and we ensure that each intermediate state $\Gamma_i : v_i$ satisfies some invariant. This is enough to show soundness in a sequential setting. In a persistent setting, an operation may change the intermediate state $\Gamma_i : v_i$ to $\Delta_i : v_i$ for $\Gamma_i \sqsubseteq \Delta_i$. Can we still apply the operation F_i to $\Delta_i : v_i$?

▶ **Definition 9** (Persistence). *An accessible cost model $((\Downarrow, \Phi, \ulcorner \cdot \urcorner), \sqsubseteq)$ is persistent if for all $\Gamma : e \Downarrow_n \Gamma' : v$ and $\Gamma \sqsubseteq \Delta$, there exist Δ', k such that $\Delta : e \Downarrow_k \Delta' : v$, $k \leq n$ and $\Gamma' \sqsubseteq \Delta'$. It is uniformly persistent if $k = n$.*

If the cost model is persistent, we can apply F_i to $\Delta_i : v_i$. It is guaranteed that this yields the same result v_{i+1}, does not take more steps, and that the new state $\Delta_{i+1} : v_{i+1}$ is still accessible from $\Gamma_{i+1} : v_{i+1}$. This allows us to inductively shift all subsequent operations to the extended heap:



In a persistent cost model, we can thus reason about persistent usage using sequential reasoning. We do not have to consider how the data structure may change in between operations, since our reasoning can be applied to any accessible state automatically. In particular, our reasoning does not have to establish monotonicity explicitly, since the evaluation itself is guaranteed to be monotonic.

▶ **Lemma 10** ([26]). *The real cost model* $((\Downarrow^{\mathrm{R}}, \Phi^{\mathrm{R}}, {}^{\ulcorner}\cdot{}^{\urcorner\mathrm{R}}), \sqsubseteq^{\mathrm{R}})$ *is persistent.*

Proof. We prove this using induction. The interesting case is if the evaluation forces a thunk, which is also forced in the larger heap. Then the evaluation succeeds using the (recall) rule, since thunk evaluation is deterministic. The full proof can be found in the appendix.

$$\begin{array}{ccc} \Gamma & \xrightarrow{\text{(force)}} & \Gamma' \\ \big\downarrow{\scriptstyle [\text{force}]} & & \vdots{\scriptstyle [\text{refl}]} \\ \Delta & \dashrightarrow{\scriptstyle \text{(recall)}} & \Delta' \end{array}$$

◀

▶ Remark 11. To show that the real cost model is persistent, we need to assume without loss of generality that we never allocate two values under the same name. For example, we can allocate $\mathrm{fold}\,(\mathrm{inl}\,v)$ at a if the heap is $\emptyset$, but this fails if the heap is $\{a \mapsto \mathrm{fold}\,(\mathrm{inr}\,w)\}$, even though $\emptyset \sqsubseteq \{a \mapsto \mathrm{fold}\,(\mathrm{inl}\,v)\}$. As such, the real cost model is only persistent if name clashes do not occur.

We could fix this issue in a more formal way by maintaining an explicit substitution in the definition of persistence. A cost model would be persistent if for all $\Gamma : e \Downarrow_n \Gamma' : v$ and $\Gamma \sqsubseteq \Delta$, there exist Δ', k, w and a substitution of pointers σ such that $\Delta : e \Downarrow_k \Delta' : w$, $k \leq n$, $\sigma(v) = w$, $\sigma(\Gamma) = \Gamma$ and $\sigma(\Gamma') \sqsubseteq \Delta'$.

3.6 The Banker's Method, Revisited

The banker's method is not persistent. To see why, consider how the heaps may change during evaluation. Because we can both save and spend credits, our accessibility relation has to account for arbitrary changes in the number of credits:

$$\frac{0 \leq n \leq m}{\Gamma, a \mapsto_n hv \sqsubseteq \Gamma, a \mapsto_m hv}\ [\text{save}] \qquad \frac{n \geq m \geq 0}{\Gamma, a \mapsto_n hv \sqsubseteq \Gamma, a \mapsto_m hv}\ [\text{spend}]$$

▶ **Lemma 12.** *The banker's cost model* $((\Downarrow^{\mathrm{B}}, \Phi^{\mathrm{B}}, {}^{\ulcorner}\cdot{}^{\urcorner\mathrm{B}}), \sqsubseteq^{\mathrm{B}})$ *is accessible.*

▶ **Proposition 13** ([25]). *The banker's cost model* $((\Downarrow^{\mathrm{B}}, \Phi^{\mathrm{B}}, {}^{\ulcorner}\cdot{}^{\urcorner\mathrm{B}}), \sqsubseteq^{\mathrm{B}})$ *is not persistent.*

Proof. Define $\Gamma = \{a \mapsto_5 \mathrm{fold}\,v\}$, $\Gamma' = \{a \mapsto_0 \mathrm{fold}\,v\}$ and $e = \mathrm{spend}\,5\,\mathrm{from}\,a\,\mathrm{on}\,()$. Then $\Gamma : e \Downarrow_0^{\mathrm{B}} \Gamma' : ()$. Define $\Delta = \{a \mapsto_0 \mathrm{fold}\,v\}$. Then $\Gamma \sqsubseteq \Delta$ by the [spend] rule. But $\Delta : e$ is stuck, since the (spend) rule only applies if there are enough credits.

$$\begin{array}{ccc} \Gamma & \xrightarrow{\text{(spend)}} & \Gamma' \\ \big\downarrow{\scriptstyle [\text{spend}]} & & \\ \Delta & & \end{array}$$

◀

This proposition shows the essential problem with the banker's method in a persistent setting. In order to perform amortised analysis of a data structure, we need to establish an invariant on the number of credits it contains. But in a persistent setting, an operation on a different version of the data structure may spend the credits and invalidate these invariants.

4 Persistent Amortised Analysis with Credits

The traditional banker's method does not work in a persistent setting, since it allows credits to be saved and spent arbitrarily. Okasaki [25] proposes to fix this issue by augmenting data structures with thunks. We discuss his reasoning style in the next section. Beforehand, we want to discuss a simpler reasoning style that is inspired by Danielsson's variant of debit passing style [4]. In this style, credits are saved only on lazy thunks, and may only be spent on the update when the thunk is forced [26]. In this work, we omit all references to debits and consequently call it credit passing style.

Why does this approach allow for persistent amortised analysis? One way to understand this is in terms of *linearity*: the restriction that credits may not be duplicated. The traditional analysis fails because it duplicates credits when spending them on an operation, and maintains them on the previous version of the data structure at the same time. In contrast, lazy thunks keep their content linear: even if there are several references to the thunk, it is only forced once and the credits on it are only spent once. If credits are only saved on lazy thunks, they stay linear, even if the thunks themselves are used non-linearly.

4.1 Credit Passing Style

While the banker's method allows credit annotations on all heap values, credit passing style only allows them on lazy thunks in the heap:

$$\Gamma ::= \emptyset \mid \Gamma, a \mapsto hv \mid \Gamma, a \mapsto_n \mathrm{lazy}_F\ v$$

The (memo) and (recall) rules remain unchanged, but we need to modify the rules for lazy thunks. When we force a lazy thunk, we now take the credits stored on the thunk to pay for the computation. If the thunk does not have enough credits, the evaluation can get stuck. To ensure that enough credits are available, we add a (save) rule, which saves credits on a lazy thunk; credits saved on a memoised thunk are wasted.

$$\frac{F \in \mathcal{F} \qquad a \notin \mathrm{dom}(\Gamma)}{\Gamma : \mathrm{lazy}_F\ v \Downarrow_0 (\Gamma, a \mapsto_0 \mathrm{lazy}_F\ v) : a}\ (\text{lazy}) \qquad \frac{a \mapsto \mathrm{memo}\ v \in \Gamma}{\Gamma : \mathrm{save}\, m\, a \Downarrow_m \Gamma : a}\ (\text{waste})$$

$$\frac{}{(\Gamma, a \mapsto_n \mathrm{lazy}_F\ v) : \mathrm{save}\, m\, a \Downarrow_m (\Gamma, a \mapsto_{n+m} \mathrm{lazy}_F\ v) : a}\ (\text{save})$$

$$\frac{\Gamma : F\, v \Downarrow_k \Delta : w \qquad k \leq n}{(\Gamma, a \mapsto_n \mathrm{lazy}_F\ v) : \mathrm{force}\ a \Downarrow_0 (\Delta, a \mapsto \mathrm{memo}\ w) : w}\ (\text{force})$$

We call this the credit passing semantics and write $\Downarrow_k^{\mathrm{C}}$ to refer to it. The potential function Φ^{C} counts the total number of credits in the heap and the $\ulcorner \cdot \urcorner^{\mathrm{C}}$ function removes the credit annotations:

$$\Phi^{\mathrm{C}}(\emptyset) = 0 \qquad \Phi^{\mathrm{C}}(\Gamma, a \mapsto_n \mathrm{lazy}_F\ v) = \Phi^{\mathrm{C}}(\Gamma) + n \qquad \Phi^{\mathrm{C}}(\Gamma, a \mapsto hv) = \Phi^{\mathrm{C}}(\Gamma)$$

$$\ulcorner \emptyset \urcorner^{\mathrm{C}} = \emptyset \qquad \ulcorner \Gamma, a \mapsto_n \mathrm{lazy}_F\ v \urcorner^{\mathrm{C}} = \ulcorner \Gamma \urcorner^{\mathrm{C}}, a \mapsto \mathrm{lazy}_F\ v \qquad \ulcorner \Gamma, a \mapsto hv \urcorner^{\mathrm{C}} = \ulcorner \Gamma \urcorner^{\mathrm{C}}, a \mapsto hv$$

▶ **Lemma 14.** *The credit passing cost model* $(\Downarrow^{\mathrm{C}}, \Phi^{\mathrm{C}}, \ulcorner\cdot\urcorner^{\mathrm{C}})$ *is sound.*

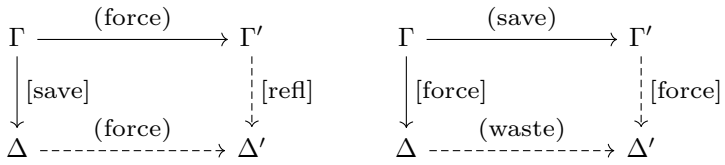
Let us consider how the heaps evolve in the credit passing semantics. Our accessibility relation $\sqsubseteq^{\mathrm{C}}$ extends $\sqsubseteq^{\mathrm{R}}$ by the rules:

$$\frac{hv = \mathrm{lazy}_F\ v \qquad n \leq m}{(\Gamma, a \mapsto_n hv) \sqsubseteq (\Gamma, a \mapsto_m hv)}\ [\mathrm{save}] \qquad \frac{\Gamma \sqsubseteq \Delta \qquad \Gamma : F\,v \Downarrow_k \Delta : w \qquad k \leq n}{(\Gamma, a \mapsto_n \mathrm{lazy}_F\ v) \sqsubseteq (\Delta, a \mapsto \mathrm{memo}\ w)}\ [\mathrm{force}]$$

Unlike in the traditional banker's method, the number of credits on a *lazy* thunk may only increase in this semantics. It can decrease when the thunk is forced, but at that point the thunk is replaced by a memoised value, which costs nothing to force. This ensures that the heaps can only become "better" over time:

▶ **Lemma 15.** *The credit passing cost model* $((\Downarrow^{\mathrm{C}}, \Phi^{\mathrm{C}}, \ulcorner\cdot\urcorner^{\mathrm{C}}), \sqsubseteq^{\mathrm{C}})$ *is uniformly persistent.*

Proof. If we force a thunk on which more credits have been saved, those credits are discarded. If we save credits on a thunk that has been forced, we waste these credits.



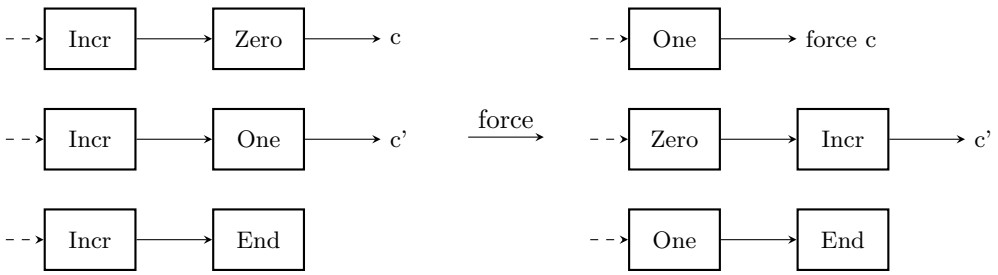
◀

Note that credits cannot be moved between thunks. If we could do so, we would no longer have a lower bound on the number of credits stored on any particular thunk, which would break the persistence property. This is a major difference to the traditional banker's method, which allows credits to be moved arbitrarily.

In the sequential setting, the banker's method is equivalent to the physicist's method, where the latter can be seen as a special case of the former by moving all credits to the root. In contrast, these methods are not equivalent in the persistent setting. While Okasaki also presents a physicist's method, it has the draconian restriction that any thunk can only be forced if *all* thunks may be forced [25, page 69]. This restriction enforces that only *excess* credits are moved, as it establishes the execution cost as a lower bound on the number of credits stored on each thunk. However, it is very rare for a data structure to contain enough credits to force all thunks, which makes Okasaki's physicist's method much less useful.

4.2 Persistent Binary Counter

To illustrate the credit passing technique, we add thunks to the binary counter. We consider thunks as internal nodes that correspond to a delayed increment operation. When we force a thunk, we perform one step of the increment operation:



Again, we define the heaps of binary counters inductively. This time, each zero-bit is followed by a lazy increment thunk. We assign one credit to increment thunks, but not to other heap allocations.

$$\begin{aligned}
&\text{Counter}(a \mapsto \text{fold}\,\texttt{End} : a) \\
\text{Counter}(\Gamma : a) &\implies \text{Counter}(\Gamma, b \mapsto \text{fold}\,\texttt{Cons}(\texttt{One}, a) : b) && (b \notin \text{dom}(\Gamma)) \\
\text{Counter}(\Gamma : a) &\implies \text{Counter}(\Gamma, c \mapsto \text{fold}\,\texttt{Cons}(\texttt{Zero}, b), b \mapsto_1 \text{lazy}_{\texttt{Incr}}\ a : c) && (b, c \notin \text{dom}(\Gamma))
\end{aligned}$$

We change the increment operation as follows:

$$\begin{aligned}
&\text{incr}(c) = \text{force}(\text{save}\,2\,(\text{lazy}_{\texttt{Incr}}\ c)) \\
&\texttt{Incr}(c) = \text{case}(\text{unfold}\,c)\{ \\
&\quad \texttt{End} \mapsto \text{fold}\,\texttt{Cons}(\texttt{One}, c) \\
&\quad \texttt{Cons}(n, a) \mapsto \text{case}\,n\{ \\
&\qquad \texttt{Zero} \mapsto \text{fold}\,\texttt{Cons}(\texttt{One}, \text{force}(\text{save}\,1\,a)) \\
&\qquad \texttt{One} \mapsto \text{fold}\,\texttt{Cons}(\texttt{Zero}, \text{save}\,1\,(\text{lazy}_{\texttt{Incr}}\ a))\}\}
\end{aligned}$$

To increment a counter c, we just add an $\texttt{Incr}(c)$ thunk to the front, save two credits on it and force it. To show that this works, we have to prove that the thunk can be forced in two steps and the resulting counter is still well-formed.

▶ **Lemma 16.** *If* $\text{Counter}(\Gamma : c)$*, then* $\Gamma : \mathit{Incr}(c) \Downarrow_k^{\text{C}} \Delta : c'$ *with* $\text{Counter}(\Delta : c')$ *and* $k \leq 2$*.*

Proof. By induction on the Counter predicate. In each case, the function call takes one step.

- **(End)**: From $\text{Counter}(\Gamma : c)$, we obtain $\text{Counter}(\Gamma, c' \mapsto \text{fold}\,\texttt{Cons}(\texttt{One}, c) : c')$, for a total cost of 1.
- **(Zero)**: From $\text{Counter}(\Gamma, c \mapsto \text{fold}\,\texttt{Cons}(\texttt{Zero}, a), a \mapsto_1 \text{lazy}_{\texttt{Incr}}\ b : c)$, we deduce $\text{Counter}(\Gamma : b)$. We save a credit on a for a total of two credits. This allows us to force the delayed `Incr(b)` computation, which yields $\text{Counter}(\Delta : b')$ by the induction hypothesis. We obtain $\text{Counter}(\Delta, c' \mapsto \text{fold}\,\texttt{Cons}(\texttt{One}, b') : c')$. Because we save one credit, the total cost is 2.
- **(One)**: From $\text{Counter}(\Gamma, c \mapsto \text{fold}\,\texttt{Cons}(\texttt{One}, a) : c)$, we deduce $\text{Counter}(\Gamma : a)$. We allocate a new thunk and save a credit on it, thus obtaining $\text{Counter}(\Gamma, c' \mapsto \text{fold}\,\texttt{Cons}(\texttt{Zero}, b), b \mapsto_1 \text{lazy}_{\texttt{Incr}}\ a : c')$. Because we save one credit, the total cost is 2. ◀

In the persistent binary counter, every lazy thunk contains an `Incr` operation with a certain number of credits on it. In more advanced data structures, there are usually thunks for different operations with different numbers of credits on them. For example, a persistent double-ended queue would contain thunks for delayed "cons", "tail", "snoc", and "init" operations. This can be supported in our framework by annotating thunks $\text{lazy}_{F_i}\ v$ with different operations $F_1, \ldots, F_n$. Of course, if different operations F_i have different costs, then the reasoning has to accommodate that – this is a frequent source of complexity in Okasaki's book.

4.3 Credit Inheritance

Lorenzen [20] proposes credit inheritance to analyse Okasaki's Banker's Queue. Credit passing style cannot be used to analyse that data structure, as it wastes credits that are saved on a memoised thunk. To solve this issue, credit inheritance allows each thunk to designate

another thunk as its *heir*. If a thunk has an excess of credits when forced, it may pass excess credits on to its heir instead of wasting them. Similarly, once a thunk is memoised, all credits saved on it are passed on to the heir as well.

We extend the syntax of heaps to add an heir h on memoised thunks:

$$\Gamma ::= \emptyset \mid \Gamma, a \mapsto hv \mid \Gamma, a \mapsto_n \text{lazy}_F\ v \mid \Gamma, a \mapsto_h \text{memo}\ v$$

Similarly, we extend all existing rules to propagate the heir. Rules that do not have the judgement as a premise (like (value) or (unfold)) use the empty heir $\emptyset$. The other rules propagate the heir from their premise. Since the (let) rule has two premises, we add a side-condition that the premises do not choose different heirs:

$$\frac{\Gamma : e_1 \Downarrow_k^{h_1} \Delta : v \qquad \Delta : e_2[v/x] \Downarrow_l^{h_2} \Theta : w \qquad |h_1 \cup h_2| \leq 1}{\Gamma : \text{let}\, x = e_1 \,\text{in}\, e_2 \Downarrow_{k+l}^{h_1 \cup h_2} \Theta : w}\ (\text{let})$$

The key new rule is the (pass) rule, which allows us to designate another thunk as the heir h and pass on credits from our computation to the heir. We record the heir on the reduction relation $\Downarrow_k^{\{h\}}$ of the current computation. This rule does not specify how many credits m are passed on to the heir; this will be determined later by the (force) rule.

$$\frac{\Gamma : \text{save}\, m\, h \Downarrow_m^{\emptyset} \Delta : v}{\Gamma : \text{pass}\, h \Downarrow_m^{\{h\}} \Delta : v}\ (\text{pass})$$

The (force) rule installs the heir annotation on the memoised thunk and removes it from the judgement. This time we ask that the computation in the thunk spends *all* the credits available on the thunk. This ensures that the number m chosen in (pass) is as large as possible and no excess credits are wasted.

$$\frac{\Gamma : F\, v \Downarrow_k^{\{h\}} \Delta : w \qquad k = n}{(\Gamma, a \mapsto_n \text{lazy}_F\ v) : \text{force}\ a \Downarrow_0^{\emptyset} (\Delta, a \mapsto_h \text{memo}\ w) : w}\ (\text{force})$$

The (pass) rule allows us to choose the number of credits m to pass on seemingly without constraint. However, the requirement that the (force) rule spends all credits implies that m has to be the difference between the number of credits and the cost of the computation. It is unfortunate that the rules enforce this invariant only indirectly, but this seems unavoidable in a big-step semantics. In a small-step semantics, the (pass) rule would have direct access to the number of credits that are left on the thunk and could choose m deterministically. This is done in the testing framework of Lorenzen [20].

Finally, the heir annotation allows us to provide the (inherit) rule. Unlike the (waste) rule, which discards credits placed on memoised thunks, the (inherit) rule allows us to retain these credits by passing them on to the heir.

$$\frac{\Gamma : \text{save}\, m\, h \Downarrow_m^{\emptyset} \Delta : v}{(\Gamma, a \mapsto_h \text{memo}\ w) : \text{save}\, m\, a \Downarrow_m^{\emptyset} (\Delta, a \mapsto_h \text{memo}\ w) : a}\ (\text{inherit})$$

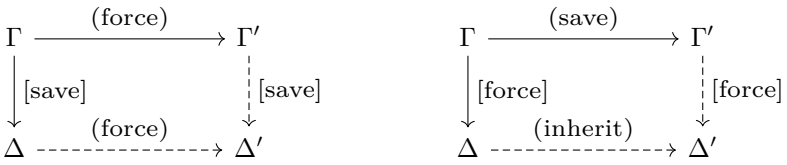
We call this the credit-inheritance semantics and write $\Downarrow_k^{\text{CI}}$ for $\Downarrow_k^{\emptyset}$. The potential function $\Phi^{\text{CI}} = \Phi^{\text{C}}$ counts the total number of credits in the heap and the $\ulcorner \cdot \urcorner^{\text{CI}}$ function also removes the heir annotations ($\ulcorner \Gamma, a \mapsto_h \text{memo}\ v \urcorner^{\text{CI}} = \ulcorner \Gamma \urcorner^{\text{CI}}, a \mapsto \text{memo}\ v$). In the real semantics, we interpret "pass h" as a no-op that returns h without changing the heap.

▶ **Lemma 17.** *The credit-inheritance cost model* $(\Downarrow^{\text{CI}}, \Phi^{\text{CI}}, \ulcorner \cdot \urcorner^{\text{CI}})$ *is sound.*

The $\sqsubseteq^{\text{CI}}$ relation differs from $\sqsubseteq^{\text{C}}$ only in the new heir annotation in the [force] rule.

▶ **Lemma 18.** *The credit-inheritance cost model* $((\Downarrow^{\mathrm{CI}}, \Phi^{\mathrm{CI}}, \ulcorner\cdot\urcorner^{\neg\mathrm{CI}}), \sqsubseteq^{\mathrm{CI}})$ *is uniformly persistent.*

Proof. If we force a thunk on which more credits have been saved, those extra credits will be passed on to the heir instead of being discarded. If we save credits on a thunk that has already been forced, those credits will be inherited by the heir instead of being wasted.



◀

Credit inheritance is loosely inspired by Okasaki's debit inheritance [25]. However, it usually leads to a different style of reasoning. For example, Lorenzen [20, Section 6.6] uses it to analyse the Banker's Queue. In that data structure, the "append" operation creates a long list of thunks, each of which needs a credit to run. This is an ideal setting for credit inheritance, since each thunk can name its successor in the list as its heir. This makes it possible to store credits on the head of the list, from which the (inherit) rule propagates them to the first lazy thunk. To reason about the queue, we thus do not need to know at which index of the list the first lazy thunk can be found; as long as the amount of credits saved on the head equals the length of the list, the computation succeeds.

Credit inheritance also shows the power of using credits rather than debits. We would not be able to perform the same analysis if we simply replaced credits by debits. Debits require us to know the cost of updating a thunk when creating it. In contrast, credit inheritance allows us to let the cost of updating a thunk grow dynamically to match the number of credits we have: all excess credits are simply passed on to the heir. It is possible to emulate this in a debit-based analysis, but we would have to wait until all debits are paid off, and then change the thunk retroactively to include further save-statements that increase its debits [28].

5 Persistent Amortised Analysis with Debits

Okasaki does not reason in terms of credits, but uses *debits*: a number that describes how many credits we have to save on a thunk before we can force it. We can view a debit as just a negative credit: if the evaluation of a thunk takes k steps and the thunk has n credits, we may equivalently say that the thunk has $k - n$ debits. But as we have seen in the last section, one can perform persistent amortised analysis purely in terms of credits. If debits are not necessary, then why does Okasaki's work use debits at all?

We believe that Okasaki's use of debits should be seen as part of his evaluation model of thunks. In his analysis, Okasaki acts as if thunks were evaluated upon creation. But he does not pay for the evaluation and instead places the cost as a debit on the result of the thunk. Debits thus act as a barrier that prevents access to the result until the evaluation has been paid for. Okasaki motivates debits by analogy to a "layaway plan", where you select what to buy and the store holds it for you until you have fully paid for it [25, page 60].

Debits are a natural choice when we assume that thunks are evaluated upon creation. In that setting, we know the cost of updating the thunk. We could still use credits and count up until we have enough credits to cover the cost, but it is much more elegant to use debits and count down until we reach zero remaining debits instead.

5.1 The Persistent Banker's Method

Okasaki defines the amortised cost of an operation informally as the *unshared* cost of actually performing work plus the number of debits discharged [25, page 60]. To formalise this, we track two numbers $\Downarrow_{k,k'}$, where k is the total number of time steps taken by the evaluation and k' is the number of debits discharged during the evaluation. All previous rules can be extended to this setting by either propagating k' from their single premise to their conclusions, returning $k' = 0$ if no premise exists, or, in the (let) rule, adding the k' numbers of the two premises.

In our heaps, we now allow debits to be placed on memoised thunks. Unlike credits, debits may be negative. We write memo w for a memoised thunk that has been accessed by the program and thus holds no debits. For a memoised thunk that has not yet been accessed, we write $\text{memo}^{\Delta}_{F\,v}\ w$ to record the computation $F\,v$ that computes w, as well as the new allocations Δ created by the computation. These additional records are not necessary for reasoning, but enable us to connect this semantics to the real semantics later. Our syntax for heaps is now:

$$\Gamma ::= \emptyset \mid \Gamma, a \mapsto hv \mid \Gamma, a \mapsto_n \text{memo}^{\Delta}_{F\,v}\ w$$

In the (lazy) rule, we now run the computation of the thunk immediately, but do not pay for it. Instead, we place the unshared cost as a debit on the resulting thunk. However, if the thunk itself pays off debits, this cost is propagated to the outside. The (force) rule allows us to access the value of a memoised thunk once the debt has been paid off. Once we access it, we remove the annotations from the thunk. Finally, the (save) rule allows us to pay off debits on a memoised thunk and if the thunk has no debits left, saved credits end up being wasted.

$$\frac{\Gamma : F\,v \Downarrow_{k,k'} \Delta : w \qquad a \notin \text{dom}(\Delta)}{\Gamma : \text{lazy}_F\ v \Downarrow_{0,k'} (\Delta, a \mapsto_k \text{memo}^{\Delta \setminus \Gamma}_{F\,v}\ w) : a}\ (\text{lazy})$$

$$\frac{n \leq 0}{\Gamma, a \mapsto_n \text{memo}^{\Delta}_{F\,v}\ w : \text{force}\ a \Downarrow_{0,0} \Gamma, a \mapsto \text{memo}\ w : w}\ (\text{force})$$

$$\frac{hv = \text{memo}^{\Delta}_{F\,v}\ w}{(\Gamma, a \mapsto_n hv) : \text{save}\,m\,a \Downarrow_{0,m} (\Gamma, a \mapsto_{n-m} hv) : a}\ (\text{save})$$

$$\frac{a \mapsto \text{memo}\ w \in \Gamma}{\Gamma : \text{save}\,m\,a \Downarrow_{0,m} \Gamma : a}\ (\text{waste})$$

We call this the debit semantics and write $\Downarrow^{\text{D}}_{k+k'}$ for $\Downarrow_{k,k'}$. To show that this semantics is sound, we need to undo the evaluation performed in the (lazy) rule. If a thunk still has an annotation, we restore it to the lazy state and remove all new allocations that were added during the computation. In the potential function, we count the number of debits that were paid off, where k^{R}_{Fv} is the real cost of evaluating the thunk and n is its number of debits:

$$\Phi^{\text{D}}(\emptyset) = 0 \qquad \Phi^{\text{D}}(\Gamma, a \mapsto_n \text{memo}^{\Delta}_{F\,v}\ w) = \Phi^{\text{D}}(\Gamma) + k^{\text{R}}_{Fv} - n \qquad \Phi^{\text{D}}(\Gamma, a \mapsto hv) = \Phi^{\text{D}}(\Gamma)$$
$$\ulcorner\emptyset\urcorner^{\text{D}} = \emptyset \quad \ulcorner\Gamma, a \mapsto_n \text{memo}^{\Delta}_{F\,v}\ w\urcorner^{\text{D}} = \ulcorner\Gamma \setminus \Delta\urcorner^{\text{D}}, a \mapsto \text{lazy}_F\ v \quad \ulcorner\Gamma, a \mapsto hv\urcorner^{\text{D}} = \ulcorner\Gamma\urcorner^{\text{D}}, a \mapsto hv$$

▶ **Lemma 19.** *The debit cost model* $(\Downarrow^{\text{D}}, \Phi^{\text{D}}, \ulcorner\cdot\urcorner^{\text{D}})$ *is sound.*

Let us consider how the heaps evolve in the debit semantics. Our accessibility relation $\sqsubseteq^{\text{D}}$ on heaps is defined as follows:

$$\frac{\Gamma \sqsubseteq \Delta \qquad \Gamma : F\,v \Downarrow_{k,k'} \Delta : w \qquad a \notin \mathrm{dom}(\Delta)}{\Gamma \sqsubseteq (\Delta, a \mapsto_k \mathrm{memo}_{F\,v}^{\Delta \setminus \Gamma}\ w)}\ [\text{lazy}]$$

$$\frac{hv = \mathrm{memo}_{F\,v}^{\Delta}\ w \qquad n \geq m}{(\Gamma, a \mapsto_n hv) \sqsubseteq (\Gamma, a \mapsto_m hv)}\ [\text{save}] \qquad \frac{n \leq 0}{(\Gamma, a \mapsto_n \mathrm{memo}_{F\,v}^{\Delta}\ w) \sqsubseteq (\Gamma, a \mapsto \mathrm{memo}\ w)}\ [\text{force}]$$

Again, the number of debits on memoised thunks may only decrease in this semantics. This ensures that the heaps can only become "better" over time:

▶ **Lemma 20.** *The debit cost model* $((\Downarrow^{\mathrm{D}}, \Phi^{\mathrm{D}}, {}^{\ulcorner}\cdot^{\urcorner_{\mathrm{D}}}), \sqsubseteq^{\mathrm{D}})$ *is uniformly persistent.*

Perhaps the most surprising aspect of the debit semantics is that it has to track both k and k', the unshared cost of evaluation and the number of debits discharged. This is necessary to split these costs in the (lazy) rule, where the cost k' may not be placed as a debit on the resulting thunk, but has to be propagated to the outside instead. Indeed, for a top-level computation that lives outside of any thunks, this distinction does not matter and we can specify the total cost as $\Downarrow^{\mathrm{D}}_{k+k'}$ for $\Downarrow_{k,k'}$.

▶ **Example 21.** The debit semantics would be unsound if the (lazy) rule placed k' as a debit on the resulting thunk. Assume we are given a heap $\Gamma = \{a \mapsto_n \mathrm{memo}_{F_1\ v}\ w\}$ and the expression $e = \mathrm{lazy}_{F_2}()$ for $F_2(()) = \mathrm{save}\, n\, a$. Our correct (lazy) rule allows us to derive:

$$\Gamma : e \Downarrow_{0,n} \Delta : b \text{ with } \Delta = \{a \mapsto_0 \mathrm{memo}_{F_1\ v}\ w, b \mapsto_1 \mathrm{memo}^{\emptyset}_{F_2\ ()}\ a\}$$

That is, the $\mathrm{save}\, n\, a$ operation is performed, which removes the debits from a, and we propagate the cost n to the outside. Instead, the unsound (lazy) rule places the discharged debits on the resulting thunk and allows us to derive:

$$\Gamma : e \Downarrow_{0,0} \Delta : b \text{ with } \Delta = \{a \mapsto_0 \mathrm{memo}_{F_1\ v}\ w, b \mapsto_{n+1} \mathrm{memo}^{\emptyset}_{F_2\ ()}\ a\}$$

This derivation does not propagate the cost n to the outside, but simply moves n debits from a to b. Since a now has no remaining debits, we may force it, without forcing b at all. This allows us to access w at no cost and thus breaks soundness.

5.2 Persistent Binary Counter

To illustrate Okasaki's reasoning, we analyse the binary counter using debits. Again, we define the heaps of counters inductively. We assume that all thunks are memoised, but place one debit on memoised thunks. For simplicity, we omit the annotations $F\,v, \Delta$ on unaccessed memoised thunks.

$$\begin{aligned}
&\mathrm{Counter}(a \mapsto \mathrm{fold}\,\texttt{End} : a) \\
\mathrm{Counter}(\Gamma : a) &\implies \mathrm{Counter}(\Gamma, b \mapsto \mathrm{fold}\,\texttt{Cons}(\texttt{One}, a) : b) && (b \notin \mathrm{dom}(\Gamma)) \\
\mathrm{Counter}(\Gamma : a) &\implies \mathrm{Counter}(\Gamma, c \mapsto \mathrm{fold}\,\texttt{Cons}(\texttt{Zero}, b), b \mapsto_1 \mathrm{memo}\ a : c) && (b, c \notin \mathrm{dom}(\Gamma))
\end{aligned}$$

The implementation of the persistent binary counter stays largely the same as in Section 4. The only difference is that we need to save one less credit on the increment thunks. That is, we modify the incr operation by replacing $\mathrm{save}\,2\,(\mathrm{lazy}_{\texttt{Incr}}\ c)$ with $\mathrm{save}\,1\,(\mathrm{lazy}_{\texttt{Incr}}\ c)$ and the lazy `Incr` function by replacing $\mathrm{save}\,1\,(\mathrm{lazy}_{\texttt{Incr}}\ a)$ with $\mathrm{lazy}_{\texttt{Incr}}\ a$. In our proof, we now assume that all thunks are evaluated upon creation and place the cost of that evaluation as a debit on the resulting thunk. As in the debit semantics, we account separately for the cost of paying off debits.

▶ **Lemma 22.** *If* $\mathrm{Counter}(\Gamma : c)$*, then* $\Gamma : \mathrm{Incr}(c) \Downarrow^{\mathrm{D}}_{1,k'} \Delta : c'$ *with* $\mathrm{Counter}(\Delta : c')$ *and* $k' \leq 1$*.*

Proof. By induction on the Counter predicate. In each case, the function call takes one step.

- **(End)**: From $\mathrm{Counter}(\Gamma : c)$, we obtain $\mathrm{Counter}(\Gamma, c' \mapsto \mathrm{fold}\,\mathtt{Cons}(\mathtt{One}, c) : c')$.
- **(Zero)**: From $\mathrm{Counter}(\Gamma, c \mapsto \mathrm{fold}\,\mathtt{Cons}(\mathtt{Zero}, b), b \mapsto_1 \mathrm{memo}\ a : c)$, we deduce $\mathrm{Counter}(\Gamma : a)$. We pay off the debit on b and obtain $\mathrm{Counter}(\Gamma, c' \mapsto \mathrm{fold}\,\mathtt{Cons}(\mathtt{One}, a) : c')$.
- **(One)**: From $\mathrm{Counter}(\Gamma, c \mapsto \mathrm{fold}\,\mathtt{Cons}(\mathtt{One}, a) : c)$, we deduce $\mathrm{Counter}(\Gamma : a)$. By the induction hypothesis, $\Gamma : \mathrm{Incr}(a) \Downarrow^{\mathrm{D}}_{1,1} \Delta : c'$ with $\mathrm{Counter}(\Delta : c')$. We obtain $\mathrm{Counter}(\Delta, c'' \mapsto \mathrm{fold}\,\mathtt{Cons}(\mathtt{Zero}, b), b \mapsto_1 \mathrm{memo}\ c' : c'')$, where we place the cost of evaluation as a debit on b and propagate the cost of paying off the debit. ◀

This proof is quite similar to the sequential one (Lemma 6). By pretending that `Incr` thunks are evaluated immediately, we can reason about them just as we reasoned about the `incr` function. The main difference is that this proof places debits on zero-bits, while the sequential proof places credits on one-bits. In the recursive case, where one-bits are turned into zero-bits, the sequential program consumes credits while the persistent program produces debits.

This proof is typical for Okasaki's debit method. Perhaps surprisingly, this proof reasons about thunks that have not yet been created. On a counter consisting only of one-bits, the program creates an increment thunk and forces it to flip the first one-bit to a zero-bit. However, in the proof, we act as if we continued evaluating the thunk and flipped all one-bits to zero-bits immediately… even though the program has not yet created the thunks for those flips. This is possible because thunk evaluation is deterministic: One can argue that "in the future, thunk a will create a new thunk b" and then reason about b before it physically exists [25, page 67].

5.3 Debit Inheritance

In Example 21, we saw that it is generally unsound to move debits between thunks. This is sound, however, if we can guarantee an evaluation order between the thunks. For example, we might know that b will always be forced before a. Okasaki then allows b to *inherit* the debit from a [25, page 67].

An important special case where it is guaranteed that b will always be forced before a is if a is created during the evaluation of b. Even though the (lazy) rule evaluates b immediately, the result of that computation (and thus a) only becomes accessible once b has been forced. This ensures that a is only accessible from outside the thunk once b's debit has been paid off.

To encode this in our semantics, we modify the (lazy) rule. For a freshly allocated thunk, we allow propagating some of the unshared work to the outside.

$$\frac{\Gamma : F\,v \Downarrow_{(k_1+k_2),k'} \Delta : w \qquad a \notin \mathrm{dom}(\Gamma)}{\Gamma : \mathrm{lazy}_F\ v \Downarrow_{k_1,k'} (\Delta, a \mapsto_{k_2} \mathrm{memo}^{\Delta\setminus\Gamma}_{F\,v}\ w) : a}\ (\mathrm{lazy})$$

Our modification may seem counter-intuitive: rather than changing how thunks discharge debits, we change how freshly allocated thunks distribute their unshared cost. However, if b is a thunk that creates a during its evaluation, then this (lazy) rule allows us to place the unshared cost of a on b. Note that this cannot be emulated by the (*save*) rule, since a's debits become part of the unshared cost k of b, instead of the discharge cost k' of b.

This (lazy) rule captures Okasaki's reasoning style for *monolithic* computations, which create new thunks that have to be forced in the computation itself. When reasoning about them, "all debits are usually assigned to the root" [25, page 61].

We call this the debit-inheritance semantics and write $\Gamma : e \Downarrow^{\mathrm{DI}}_{k+k'} \Delta : w$ for $\Gamma : e \Downarrow_{k,k'} \Delta : w$. We reuse the Φ^{D} and $\ulcorner \cdot \urcorner^{\mathrm{D}}$ defined for the debit semantics.

▶ **Lemma 23.** *The debit-inheritance cost model* $(\Downarrow^{\mathrm{DI}}, \Phi^{\mathrm{D}}, \ulcorner \cdot \urcorner^{\mathrm{D}})$ *is sound.*

The $\sqsubseteq^{\mathrm{DI}}$ relation differs from $\sqsubseteq^{\mathrm{D}}$ only in that the new debit is k_2 instead of k in the [lazy] rule.

▶ **Lemma 24.** *The debit-inheritance cost model* $((\Downarrow^{\mathrm{DI}}, \Phi^{\mathrm{D}}, \ulcorner \cdot \urcorner^{\mathrm{D}}), \sqsubseteq^{\mathrm{DI}})$ *is uniformly persistent.*

Debit inheritance is in some sense dual to credit inheritance. While credit inheritance passes on excess credits from a thunk to one of its children, debit inheritance passes on debits from a child to its parent. Aside from this similarity, the two techniques are quite different though. For example, debit inheritance allows a parent to inherit the debits from multiple children, while credit inheritance only allows passing on credits to a single child.

6 Related Work

Credits and Debits

Okasaki [25] claims that debits are necessary for reasoning about persistent usage, and that using credits is unsound. When a data structure with credits is used persistently, credits could be spent more than once, which yields wrong time bounds. But debits do not suffer from this issue: "although savings can only be spent once, it does no harm to pay off debt more than once" [25, page 59]. However, debits work only because they are associated with thunks. Thunks guarantee that the resources they hold are used linearly, even if there are many references to the thunk itself. Pilkiewicz and Pottier [26] and Mével et al. [24] exploit this property to implement a model of thunks that holds credits in the lazy state. They show that this model does not duplicate credits and is monotonic, but continue to use debits in their interface of thunks.

Reasoning using credits is especially useful when the cost of evaluating a thunk may change over time. If thunks become cheaper to evaluate, a credit-based interface allows forcing the thunk as soon as the stored credits exceed the present cost, while a debit-based interface only allows forcing once the debits assigned initially have been paid off.

Conversely, we might want to make thunks more expensive over time. In Okasaki's Banker's Queue, the topmost thunk of the stream should pass all its credits to the next thunk. Pottier et al. [28] model this using their Thunk-Consequence rule, which allows them to add "save" calls to an existing thunk. As this increases the execution cost of the thunk, they need to increase the number of debits retroactively. In our work, the same can be achieved more directly by using credit inheritance, which dynamically passes on all remaining credits after covering the execution cost. This design is only possible because we track the execution cost and the amount of credits separately.

Reasoning About the Future

In the absence of side-effects, it does not matter when a thunk is evaluated (shown formally by Hackett and Hutton [10]). As such, Okasaki's reasoning style may assume that thunks are evaluated immediately upon creation. However, to obtain the right time complexity of the program, we may not charge it the cost of evaluating the thunk upon creation. Instead, Okasaki uses debits as a "layaway plan", where the program may access the result of a thunk only once all debits on the thunk have been paid off.

Perhaps less appreciated is that this restriction applies not just to the result of the thunk, but also to all side-effects that are performed during the evaluation of the thunk. In particular, if the thunk performs a side-effect, the program may not register that the side-effect occurred until all debits have been paid off.

Crucially, saving credits on another thunk is a side-effect. Thus, Okasaki cannot generally create debits for paying off thunks, or he would run into the soundness issue described in Example 21. While he does create debits for paying off thunks in his debit passing style, he restricts it to those thunks that are guaranteed to be forced after the enclosing thunk [25, page 174]. As he notes, this restriction is similar to that of the debit-inheritance semantics. The restriction is an important difference from Danielsson's version of debit passing style [4], which allows saving credits on any thunk. Danielsson's version is sound because the side-effect is only executed when the thunk is forced; in contrast, Okasaki's version executes the side-effect immediately and thus has to be more restrictive.

In Okasaki's Banker's Queue, it is necessary to pay off a thunk that will be created when evaluating another thunk. Okasaki's method makes it possible to save credits on the inner thunk directly. In contrast, Danielsson [4] assumes that thunks are evaluated on forcing and cannot do so. Instead, he proposes deep payment as an alternative. Deep payment injects a "save" operation into the computation of the enclosing thunk to pay off the debits of the inner thunk once the enclosing thunk is forced. Pottier et al. [28] generalise this principle to their "Thunk-Consequence" rule, which allows arbitrary implications to be injected.

Mutable References in Thunks

The language we consider in this paper is purely functional and does not include mutable references. Unfortunately, almost all the results in this paper break if mutable references are included. For example, persistence requires that the evaluation of a thunk is deterministic, so that we do not have to revert to the lazy state when returning to a previous version. But thunk evaluation is not guaranteed to be deterministic if the thunks can read from mutable references. Similarly, the cost of evaluating a thunk (and thus its number of debits) may depend on the state of mutable references. If we determine the cost of forcing a thunk given the state of the heap at the time of its creation, then this cost may change if mutable references are written to. Thus it would be unsound to assign a fixed number of debits to a thunk at creation time and force it once enough debits have been assigned; after all, the cost of forcing the thunk may have gone up. This restriction does not impact our ability to model Okasaki's data structures, which can all be implemented without mutable references.

Pottier et al. [28] avoid the requirement for deterministic execution by specifying the result of a thunk only up to an invariant. This makes it possible to run a non-deterministic computation in a thunk, as long as the result satisfies the invariant. As their work is embedded in Iris, it can interface with code that includes references and even concurrency.

Persistence and Monotonicity

Our characterisation of persistence is an instance of heap monotonicity [7]. It is inspired by the work of Pilkiewicz and Pottier [26], who show that thunks are monotonic. In Danielsson's work [4], the persistence of thunks is not explicitly stated, but follows from his type preservation result. Type preservation is explicitly related to heap monotonicity in proofs that use logical relations [1]. Mével et al. [24] and Pottier et al. [28] show persistence using the persistent modality of Iris.

There are several variants of persistence. Confluent persistence [6] allows changes to different versions of a data structure at the same time. For example, this makes it possible to concatenate a confluently persistent queue to a previous version of itself. Semi-persistence allows changes to previous versions only if the current version is abandoned. This is particularly useful for backtracking algorithms; see Allain et al. [2] for an overview. We discuss these variants in the full version of this paper.

Linearity

There is a long history of using type systems for amortised analysis [13, 11, 31, 9]; see Hoffmann and Jost [12] for an overview. Common to these approaches is the use of a linear type system to control the usage of resources. In particular, amortised analysis is only correct if credits are not duplicated, which is guaranteed by linearity. This makes these approaches unsuitable for amortised analysis of persistent usage. However, they can still be used for amortised analysis of sequential usage and worst-case analysis of persistent usage.

Jost et al. [14] extend amortised resource analysis to thunks, following Danielsson's approach [4]. Madhavan et al. [23] design a tool for estimating the cost of functional programs with thunks. They apply their tool to Okasaki's real-time queue and deque, but do not propose a general reasoning principle.

The credit-based approach to persistent amortised analysis is connected to linearity by storing credits on thunks. Thunks are only evaluated once, even if they are shared, which guarantees that their content in the lazy state is used linearly [26, 24, 22].

7 Conclusion

We have presented an operational account of amortised analysis in a persistent setting. Our characterisation of sound and persistent cost models allows us to precisely distinguish between approaches that are persistent and those that are not. We show that amortised analysis can be performed using credits alone, without the need for debits. We also provide an operational semantics for Okasaki's use of debits and show that it is sound and persistent.

Which Reasoning Principle is Best?

While our work describes how lazy data structures can be analysed, it leaves open how they should be analysed. The credit-based analysis is quite close to the actual implementation on a machine and allows us to provide an invariant that describes the state of the data structure at any point in time. However, the credit-based analysis is somewhat more involved than the debit-based analysis, which can side-step many operational details to arrive at a shorter invariant. We find reasoning using credits more intuitive, but we have come to appreciate the brevity of debit-based reasoning as well. Furthermore, while our work concerns a strict language where thunks are only used sparsely, a lazy language where thunks are ubiquitous may instead benefit from a reasoning style based on demand [10, 8, 19, 34].

Beyond Thunks

There could be a more general interface for persistent mutation in functional programming languages. As we have seen, thunks are sound because updates do not change their content semantically, and thunks are persistent because updates can only reduce the costs of future operations. The first property is shared by quotient types, where an update that exchanges equal representatives of the quotient does not break referential transparency [30]. In fact,

thunks can be seen as an instance of this principle [22]. However, we additionally need to ensure monotonicity of updates. For example, it would not be persistent to revert a thunk back to its lazy state after it has been memoised. We are interested in exploring a more general interface for monotonic updates, like LVars [16] or stable references [27, 15].

References

1 Amal Jamil Ahmed. *Semantics of types for mutable state*. Princeton University, 2004.

2 Clément Allain, Basile Clément, Alexandre Moine, and Gabriel Scherer. Snapshottable stores. *Proceedings of the ACM on Programming Languages*, 8(ICFP):338–369, 2024. doi:10.1145/3674637.

3 Guy Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 226–237, 1995. doi:10.1145/224164.224210.

4 Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, POPL '08, pages 133–144, New York, NY, USA, 2008. ACM. doi:10.1145/1328438.1328457.

5 James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989. doi:10.1016/0022-0000(89)90034-2.

6 James R Driscoll, Daniel DK Sleator, and Robert E Tarjan. Fully persistent lists with catenation. *Journal of the ACM (JACM)*, 41(5):943–959, 1994. doi:10.1145/185675.185791.

7 Manuel Fähndrich and K Rustan M Leino. Heap monotonic typestates. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2003.

8 Kenneth Foner, Hengchu Zhang, and Leonidas Lampropoulos. Keep your laziness in check. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–30, 2018. doi:10.1145/3236797.

9 Harrison Grodin and Robert Harper. Amortized analysis via coalgebra. *Electronic Notes in Theoretical Informatics and Computer Science*, 4, 2024. doi:10.46298/entics.14797.

10 Jennifer Hackett and Graham Hutton. Call-by-need is clairvoyant call-by-value. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–23, 2019. doi:10.1145/3341718.

11 Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(3):1–62, 2012. doi:10.1145/2362389.2362393.

12 Jan Hoffmann and Steffen Jost. Two decades of automatic amortized resource analysis. *Mathematical Structures in Computer Science*, 32(6):729–759, 2022. doi:10.1017/S0960129521000487.

13 Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *European Symposium on Programming*, pages 22–37. Springer, 2006. doi:10.1007/11693024_3.

14 Steffen Jost, Pedro Vasconcelos, Mário Florido, and Kevin Hammond. Type-based cost analysis for lazy functional languages. *Journal of Automated Reasoning*, 59(1):87–120, 2017. doi:10.1007/s10817-016-9398-9.

15 Haim Kaplan, Chris Okasaki, and Robert Endre Tarjan. Simple confluently persistent catenable lists. *SIAM J. Comput.*, 30(3):965–977, 2000. doi:10.1137/S0097539798339430.

16 Lindsey Kuper and Ryan R Newton. Lvars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 71–84, 2013. doi:10.1145/2502323.2502326.

17 John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 144–154, 1993. doi:10.1145/158511.158618.

18 Paul-Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and computation*, 185(2):182–210, 2003. doi:10.1016/S0890-5401(03)00088-9.

19 Yao Li, Li-yao Xia, and Stephanie Weirich. Reasoning about the garden of forking paths. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–28, 2021. doi:10.1145/3473585.

20 Anton Lorenzen. Lightweight testing of persistent amortized time complexity in the credit monad. In *Proceedings of the 18th ACM SIGPLAN International Haskell Symposium*, 2025. doi:10.1145/3759164.3759351.

21 Anton Lorenzen. Persistent amortised analysis, operationally, 2026. arXiv:2605.09411.

22 Anton Lorenzen, Daan Leijen, Wouter Swierstra, and Sam Lindley. First-order laziness. *Proceedings of the ACM on Programming Languages*, 9(ICFP), 2025. doi:10.1145/3747530.

23 Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. Contract-based resource verification for higher-order functions with memoization. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 330–343. ACM, 2017. doi:10.1145/3009837.3009874.

24 Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time credits and time receipts in iris. In *Programming Languages and Systems: 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings 28*, pages 3–29. Springer, 2019. doi:10.1007/978-3-030-17184-1_1.

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

26 Alexandre Pilkiewicz and François Pottier. The essence of monotonic state. In *Proceedings of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 73–86, 2011. doi:10.1145/1929553.1929565.

27 Juliette Ponsonnet and François Pottier. Verified Persistent Catenable Deques. In *JFLA 2026 – 37es Journées Francophones des Langages Applicatifs*, volume JFLA 2026 – 37es Journées Francophones des Langages Applicatifs, Oberbronn, France, January 2026. Marie Kerjean and Yannick Zakowski. URL: https://hal.science/hal-05427954.

28 François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. Thunks and debits in separation logic with time credits. *Proceedings of the ACM on Programming Languages*, 8(POPL):1482–1508, 2024. doi:10.1145/3632892.

29 Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, July 1986. doi:10.1145/6138.6151.

30 Daniel Selsam, Simon Hudon, and Leonardo de Moura. Sealing pointer-based optimizations behind pure functions. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–20, 2020. doi:10.1145/3408997.

31 Hugo Simoes, Pedro Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 165–176, 2012. doi:10.1145/2364527.2364575.

32 Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985. doi:10.1137/0606031.

33 Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978. doi:10.1145/359460.359478.

34 Li-yao Xia, Laura Israel, Maite Kramarz, Nicholas Coltharp, Koen Claessen, Stephanie Weirich, and Yao Li. Story of your lazy function's life: A bidirectional demand semantics for mechanized cost analysis of lazy programs. *Proc. ACM Program. Lang.*, 8(ICFP), August 2024. doi:10.1145/3674626.