

Persistent Amortized Analysis, Operationally

Anton Lorenzen  

University of Edinburgh, United Kingdom

Abstract

Amortized 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 amortized 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 amortizing expensive restructuring work. In his seminal book, Chris Okasaki showed how to extend amortized analysis to persistent usage. His method works by storing debits on thunks. However, Okasaki describes his approach only informally, which makes it hard to understand precisely when and why it works.

In this work, we provide an account of persistent amortized analysis in terms of operational semantics for a call-by-value lambda calculus with thunks. We give a new definition of persistence for these semantics and show formally that traditional amortized analysis is not persistent. Then we show that it is possible to perform persistent amortized analysis by storing credits on thunks. This refutes the folklore belief that debits are necessary for persistent amortized analysis. Finally, we provide a semantics for Okasaki's debit-based approach. Our work clarifies the formal foundation of Okasaki's work and makes it accessible to a wider audience.

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, Amortized Analysis

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Acknowledgements I thank Francois Pottier, Wouter Swierstra, Kim Worrall and Kengo Hirata for their helpful feedback on an earlier version of this paper. I thank the participants of the AUTOSARD 2025 workshop for lively discussions.

1 Introduction

Amortized 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 amortized 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 [4, 22, 5] 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 amortized analysis. Consider a version of a persistent data structure that is highly unbalanced. In a traditional amortized analysis, this version is assigned *credits*, which may be spent by a future operation to perform expensive restructuring work to rebalance the data structure [24]. The credits may be spent on this work if the old version is no longer used. But if the old version is still accessible, each operation on the old version will perform the same expensive restructuring work again. Since the credits can only pay for one of these operations, this makes the amortized analysis invalid.



© Anton Lorenzen;
licensed under Creative Commons License CC-BY 4.0
42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:31



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Okasaki [18] proposed that *thunks* can be used to achieve amortized 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 *memoized* state. Its value in the memoized 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 memoized value, but it will not learn whether the update was performed to obtain it. This makes it possible to share restructuring work between different versions of a data structure. If an operation memoizes a thunk, it will be memoized 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.

Thunks allow Okasaki to reason about persistent data structures as if they were used sequentially. Pilkiewicz and Pottier [19] and Mével et al. [17] formalize this intuition by proving that thunks are *monotonic*: When reasoning about a persistent data structure, we may assume that a thunk is lazy and that we have to pay for updating it, but if the thunk was already memoized, our reasoning will still be valid and simply over-approximate the cost.

In his seminal book, Okasaki [18] adds thunks to many well-known data structures and shows that his variations enjoy good amortized time bounds, even when used persistently. He justifies his analysis using several informal reasoning principles. Yet, even experts find it hard to nail down precisely how and when Okasaki’s reasoning principles work. This has created interest in formalizing Okasaki’s reasoning principles. Danielsson [3] 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. [21] formalize debit-passing-style in Iris and also provide a formal foundation for reasoning about the future evaluation of thunks.

But the role of debits in Okasaki’s work has remained underexplored. Okasaki asserts that persistent amortized analysis requires *debits* and that using *credits* is unsound. This has become a folklore belief that lead debits to be the foundation of subsequent work [3, 21, 9, 17, 19]. Even though several of these works already used credits under the hood, their reasoning principles are still based on debits. Lorenzen [15] argues that reasoning purely in terms of credits is sound as well, but does not prove this.

The paper is structured as follows:

In Section 3, we give a new characterization of persistence as the heap monotonicity of an operational semantics. 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 amortized analysis is not persistent.

In Section 4, we give operational semantics for credit-passing-style. Credit-passing-style is a variant of Okasaki’s and Danielsson’s debit-passing-style, that places *credits on lazy thunks*. We show that it is sound and persistent in our operational model. This is the first formal proof that persistent amortized analysis can be performed purely in terms of credits and refutes the folklore belief that only debits enable persistent amortized analysis.

In Section 5, we extend our operational model to Okasaki’s persistent Banker’s method. Rather than store credits on lazy thunks, this method stores *debits on memoized thunks*. This allows Okasaki to disregard lazy thunks entirely and assume that all thunks are already memoized, even if it is not yet possible to pay for the update. This motivates Okasaki’s use of debits as a barrier or “layaway plan” that prevents us from accessing a value before we have paid for it. We show that this is sound and persistent in our operational model.

Our work clarifies the formal foundation of Okasaki’s work and makes it accessible to a wider audience. Our formal model of persistence also provides a foundation to generalize

thinks to more powerful persistent primitives, such as stable references [20, 10].

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)	
	a, b, c	(pointers)	hv	(heap values)
	$()$	(unit)	$\text{let } x = e \text{ in } e$	(let binding)
	$\text{inl } v \mid \text{inr } v$	(sum)	$\text{case } v \{ \text{inl } x \mapsto e; \text{inr } y \mapsto e \}$	(case split)
	(v, v)	(pair)	$\text{let}(x, y) = v \text{ in } e$	(destruct a pair)
	$\lambda x. e$	(lambda)	$v w$	(lambda application)
$hv ::= \text{fold } v$	(allocate)	$\text{unfold } v$	(dereference)	
$\mathcal{F} ::= \emptyset \mid \mathcal{F}, F(x) = e$		$F v$	(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 [13], 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 variables 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 . This allows us to assume that each operation is performed on values of the correct shape: for example, each case-split is performed on either an `inl v` or an `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.

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

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

$$\begin{array}{c}
 \frac{a \mapsto \text{fold } v \in \Gamma}{\Gamma : \text{unfold } a \Downarrow_0 \Gamma : v} \text{ (unfold)} \quad \frac{\Gamma : e[v_1/x, v_2/y] \Downarrow_k \Delta : w}{\Gamma : \text{let}(x, y) = (v_1, v_2) \text{ in } e \Downarrow_k \Delta : w} \text{ (split)} \\
 \\
 \frac{\Gamma : e[v/x] \Downarrow_k \Delta : w}{\Gamma : (\lambda x. e)v \Downarrow_{k+1} \Delta : w} \text{ (app)} \quad \frac{F(x) = e \in \mathcal{F} \quad \Gamma : e[v/x] \Downarrow_k \Delta : w}{\Gamma : Fv \Downarrow_{k+1} \Delta : w} \text{ (call)}
 \end{array}$$

This semantics closely approximates the cost of executing a functional program in practice. Like Danielsson [3], 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 [2].

2.2 Thunks

To model thunks, we extend our syntax and semantics. We add two new heap values: lazy thunks and memoized 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 memoized thunk holds the value that was computed by the update. We can force a thunk to obtain its memoized 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 [16].

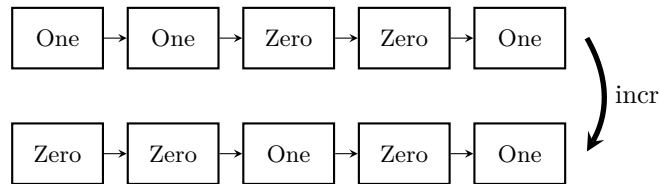
$$\begin{array}{l}
 hv ::= \dots \mid \text{memo } v \quad (\text{memoized value}) \quad e ::= \dots \mid \text{force } v \quad (\text{force thunk}) \\
 \quad \mid \text{lazy}_F v \quad (\text{lazy computation})
 \end{array}$$

To extend our semantics with thunks, we use a variant of Launchbury's natural semantics [12], that makes thunks first-order [16]. We allocate a thunk in a fresh memory cell using the (lazy) and (memo) rules. If a thunk is memoized, we retrieve its value using the (recall) rule at a cost of one step. 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 memoized value back into the heap. This costs one step plus the cost of computing the value.

$$\begin{array}{c}
 \frac{a \notin \text{dom}(\Gamma) \quad F \in \mathcal{F}}{\Gamma : \text{lazy}_F v \Downarrow_0 (\Gamma, a \mapsto \text{lazy}_F v) : a} \text{ (lazy)} \\
 \\
 \frac{\Gamma : Fv \Downarrow_k \Delta : w}{(\Gamma, a \mapsto \text{lazy}_F v) : \text{force } a \Downarrow_k (\Delta, a \mapsto \text{memo } w) : w} \text{ (force)} \\
 \\
 \frac{a \notin \text{dom}(\Gamma)}{\Gamma : \text{memo } v \Downarrow_0 (\Gamma, a \mapsto \text{memo } v) : a} \text{ (memo)} \quad \frac{a \mapsto \text{memo } v \in \Gamma}{\Gamma : \text{force } a \Downarrow_0 \Gamma : v} \text{ (recall)}
 \end{array}$$

3 Persistent Amortized Analysis

To illustrate the traditional method of amortized 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 Bankers method of amortized analysis [24], which shows that the *amortized time* per increment is only $O(1)$.

3.1 Bankers Method

The Bankers method makes it possible to amortize expensive operations against cheap operations. 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:

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

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

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_k^R . The Bankers semantics extends the real semantics with the (save) and (spend) rules, and we denote it by \Downarrow_k^B .

We show that (save) and (spend) rules are correct by comparing the Bankers 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 $\lceil \cdot \rceil : \text{Heap}^B \rightarrow \text{Heap}^R$ that embeds the Bankers heaps with credits into the real heaps without credits.

► **Definition 1 (Cost Model).** A cost model $(\Downarrow, \Phi, \lceil \cdot \rceil)$ consists of:

- a class of heaps Heap which may be embedded into real heaps by $\lceil \cdot \rceil : \text{Heap} \rightarrow \text{Heap}^R$
- a big-step operational semantics $\Downarrow : (\text{Heap} \times \text{Expr}) \rightarrow \mathbb{N} \times \text{Heap} \times \text{Value}$
- a potential function $\Phi : \text{Heap} \rightarrow \mathbb{N}$.

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 2 (Soundness [3]).** A cost model $(\Downarrow, \Phi, \lceil \cdot \rceil)$ is sound if for all $\Gamma : e \Downarrow_n \Delta : v$:

- $\lceil \Gamma \rceil : e \Downarrow_k^R \lceil \Delta \rceil : v$
- $k + \Phi(\Delta) - \Phi(\Gamma) \leq n$.

► **Corollary 3.** *The real cost model $(\Downarrow^R, \Phi^R, \ulcorner \cdot \urcorner^R)$ is sound for $\Phi^R(\Gamma) = 0$ and $\ulcorner \Gamma \urcorner^R = \Gamma$.*

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

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

► **Lemma 4** ([24]). *The Bankers cost model $(\Downarrow^B, \Phi^B, \ulcorner \cdot \urcorner^B)$ is sound.*

Proof. By induction on the derivation of $\Gamma : e \Downarrow_n^B \Delta : v$. 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 Amortized Analysis of Binary Counters

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

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

$$\text{Counter}(a \mapsto_0 \text{fold End} : a)$$

$$\text{Counter}(\Gamma : a) \implies \text{Counter}(\Gamma, b \mapsto_0 \text{fold Cons}(\text{Zero}, a) : b) \quad (b \notin \text{dom}(\Gamma))$$

$$\text{Counter}(\Gamma : a) \implies \text{Counter}(\Gamma, b \mapsto_1 \text{fold Cons}(\text{One}, a) : b) \quad (b \notin \text{dom}(\Gamma))$$

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.

$$\begin{aligned} \text{incr}(c) = & \text{case}(\text{unfold } c) \{ \\ & \text{End} \mapsto \text{save } 1 \text{ (fold Cons}(\text{One}, c)) \\ & \text{Cons}(n, a) \mapsto \text{case } n \{ \\ & \quad \text{Zero} \mapsto \text{save } 1 \text{ (fold Cons}(\text{One}, a)) \\ & \quad \text{One} \mapsto \text{spend } 1 \text{ from } c \text{ on fold Cons}(\text{Zero}, \text{incr}(a)) \} \} \end{aligned}$$

► **Lemma 5.** *If $\text{Counter}(\Gamma : c)$, then $\Gamma : \text{incr}(c) \Downarrow_2^B \Delta : c'$ with $\text{Counter}(\Delta : c')$.*

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

- **(End):** From $\text{Counter}(\Gamma : c)$, we obtain $\text{Counter}(\Gamma, c' \mapsto_1 \text{fold Cons}(\text{One}, c) : c')$. We save one credit on the new one-bit, for a total cost of 2.
- **(Zero):** From $\text{Counter}(\Gamma, c \mapsto_0 \text{fold Cons}(\text{Zero}, a) : c)$, we deduce $\text{Counter}(\Gamma : a)$ and obtain $\text{Counter}(\Gamma, c' \mapsto_1 \text{fold Cons}(\text{One}, a) : c')$. We save one credit on the new one-bit, for a total cost of 2.
- **(One):** From $\text{Counter}(\Gamma, c \mapsto_1 \text{fold Cons}(\text{One}, a) : c)$, we deduce $\text{Counter}(\Gamma : a)$. By the induction hypothesis, thus $\Gamma : \text{incr}(a) \Downarrow_2^B \Delta : c'$ with $\text{Counter}(\Delta : c')$. We obtain $\text{Counter}(\Delta, c'' \mapsto_0 \text{fold Cons}(\text{Zero}, c') : c'')$. 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 like Koka, it is possible to increment the same 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 all three one-bits. In total the program spends six credits but only saves five credits. Clearly this is wrong: we can not spend more credits than we have.

The problem is that Lemma 5 does not apply to the second increment. Given $\text{Counter}(\Gamma : \text{ones})$, we can perform the first increment, yielding $\text{Counter}(\Delta : c_1)$. But there is no guarantee that $\text{Counter}(\Delta : \text{ones})$ holds, which prevents us from applying the lemma again.

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

► **Proposition 6.** *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 amortized time bounds in a persistent setting. For example, Binomial Heaps [25] extend binary counters to a priority queue by associating each one-bit with a tree. However, their amortized bound does not hold in a persistent setting [18].

3.5 Persistence

To describe when an analysis holds in a persistent setting, we need to formalize how the heap may change during evaluation. We use a partial order \sqsubseteq on heaps, which we call the *accessibility* relation [1].

► **Definition 7 (Accessibility).** *An accessible cost model $(\Downarrow, \Phi, \lceil \cdot \rceil)$ equipped with a partial order \sqsubseteq on heaps such that $\Gamma \sqsubseteq \Delta$ iff $\Gamma : e \Downarrow_n \Delta : v$ for some e, n, v .*

Let us say that an analysis describes a sequence of operations F_1, F_2, \dots, 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 8 (Persistence).** *An accessible cost model $((\Downarrow, \Phi, \lceil \cdot \rceil), \sqsubseteq)$ 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:

$$\begin{array}{ccccccc}
 \Gamma_1 : v_1 & \xrightarrow{F_1 v_1 \Downarrow_{n_1}} & \Gamma_2 : v_2 & \xrightarrow{F_2 v_2 \Downarrow_{n_2}} & \Gamma_3 : v_3 & \xrightarrow{F_3 v_3 \Downarrow_{n_3}} & \dots & \xrightarrow{F_m v_m \Downarrow_{n_m}} & \Gamma_{m+1} : v_{m+1} \\
 & & \downarrow \sqsubseteq & & \downarrow \sqsubseteq & & \downarrow \sqsubseteq & & \downarrow \sqsubseteq \\
 & & \Delta_2 : v_2 & \xrightarrow{F_2 v_2 \Downarrow_{k_2}} & \Delta_3 : v_3 & \xrightarrow{F_3 v_3 \Downarrow_{k_3}} & \dots & \xrightarrow{F_m v_m \Downarrow_{k_m}} & \Delta_{m+1} : v_{m+1}
 \end{array}$$

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 consider monotonicity explicitly, since the evaluation itself is guaranteed to be monotonic.

For the real cost model, the heap only changes by adding new allocations and forcing thunks:

$$\begin{array}{c}
 \frac{}{\Gamma \sqsubseteq \Gamma} \text{ [refl]} \qquad \frac{\Gamma \sqsubseteq \Delta \quad \Delta \sqsubseteq \Theta}{\Gamma \sqsubseteq \Theta} \text{ [trans]} \qquad \frac{a \notin \text{dom}(\Gamma)}{\Gamma \sqsubseteq \Gamma, a \mapsto hv} \text{ [alloc]} \\
 \frac{\Gamma \sqsubseteq \Delta \quad \Gamma : Fv \Downarrow_k \Delta : w}{(\Gamma, a \mapsto \text{lazy}_F v) \sqsubseteq (\Delta, a \mapsto \text{memo } w)} \text{ [force]}
 \end{array}$$

► **Lemma 9** ([19]). *The real cost model $((\Downarrow^R, \Phi^R, \ulcorner^R), \sqsubseteq^R)$ 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. ◀

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

► **Remark 10.** To show that the real cost model is persistent, we assume without loss of generality that we never allocate two values under the same name. For example, we can allocate fold v at a if the heap is \emptyset , but this fails if the heap is $\{a \mapsto \text{fold } w\}$, even though $\emptyset \sqsubseteq \{a \mapsto \text{fold } w\}$. But the names in the heap are always chosen freshly and so we may assume that name clashes never occur.

Intuitively, persistence means that $\Gamma \sqsubseteq \Delta$ implies that Δ is “better” for amortization than Γ . Thunks are persistent, because every update leaves them in a better state [19].

3.6 The Bankers Method, Revisited

The Bankers 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 11.** *The Bankers cost model $((\Downarrow^B, \Phi^B, \ulcorner^B), \sqsubseteq^B)$ is accessible.*

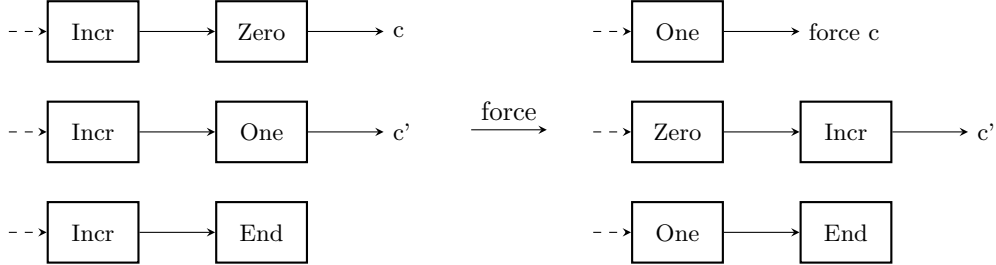
► **Proposition 12** ([18]). *The Bankers cost model $((\Downarrow^B, \Phi^B, \ulcorner^B), \sqsubseteq^B)$ is not persistent.*

Proof. Define $\Gamma = \{a \mapsto_5 \text{fold } v\}$, $\Gamma' = \{a \mapsto_0 \text{fold } v\}$ and $e = \text{spend } 5$ from a on (\cdot) . Then $\Gamma : e \Downarrow_1^B \Gamma' : (\cdot)$. Define $\Delta = \{a \mapsto_0 \text{fold } v\}$. Then $\Gamma \sqsubseteq \Delta$ by the [spend] rule. But $\Delta : e$ is stuck, since the (spend) rule only applies if there are enough credits. ◀

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

4 Persistent Amortized Analysis with Credits

To illustrate the credit-passing technique, which does work in a persistent setting, we add thinks to the binary counter. We use thinks as internal nodes that correspond to a delayed increment operation. When we force a think, 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 think. We assign one credit to increment thinks, but not to other heap allocations.

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

We change the increment operation as follows:

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

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

► **Lemma 13.** *If $\text{Counter}(\Gamma : c)$, then $\Gamma : \text{Incr}(c) \Downarrow_2^C \Delta : c'$ with $\text{Counter}(\Delta : c')$.*

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

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

◀

4.1 Credit Passing Style

The key idea that makes the analysis above work in a persistent setting is that credits are placed on lazy thunks and only spent when the thunk is forced. We allow credit annotations only on lazy thunks in the heap:

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

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. To ensure that enough credits are available, we add a (save) rule, which saves credits on a lazy thunk. If we save credits on a memoized thunk, they will be wasted.

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

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

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

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

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

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

► **Lemma 14.** *The credit-passing cost model $(\Downarrow^C, \Phi^C, \ulcorner \cdot \urcorner^C)$ is sound.*

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

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

Unlike in the traditional Bankers 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 memoized value, which costs nothing to force. This ensures that the heaps can only become “better” over time:

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

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

$$\begin{array}{ccc} \Gamma & \xrightarrow{\text{(force)}} & \Gamma' \\ \downarrow \text{[save]} & & \downarrow \text{[refl]} \\ \Delta & \xrightarrow{\text{(force)}} & \Delta' \end{array} \quad \begin{array}{ccc} \Gamma & \xrightarrow{\text{(save)}} & \Gamma' \\ \downarrow \text{[force]} & & \downarrow \text{[force]} \\ \Delta & \xrightarrow{\text{(waste)}} & \Delta' \end{array}$$

◀

4.2 Credit Inheritance

Credit passing style wastes credits when they are saved on a memoized thunk. Lorenzen [15] proposes Credit Inheritance as an alternative to avoid wasting credits. If a thunk has an excess of credits when forced, it can designate another thunk as its *heir* and pass excess credits on. This technique enables more advanced reasoning: Credit Inheritance can be used to analyse Okasaki's Bankers Queue, while credit passing style can not do so [15].

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

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

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.

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

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 only one of the premises carries an heir:

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

The (force) rule installs the heir annotation on the memoized 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.

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

Finally, the heir annotation allows us to provide the (inherit) rule. Unlike the (waste) rule, which discards credits placed on memoized 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^{CI} for \Downarrow_k^h . 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 ($\ulcorner \Gamma, a \mapsto_h \text{memo } v \urcorner^{\text{CI}} = \ulcorner \Gamma \urcorner^{\text{CI}}, a \mapsto \text{memo } v$)

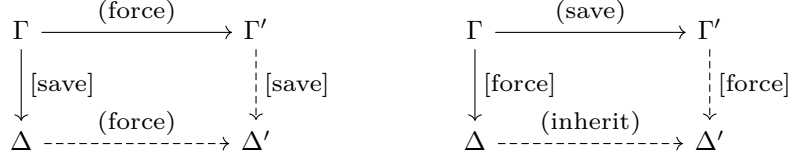
► **Lemma 16.** *The credit-inheritance cost model $(\Downarrow^{\text{CI}}, \Phi^{\text{CI}}, \ulcorner \cdot \urcorner^{\text{CI}})$ is sound.*

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

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

23:12 Persistent Amortized Analysis, Operationally

Proof. If we force a thunk that more credits have been saved on, 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.



5 Persistent Amortized 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. If the evaluation of a thunk takes k steps and the thunk has n credits, we can equivalently say that the thunk has $k - n$ debits. But as we have seen in the last section, one can perform persistent amortized analysis purely in terms of credits. If debits are not necessary for amortized analysis, 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 act as a barrier that prevents accesses 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 [18, page 60]. This requires us to remember the "price", or cost of evaluation of each thunk. We could still use credits to count up to this cost, but it is much more elegant to use debits and count this cost down to zero instead.

To illustrate this, we again define the heaps of counters inductively. We assume that all thunks are memoized, but place one debit on memoized thunks:

$$\text{Counter}(a \mapsto \text{fold } \mathbf{End} : a)$$

$$\text{Counter}(\Gamma : a) \implies \text{Counter}(\Gamma, b \mapsto \text{fold } \mathbf{Cons}(\mathbf{One}, a) : b) \quad (b \notin \text{dom}(\Gamma))$$

$$\text{Counter}(\Gamma : a) \implies \text{Counter}(\Gamma, c \mapsto \text{fold } \mathbf{Cons}(\mathbf{Zero}, b), b \mapsto_1 \text{memo } a : c) \quad (b, c \notin \text{dom}(\Gamma))$$

We use the lazy increment operation defined in Section 4, but modify it slightly by replacing $\text{save } 1$ ($\text{lazy}_{\text{Incr}} c$) by $\text{lazy}_{\text{Incr}} c$. In our proof, we now assume that all thunks we create are evaluated immediately and place the cost of that evaluation as a debit on the resulting thunk. However, we account separately for the cost of paying off debits, which is not part of the cost of evaluating thunks.

► **Lemma 18.** *If $\text{Counter}(\Gamma : c)$, then $\Gamma : \text{Incr}(c) \Downarrow_{1,1}^D \Delta : c'$ with $\text{Counter}(\Delta : c')$.*

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

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



This statement is quite similar to the sequential one (Lemma 5). 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 proof consumes credits while the persistent proof produces debits.

This proof is typical for Okasaki's debit method. Note how this proof reasons about thunks that do not exist in the data structure yet. On a counter consisting only of one-bits, our implementation 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. 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 actually exists.

5.1 The Persistent Bankers Method

Okasaki defines the amortized cost of an operation informally as the *unshared* cost of actually performing work plus the number of debits discharged [18, page 60]. To formalize 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 propagating k' from their premises to their conclusions (or returning $k' = 0$ if no premise exists).

In our heaps, we now allow debits to be placed on memoized thunks. Debts may be negative. For a memoized thunk with debits, we also record the computation Fv that was used to create it as well as the new allocations Δ created by the computation:

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

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) function allows us to access the value of a memoized 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 memoized thunk and if the thunk has no debits left, saved credits end up being wasted.

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

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

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

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

We call this the debit semantics and write $\Downarrow_{k+k'}^D$ 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 is the real cost of evaluating the thunk and n is its number of debits:

$$\begin{aligned} \Phi^D(\emptyset) &= 0 & \Phi^D(\Gamma, a \mapsto_n \text{memo}_{F_v}^\Delta w) &= \Phi^D(\Gamma) + k^R - n & \Phi^D(\Gamma, a \mapsto hv) &= \Phi^D(\Gamma) \\ \ulcorner \emptyset \urcorner^D &= \emptyset & \ulcorner \Gamma, a \mapsto_n \text{memo}_{F_v}^\Delta w \urcorner^D &= \ulcorner \Gamma \setminus \Delta \urcorner^D, a \mapsto \text{lazy}_F v & \ulcorner \Gamma, a \mapsto hv \urcorner^D &= \ulcorner \Gamma \urcorner^D, a \mapsto hv \end{aligned}$$

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

► **Example 20.** To illustrate why the semantics has to track both k and k' , consider a (lazy) rule that places both as a debit on the resulting thunk. Given a heap $\{a \mapsto_n \text{memo}_{F_1} v w\}$, we may now allocate a new thunk $\text{lazy}_{F_2}()$ for $F_2(()) = \text{save } n a$. If we were to place the discharged debits on the resulting thunk, our resulting heap would be $\{a \mapsto_0 \text{memo}_{F_1} v w, b \mapsto_{n+1} \text{memo}_{F_2} () a\}$. Effectively, we would move n debits from a to b . Since a now has no remaining debits, we could force it for free, without forcing b first. This allows us to access w without paying off its debits and thus breaks soundness.

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

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

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

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

► **Lemma 21.** *The debit cost model $((\Downarrow^D, \Phi^D, \ulcorner \cdot \urcorner^D), \sqsubseteq^D)$ is uniformly persistent.*

5.2 Debit Inheritance

In the last section, 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 [18, 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 propagate unshared work to the outside.

$$\frac{a \notin \text{dom}(\Gamma) \quad \Gamma : F v \Downarrow_{k,k'} \Delta : w}{\Gamma : \text{lazy}_F v \Downarrow_{k,k'} (\Delta, a \mapsto_0 \text{memo}_{F_v}^{\Delta \setminus \Gamma} w) : a} \text{ (lazy)}$$

This 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” [18, page 61]. The (lazy) rule makes it possible for the computation to inherit all the debits from the thunks that it creates during its evaluation.

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

► **Lemma 22.** *The debit-inheritance cost model $(\Downarrow^{\text{DI}}, \Phi^{\text{D}}, \Upsilon \cdot \Upsilon^{\text{D}})$ is sound.*

The \sqsubseteq^{DI} relation differs from \sqsubseteq^{D} only in that the new debit is 0 instead of k in the [lazy] rule.

► **Lemma 23.** *The debit-inheritance cost model $(\Downarrow^{\text{DI}}, \Phi^{\text{D}}, \Upsilon \cdot \Upsilon^{\text{D}}), \sqsubseteq^{\text{DI}}$ is uniformly persistent.*

6 Related Work

We compare to the main related works by considering several aspects of reasoning about persistent amortized complexity.

Credits and Debits

Okasaki [18] claims that debits are necessary for reasoning about persistent usage, and that using credits is unsound. His first argument centers on the lack of linearity in a persistent setting: the persistent usage of a data structure could cause credits to be spent more than once, which would break soundness. 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” [18, page 59].

However, this argument ignores an important aspect of thunks: they are only evaluated once. This guarantees that the resources they hold are used linearly, even if there are many references to the thunk itself. Pilkiewicz and Pottier [19] and Mével et al. [17] 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. Yet, these works continue to use debits in their interface of thunks. Lorenzen [15] propose a purely credit-based interface for reasoning about thunks, but do not formalize it or prove soundness.

Reasoning about the Future

Okasaki’s reasoning style usually assumes that thunks are evaluated immediately: in the absence of side-effects, it does not matter when a thunk is evaluated (shown formally by Hackett and Hutton [8]). This leads to his second argument for debits as a “layaway plan”.

However, saving credits on a thunk is a side-effect. Thus, Okasaki does not generally create debits for paying off thunks (see Example 20). He does create debits for paying off thunks in his debit-passing-style, but restricts it to those thunks that are guaranteed to be forced after the enclosing thunk [18, page 174]. This is an important difference to Danielsson’s version of debit-passing-style [3], 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.

For some data structures, like the Bankers 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 [3] assumes that thunks are evaluated on forcing and can not 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. [21] generalize

23:16 Persistent Amortized Analysis, Operationally

this principle to their ‘Think-Consequence’ rule, which allows arbitrary implications to be injected.

Side-effects 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 roll back to the lazy state when going back 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 changed. This restriction does not impact our ability to model Okasaki’s data structures, which can all be implemented without mutable references.

Pottier et al. [21] 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. Their work is embedded in Iris and can interface with code that includes references and even concurrency.

Persistence and Monotonicity

Our characterization of persistence is an instance of heap monotonicity [6]. It is inspired by the work of Pilkiewicz and Pottier [19], who show that thunks are monotonic. In Danielsson’s work [3], 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. [17] and Pottier et al. [21] show persistence using the persistent modality of Iris.

7 Conclusion

We have presented an operational account of amortized analysis in a persistent setting. Our new characterization of sound and persistent cost models allows us to precisely distinguish between approaches that are persistent and those that are not. We give the first proof of the soundness and persistence of amortized analysis with credits in a persistent setting, which refutes the folklore belief that credits are unsound for persistent amortized analysis. We also provide a new model 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 does not prescribe 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. Personally, 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 [8, 7, 14, 26].

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 [23] (and, in fact, thunks can be seen as an instance of this principle [16]). However, we additionally need to ensure monotonicity of updates. We are interested in exploring a more general interface for monotonic updates, like LVars [11] or stable references [20, 10].

References

- 1 Amal Jamil Ahmed. *Semantics of types for mutable state*. Princeton University, 2004.
- 2 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.
- 3 Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 133–144, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1328438.1328457.
- 4 James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989. doi:https://doi.org/10.1016/0022-0000(89)90034-2.
- 5 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.
- 6 Manuel Fähndrich and K Rustan M Leino. Heap monotonic tpestates. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2003.
- 7 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.
- 8 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.
- 9 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.
- 10 Haim Kaplan, Chris Okasaki, and Robert E Tarjan. Simple confluently persistent catenable lists. *SIAM Journal on Computing*, 30(3):965–977, 2000. doi:10.1007/BFb0054360.
- 11 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.
- 12 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.
- 13 Paul-Blain Levy, John Power, and Hayo Thieleck. Modelling environments in call-by-value programming languages. *Information and computation*, 185(2):182–210, 2003.

- 14 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.
- 15 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.
- 16 Anton Lorenzen, Daan Leijen, Wouter Swierstra, and Sam Lindley. First-order laziness. *Proceedings of the ACM on Programming Languages*, (ICFP), 2025. doi:10.1145/3747530.
- 17 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.
- 18 Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- 19 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.
- 20 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>.
- 21 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.
- 22 Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, July 1986. doi:10.1145/6138.6151.
- 23 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.
- 24 Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.
- 25 Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.
- 26 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), aug 2024. doi:10.1145/3674626.

A Proofs

A.1 Soundness

Lemma 4

Proof. Induction over the derivation of $\Gamma : e \Downarrow_n^B \Delta : v$. We only have to consider the (save) and (spend) rules, which are new to this cost model. In the real cost model, save m a is a no-op and spend m from a on e just evaluates e . For (save), we have:

$$\begin{aligned} \Phi^B(\Gamma, a \mapsto_{n+m} hv) - \Phi^B(\Gamma, a \mapsto_n hv) &= (\Phi^B(\Gamma) + n + m) - (\Phi^B(\Gamma) + n) \\ &= m \end{aligned}$$

For (spend), we apply the lemma inductively to the premise. Assume the real semantics takes k steps and the Bankers semantics k' steps on the premise. Then we have $k + \Phi^B(\Delta) - \Phi^B(\Gamma, a \mapsto_n hv) \leq k'$. We have:

$$\begin{aligned} k + \Phi^B(\Delta) - \Phi^B(\Gamma, a \mapsto_{n+m} hv) &= k + \Phi^B(\Delta) - (\Phi^B(\Gamma) + n + m) \\ &= k + \Phi^B(\Delta) - \Phi^B(\Gamma, a \mapsto_n hv) - m \\ &\leq k' - m \\ &\leq \max(k' - m, 0) \end{aligned}$$

◀

Lemma 14

Proof. Induction over the derivation of $\Gamma : e \Downarrow_n^C \Delta : v$. We only consider the new rules. Again, save m a is a no-op in the real cost model.

- **(lazy)**: We have $\overline{\Gamma}, a \mapsto_0 \text{lazy}_F v \overline{\Gamma}^C = \overline{\Gamma}^C, a \mapsto \text{lazy}_F v$ and

$$\begin{aligned} \Phi^C(\Gamma, a \mapsto_0 \text{lazy}_F v) - \Phi^C(\Gamma) &= (\Phi^C(\Gamma) + 0) - \Phi^C(\Gamma) \\ &= 0 \end{aligned}$$

- **(waste)**: Trivial, since save m a is a no-op in this rule.
- **(save)**: We have $\overline{\Gamma}, a \mapsto_n \text{lazy}_F v \overline{\Gamma}^C = \overline{\Gamma}^C, a \mapsto \text{lazy}_F v = \overline{\Gamma}, a \mapsto_{n+m} \text{lazy}_F v \overline{\Gamma}^C$ and

$$\begin{aligned} \Phi^C(\Gamma, a \mapsto_{n+m} \text{lazy}_F v) - \Phi^C(\Gamma, a \mapsto_n \text{lazy}_F v) &= \Phi^C(\Gamma) + n + m - \Phi^C(\Gamma) - n \\ &= m \end{aligned}$$

- **(force)**: By the induction hypothesis, we have $\overline{\Gamma}^C : F v \Downarrow_{k'}^R \overline{\Delta}^C : w$ and $k' + \Phi^C(\Delta) - \Phi^C(\Gamma) \leq k$. By the (force) rule, we have $\overline{\Gamma}, a \mapsto_n \text{lazy}_F v \overline{\Gamma}^C : \text{force } a \Downarrow_{k'}^R \overline{\Delta}, a \mapsto \text{memo } w \overline{\Gamma}^C : w$. The second condition holds as:

$$\begin{aligned} k' + \Phi^C(\Delta, a \mapsto \text{memo } w) - \Phi^C(\Gamma, a \mapsto_n \text{lazy}_F v) &= k' + \Phi^C(\Delta) - \Phi^C(\Gamma) - n \\ &\leq k - n \\ &\leq n - n \\ &= 0 \end{aligned}$$

◀

23:20 Persistent Amortized Analysis, Operationally

► **Lemma 24.** *If $\Gamma : \text{save } ma \Downarrow_m^{\text{CI}} \Delta : v$, then $\Uparrow^{\neg\text{CI}} = \Uparrow^{\neg\text{CI}}$, $\Uparrow^{\neg\text{CI}} : \text{save } ma \Downarrow_0^{\text{R}} \Uparrow^{\neg\text{CI}} : a$ and $\Phi^{\text{CI}}(\Delta) - \Phi^{\text{CI}}(\Gamma) \leq m$.*

Proof. By induction over the derivation of $\Gamma : \text{save } ma \Downarrow_m^{\text{CI}} \Delta : v$.

■ **(waste):** Trivial, since $\Phi^{\text{CI}}\Gamma - \Phi^{\text{CI}}\Gamma = 0 \leq m$.

■ **(save):** We have $\Uparrow^{\neg\text{CI}}, a \mapsto_n \text{lazy}_F v^{\neg\text{C}} = \Uparrow^{\neg\text{C}}, a \mapsto \text{lazy}_F v = \Uparrow^{\neg\text{C}}, a \mapsto_{n+m} \text{lazy}_F v^{\neg\text{C}}$ and

$$\begin{aligned} \Phi^{\text{C}}(\Gamma, a \mapsto_{n+m} \text{lazy}_F v) - \Phi^{\text{C}}(\Gamma, a \mapsto_n \text{lazy}_F v) &= \Phi^{\text{C}}(\Gamma) + n + m - \Phi^{\text{C}}(\Gamma) - n \\ &= m \end{aligned}$$

■ **(inherit):** By the induction hypothesis, we have $\Uparrow^{\neg\text{CI}} = \Uparrow^{\neg\text{CI}}$, $\Uparrow^{\neg\text{CI}} : \text{save } mh \Downarrow_0^{\text{R}} \Uparrow^{\neg\text{CI}} : h$ and $\Phi^{\text{CI}}(\Delta) - \Phi^{\text{CI}}(\Gamma) \leq m$. We have

$$\begin{aligned} \Uparrow^{\neg\text{CI}}, a \mapsto_h \text{memo } w^{\neg\text{CI}} &= \Uparrow^{\neg\text{CI}}, a \mapsto \text{memo } w \\ &= \Uparrow^{\neg\text{CI}}, a \mapsto \text{memo } w \\ &= \Uparrow^{\neg\text{CI}}, a \mapsto_h \text{memo } w^{\neg\text{CI}} \end{aligned}$$

and thus $\Uparrow^{\neg\text{CI}}, a \mapsto_h \text{memo } w^{\neg\text{CI}} : \text{save } ma \Downarrow_0^{\text{R}} \Uparrow^{\neg\text{CI}}, a \mapsto_h \text{memo } w^{\neg\text{CI}} : a$ and

$$\begin{aligned} \Phi^{\text{CI}}(\Delta, a \mapsto_h \text{memo } w) - \Phi^{\text{CI}}(\Gamma, a \mapsto_h \text{memo } w) &= \Phi^{\text{CI}}(\Delta) - \Phi^{\text{CI}}(\Gamma) \\ &\leq m \end{aligned}$$

◀

Lemma 16

Proof. By induction over the derivation of $\Gamma : e \Downarrow_n^{\text{CI}} \Delta : v$. We only consider the new rules. Again, $\text{save } ma$ and $\text{pass } h$ are no-ops in the real cost model.

■ **(pass):** By Lemma 24, we have $\Uparrow^{\neg\text{CI}} = \Uparrow^{\neg\text{CI}}$, $\Uparrow^{\neg\text{CI}} : \text{save } mh \Downarrow_0^{\text{R}} \Uparrow^{\neg\text{CI}} : h$ and $\Phi^{\text{CI}}(\Delta) - \Phi^{\text{CI}}(\Gamma) \leq m$. Since $\text{pass } h$ is a no-op in the real cost model, this concludes the proof.

■ **(force):** By the induction hypothesis, we have $\Uparrow^{\neg\text{C}} : Fv \Downarrow_{k'}^{\text{R}} \Uparrow^{\neg\text{C}} : w$ and $k' + \Phi^{\text{C}}(\Delta) - \Phi^{\text{C}}(\Gamma) \leq k$. By the (force) rule, we have $\Uparrow^{\neg\text{C}}, a \mapsto_n \text{lazy}_F v^{\neg\text{C}} : \text{force } a \Downarrow_{k'}^{\text{R}} \Uparrow^{\neg\text{C}}, a \mapsto_h \text{memo } w^{\neg\text{C}} : w$. The second condition holds as:

$$\begin{aligned} k' + \Phi^{\text{C}}(\Delta, a \mapsto_h \text{memo } w) - \Phi^{\text{C}}(\Gamma, a \mapsto_n \text{lazy}_F v) &= k' + \Phi^{\text{C}}(\Delta) - \Phi^{\text{C}}(\Gamma) - n \\ &\leq k - n \\ &= n - n \\ &= 0 \end{aligned}$$

■ **(inherit):** By Lemma 24. ◀

► **Remark 25.** For the debit cost model, we ensure the following invariant on heaps: For every heap $\Gamma, a \mapsto_k \text{memo}_{Fv}^{\Delta} w$, we request that $\Uparrow \setminus \Delta^{\text{D}} : Fv \Downarrow_{k^{\text{R}}}^{\text{R}} \Uparrow^{\text{D}} : w$ and that $k \leq k^{\text{R}}$.

► **Lemma 26.** *If $\Gamma : e \Downarrow_{k,k'}^{\text{D}} \Delta : v$, and $\Uparrow^{\text{D}} : e \Downarrow_{k^{\text{R}}}^{\text{R}} \Uparrow^{\text{D}} : v$, then $k \leq k^{\text{R}}$.*

Proof. By induction over the derivation of $\Gamma : e \Downarrow_{k,k'}^{\text{D}} \Delta : v$. The property hold for all rules. ◀

Lemma 19

Proof. We first show that both semantics produce the same final heap and value. By induction over the derivation of $\Gamma : e \Downarrow_{k,k'}^D \Delta : v$. We only consider the new rules.

- **(lazy):** We have

$$\begin{aligned} \ulcorner \Delta, a \mapsto_k \text{memo}_{Fv}^{\Delta \setminus \Gamma} w \urcorner^D &= \ulcorner \Delta \setminus (\Delta \setminus \Gamma) \urcorner^D, a \mapsto \text{lazy}_F v \\ &= \ulcorner \Gamma \urcorner^D, a \mapsto \text{lazy}_F v \end{aligned}$$

By the induction hypothesis, we have $\ulcorner \Gamma \urcorner^D : e \Downarrow_k^R \ulcorner \Delta \urcorner^D : v$, and thus our invariant holds.

- **(force):** We have

$$\ulcorner \Gamma, a \mapsto_n \text{memo}_{Fv}^{\Delta} w \urcorner^D = \ulcorner \Gamma \setminus \Delta \urcorner^D, a \mapsto \text{lazy}_F v$$

and

$$\ulcorner \Gamma, a \mapsto \text{memo } w \urcorner^D = \ulcorner \Gamma \urcorner^D, a \mapsto \text{memo } w$$

By our invariant, we have $\ulcorner \Gamma \setminus \Delta \urcorner^D : Fv \Downarrow_k^R \ulcorner \Gamma \urcorner^D : w$. We can thus apply the (force) rule in the real cost model to obtain the claim.

- **(save), (waste):** Trivial, since $\ulcorner \Gamma, a \mapsto_n \text{memo}_{Fv}^{\Delta} w \urcorner^D = \ulcorner \Gamma \urcorner^D, a \mapsto \text{memo } w = \ulcorner \Gamma, a \mapsto_{n-m} \text{memo}_{Fv}^{\Delta} w \urcorner^D$.

Now we show the cost condition. By induction over the derivation of $\Gamma : e \Downarrow_{k,k'}^D \Delta : v$. We only consider the new rules.

- **(lazy):** Let k^R be the time taken for the thunk in the real semantics. By the induction hypothesis, we have $k^R + \Phi^D(\Delta) - \Phi^D(\Gamma) \leq k + k'$. We have

$$\begin{aligned} \Phi^D(\Delta, a \mapsto_k \text{memo}_{Fv}^{\Delta} w) - \Phi^D(\Gamma) &= (\Phi^D(\Delta) + k^R - k) - \Phi^D(\Gamma) \\ &\leq k^R - k + k + k' - k^R \\ &\leq k' \end{aligned}$$

By Lemma 26, we have $k \leq k^R$, which ensures that Φ^D remains non-negative.

- **(force):** Let k^R be the time taken for the thunk in the real semantics.

$$\begin{aligned} k^R + \Phi^D(\Gamma, a \mapsto \text{memo } w) - \Phi^D(\Gamma, a \mapsto_n \text{memo}_{Fv}^{\Delta} w) \\ &= k^R + \Phi^D(\Gamma) - \Phi^D(\Gamma) - k^R + n \\ &\leq n \\ &\leq 0 \end{aligned}$$

- **(save):** We have

$$\begin{aligned} \Phi^D(\Gamma, a \mapsto_{n-m} \text{memo}_{Fv}^{\Delta} w) - \Phi^D(\Gamma, a \mapsto_n \text{memo}_{Fv}^{\Delta} w) \\ &= (\Phi^D(\Gamma) + k^R - (n - m)) - (\Phi^D(\Gamma) + k^R - n) \\ &= m - n + n = m \end{aligned}$$

- **(waste):** Trivial, since it doesn't change the heap and $0 \leq m$. ◀

► **Remark 27.** For the debit inheritance cost model, we relax the invariant that $k \leq k^R$ for every memoized thunk and include on the right hand side the costs of the children thunks as well.

23:22 Persistent Amortized Analysis, Operationally

► **Lemma 28.** *If $\Gamma : e \Downarrow_{k,k'}^D \Delta : v$, $\Gamma^{\neg D} : e \Downarrow_{k^R}^R \Delta^{\neg D} : v$, and k_1^R, \dots, k_n^R are the real costs of the children thunks, then $k \leq k^R + k_1^R + \dots + k_n^R$.*

Proof. By induction over the derivation of $\Gamma : e \Downarrow_{k,k'}^D \Delta : v$. We have $k \leq k^R$ in all rules except for the new (lazy) rule with inheritance. In that case, we have to charge for the time taken to evaluate the thunk as well. ◀

Lemma 22

Proof. By induction over the derivation of $\Gamma : e \Downarrow_{k,k'}^{DI} \Delta : v$. We only have to consider the new credit annotation on the (lazy) rule.

■ **(lazy):** Let k^R be the time taken for the thunk in the real semantics. By the induction hypothesis, we have $k^R + \Phi^D(\Delta) - \Phi^D(\Gamma) \leq k + k'$. We have

$$\begin{aligned} \Phi^D(\Delta, a \mapsto \text{memo}_{F,v}^{\Delta} w) - \Phi^D(\Gamma) &= (\Phi^D(\Delta) + k^R) - \Phi^D(\Gamma) \\ &\leq k^R + k + k' - k^R \\ &\leq k + k' \end{aligned}$$

◀

A.2 Accessibility

To show results about persistence, it is convenient to consider only that subset of the big-step rules that interact with the heap. Rules like (case), (split), (app) or (call) are independent of the heap and do not affect persistence. We thus define a version of the relation \sqsubseteq that characterizes expressions that interact with the heap. We write $\Gamma \sqsubseteq_{e,k,w} \Delta$ if $\Gamma : e \Downarrow_k \Delta : w$. We write $\sqsubseteq_{-,k,w}$ if different e yield the same relation.

For the real cost model, the rules are as follows:

$$\begin{aligned} \frac{}{\Gamma \sqsubseteq_{v,0,v} \Gamma} \text{(refl)} \quad & \frac{\Gamma \sqsubseteq_{e_1,k,v} \Delta \quad \Delta \sqsubseteq_{e_2[v/x],l,w} \Theta}{\Gamma \sqsubseteq_{\text{let } x=e_1 \text{ in } e_2, k+l, w} \Theta} \text{(trans)} \quad & \frac{\Gamma \sqsubseteq_{e,k,w} \Delta}{\Gamma \sqsubseteq_{-,k,w} \Delta} \text{(step)} \\ \frac{a \notin \text{dom}(\Gamma)}{\Gamma \sqsubseteq_{-,0,a} \Gamma, a \mapsto hv} \text{(alloc)} \quad & \frac{a \mapsto \text{fold } v \in \Gamma}{\Gamma \sqsubseteq_{\text{unfold } a,0,v} \Gamma} \text{(unfold)} \\ & \frac{\Gamma \sqsubseteq_{Fv,k,w} \Delta}{(\Gamma, a \mapsto \text{lazy}_F v) \sqsubseteq_{\text{force } a,k,w} (\Delta, a \mapsto \text{memo } w)} \text{(force)} \\ & \frac{a \mapsto \text{memo } w \in \Gamma}{\Gamma \sqsubseteq_{\text{force } a,0,w} \Gamma} \text{(recall)} \end{aligned}$$

► **Lemma 29.** $\Gamma \sqsubseteq_{e,k,w}^R \Delta$ iff $\Gamma : e \Downarrow_k^R \Delta : w$.

Proof. The rules of $\sqsubseteq_{e,k,w}$ relate to the big-step semantics as follows:

- **(refl):** (value)
- **(trans):** (let)
- **(step):** (case), (split), (app), (call)
- **(alloc):** (fold), (lazy), (memo)
- **(unfold):** (unfold)
- **(force):** (force)
- **(recall):** (recall)

◀

► **Lemma 30.** $\Gamma \sqsubseteq^R \Delta$ iff there exists e, k, w such that $\Gamma \sqsubseteq_{e,k,w}^R \Delta$.

Proof. Direction (\Rightarrow): By induction on $\Gamma \sqsubseteq \Delta$. We maintain that w consists of nested pairs that contain all heap locations in $\Delta \setminus \Gamma$.

- **[refl]**: Immediately by the (refl) rule using $e = w = ()$.
- **[trans]**: Apply the induction hypothesis on both premises, yielding $\Gamma \sqsubseteq_{e_1,k,v} \Delta$ and $\Delta \sqsubseteq_{e_2,l,w} \Theta$. Let $e = \text{let } x = e_1 \text{ in } \dots \text{let } y = e_2 \text{ in } (x, y)$, where \dots splits v into its components. We obtain $\Gamma \sqsubseteq_{e,k',(v,w)} \Theta$ by the (trans) and (step) rules for some k' .
- **[alloc]**: By the (alloc) rule.
- **[force]**: Apply Lemma 29 on the premise and then use the (force) rule.

Direction (\Leftarrow): By induction on $\Gamma \sqsubseteq_{e,k,w} \Delta$.

- **(refl), (alloc)**: Immediately by the [refl]/[alloc] rule.
- **(trans)**: Apply the induction hypothesis on both premises and then use the [trans] rule.
- **(step)**: Apply the induction hypothesis on the premise.
- **(unfold), (recall)**: Use [refl].
- **(force)**: Apply Lemma 29 on the premise to obtain $\Gamma : Fv \Downarrow_k^R \Delta : w$. Use the induction hypothesis to obtain $\Gamma \sqsubseteq \Delta$. Then use the [force] rule.

◀

► **Lemma 31.** The real cost model $((\Downarrow^R, \Phi^R, \ulcorner \cdot \urcorner^R), \sqsubseteq^R)$ is accessible.

Proof. By Lemma 29 and Lemma 30.

◀

For the Bankers cost model, we add the following two rules:

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

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

► **Lemma 32.** $\Gamma \sqsubseteq_{e,k,w}^B \Delta$ iff $\Gamma : e \Downarrow_k^B \Delta : w$.

Proof. The rules of $\sqsubseteq_{e,k,w}$ relate to the big-step semantics as follows:

- **(save)**: (save)
- **(spend)**: (spend)

◀

► **Lemma 33.** $\Gamma \sqsubseteq^B \Delta$ iff there exists e, k, w such that $\Gamma \sqsubseteq_{e,k,w}^B \Delta$.

Proof. Direction (\Rightarrow): By induction on $\Gamma \sqsubseteq \Delta$.

- **[save]**: Immediately by the (save) rule.
- **[spend]**: Apply the (spend) rule using $e = \text{spend } m \ \text{from } a \ \text{on } ()$ and the [refl] rule.

Direction (\Leftarrow): By induction on $\Gamma \sqsubseteq_{e,k,w}^B \Delta$.

- **(save)**: Immediately by the [save] rule.
- **(spend)**: Apply the induction hypothesis on the premise, obtaining $\Gamma, a \mapsto_m hv \sqsubseteq \Delta$. Then use the [spend] rule and join the two derivations using [trans].

◀

Lemma 11

Proof. By Lemma 32 and Lemma 33. ◀

For the credit-passing cost model, we add the following three rules:

$$\frac{hv = \text{lazy}_F v}{(\Gamma, a \mapsto_n hv) \sqsubseteq_{\text{save } m a, m, a} (\Gamma, a \mapsto_{n+m} hv)} \text{ (save)} \quad \frac{a \mapsto \text{memo } v \in \Gamma}{\Gamma \sqsubseteq_{\text{save } m a, m, a} \Gamma} \text{ (waste)}$$

$$\frac{\Gamma \sqsubseteq_{F v, k, w} \Delta \quad k \leq n}{(\Gamma, a \mapsto_n \text{lazy}_F v) \sqsubseteq_{\text{force } a, 0, w} (\Delta, a \mapsto \text{memo } w)} \text{ (force)}$$

► **Lemma 34.** $\Gamma \sqsubseteq_{e, k, w}^C \Delta$ iff $\Gamma : e \Downarrow_k^C \Delta : w$.

Proof. The rules of $\sqsubseteq_{e, k, w}$ relate to the big-step semantics as follows:

- **(save)**: (save)
- **(waste)**: (waste)
- **(force)**: (force)

We continue using the (recall) and (lazy) rules from the real cost model. ◀

► **Lemma 35.** $\Gamma \sqsubseteq^C \Delta$ iff there exists e, k, w such that $\Gamma \sqsubseteq_{e, k, w}^C \Delta$.

Proof. Direction (\Rightarrow): By induction on $\Gamma \sqsubseteq \Delta$.

- **[save]**: Immediately by the (save) rule.
- **[force]**: Apply Lemma 34 on the premise, obtaining $\Gamma \sqsubseteq_{F v, k, w} \Delta$. Then use the (force) rule.

Direction (\Leftarrow): By induction on $\Gamma \sqsubseteq_{e, k, w}^C \Delta$.

- **(save)**: Immediately by the [save] rule.
- **(waste)**: Immediately by the [refl] rule.
- **(force)**: Apply Lemma 34 on the premise, obtaining $\Gamma : F v \Downarrow_k^C \Delta : w$. Use the induction hypothesis to obtain $\Gamma \sqsubseteq \Delta$. Then use the [force] rule. ◀

► **Lemma 36.** The credit-passing cost model $((\Downarrow^C, \Phi^C, \ulcorner \cdot \urcorner^C), \sqsubseteq^C)$ is accessible.

Proof. By Lemma 34 and Lemma 35. ◀

For the credit-inheritance cost model, we parameterize the $\sqsubseteq_{e, k, v}$ relation with a heir.

$$\frac{\Gamma \sqsubseteq_{\text{save } m h, m, v}^{\emptyset} \Delta}{\Gamma \sqsubseteq_{\text{pass } h, m, v}^{\{h\}} \Delta} \text{ (pass)} \quad \frac{\Gamma \sqsubseteq_{F v, k, w}^{\{h\}} \Delta \quad k = n}{(\Gamma, a \mapsto_n \text{lazy}_F v) \sqsubseteq_{\text{force } a, 0, w}^{\emptyset} (\Delta, a \mapsto_h \text{memo } w)} \text{ (force)}$$

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

► **Lemma 37.** $\Gamma \sqsubseteq_{e, k, w}^h \Delta$ iff $\Gamma : e \Downarrow_k^h \Delta : w$.

Proof. The rules of $\sqsubseteq_{e, k, w}$ relate to the big-step semantics as follows:

- **(pass)**: (pass)
- **(force)**: (force)
- **(inherit)**: (inherit) ◀

► **Lemma 38.** $\Gamma \sqsubseteq^{\text{CI}} \Delta$ iff there exists e, k, w such that $\Gamma \sqsubseteq_{e,k,w}^h \Delta$.

Proof. Direction (\Rightarrow): By Lemma 35. Direction (\Leftarrow):

- **(pass)**: Use the induction hypothesis on the premise to obtain $\Gamma \sqsubseteq^{\text{CI}} \Delta$.
- **(force)**: Apply Lemma 37 on the premise, obtaining $\Gamma : Fv \Downarrow_k^{\{h\}} \Delta : w$. Use the induction hypothesis to obtain $\Gamma \sqsubseteq^{\text{CI}} \Delta$. Then use the [force] rule.
- **(inherit)**: Use the induction hypothesis on the premise to obtain $\Gamma \sqsubseteq^{\text{CI}} \Delta$. Then show by induction on $\Gamma \sqsubseteq^{\text{CI}} \Delta$ that $\Gamma, a \mapsto_h \text{memo } w \sqsubseteq^{\text{CI}} \Delta, a \mapsto_h \text{memo } w$.

► **Lemma 39.** The credit-inheritance cost model $((\Downarrow^{\text{CI}}, \Phi^{\text{CI}}, \ulcorner \cdot \urcorner^{\text{CI}}), \sqsubseteq^{\text{CI}})$ is accessible.

Proof. By Lemma 37 and Lemma 38.

For the debit cost model, we add the following four rules:

$$\frac{\Gamma \sqsubseteq_{Fv,k,k',w} \Delta \quad a \notin \text{dom}(\Delta)}{\Gamma \sqsubseteq_{\text{lazy}_F v,0,k',a} (\Delta, a \mapsto_k \text{memo}_{Fv}^{\Delta \setminus \Gamma} w)} \text{ (lazy)} \quad \frac{\text{memo } w \in \Gamma}{\Gamma \sqsubseteq_{\text{save } m a,0,m,a} \Gamma} \text{ (waste)}$$

$$\frac{hw = \text{memo}_{Fv}^{\Delta} w}{(\Gamma, a \mapsto_n hv) \sqsubseteq_{\text{save } m a,0,m,a} (\Gamma, a \mapsto_{n-m} hv)} \text{ (save)}$$

$$\frac{n \leq 0}{(\Gamma, a \mapsto_n \text{memo}_{Fv}^{\Delta} w) \sqsubseteq_{\text{force } a,0,0,w} (\Gamma, a \mapsto \text{memo } w)} \text{ (force)}$$

► **Lemma 40.** $\Gamma \sqsubseteq_{e,k,k',w}^{\text{D}} \Delta$ iff $\Gamma : e \Downarrow_{k,k'}^{\text{D}} \Delta : w$.

Proof. The rules of $\sqsubseteq_{e,k,k',w}$ relate to the big-step semantics as follows:

- **(lazy)**: (lazy)
- **(waste)**: (waste)
- **(save)**: (save)
- **(force)**: (force)

► **Lemma 41.** $\Gamma \sqsubseteq^{\text{D}} \Delta$ iff there exists e, k, k', w such that $\Gamma \sqsubseteq_{e,k,k',w}^{\text{D}} \Delta$.

Proof. Direction (\Rightarrow): By induction on $\Gamma \sqsubseteq \Delta$.

- **[lazy]**: Apply Lemma 40 on the premise, obtaining $\Gamma \sqsubseteq_{Fv,k,k',w} \Delta$. Use the induction hypothesis to obtain $\Gamma \sqsubseteq^{\text{D}} \Delta$. Then use the (lazy) rule.
- **[save]**: Immediately by the (save) rule.
- **[force]**: Immediately by the (force) rule.

Direction (\Leftarrow): By induction on $\Gamma \sqsubseteq_{e,k,k',w}^{\text{D}} \Delta$.

- **(lazy)**: Apply Lemma 40 on the premise, obtaining $\Gamma : Fv \Downarrow_{k,k'}^{\text{D}} \Delta : w$. Then use the [lazy] rule.
- **(waste)**: Immediately by the [refl] rule.
- **(save)**: Immediately by the [save] rule.
- **(force)**: Immediately by the [force] rule.

► **Lemma 42.** The debit cost model $((\Downarrow^{\text{D}}, \Phi^{\text{D}}, \ulcorner \cdot \urcorner^{\text{D}}), \sqsubseteq^{\text{D}})$ is accessible.

Proof. By Lemma 40 and Lemma 41.

For the debit inheritance cost model, we change the (lazy) rule:

$$\frac{\Gamma \sqsubseteq_{Fv,k,k',w} \Delta \quad a \notin \text{dom}(\Delta)}{\Gamma \sqsubseteq_{\text{lazy}_F v,k,k',a} (\Delta, a \mapsto_0 \text{memo}_{Fv}^{\Delta \setminus \Gamma} w)} \text{ (lazy)}$$

► **Lemma 43.** *The debit inheritance cost model $((\Downarrow^{\text{DI}}, \Phi^{\text{D}}, \ulcorner \cdot \urcorner^{\text{D}}), \sqsubseteq^{\text{DI}})$ is accessible.*

Proof. As in Lemma 42. ◀

A.3 Persistence

In the last section, we developed the $\Gamma \sqsubseteq_{e,k,w} \Delta$ relation that holds if and only if there is a derivation $\Gamma : e \Downarrow_k \Delta : w$ of the big-step semantics, but abstracts from the concrete expression e . Since the two notions are equivalent, we prove persistence using this relation:

► **Corollary 44.** *An accessible cost model $((\Downarrow, \Phi, \ulcorner \cdot \urcorner), \sqsubseteq)$ is persistent iff $\Gamma \sqsubseteq \Delta$ and $\Gamma \sqsubseteq_{e,n,w} \Gamma'$ implies $\Delta \sqsubseteq_{e,k,w} \Delta'$ for $k \leq n$, $\Gamma' \sqsubseteq \Delta'$.*

$$\begin{array}{ccc} \Gamma & \xrightarrow{\sqsubseteq_{e,n,w}} & \Gamma' \\ \sqsubseteq \downarrow & & \downarrow \sqsubseteq \\ \Delta & \xrightarrow{\sqsubseteq_{e,k,w}} & \Delta' \end{array} \quad (k \leq n)$$

► **Lemma 45.** *If $\Gamma \sqsubseteq_{e,k,v}^{\text{R}} \Delta$ and $b \notin \text{dom}(\Delta)$, then $\Gamma, b \mapsto hv \sqsubseteq_{e,k,v}^{\text{R}} \Delta, b \mapsto hv$.*

Proof. By induction over the derivation of $\Gamma \sqsubseteq_{e,k,v}^{\text{R}} \Delta$.

- **(refl):** Trivial
- **(trans):** Use the induction hypothesis on both premises and apply (trans).
- **(step):** Use the induction hypothesis on the premise and then apply (step).
- **(alloc):** Trivial, as $a \neq b$ since $b \notin \text{dom}(\Delta)$.
- **(unfold), (recall):** Trivial
- **(force):** Since $b \notin \text{dom}(\Delta)$, $a \neq b$. Use the induction hypothesis on the premise and then apply (force). ◀

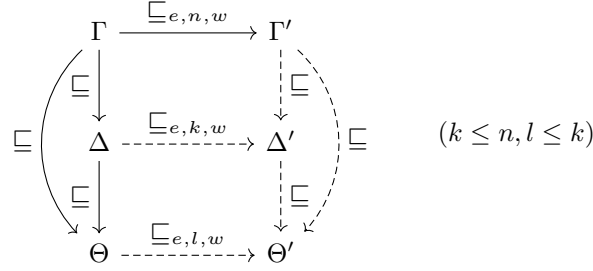
Lemma 9

Proof. By induction over $\Gamma \sqsubseteq_{e,n,w} \Gamma'$.

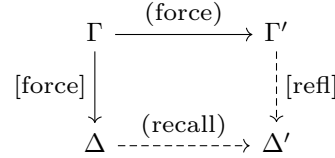
- **(refl):** Using (refl).
- **(trans):** Assume that $\Gamma \sqsubseteq_{e_1,k,v} \Gamma''$ and $\Gamma'' \sqsubseteq_{e_2[v/x],l,w} \Gamma'$. Use the induction hypothesis on the first premise, yielding $\Delta \sqsubseteq_{e_1,k',v} \Delta''$ with $k' \leq k$, $\Gamma'' \sqsubseteq \Delta''$. Then apply the induction hypothesis on the second premise, yielding $\Delta'' \sqsubseteq_{e_2[v/x],l',w} \Delta'$ with $l' \leq l$, $\Gamma' \sqsubseteq \Delta'$. Then use (trans) where $k' + l' \leq k + l$ and $\Gamma' \sqsubseteq \Delta'$.

$$\begin{array}{ccc} \Gamma & \xrightarrow{\sqsubseteq_{\text{let } x=e_1 \text{ in } e_2, k+l, w}} & \Gamma' \\ \Gamma \xrightarrow{\sqsubseteq_{e_1, k, v}} \Gamma'' \xrightarrow{\sqsubseteq_{e_2[v/x], l, w}} \Gamma' & & \\ \sqsubseteq \downarrow & & \downarrow \sqsubseteq \\ \Delta & \xrightarrow{\sqsubseteq_{e_1, k', v}} \Delta'' \xrightarrow{\sqsubseteq_{e_2[v/x], l', w}} \Delta' & \\ \Delta \xrightarrow{\sqsubseteq_{\text{let } x=e_1 \text{ in } e_2, k'+l', w}} \Delta' & & \end{array} \quad (k' \leq k, l' \leq l)$$

- **(step)**: Use the induction hypothesis on the premise and then apply (step).
- **(alloc)**: We assume w.l.o.g. that allocations never clash. Thus $a \notin \text{dom}(\Delta)$ and we use (alloc). Further, $\Gamma \sqsubseteq \Delta$ implies $\Gamma, a \mapsto hv \sqsubseteq \Delta, a \mapsto hv$ by Lemma 45 and Lemma 30.
- other cases: By induction over $\Gamma \sqsubseteq \Delta$.
 - $_$, **[refl]**: By assumption.
 - $_$, **[trans]**: Use the induction hypothesis on both premises, yielding $\Delta \sqsubseteq_{e,k,w} \Delta'$ with $k \leq n$, $\Gamma' \sqsubseteq \Delta'$ and $\Theta \sqsubseteq_{e,l,w} \Theta'$ with $l \leq k$, $\Delta' \sqsubseteq \Theta'$. Then $l \leq n$ and $\Gamma' \sqsubseteq \Theta'$.



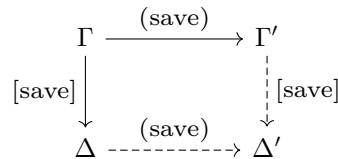
- $_$, **[alloc]**: We assume w.l.o.g. that allocations never clash. Thus $a \notin \text{dom}(\Gamma')$ and we use [alloc]. By Lemma 45, $\Gamma \sqsubseteq_{e,k,v} \Gamma'$ implies $\Gamma, a \mapsto hv \sqsubseteq_{e,k,v} \Gamma', a \mapsto hv$. We have $\Gamma' \sqsubseteq \Gamma', a \mapsto hv$ by [alloc].
- **(unfold)**, **[force]**: Let unfold a and force b . We have that $a \neq b$, since $a \mapsto \text{fold } v$ and $b \mapsto \text{lazy}_F v$. By the induction hypothesis, $\Delta \sqsubseteq_{\text{unfold } a,0,v} \Delta$. We apply Lemma 45 to obtain $\Delta, b \mapsto \text{memo } w \sqsubseteq_{\text{unfold } a,0,v} \Delta, b \mapsto \text{memo } w$.
- **(recall)**, **[force]**: Let force a and force b . We have that $a \neq b$, since $a \mapsto \text{memo } w'$ and $b \mapsto \text{lazy}_F v$. By the induction hypothesis, $\Delta \sqsubseteq_{\text{force } a,0,w'} \Delta'$. We apply Lemma 45 to obtain $\Delta, b \mapsto \text{memo } w \sqsubseteq_{\text{force } a,0,w'} \Delta', b \mapsto \text{memo } w$.
- **(force)**, **[force]**: Let force a and force b . If $a \neq b$, we proceed as in the (recall), [force] case. If $a = b$, then apply the (recall) rule to obtain $\Delta, b \mapsto \text{memo } w \sqsubseteq_{\text{force } a,0,w} \Delta, b \mapsto \text{memo } w$. We have $1 \leq k + 1$ and $\Delta, b \mapsto \text{memo } w \sqsubseteq \Delta, b \mapsto \text{memo } w$ by [refl].



Lemma 15

Proof. We extend the proof of Lemma 9 for the new rules. By induction over $\Gamma \sqsubseteq \Delta$.

- **(unfold)**, **[save]**: Let unfold a and save $m b$. We have that $a \neq b$, since $a \mapsto \text{fold } v$ and $b \mapsto_n \text{lazy}_F v$. Then $\Delta \sqsubseteq_{\text{unfold } a,0,v} \Delta$.
- **(recall)**, **[save]**: Let force a and save $m b$. We have that $a \neq b$, since $a \mapsto \text{memo } w$ and $b \mapsto_n \text{lazy}_F v$. Then $\Delta \sqsubseteq_{\text{force } a,0,w} \Delta$.
- **(save)**, **[save]**: Let save $m a$ and save $m' b$. If $a \neq b$, we proceed as in the previous cases. If $a = b$, then apply the [save] rule to obtain $\Delta, a \mapsto_{n+m'} \text{lazy}_F v \sqsubseteq_{\text{save } m a, m, a} \Delta, a \mapsto_{n+m'+m} \text{lazy}_F v$.



23:28 Persistent Amortized Analysis, Operationally

- **(waste)**, **[save]**: Let $\text{save } m a$ and $\text{save } m' b$. We have that $a \neq b$, since $a \mapsto_n \text{lazy}_F v$ and $b \mapsto \text{memo } w$. Then $\Delta \sqsubseteq_{\text{save } m a, m, a} \Delta$.
- **(force)**, **[save]**: Let $\text{force } a$ and $\text{save } m b$. If $a \neq b$, we proceed as in the previous cases. If $a = b$, then apply the **[force]** rule to obtain $\Gamma, a \mapsto_{n+m} \text{lazy}_F v \sqsubseteq_{\text{force } a, 0, a} \Delta, a \mapsto \text{memo } w$ since $k \leq n$ implies $k \leq n + m$.

$$\begin{array}{ccc} \Gamma & \xrightarrow{\text{(force)}} & \Gamma' \\ \text{[save]} \downarrow & & \downarrow \text{[refl]} \\ \Delta & \xrightarrow{\text{(force)}} & \Delta' \end{array}$$

- **(save)**, **[force]**: Let $\text{save } m a$ and $\text{force } b$. If $a \neq b$, we proceed as in the previous cases. If $a = b$, then apply the **(waste)** rule to obtain $\Delta, a \mapsto \text{memo } w \sqsubseteq_{\text{save } m a, m, a} \Delta, a \mapsto \text{memo } w$.

$$\begin{array}{ccc} \Gamma & \xrightarrow{\text{(save)}} & \Gamma' \\ \text{[force]} \downarrow & & \downarrow \text{[force]} \\ \Delta & \xrightarrow{\text{(waste)}} & \Delta' \end{array}$$

- **(waste)**, **[force]**: Let $\text{save } m a$ and $\text{force } b$. We have that $a \neq b$, since $a \mapsto \text{memo } w$ and $b \mapsto_n \text{lazy}_F v$. Then $\Delta \sqsubseteq_{\text{force } b, 0, w'} \Delta'$.
- **((unfold), [force]), ((recall), [force]), ((force), [force])**: As in Lemma 9. ◀

In the credit inheritance semantics, the number of time steps is not a deterministic function of the initial heap and expression. Instead, if there is an heir, we can choose the amount of steps non-deterministically: $\Downarrow^{\{h\}}: (\text{Heap} \times \text{Expr} \times \mathbb{N}) \rightarrow \text{Heap} \times \text{Value}$. However, we can show that if we use more steps than necessary, this only results in an increase in the credit assigned to the heir.

► **Lemma 46.** *Let $\Gamma : e \Downarrow_n^{\{h\}} \Delta : w$ and $\Gamma' : e \Downarrow_m^{\{h\}} \Delta' : w'$ for $n \leq m$. Then $w = w'$ and $\Delta \sqsubseteq_{\text{save } m-n h, m-n, v} \Delta'$.*

Proof. We strengthen the lemma to allow the starting heap to differ:

Let $\Gamma : e \Downarrow_n^{\{h\}} \Delta : w$ and $\Gamma' : e \Downarrow_m^{\{h\}} \Delta' : w'$ for $n \leq m$ and $\Gamma \sqsubseteq_{\text{save } k'-k h, k'-k, v} \Gamma'$. Then $w = w'$ and $\Delta \sqsubseteq_{\text{save } k'-k+m-n h, k'-k+m-n, v} \Delta'$.

By induction over the derivations. All cases follow directly from the induction hypothesis, except for the **(pass)**, **(save)** and **(force)** rules.

- **(pass)**: We have:

$$\begin{array}{l} \Gamma \sqsubseteq_{\text{save } k'-k h, k'-k, h} \Gamma' \\ \Gamma' \sqsubseteq_{\text{pass } h, m, h}^{\{h\}} \Delta' \\ \Gamma \sqsubseteq_{\text{pass } h, n, h}^{\{h\}} \Delta \end{array}$$

The **save** rule does not change the heap except for adding credits. This implies that $\text{save } \cdot h$ credits the same cell on all heaps. We can combine the first two facts to obtain $\Gamma \sqsubseteq_{\text{save } k'-k+m h, k'-k+m, h} \Delta'$. By the last fact, thus $\Delta \sqsubseteq_{\text{save } k'-k+m-n h, k'-k+m-n, h} \Delta'$.

- **(save)**: Let $\text{save } m a$. Then $n = m$ and $w = w'$ by the induction hypothesis. If $a \neq h$, the claim follows by induction. Let $a = h$. We have $\Gamma \sqsubseteq_{\text{save } k'-k h, k'-k, v} \Gamma'$ by the premise. Again, we can combine this with the **(save)** rule to obtain $\Delta \sqsubseteq_{\text{save } k'-k+m h, k'-k+m, v} \Delta'$

- **(force)**: Let force a . Then $n = m$ and $w = w'$ by the induction hypothesis. If $a \neq h$, the claim follows by induction. Let $a = h$. We have $\Gamma \sqsubseteq_{\text{save } k'-k, h, k'-k, v} \Gamma'$ by the premise. Let $a \mapsto_k \text{lazy}_F v \in \Gamma$ and $a \mapsto_{k'} \text{lazy}_F v \in \Gamma'$. By the induction hypothesis, we get $\Gamma : Fv \Downarrow_k^{\{h'\}} \Delta : w$ and $\Gamma' : Fv \Downarrow_{k'}^{\{h'\}} \Delta' : w$ and $\Delta \sqsubseteq_{\text{save } k'-k, h', k'-k, v} \Delta'$. In the resulting heaps Δ and Δ' , we now have $h \mapsto_{h'} \text{memo } w$. By the (inherit) rule, we get $\Delta \sqsubseteq_{\text{save } k'-k, h, k'-k, v} \Delta'$. ◀

Lemma 17

Proof. We extend the proof of Lemma 15 for the new rules.

- **(pass)**: Follows directly from the induction hypothesis on the premise.
- **(inherit)**: By the induction hypothesis on the premise, we have $\Delta \sqsubseteq_{\text{save } m, h, m, h} \Delta'$ and $\Gamma' \sqsubseteq \Delta'$. We then get $\Delta, a \mapsto_h \text{memo } w \sqsubseteq_{\text{save } m, a, m, a} \Delta', a \mapsto_h \text{memo } w$ by the (inherit) rule. We have $\Gamma', a \mapsto_h \text{memo } w \sqsubseteq \Delta', a \mapsto_h \text{memo } w$ by Lemma 45.
- **other cases**:: By induction over $\Gamma \sqsubseteq \Delta$.
 - **(save), [force]**: Let save $m a$ and force b . If $a \neq b$, we proceed as in the previous proof. If $a = b$, then $\Gamma = \Gamma_1, a \mapsto_n \text{lazy}_F v$ and $\Delta = \Delta_1, a \mapsto_h \text{memo } w$. Apply the (inherit) rule to obtain $\Delta \sqsubseteq_{\text{save } m, a, m, a} \Delta'$. By [force], we have $\Gamma_1 : Fv \Downarrow_n^{\{h\}} \Delta_1 : w$. By Lemma 46, we have $\Gamma_1 : Fv \Downarrow_{n+m}^{\{h\}} \Delta'_1 : w$ for $\Delta_1 \sqsubseteq_{\text{save } m, a, m, a} \Delta'_1$. Apply Lemma 45 to obtain $\Delta_1, a \mapsto_h \text{memo } w \sqsubseteq_{\text{save } m, a, m, a} \Delta'_1, a \mapsto_h \text{memo } w$.

$$\begin{array}{ccc}
 \Gamma & \xrightarrow{\text{(save)}} & \Gamma' \\
 \text{[force]} \downarrow & & \downarrow \text{[force]} \\
 \Delta & \xrightarrow{\text{(inherit)}} & \Delta'
 \end{array}$$

- **(force), [save]**: Let force a and save $m b$. If $a \neq b$, we proceed as in the previous proof. If $a = b$, then we have $\Gamma = \Gamma_1, a \mapsto_n \text{lazy}_F v$ and $\Gamma' = \Gamma'_1, a \mapsto_h \text{memo } w$ where $\Gamma_1 : Fv \Downarrow_n^{\{h\}} \Gamma'_1 : w$. Thus also $\Gamma_1 : Fv \Downarrow_{n+m}^{\{h\}} \Gamma''_1 : w$ and $\Gamma_1, a \mapsto_{n+m} \text{lazy}_F v \Downarrow_1 \Gamma''_1, a \mapsto_h \text{memo } w$ by the (force) rule. By Lemma 46, we have $\Gamma'_1 \sqsubseteq_{\text{save } m, a, m, a} \Gamma''_1$. By Lemma 45, we get $\Gamma'_1, a \mapsto_h \text{memo } w \sqsubseteq_{\text{save } m, a, m, a} \Gamma''_1, a \mapsto_h \text{memo } w$.

$$\begin{array}{ccc}
 \Gamma & \xrightarrow{\text{(force)}} & \Gamma' \\
 \text{[save]} \downarrow & & \downarrow \text{[save]} \\
 \Delta & \xrightarrow{\text{(force)}} & \Delta'
 \end{array}$$

- **(force), [force]**: As in the proof of Lemma 9 since the $\Downarrow^{\{h\}}$ relation is deterministic for fixed step count as shown by Lemma 46.

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

Lemma 21

Proof. We extend the proof of Lemma 9 for the new rules. ◀

23:30 Persistent Amortized Analysis, Operationally

- **(lazy)**: By the induction hypothesis on the premise, we have $\Delta \sqsubseteq_{Fv, k'', k', w} \Delta'$ for $k'' \leq k$ and $\Gamma' \sqsubseteq \Delta'$. We then get $\Delta \sqsubseteq_{\text{lazy}_F v, 0, k', a} \Delta', a \mapsto_{k''} \text{memo}_{Fv}^{\Delta \setminus \Gamma} w$ by the **(lazy)** rule. We have $\Gamma', a \mapsto_{k''} \text{memo}_{Fv}^{\Delta \setminus \Gamma} w \sqsubseteq \Delta', a \mapsto_{k''} \text{memo}_{Fv}^{\Delta \setminus \Gamma} w$ by the **[trans]** of $\Gamma' \sqsubseteq \Delta'$ and **[save]**.

$$\begin{array}{ccc} \Gamma & \xrightarrow{\text{(lazy)}} & \Gamma' \\ \sqsubseteq \downarrow & & \downarrow \sqsubseteq \text{[trans] [save]} \\ \Delta & \xrightarrow{\text{(lazy)}} & \Delta' \end{array}$$

- other cases: By induction over $\Gamma \sqsubseteq \Delta$.
 - $_$, **[lazy]**: By the induction hypothesis we have $\Delta \sqsubseteq_{e, l, k', v} \Delta'$ for some $l \leq k$. Since $a \notin \text{dom}(\Delta')$, this implies by Lemma 45 that $\Delta, a \mapsto_{k''} \text{memo}_{Fv}^{\Delta \setminus \Gamma} w \sqsubseteq_{e, l, k', v} \Delta', a \mapsto_{k''} \text{memo}_{Fv}^{\Delta \setminus \Gamma} w$. We get $\Gamma' \sqsubseteq \Delta', a \mapsto_{k''} \text{memo}_{Fv}^{\Delta \setminus \Gamma} w$ by **[lazy]**.

$$\begin{array}{ccc} \Gamma & \xrightarrow{\sqsubseteq_{e, k, k', v}} & \Gamma' \\ \text{[lazy]} \downarrow & & \downarrow \text{[lazy]} \\ \Delta & \xrightarrow{\sqsubseteq_{e, l, k', v}} & \Delta' \end{array}$$

- **((unfold), [save]), ((recall), [save]), ((waste), [save]), ((waste), [force]), ((unfold), [force]), ((recall), [force]))**: In all of these rules, the addresses they act on are different. This allows us to permute the rules as in the previous proofs.
- **(save), [save]**: Let $\text{save } m a$ and $\text{save } m' b$. If $a \neq b$, we proceed as in the previous cases. If $a = b$, then apply the **[save]** rule to obtain $\Delta, a \mapsto_{n-m'} \text{lazy}_F v \sqsubseteq_{\text{save } m a, 0, m, a} \Delta, a \mapsto_{n-m'-m} \text{lazy}_F v$.

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

- **(force), [save]**: Let $\text{force } a$ and $\text{save } m b$. If $a \neq b$, we proceed as in the previous cases. If $a = b$, then apply the **(force)** rule to obtain $\Gamma, a \mapsto_{n-m} \text{memo}_{Fv}^{\Delta} w \sqsubseteq_{\text{force } a, 0, 0, a} \Delta, a \mapsto \text{memo } w$ since $n \leq 0$ implies $n - m \leq 0$.

$$\begin{array}{ccc} \Gamma & \xrightarrow{\text{(force)}} & \Gamma' \\ \text{[save]} \downarrow & & \downarrow \text{[refl]} \\ \Delta & \xrightarrow{\text{(force)}} & \Delta' \end{array}$$

- **(save), [force]**: Let $\text{save } m a$ and $\text{force } b$. If $a \neq b$, we proceed as in the previous cases. If $a = b$, then apply the **(waste)** rule to obtain $\Delta, a \mapsto \text{memo } w \sqsubseteq_{\text{save } m a, 0, m, a} \Delta, a \mapsto \text{memo } w$.

$$\begin{array}{ccc} \Gamma & \xrightarrow{\text{(save)}} & \Gamma' \\ \text{[force]} \downarrow & & \downarrow \text{[force]} \\ \Delta & \xrightarrow{\text{(waste)}} & \Delta' \end{array}$$

- **(force), [force]**: Let $\text{force } a$ and $\text{force } b$. If $a \neq b$, we proceed as in the previous cases. If $a = b$, then apply the **(recall)** rule to obtain $\Delta, a \mapsto \text{memo } w \sqsubseteq_{\text{force } a, 0, 0, a} \Delta, a \mapsto \text{memo } w$. We have $\Delta, a \mapsto \text{memo } w \sqsubseteq \Delta, a \mapsto \text{memo } w$ by **[refl]**.

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

◀

Lemma 23

Proof. We extend the proof of Lemma 21 for the new (lazy) rule.

- **(lazy):** By the induction hypothesis on the premise, we have $\Delta \sqsubseteq_{Fv, k'', k', w} \Delta'$ for $k'' \leq k$ and $\Gamma' \sqsubseteq \Delta'$. We then get $\Delta \sqsubseteq_{\text{lazy}_F v, k'', k', a} \Delta', a \mapsto_0 \text{memo}_{Fv}^{\Delta \setminus \Gamma} w$ by the (lazy) rule. We have $\Gamma', a \mapsto_0 \text{memo}_{Fv}^{\Delta \setminus \Gamma} w \sqsubseteq \Delta', a \mapsto_0 \text{memo}_{Fv}^{\Delta \setminus \Gamma} w$ by $\Gamma' \sqsubseteq \Delta'$ and Lemma 45.

$$\begin{array}{ccc}
 \Gamma & \xrightarrow{\text{(lazy)}} & \Gamma' \\
 \sqsubseteq \downarrow & & \downarrow \sqsubseteq \\
 \Delta & \xrightarrow{\text{(lazy)}} & \Delta'
 \end{array}$$

- **_, [lazy]:** By the induction hypothesis we have $\Delta \sqsubseteq_{e, l, k', v} \Delta'$ for some $l \leq k$. Since $a \notin \text{dom}(\Delta')$, this implies by Lemma 45 that $\Delta, a \mapsto_0 \text{memo}_{Fv}^{\Delta \setminus \Gamma} w \sqsubseteq_{e, l, k', v} \Delta', a \mapsto_0 \text{memo}_{Fv}^{\Delta \setminus \Gamma} w$. We get $\Gamma' \sqsubseteq \Delta', a \mapsto_0 \text{memo}_{Fv}^{\Delta \setminus \Gamma} w$ by [lazy].

$$\begin{array}{ccc}
 \Gamma & \xrightarrow{\sqsubseteq_{e, k, k', v}} & \Gamma' \\
 \text{[lazy]} \downarrow & & \downarrow \text{[lazy]} \\
 \Delta & \xrightarrow{\sqsubseteq_{e, l, k', v}} & \Delta'
 \end{array}$$

◀