Thunks and Debits in Separation Logic with Time Credits

FRANÇOIS POTTIER, Inria, France
ARMAËL GUÉNEAU, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, France
JACQUES-HENRI JOURDAN, Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, France
GLEN MÉVEL, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, France

A thunk is a mutable data structure that offers a simple memoization service: it stores either a suspended computation or the result of this computation. Okasaki [1999] presents many data structures that exploit thunks to achieve good amortized time complexity. He analyzes their complexity by associating a debit with every thunk. A debit can be paid off in several increments; a thunk whose debit has been fully paid off can be forced. Quite strikingly, a debit is associated also with future thunks, which do not yet exist in memory. Some of the debit of a faraway future thunk can be transferred to a nearer future thunk. We present a complete machine-checked reconstruction of Okasaki's reasoning rules in Iris$^{\$}$, a rich separation logic with time credits. We demonstrate the applicability of the rules by verifying a few operations on streams as well as several of Okasaki's data structures, namely the physicist's queue, implicit queues, and the banker's queue.

CCS Concepts: • **Theory of computation** → **Separation logic**; **Program verification**.

Additional Key Words and Phrases: program verification, separation logic, time complexity

ACM Reference Format:
François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. 2024. Thunks and Debits in Separation Logic with Time Credits. *Proc. ACM Program. Lang.* 8, POPL, Article 50 (January 2024), 27 pages. https://doi.org/10.1145/3632892

1 INTRODUCTION

In a famous book, Okasaki [1999] presents several data structures for purely functional programs. These data structures are *persistent* [Driscoll et al. 1989], that is, apparently immutable. Instead of mutating its argument, an update operation returns a new data structure, leaving the observable content of the original data structure intact.

In order to achieve good time complexity, Okasaki uses thunks, also known as suspensions. A *thunk* is a mutable data structure that offers a simple memoization service: it stores either a suspended computation or the result of this computation. The use of thunks does not affect the functional behavior of a program, but allows delaying a computation until the moment where its result is definitely needed, while still sharing this result. Thus, the use of thunks can improve the time complexity of a program, without impacting its final result. For this reason, thunks are accepted as an essential part of the "functional programming" toolbox [Hughes 1989]. In fact, in the functional programming language Haskell, creating and forcing thunks are implicit operations.

Thunks are a basic building block in the construction of *streams*, whose definition appears in Figure 1. A stream is a lazy list: each stream cell is wrapped in a thunk, so it is evaluated only

Authors' addresses: François Pottier, francois.pottier@inria.fr, Inria, Paris, France; Armaël Guéneau, armael.gueneau@inria.fr, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190, Gif-sur-Yvette, France; Jacques-Henri Jourdan, jacques-henri.jourdan@cnrs.fr, Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, 91190, Gif-sur-Yvette, France; Glen Mével, glen.mevel@crans.org, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190, Gif-sur-Yvette, France.

ACM 2475-1421/2024/1-ART50
https://doi.org/10.1145/3632892

```
 1 type 'a stream = ('a cell) thunk
 2  and 'a cell   =  Nil | Cons of 'a * 'a stream
 3
 4 let nil () : 'a stream =
 5   Thunk.create @@ fun () -> Nil
 6
 7 let uncons (s : 'a stream) : 'a * 'a stream =
 8   match Thunk.force s with
 9   | Nil          -> assert false (* dead branch *)
10   | Cons (x, s) -> x, s
11
12 let rec append (s1 : 'a stream) (s2 : 'a stream) : 'a stream =
13   Thunk.create @@ fun () -> match Thunk.force s1 with
14   | Nil           -> Thunk.force s2
15   | Cons (x, s1) -> Cons (x, append s1 s2)
16
17 let rec revl_append (l : 'a list) (c : 'a cell) : 'a cell =
18   match l with
19   | []     -> c
20   | x :: l -> revl_append l (Cons (x, Thunk.create @@ fun () -> c))
21
22 let revl (l : 'a list) : 'a stream =
23   Thunk.create @@ fun () -> revl_append l Nil
```

Fig. 1. Streams: OCaml Code

on demand. Figure 1 presents several functions on streams. The functions `Thunk.create` and `Thunk.force` are used to create and force thunks. The function `append` (line 12) lazily appends two streams: a thunk in the stream `s1` and `s2` is forced only when needed, that is, only when the corresponding thunk in the result stream is forced. `revl` (line 22) reverses a list, returning a stream. It is also lazy, insofar as possible: the expensive list reversal operation (performed by the auxiliary function `revl_append`) is carried out only when the first thunk of the result stream is forced.

Okasaki's complexity bounds are *amortized* [Tarjan 1985]: that is, the actual cost of an operation may be higher than its advertised cost. Still, the advertised complexity bounds are sound, and can be used to compute a safe bound on the global execution time of a program. The basic idea of amortization [Tarjan 1985] is simple: if the advertised cost of some operations is slightly higher than their actual cost, then a sequence of operations accumulates a certain amount of *credit*, which can be used to justify an occasional "expensive" operation, whose actual cost exceeds its advertised cost. One must stress, however, that Okasaki does not reason in terms of credit, and cannot do so, because he needs thunks to be shareable. Sharing credit leads to an unsound analysis, as credit can then be spent twice. Instead, Okasaki reasons in terms of *debit*. With every thunk, he associates a debit, a nonnegative integer number, which indicates how much one must pay *before* this thunk can be forced. In other words, Okasaki ensures that the actual cost of forcing a thunk is always *paid for in advance*. Once its debt has been paid off, a thunk can be forced as many times as one wishes, essentially "for free", that is, at cost $O(1)$. Even if thunks are shared, this analysis is sound: indeed, duplicating a debt leads to an over-approximation of a program's time complexity.

In summary, Okasaki's thunks with debits support three main operations: creation, forcing, and *paying*. When a new thunk is created, its debit is the cost of the suspended computation. If the debit associated with this thunk becomes zero, then this thunk can be forced. Between these two points in time, this thunk's debit can be reduced by paying, in one or more increments. Paying is a ghost operation: it has no runtime effect, as there is no runtime accounting of debits. Paying decreases the debit of a thunk from n down to $n - k$ and is itself regarded as an operation that costs k units. Danielsson [2008] must be credited for this crisp explanation of Okasaki's analysis technique.

Because every cell in a stream involves a thunk, it is natural to represent the "cost" of a stream as a sequence of debits, recording the cost of forcing each stream cell, or, more accurately, the remaining debt associated with each stream cell. The "cost" of each operation on streams can then be expressed in terms of sequences of debits. For example, the debit sequence of the stream returned by `append` is roughly the concatenation of the debit sequences of the two argument streams, plus an extra debit of $O(1)$ that is added to every thunk in the first segment of the new debit sequence, so as to account for the cost of concatenation. (Details appear in §6.3 and Figure 16.) As another example, in the stream returned by `revl`, the first thunk has debit $O(n)$, where n is the length of the list `l`, while the following thunks have debit zero: indeed, the expensive list reversal operation is performed when the first thunk is forced, and the remaining thunks require no computation.

Challenges. In this paper, we wish to give a mechanized and foundational account of Okasaki's debit-based reasoning discipline. We want our reasoning rules to be expressive enough to verify the functional correctness and time complexity of Okasaki's functional data structures and of (possibly imperative) client programs that use them. This requires addressing several challenges:

(1) We need a program logic that can express and verify worst-case bounds on execution time. This logic must allow debit-based reasoning about thunks, but must not be restricted to this style: code that does not involve thunks typically requires credit-based reasoning.
(2) This logic must allow reasoning not only about purely functional data structures, but also about imperative data structures and algorithms.
(3) As demonstrated by `append`, it is common to construct a thunk which (when forced) forces another thunk. Unfortunately, in the presence of mutable state, one can also construct a *reentrant* thunk, that is, a thunk that attempts to force itself, resulting (at best) in a graceful runtime error or (at worst) in undesirable behavior, such as divergence. We want the logic to statically forbid reentrancy, thereby offering a strong guarantee: *a verified program cannot fail at runtime and must terminate within the advertised time bound.*
(4) Although we have written that it is "natural" to associate a sequence of debits with a stream, this style of reasoning is in fact highly nontrivial, as it involves keeping track of and updating the debt of *future* thunks. That is, a thunk's debit conceptually exists and can be updated *before* this thunk actually exists in the machine's memory! This is known as a *deep payment.* As a key example, we show that streams enjoy an intuitively simple *debit forwarding* rule, which allows *shifting debits towards the left* in a sequence of debits. To illustrate this rule, consider `append f (revl r)`, where `f` is a stream of length n and `r` is a list of length n. Suppose that every thunk in the stream `f` has debit $O(1)$. According to the specifications of `revl` and `append` that we have sketched, this expression produces a stream whose debit sequence is $O(1)$ everywhere, except for the central thunk, whose debit is $O(n)$. Debit forwarding allows us to change our view of this stream by distributing the high debt of the central thunk onto the preceding thunks. The debit of every thunk in the first half of the sequence is increased by $O(1)$, therefore remains $O(1)$, while the debit of the central thunk is reduced to $O(1)$. This nontrivial reasoning step is carried out *before* the stream constructed by `append f (revl r)` is forced. It plays a crucial role in the verification of the banker's queue (§7).

State of the Art. Several papers in the literature address some, but not all, of these challenges. Danielsson [2008] proposes a type system, embedded in Agda, featuring an abstract type *Thunk n a* and operations on this type. The parameter *n* is the debit associated with this thunk; the parameter *a* is the type of its result. This pioneering work exhibits several major limitations: it is limited to purely functional programs and debit-based reasoning; it does not seem to prevent reentrancy or nontermination (§9); and the proof of type soundness is not formally connected with the API that a user of the library relies upon. Furthermore, deep payment appears in the API but is not covered by the soundness proof. Thus, Danielsson addresses Challenge 4, but not Challenges 1 to 3.

Several authors [Atkey 2011; Hoffmann et al. 2013; Charguéraud and Pottier 2017; Haslbeck and Nipkow 2018; Zhan and Haslbeck 2018; Mével et al. 2019; Haslbeck and Lammich 2021] use *separation logic with time credits* to verify worst-case amortized time complexity bounds for possibly imperative programs. The basic idea is simple: one arranges for every step of computation to consume one *time credit*, where a time credit is a ghost resource. Because credits cannot be forged or duplicated, the amount of credit that is initially made available (in the precondition of a closed program) bounds the total execution time of this program. Among these authors, only Mével et al. [2019] formalize thunks and debits; however, their reasoning rules support neither thunks that force thunks nor debit forwarding. Thus, they address Challenges 1 and 2, but not Challenges 3 or 4. Their work is discussed in greater detail at the end of the paper (§9).

Contributions. In this paper, we address challenges 1–4 and carry out a foundational verification of the time complexity of three of Okasaki's notorious algorithms [1999]. To address Challenges 1 and 2, we use Iris$^{\$}$ [Mével et al. 2019], a Separation Logic with time credits that is implemented inside the Coq proof assistant on top of the Iris framework [Jung et al. 2018]. Inside this existing logic, we verify a "thunk" library, equipped with a rich "debit" API, which supports nested thunks and deep payment, addressing Challenges 3 and 4. Using this library, we verify several of Okasaki's data structures, namely the banker's queue [Okasaki 1999, §6.3.2] (§7), the physicist's queue [Okasaki 1999, §6.4.2], and implicit queues [Okasaki 1999, §11.1] (§8). In each case, we verify functional correctness, termination, and worst-case amortized time complexity. Thanks to the modularity of Iris$^{\$}$, our verified data structures can be used as part of verified programs that involve time-credit-based reasoning, mutable state, or shared-memory concurrency.

On Thread Safety. Although our thunks can be used in a concurrent program, they are not thread-safe: two threads must never race to force a thunk. Our thunk API enforces this restriction by using affine tokens that represent a permission to force a thunk. Nevertheless, we are able to verify Okasaki's data structures, because they are sequential data structures. Our verified APIs use affine tokens, where needed, to enforce sequential access. For instance, the reasoning rule Banker-Extract in Figure 18 uses the token $\lightning^{\infty}$ to forbid concurrent calls to *extract*.

Haskell's thunks, in comparison, are thread-safe: it is permitted for two threads to concurrently attempt to force a thunk. This implies that one must be prepared to accept duplication of work (where the suspended computation is performed twice) or blocking (where one thread waits for another thread to complete the computation and update the thunk). Haskell's implementation involves a combination of duplication and blocking [Marlow et al. 2009]. However, our time complexity analysis allows neither duplication nor blocking: we do not know how to bound the extra time that would be spent performing duplicated work or waiting for a thunk to be updated. For this reason, in this paper, we stick with a simple view of thunks as a sequential data structure.

Hoffmann et al. [2013] propose reasoning about lock-free data structures using time credits. Their key insight is that a thread that is able to make progress (typically via a successful CAS) must pay for the cost of CAS failures and retries in other threads. It is not clear whether this idea could be exploited to reason about the worst-case time complexity of concurrent thunks.

Road Map. Our thunk library is organized in three layers of abstraction. After recalling some of the concepts of Iris (§2), we present the bottom layer, a novel ghost data structure, the *piggy bank* (§3). None of the operations on piggy banks has a runtime effect. The main three operations on piggy banks, namely creating, paying, and breaking (forcing) a bank, are ghost updates. In the second layer (§4), we implement thunks and establish reasoning rules that match Okasaki's informal rules. These rules include Thunk-Consequence (Figure 5), which, together with Thunk-Pay (Figure 6), justifies deep payment. Our construction of thunks relies on piggy banks in two distinct places and can associate many piggy banks with a single thunk. In the third and last layer (§5), we equip thunks with a notion of *height* that simplifies the way in which we rule out reentrancy while still allowing thunks to force thunks.

On top of thunks, we implement streams (§6). We establish a number of reasoning rules about streams, including the debit forwarding rule Stream-Forward-Debit (Figure 13), which distributes a debit over several thunks in a stream, by moving parts of this debit from the right toward the left, that is, from thunks that are more distant in the future towards thunks that are closer in the future.

After presenting our proofs of three of Okasaki's data structures (§7, §8), we review the related work (§9) and conclude (§10). All of our results are machine-checked [Pottier et al. 2023]. For readability, in the paper, we present code in OCaml syntax. The code that we actually verify is expressed in HeapLang, an untyped call-by-value λ-calculus with dynamically allocated mutable state, whose definition is bundled with Iris.

2 A REFRESHER ON IRIS AND IRIS$^\$$

Even a basic introduction to Iris [Jung et al. 2018] could occupy a whole paper. We recall some of the key concepts, intuition, and notation of Iris, and we hope that a reader who is not an expert in Iris can grasp the intuition behind the abstractions that we build. For example, a reader who looks at the reasoning rule Thunk-Pay (Figure 6) should at least understand that it is a ghost update ($\Rrightarrow$) that consumes k time credits ($\$k$) and decreases the debit of a thunk from n down to $n - k$.

Assertions. Separation logic uses *assertions* to describe certain *knowledge* about the world and to encode *permissions* to change the world in certain ways. By "the world", we mean both the physical state of the machine and the ghost state that has been allocated as part of the proof. Some assertions are *pure*, that is, independent of the world. For example, the assertion $\ulcorner x = 0 \urcorner$ asserts that the equation $x = 0$, which involves the mathematical variable x, holds. Pure assertions are a special case of *persistent* assertions. Although persistent assertions may depend on the world, once they hold, they hold forever. For instance, the assertion $\mathit{Thunk}\ \mathcal{F}\ t\ n\ R\ \phi$, which asserts (among other things) that t is the address of a valid thunk in memory, is persistent. This reflects the fact that a thunk cannot be destroyed.[1] A persistent assertion is duplicable: if P is persistent, then P entails the conjunction $P * P$. The fact that *Thunk* is persistent reflects that it is safe to share a thunk. Finally, an assertion that is neither pure nor persistent is *affine*. An affine assertion typically represents a combination of knowledge and permission. For instance, the *points-to* assertion $t \mapsto v$ represents both the exclusive knowledge that the memory location t currently contains the value v and an exclusive permission to write a new value at this location.

The natural notions of conjunction and implication are the separating conjunction $*$ and the magic wand $-\!\!*$. (The non-separating conjunction $\wedge$ and implication $\Rightarrow$ are not used in this paper.) A magic wand $P -\!\!* Q$ can be read as an implication; however, one must keep in mind that (unless P is persistent) applying this magic wand *consumes* P. A magic wand itself is *not* persistent, so it can be applied only once. It can be made persistent by using the *persistence modality*: $\square(P -\!\!* Q)$ is a magic wand that can be used as many times as one wishes.

[1]HeapLang does not have explicit memory deallocation. We assume that a garbage collector reclaims unreachable objects.

Ghost State. Like physical state, *ghost state* is dynamically allocated. The law True $\Rrightarrow \exists\gamma.\boxed{\gamma \mapsto m}$ (provided by Iris) allocates a fresh ghost cell, at address γ, whose initial content is m. We write γ, δ, π for ghost addresses. A *ghost update* assertion $P \Rrightarrow Q$ means that, by consuming P and by updating the ghost state, it is possible to reach a state where Q holds. A ghost update is applied as part of a proof; such an application is not visible in the code. The content m of a ghost cell γ is an element of a *camera* M that is implicitly associated with γ and that is chosen when this ghost cell is allocated. For our purposes, a camera is a commutative monoid $(M, \cdot)$ equipped with a notion of *validity*, such that *valid* $(m_1 \cdot m_2)$ implies *valid* $m_1 \wedge$ *valid* m_2. By design, the logic guarantees that the content of a ghost cell is always a valid element. This is expressed by the law $\boxed{\gamma \mapsto m} \vdash \ulcorner \mathit{valid}\ m \urcorner$. The *ghost points-to* assertion $\boxed{\gamma \mapsto m}$ means that m is *one fragment* of the content of the ghost cell γ, and represents the ownership of just this fragment. This is reflected by the composition law $\boxed{\gamma \mapsto m_1 \cdot m_2} \equiv \boxed{\gamma \mapsto m_1} * \boxed{\gamma \mapsto m_2}$, which allows ghost points-to assertions to be split and joined, and by the *frame-preserving update* law, which is stated as follows: if for every m' *valid* $(m_1 \cdot m')$ implies *valid* $(m_2 \cdot m')$ then $\boxed{\gamma \mapsto m_1} \Rrightarrow \boxed{\gamma \mapsto m_2}$.

A *meta witness* $t \rightsquigarrow \gamma$, a persistent assertion, indicates that the ghost address γ has been associated with the physical memory location t. The law $t \rightsquigarrow \gamma_1 * t \rightsquigarrow \gamma_2 \vdash \ulcorner \gamma_1 = \gamma_2 \urcorner$ (provided by Iris) states that this association is unique: it forms a (partial) map of physical locations to ghost addresses.

Invariants. Roughly speaking, an *invariant* is an assertion which, by convention and from a certain point on, must hold "at all times". The assertion $\boxed{I}$ indicates that the assertion I has been made an invariant. The law $I \Rrightarrow \boxed{I}$ dynamically establishes a new invariant. Even if I is not persistent, $\boxed{I}$ is persistent: the knowledge that an invariant exists can be shared. This knowledge allows *accessing* the invariant, that is, *opening* and *closing* it. Opening an invariant produces the assertion I, allowing the user to exploit I and possibly to destroy it, thereby temporarily violating the invariant. Closing an invariant requires the user to provide I and consumes it, thereby re-establishing the invariant.

Thus, the claim that an invariant holds "at all times" is a white lie. An invariant holds at all times *except while it is being accessed.* Therefore, the logic must forbid reentrant access to an invariant, that is, forbid opening an invariant that is already open. To this end, the use of invariants is subject to two constraints.

First, invariant accesses are *atomic*: opening and closing an invariant must take place during a single physical computation step. This guarantees that, under an interleaving semantics, two distinct threads cannot access an invariant at the same time.

Second, *invariants* $\boxed{I}^N$ are in fact labeled with a *namespace* N. This annotation is used to forbid reentrant access within a thread: two invariants can be simultaneously opened only if they are labeled with disjoint namespaces.[2] Enforcing this policy requires keeping track, at all times, of which invariants can currently be accessed. We omit the details, but note that the *ghost update* connective $\Rrightarrow_{\mathcal{E}}$ must be indexed with a mask $\mathcal{E}$. In short, $P \Rrightarrow_{\mathcal{E}} Q$ means that P can be transformed into Q while accessing only those invariants whose namespace N is in the set $\mathcal{E}$. One may omit this mask when it is $\top$.

The *later* modality $\triangleright$ weakens an assertion. Roughly, the assertion $\triangleright P$ means that the assertion P will hold in the next time step, that is, after the next atomic instruction is executed. This modality appears in the reasoning rules for invariants, where it serves to forbid certain logical paradoxes. Our claim that opening an invariant produces I and closing it consumes I was another white lie: these operations actually produce and consume $\triangleright I$. This is visible in some of our reasoning rules, such as PiggyBank-Break (Figure 2), but can otherwise be ignored.

[2] A *namespace* N, T is a string. A *mask* $\mathcal{E}, \mathcal{F}$ is a set of such strings. If T is a namespace, then its *upward closure* $\uparrow T$, a mask, is the set of all strings that admit T as a prefix. The *full mask* $\top$ is the set of all strings. We write $\mathcal{E}_1 \mathbin{\#} \mathcal{E}_2$ when the masks $\mathcal{E}_1$ and $\mathcal{E}_2$ are disjoint. We say that two namespaces T_1 and T_2 are disjoint when $\uparrow T_1 \mathbin{\#} \uparrow T_2$ holds.

Hoare Triples. A specification traditionally takes the form $\{P\}\ e\ \{\phi\}$, where the precondition P is an assertion about the initial state, the expression e is the program fragment of interest, and the postcondition ϕ describes the result value and the final state: if v is the result value then $\phi\ v$ is an assertion about the final state. We typically write $\{P\}\ e\ \{\lambda v. Q\}$ in order to bind the result value in the postcondition. This can be read as follows: "provided the initial state satisfies P, executing e cannot crash, and if it terminates, then it returns some value v such that the final state satisfies Q". A triple is persistent: it allows executing the expression e as many times as one wishes. We occasionally need a *one-shot triple*, written **once** $\{P\}\ e\ \{\phi\}$. Its meaning is the same as that of a persistent triple, except that it allows e to be executed at most once. It is an affine assertion. An ordinary triple is a one-shot triple wrapped in the persistence modality $\square$.

Time Credits. Iris$^\$$ [Mével et al. 2019] extends Iris with *time credits*. The assertion $\$n$ represents n time credits. It is affine: time credits can be discarded but not duplicated. The reasoning rules of the logic ensure that every instruction consumes one time credit. As a result, if the triple $\{\$n\}\ e\ \{\phi\}$ holds, where e is a *closed* expression (a complete program), then e does not crash and *must terminate in at most n steps.* In other words, the logic offers a worst-case time complexity guarantee.

In general, a specification provides a worst-case *amortized* time complexity guarantee. For instance, Thunk-Force (Figure 6) does *not* guarantee that *force t* runs in at most $\underline{F}$ steps. Although only $\underline{F}$ time credits are ostensibly visible in the precondition, the assertion $\mathit{Thunk}\ \mathcal{F}\ t\ 0\ R\ \phi$, which is also part of the precondition, offers access to an invariant which possibly contains more credits. Thus, this specification means that the *amortized* time complexity of *force* is $\underline{F}$.

Abstract literal constants, such as $\underline{N}$ and $\underline{F}$, hide concrete numeric constants. For instance, $\underline{N}$, the cost of creating a new thunk, is 3, while $\underline{F}$, the cost of forcing a thunk, is 11. In our Coq proofs, these literal constants are defined at the top level and are made opaque as soon as possible. In the paper, their definitions are omitted. Every literal constant is denoted by an underlined capital letter.

3 PIGGY BANKS

Once upon a time, in a faraway university, a class of students wanted to throw a big party. Alas, food, drinks, disguises, and other equipment were then and there very expensive. So, the students' first action was to install in the classrooms, in the students' lounge, in the dormitories, and in several other locations, a number of porcelain piggy banks. The students declared that everyone could contribute whatever amount he or she desired, in whatever location and at whatever time he or she desired. They agreed that, once the total accumulated amount reached a hundred sovereigns, they would break all piggy banks and throw a big party.

Alas, because one could not see through a piggy bank, one could not tell how much money was inside it. And because piggy banks were installed in many places, there was no coordination between contributors. No student could be reliably informed of all contributions, and there was no way of maintaining a registry of all contributions. Faced with these difficulties, the students adopted a habit of telling each other how much money they thought remained to be collected. One morning, in the students' lounge, Charles was told by Brian, "97 to go". There, Charles put one sovereign into the piggy bank. As he exited the room, he ran into Sophie and Sara, whom he told, "96 to go". Later on during that day, at different times and in different places, Sophie and Sara each contributed one sovereign. In the evening, each of them independently told Brian, "95 to go".

When the students finally determined that the piggy banks could safely be broken, they found that they had accumulated much more than a hundred sovereigns. It was a big party.

This fable is meant to suggest that, independently of physical mechanisms such as a distributed piggy bank in a university or a thunk in a computer's memory, there is a sound and useful pattern of reasoning about credit that is accumulated via uncoordinated payments. This pattern involves *true debit*, or "how much really is still missing", *apparent debit*, or "how much Sophie thinks is still missing", and the property that an apparent debit is always an over-approximation of the true debit.

To isolate this pattern and to establish its logical soundness, we develop the *piggy bank*, a ghost data structure, and show that it supports the desired reasoning rules. There is no code: a piggy bank is a purely logical concept. Our construction of thunks (§4.2.1, §4.2.2) uses piggy banks in two distinct places and can cause an unbounded number of piggy banks to be associated with a single thunk. This suggests that it is worthwhile to make piggy banks a stand-alone abstraction.

3.1 Piggy Banks: Interface

A piggy bank can be abstractly described as a ghost data structure that is in one of two states: "pending" or "forced". Furthermore, the transition from the pending state to the forced state is not instantaneous, and may require executing some user code: meanwhile, the piggy bank is in a third state, the "transient" state. Executing this user code has a certain cost: the time credits accumulated in the piggy bank are intended to pay for this.

There is no need for concrete descriptions of the pending state and of the forced state. We assume that these states are described by two parameters $P : \mathbb{N} \rightarrow \mathit{iProp}$ and $Q : \mathit{iProp}$, where *iProp* is the type of Iris assertions. The assertion $P\ nc$ means that the piggy bank is in the pending state and that *nc*, standing for "necessary credits", is the number of credits that we aim to accumulate before transitioning to the forced state. The assertion Q means that the piggy bank is in the forced state.

We want a piggy bank to be shared between several participants, so it must be described by a persistent predicate, *PiggyBank*. Participants must be allowed to pay, that is, to insert time credits into the piggy bank. This is a ghost operation. Each participant must be able to pay independently, without coordinating with other participants, so, as suggested by the fable, the *PiggyBank* assertion must keep track of an apparent debit, that is, a nonnegative number of debits, *n*. Once a participant sees an apparent debit of zero, we want this participant to be able to deduce that enough credit has been accumulated to allow the transition to take place. We want this participant to be allowed to *break* (or *force*) the bank and either perform the transition, or discover that the transition has been performed already by another participant.

Paying and breaking the bank have rather different characteristics. We wish to think of paying as an atomic update of the ghost state of the piggy bank; and we would like paying to be permitted at all times. The act of breaking the bank, on the other hand, cannot be regarded as atomic. A participant who breaks the bank and finds it in the pending state is expected to perform a transition to the forced state, that is, to update the physical state from $P\ nc$ to Q. This can require many steps of computation. For instance, forcing a thunk requires calling a user-supplied function and updating the physical state of the thunk. These considerations suggest that breaking the bank must be viewed as a sequence of two ghost updates, between which the user may execute some physical code. One update causes a transition from the pending state to the transient state; another update causes a transition from the transient state to the forced state.

Between these updates, the piggy bank is in the transient state, where neither $P\ nc$ nor Q holds. So, while the piggy bank is being forced, one must forbid any attempt to force it again. This is made possible by the use of an exclusive token $\text{\textit{\L}}^{\mathcal{F}}$. The transition that enters the transient state consumes this token, while the transition that leaves the transient state produces it again. As we will see (§4), piggy banks are used in a nested manner, with unbounded nesting depth: this calls for an infinite number of distinct tokens. Taking inspiration from Iris's invariants, we organize these tokens using *namespaces* and *masks*. (In fact, we reuse the tokens of Iris's non-atomic invariant library.) The token $\text{\textit{\L}}^{\mathcal{F}}$ denotes the ownership of all tokens in the possibly infinite set $\mathcal{F}$. The full token $\text{\textit{\L}}^{\top}$ is created once and for all at the beginning of the execution of the program.

PiggyBank-Persist

$$\mathsf{persistent}(\mathit{PiggyBank}\ P\ Q\ N\ T\ n)$$

PiggyBank-Create

$$P\ n \Rrightarrow_{\mathcal{E}} \mathit{PiggyBank}\ P\ Q\ N\ T\ n$$

PiggyBank-Increase-Debit

$$\frac{n_1 \leq n_2}{\mathit{PiggyBank}\ P\ Q\ N\ T\ n_1 \mathrel{-\!\!*} \mathit{PiggyBank}\ P\ Q\ N\ T\ n_2}$$

PiggyBank-Pay

$$\frac{\uparrow N \subseteq \mathcal{E}}{\mathit{PiggyBank}\ P\ Q\ N\ T\ n * \$k \Rrightarrow_{\mathcal{E}} \mathit{PiggyBank}\ P\ Q\ N\ T\ (n-k)}$$

PiggyBank-Break

$$\frac{\uparrow N \subseteq \mathcal{E} \qquad \uparrow T \subseteq \mathcal{F}}{\mathit{PiggyBank}\ P\ Q\ N\ T\ 0 * \lightning^{\mathcal{F}} \Rrightarrow_{\mathcal{E}} \exists nc. \begin{pmatrix} ((\triangleright P\ nc * \$nc) \vee \triangleright Q) * \\ (\triangleright Q \Rrightarrow_{\mathcal{E}} \lightning^{\mathcal{F}}) \end{pmatrix}}$$

PiggyBank-Peek

$$\frac{\uparrow N \subseteq \mathcal{E} \qquad \uparrow T \subseteq \mathcal{F}}{\begin{array}{l} \mathit{PiggyBank}\ P\ Q\ N\ T\ n\ *\ \lightning^{\mathcal{F}} \mathrel{-\!\!*} \\ (\forall nc,\ \triangleright P\ nc \mathrel{-\!\!*} \triangleright \mathsf{False}) \mathrel{-\!\!*} \\ (\triangleright Q \mathrel{-\!\!*} \triangleright \square Q') \Rrightarrow_{\mathcal{E}} \\ \mathit{PiggyBank}\ P\ Q\ N\ T\ 0\ *\ \triangleright \square Q'\ *\ \lightning^{\mathcal{F}} \end{array}}$$

Fig. 2. Piggy Banks: Reasoning Rules

A piggy bank is parameterized with two namespaces N and T. The namespace N decorates an invariant that is opened when paying and when forcing the piggy bank. The namespace T determines what token $\lightning^{\mathcal{F}}$ must be supplied when the piggy bank is forced. Together, the parameters N, T can be thought of as a "region" in which the piggy bank exists.

Our reasoning rules for piggy banks appear in Figure 2. The assertion $\mathit{PiggyBank}\ P\ Q\ N\ T\ n$ means that there exists a piggy bank whose pending and forced states are described by P and Q, whose region is N, T, and whose number of debits is n. The parameter n is the most interesting one: in the rules of Figure 2, the four other parameters are fixed.

The rule PiggyBank-Persist states that a *PiggyBank* assertion is persistent. That is, a piggy bank can be shared. PiggyBank-Create allocates a new piggy bank. It is a ghost update. The piggy bank must initially be in its pending state: the user must establish the assertion $P\ n$, which is consumed. The user chooses n, the number of credits that must be accumulated before the piggy bank can be broken. Thus, n is the initial value of the true debit and is also the initial apparent debit. PiggyBank-Increase-Debit states that *PiggyBank* is covariant in the parameter n. In other words, it is safe to increase an apparent debit. This rule is intuitively sound because it preserves the fact that an apparent debit is an over-approximation of the true debit. PiggyBank-Pay, also a ghost update, allows contributing k time credits to a piggy bank. The apparent debit decreases from n to $n - k$. (This is subtraction in the natural numbers, so $n - k \geq 0$ holds.) This rule does not require any affine token.

We now come to the most complex rule, PiggyBank-Break. This rule allows the user to break a piggy bank whose apparent debit is 0. This is intuitively permitted because, if the apparent debit is 0, then the true debit must be 0 as well. As explained earlier, this rule is expressed via two nested ghost updates. The user has the freedom to apply these updates at two distinct points in time and to execute some code between them. The outermost update initiates the process of breaking the bank. It consumes the affine token $\lightning^{\mathcal{F}}$, where $\uparrow T \subseteq \mathcal{F}$ must hold: this forbids an attempt to break this piggy bank again while it is already being broken. It produces the following situation: for some value of nc, both of the following assertions hold:

(1) either the piggy bank is in its pending state $P\ nc$, in which case nc time credits are available (because the true debit is zero), or the piggy bank is already in its forced state Q;

$$\mathit{PiggyBank}\ P\ Q\ N\ T\ n \triangleq$$
$$\exists\, i,\ \pi,\ nc.\ \boxed{\begin{array}{rl} \exists ac.\ \boxed{\pi \mapsto \bullet\, ac} \ * & \\ & \diamond_i \ *\ P\ nc\ *\ \$ac \\ \vee & ↯^{\uparrow T}\ *\ \ulcorner nc \le ac \urcorner \\ \vee & \diamond_i\ *\ Q\ *\ \ulcorner nc \le ac \urcorner \end{array}}^{N} \ *\ \boxed{\pi \mapsto \circ\,(nc - n)}$$

Fig. 3. Piggy Banks: Internal Definition

(2) whatever state the piggy bank is currently in, the user must bring it into the forced state Q so as to be able to apply the innermost ghost update, which ends the process of breaking the bank and causes the affine token $↯^{\mathcal{F}}$ to re-appear.

The variable nc is existentially quantified because the cost of moving from the pending state to the forced state is unknown: it cannot be deduced from the *PiggyBank* assertion. The fact that this variable is shared between the conjuncts $P\ nc$ and $\$nc$ guarantees that "enough" credit is available.

Whereas forcing requires an affine token (which disappears while forcing is in progress, and re-appears when the process is complete), paying does not require a token. Therefore, while a bank is being forced, forcing it again is disallowed, but paying remains permitted.

PiggyBank-Peek states that if the user is somehow able to prove that this piggy bank cannot be in its pending state (that is, $P\ nc$ implies false), then it must be in its forced state and the piggy bank's apparent debit can be set to 0. This rule is later exploited to establish the rule ThunkVal-Force.

3.2 Piggy Banks: Construction

The definition of the predicate *PiggyBank* appears in Figure 3. It may be of interest mainly to readers who are familiar with Iris; other readers may wish to skip this part.

A ghost cell, π, appears in the definition. This cell keeps track of the total amount of the payments that the piggy bank has received. This amount grows in a monotonic manner: it can never decrease. Two kinds of assertions control this cell. The affine assertion $\boxed{\pi \mapsto \bullet\, ac}$ represents the exact knowledge of the current value of the cell and the authority to increase this value. The persistent assertion $\boxed{\pi \mapsto \circ\, k}$ is a witness that the value of the cell is at least k. These assertions satisfy the agreement law $\boxed{\pi \mapsto \bullet\, ac} * \boxed{\pi \mapsto \circ\, k} \vdash \ulcorner k \le ac \urcorner$, the update law $\boxed{\pi \mapsto \bullet\, ac} \Rrightarrow \boxed{\pi \mapsto \bullet\,(ac + k)}$, and the witness creation law $\boxed{\pi \mapsto \bullet\, ac} \Rrightarrow \boxed{\pi \mapsto \circ\, ac}$.

The invariant that appears in Figure 3 holds the authoritative view $\boxed{\pi \mapsto \bullet\, ac}$, which guarantees that ac (for "available credit") is the total amount of payment received so far. Furthermore, this invariant contains a three-way disjunction: the piggy bank must be in the pending state $\diamond_i * P\ nc * \$ac$, in the transient state $↯^{\mathcal{F}} * \ulcorner nc \le ac \urcorner$, or in the forced state $\diamond_i * Q * \ulcorner nc \le ac \urcorner$.

The exclusive tokens $\diamond_i$ and $↯^{\uparrow T}$ allow maintaining some knowledge, outside of the invariant, about which of these three states the invariant must be in. More precisely, if (while the invariant is closed) one holds the token $↯^{\uparrow T}$ then one can deduce that the invariant must be in the pending state or in the forced state, but cannot be in the transient state, because the conjunction $↯^{\uparrow T} * ↯^{\uparrow T}$ implies a contradiction. Dually, if one holds the *transient token* $\diamond_i$ then one can deduce that the invariant must be in the transient state. A fresh transient token $\diamond_i$ together with its identifier i are created when a new piggy bank is created.

In the pending state, the invariant contains the assertion $P\ nc$, where nc stands for "necessary credit", as well as ac time credits. In the transient and forced states, where forcing has already

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

Fig. 4. Thunks: OCaml Code

begun, the invariant guarantees $nc \leq ac$, which means that the available credit has exceeded the necessary credit. In the forced state, the invariant contains the assertion Q.

The last part of Figure 3 is the persistent witness $\boxed{\pi \mapsto \circ\,(nc - n)}$. By opening the invariant and by exploiting the agreement law, this witness allows obtaining the inequality $nc - n \leq ac$. If the apparent debit n is zero, then one obtains $nc \leq ac$: that is, there is enough accumulated credit to cover the cost of the transition from the pending state to the forced state.

Theorem 3.1. *The predicate PiggyBank satisfies the reasoning rules of Figure 2.*

4 THUNKS

Our implementation of thunks appears in Figure 4. In the following, we first present the reasoning rules that we wish to establish about thunks (§4.1). Then, we present the nontrivial construction that lets us obtain these rules (§4.2).

4.1 Thunks: Interface

The predicate $\mathit{Thunk}\ \mathcal{F}\ t\ n\ R\ \phi$ describes a thunk at location t in memory. The mask $\mathcal{F}$ plays the same role as in the previous section (§3): in short, it determines which token $⚡^{\mathcal{F}}$ is required to force this thunk. The parameter n also plays the same role as in the previous section: it is an apparent debit associated with this thunk. The parameter R is a resource that is required and preserved by the suspended computation. The presence of this parameter allows us to describe thunks that have side effects and thunks whose execution requires a certain token. The latter category includes thunks that force other thunks, something that is commonly needed. The parameter $\phi : \mathit{Val} \longrightarrow \mathit{iProp}$ determines the postcondition of the thunk: once this thunk is forced and produces a value v, the assertion $\Box\ \phi\ v$ can be expected to hold.[3]

The full set of reasoning rules for thunks appears in Figures 5, 6, and 7. The rules are split in three groups for reasons that will be apparent in the next subsection (§4.2).

Thunk-Create (Figure 5) describes the creation of a new thunk via the function call *create f*. The behavior of f is specified by the premise $\mathit{isAction}\ f\ n\ R\ \phi$, which denotes the one-shot triple **once** $\{R * \$n\}\ f()\ \{\lambda v.\ R * \Box\ \phi\ v\}$. This assertion is a permission to call $f()$ at most once. It indicates that the call $f()$ consumes n time credits and must return a value v such that $\Box\ \phi\ v$ holds. The resource R is required, but not consumed, by this call. Under this hypothesis on f, Thunk-Create states that *create f* costs a constant amount of credits $\$\underline{N}$ and returns a thunk whose apparent debit, resource requirement, and postcondition are described by n, R, ϕ. The abstract integer constant $\underline{N}$ is part of the thunk API; its concrete value is not revealed. The mask $\mathcal{F}$ is chosen

[3]Instead of requiring the postcondition to be in the syntactic form $\Box\,\phi$, we could equivalently allow an arbitrary *persistent* postcondition. The postcondition of a thunk must be persistent because a thunk can be forced arbitrarily many times yet always returns the same value.

$$\textsc{Thunk-Create}\quad \frac{\uparrow T \subseteq \mathcal{F}}{\{\$\underline{N} \ast \mathit{isAction}\ f\ n\ R\ \phi\}\ \mathit{create}\ f\ \{\lambda t.\ \mathit{Thunk}\ \mathcal{F}\ t\ n\ R\ \phi\}}$$

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

Fig. 5. Thunks: Creation Rule and Consequence Rule

$$\textsc{Thunk-Persist}\quad \mathrm{persistent}(\mathit{Thunk}\ \mathcal{F}\ t\ n\ R\ \phi)$$

$$\textsc{Thunk-Increase-Debit}\quad \frac{n_1 \leq n_2}{\begin{array}{l}\mathit{Thunk}\ \mathcal{F}\ t\ n_1\ R\ \phi \mathrel{-\!\!\ast} \\ \mathit{Thunk}\ \mathcal{F}\ t\ n_2\ R\ \phi\end{array}}$$

$$\textsc{Thunk-Pay}\quad \frac{\uparrow \mathit{ThunkPayment} \subseteq \mathcal{E}}{\begin{array}{l}\mathit{Thunk}\ \mathcal{F}\ t\ n\ R\ \phi \ast \$k \Rrightarrow_{\mathcal{E}} \\ \mathit{Thunk}\ \mathcal{F}\ t\ (n-k)\ R\ \phi\end{array}}$$

$$\textsc{Thunk-Force}\quad \{\mathit{Thunk}\ \mathcal{F}\ t\ 0\ R\ \phi \ast \$\underline{F} \ast \lightning^{\mathcal{F}} \ast R\}\ \mathit{force}\ t\ \{\lambda v.\ \mathit{ThunkVal}\ t\ v \ast \Box\ \phi\ v \ast \lightning^{\mathcal{F}} \ast R\}$$

Fig. 6. Thunks: Common Reasoning Rules

$$\textsc{ThunkVal-Persist}\quad \mathrm{persistent}(\mathit{ThunkVal}\ t\ v) \qquad \textsc{ThunkVal-Timeless}\quad \mathrm{timeless}(\mathit{ThunkVal}\ t\ v)$$

$$\textsc{ThunkVal-Confront}\quad \mathit{ThunkVal}\ t\ v_1 \ast \mathit{ThunkVal}\ t\ v_2 \vdash \ulcorner v_1 = v_2 \urcorner$$

$$\textsc{ThunkVal-Force}\quad \{\mathit{ThunkVal}\ t\ v \ast \$\underline{F}\}\ \mathit{force}\ t\ \{\lambda v'.\ \ulcorner v' = v \urcorner\}$$

Fig. 7. Forced-Thunk Witnesses: Reasoning Rules

by the user. It determines which token $\lightning^{\mathcal{F}}$ is required when forcing this thunk. A systematic way of instantiating this parameter is discussed in the next section (§5).

The rules Thunk-Persist, Thunk-Increase-Debit, and Thunk-Pay (Figure 6) are analogous to PiggyBank-Persist, PiggyBank-Increase-Debit, and PiggyBank-Pay. They state that thunks can be shared, that it is permitted to over-approximate an apparent debit, and that an apparent debit can be reduced by paying. (In Thunk-Pay, one can see that the parameter N of piggy banks has been instantiated with a fixed namespace $\mathit{ThunkPayment}$.)

Thunk-Force allows forcing a thunk whose apparent debit is zero. This consumes $\$\underline{F}$, a constant number of time credits. In other words, the amortized time complexity of this operation is $O(1)$, because the cost of the suspended computation has been paid for in advance. The token $\lightning^{\mathcal{F}}$ and the resource R are required and preserved. The token $\lightning^{\mathcal{F}}$ is required to force the thunk itself, whereas the token R is required by the suspended computation. These two assertions are *separately* required: that is, they cannot be the same token. Indeed, as will be evident in the next subsection (§4.2), the process of breaking the thunk's piggy bank begins (therefore, the token $\lightning^{\mathcal{F}}$ disappears) before the suspended computation is executed, and ends after the suspended computation terminates.

Forcing a thunk produces a value v such that the persistent assertions $\mathit{ThunkVal}\ t\ v$ and $\Box\ \phi\ v$ hold. The assertion $\mathit{ThunkVal}\ t\ v$ is a witness that the value of the thunk t has been forever decided and that it is v. The assertion $\Box\ \phi\ v$ means that $\phi\ v$ holds now and forever and can be exploited as many times as one wishes.

The predicate *ThunkVal* satisfies the reasoning rules in Figure 7. The rule ThunkVal-Persist reflects the fact that once the association between t and v has been decided, it remains fixed forever. The rule ThunkVal-Timeless states that $\triangleright$ *ThunkVal* t v is essentially the same as *ThunkVal* t v; this is of technical interest only. The agreement law ThunkVal-Confront states that if a thunk has been forced twice in the past then the same value must have been returned twice. Finally, ThunkVal-Force allows forcing a thunk that has been forced already. This assumption is reflected by the appearance of the assertion *ThunkVal* t v in the precondition. There is no requirement that the thunk's apparent debit be zero. As in Thunk-Force, forcing consumes $\$\underline{F}$. The value that is obtained by forcing this thunk must be v, the value that was predicted by the witness *ThunkVal* t v. Unlike Thunk-Force, this rule requires neither the token $\lightning^{\mathcal{F}}$ nor the resource R. They are not necessary because no suspended computation is executed. Also unlike Thunk-Force, and perhaps surprisingly, this rule does not guarantee $\Box\ \phi\ v$. In the presence of Thunk-Consequence, this cannot be guaranteed. In short, the assumption *ThunkVal* t v guarantees that this physical thunk has been forced already; but it could be wrapped in a number of (ghost) proxy thunks (§4.2.2) that have not been forced yet.

The rule Thunk-Consequence (Figure 5) allows changing the postcondition of a thunk from $\Box\,\phi$ to $\Box\,\psi$. In the special case where n_2 is zero, this rule weakens the postcondition of a thunk, which is why we name it the *consequence rule*. In the case where n_2 is nonzero, this rule *increases* the apparent debit of the thunk from n_1 to $n_1 + n_2$. In return, the update[4] from $\Box\ \phi\ v$ to $\Box\ \psi\ v$ is allowed to consume n_2 time credits.

4.2 Thunks: Construction

It is not easy to define the predicates *Thunk* and *ThunkVal* in such a way that all of the rules of Figures 5, 6, and 7 are satisfied. A key contribution of this paper is to propose a definition of *Thunk* that validates all of the desired rules. Although it is technically possible to give a monolithic definition, we prefer to approach the problem in three stages, as follows.

(1) We define a predicate *BasicThunk* that satisfies all of the desired rules except the consequence rule. It is analogous to Mével et al.'s *isThunk*, but instead of relying directly on Iris's "non-atomic invariant" construction, it is built on top of our "piggy bank" abstraction. As a result, *BasicThunk* validates Thunk-Pay.

(2) We remark that applying the reasoning rule Thunk-Consequence to an existing thunk t seems closely related to constructing a new thunk t' via the expression *create* $(\lambda().\,\mathit{force}\ t)$. The difference is that the consequence rule is a ghost operation and does not create a new thunk at runtime. Still, this analogy suggests that *applying the consequence rule should allocate a new piggy bank.* Guided by this idea, we propose a construction that supports *one* application of the consequence rule. Assuming that we have a predicate *Thunk* that satisfies the rules of Figure 6, we construct a new predicate *ProxyThunk*, which also satisfies these rules, and we establish a version of the consequence rule that expects a *Thunk* and produces a *ProxyThunk*.

(3) We show that this construction can be iterated as many times as desired. By building on top of *BasicThunk* and *ProxyThunk*, we are able to propose a definition of *Thunk* that satisfies all of the desired rules, including Thunk-Consequence.

This three-stage construction is presented in the next three subsections (§4.2.1, §4.2.2, §4.2.3). We remark that the definition of the predicate *ThunkVal* is not problematic. We give a definition of *ThunkVal* in the first stage (§4.2.1) and keep this definition in the following stages.

[4] We write *isUpdate* $n\ R\ \phi\ \psi$ for the assertion $\forall v.\ (R * \$n * \Box\ \phi\ v) \Rrightarrow_{\top} (R * \Box\ \psi\ v)$. This update is affine; it can be used only once. It consumes n time credits, requires $\Box\ \phi\ v$, and establishes $\Box\ \psi\ v$. The resource R is required, but not consumed. The full mask $\top$ allows this update to access all atomic invariants. In particular, Thunk-Pay can be exploited.

$$
\begin{array}{l}
\mathit{BasicThunk}\ \mathcal{F}\ t\ n\ R\ \phi \triangleq \\
\quad \exists \delta, T.\ \ulcorner \uparrow T \subseteq \mathcal{F} \urcorner\ *\ t \rightsquigarrow \delta\ * \\
\quad\quad \mathit{PiggyBank} \\
\quad\quad\quad (\lambda nc.\ \exists f.\ \boxed{\delta \mapsto ?}\ *\ t \mapsto \mathit{UNEVALUATED}\ f\ *\ \mathit{isAction}\ f\ nc\ R\ \phi) \qquad \text{— pending state} \\
\quad\quad\quad (\exists v.\ \boxed{\delta \mapsto v}\ *\ t \mapsto \mathit{EVALUATED}\ v\ *\ \Box\ \phi\ v) \qquad \text{— forced state} \\
\quad\quad\quad \mathit{ThunkPayment}\ T\ n \\
\mathit{ThunkVal}\ t\ v \triangleq \\
\quad \exists \delta.\quad t \rightsquigarrow \delta\ *\ \boxed{\delta \mapsto v}
\end{array}
$$

Fig. 8. Basic Thunks and Forced-Thunk Witnesses: Definitions

4.2.1 Basic Thunks. The definition of the predicate *BasicThunk* appears in Figure 8. Although this definition may at first sight seem somewhat cryptic, it is actually fairly straightforward. It involves two main ingredients: a ghost cell δ and a piggy bank.

The ghost cell δ records whether the value of this thunk is still undecided or decided. This ghost cell inhabits the camera $\mathrm{Ex}(()) +_{\frac{\cdot}{\cdot}} \textsc{Ag}(\mathit{Val})$, also known as the "one-shot" camera [Jung et al. 2018, §2.1]. This gives rise to the following assertions and laws. The assertion $\boxed{\delta \mapsto ?}$ means that the value is not decided yet. This assertion is affine: it represents a unique permission to make a decision. The assertion $\boxed{\delta \mapsto v}$ means that the value has been decided and that this value is v. This assertion is persistent: once a value has been decided, this decision cannot be undone, so the information that the value is v remains valid forever and can be shared. These assertions satisfy the decision law $\boxed{\delta \mapsto ?} \Rrightarrow \boxed{\delta \mapsto v}$, the agreement law $\boxed{\delta \mapsto v_1} * \boxed{\delta \mapsto v_2} \vdash \ulcorner v_1 = v_2 \urcorner$, and the disagreement law $\boxed{\delta \mapsto ?} * \boxed{\delta \mapsto v} \vdash \mathsf{False}$. A meta witness $t \rightsquigarrow \delta$ records that the ghost cell δ is uniquely associated with the thunk t. This ensures that all *BasicThunk* and *ThunkVal* assertions for the thunk t refer to the same ghost cell δ.

The concept of a piggy bank has been presented already (§3). There remains to explain how the parameters P and Q, which represent the pending and forced states of the piggy bank, are instantiated in Figure 8. In the pending state, the ghost cell δ is undecided; the physical cell t contains the value *UNEVALUATED* f; and there exists a unique permission to invoke $f()$. The cost of this invocation, nc, is not known, but the piggy bank is set up so that this cost must be fully paid for before the piggy bank can be forced. The apparent debit n of the piggy bank is also the apparent debit of the thunk. In the forced state, the ghost cell δ has been set to v, for some value v; the physical memory cell t contains the value *EVALUATED* v; and the postcondition $\Box\ \phi\ v$ is satisfied.

Theorem 4.1. *The predicate BasicThunk satisfies the rule* Thunk-Create *in Figure 5, where Thunk is replaced with BasicThunk. Furthermore, it satisfies all of the rules in Figure 6, where the same replacement is made. Finally, the predicate ThunkVal satisfies the rules in Figure 7.*

4.2.2 Proxy Thunks. Alas, basic thunks do not satisfy the consequence rule. The problem can be traced back to the piggy bank invariants, which fix the postcondition ϕ and the number of necessary credits *nc*. This forbids installing a new postcondition and a new number of necessary credits.

Fortunately, there is a simple way of working around this problem. The idea is to *allocate a new piggy bank* when the consequence rule is applied to an existing thunk t. If the existing thunk has an apparent debit of n_1 and if the update from ϕ to ψ has a cost of n_2, then the number of time credits that the new piggy bank aims to collect is set to $n_1 + n_2$. Thus, the apparent debit of the new piggy bank is $n_1 + n_2$. Once the new piggy bank has reached its aim, breaking it produces $n_1 + n_2$

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

Proxy-Create

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

Fig. 9. Proxy Thunks: Definition and Creation Rule

$$
\begin{aligned}
\mathit{RecThunk}\ 0\ \mathcal{F}\ t\ n\ R\ \phi &\triangleq \mathit{BasicThunk}\ \mathcal{F}\ t\ n\ R\ \phi \\
\mathit{RecThunk}\ (d+1)\ \mathcal{F}\ t\ n\ R\ \phi &\triangleq \mathit{ProxyThunk}\ [\mathit{Thunk} := \mathit{RecThunk}\ d]\ \ \mathcal{F}\ t\ n\ R\ \phi \\
\mathit{Thunk}\ \mathcal{F}\ t\ n\ R\ \phi &\triangleq \exists T,\ d,\ \mathcal{F}'.\ \ulcorner \forall d'.\ d < d' \Rightarrow \mathcal{F}' \mathrel{\#} \uparrow(T \cdot d') \urcorner * \\
&\qquad \ulcorner \mathcal{F}' \subseteq \uparrow T \subseteq \mathcal{F} \urcorner * \\
&\qquad \mathit{RecThunk}\ d\ \mathcal{F}'\ t\ n\ R\ \phi
\end{aligned}
$$

Fig. 10. Thunks: Definition

credits. Out of these, n_1 credits are used to force the thunk, producing a value v such that $\Box\ \phi\ v$ holds. The remaining n_2 credits are then used to execute the ghost update and obtain $\Box\ \psi\ v$.

In this subsection, for simplicity, we focus on *one* application of the consequence rule. We assume that we have a predicate *Thunk* that satisfies the rules of Figure 6. We refer to this set of rules as the "common thunk API". We construct a new predicate *ProxyThunk*, which also satisfies the common thunk API. Its definition appears in Figure 9. The "creation rule" for proxy thunks, also shown in Figure 9, is a consequence rule that expects a *Thunk* and produces a *ProxyThunk*. The term "proxy thunk" is meant to suggest that a proxy thunk is a ghost wrapper around a pre-existing thunk.

The main components in the definition of proxy thunks are the underlying thunk, whose apparent debit is n_1, and the proxy thunk's piggy bank. The apparent debit n of the piggy bank is the apparent debit of the proxy thunk. The pending state of this piggy bank contains a one-shot ghost update from ϕ to ψ, whose cost is n_2. The equation $nc = n_1 + n_2$ records the fact that this piggy bank aims to collect enough credit to force the underlying thunk and apply this update. The forced state contains just a forced-thunk witness *ThunkVal* $t\ v$ together with the postcondition $\Box\ \psi\ v$.

The side condition $\mathcal{F}_1 \uplus \uparrow T \subseteq \mathcal{F}$ guarantees that out of the token $\lightning^{\mathcal{F}}$, which the user supplies when forcing the proxy thunk, we can extract the tokens $\lightning^{\mathcal{F}_1} * \lightning^{\uparrow T}$, which are required in order to simultaneously break the proxy's piggy bank and force the underlying thunk.

Theorem 4.2. *The predicate ProxyThunk satisfies the rule* Proxy-Create *in Figure 9. Furthermore, it satisfies all of the rules in Figure 6, where Thunk is replaced with ProxyThunk.*

4.2.3 Thunks. The construction of the previous subsection is heterogeneous and allows applying the consequence rule once: when applied to a thunk, it produces a proxy thunk. Fortunately, this construction is generic: it can be applied to an arbitrary predicate *Thunk*, provided this predicate is persistent and satisfies the common thunk API in Figure 6. Both *BasicThunk* and *ProxyThunk* meet these requirements. Thus, the construction can be iterated. We do so in Figure 10.

HThunk-Persist
$$\mathrm{persistent}(\mathit{HThunk}\ h\ t\ n\ \phi)$$

HThunk-Create
$$\{\$\underline{N} \ * \ \mathit{isAction}\ f\ n\ (\lightning^{h})\ \phi\}\ \mathit{create}\ f\ \{\lambda t.\ \mathit{HThunk}\ h\ t\ n\ \phi\}$$

HThunk-Inc-Height-Debit
$$\frac{h_1 \le h_2 \qquad n_1 \le n_2}{\mathit{HThunk}\ h_1\ t\ n_1\ \phi \mathrel{-\!\!*} \mathit{HThunk}\ h_2\ t\ n_2\ \phi}$$

HThunk-Pay
$$\frac{\uparrow\mathit{ThunkPayment} \subseteq \mathcal{E}}{\mathit{HThunk}\ h\ t\ n\ \phi \ * \ \$k \Rrightarrow_{\mathcal{E}} \mathit{HThunk}\ h\ t\ (n-k)\ \phi}$$

HThunk-Consequence
$$\begin{array}{l}\mathit{HThunk}\ h\ t\ n_1\ \phi \mathrel{-\!\!*} \\ (\forall v.\ \$n_2 \mathrel{-\!\!*} \square\, \phi\ v \Rrightarrow_{\top} \square\, \psi\ v) \Rrightarrow_{\mathcal{E}} \\ \mathit{HThunk}\ h\ t\ (n_1+n_2)\ \psi\end{array}$$

HThunk-Force
$$\begin{array}{l}\left\{\begin{array}{l}\mathit{HThunk}\ h\ t\ 0\ \phi \ * \\ \$\underline{F} \ * \ \lightning^{h'} \ * \ \ulcorner h < h' \urcorner\end{array}\right\} \\ \quad \mathit{force}\ t \\ \{\lambda v.\ \square\, \phi\ v \ * \ \mathit{ThunkVal}\ t\ v \ * \ \lightning^{h'}\}\end{array}$$

Fig. 11. Height-Indexed Thunks: Reasoning Rules

The definition is conceptually straightforward. First, we inductively define a predicate *RecThunk d*, which layers d proxy thunks on top of a base thunk. Then, we define the predicate *Thunk* via an existential quantification over d: that is, we say that a "thunk" is a basic thunk wrapped in an arbitrary number of proxy thunks. Two technical formulae involving masks record that (1) we have an infinite family of pairwise disjoint masks, namely $\uparrow(T \cdot d)$, where d is an integer index; and (2) after d levels of proxy thunks have been stacked above a basic thunk, the masks up to level d have been used up, but the masks above level d are still available for use.

Theorem 4.3. *The predicate Thunk satisfies all of the rules in Figures 5 and 6.*

A new piggy bank is created at two different times: when a thunk is first created, and when the consequence rule is applied to an existing thunk. Thus, an arbitrary number of piggy banks can be simultaneously associated with a single thunk, and can be simultaneously active. Fortunately, in our proofs, this global view is never needed.

5 HEIGHT-INDEXED THUNKS

The predicate *Thunk* is quite general but can be a little difficult to use. When a thunk is forced, one must *separately* supply the token $\lightning^{\mathcal{F}}$, which allows forcing the thunk itself, and the resource R, which allows the suspended computation to have certain effects. When one wishes to construct a thunk that forces one or more other thunks, the parameter R must typically be instantiated with a token of the form $\lightning^{\mathcal{F}'}$ where $\mathcal{F}$ and $\mathcal{F}'$ are disjoint. In short, we have set up a token-based discipline that forbids reentrant thunks. This is good, but this discipline can be heavy and confusing.

In order to address this difficulty once and for all and to save the end user some pain, we set up a simple system based on natural integer *heights h*. On top of the predicate *Thunk*, we define a new predicate $\mathit{HThunk}\ h\ t\ n\ \phi$ where the two parameters $\mathcal{F}$ and R are replaced with a single parameter h. Our intent is to allow a thunk at height h to force thunks at lower heights only. A thunk cannot force a thunk that lies at the same height as itself or higher. A thunk at height h can *construct* or *return* a thunk at an arbitrary height: no constraint relates the parameters h and ϕ.

For simplicity, this API removes the ability for a suspended computation to have side effects other than forcing thunks: that is, the parameter R disappears. It could be preserved if desired.

We omit the definition of the predicate *HThunk*, which is fairly administrative. Its reasoning rules appear in Figure 11. The affine token $\lightning^{h}$ allows forcing thunks whose height is less than h

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

Fig. 12. Streams: Definition

(HThunk-Force). When a thunk is created at height h, the token that is passed to the suspended computation is $\not{t}^h$ (HThunk-Create). Thus, the new thunk can force thunks at lower heights only. A height is not a creation time: indeed, a thunk at height 0 can be created after or created by a thunk at height 1. Instead, a height represents the length of a dependency chain: a thunk at height 2 is a thunk that can force a thunk that can force a thunk. Heights can be safely over-approximated: this is stated by HThunk-Inc-Height-Debit. In a token, h can be instantiated with ∞. The token $\not{t}^\infty$ can force thunks of arbitrary height. It appears in the API of the banker's queue (Figure 18).

6 STREAMS

A *stream* is a list whose elements are computed on demand and memoized. In lazy programming languages, such as Haskell, this data structure is referred to simply as a "list". In a strict programming language, such as OCaml, lists and streams are distinct (albeit closely related) data structures. The definition of streams as an algebraic data type appears in the first two lines of Figure 1. In short, a *stream* s is a thunk, which, once forced, produces a cell; and a *cell* is either the value *Nil* or a value of the form $\mathit{Cons}(x, s')$, where x is an element and s' is again a stream. A stream can be thought of as a chain of thunks, where each thunk produces the next thunk in the chain.

In the following, we define a predicate $\mathit{Stream}\ h\ s\ \mathit{ds}\ \mathit{xs}$, which describes a stream (§6.1); we establish several reasoning rules that this predicate satisfies (§6.2); and we establish specifications for a few common operations on streams (§6.3). We do not verify a full-fledged stream library; we verify only the operations needed by the banker's queue, which are shown in Figure 1.

6.1 The Predicate *Stream*

In the predicate $\mathit{Stream}\ h\ s\ \mathit{ds}\ \mathit{xs}$, the parameter h plays the same role as in the predicate *HThunk*. It is an integer height h, and it indicates that the token $\not{t}^h$ is required in order to force every thunk in the stream. The parameter s is the stream itself; it is the location in memory of the thunk that represents the head of the stream. The parameter xs is the sequence of the elements of the stream. It predicts the shape of the value produced by each thunk in the stream, where a shape is either *Nil* or $\mathit{Cons}(x, \_)$. The parameter ds is the sequence of debits associated with each thunk in the stream. It tells how much remains to be paid in order to force each thunk. It is worth noting that xs and ds predict the value and apparent cost of each thunk in the stream possibly *before* this thunk is even constructed in memory.

The definitions of the predicates *Stream* and *StreamCell* appear in Figure 12. They are mutually inductive. They are straightforward, so we do not paraphrase them. Let us just point out that they rely on height-indexed thunks: each section in the paper builds on the previous section. Because a stream of n elements involves $n + 1$ thunks, in an assertion $\mathit{Stream}\ h\ s\ \mathit{ds}\ \mathit{xs}$, one can informally[5] expect $|\mathit{ds}| = |\mathit{xs}| + 1$. In $\mathit{StreamCell}\ h\ c\ \mathit{ds}\ \mathit{xs}$, one can informally expect $|\mathit{ds}| = |\mathit{xs}|$.

[5]Technically, $\mathit{Stream}\ h\ s\ \mathit{ds}\ \mathit{xs} \vdash |\mathit{ds}| = |\mathit{xs}| + 1$ does *not* hold. A proof attempt fails, because the postcondition of a thunk does not hold until this thunk has been forced. One could strengthen the definition of *Stream* so that this entailment holds, but we have not felt the need to do so. The weaker entailment $\mathit{Stream}\ h\ s\ \mathit{ds}\ \mathit{xs} \vdash |\mathit{ds}| > 0$ does hold and has been sufficient.

Stream-Persist

$$\mathsf{persistent}(\mathit{Stream}\ h\ s\ ds\ xs)$$

Stream-Increase-Height

$$\frac{h_1 \le h_2}{\mathit{Stream}\ h_1\ s\ ds\ xs \twoheadrightarrow \mathit{Stream}\ h_2\ s\ ds\ xs}$$

Stream-Forward-Debit

$$\frac{\uparrow \mathit{ThunkPayment} \subseteq \mathcal{E} \qquad (m)\ ds_1 \le ds_2\ (n)}{\mathit{Stream}\ h\ s\ ds_1\ xs * \$m \Rrightarrow_{\mathcal{E}} \mathit{Stream}\ h\ s\ ds_2\ xs}$$

Stream-Force

$$\left\{\begin{array}{l}\mathit{Stream}\ h\ s\ (0 :: ds)\ xs\ * \\ \$\underline{F}' * \text{↯}^{h'} * \ulcorner h < h' \urcorner\end{array}\right\}\ \mathit{force}\ s\ \left\{\lambda c.\ \begin{array}{l}\mathit{StreamCell}\ h\ c\ ds\ xs\ * \\ \mathit{ThunkVal}\ s\ c * \text{↯}^{h'}\end{array}\right\}$$

Stream-Create

$$\{\$\underline{N}' * \mathit{isCellAction}\ h\ d\ e\ ds\ xs\}\ \mathit{create}\ (\lambda().e)\ \{\lambda s.\ \mathit{Stream}\ h\ s\ (d :: ds)\ xs\}$$

Fig. 13. Streams: Reasoning Rules

Several restrictions are intentionally built into our definition of the predicate *Stream*. First, by making the parameters *ds* and *xs* finite lists, we restrict our attention to finite streams. Second, by parameterizing *Stream* with *xs*, we restrict our attention to deterministic streams, whose elements can be predicted ahead of time. Third, by constructing streams on top of height-indexed thunks, which do not allow a thunk to have side effects (beside forcing other thunks), we restrict our attention to streams without side effects. Modeling potentially infinite streams, non-deterministic streams, or effectful streams is left to future work.

To allow possibly infinite streams, one should define *Stream* using guarded recursion [Jung et al. 2018, §5.6], which is possible because *HThunk* is *contractive*—a fact that we have proved. Then, *ds* and *xs* could be possibly infinite lists. We expect this approach to work, but have not investigated it.

6.2 Reasoning Rules for Streams

Our reasoning rules for streams appear in Figure 13. Most of them are reformulations of the corresponding rules for height-indexed thunks (Figure 11), so we do not explain them again. Stream-Force requires the head thunk to have zero debits. Stream-Create relies on the auxiliary predicate *isCellAction*[6] in the same way that Thunk-Create and HThunk-Create rely on *isAction*.

The most notable rule in Figure 13 is Stream-Forward-Debit. This rule allows managing a stream's debit in several ways. It allows paying (that is, consuming a number of time credits) so as to decrease the cost of a thunk, which can be either the head thunk or a deeper thunk. Furthermore, it allows moving debits from the right towards the left in the list *ds*. In other words, it allows transferring some of the debit of a faraway thunk to a thunk that lies nearer in the future. This is intuitively sound because this implies that one must pay earlier. Technically, the proof of soundness of Stream-Forward-Debit relies on the consequence rule HThunk-Consequence.

Stream-Forward-Debit involves the *debit subsumption* judgement $(m)\ ds_1 \le ds_2\ (n)$, whose intuitive meaning is as follows: provided one pays m time credits *now*, it is safe to transform the sequence ds_1 into the sequence ds_2, and this results in n leftover time credits *in the future*, after the thunks described by the lists ds_1 and ds_2 have been forced.

[6] We write $\mathit{isCellAction}\ h\ d\ e\ ds\ xs$ for **once** $\{\text{↯}^h * \$d\}\ e\ \{\lambda c.\ \mathit{StreamCell}\ h\ c\ ds\ xs * \text{↯}^h\}$. This assertion is a permission to execute the expression e at most once. It indicates that e may consume d time credits and must return a stream cell c such that $\mathit{StreamCell}\ h\ c\ ds\ xs$ holds. The resource ↯^h may be used and must be preserved.

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

Fig. 14. Subsumption over Sequences of Debits: Definition

$$\textsc{Sub-Variance}\ \frac{(m)\ ds_1 \leq ds_2\ (n) \quad m \leq m' \qquad n' \leq n}{(m')\ ds_1 \leq ds_2\ (n')} \qquad \textsc{Sub-Refl}\ (m)\ ds \leq ds\ (m) \qquad \textsc{Sub-Trans}\ \frac{(m_1)\ ds_1 \leq ds_2\ (n_1) \quad (m_2)\ ds_2 \leq ds_3\ (n_2)}{(m_1 + m_2)\ ds_1 \leq ds_3\ (n_1 + n_2)}$$

$$\textsc{Sub-Append}\ \frac{(m)\ ds_1 \leq ds_2\ (n) \quad (n)\ ds_1' \leq ds_2'\ (k)}{(m)\ ds_1 \mathbin{++} ds_1' \leq ds_2 \mathbin{++} ds_2'\ (k)} \qquad \textsc{Sub-Add-Slack}\ \frac{(m)\ ds_1 \leq ds_2\ (n)}{(m + k)\ ds_1 \leq ds_2\ (n + k)} \qquad \textsc{Sub-Repeat}\ \frac{d_1 \leq d_2}{(0)\ d_1^n \leq d_2^n\ (n \times (d_2 - d_1))}$$

Fig. 15. Subsumption over Sequences of Debits: Properties

The presence of the parameter n in this judgement may seem surprising, especially since the n left-over credits are unused (wasted) by Stream-Forward-Debit. Still, this parameter is useful because it enables compositional proofs of subsumption; this is most visible in Sub-Append (Figure 15).

An inductive definition of the subsumption judgement appears in Figure 14. In Sub-Cons, it may be the case that d_1 is greater than d_2. In this case, the premise $d_1 \leq m + d_2$ allows part of the m time credits at hand to be spent on the first thunk, *decreasing* its apparent cost from d_1 to d_2. It may also be the case that d_1 is less than or equal to d_2. In that case, the apparent cost of the first thunk is *increased*, causing *more than m* time credits to become available for spending on the thunks that follow. In either case, the number of credits that can be spent on the tail of the stream is $(m + d_2) - d_1$, which is why this number appears in the second premise of Sub-Cons.

An alternative characterization of the subsumption judgement, which helps see why Stream-Forward-Debit is sound, is the following. Intuitively, for this rule to be sound, it must be the case that, by applying this rule, one promises to pay no less and to pay no later. Thus, for every index i, it must be the case that forcing the first i thunks in the stream appears no less expensive after the rule is applied than it did before the rule was applied. The subsumption judgement indeed satisfies a property of this kind: it is expressed by the following lemma.

Lemma 6.1 (Subsumption of Sequences of Debits, expressed in terms of Partial Sums). *Suppose the lists ds_1 and ds_2 have the same length. Then the judgement $\exists n.\ (m)\ ds_1 \leq ds_2\ (n)$ is equivalent to $\forall i.\ \sum(take\ i\ ds_1) \leq m + \sum(take\ i\ ds_2)$.*

The subsumption judgement enjoys a number of reasoning rules, which are presented in Figure 15. The judgement $(m)\ ds_1 \leq ds_2\ (n)$ is covariant in m and contravariant and n (Sub-Variance). It is reflexive (Sub-Refl), transitive (Sub-Trans), and compatible with list concatenation (Sub-Append). Extra credit at the left end translates to extra credit at the right end (Sub-Add-Slack). Finally, increasing the apparent cost of the first n thunks from d_1 to d_2 results in $n \times (d_2 - d_1)$ extra credit at the right end (Sub-Repeat). As a more readable special case, an increase of 1 in the debit of the first n thunks justifies a decrease of n in the debit of the thunk that follows. In other words, an expensive thunk can be made to appear cheap provided it lies far enough away in the future. This reasoning rule plays a key role in Okasaki's amortized analysis of the banker's queue (§7).

Stream-Revl

$$\{List\ l\ xs\ *\ \$\underline{R}'\ *\ \ulcorner n = |xs| \urcorner\}$$
$$revl\ l$$
$$\{\lambda s.\ Stream\ h\ s\ (\underline{R}n :: 0^n)\ (rev\ xs)\}$$

Stream-Append

$$\left\{\begin{array}{l} Stream\ h\ s_1\ ds_1\ xs_1\ * \\ Stream\ h\ s_2\ ds_2\ xs_2\ *\ \$\underline{P} \end{array}\right\}$$
$$append\ s_1\ s_2$$
$$\{\lambda s.\ Stream\ (h+1)\ s\ (ds_1 \bowtie ds_2)\ (xs_1 +\!\!+\ xs_2)\}$$

Fig. 16. Streams: Specifications of *revl* and *append*

6.3 Operations on Streams

There remains to present the specifications of the operations on streams whose code appears in Figure 1. For the sake of brevity, we omit the specifications of the tiny functions *nil* and *uncons*: they are similar to Stream-Create and Stream-Force. We also omit the specification of *revl_append*. We present the specifications of the last two functions, *revl* and *append*, in Figure 16.

Stream-Revl states that *revl* transforms an immutable list l whose elements form the sequence xs into a stream s whose elements form the sequence $rev\ xs$. (The pure assertion $List\ l\ xs$ indicates that the value l is an immutable list whose elements form the sequence xs.) If the sequence xs has length n, then this stream involves $n+1$ thunks. The postcondition in Stream-Revl indicates that the first thunk is expensive, while the remaining n thunks are cheap: the first thunk carries debit $\underline{R}n$ (where $\underline{R}$ is a constant), while every other thunk carries debit zero. The first thunk is expensive because, when it is forced, $revl\_append\ \ l\ \ Nil$ is invoked (line 23 in Figure 1). This function call requires linear time because it eagerly traverses the list l and immediately constructs the remaining n thunks. These thunks are cheap because they immediately return a pre-existing value. The function call $revl\ l$ itself has constant time complexity: Stream-Revl requires a constant amount $\underline{R}'$ of time credits.

Stream-Append states that $append\ s_1\ s_2$ has constant complexity: it consumes $\underline{P}$ time credits. If the streams s_1 and s_2 represent the sequences of elements xs_1 and xs_2 then the stream returned by *append* represents the sequence $xs_1 +\!\!+\ xs_2$. More crucially, if the sequences of debits associated with s_1 and s_2 are ds_1 and ds_2, then the sequence of debits associated with this stream is $ds_1 \bowtie ds_2$. The *debit join* operation $\bowtie$ is defined as follows, where $\underline{A}$ and $\underline{B}$ are integer constants:

$$(ds_1 +\!\!+ [d_1]) \bowtie (d_2 :: ds_2) \quad \triangleq \quad map\ (\underline{A} + \_)\ ds_1\ +\!\!+\ (\underline{A} + d_1 + \underline{B} + d_2) :: ds_2$$

If ds_1 has length n_1+1 and ds_2 has length $1+n_2$ then $ds_1 \bowtie ds_2$ has length n_1+1+n_2. The computation of $ds_1 \bowtie ds_2$ can be informally described as follows: first, add $\underline{A}$ to every element of ds_1; then, meld the two sequences, by fusing (adding) the last element of the first sequence with the first element of the second sequence; finally, add $\underline{B}$ to this fused element. This specification reflects the fact that (1) the overall cost of a stream concatenation operation is $\underline{A}(n_1+1)+\underline{B}$, where n_1 is the number of elements of the first stream; and (2) the cost of concatenation is distributed across the first n_1+1 thunks of the result stream: each of the first n_1 thunks bears a cost of $\underline{A}$; the next thunk bears a cost of $\underline{A}+\underline{B}$; and the remaining n_2 thunks bear no cost.

Stream-Append states that if the streams s_1 and s_2 have height h then the stream returned by *append* has height $h+1$. This reflects the fact that a thunk in this new stream can depend on (force) a thunk in the stream s_1 or in the stream s_2. Such precise height information is necessary during the inductive proof of *append*, and can be necessary also in some usage scenarios of *append*. In the banker's queue (§7), it is not needed: there, we work with streams of unknown height.

```
type 'a queue =
  { lenf: int; f: 'a stream; lenr: int; r: 'a list }

let empty () =
  { lenf = 0; f = nil(); lenr = 0; r = [] }

let check ({ lenf = lenf ; f = f; lenr = lenr; r = r } as q) =
  if lenf >= lenr then q
  else { lenf = lenf + lenr; f = append f (revl r); lenr = 0; r = [] }

let snoc q x =
  check { q with lenr = q.lenr + 1; r = x :: q.r }

let extract q =
  let x, f = uncons q.f in
  x, check { q with f = f; lenf = q.lenf - 1 }
```

Fig. 17. The Banker's Queue: OCaml Code

7 THE BANKER'S QUEUE

The banker's queue [Okasaki 1999, §6.3.2] is a persistent FIFO queue whose main operations, *snoc* and *extract*, have constant amortized time complexity. In the following, we present a specification for the banker's queue, explain how the banker's queue is implemented, and verify that the code satisfies the specification. Because the implementation involves a stream,[7] we rely on the stream library presented in the previous section (§6).

It is worth pointing out that we do not simply replicate Okasaki's analysis of the queue. Instead, we propose a *simpler analysis*, which is made possible by the powerful reasoning rule Stream-Forward-Debit. Instead of working with iterated sums of debits, as Okasaki does, we use a sequence of elementary proof steps that rely on the properties of debit subsumption (Figures 14 and 15).

Interface and Specification of the Banker's Queue. Three functions make up the main entry points of the library: *empty* creates a new empty queue; *snoc* inserts an element at the rear end of a queue; *extract* extracts an element at the front end of a queue.

Every operation has constant amortized time complexity. This implies that every sequence of operations on queues has linear cost in the number of operations. This is true even if queues are used in a non-linear manner, that is, even if operations are applied not only to the newest version of a queue, but also to old versions.

Our formal specification of the banker's queue appears in Figure 18. The assertion *BQueue q xs* means that *q* is a queue whose elements form the sequence *xs*. This assertion is persistent. This means that queues can be shared and that the queue operations are not destructive: one may apply an operation to an old queue. The rules Banker-Empty, Banker-Snoc and Banker-Extract provide specifications for *empty*, *snoc*, and *extract*. Each of them requires a constant amount of time credits in its precondition ($\underline{E}$, $\underline{S}$ and $\underline{X}$ respectively). The specification of *extract* requires and preserves the token $\lightning^{\infty}$, which allows forcing thunks of arbitrary height (§5). Requiring this token is

[7]In Okasaki's code, for uniformity, two streams are used. However, only the front stream actually must be a stream: as noted by Okasaki [1999] in a footnote on page 64, the rear stream can be a list. We use a rear list and verify that Okasaki's footnote is correct. We believe that there would be no difficulty in verifying Okasaki's version, where a rear stream is used. Every suspension in the rear stream would be fully paid for upon construction and would therefore have debit zero.

Banker-Persistent
$$\mathsf{persistent}(BQueue\ q\ xs)$$

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

Banker-Snoc
$$\{\$\underline{S}\ *\ BQueue\ q\ xs\}\ snoc\ q\ x\ \{\lambda q'.\ BQueue\ q'\ (xs \mathbin{++} [x])\}$$

Banker-Extract
$$\{\$\underline{X}\ *\ BQueue\ q\ (x :: xs)\ *\ ↯^{\infty}\}\ extract\ q\ \{\lambda(x', q').\ \ulcorner x' = x \urcorner\ *\ BQueue\ q'\ xs\ *\ ↯^{\infty}\}$$

Fig. 18. Banker's Queue: Public Interface

$$\begin{array}{rcl}
bqueueDebits\ nf\ nr & \triangleq & \underline{K}^{nf-nr} \mathbin{++} 0^{\min(nf,nr)+1} \\
BQueueRaw\ q\ fs\ rs & \triangleq & \exists s,\ h,\ l. \left\{ \begin{array}{l} \ulcorner q = (|fs|, s, |rs|, l) \urcorner\ * \\ Stream\ h\ s\ (bqueueDebits\ |fs|\ |rs|)\ fs\ *\ List\ l\ rs \end{array} \right. \\
BQueue\ q\ xs & \triangleq & \exists fs,\ rs.\ BQueueRaw\ q\ fs\ rs\ *\ \ulcorner xs = fs \mathbin{++} rev\ rs \wedge |rs| \le |fs| \urcorner
\end{array}$$

Fig. 19. Banker's Queue: Definitions

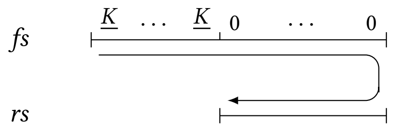

Fig. 20. A balanced banker's queue. The front stream is annotated with debits. The arrow indicates the logical order of elements.

Banker-Check
$$\{\$\underline{H} * BQueueRaw\ q\ fs\ rs * \ulcorner |rs| \le |fs| + 1 \urcorner\}$$
$$check\ q$$
$$\{\lambda q'.\ BQueue\ q'\ (fs \mathbin{++} rev\ rs)\}$$

Fig. 21. Banker's Queue: Specification of *check*

necessary because HeapLang has shared-memory concurrency and our implementation of thunks is (by design) not thread-safe. The token discipline prevents two threads from racing on a thunk.

Implementation of the Banker's Queue. The implementation appears in Figure 17. A queue is a record of four fields: a "front" stream of elements `f`, a "rear" list of elements `r`, and their respective lengths, `lenf` and `lenr`. Elements are inserted into the queue by prepending to the rear list, and are extracted from the queue by extracting from the front stream. As a result, the elements of the rear list are stored in logically reverse order: if the elements of the front stream form the sequence *fs* and if the elements of the rear list form the sequence *rs*, then the sequence of elements contained in the queue is $fs \mathbin{++} rev\ rs$. The inequality $|rs| \le |fs|$ is maintained: the rear list never contains more elements than the front stream.

When the length of the rear list exceeds the length of the front stream, the queue must be rebalanced. This is done by the auxiliary function *check*. Rebalancing involves reversing the rear list, converting it into a stream, and appending this stream at the end of the front stream. The reversal and conversion into a stream are performed by *revl*. According to Stream-Revl (Figure 16), an invocation of *revl* has constant time complexity, but returns a stream whose first thunk has linear cost. Thus, rebalancing itself is cheap, but constructs an expensive thunk, which (after rebalancing) appears in the middle of the front stream. Okasaki's insight is that the cost of this expensive thunk can be distributed onto the linear number of thunks that appear in front of it. This translates to a constant amount of extra debit per thunk, which is acceptable.

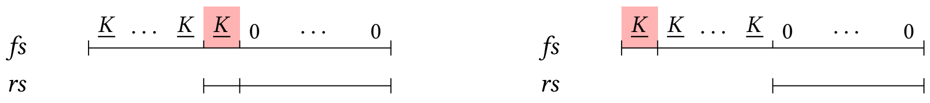

Fig. 22. Left: after *snoc* has inserted an element in the rear list. Right: before *extract* removes an element of the front stream. Highlighted in red: the thunk whose debit must be paid off.

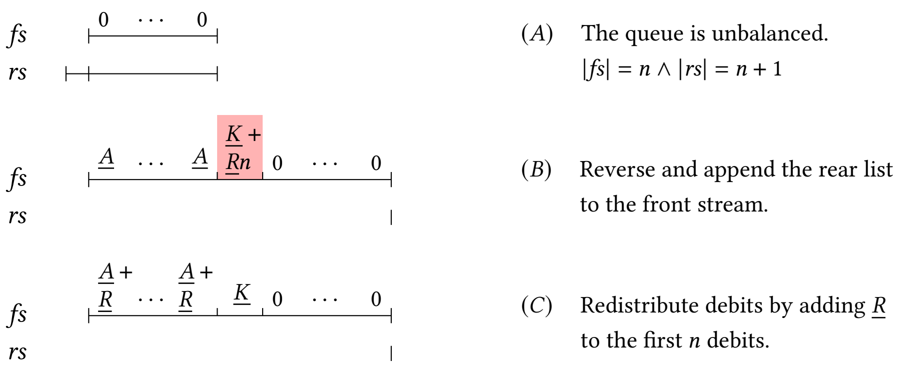
fs
rs
0 ... 0
(A) The queue is unbalanced.
|fs| = n ∧ |rs| = n + 1
fs
rs
A ... A K + Rn 0 ... 0
(B) Reverse and append the rear list to the front stream.
fs
rs
A + R ... A + R K 0 ... 0
(C) Redistribute debits by adding R to the first n debits.

Fig. 23. Rebalancing. In red: the costly thunk whose debit must be distributed among the front thunks.

The Predicate BQueue. Figure 19 presents the definition of *BQueue q xs*. It is constructed by combining the assertions $xs = fs \mathbin{++} rev\ rs$ and $|rs| \leq |fs|$, which we have explained already, with a lower-level assertion, *BQueueRaw q fs rs*. This assertion states that q is a 4-tuple, requires the two length fields to contain the integer values $|fs|$ and $|rs|$, and uses the predicates *Stream* and *List* to describe the front stream and rear list. The most noteworthy aspect is that the debit sequence of the front stream is fully determined: it is *bqueueDebits* $|fs|\ |rs|$. The definition of *bqueueDebits* states that the first $nf - nr$ thunks in the front stream carry debit $\underline{K}$, where $\underline{K} \triangleq \underline{A} + \underline{B} + \underline{R}$, whereas the remaining thunks carry debit zero. This is illustrated in Figure 20. In our illustrations (Figures 20, 22, and 23), the debit associated with the very last thunk of the front stream, which is always 0, is never shown.

Specification of check. The specification of the function *check*, which rebalances a queue, appears in Figure 21. It states that *check* accepts an imbalanced queue and returns a balanced queue. Because *check* is called by every operation, *check* can expect that the length of the rear list exceeds the length of the front stream by at most one.

Verifying snoc and extract. snoc causes the rear list to grow by one element. To preserve the debit invariant (Figure 20), one must pay for the last thunk in the front stream whose debit is nonzero. This is illustrated in Figure 22 (left). This is done by applying Stream-Forward-Debit. This requires proving the subsumption judgement $(\underline{K})\ \underline{K}^n \mathbin{++} \underline{K} :: 0^m \leq \underline{K}^n \mathbin{++} 0 :: 0^m\ (0)$, which follows from Sub-Append, Sub-Cons, and Sub-Refl.

The function *extract* forces and discards the first thunk of the front stream, as pictured in Figure 22 (right). Thus, it is necessary to first pay for this thunk. This can be done by applying Stream-Forward-Debit and proving the subsumption judgement $(\underline{K})\ \underline{K} :: \underline{K}^n \mathbin{++} 0^m \leq 0 :: \underline{K}^n \mathbin{++} 0^m\ (0)$, which follows from Sub-Cons and Sub-Refl. This is a trivial instance of Stream-Forward-Debit, that is, an ordinary payment as opposed to a deep payment.

Verifying check. Because *check* empties the rear list, it is clear that it restores the invariant $|rs| \leq |fs|$. How *check* restores the debit invariant is more subtle. Let us consider an unbalanced queue whose front and rear sequences of elements are *fs* and *rs*. Let us write n for $|fs|$, so that we have $|rs| = n + 1$. According to the debit invariant, every thunk in the front stream has debit zero. This situation is represented in step (A) of Figure 23. The proof proceeds as follows:

(1) According to STREAM-REVL and STREAM-APPEND, reversing the rear list and appending the result to the front stream produces a stream whose debits are $0^{n+1} \bowtie \underline{R}(n+1) :: 0^{n+1}$. This debit sequence has length $2n+2$: this is consistent with the fact that the new front stream has $2n+1$ elements. By definition of the debit join operator $\bowtie$, this is: $\underline{A}^n \mathbin{++} (\underline{A} + \underline{B} + \underline{R}(n+1)) :: 0^{n+1}$. Because $\underline{K}$ is $\underline{A} + \underline{B} + \underline{R}$, this sequence of debits is also $\underline{A}^n \mathbin{++} (\underline{K} + \underline{R}n) :: 0^{n+1}$. It is depicted in step (B) of Figure 23.

(2) Then, the key step of the proof is to *distribute* the expensive debit $\underline{K} + \underline{R}n$ onto earlier debits. We increase each of the first n debits by $\underline{R}$, so as to be allowed to reduce the expensive debit from $\underline{K} + \underline{R}n$ down to $\underline{K}$. The result is illustrated in step (C) of Figure 23. This redistribution of debit is permitted by STREAM-FORWARD-DEBIT provided we establish the subsumption judgement $(0)\ \underline{A}^n \mathbin{++} (\underline{K} + \underline{R}n) :: 0^{n+1} \leq (\underline{A} + \underline{R})^n \mathbin{++} \underline{K} :: 0^{n+1}\ (0)$. This judgement follows from SUB-APPEND, SUB-REPEAT, SUB-CONS, and SUB-REFL.

(3) Because $\underline{A} + \underline{R} \leq \underline{K}$ holds, we can now over-approximate every debit by $\underline{K}$, except the last one, which must remain zero. We exploit the subsumption judgement $(0)\ (\underline{A} + \underline{R})^n \mathbin{++} \underline{K} :: 0^{n+1} \leq \underline{K}^{2n+1} \mathbin{++} [0]\ (0)$. The debit sequence $\underline{K}^{2n+1} \mathbin{++} [0]$ is equal to *bqueueDebits* $(2n+1)\ 0$, which is the expected sequence of debits for a balanced queue whose front stream has length $2n + 1$ and whose rear list is empty.

8 THE PHYSICIST'S QUEUE; IMPLICIT QUEUES

We verify two additional persistent data structures found in Okasaki's book. Both exploit thunks to achieve amortized constant time complexity.

The physicist's queue [Okasaki 1999, §6.4.2] is similar to the banker's queue insofar as it involves front and rear lists of elements and rebalances them when necessary. However, a physicist's queue involves a single thunk, which lies at the root of the data structure, whereas the banker's queue involves nested thunks (a stream). For this reason, the analysis of the physicist's queue is easier: in particular, the rule THUNK-CONSEQUENCE is not needed. Okasaki analyzes the physicist's queue using the *physicist's method*, which associates a "potential function" with the data structure. We use the *banker's method*, which associates a debit with each thunk. Because this data structure involves only one thunk, the two methods essentially coincide in this case. The physicist's queue does involve a thunk that forces another thunk. Our height-indexed thunks (§5) allow this.

Implicit queues [Okasaki 1999, §11.1] are a more complex data structure. They rely on thunks directly, and do not use streams. Their implementation involves recursive slowdown: the queue is structured in layers, where each layer stores twice as many elements as the previous layer. When one layer is at hand, accessing the next layer requires forcing a thunk. Therefore, an implicit queue has the same general structure as a stream: it involves a sequence of nested thunks. Our implementation and our debit invariant closely follow Okasaki's. They match Danielsson's as well [Danielsson 2008, §8.1], although Danielsson chooses a slightly different way of defining the data structure. To carry out the complexity analysis, we make full use of the height-indexed thunk API (§5). Here, the use of THUNK-CONSEQUENCE is crucial.

9 RELATED WORK

Okasaki [1999] invents the debit-based approach to the amortized time complexity analysis of lazy, purely functional data structures. He describes this approach in a clear but informal way. Danielsson [2008] uses Agda to define a formal complexity analysis, to prove its soundness, and to verify some of Okasaki's data structures, including the banker's queue and implicit queues. Deep payment is used in the verification of the banker's queue, but is not supported in the proof of soundness of the analysis. Like Okasaki's informal discipline, Danielsson's system is purely based on debits and does not include a subsystem that aims to forbid reentrancy, such as our "height" discipline (§5). It could be that he does not need such a subsystem because his type system guarantees termination. However, in the presence of the fixed point combinator fix, it is not clear whether this is true. Danielsson does not prove that every well-typed program terminates. He establishes a weak time complexity guarantee: if a program has type τ and *if this program reaches a weak head normal form* in n steps then $n \leq time(\tau)$ holds. Thus, the possibility that the program diverges remains open. Atkey [2011] suggests extending separation logic with time credits and using it to carry out amortized time complexity analyses. Pilkiewicz and Pottier [2011] independently introduce the concept of time credit in an affine type system and suggest that time credits, in combination with monotonic ghost state and a form of invariant, can be used to reconstruct Okasaki's debit-based analysis of thunks. Their work is however informal.

Mével et al. [2019] carry out a similar programme in the formal setting of Iris$^{\$}$, which they construct on top of Iris. They do not verify any of Okasaki's algorithms. In fact, their reasoning rules exhibit three main shortcomings, which limit their expressive power.

First, their specification of *force* [Mével et al. 2019, Figure 6] requires and returns an affine token $\lightning$ that is not transmitted to the suspended computation: this prevents a thunk from forcing a thunk. They acknowledge this restriction and note that they have implemented "a more flexible discipline" based on more fine-grained tokens, but do not describe it. We develop such a discipline. In our low-level thunk API, the rule Thunk-Force requires and returns an affine token $\lightning^{\mathcal{F}}$ and transmits a resource R to the suspended computation. Instantiating R with suitable tokens allows a thunk to force other thunks. Our high-level thunk API, which is novel, takes advantage of this. There, thunks and tokens are indexed with heights, and the token $\lightning^{h}$ allows forcing thunks whose height is less than h: this is visible in the rule HThunk-Force.

Second, Mével et al.'s construction of the predicate *isThunk* [Mével et al. 2019, §7.4] does not validate *deep payment*, which allows shifting debits from the far future to the near future. A 2-debit thunk whose result is a 1-debit thunk can be viewed, via a deep payment, as a 3-debit thunk whose result is a 0-debit thunk. Deep payment changes both the debit n and the postcondition ϕ of a thunk: in the previous example, a thunk whose debit is 2 and whose promise (postcondition) is to return a 1-debit thunk becomes a thunk whose debit is 3 and whose promise is to return a 0-debit thunk. Yet, in Mével et al.'s construction, the parameters n and ϕ appear inside an Iris non-atomic *invariant*. This forbids any change in n or ϕ and precludes deep payment. Coming up with an alternative construction that removes this restriction is not easy: one must keep in mind that, so that sharing thunks is permitted, the predicate *Thunk* must be *persistent*. We justify deep payment, which we obtain as a combination of the rules Thunk-Consequence and Thunk-Pay. Our key insight is to separate *thunks*, a data structure that exists at runtime, and *piggy banks*, a ghost data structure. In our construction, an application of the rule Thunk-Consequence creates a *new* piggy bank for an *existing* thunk, so, in the end, many piggy banks can be associated with a single thunk. This may seem mind-boggling; fortunately, by virtue of working in Separation Logic, we reason in a local manner, and never need to think about more than one piggy bank at a time.

Third, in Mével et al.'s construction, both *force* and *pay* require an affine token. This forbids not only a thunk that forces another thunk, but also a thunk that performs a payment on another thunk. This is inconvenient, potentially problematic, and seems intuitively unnecessary, because payment is a ghost operation. Our rule Thunk-Pay does not require any token.

Inspired by Danielsson's work, McCarthy et al. [2016] define in Coq a monad that keeps track of costs. They place emphasis on obtaining clean OCaml code via Coq's extraction facility. They use the pure, call-by-value fragment of OCaml; no thunks are involved. Also inspired by Danielsson, Handley et al. [2020] develop a semi-automated system, based on Liquid Haskell, to verify the time complexity of Haskell programs. A *pay* combinator is supported; deep payment is not. The soundness of the system is stated but not formally verified. Madhavan et al. [2017] present a system that infers and verifies resource bounds for higher-order functional programs that involve thunks or memoization tables. Nipkow and Brinkop [2019] verify the amortized complexity of several functional data structures in Isabelle/HOL. These data structures do not involve thunks, and the analysis is credit-based, not debit-based. Hackett and Hutton [2019] propose both an operational semantics and a denotational cost semantics for lazy (call-by-need) programs, based on the idea of *clairvoyant evaluation*, where the mutable state inherent in thunks is replaced with nondeterminism. Inspired by this idea, Li et al. [2021] define the *clairvoyance monad*, a model of laziness that is shallowly embedded inside Coq, and develop two program logics of over- and under-approximation to reason about the cost of lazy programs. They do not reason in terms of debits.

10 CONCLUSION

We have proposed a set of debit-based reasoning rules for thunks and proved their soundness in a foundational, machine-checked setting, namely separation logic with time credits. We have checked that our rules suffice to verify three of Okasaki's data structures, namely the banker's queue, the physicist's queue, and implicit queues. We believe that our rules capture all of Okasaki's informal reasoning principles and that they should suffice to verify the rest of Okasaki's book. However, this can be ascertained only by actually performing this work!

We view this result as an enlightening and useful bridge between the worlds of purely functional programming and imperative programming.

From a technical point of view, ghost piggy banks are an original concept. Our modular construction of thunks on top of piggy banks, in several steps, is also original. A monolithic definition could be given, but we believe that it would be much more difficult to understand.

Our proofs are available online [Pottier et al. 2023]. One area where engineering work is needed is in the quality of the implementation of Iris$^{\$}$ [Mével et al. 2019]. The fact that Iris$^{\$}$ is implemented on top of Iris via a program transformation, the *tick translation*, should be an implementation detail; yet it is currently apparent. We find that this creates unnecessary difficulty for the end user.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their remarks and suggestions. In particular, a suggestion by one of the reviewers led to a simplified definition of piggy banks, which involves a single invariant (Figure 3). Our earlier definition used two distinct invariants and was slightly more complex.

DATA AVAILABILITY STATEMENT

Our mechanized proofs are available in our repository [Pottier et al. 2023] and as an artifact at https://zenodo.org/doi/10.5281/zenodo.8425002.

REFERENCES

Robert Atkey. 2011. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science* 7, 2:17 (2011). http://bentnib.org/amortised-sep-logic-journal.pdf

Arthur Charguéraud and François Pottier. 2017. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning* (Sept. 2017). http://cambium.inria.fr/~fpottier/publis/chargueraud-pottier-uf-sltc.pdf

Nils Anders Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Principles of Programming Languages (POPL)*. http://www.cse.chalmers.se/~nad/publications/danielsson-popl2008.pdf

James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. 1989. Making Data Structures Persistent. *J. Comput. System Sci.* 38, 1 (1989), 86–124. https://doi.org/10.1016/0022-0000(89)90034-2

Jennifer Hackett and Graham Hutton. 2019. Call-by-need is clairvoyant call-by-value. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 114:1–114:23. https://doi.org/10.1145/3341718

Martin A. T. Handley, Niki Vazou, and Graham Hutton. 2020. Liquidate your assets: reasoning about resource usage in Liquid Haskell. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 24:1–24:27. https://doi.org/10.1145/3371092

Maximilian P. L. Haslbeck and Peter Lammich. 2021. For a Few Dollars More - Verified Fine-Grained Algorithm Analysis Down to LLVM. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 12648)*. Springer, 292–319. https://www21.in.tum.de/~haslbema/documents/Haslbeck_Lammich_LLVM_with_Time.pdf

Maximilian P. L. Haslbeck and Tobias Nipkow. 2018. Hoare Logics for Time Bounds: A Study in Meta Theory. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 10805)*. Springer, 155–171. https://www21.in.tum.de/~nipkow/pubs/tacas18.pdf

Jan Hoffmann, Michael Marmar, and Zhong Shao. 2013. Quantitative Reasoning for Proving Lock-Freedom. In *Logic in Computer Science (LICS)*. 124–133. http://www.cs.cmu.edu/~janh/papers/lockfree2013.pdf

John Hughes. 1989. Why Functional Programming Matters. *Computer Journal* 32, 2 (1989), 98–107. http://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf

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

Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. 2017. Contract-based resource verification for higher-order functions with memoization. In *Principles of Programming Languages (POPL)*. 330–343. https://doi.org/10.1145/3009837.3009874

Simon Marlow, Simon L. Peyton Jones, and Satnam Singh. 2009. Runtime support for multicore Haskell. In *International Conference on Functional Programming (ICFP)*. 65–78. https://www.microsoft.com/en-us/research/wp-content/uploads/2009/09/multicore-ghc.pdf

Jay A. McCarthy, Burke Fetscher, Max S. New, Daniel Feltey, and Robert Bruce Findler. 2016. A Coq Library for Internal Verification of Running-Times. In *Functional and Logic Programming (Lecture Notes in Computer Science, Vol. 9613)*. Springer, 144–162. https://users.cs.northwestern.edu/~robby/publications/papers/flops2016-mfnff.pdf

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time credits and time receipts in Iris. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 11423)*. Springer, 1–27. http://cambium.inria.fr/~fpottier/publis/mevel-jourdan-pottier-time-in-iris-2019.pdf

Tobias Nipkow and Hauke Brinkop. 2019. Amortized Complexity Verified. *Journal of Automated Reasoning* 62, 3 (2019), 367–391. https://www21.in.tum.de/~nipkow/pubs/jar18.pdf

Chris Okasaki. 1999. *Purely Functional Data Structures.* Cambridge University Press. https://doi.org/10.1017/CBO9780511530104

Alexandre Pilkiewicz and François Pottier. 2011. The essence of monotonic state. In *Types in Language Design and Implementation (TLDI)*. http://cambium.inria.fr/~fpottier/publis/pilkiewicz-pottier-monotonicity.pdf

François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. 2023. Thunks and Debits in Separation Logic with Time Credits: Coq Formalization. https://gitlab.inria.fr/cambium/iris-time-proofs.

Robert Endre Tarjan. 1985. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods* 6, 2 (1985), 306–318. http://dx.doi.org/10.1137/0606031

Bohua Zhan and Maximilian P. L. Haslbeck. 2018. Verifying Asymptotic Time Complexity of Imperative Programs in Isabelle. In *International Joint Conference on Automated Reasoning*. http://arxiv.org/abs/1802.01336

Received 2023-07-11; accepted 2023-11-07