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, No Institution, 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 debt with every thunk. A debt can be paid off in several increments; a thunk whose debt has been fully paid off can be forced. Quite strikingly, a debt is associated also with future thunks, which do not yet exist in memory. Some of the debt 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**.

ACM Reference Format:
François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. 2023. Thunks and Debits in Separation Logic with Time Credits. 1, 1 (March 2023), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This paper is concerned with program verification techniques that not only can guarantee that a program does not crash and produces a correct result, but also can bound the time complexity of this program.[1] We are interested in verification techniques that apply to actual executable source code, as opposed to pseudocode; in compositional verification techniques, which allow verifying a program component (say, a data structure) independently of its uses; and in formal verification techniques, which allow constructing machine-checked proofs that rely on a small trusted base.

Two Strands of Previous Work. Inside this general area, we are more specifically interested in bridging the gap between two strands of previous work. One strand is concerned with the analysis of imperative data structures and algorithms. The other is concerned with the analysis of lazy purely functional data structures. These data structures involve *suspensions*, also known as *thunks*.

[1]As is traditional in the analysis of algorithms, by the *time complexity* of a program, we mean the number of instructions required to execute this program. Such an instruction count is not necessarily closely related with, or a good predictor of, an actual execution time on a physical machine.

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, No Institution, Paris, France.

XXXX-XXXX/2023/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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.

In the first strand, to verify the time complexity of strict, possibly imperative programs, several *separation logics with time credits* [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] have been developed. Time credits do not exist at runtime: they are assertions. They typically appear in function preconditions and in data structure invariants. Every instruction consumes one credit; duplicating or forging credits is forbidden. These logics allow worst-case time complexity analyses, including amortized analyses in the style pioneered by Tarjan [1985], where time credits are set aside in a data structure invariant so as to pay for future expensive operations.

In the second strand, to analyze the amortized complexity of purely functional data structures, which exploit lazy evaluation to achieve good complexity, Okasaki [1999] presents a different approach. He points out a fundamental limitation of credit-based analyses: because time credits are *affine* (not duplicable or shareable), they cannot be used in the analysis of a *persistent* (shareable) data structure. To work around this limitation, Okasaki reasons in terms of *debits* instead of credits. A debit is a debt that is associated with a thunk. It can be understood as the number of credits that remain to be paid before a thunk can be forced. Okasaki's key insight is that it is safe to duplicate a debt: this leads to over-approximating the true cost of a computation, which is acceptable. Therefore, in a debit-based system, thunks can be viewed as persistent (shareable) data structures.

Bridging the Gap. What do we mean by "bridging the gap" between these two strands of work? Why is it desirable to bridge this gap? What preliminary results does the literature offer in this direction, and what is missing still?

By "bridging the gap", we mean that we propose to reconstruct Okasaki's reasoning rules and complexity analyses on top of a separation logic with time credits. In so doing, we reap several benefits. We show that there is in fact no gap, and that Okasaki's reasoning rules can be presented as a *library* inside an existing logic. We shed new light on these rules and on the (nontrivial) reasons why they are sound. We show that they are compatible with a rich separation logic and can be applied to programs that exploit not only thunks but also imperative features. Finally, whereas Okasaki's analyses are carried out with pen and paper, we rely on a separation logic whose metatheory is machine-checked and we carry out machine-checked complexity analyses, thereby achieving a high degree of assurance and relying on a small trusted base. In this paper, we use Iris$^{\$}$ [Mével et al. 2019], an extension of the program logic Iris [Jung et al. 2018] with time credits.

Stepping Stones. To help bridge this gap, two important stepping stones exist in the literature. First, Danielsson [2008] proposes the first formal account of Okasaki's reasoning rules, in the form of an Agda library. The library offers an abstract type *Thunk* n a and a number of operations on this type. The integer parameter n is the debt associated with this thunk; the parameter a is the type of its result. The type *Thunk* serves both as the type of computations (it is the ambient monad in which computations are expressed) and as the type of thunks (viewed as a data structure). A key operation, *pay*, reflects one of Okasaki's main reasoning rules: by paying k now, one can decrease the debt of a thunk from n down to $n - k$. The library has no implementation: the type *Thunk* and its operations are axioms. Still, Danielsson proves (separately) that his type discipline is sound with respect to a cost-aware operational semantics. This work represents an important step. However, it is limited in several ways (§9). One important limitation is that it supports only purely functional programs and debit-based reasoning; there is no support for mutable state or credit-based reasoning.

Second, Mével et al. [2019], who extend Iris with time credits, claim to "present the first machine-checked reconstruction of Okasaki's debits in terms of time credits". At first sight, this appears to be true: they do indeed propose an abstract predicate *isThunk* t n ϕ, which they implement using time credits, monotonic ghost state, and an Iris "non-atomic invariant" [Mével et al. 2019, §7.4]. The parameter t is (the memory location of) the thunk. The parameter n is the debt associated with this thunk, that is, the number of time credits that remain to be paid before this thunk can be forced. The parameter ϕ is the postcondition of the thunk: forcing this thunk will produce a value v such that $\phi\ v$ holds. Mével et al. prove that this predicate enjoys a number of desirable reasoning rules [Mével et al. 2019, Fig. 6], including a "payment" rule that serves a similar purpose as Danielsson's *pay* operation: it is a ghost update that consumes k time credits and decreases a thunk's debt from n down to $n - k$.

Shortcomings of Mével et al.'s work. Mével et al.'s work also represents an important step. However, Mével et al. do not apply their reasoning rules to the analysis of some of Okasaki's data structures. Had they attempted to do so, they would have hit three shortcomings of their reasoning rules.

(1) These rules do not allow a thunk to force another thunk. This is visible in the fact that forcing a thunk requires a unique token $\text{\textit{ʄ}}$, yet, when a thunk is created, this token is *not* made available to the suspended computation. This severe limitation forbids crucial operations over *lazy lists*, also known as *streams*. Mével et al. indicate that they "have implemented a more flexible discipline" where thunks inhabit regions and there is one token per region, but do not provide details. This limitation arises out of the fundamental need to prevent the construction of *reentrant* thunks. We come back to this issue below.
(2) Mével et al.'s API lacks a key reasoning rule, namely the *consequence rule*. A simple form of this rule states that if ϕ entails ψ[2] then *isThunk* $t\ n\ \phi$ implies *isThunk* $t\ n\ \psi$. This simplified rule, which *weakens* a thunk's postcondition, is valid in Mével et al.'s system, up to a standard trick.[3] However, a more powerful form, which we name Thunk-Consequence, is needed and, based on Mével et al.'s definitions, cannot be justified. This rule can *strengthen* a thunk's postcondition, provided one *pays* for this. It states roughly that if $\$n_2 \twoheadrightarrow \forall v.\ \phi\ v \Rrightarrow \psi\ v$ holds[4] then *isThunk* $t\ n_1\ \phi$ implies *isThunk* $t\ (n_1 + n_2)\ \psi$. In other words, if the conversion of $\phi\ v$ into $\psi\ v$ consumes n_2 time credits, then changing a thunk's postcondition from ϕ to ψ is permitted, provided *the debt of this thunk is increased* from n_1 to $n_1 + n_2$. This ensures that, by the time this thunk is forced, enough credit has been raised to cover not only the cost of forcing this thunk (which is n_1) but also the cost of converting $\phi\ v$ into $\psi\ v$ (which is n_2).
(3) Mével et al.'s rules do not allow a thunk to pay for another thunk. This is visible in the fact that paying requires the token $\text{\textit{ʄ}}$. Mével et al. note that "it should be possible to remove this requirement", but do not do so. Yet, it is indeed crucial to establish a rule, named Thunk-Pay in this paper, which does not have this requirement. Combining Thunk-Consequence and Thunk-Pay lets us justify *deep payment*, a subtle concept whose need has been noted by Okasaki [1999, §6.3.2] and by Danielsson [2008, §11]. Deep payment consists in paying in advance for a thunk t' which possibly has not even been constructed yet, but whose construction has been scheduled. This is the case, for instance, if the thunk t that is expected to produce the thunk t' already exists: in such a situation, applying Thunk-Consequence to t allows applying Thunk-Pay to t', and this can be done before t is forced, so possibly before t' is constructed. More generally, deep payment serves to establish the reasoning rule Stream-Forward-Debt, which offers a powerful and intuitive way of understanding and distributing the debt carried by each thunk in a stream.

In this paper, we aim to address these shortcomings so as to fully bridge the gap.

[2] By "ϕ entails ψ", we mean that $\vdash \forall v.\ \phi\ v \twoheadrightarrow \psi\ v$ holds.
[3] The trick is to wrap *isThunk* in an existential quantification: $\mathit{isThunk}'\ t\ n\ \psi \triangleq \exists \phi.\ \mathit{isThunk}\ t\ n\ \phi * \Box(\forall v.\ \phi\ v \twoheadrightarrow \psi\ v)$.
[4] The assertion $\$n$ represents n time credits. The connective $\Rrightarrow$ represents a ghost update.

Ruling Out Reentrancy. Let us now point out a design constraint that must be obeyed. Okasaki's approach to the time complexity analysis of thunks fundamentally relies on the property that a suspended computation is executed at most once. Indeed, according to Okasaki, it is enough to pay *once* for the cost of a thunk; then, this thunk can be forced as many times as one wishes, at no extra cost.[5] However, this property does not automatically hold. In the presence of a fixed point combinator or of heap-allocated mutable state, it is possible to construct an ill-behaved *reentrant* thunk which, when forced, attempts to again force itself. For the complexity analysis to be sound, this must be forbidden: that is, the static reasoning rules of the program logic must forbid it.

We find that, to verify in Iris an implementation of thunks , one is essentially forced to use Iris's "non-atomic invariants" (§2), whose access is governed by unique tokens. In Mével et al.'s system, a single token $\natural$ controls every thunk in the universe. We propose a more flexible system (§5) where tokens $\natural_p^h$ are indexed with a "non-atomic pool" p and with an integer height h. We find this system flexible enough to implement a basic streams library (§6) and verify several data structures (§7, §8). It is important to see that as soon as one decides to statically forbid reentrant thunks, one is forced, in the design of a streams library, to statically forbid improductive streams. Thus, there is a connection (which we have not yet explored) between our work and the difficult and long-standing problem of ruling out improductive streams, which has received attention in the settings of type theory and of synchronous programming languages [Guatto 2018; Veltri and van der Weide 2019; Rusu and Nowak 2022].

Summary of Contributions and Road Map. We start from first principles. We write code in HeapLang, an untyped call-by-value λ-calculus equipped with dynamically allocated mutable state, whose definition is bundled with Iris.[6] To reason about the functional correctness and time complexity of this code, we use an off-the-shelf program logic, namely Iris$^\$$ [Mével et al. 2019]. Both HeapLang and Iris$^\$$ are formalized inside the Coq proof assistant. Thus, our trusted base includes just Coq and the operational semantics of HeapLang.

We work our way up through several layers of abstraction. After recalling some of the concepts of Iris (§2), we implement 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. The corresponding reasoning rules distill the essence of Okasaki's debit-based reasoning. Then (§4), we implement thunks in HeapLang and establish the desired reasoning rules about thunks, in correspondence with Okasaki's informal rules. These rules include the challenging rules Thunk-Consequence and Thunk-Pay and do not have the shortcomings discussed earlier. Our construction relies on piggy banks in two distinct places and potentially associates *an unbounded number of piggy banks* with a single thunk at runtime. In another layer (§5), we equip thunks with a notion of *height* that simplifies the way in which we rule out reentrancy. Then, on top of thunks, we implement streams (§6). We establish a number of reasoning rules about streams, including Stream-Forward-Debt, which distributes a debt over several thunks in a stream, by moving part of this debt 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. Finally, on top of streams, we implement and verify several purely functional data structures, including 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). Thus, we provide the first machine-checked verification of the time complexity of these data structures in the foundational setting of a separation logic with time credits. We believe that our work provides a nice example of the construction of high-level

[5]In fact, forcing a thunk whose debt has been fully paid off still has a constant cost.

[6]For readability, in the paper, we present our code in OCaml syntax. The code that we actually verify is HeapLang code.

abstractions on top of low-level logical concepts such as time credits, ghost state, and invariants. All of our results have been machine-checked, and our proofs are available [Anonymous 2023].

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

Even a basic introduction to Iris [Jung et al. 2018] might occupy more space than is available in this paper. In this section, 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 at least grasp the intuition behind the abstractions that we build. As an example, we need a reader who looks at the reasoning rule Thunk-Pay (Figure 5) to at least understand that it is a ghost update ($\Rrightarrow$) that consumes k time credits ($\$k$) and decreases the debt 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 $\mathit{Thunk}\ p\ \mathcal{F}\ t\ n\ R\ \phi$, 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.[7] 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 $\twoheadrightarrow$. (The non-separating conjunction $\wedge$ and implication $\Rightarrow$ are not used in this paper.) A magic wand $P \twoheadrightarrow Q$ 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*: $\square(P \twoheadrightarrow Q)$ is a magic wand that can be used as many times as one wishes.

Ghost state. Like physical state, *ghost state* is dynamically allocated. The law $\mathsf{True} \Rrightarrow \exists\gamma. \boxed{\gamma \mapsto m}$ (provided by Iris) allocates a fresh ghost cell, at address γ, whose initial content is m. We write $\gamma, \delta, \varphi, \pi$ 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 $\mathit{valid}\ (m_1 \cdot m_2)$ implies $\mathit{valid}\ m_1 \wedge \mathit{valid}\ m_2$. 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' $\mathit{valid}\ (m_1 \cdot m')$ implies $\mathit{valid}\ (m_2 \cdot m')$ then $\boxed{\gamma \mapsto m_1} \Rrightarrow \boxed{\gamma \mapsto m_2}$.

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

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. But how does one tell whether an invariant is currently open or closed, and how long may an invariant remain open?

One can imagine more than one way of answering these questions. Indeed Iris offers two flavors of invariants, which represent two incomparable points in the design space. An *atomic invariant* can be violated only during an atomic instruction and must be immediately restored. This may seem restrictive; on the upside, an atomic invariant can be accessed without presenting an affine token. A *non-atomic invariant* can remain violated for an unbounded time. This may seem flexible; on the downside, accessing a non-atomic invariant requires presenting an affine token.

An *atomic invariant* $\boxed{I}^A$ is labeled with a *namespace* A. This annotation is used to forbid reentrant access: two invariants can be simultaneously opened only if they are labeled with disjoint namespaces.[8] 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 A is in the set $\mathcal{E}$. One may omit this mask when it is $\top$.

A *non-atomic invariant* $\boxed{I}_p^N$ is labeled with a *pool* p and a namespace N. Opening such an invariant consumes an affine *token* $\lightning_p^{\mathcal{F}}$, where $\uparrow N \subseteq \mathcal{F}$ must hold. Closing the invariant causes this token to re-appear. Such a token can be split, thanks to the axiom $\lightning_p^{\mathcal{F}_1 \uplus \mathcal{F}_2} \equiv \lightning_p^{\mathcal{F}_1} * \lightning_p^{\mathcal{F}_2}$, so two non-atomic invariants can be simultaneously opened if they are annotated with disjoint namespaces. A fresh pool can be allocated at any time, together with a new token that governs it, thanks to the law $\mathsf{True} \Rrightarrow \exists p.\ \lightning_p^{\top}$.

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 1), but can otherwise be ignored.

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 use the sugared form $\{P\}\ e\ \mathsf{returns}\ (\exists\vec{x})\ v\ \{Q\}$ as a short-hand for $\{P\}\ e\ \{\lambda v'.\ \exists\vec{x}.\ \ulcorner v' = v \urcorner * Q\}$. This can be read as follows: "provided the initial state satisfies P,

[8] A *namespace* A, N is a string. A *mask* $\mathcal{E}, \mathcal{F}$ is a set of such strings. If N is a namespace, then its *upward closure* $\uparrow N$, a mask, is the set of all strings that admit N as a prefix. The *full mask* $\top$ is the set of all strings. We write $\mathcal{E}_1 \# \mathcal{E}_2$ when the masks $\mathcal{E}_1$ and $\mathcal{E}_2$ are disjoint. We say that two namespaces N_1 and N_2 are disjoint when $\uparrow N_1 \# \uparrow N_2$ holds.

then e does not crash, and if it terminates, then, for some $\vec{x}$, it returns the value v, and the final state satisfies Q". A triple is persistent: it allows the expression e to be executed as many times as one wishes. We occasionally need a *one-shot triple*, written $\mathbf{1}\ \{P\}\ e$ returns $(\exists\vec{x})\ v\ \{Q\}$. 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 a persistence modality $\Box$.

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, though, a specification provides a worst-case *amortized* time complexity guarantee. For instance, Thunk-Force (Figure 5) does *not* guarantee that *force t* runs in at most 11 steps. Although only 11 time credits are ostensibly visible in the precondition, the assertion $\mathit{Thunk}\ p\ \mathcal{F}\ t\ 0\ R\ \phi$, which is also part of the precondition, offers access to an invariant which possibly contains more credits. One should take this specification to mean that the amortized time complexity of *force* is 11.

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 intended 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 logical reasoning about credit that is accumulated via uncoordinated payments. This pattern involves the concepts of *true debt*, or "how much really is still missing", *apparent debt*, or "how much Sophie thinks is still missing", and the property that an apparent debt is always an over-approximation of the true debt. It seems desirable to isolate this pattern and to establish its logical soundness independently of any physical mechanism.

This is the aim of this section. Inside Iris, we develop the *piggy bank*, a ghost data structure, and we equip it with an API that supports the desired reasoning rules. There is no code: a piggy bank is a purely logical concept. The piggy bank is used at two distinct levels in our construction of thunks (§4.2.1, §4.2.2), so it seems worthwhile to make it 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 (either it is pending, or it is forced) and whose transition from the pending state to the forced state has a certain cost (that is, the transition requires a certain number of time credits).

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 $P : \mathbb{N} \to \mathit{iProp}$ 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 debt, that is, a nonnegative number of debits, n. Once a participant sees an apparent debt 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.

Although paying and breaking the bank are ghost operations, these two operations 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. While this computation is ongoing, the piggy bank is not in a valid state: neither $P\ nc$ nor Q holds. So, while the piggy bank is being forced, one must forbid any attempt to force it again. These considerations suggest that breaking the bank must be viewed as a sequence of two ghost updates: an update that causes a transition from the pending state to a transient state, and an update that causes a transition from this transient state to the forced state.

These considerations suggest that the implementation of the piggy bank should involve both an atomic invariant and a non-atomic invariant (§2). This, in turn, suggests that the predicate *PiggyBank* should be parameterized with a namespace A (serving as an index for the atomic invariant) and with a pool p and a namespace N (serving as indices for the non-atomic invariant). Together, the parameters A, p, N can be thought of as a "region" in which the piggy bank exists.

Our reasoning rules for piggy banks appear in Figure 1. The assertion $\mathit{PiggyBank}\ P\ Q\ A\ p\ N\ n$ means that there exists a piggy bank whose pending and forced states are described by P and Q, whose region is A, p, N, and whose number of debits is n. The parameter n is the most interesting one: in the rules of Figure 1, the five other parameters are fixed.

The rule PiggyBank-Persist states that a *PiggyBank* assertion is persistent. That is, a piggy bank can be shared. PiggyBank-Increase-Debt states that *PiggyBank* is covariant in the parameter n. In other words, it is safe to increase an apparent debt. This rule is intuitively sound because it preserves the fact that an apparent debt is an over-approximation of the true debt. 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 $P\ nc$, which is consumed. The user chooses *nc*, the number of credits that must be accumulated before the piggy bank can be broken. Thus, *nc* is the initial value of the true debt, and it is also the initial apparent debt. PiggyBank-Pay, also

PiggyBank-Persist

$$\text{persistent}(\mathit{PiggyBank}\ P\ Q\ A\ p\ N\ n)$$

PiggyBank-Increase-Debt

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

PiggyBank-Create

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

PiggyBank-Pay

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

PiggyBank-Break

$$\frac{\uparrow A \subseteq \mathcal{E} \qquad \uparrow N \subseteq \mathcal{E} \qquad \uparrow N \subseteq \mathcal{F}}{\mathit{PiggyBank}\ P\ Q\ A\ p\ N\ 0 * ⚡_p^{\mathcal{F}} \Rrightarrow_{\mathcal{E}} \exists nc. \left(\begin{array}{l} ((\triangleright P\ nc * \$nc) \vee \triangleright Q) * \\ (\triangleright Q \Rrightarrow_{\mathcal{E}} ⚡_p^{\mathcal{F}}) \end{array} \right)}$$

PiggyBank-Peek

$$\frac{\uparrow A \subseteq \mathcal{E} \qquad \uparrow N \subseteq \mathcal{E} \qquad \uparrow N \subseteq \mathcal{F}}{\begin{array}{l} \mathit{PiggyBank}\ P\ Q\ A\ p\ N\ n\ *\ ⚡_p^{\mathcal{F}} \mathrel{-\!\!*} \\ (\forall nc,\ \triangleright P\ nc \mathrel{-\!\!*} \triangleright \mathsf{False}) \mathrel{-\!\!*} \\ (\triangleright Q \mathrel{-\!\!*} \triangleright \square Q') \Rrightarrow_{\mathcal{E}} \\ \mathit{PiggyBank}\ P\ Q\ A\ p\ N\ 0\ *\ \triangleright \square Q'\ *\ ⚡_p^{\mathcal{F}} \end{array}}$$

Fig. 1. Piggy Banks: Reasoning Rules

a ghost update, allows contributing k time credits to a piggy bank. The apparent debt decreases from n to $n-k$. (This is subtraction in the natural numbers, so $n-k \geq 0$ holds.)

We now come to the most complex rule, PiggyBank-Break. This rule allows the user to break a piggy bank whose apparent debt is 0. This is intuitively permitted because, if the apparent debt is 0, then the true debt must be 0 as well. As explained earlier, this rule is expressed via two nested ghost updates. The outermost update initiates the process of breaking the bank. It consumes the affine token $⚡_p^{\mathcal{F}}$, where $\uparrow N \subseteq \mathcal{F}$ 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 debt is zero), or the piggy bank is already in its forced state Q;
(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 $⚡_p^{\mathcal{F}}$ 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. We have explained earlier why this matters (§1, item 3).

In the interest of space, we explain PiggyBank-Peek only briefly. This rule 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 stateand the piggy bank's apparent debt can be set to 0. This rule is later exploited to establish the rule Thunk-Force-Forced.

$$
\begin{array}{l}
\mathit{PiggyBank}\ P\ Q\ A\ p\ N\ n \triangleq \\
\quad \exists\, \varphi, \pi, \mathit{nc}. \\
\quad\quad \boxed{\exists \mathit{forced}.\ \boxed{\varphi \mapsto \bullet\, \mathit{forced}} \ast \text{if}\ \neg\mathit{forced}\ \text{then}\ P\ \mathit{nc}\ \text{else}\ Q}^{N}_{p} \ast \\
\quad\quad \boxed{\exists \mathit{forced}, \mathit{ac}.\ \boxed{\varphi \mapsto \circ\, \mathit{forced}} \ast \boxed{\pi \mapsto \bullet\, \mathit{ac}} \ast \text{if}\ \neg\mathit{forced}\ \text{then}\ \$\mathit{ac}\ \text{else}\ \lceil \mathit{nc} \leq \mathit{ac} \rceil}^{A} \ast \\
\quad\quad \boxed{\pi \mapsto \circ\, (\mathit{nc} - n)}
\end{array}
$$

Fig. 2. Piggy Banks: Internal Definition

3.2 Piggy Banks: Construction

The definition of the predicate *PiggyBank* appears in Figure 2. It may be of interest mainly to readers who are familiar with Iris; other readers may wish to skip this part. Its most interesting aspect is that it involves both an atomic invariant and a non-atomic invariant. This follows from the fact that we want paying to be atomic, a single ghost update, whereas breaking the bank must be non-atomic, a sequence of two ghost updates. The atomic invariant holds “always”: it can be violated and must be restored during an atomic instruction. The non-atomic invariant holds “except while the piggy bank is being broken”: thus, it can remain violated over a long period of time.

The two invariants are not independent of one another: they must agree on the question of whether an attempt to break the piggy bank has begun already. We impose this agreement by letting the two invariants share a ghost cell φ whose content, a Boolean value *forced*, reflects this information. The invariants cannot directly share the variable *forced*: that would make *forced* itself invariant. Instead, they share the address φ of a ghost cell whose content can change over time.

For this purpose, we use the `excl_auth` camera from the Iris library. This camera offers a way of expressing the idea that a resource has exactly two owners, whose roles are symmetric. Thus, two assertions control the ghost cell φ: the assertion $\boxed{\varphi \mapsto \bullet\, \mathit{forced}_1}$ represents the view of one owner; the assertion $\boxed{\varphi \mapsto \circ\, \mathit{forced}_2}$ is the view of the other owner. These assertions satisfy the agreement law $\boxed{\varphi \mapsto \bullet\, \mathit{forced}_1} \ast \boxed{\varphi \mapsto \circ\, \mathit{forced}_2} \vdash \lceil \mathit{forced}_1 = \mathit{forced}_2 \rceil$, which states that when the owners confront their views, they must find that they agree. They also satisfy the update law $\boxed{\varphi \mapsto \bullet\, \mathit{forced}} \ast \boxed{\varphi \mapsto \circ\, \mathit{forced}} \Rrightarrow \boxed{\varphi \mapsto \bullet\, \mathit{forced}'} \ast \boxed{\varphi \mapsto \circ\, \mathit{forced}'}$, which states that when the owners combine their powers, they are able to change the content of the ghost cell.

At this point, we can explain the first invariant in the definition of *PiggyBank* (Figure 2). It is the non-atomic invariant. It states that if the piggy bank has never been broken then it must be in the pending state $P\ \mathit{nc}$, otherwise it must be in the forced state Q. While the piggy bank is being broken, this invariant does not hold: the piggy bank is then in neither state.

Another ghost cell, π, appears in the following two lines. 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 $\boxed{\pi \mapsto \bullet\, \mathit{ac}}$ represents the authority to read the cell and to increase its value. A persistent assertion $\boxed{\pi \mapsto \circ\, k}$ is a guarantee that the value of the cell is at least k. These assertions satisfy the agreement law $\boxed{\pi \mapsto \bullet\, \mathit{ac}} \ast \boxed{\pi \mapsto \circ\, k} \vdash \lceil k \leq \mathit{ac} \rceil$ and the update law $\boxed{\pi \mapsto \bullet\, \mathit{ac}} \Rrightarrow \boxed{\pi \mapsto \bullet\, (\mathit{ac} + k)} \ast \boxed{\pi \mapsto \circ\, (\mathit{ac} + k)}$.

We can now explain the second invariant in Figure 2. It is the atomic invariant. It holds the authoritative view $\boxed{\pi \mapsto \bullet\, \mathit{ac}}$, which guarantees that ac (for “available credit”) is the total amount of payment received so far. Furthermore, if the bank has never been broken yet, then this invariant guarantees that ac time credits are available. Otherwise, no credit is available: then, the invariant guarantees $\mathit{nc} \leq \mathit{ac}$, which means that the available credit has exceeded the necessary credit.

```
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                                 (* look up memoized value *)
```

Fig. 3. Thunks: OCaml Code

Thunk-Create

$$\frac{\uparrow N \subseteq \mathcal{F}}{\begin{array}{c}\{\$3 \; * \; \mathit{isAction}\; f \; n \; R \; \phi\} \\ \mathit{create}\; f \\ \text{returns } (\exists t)\; t\; \{\mathit{Thunk}\; p \; \mathcal{F} \; t \; n \; R \; \phi\}\end{array}}$$

Thunk-Consequence

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

Fig. 4. Thunks: Creation Rule and Consequence Rule

Thunk-Persist

$$\mathrm{persistent}(\mathit{Thunk}\; p \; \mathcal{F} \; t \; n \; R \; \phi)$$

Thunk-Increase-Debt

$$\frac{n_1 \leq n_2}{\begin{array}{l}\mathit{Thunk}\; p \; \mathcal{F} \; t \; n_1 \; R \; \phi \; -\!\!* \\ \mathit{Thunk}\; p \; \mathcal{F} \; t \; n_2 \; R \; \phi\end{array}}$$

Thunk-Pay

$$\frac{\uparrow \mathit{ThunkPayment} \subseteq \mathcal{E}}{\begin{array}{l}\mathit{Thunk}\; p \; \mathcal{F} \; t \; n \; R \; \phi * \$k \; \Rrightarrow_{\mathcal{E}} \\ \mathit{Thunk}\; p \; \mathcal{F} \; t \; (n - k) \; R \; \phi\end{array}}$$

Thunk-Force

$$\begin{array}{c}\left\{\begin{array}{l}\mathit{Thunk}\; p \; \mathcal{F} \; t \; 0 \; R \; \phi \; * \\ \$11 * \lightning^{\mathcal{F}}_{p} * R\end{array}\right\} \\ \mathit{force}\; t \\ \text{returns } (\exists v)\; v\; \{\mathit{ThunkVal}\; t \; v * \Box \; \phi \; v * \lightning^{\mathcal{F}}_{p} * R\}\end{array}$$

Thunk-Force-Forced

$$\begin{array}{c}\left\{\begin{array}{l}\mathit{Thunk}\; p \; \mathcal{F} \; t \; n \; R \; \phi * \mathit{ThunkVal}\; t \; v \; * \\ \$11 * \lightning^{\mathcal{F}}_{p}\end{array}\right\} \\ \mathit{force}\; t \\ \text{returns } v\; \{\lightning^{\mathcal{F}}_{p}\}\end{array}$$

Fig. 5. Thunks: Common Reasoning Rules

On the last line of Figure 2, we find a persistent witness $\boxed{\pi \mapsto \circ\,(nc - n)}$. By opening the atomic invariant and by exploiting the agreement law, this witness allows obtaining the inequality $nc - n \leq ac$. If the apparent debt 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 1.*

4 THUNKS

Our implementation of thunks appears in Figure 3. 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 $\mathit{Thunk}\; p \; \mathcal{F} \; t \; n \; R \; \phi$ describes a thunk at location t in memory. The pool p and the mask $\mathcal{F}$ play the same role as in the previous section (§3): in short, they determine which

ThunkVal-Persist	ThunkVal-Timeless	ThunkVal-Confront
$\mathrm{persistent}(\mathit{ThunkVal}\ t\ v)$	$\mathrm{timeless}(\mathit{ThunkVal}\ t\ v)$	$\mathit{ThunkVal}\ t\ v_1 \ast \mathit{ThunkVal}\ t\ v_2 \vdash \lceil v_1 = v_2 \rceil$

Fig. 6. Forced-Thunk Witnesses: Reasoning Rules

token $\text{\textit{\char"21AF}}_p^{\mathcal{F}}$ is required to force this thunk. The parameter n also plays the same role as in the previous section: it is an apparent debt 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 $\phi : \mathit{Val} \rightarrow \mathit{iProp}$ determines the postcondition of the thunk: it is $\square\,\phi$. That is, once this thunk is forced and produces a value v, the assertion $\square\ \phi\ v$ can be expected to hold.[9]

The full set of reasoning rules for thunks appears in Figures 4, 5, and 6. The rules are split in three groups for reasons that will be apparent in the next subsection (§4.2).

Thunk-Create (Figure 4) describes the creation of a new thunk via the function call *create* f. The behavior of f is specified by the premise *isAction* $f\ n\ R\ \phi$, which denotes the one-shot triple $\mathbf{1}\ \{R \ast \$n\}\ f()\ \text{returns}\ (\exists v)\ v\ \{R \ast \square\ \phi\ v\}$. 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 $\square\ \phi\ v$ holds. The resource R is required, but not consumed, by this call. Under this hypothesis on f, Thunk-Create states that *create* f costs \$3 and returns a thunk whose apparent debt, resource requirement, and postcondition are described by n, R, ϕ. The pool p and mask $\mathcal{F}$ are chosen by the user. These parameters determine which token $\text{\textit{\char"21AF}}_p^{\mathcal{F}}$ will be required when forcing this thunk. A systematic way of instantiating these parameters is discussed in the next section (§5).

The rules Thunk-Persist, Thunk-Increase-Debt, and Thunk-Pay (Figure 5) are analogous to PiggyBank-Persist, PiggyBank-Increase-Debt, and PiggyBank-Pay. They state that thunks can be shared, that it is permitted to over-approximate an apparent debt, and that an apparent debt can be reduced by paying. (In Thunk-Pay, one can see that the parameter A of piggy banks has been instantiated with a fixed namespace *ThunkPayment*.)

Thunk-Force allows forcing a thunk whose apparent debt is zero. This consumes \$11, 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 $\text{\textit{\char"21AF}}_p^{\mathcal{F}}$ and the resource R are required and preserved. The token $\text{\textit{\char"21AF}}_p^{\mathcal{F}}$ 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 $\text{\textit{\char"21AF}}_p^{\mathcal{F}}$ 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 *ThunkVal* $t\ v$ and $\square\ \phi\ v$ hold. The assertion *ThunkVal* $t\ v$ is a witness that the value of the thunk t has been forever decided and that it is v. The assertion $\square\ \phi\ v$ 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 6. 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* $t\ v$;

[9] Instead of requiring the postcondition to be in the syntactic form $\square\,\phi$, 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.

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.

THUNK-FORCE-FORCED (Figure 5) allows forcing a thunk that has been forced already. This hypothesis is reflected by the appearance of the assertion *ThunkVal* t v in the precondition. There is no requirement that the thunk's apparent debt be zero. The value that is obtained by forcing this thunk must be v, the value that was predicted by the witness *ThunkVal* t v. Like THUNK-FORCE, this rule consumes \$11 and requires and preserves $\lightning_p^{\mathcal{F}}$. Unlike THUNK-FORCE, it does not require the resource R. Furthermore, perhaps surprisingly, it does not guarantee that $\Box\ \phi\ v$ holds. In the presence of THUNK-CONSEQUENCE, this cannot be guaranteed.

The rule THUNK-CONSEQUENCE (Figure 4) allows changing the postcondition of a thunk from $\Box\,\phi$ to $\Box\,\psi$. As explained in the introduction (§1), 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 debt of the thunk from n_1 to $n_1 + n_2$. In return, the update[10] 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 4, 5, and 6 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 a single non-atomic invariant, it is built on top of our "piggy bank" abstraction, which combines an atomic invariant and a non-atomic invariant. 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* $(\lambda().\,$*force* $t)$. Although applying the consequence rule is a ghost operation and therefore does not create a new thunk at runtime, this analogy suggests that *applying the consequence rule should allocate a new piggy bank*. Guided by this key idea, we propose a construction that supports *one* application of the consequence rule. Assuming that we have some predicate *Thunk* that satisfies the rules of Figure 5, we construct a new predicate *ProxyThunk*, which also satisfies the rules of Figure 5, and we establish a version of the consequence rule that expects a *Thunk* and produces a *ProxyThunk*.

(3) It is then a relatively simple exercise to prove 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 construction in three stages 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.

4.2.1 Basic Thunks. The definition of the predicate *BasicThunk* appears in Figure 7. 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 $\mathrm{Ex}(()) +_{\lightning} \mathrm{AG}(\mathit{Val})$, also known as the "one-shot" camera [Jung et al.

[10] 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.

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

Fig. 7. Basic Thunks and Forced-Thunk Witnesses: Definitions

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 $\boxed{\delta \mapsto v_1} * \boxed{\delta \mapsto v_2} \vdash \lceil v_1 = v_2 \rceil$, and the disagreement law $\boxed{\delta \mapsto ?} * \boxed{\delta \mapsto v} \vdash$ 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 7. 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 debt n of the piggy bank is also the apparent debt 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 4, where Thunk is replaced with BasicThunk. Furthermore, it satisfies all of the rules in Figure 5, where the same replacement is made. Finally, the predicate ThunkVal satisfies the rules in Figure 6.*

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 debt 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 debt of the new piggy bank is $n_1 + n_2$. Once the new piggy bank has reached its aim, breaking it produces $n_1 + n_2$ 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 5. 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 8. The "creation rule" for proxy thunks, also shown in

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

Proxy-Create

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

Fig. 8. Proxy Thunks: Definition and Creation Rule

Figure 8, 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 debt is n_1, and the proxy thunk's piggy bank. The apparent debt n of the piggy bank is the apparent debt 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 $\mathit{ThunkVal}\ t\ v$ together with the postcondition $\Box\ \psi\ v$.

The side condition $\mathcal{F}_1 \uplus \uparrow N \subseteq \mathcal{F}$ guarantees that out of the token $\lightning_p^{\mathcal{F}}$, which the user supplies when forcing the proxy thunk, we can extract the tokens $\lightning_p^{\mathcal{F}_1} * \lightning_p^{\uparrow N}$, 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 8. Furthermore, it satisfies all of the rules in Figure 5, 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*, if this predicate is persistent and satisfies the common thunk API in Figure 5. Both *BasicThunk* and *ProxyThunk* meet these requirements. Thus, the construction can be iterated. We do so in Figure 9. The definition is conceptually straightforward. An existential quantification is used to construct the greatest predicate *Thunk* that satisfies certain properties. Two technical formulae involving masks record that (1) we have an infinite family of pairwise disjoint masks, namely $\uparrow(N \cdot d)$, 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.

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

Fig. 9. Thunks: Definition

Theorem 4.3. *The predicate Thunk satisfies all of the rules in Figures 4 and 5.*

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.

HThunk-Create
$$\{\$3 \ast \mathit{isAction}\ f\ n\ (\lightning_p^h)\ \phi\}\ \mathit{create}\ f\ \text{returns}\ (\exists t)\ t\ \{\mathit{HThunk}\ p\ h\ t\ n\ \phi\}$$

HThunk-Consequence
$$\mathit{HThunk}\ p\ h\ t\ n_1\ \phi \twoheadrightarrow (\forall v.\ \$n_2 \twoheadrightarrow \Box\ \phi\ v \Rrightarrow_\top \Box\ \psi\ v) \Rrightarrow_\mathcal{E} \mathit{HThunk}\ p\ h\ t\ (n_1 + n_2)\ \psi$$

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

HThunk-Inc-Height-Debt
$$\frac{h_1 \leq h_2 \qquad n_1 \leq n_2}{\mathit{HThunk}\ p\ h_1\ t\ n_1\ \phi \twoheadrightarrow \mathit{HThunk}\ p\ h_2\ t\ n_2\ \phi}$$

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

HThunk-Force
$$\left\{\begin{array}{l}\mathit{HThunk}\ p\ h\ t\ 0\ \phi \ast \\ \$11 \ast \lightning_p^{h'} \ast \lceil h < h' \rceil\end{array}\right\}\ \mathit{force}\ t\ \text{returns}\ (\exists v)\ v\ \{\Box\ \phi\ v \ast \mathit{ThunkVal}\ t\ v \ast \lightning_p^{h'}\}$$

HThunk-Force-Forced
$$\left\{\begin{array}{l}\mathit{HThunk}\ p\ h\ t\ n\ \phi \ast \mathit{ThunkVal}\ t\ v \ast \\ \$11 \ast \lightning_p^{h'} \ast \lceil h < h' \rceil\end{array}\right\}\ \mathit{force}\ t\ \text{returns}\ v\ \{\lightning_p^{h'}\}$$

Fig. 10. Height-Indexed Thunks: Reasoning Rules

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 $\lightning_p^{\mathcal{F}}$, 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 $\lightning_p^{\mathcal{F}'}$ 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. We define a new predicate $\mathit{HThunk}\ p\ h\ t\ n\ \phi$ 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, that is, at heights less than h. 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.

For the sake of brevity, the definition of the predicate *HThunk* is omitted. Its reasoning rules appear in Figure 10. The affine token $\lightning_p^h$ allows forcing thunks whose height is less than h: this is visible in HThunk-Force and HThunk-Force-Forced. When a thunk is created at height h, the token that is passed to the suspended computation is $\lightning_p^h$: this is visible in HThunk-Create. Thus, the new thunk can force thunks at lower heights only. We remark that a height is not a creation time. Indeed, a thunk at height 0 can be created after, and even 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-Debt. In a token, h can be instantiated with ∞. The token $\lightning_p^\infty$ can force thunks of arbitrary height. It appears in the API of the banker's queue (Figure 18).

```
type 'a stream = 'a cell thunk
 and 'a cell   =  Nil | Cons of 'a * 'a stream

let nil () : 'a stream =
  create @@ fun () -> Nil

let uncons (s : 'a stream) : 'a * 'a stream =
  match force s with
  | Nil          -> assert false (* dead branch *)
  | Cons (x, s) -> x, s

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

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

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

Fig. 11. Streams: OCaml Code

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 11. 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 $Cons(x, s')$, 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 *Stream* $p\ h\ s\ ds\ xs$, 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 11.

6.1 The predicate *Stream*

The parameters p and h of the predicate *Stream* play the same role as the parameters p and h of the predicate *HThunk*. They are a non-atomic pool p and an integer height h, and they indicate that the token $\lightning_p^h$ 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 $Cons(x, \_)$. 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

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

Fig. 12. Streams: Definition

Stream-Persist

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

Stream-Increase-Height

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

Stream-Force

$$\left\{ \begin{array}{l} \mathit{Stream}\ p\ h\ s\ (0 :: \mathit{ds})\ \mathit{xs}\ * \\ \$11 * \lightning_p^{h'} * \lceil h < h' \rceil \end{array} \right\}\ \mathit{force}\ s\ \text{returns } (\exists c)\ c \left\{ \begin{array}{l} \mathit{StreamCell}\ p\ h\ c\ \mathit{ds}\ \mathit{xs}\ * \\ \mathit{ThunkVal}\ s\ c * \lightning_p^{h'} \end{array} \right\}$$

Stream-Force-Forced

$$\left\{ \begin{array}{l} \mathit{Stream}\ p\ h\ s\ (d :: \mathit{ds})\ \mathit{xs} * \mathit{ThunkVal}\ s\ c\ * \\ \$11 * \lightning_p^{h'} * \lceil h < h' \rceil \end{array} \right\}\ \mathit{force}\ s\ \text{returns } c \left\{ \lightning_p^{h'} \right\}$$

Stream-Forward-Debt

$$\frac{\uparrow \mathit{ThunkPayment} \subseteq \mathcal{E} \qquad (m)\ \mathit{ds}_1 \leq \mathit{ds}_2\ (n)}{\mathit{Stream}\ p\ h\ s\ \mathit{ds}_1\ \mathit{xs} * \$m \Rrightarrow_{\mathcal{E}} \mathit{Stream}\ p\ h\ s\ \mathit{ds}_2\ \mathit{xs}}$$

Stream-Create

$$\frac{}{\{\$5 * \mathit{isCellAction}\ p\ h\ d\ e\ \mathit{ds}\ \mathit{xs}\}\ \mathit{create}\ (\lambda().e)\ \text{returns } (\exists s)\ s\ \{\mathit{Stream}\ p\ h\ s\ (d :: \mathit{ds})\ \mathit{xs}\}}$$

Fig. 13. Streams: Reasoning Rules

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. Because a stream of n elements involves $n + 1$ thunks, in an assertion $\mathit{Stream}\ p\ h\ s\ \mathit{ds}\ \mathit{xs}$, one can informally[11] expect $|\mathit{ds}| = |\mathit{xs}| + 1$. In $\mathit{StreamCell}\ p\ h\ c\ \mathit{ds}\ \mathit{xs}$, one can informally expect $|\mathit{ds}| = |\mathit{xs}|$.

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 10), so we do not explain them again. Stream-Force requires the head thunk to have zero debits. Stream-Create relies on the auxiliary

[11] Technically, $\mathit{Stream}\ p\ h\ s\ \mathit{ds}\ \mathit{xs} \vdash |\mathit{ds}| = |\mathit{xs}| + 1$ does *not* hold. A straightforward 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 $\mathit{Stream}\ p\ h\ s\ \mathit{ds}\ \mathit{xs} \vdash |\mathit{ds}| > 0$ does hold and has been sufficient for our purposes.

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

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

predicate *isCellAction*[12] in the same way that Thunk-Create and HThunk-Create rely on the auxiliary predicate *isAction*.

The most notable rule in Figure 13 is Stream-Forward-Debt. This rule allows managing a stream's debt 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 debt 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-Debt relies on the consequence rule HThunk-Consequence.

Stream-Forward-Debt involves the *debit subsumption* judgement $(m)\ ds_1 \leq ds_2\ (n)$, 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.

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-Debt. 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.

To a reader who has difficulty understanding this definition, we may propose an alternative characterization of the subsumption judgement, which helps see why Stream-Forward-Debt is sound. Intuitively, for this rule to be sound, it must be the case that, by applying this rule, one

[12] We write *isCellAction* $p\ h\ d\ e\ ds\ xs$ for $\mathbf{1}\ \{\lightning_p^h * \$d\}\ e$ returns $(\exists c)\ c\ \{\mathit{StreamCell}\ p\ h\ c\ ds\ xs * \lightning_p^h\}$. 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 *StreamCell* $p\ h\ c\ ds\ xs$ holds. The resource $\lightning_p^h$ may be used and must be preserved.

Stream-Revl

$$\left\{\begin{array}{l} \mathit{List}\ l\ \mathit{xs}\ \ * \\ \$13\ \ *\ \ \lceil n = |\mathit{xs}| \rceil \end{array}\right\}$$
$$\mathit{revl}\ l$$
$$\text{returns } (\exists s)\ s \left\{\begin{array}{l} \mathit{Stream}\ p\ h\ s \\ \quad (19n :: 0^n)\ (\mathit{rev}\ \mathit{xs}) \end{array}\right\}$$

Stream-Append

$$\left\{\begin{array}{l} \mathit{Stream}\ p\ h\ s_1\ \mathit{ds}_1\ \mathit{xs}_1\ \ * \\ \mathit{Stream}\ p\ h\ s_2\ \mathit{ds}_2\ \mathit{xs}_2\ \ *\ \ \$8 \end{array}\right\}$$
$$\mathit{append}\ s_1\ s_2$$
$$\text{returns } (\exists s)\ s \left\{\begin{array}{l} \mathit{Stream}\ p\ (h+1)\ s \\ \quad (\mathit{ds}_1 \bowtie \mathit{ds}_2)\ (\mathit{xs}_1 \mathbin{++} \mathit{xs}_2) \end{array}\right\}$$

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

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 $\forall i.\ \sum(\mathit{take}\ i\ ds_1) \leq m + \sum(\mathit{take}\ i\ ds_2)$.*

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 debt of the first n thunks justifies a decrease of n in the debt 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).

6.3 Operations on Streams

There remains to present the specifications of the operations on streams whose code appears in Figure 11. 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 $\mathit{rev}\ xs$. (The pure assertion $\mathit{List}\ l\ xs$ 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 $19n$, while every other thunk carries debit zero. The first thunk is expensive because, when it is forced, *revl_append* l *Nil* is invoked (Line 18 in Figure 11). 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 just 13 time credits.

Stream-Append states that *append* s_1 s_2 has constant complexity: it consumes just 8 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 $xs_1 \mathbin{++} xs_2$. 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 $ds_1 \bowtie ds_2$.

```
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

The *debit join* operation $\bowtie$ is defined as follows, where $A \triangleq 30$ and $B \triangleq 11$:

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

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 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 B to this fused element. This specification reflects the fact that (1) the overall cost of a stream concatenation operation is $A(n_1 + 1) + B$, 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 A; the next thunk bears a cost of $A + B$; 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.

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, we rely on the streams 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-Debt. 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).

Banker-Persistent
$\text{persistent}(\mathit{BQueue}\ p\ q\ \mathit{xs})$

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

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

Banker-Extract
$$\begin{array}{c}\{\$165\ *\ \mathit{BQueue}\ p\ q\ (x :: \mathit{xs})\ *\ \lightning_p^\infty\}\\ \mathit{extract}\ q\\ \text{returns}\ (\exists q')\ (x, q')\ \{\mathit{BQueue}\ p\ q'\ \mathit{xs}\ *\ \lightning_p^\infty\}\end{array}$$

Fig. 18. Banker's Queue: Public Interface

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 $\mathit{BQueue}\ p\ q\ \mathit{xs}$ means that q is a queue, allocated in pool p, 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 ostensibly requires a constant amount of time credits in its precondition. The specification of *extract* requires and preserves the token $\lightning_p^\infty$, which allows forcing thunks of arbitrary height (§5). As noted earlier (§2), a fresh pool can be allocated at any time, together with a new token for it, thanks to the law $\mathsf{True} \Rrightarrow \exists p.\ \lightning_p^\infty$, which is also part of the public specification.

To a reader who wonders why *BQueue* must be parameterized with a pool p, and why *extract* must require a token, we point out that this is actually necessary. 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 $\mathit{fs} \mathbin{++} \mathit{rev}\ \mathit{rs}$. The inequality $|\mathit{rs}| \leq |\mathit{fs}|$ 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 linear debt associated with this thunk can be distributed onto the linear number of thunks that appear in front of it. This translates to a constant amount of extra debt per thunk, which is acceptable.

$$
\begin{array}{rcl}
K & \triangleq & 60 \\
\mathit{bqueueDebits}\ \mathit{nf}\ \mathit{nr} & \triangleq & K^{\mathit{nf}-\mathit{nr}} \mathbin{++} 0^{\min(\mathit{nf},\mathit{nr})+1} \\
\mathit{BQueueRaw}\ p\ q\ \mathit{fs}\ \mathit{rs} & \triangleq & \exists s,\ h,\ l. \begin{cases} \lceil q = (|\mathit{fs}|, s, |\mathit{rs}|, l) \rceil \ * \\ \mathit{Stream}\ p\ h\ s\ (\mathit{bqueueDebits}\ |\mathit{fs}|\ |\mathit{rs}|)\ \mathit{fs}\ *\ \mathit{List}\ l\ \mathit{rs} \end{cases} \\
\mathit{BQueue}\ p\ q\ \mathit{xs} & \triangleq & \exists \mathit{fs},\ \mathit{rs}.\ \mathit{BQueueRaw}\ p\ q\ \mathit{fs}\ \mathit{rs}\ *\ \lceil \mathit{xs} = \mathit{fs} \mathbin{++} \mathit{rev}\ \mathit{rs} \wedge |\mathit{rs}| \le |\mathit{fs}| \rceil
\end{array}
$$

Fig. 19. Banker's Queue: Definitions

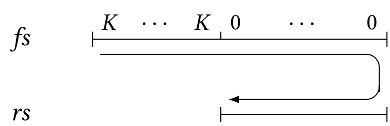

Fig. 20. A balanced banker's queue. The front stream is annotated with debits. The arrow indicates the logical order of elements.

Banker-Check

$$
\begin{array}{c}
\{\$48 * \mathit{BQueueRaw}\ p\ q\ \mathit{fs}\ \mathit{rs} * \lceil |\mathit{rs}| \le |\mathit{fs}| + 1 \rceil\} \\
\mathit{check}\ q \\
\text{returns } (\exists q')\ q'\ \{\mathit{BQueue}\ p\ q'\ (\mathit{fs} \mathbin{++} \mathit{rev}\ \mathit{rs})\}
\end{array}
$$

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

The predicate BQueue. Figure 19 presents the definition of $\mathit{BQueue}\ p\ q\ \mathit{xs}$. It is constructed by combining the assertions $\mathit{xs} = \mathit{fs} \mathbin{++} \mathit{rev}\ \mathit{rs}$ and $|\mathit{rs}| \le |\mathit{fs}|$, which we have explained already, with a lower-level assertion, $\mathit{BQueueRaw}\ p\ q\ \mathit{fs}\ \mathit{rs}$. This assertion states that q is a 4-tuple, requires the two length fields to contain the integer values $|\mathit{fs}|$ and $|\mathit{rs}|$, and uses the predicates *Stream* and *List* to describe the front stream and rear list. The most noteworthy aspect is that the sequence of debits associated with the front stream is fully determined: it is $\mathit{bqueueDebits}\ |\mathit{fs}|\ |\mathit{rs}|$. The definition of *bqueueDebits* states that the first $\mathit{nf} - \mathit{nr}$ thunks in the front stream carry debit K, 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 debt is nonzero. This is illustrated in Figure 22 (left). This is done by applying Stream-Forward-Debt. This requires proving the subsumption judgement $(K)\ K^n \mathbin{++} K :: 0^m \le K^n \mathbin{++} 0 :: 0^m\ (0)$, 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-Debt and proving the subsumption judgement $(K)\ K :: K^n \mathbin{++} 0^m \le 0 :: K^n \mathbin{++} 0^m\ (0)$, which follows from Sub-Cons and Sub-Refl. This is a trivial instance of Stream-Forward-Debt, 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 $|\mathit{rs}| \le |\mathit{fs}|$. 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 $|\mathit{fs}|$, so that we

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 debt must be paid off.

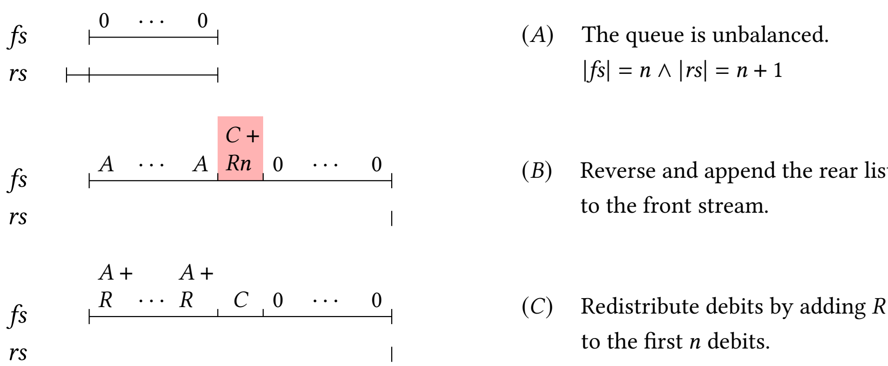

Fig. 23. Rebalancing. In red: the costly thunk whose debt must be distributed among the front thunks.

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 $0^{n+1} \bowtie 19(n+1) :: 0^{n+1}$. 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: $A^n \mathbin{++} (A + B + 19(n+1)) :: 0^{n+1}$. By posing $C \triangleq A + B + 19$ and $R \triangleq 19$, this sequence of debits is written $A^n \mathbin{++} (C + Rn) :: 0^{n+1}$. It is depicted in step (B) of Figure 23.
(2) Then, the key step of the proof is to *distribute* the expensive debit $C + Rn$ onto earlier debits. We increase each of the first n debits by R, so as be allowed to reduce the expensive debit from $C + Rn$ down to C. The result is illustrated in step (C) of Figure 23. This redistribution of debt is permitted by Stream-Forward-Debt provided we establish the subsumption judgement $(0)\ A^n \mathbin{++} (C + Rn) :: 0^{n+1} \le (A + R)^n \mathbin{++} C :: 0^{n+1}\ (0)$. This judgement follows from Sub-Append, Sub-Repeat, Sub-Cons, and Sub-Refl.
(3) Because we have chosen K as an upper bound for $A + R$ and C, we can now over-approximate every debit by K, except for the very last debit, which must remain zero. We exploit the subsumption judgement $(0)\ (A + R)^n \mathbin{++} C :: 0^{n+1} \le K^{2n+1} \mathbin{++} [0]\ (0)$. The debit sequence $K^{2n+1} \mathbin{++} [0]$ 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. It involves front and rear lists of elements and rebalances them when necessary. Its analysis is somewhat more elementary than the banker's queue's, because a physicist's queue involves a single thunk instead of a stream. The rule Thunk-Consequence is not needed. 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. Their implementation relies on 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 $n \le time(\tau)$ 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. Unfortunately, their work exhibits several shortcomings, which prevents them from justifying deep payment (§1).

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

For the first time, we have fully reconstructed Okasaki's debit-based reasoning rules, as well as Okasaki's analyses of the banker's queue, the physicist's queue, and implicit queues, in the formal and foundational setting of separation logic with time credits. Our proofs are machine-checked [Anonymous 2023]. 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 and make unusual use of atomic and non-atomic invariants in combination. Our modular construction of thunks on top of piggy banks, in several steps, is original. We hope that the reader finds it elegant.

We have used concrete constants everywhere: e.g., *force* costs 11. In the future, following Guéneau and co-authors [Guéneau et al. 2018; Guéneau 2019; Guéneau et al. 2019], it would be desirable to use O notation in specifications. Another aspect 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.

REFERENCES

Anonymous. 2023. Thunks and Debits in Separation Logic with Time Credits: Coq Formalization. Anonymized supplementary material, submitted together with the paper.

Robert Atkey. 2011. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science* 7, 2:17 (2011).

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).

Nils Anders Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Principles of Programming Languages (POPL)*.

Adrien Guatto. 2018. A Generalized Modality for Recursion. In *Logic in Computer Science (LICS)*. 482–491.

Armaël Guéneau. 2019. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs*. Ph.D. Dissertation. Université de Paris.

Armaël Guéneau, Arthur Charguéraud, and François Pottier. 2018. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 10801)*. Springer, 533–560.

Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. 2019. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In *Interactive Theorem Proving (ITP) (Leibniz International Proceedings in Informatics, Vol. 141)*. 18:1–18:20.

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.

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.

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.

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.

Jan Hoffmann, Michael Marmar, and Zhong Shao. 2013. Quantitative Reasoning for Proving Lock-Freedom. In *Logic in Computer Science (LICS)*. 124–133.

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.

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.

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.

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.

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.

Tobias Nipkow and Hauke Brinkop. 2019. Amortized Complexity Verified. *Journal of Automated Reasoning* 62, 3 (2019), 367–391.

Chris Okasaki. 1999. *Purely Functional Data Structures*. Cambridge University Press.

Alexandre Pilkiewicz and François Pottier. 2011. The essence of monotonic state. In *Types in Language Design and Implementation (TLDI)*.

Vlad Rusu and David Nowak. 2022. Defining Corecursive Functions in Coq Using Approximations. In *European Conference on Object-Oriented Programming (ECOOP) (LIPIcs, Vol. 222)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 12:1–12:24.

Robert Endre Tarjan. 1985. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods* 6, 2 (1985), 306–318.

Niccolò Veltri and Niels van der Weide. 2019. Guarded Recursion in Agda via Sized Types. In *Formal Structures for Computation and Deduction (FSCD) (LIPIcs, Vol. 131)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:19.

Bohua Zhan and Maximilian P. L. Haslbeck. 2018. Verifying Asymptotic Time Complexity of Imperative Programs in Isabelle. In *International Joint Conference on Automated Reasoning*.