Time Credits and Time Receipts in Iris

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

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

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

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

— Lewis Carroll, *Alice in Wonderland*

1 Introduction

A program logic, such as Hoare logic or Separation Logic, is a set of deduction rules that can be used to reason about the behavior of a program. To this day, considerable effort has been invested in developing ever-more-powerful program logics that control the *extensional* behavior of programs, that is, logics that guarantee that a program safely computes a valid final result. A lesser effort has been devoted to logics that allow reasoning not just about safety and functional correctness, but also about *intensional* aspects of a program's behavior, such as its time consumption and space usage.

In this paper, we are interested in narrowing the gap between these lines of work. We present a formal study of two mechanisms by which a standard program logic can be extended with means of reasoning about time. As a starting point, we take Iris [13,11,14,12], a powerful evolution of Concurrent Separation Logic [3]. We extend Iris with two elementary time-related concepts, namely *time credits* [1,9,4] and *time receipts*.

Time credits and time receipts are independent concepts: it makes sense to extend a program logic with either of them in isolation or with both of them simultaneously. They are dual concepts: every computation step *consumes one time credit* and *produces one time receipt*. They are purely static: they do not exist at runtime. We view them as Iris assertions. Thus, they can appear in the correctness statements that we formulate about programs and in the proofs of these statements.

Time credits can be used to establish an upper bound on the execution time of a program. Dually, time receipts can be used to establish a lower bound, and (as explained shortly) can be used to prove that certain undesirable events cannot occur until a very long time has elapsed.

Until now, time credits have been presented as an ad hoc extension of some fixed flavor of Separation Logic [1,9,4]. In contrast, we propose a construction which in principle allows time credits to be introduced on top of an arbitrary "base logic", provided this base logic is a sufficiently rich variety of Separation Logic. In order to make our definitions and proofs more concrete, we use Iris as the base logic. Our construction involves *composing* the base logic with a program transformation that inserts a *tick*() instruction in front of every computation step. As far as a user of the composite logic is concerned, the *tick*() instruction and the assertion \$1, which represents one time credit, are abstract: the only fact to which the user has access is the Hoare triple $\{\$1\}\ \mathit{tick}()\ \{\mathsf{True}\}$, which states that "*tick*() consumes one time credit".

There are two reasons why we choose Iris [12] as the base logic. First, in the proof of soundness of the composite logic, we must exhibit concrete definitions of *tick* and \$1 such that $\{\$1\}\ \mathit{tick}()\ \{\mathsf{True}\}$ holds. Several features of Iris, such as ghost state and shared invariants, play a key role in this construction. Second, at the user level, the power of Iris can also play a crucial role. To illustrate this, we present the first machine-checked reconstruction of Okasaki's debits [18] in terms of time credits. The construction makes crucial use of both time credits and Iris' ghost monotonic state and shared invariants.

Time receipts are a new concept, a contribution of this paper. To extend a base logic with time receipts, we follow the exact same route as above: we compose the base logic with the *same* program transformation as above, which we refer to as "the tick translation". In the eyes of a user of the composite logic, the *tick*() instruction and the assertion $⧗1$, which represents one time receipt, are again abstract: this time, the only published fact about *tick* is the triple $\{\mathsf{True}\}\ \mathit{tick}()\ \{⧗1\}$, which states that "*tick*() produces one time receipt".

Thus far, the symmetry between time credits and time receipts seems perfect: whereas time credits allow establishing an upper bound on the cost of a program fragment, time receipts allow establishing a lower bound. This raises a pragmatic question, though: why invest effort, time and money into a formal proof that a piece of code is slow? What might be the point of such an endeavor? Taking inspiration from Clochard *et al.* [5], we answer this question by turning slowness into a quality. If there is a certain point at which a process might fail, then by showing that this process is slow, we can show that failure is far away into the future. More specifically, Clochard *et al.* propose two abstract types of integer counters, dubbed "one-time" integers and "peano" integers, and provide a paper proof that these counters cannot overflow in a feasible time: that is, it would take infeasible time (say, centuries) for an execution to reach a point where overflow actually occurs. To reflect this idea, we abandon the symmetry between time credits and time receipts and publish a fact about time receipts which has no counterpart on the time-credit side. This fact is an implication: $⧗N \Rrightarrow_{\top} \mathsf{False}$,

that is, "N time receipts imply False". The global parameter N can be adjusted so as to represent one's idea of a running time that is infeasible, perhaps due to physical limitations, perhaps due to assumptions about the conditions in which the software is operated. In this paper, we explain what it means for the composite program logic to remain sound in the presence of this axiom, and provide a formal proof that Iris, extended with time receipts, is indeed sound. Furthermore, we verify that Clochard *et al.*'s ad hoc concepts of "one-time" integers and "peano" integers can be reconstructed in terms of time receipts, a more fundamental concept.

Finally, to demonstrate the combined use of time credits and receipts, we present a proof of the Union-Find data structure, where credits are used to express an amortized time complexity bound and receipts are used to prove that a node's integer rank cannot overflow, even if it is stored in very few bits.

In summary, the contributions of this paper are as follows:

1. A way of extending an off-the-shelf program logic with time credits and/or receipts, by composition with a program transformation.
2. Extensions of Iris with time credits and receipts, accompanied with machine-checked proofs of soundness.
3. A machine-checked reconstruction of Okasaki's debits as a library in Iris with time credits.
4. A machine-checked reconstruction of Clochard *et al.*'s "one-time" integers and "peano" integers in Iris with time receipts.
5. A machine-checked verification of Union-Find in Iris with time credits and receipts, offering both an amortized complexity bound and a safety guarantee despite the use of machine integers of very limited width.

All of the results reported in this paper have been checked in Coq [17].

2 A user's overview of time credits and time receipts

2.1 Time credits

A small number of axioms, presented in Figure 1, govern time credits. The assertion $\$n$ denotes n time credits. The splitting axiom, a logical equivalence, means that *time credits can be split and combined*. Because Iris is an affine logic, it is implicitly understood that *time credits cannot be duplicated, but can be thrown away*.

The axiom timeless($\$n$) means that time credits are independent of Iris' step-indexing. In practice, this allows an Iris invariant that involves time credits to be acquired without causing a "later" modality to appear [12, §5.7]. The reader can safely ignore this detail.

The last axiom, a Hoare triple, means that *every computation step requires and consumes one time credit*. As in Iris, the postconditions of our Hoare triples are λ-abstractions: they take as a parameter the return value of the term. At this point, *tick* () can be thought of as a pseudo-instruction that has no runtime effect and is implicitly inserted in front of every computation step.

$$
\begin{array}{cl}
\$: \mathbb{N} \to \mathit{iProp} & \text{— there is such a thing as “}n\text{ time credits”} \\
\mathsf{timeless}(\$n) & \text{— an Iris technicality} \\
\mathsf{True} \Rrightarrow_\top \$0 & \text{— zero credits can be created out of thin air} \\
\$(n_1 + n_2) \equiv \$n_1 \ast \$n_2 & \text{— credits can be split and combined} \\
\mathit{tick} : \mathit{Val} & \text{— there is a } \mathit{tick} \text{ pseudo-op} \\
\{\$1\}\ \mathit{tick}\ (v)\ \{\lambda w.\, w = v\} & \text{— } \mathit{tick} \text{ consumes one credit}
\end{array}
$$

Fig. 1. The axiomatic interface *TCIntf* of time credits

$$
\begin{array}{cl}
\text{⧗} : \mathbb{N} \to \mathit{iProp} & \text{— there is such a thing as “}n\text{ time receipts”} \\
\mathsf{timeless}(\text{⧗}n) & \text{— an Iris technicality} \\
\mathsf{True} \Rrightarrow_\top \text{⧗}0 & \text{— zero receipts can be created out of thin air} \\
\text{⧗}(n_1 + n_2) \equiv \text{⧗}n_1 \ast \text{⧗}n_2 & \text{— receipts can be split and combined} \\
\mathit{tick} : \mathit{Val} & \text{— there is a } \mathit{tick} \text{ pseudo-op} \\
\{\mathsf{True}\}\ \mathit{tick}\ (v)\ \{\lambda w.\, w = v \ast \text{⧗}1\} & \text{— } \mathit{tick} \text{ produces one receipt} \\
\text{⧗}N \Rrightarrow_\top \mathsf{False} & \text{— no machine runs for } N \text{ time steps}
\end{array}
$$

Fig. 2. The axiomatic interface of exclusive time receipts (further enriched in Figure 3)

Time credits can be used to express *worst-case time complexity guarantees*. For instance, a sorting algorithm could have the following specification:

$$
\begin{array}{c}
\{\mathit{array}(a, xs) \ \ast\ n = |xs| \ \ast\ \$(6n \log n)\} \\
\mathit{sort}(a) \\
\{\mathit{array}(a, xs') \wedge xs' = \ldots\}
\end{array}
$$

Here, $\mathit{array}(a, xs)$ asserts the existence and unique ownership of an array at address a, holding the sequence of elements xs. This Hoare triple guarantees not only that the function call $\mathit{sort}(a)$ runs safely and has the effect of sorting the array at address a, but also that $\mathit{sort}(a)$ runs in at most $6n \log n$ time steps, where n is the length of the sequence xs, that is, the length of the array. Indeed, only $6n \log n$ time credits are provided in the precondition, so the algorithm does not have permission to run for a greater number of steps.

2.2 Time receipts

In contrast with time credits, time receipts are a new concept, a contribution of this paper. We distinguish two forms of time receipts. The most basic form, *exclusive time receipts*, is the dual of time credits, in the sense that *every computation step produces one time receipt*. The second form, *persistent time receipts*, exhibits slightly different properties. Inspired by Clochard *et al.* [5], we show that time receipts can be used to *prove that certain undesirable events, such as integer overflows, cannot occur unless a program is allowed to execute for a very, very long time*—typically centuries. In the following, we explain that exclusive

time receipts allow reconstructing Clochard *et al.*'s "one-time" integers [5, §3.2], which are so named because they are not duplicable, whereas persistent time receipts allow reconstructing their "peano" integers [5, §3.2], which are so named because they do not support unrestricted addition.

Exclusive time receipts The assertion $⧗ n$ denotes n time receipts. Like time credits, these time receipts are "exclusive", by which we mean that they are not duplicable. The basic laws that govern exclusive time receipts appear in Figure 2. They are the same laws that govern time credits, with two differences. The first difference is that time receipts are the dual of time credits: the specification of *tick*, in this case, states that *every computation step produces one time receipt.*[3] The second difference lies in the last axiom of Figure 2, which has no analogue in Figure 1, and which we explain below.

In practice, how do we expect time receipts to be exploited? They can be used to prove lower bounds on the execution time of a program: if the Hoare triple $\{\mathsf{True}\}\ p\ \{⧗ n\}$ holds, then the execution of the program p cannot terminate in less than n steps. Inspired by Clochard *et al.* [5], we note that time receipts can also be used to *prove that certain undesirable events cannot occur in a feasible time*. This is done as follows. Let N be a fixed integer, chosen large enough that a modern processor cannot possibly execute N operations in a feasible time.[4] The last axiom of Figure 2, $⧗ N \Rrightarrow_\top \mathsf{False}$, states that N time receipts imply a contradiction.[5] This axiom informally means that *we won't compute for N time steps*, because we cannot, or because we promise not to do such a thing. A consequence of this axiom is that $⧗ n$ implies $n < N$: that is, *if we have observed n time steps, then n must be small.*

Adopting this axiom weakens the guarantee offered by the program logic. A Hoare triple $\{\mathsf{True}\}\ p\ \{\mathsf{True}\}$ no longer implies that the program p is forever safe. Instead, it means that p is $(N-1)$-safe: the execution of p cannot go wrong until at least $N-1$ steps have been taken. Because N is very large, for many practical purposes, this is good enough.

How can this axiom be exploited in practice? We hinted above that it can be used to prove the absence of certain integer overflows. Suppose that we wish to use signed w-bit machine integers as a representation of mathematical integers. (For instance, let w be 64.) Whenever we perform an arithmetic operation, such as an addition, we must prove that no overflow can occur. This is reflected in the specification of the addition of two machine integers:

$$\begin{array}{c}\{\iota(x_1) = n_1 \ *\ \iota(x_2) = n_2 \ *\ -2^{w-1} \leq n_1 + n_2 < 2^{w-1}\} \\ add(x_1, x_2) \\ \{\lambda x.\, \iota(x) = n_1 + n_2\}\end{array}$$

[3] For now, we discuss time credits and time receipts separately, which is why we have different specifications for *tick* in either case. They are combined in §6.

[4] For a specific example, let N be 2^{63}. Clochard *et al.* note that, even at the rate of one billion operations per second, it takes more than 292 years to execute 2^{63} operations. On a 64-bit machine, 2^{63} is also the maximum representable signed integer, plus one.

[5] The connective $\Rrightarrow_\top$ is an Iris view shift, that is, a transition that can involve a side effect on ghost state.

Here, the variables x_i denote machine integers, while the auxiliary variables n_i denote mathematical integers, and the function ι is the injection of machine integers into mathematical integers. The conjunct $-2^{w-1} \leq n_1 + n_2 < 2^{w-1}$ in the precondition represents an obligation to prove that no overflow can occur.

Suppose now that the machine integers x_1 and x_2 represent the lengths of two disjoint linked lists that we wish to concatenate. To construct each of these lists, we must have spent a certain amount of time: as proofs of this work, let us assume that the assertions $⧗ n_1$ and $⧗ n_2$ are at hand. Let us further assume that the word size w is sufficiently large that it takes a very long time to count up to the largest machine integer. That is, let us make the following assumption:

$$N \leq 2^{w-1} \qquad \text{(large word size assumption)}$$

(E.g., with $N = 2^{63}$ and $w = 64$, this holds.) Then, we can prove that the addition of x_1 and x_2 is permitted. This goes as follows. From the separating conjunction $⧗ n_1 \ * \ ⧗ n_2$, we get $⧗(n_1 + n_2)$. The existence of these time receipts allows us to deduce $0 \leq n_1 + n_2 < N$, which implies $0 \leq n_1 + n_2 < 2^{w-1}$. Thus, the precondition of the addition operation $\mathit{add}(x_1, x_2)$ is met.

In summary, we have just verified that the addition of two machine integers satisfies the following alternative specification:

$$\begin{array}{c} \{\iota(x_1) = n_1 \ * \ ⧗ n_1 \ * \ \iota(x_2) = n_2 \ * \ ⧗ n_2\} \\ \mathit{add}(x_1, x_2) \\ \{\lambda x.\, \iota(x) = n_1 + n_2 \ * \ ⧗(n_1 + n_2)\} \end{array}$$

This can be made more readable and more abstract by defining a "clock" to be a machine integer x accompanied with $\iota(x)$ time receipts:

$$\mathit{clock}(x) \triangleq \exists n.(\iota(x) = n \ * \ ⧗ n)$$

Then, the above specification of addition can be reformulated as follows:

$$\begin{array}{c} \{\mathit{clock}(x_1) \ * \ \mathit{clock}(x_2)\} \\ \mathit{add}(x_1, x_2) \\ \{\lambda x.\, \mathit{clock}(x) \ * \ \iota(x) = \iota(x_1) + \iota(x_2)\} \end{array}$$

In other words, clocks support unrestricted addition, without any risk of overflow. However, because time receipts cannot be duplicated, neither can clocks: $\mathit{clock}(x)$ does not entail $\mathit{clock}(x) \ * \ \mathit{clock}(x)$. In other words, a clock is uniquely owned. One can think of a clock x as a *hard-earned integer*: the owner of this clock has spent x units of time to obtain it.

Clocks are a reconstruction of Clochard *et al.*'s "one-time integers" [5], which support unrestricted addition, but cannot be duplicated. Whereas Clochard *et al.* view one-time integers as a primitive concept, and offer a direct paper proof of their soundness, we have just reconstructed them in terms of a more elementary notion, namely time receipts, and in the setting of a more powerful program logic, whose soundness is machine-checked, namely Iris.

$\blacklozenge : \mathbb{N} \to \mathit{iProp}$	— there is such a thing as "n exclusive time receipts"
$\boxtimes : \mathbb{N} \to \mathit{iProp}$	— and "a persistent receipt for n steps"
$\mathsf{timeless}(\blacklozenge\, n) \land \mathsf{timeless}(\boxtimes\, n)$	— an Iris technicality
$\mathsf{persistent}(\boxtimes\, n)$	— persistent receipts are persistent
$\mathsf{True} \Rrightarrow_\top \blacklozenge 0$	— zero receipts can be created out of thin air
$\blacklozenge(n_1 + n_2) \equiv \blacklozenge n_1 \ast \blacklozenge n_2$	— exclusive receipts obey addition
$\boxtimes \max(n_1, n_2) \equiv \boxtimes n_1 \ast \boxtimes n_2$	— persistent receipts obey maximum
$\blacklozenge n \Rrightarrow_\top \blacklozenge n \ast \boxtimes n$	— taking a snapshot of n exclusive receipts yields a persistent receipt for n steps
$\boxtimes N \Rrightarrow_\top \mathsf{False}$	— no machine runs for N time steps
$\mathit{tick} : \mathit{Val}$	— there is a *tick* pseudo-op
$\{\boxtimes n\}$ $\mathit{tick}\ (v)$ $\{\lambda w.\, w = v \ast \blacklozenge 1 \ast \boxtimes(n+1)\}$	— *tick* produces one exclusive receipt, and can increment an existing persistent receipt

Fig. 3. The axiomatic interface *TRIntf* of time receipts

Persistent time receipts In addition to exclusive time receipts, it is useful to introduce a persistent form of time receipts.[6] The axioms that govern both exclusive and persistent time receipts appear in Figure 3.

We write $\boxtimes n$ for a persistent receipt, a witness that at least n units of time have elapsed. (We avoid the terminology "n persistent time receipts", in the plural form, because persistent time receipts are not additive. We view $\boxtimes n$ as one receipt whose face value is n.) This assertion is persistent, which in Iris terminology means that once it holds, it holds forever. This implies, in particular, that it is duplicable: $\boxtimes n \equiv \boxtimes n \ast \boxtimes n$. It is created just by observing the existence of n exclusive time receipts, as stated by the following axiom, also listed in Figure 3: $\blacklozenge n \Rrightarrow_\top \blacklozenge n \ast \boxtimes n$. Intuitively, someone who has access to the assertion $\boxtimes n$ is someone who knows that n units of work have been performed, even though they have not necessarily "personally" performed that work. Because this knowledge is not exclusive, the conjunction $\boxtimes n_1 \ast \boxtimes n_2$ does not entail $\boxtimes(n_1+n_2)$. Instead, we have the following axiom, also listed in Figure 3: $\boxtimes(\max(n_1, n_2)) \equiv \boxtimes n_1 \ast \boxtimes n_2$.

More subtly, the specification of *tick* in Figure 3 is stronger than the one in Figure 2. According to this strengthened specification, *tick* () does not just produce an exclusive receipt $\blacklozenge 1$. In addition to that, if a persistent time receipt $\boxtimes n$ is at hand, then *tick* () is able to increment it and to produce a new persistent receipt $\boxtimes(n+1)$, thus reflecting the informal idea that a *new* unit of time has just been spent. A user who does not wish to make use of this feature can pick $n = 0$ and recover the specification of *tick* in Figure 2 as a special case.

Finally, because $\boxtimes n$ means that n steps have been taken, and because we promise never to reach N steps, we adopt the axiom $\boxtimes N \Rrightarrow_\top \mathsf{False}$, also listed

[6] Instead of viewing persistent time receipts as a primitive concept, one could define them as a library on top of exclusive time receipts. Unfortunately, this construction leads to slightly weaker laws, which is why we prefer to view them as primitive.

in Figure 3. It implies the earlier axiom $⧗ N \Rrightarrow_\top \mathsf{False}$, which is therefore not explicitly shown in Figure 3.

In practice, how are persistent time receipts exploited? By analogy with clocks, let us define a predicate for a machine integer x accompanied with $\iota(x)$ persistent time receipts:

$$\mathit{snapclock}(x) \triangleq \exists n.(\iota(x) = n \ * \ ⧖ n)$$

By construction, this predicate is persistent, therefore duplicable:

$$\mathit{snapclock}(x) \equiv \mathit{snapclock}(x) \ * \ \mathit{snapclock}(x)$$

We refer to this concept as a "snapclock", as it is not a clock, but can be thought of as a snapshot of some clock. Thanks to the axiom $⧗ k \Rrightarrow_\top ⧗ k \ * \ ⧖ k$, we have:

$$\mathit{clock}(x) \Rrightarrow_\top \mathit{clock}(x) \ * \ \mathit{snapclock}(x)$$

Furthermore, snapclocks have the valuable property that, by performing just one step of extra work, a snapclock can be incremented, yielding a new snapclock that is greater by one. That is, the following Hoare triple holds:

$$\begin{array}{c}\{\mathit{snapclock}(x)\} \\ \mathit{tick}\ (); \mathit{add}(x, 1) \\ \{\lambda x'.\ \mathit{snapclock}(x') \ * \ \iota(x') = \iota(x) + 1\}\end{array}$$

The proof is not difficult. Unfolding $\mathit{snapclock}(x)$ in the precondition yields $⧖ n$, where $\iota(x) = n$. As per the strengthened specification of *tick*, the execution of $\mathit{tick}\ ()$ then yields $⧗ 1 \ * \ ⧖(n+1)$. As in the case of clocks, the assertion $⧖(n+1)$ implies $0 \leq n + 1 < 2^{w-1}$, which means that no overflow can occur. Finally, $⧗ 1$ is thrown away and $⧖(n+1)$ is used to justify $\mathit{snapclock}(x')$ in the postcondition.

Adding two arbitrary snapclocks x_1 and x_2 is illegal: from the sole assumption $\mathit{snapclock}(x_1) \ * \ \mathit{snapclock}(x_2)$, one cannot prove that the addition of x_1 and x_2 won't cause an overflow, and one cannot prove that its result is a valid snapclock. However, snapclocks do support a restricted form of addition. The addition of two snapclocks x_1 and x_2 is safe, and produces a valid snapclock x, provided it is known ahead of time that its result is less than some preexisting snapclock y:

$$\begin{array}{c}\{\mathit{snapclock}(x_1) \ * \ \mathit{snapclock}(x_2) \ * \ \iota(x_1 + x_2) \leq \iota(y) \ * \ \mathit{snapclock}(y)\} \\ \mathit{add}(x_1, x_2) \\ \{\lambda x.\ \mathit{snapclock}(x) \ * \ \iota(x) = \iota(x_1) + \iota(x_2)\}\end{array}$$

Snapclocks are a reconstruction of Clochard *et al.*'s "peano integers" [5], which are so named because they do not support unrestricted addition. Clocks and snapclocks represent different compromises: whereas clocks support addition but not duplication, snapclocks support duplication but not addition. They are useful in different scenarios: as a rule of thumb, if an integer counter is involved in the implementation of a mutable data structure, then one should attempt to view it as a clock; if it is involved in the implementation of a persistent data structure, then one should attempt to view it as a snapclock.

3 HeapLang and the tick translation

In the next section (§4), we extend Iris with time credits, yielding a new program logic Iris$^{\$}$. We do this *without modifying* Iris. Instead, we *compose* Iris with a program transformation, the "tick translation", which inserts *tick*() instructions into the code in front of every computation step. In the construction of Iris$^{⧗}$, our extension of Iris with time receipts, the tick translation is exploited in a similar way (§5). In this section (§3), we define the tick translation and state some of its properties.

Iris is a generic program logic: it can be instantiated with an arbitrary calculus for which a small-step operational semantics is available [12]. Ideally, our extension of Iris should take place at this generic level, so that it, too, can be instantiated for an arbitrary calculus. Unfortunately, it seems difficult to define the tick translation and to prove it correct in a generic manner. For this reason, we choose to work in the setting of HeapLang [12], an untyped λ-calculus equipped with Booleans, signed machine integers, products, sums, recursive functions, references, and shared-memory concurrency. The three standard operations on mutable references, namely allocation, reading, and writing, are available. A compare-and-set operation $\texttt{CAS}(e_1, e_2, e_3)$ and an operation for spawning a new thread are also provided. As the syntax and operational semantics of HeapLang are standard and very much irrelevant in this paper, we omit them. They appear in our online repository [17].

The tick translation transforms a HeapLang expression e to a HeapLang expression $\langle\!\langle e \rangle\!\rangle_{tick}$. It is parameterized by a value *tick*. Its effect is to insert a call to *tick* in front of every operation in the source expression e. The translation of a function application, for instance, is as follows:

$$\langle\!\langle e_1\ (e_2) \rangle\!\rangle_{tick} = tick\ (\langle\!\langle e_1 \rangle\!\rangle_{tick})\ (\langle\!\langle e_2 \rangle\!\rangle_{tick})$$

For convenience, we assume that *tick* can be passed an arbitrary value v as an argument, and returns v. Because evaluation in HeapLang is call-by-value and happens to be right-to-left[7], the above definition means that, after evaluating the argument $\langle\!\langle e_2 \rangle\!\rangle_{tick}$ and the function $\langle\!\langle e_1 \rangle\!\rangle_{tick}$, we invoke *tick*, then carry on with the function call. This translation is syntactically well-behaved: it preserves the property of being a value, and commutes with substitution. This holds for every value *tick*.

As far the end user is concerned, *tick* remains abstract (§2). Yet, in our constructions of Iris$^{\$}$ and Iris$^{⧗}$, we must provide a concrete implementation of it in HeapLang. This implementation, named $tick_c$, appears in Figure 4. A global integer counter c stores the number of computation steps that the program is still allowed to take. The call $tick_c\ ()$ decrements a global counter c, if this counter holds a nonzero value, and otherwise invokes *oops* ().

At this point, the memory location c and the value *oops* are parameters.

[7] If HeapLang used left-to-right evaluation, the definition of the translation would be slightly different, but the lemmas that we prove would be the same.

$$
\begin{array}{l}
\mathit{tick}_c \triangleq \texttt{rec}\ \mathit{self}\,(x) = \\
\quad \texttt{let}\ k = !\,c\ \texttt{in} \\
\quad \texttt{if}\ k = 0\ \texttt{then}\ \mathit{oops}\ () \\
\quad \texttt{else if}\ \texttt{CAS}(c, k, k-1)\ \texttt{then}\ x\ \texttt{else}\ \mathit{self}\ (x)
\end{array}
$$

Fig. 4. Implementation of tick_c in HeapLang

We stress that tick_c plays a role only in the proofs of soundness of Iris$^\$$ and Iris$^{⧗}$. It is never actually executed, nor is it shown to the end user.

Once *tick* is instantiated with tick_c, one can prove that the translation is correct in the following sense: the translated code takes the same computation steps as the source code and additionally keeps track of how many steps are taken. More specifically, if the source code can make n computation steps, and if c is initialized with a value m that is sufficiently large (that is, $m \geq n$), then the translated code can make n computation steps as well, and c is decremented from m to $m - n$ in the process.

Lemma 1 (Reduction Preservation). *Assume there is a reduction sequence:*

$$(T_1, \sigma_1) \ \rightarrow_{\mathsf{tp}}^{n} \ (T_2, \sigma_2)$$

Assume c is fresh for this reduction sequence. Let $m \geq n$. Then, there exists a reduction sequence:

$$(\langle\!\langle T_1 \rangle\!\rangle, \langle\!\langle \sigma_1 \rangle\!\rangle\,[c \leftarrow m]) \ \rightarrow_{\mathsf{tp}}^{*} \ (\langle\!\langle T_2 \rangle\!\rangle, \langle\!\langle \sigma_2 \rangle\!\rangle\,[c \leftarrow m - n])$$

In this statement, the metavariable T stands for a thread pool, while σ stands for a heap. The relation $\rightarrow_{\mathsf{tp}}$ is HeapLang's "threadpool reduction". For the sake of brevity, we write just $\langle\!\langle e \rangle\!\rangle$ for $\langle\!\langle e \rangle\!\rangle_{\mathit{tick}_c}$, that is, for the translation of the expression e, where *tick* is instantiated with tick_c. This notation is implicitly dependent on the parameters c and *oops*.

The above lemma holds for every choice of *oops*. Indeed, because the counter c initially holds the value m, and because we have $m \geq n$, the counter is never about to fall below zero, so *oops* is never invoked.

The next lemma also holds for every choice of *oops*. It states that if the translated program is safe and if the counter c has not yet reached zero then the source program is not just about to crash.

Lemma 2 (Immediate Safety Preservation). *Assume c is fresh for e. Let $m > 0$. If the configuration $(\langle\!\langle e \rangle\!\rangle, \langle\!\langle \sigma \rangle\!\rangle\,[c \leftarrow m])$ is safe, then either e is a value or the configuration (e, σ) is reducible.*

By combining Lemmas 1 and 2 and by contraposition, we find that safety is preserved backwards, as follows: if, when the counter c is initialized with m, the translated program $\langle\!\langle e \rangle\!\rangle$ is safe, then the source program e is m-safe.

Lemma 3 (Safety Preservation). *If for every location c the configuration $(\langle\!\langle T \rangle\!\rangle, \langle\!\langle \sigma \rangle\!\rangle\,[c \leftarrow m])$ is safe, then the configuration (T, σ) is m-safe.*

4 Iris with time credits

The authors of Iris [12] have used Coq both to check that Iris is sound and to offer an implementation of Iris that can be used to carry out proofs of programs. The two are tied: if $\{\mathsf{True}\}\ p\ \{\mathsf{True}\}$ can be established by applying the proof rules of Iris, then one gets a self-contained Coq proof that the program p is safe.

In this section, we temporarily focus on time credits and explain how we extend Iris with time credits, yielding a new program logic Iris$^\$$. The new logic is defined in Coq and still offers an end-to-end guarantee: if $\{\$k\}\ p\ \{\mathsf{True}\}$ can be established in Coq by applying the proof rules of Iris$^\$$, then one has proved in Coq that p is safe and runs in at most k steps.

To define Iris$^\$$, we compose Iris with the tick translation. We are then able to argue that, because this program transformation is operationally correct (that is, it faithfully accounts for the passing of time), and because Iris is sound (that is, it faithfully approximates the behavior of programs), the result of the composition is a sound program logic that is able to reason about time.

In the following, we view the interface *TCIntf* as explicitly parameterized over \$ and *tick*. Thus, we write "*TCIntf* (\$) *tick*" for the separating conjunction of all items in Figure 1 except the declarations of \$ and *tick*.

We require the end user, who wishes to perform proofs of programs in Iris$^\$$, to work with Iris$^\$$ triples, which are defined as follows:

Definition 1 (Iris$^\$$ triple). *An* Iris$^\$$ triple $\{P\}\, e\, \{\Phi\}_\$$ *is syntactic sugar for:*

$$\forall(\$: \mathbb{N} \to \mathit{iProp})\quad \forall \mathit{tick} \qquad \mathit{TCIntf}\ (\$)\ \mathit{tick}\ \twoheadrightarrow\ \{P\}\, \langle\!\langle e \rangle\!\rangle_{\mathit{tick}}\, \{\Phi\}$$

Thus, an Iris$^\$$ triple is in reality an Iris triple about the instrumented expression $\langle\!\langle e \rangle\!\rangle_{\mathit{tick}}$. While proving this Iris triple, the end user is given an abstract view of the predicate \$ and the instruction *tick*. He does not have access to their concrete definitions, but does have access to the laws that govern them.

We prove that Iris$^\$$ is sound in the following sense:

Theorem 1 (Soundness of Iris$^\$$). *If* $\{\$n\}\, e\, \{\mathsf{True}\}_\$$ *holds, then the machine configuration* $(e, \varnothing)$*, where* $\varnothing$ *is the empty heap, is safe and terminates in at most* n *steps.*

In other words, a program that is initially granted n time credits cannot run for more than n steps. To establish this theorem, we proceed roughly as follows:

1. we provide a concrete definition of *tick*;
2. we provide a concrete definition of \$ and prove that *TCIntf* (\$) *tick* holds;
3. this yields $\{\$n\}\, \langle\!\langle e \rangle\!\rangle_{\mathit{tick}}\, \{\mathsf{True}\}$; from this and from the correctness of the tick translation, we deduce that e cannot crash or run for more than n steps.

Step 1. Our first step is to provide an implementation of *tick*. As announced earlier (§3), we use tick_c (Figure 4). We instantiate the parameter *oops* with *crash*, an arbitrary function whose application is unsafe. (That is, *crash* is chosen so that *crash* () reduces to a stuck term.) For the moment, c remains a parameter.

With these concrete choices of *tick* and *oops*, the translation transforms an out-of-time-budget condition into a hard crash. Because Iris forbids crashes, Iris$^{\$}$, which is the composition of the translation with Iris, will forbid out-of-time-budget conditions, as desired.

For technical reasons, we need two more lemmas about the translation, whose proofs rely on the fact that *oops* is instantiated with *crash*. They are slightly modified or strengthened variants of Lemmas 2 and 3. First, if the source code can take one step, then the translated code, supplied with zero budget, crashes. Second, if the translated code, supplied with a runtime budget of m, does *not* crash, then the source code terminates in at most m steps.

Lemma 4 (Credit Exhaustion). *Suppose the configuration (T, σ) is reducible. Then, for all c, the configuration $(\langle\!\langle T \rangle\!\rangle, \langle\!\langle \sigma \rangle\!\rangle\, [c \leftarrow 0])$ is unsafe.*

Lemma 5 (Safety Preservation, Strengthened). *If for every location c the configuration $(\langle\!\langle T \rangle\!\rangle, \langle\!\langle \sigma \rangle\!\rangle\, [c \leftarrow m])$ is safe, then (T, σ) is safe and terminates in at most m steps.*

Step 2. Our second step, roughly, is to exhibit a definition of $\$: \mathbb{N} \to \mathit{iProp}$ such that $\mathit{TCIntf}\ (\$)\ \mathit{tick}_c$ is satisfied. That is, we would like to prove something along the lines of: $\exists(\$: \mathbb{N} \to \mathit{iProp}) \quad \mathit{TCIntf}\ (\$)\ \mathit{tick}_c$. However, these informal sentences do not quite make sense. This formula is not an ordinary proposition: it is an Iris assertion, of type *iProp*. Thus, it does not make sense to say that this formula "is true" in an absolute manner. Instead, we prove in Iris that we can *make this assertion true* by performing a view shift, that is, a number of operations that have no runtime effect, such as allocating a ghost location and imposing an invariant that ties this ghost state with the physical state of the counter c. This is stated as follows:

Lemma 6 (Time Credit Initialization). *For every c and n, the following Iris view shift holds:*

$$(c \mapsto n) \quad \Rrightarrow_{\top} \quad \exists(\$: \mathbb{N} \to \mathit{iProp}) \quad (\mathit{TCIntf}\ (\$)\ \mathit{tick}_c \ \ast \ \$n)$$

In this statement, on the left-hand side of the view shift symbol, we find the "points-to" assertion $c \mapsto n$, which represents the unique ownership of the memory location c and the assumption that its initial value is n. This assertion no longer appears on the right-hand side of the view shift. This reflects the fact that, when the view shift takes place, it becomes impossible to access c directly; the only way of accessing it is via the operation tick_c.

On the right-hand side of the view shift symbol, beyond the existential quantifier, we find a conjunction of the assertion $\mathit{TCIntf}\ (\$)\ \mathit{tick}_c$, which means that the laws of time credits are satisfied, and $\$n$, which means that there are initially n time credits in existence.

In the interest of space, we provide only a brief summary of the proof of Lemma 6; the reader is referred to Appendix A for more details. In short, the assertion $\$1$ is defined in such a way that it represents an exclusive contribution

of one unit to the current value of the global counter c. In other words, we install the following invariant: at every time, the current value of c is (at least) the sum of all time credits in existence. Thus, the assertion \$1 guarantees that c is nonzero, and can be viewed as a permission to decrement c by one. This allows us to prove that the specification of *tick* in Figure 1 is satisfied by our concrete implementation $tick_c$. In particular, $tick_c$ cannot cause a crash: indeed, under the precondition \$1, c is not in danger of falling below zero, and *crash* () is not executed—it is in fact dead code.

Step 3. In the last reasoning step, we complete the proof of Theorem 1. The proof is roughly as follows. Suppose the end user has established $\{\$n\}\, e\, \{\mathsf{True}\}_{\$}$. By Safety Preservation, Strengthened (Lemma 5), to prove that $(e, \varnothing)$ is safe and runs in at most n steps, it suffices to show (for an arbitrary location c) that the translated expression $\langle\!\langle e \rangle\!\rangle$, executed in the initial heap $\varnothing\,[c \leftarrow n]$, is safe. To do so, beginning with this initial heap, we perform Time Credit Initialization, that is, we execute the view shift whose statement appears in Lemma 6. This yields an abstract predicate \$ as well as the assertions *TCIntf* (\$) *tick* and $\$n$. At this point, we unfold the Iris$^{\$}$ triple $\{\$n\}\, e\, \{\mathsf{True}\}_{\$}$, yielding an implication (see Definition 1), and apply it to \$, to $tick_c$, and to the hypothesis *TCIntf* (\$) *tick*. This yields the Iris triple $\{\$n\}\ \langle\!\langle e \rangle\!\rangle\ \{\mathsf{True}\}$. Because we have $\$n$ at hand and because Iris is sound [12], this implies that $\langle\!\langle e \rangle\!\rangle$ is safe. This concludes the proof.

This last step is, we believe, where the modularity of our approach shines. Iris' soundness theorem is re-used as a black box, without change. In fact, any program logic other than Iris could be used as a basis for our construction, as along as it is expressive enough to prove Time Credit Initialization (Lemma 6). The last ingredient, Safety Preservation, Strengthened (Lemma 5), involves only the operational semantics of HeapLang, and is independent of Iris.

This was just an informal account of our proof. For further details, the reader is referred to the online repository [17].

5 Iris with time receipts

In this section, we extend Iris with time receipts and prove the soundness of the new logic, dubbed Iris$^{⧗}$. To do so, we follow the scheme established in the previous section (§4), and compose Iris with the tick translation.

From here on, let us view the interface of time receipts as parameterized over ⧗, ⧖, and *tick*. Thus, we write "*TRIntf* (⧗) (⧖) *tick*" for the separating conjunction of all items in Figure 3 except the declarations of ⧗, ⧖, and *tick*.

As in the case of credits, the user is given an abstract view of time receipts:

Definition 2 (Iris$^{⧗}$ triple). *An* Iris$^{⧗}$ triple $\{P\}\, e\, \{\Phi\}_{⧗}$ *is syntactic sugar for:*

$$\forall (⧗, ⧖ : \mathbb{N} \to \mathit{iProp})\quad \forall \mathit{tick}\qquad \mathit{TRIntf}\ (⧗)\ (⧖)\ \mathit{tick}\ \ \twoheadrightarrow\ \ \{P\}\ \langle\!\langle e \rangle\!\rangle_{\mathit{tick}}\ \{\Phi\}$$

Theorem 2 (Soundness of Iris$^{⧗}$). *If* $\{\mathsf{True}\}\, e\, \{\mathsf{True}\}_{⧗}$ *holds, then the machine configuration* $(e, \varnothing)$ *is* $(N-1)$*-safe.*

As indicated earlier, we assume that the end user is interested in proving that crashes cannot occur until a very long time has elapsed, which is why we state the theorem in this way.[8] Whereas an Iris triple $\{\mathsf{True}\}\, e\, \{\mathsf{True}\}$ guarantees that e is safe, the Iris$^{⧗}$ triple $\{\mathsf{True}\}\, e\, \{\mathsf{True}\}_{⧗}$ guarantees that it takes at least $N-1$ steps of computation for e to crash. In this statement, N is the global parameter that appears in the axiom $⧖ N \Rrightarrow_{\top} \mathsf{False}$ (Figure 3). Compared with Iris, Iris$^{⧗}$ provides a weaker safety guarantee, but offers additional reasoning principles, leading to increased convenience and modularity.

In order to establish Theorem 2, we again proceed in three steps:

1. provide a concrete definition of *tick*;
2. provide concrete definitions of $⧗, ⧖$ and prove that *TRIntf* $(⧗)$ $(⧖)$ *tick* holds;
3. from $\{\mathsf{True}\}\, \langle\!\langle e \rangle\!\rangle_{tick}\, \{\mathsf{True}\}$, deduce that e is $(N-1)$-safe.

Step 1. In this step, we keep our concrete implementation of *tick*, namely $tick_c$ (Figure 4). One difference with the case of time credits, though, is that we plan to initialize c with $N-1$. Another difference is that, this time, we instantiate the parameter *oops* with *loop*, where *loop* () is an arbitrary divergent term.[9]

Step 2. The next step is to prove that we are able to establish the time receipt interface. We prove the following:

Lemma 7 (Time Receipt Initialization). *For every location c, the following Iris view shift holds:*

$$(c \mapsto N-1) \quad \Rrightarrow_{\top} \quad \exists (⧗, ⧖ : \mathbb{N} \to \mathit{iProp}) \quad \mathit{TRIntf}\ (⧗)\ (⧖)\ \mathit{tick}_c$$

We provide only a brief summary of the proof of Lemma 7; for further details, the reader is referred to Appendix B. Roughly speaking, we install the invariant that c holds $N-1-i$, where i is some number that satisfies $0 \leq i < N$. We define $⧗ n$ as an exclusive contribution of n units to the current value of i, and define $⧖ n$ as an observation that i is at least n. (i grows with time, so such an observation is stable.) As part of the proof of the above lemma, we check that the specification of *tick* holds:

$$\{⧖ n\}\ \mathit{tick}\ (v)\ \{\lambda w.\ w = v\ *\ ⧗ 1\ *\ ⧖(n+1)\}$$

In contrast with the case of time credits, in this case, the precondition $⧖ n$ does *not* guarantee that c holds a nonzero value. Thus, it *is* possible for *tick*() to be executed when c is zero. This is not a problem, though, because *loop*() is safe to execute in any situation: it satisfies the Hoare triple $\{\mathsf{True}\}\ \mathit{loop}()\ \{\mathsf{False}\}$. In other words, when c is about to fall below zero and therefore the invariant $i < N$ seems about to be broken, *loop* () saves the day by running away and never allowing execution to continue normally.

[8] If the user instead wishes to establish a lower bound on a program's execution time, this is possible as well.

[9] In fact, it is not essential that *loop*() diverges. What matters is that *loop* satisfy the Iris triple $\{\mathsf{True}\}\ \mathit{loop}()\ \{\mathsf{False}\}$. A fatal runtime error that Iris does *not* rule out would work just as well, as it satisfies the same specification.

Step 3. In the last reasoning step, we complete the proof of Theorem 2. Suppose the end user has established $\{\mathsf{True}\}\ e\ \{\mathsf{True}\}_{⧗}$. By Safety Preservation (Lemma 3), to prove that $(e, \varnothing)$ is $(N-1)$-safe, it suffices to show (for an arbitrary location c) that $\langle\!\langle e \rangle\!\rangle$, executed in the initial heap $\varnothing\,[c \leftarrow N-1]$, is safe. To do so, beginning with this initial heap, we perform Time Receipt Initialization, that is, we execute the view shift whose statement appears in Lemma 7. This yields two abstract predicates ⧗ and ⧖ as well as the assertion *TRIntf* (⧗) (⧖) *tick*. At this point, we unfold $\{\mathsf{True}\}\ e\ \{\mathsf{True}\}_{⧗}$ (see Definition 2), yielding an implication, and apply this implication, yielding the Iris triple $\{\mathsf{True}\}\ \langle\!\langle e \rangle\!\rangle\ \{\mathsf{True}\}$. Because Iris is sound [12], this implies that $\langle\!\langle e \rangle\!\rangle$ is safe. This concludes the proof. For further detail, the reader is again referred to our online repository [17].

6 Marrying time credits and time receipts

It seems desirable to combine time credits and time receipts in a single program logic, Iris$^{\$⧗}$. We have done so [17]. In short, following the scheme of §4 and §5, the definition of Iris$^{\$⧗}$ involves composing Iris with the tick translation. This time, *tick* serves two purposes: it consumes one time credit *and* produces one exclusive time receipt (and increments a persistent time receipt). Thus, its specification is as follows:

$$\{\$1 * ⧖\, n\}\ \mathit{tick}\ (v)\ \{\lambda w.\, w = v * ⧗\, 1 * ⧖(n+1)\}$$

Let us write *TCTRIntf* (\$) (⧗) (⧖) *tick* for the combined interface of time credits and time receipts. This interface combines all of the axioms of Figures 1 and 3, but declares a single *tick* function[10] and proposes a single specification for it, which is the one shown above.

Definition 3 (Iris$^{\$⧗}$ triple). *An* Iris$^{\$⧗}$ triple $\{P\}\ e\ \{\Phi\}_{\$⧗}$ *stands for:*

$$\forall\ (\$)\ (⧗)\ (⧖)\ \mathit{tick} \qquad \mathit{TCTRIntf}\ (\$)\ (⧗)\ (⧖)\ \mathit{tick}\ \mathrel{-\!\!*}\ \{P\}\ \langle\!\langle e \rangle\!\rangle_{\mathit{tick}}\ \{\Phi\}$$

Theorem 3 (Soundness of Iris$^{\$⧗}$). *If* $\{\$n\}\ e\ \{\mathsf{True}\}_{\$⧗}$ *holds then the machine configuration* $(e, \varnothing)$ *is* $(N-1)$*-safe. If furthermore* $n < N$ *holds, then this machine configuration terminates in at most* n *steps.*

Iris$^{\$⧗}$ allows exploiting time credits to prove time complexity bounds and, at the same time, exploiting time receipts to prove the absence of certain integer overflows. Our verification of Union-Find (§8) illustrates these two aspects.

Guéneau *et al.* [7] use time credits to reason about asymptotic complexity, that is, about the manner in which a program's complexity grows as the size of its input grows towards infinity. Does such asymptotic reasoning make sense in Iris$^{\$⧗}$, where no program is ever executed for N time steps or beyond? It seems to be the case that if a program p satisfies the triple $\{\$n\}\ p\ \{\Phi\}_{\$⧗}$, then

[10] Even though the interface provides only one *tick* function, it gets instantiated in the soundness theorem with different implementations depending on whether there are more than N time credits or not.

it also satisfies the stronger triple $\{\$\min(n, N)\}\, p\, \{\Phi\}_{\$\blacksquare}$, therefore also satisfies $\{\$N\}\, p\, \{\Phi\}_{\$\blacksquare}$. Can one therefore conclude that p has "constant time complexity"? We believe not. Provided N is considered a parameter, as opposed to a constant, one *cannot* claim that "N is $O(1)$", so $\{\$\min(n, N)\}\, p\, \{\Phi\}_{\$\blacksquare}$ does not imply that "p runs in constant time". In other words, a universal quantification on N should come *after* the existential quantifier that is implicit in the O notation. We have not yet attempted to implement this idea; this remains a topic for further investigation.

7 Application: thunks in Iris$^{\$}$

In this section, we illustrate the power of Iris$^{\$}$ by constructing an implementation of thunks as a library in Iris$^{\$}$. A *thunk*, also known as a *suspension*, is a very simple data structure that represents a suspended computation. There are two operations on thunks, namely *create*, which constructs a new thunk, and *force*, which demands the result of a thunk. A thunk memoizes its result, so that even if it is forced multiple times, the computation only takes place once.

Okasaki [18] proposes a methodology for reasoning about the amortized time complexity of computations that involve shared thunks. For every thunk, he keeps track of a *debit*, which can be thought of as an amount of credit that one must still pay before one is allowed to force this thunk. A ghost operation, *pay*, changes one's view of a thunk, by reducing the debit associated with this thunk. *force* can be applied only to a zero-debit thunk, and has amortized cost $O(1)$. Indeed, if this thunk has been forced already, then *force* really requires constant time; and if this thunk is being forced for the first time, then the cost of performing the suspended computation must have been paid for in advance, possibly in several installements, via *pay*. This discipline is sound even in the presence of sharing, that is, of multiple pointers to a thunk. Indeed, whereas duplicating a credit is unsound, duplicating a debit leads to an over-approximation of the true cost, hence is sound. Danielsson [6] formulates Okasaki's ideas as a type system, which he proves sound in Agda. Pilkiewicz and Pottier [19] reconstruct this type discipline in the setting of a lower-level type system, equipped with basic notions of time credits, hidden state, and monotonic state. Unfortunately, their type system is presented in an informal manner and does not come with a proof of type soundness.

We reproduce Pilkiewicz and Pottier's construction in the formal setting of Iris$^{\$}$. Indeed, Iris$^{\$}$ offers all of the necessary ingredients, namely time credits, hidden state (invariants, in Iris terminology) and monotonic state (a special case of Iris' ghost state). Our reconstruction is carried out inside Coq [17].

7.1 Concurrency and reentrancy

One new problem that arises here is that Okasaki's analysis, which is valid in a sequential setting, potentially becomes invalid in a concurrent setting. Suppose we wish to allow multiple threads to safely share access to a thunk. A natural,

simple-minded approach would be to equip every thunk with a lock and allow competition over this lock. Then, unfortunately, forcing would become a blocking operation: one thread could waste time waiting for another thread to finish forcing. In fact, in the absence of a fairness assumption about the scheduler, an unbounded amount of time could be wasted in this way. This appears to invalidate the property that *force* has amortized cost $O(1)$.

Technically, the manner in which this problem manifests itself in Iris$^\$$ is in the specification of locks. Whereas in Iris a spin lock can be implemented and proved correct with respect to a simple and well-understood specification [2], in Iris$^\$$, it cannot. The *lock*() method contains a potentially infinite loop: therefore, no finite amount of time credits is sufficient to prove that *lock*() is safe. This issue is discussed in greater depth later on (§9).

A distinct yet related problem is reentrancy. Arguably, an implementation of thunks should guarantee that a suspended computation is evaluated at most once. This guarantee seems particularly useful when the computation has a side effect: the user can then rely on the fact that this side effect occurs at most once. However, this property does not naturally hold: in the presence of heap-allocated mutable state, it is possible to construct an ill-behaved "reentrant" thunk which, when forced, attempts to recursively force itself. Thus, something must be done to dynamically reject or statically prevent reentrancy. In Pilkiewicz and Pottier's code [19], reentrancy is detected at runtime, thanks to a three-color scheme, and causes a fatal runtime failure. In a concurrent system where each thunk is equipped with a lock, reentrancy is also detected at runtime, and turned into deadlock; but we have explained earlier why we wish to avoid locks.

Fortunately, Iris provides us with a static mechanism for forbidding both concurrency and reentrancy. We introduce a unique token $\lightning$, which can be thought of as "permission to use the thunk API", and set things up so that *pay* and *force* require and return $\lightning$. This forbids concurrency: two operations on thunks cannot take place concurrenly. Furthermore, when a user-supplied suspended computation is executed, the token $\lightning$ is *not* transmitted to it. This forbids reentrancy.[11] The implementation of this token relies on Iris' "nonatomic invariants" (§7.4). With these restrictions, we are able to prove that Okasaki's discipline is sound.

7.2 Implementation of thunks

A simple implementation of thunks in HeapLang appears in Figure 5. A thunk can be in one of two states: *White* f and *Black* v. A white thunk is unevaluated: the function f represents a suspended computation. A black thunk is evaluated: the value v is the result of the computation that has been performed already.

[11] Therefore, a suspended computation cannot force *any* thunk. This is admittedly a very severe restriction, which rules out many useful applications of thunks. In fact, we have implemented a more flexible discipline, where thunks can be grouped in multiple "regions" and there is one token per region instead of a single global $\lightning$ token. This discipline allows concurrent or reentrant operations on provably distinct thunks, yet can still be proven sound.

```
create ≜ λf. ref(White f)
force ≜ λt. match !t with
              White f ⇒ let v = f () in t ← Black v ; v
            | Black v ⇒ v
            end
```

Fig. 5. An implementation of thunks

$\mathsf{isThunk} : \mathit{Loc} \to \mathbb{N} \to (\mathit{Val} \to \mathit{iProp}) \to \mathit{iProp}$	— there exist "thunks"
$\mathsf{persistent}(\mathsf{isThunk}\ t\ n\ \Phi)$	— thunks can be shared
$n_1 \leq n_2 \twoheadrightarrow$ $\mathsf{isThunk}\ t\ n_1\ \Phi \twoheadrightarrow \mathsf{isThunk}\ t\ n_2\ \Phi$	— it is sound to overestimate a debt
$\lightning : \mathit{iProp}$	— there exist "thunderbolts"
$\lightning$	— the user is handed one
$\{\$3 * \{\$n\}\ \langle\!\langle f\ () \rangle\!\rangle\ \{\Phi\}\}$ $\langle\!\langle \mathit{create}\ (f) \rangle\!\rangle$ $\{\lambda t. \mathsf{isThunk}\ t\ n\ \Phi\}$	— a computation of cost n gives rise to an n-debit thunk; the cost is $O(1)$
$(\forall v. \mathsf{duplicable}(\Phi\ v)) \twoheadrightarrow$ $\{\$11 * \mathsf{isThunk}\ t\ 0\ \Phi * \lightning\}$ $\langle\!\langle \mathit{force}\ (t) \rangle\!\rangle$ $\{\lambda v. \Phi\ v * \lightning\}$	— a 0-debit thunk can be forced; the thunderbolt is required; the cost is $O(1)$
$\mathsf{isThunk}\ t\ n\ \Phi * \$k * \lightning$ $\Rrightarrow_\top \mathsf{isThunk}\ t\ (n-k)\ \Phi * \lightning$	— paying reduces one's debt

Fig. 6. A simple specification of thunks in Iris$^\$$

Two colors are sufficient: because our static discipline rules out reentrancy, there is no need for a third color, whose purpose would be to dynamically detect an attempt to force a thunk that is already being forced.

7.3 Specification of thunks in Iris$^\$$

Our specification of thunks appears in Figure 6. It declares an abstract predicate $\mathsf{isThunk}\ t\ n\ \Phi$, which asserts that t is a valid thunk, that the debt associated with this thunk is n, and that this thunk (once forced) produces a value that satisfies the postcondition Φ. The number n, a *debit*, is the number of credits that remain to be paid before this thunk can be forced. The postcondition Φ is chosen by the user when a thunk is created. It must be duplicable (this is required in the specification of *force*) because *force* can be invoked several times and we must guarantee, every time, that the result v satisfies $\Phi\ v$.

The second axiom states that $\mathsf{isThunk}\ t\ n\ \Phi$ is a persistent assertion. This means that a valid thunk, once created, remains a valid thunk forever. Among

other things, it is permitted to create two pointers to a single thunk and to reason independently about each of these pointers.

The third axiom states that $\mathsf{isThunk}\ t\ n\ \Phi$ is covariant in its parameter n. Overestimating a debt still leads to a correct analysis of a program's worst-case time complexity.

Next, the specification declares an abstract assertion $\lightning$, and provides the user with one copy of this assertion. We refer to it as "the thunderbolt".

The next item in Figure 6 is the specification of *create*. It is higher-order: the precondition of *create* contains a specification of the function f that is passed as an argument to *create*. This axiom states that, if f represents a computation of cost n, then *create* (f) produces an n-debit thunk. The cost of creation itself is 3 credits. This specification is somewhat simplistic, as it does not allow the function f to have a nontrivial precondition. It is possible to offer a richer specification; we eschew it in favor of simplicity.

Next comes the specification of *force*. Only a 0-debit thunk can be forced. The result is a value v that satisfies Φ. The (amortized) cost of forcing is 11 credits. The thunderbolt appears in the pre- and postcondition of *force*, forbidding any concurrent attempts to force a thunk.

The last axiom in Figure 6 corresponds to *pay*. It is a view shift, a ghost operation. By paying k credits, one turns an n-debit thunk into an $(n-k)$-debit thunk. At runtime, nothing happens: it is the same thunk before and after the payment. Yet, after the view shift, we have a new view of the number of debits associated with this thunk. Here, paying requires the thunderbolt. It should be possible to remove this requirement; we have not yet attempted to do so.

7.4 Proof of thunks in Iris$^{\$}$

After implementing thunks in HeapLang (§7.2) and expressing their specification in Iris$^{\$}$ (§7.3), there remains to prove that this specification can be established. We sketch the key ideas of this proof.

Following Pilkiewicz and Pottier [19], when a new thunk is created, we install a new Iris invariant, which describes this thunk. The invariant is as follows:

$$\mathit{ThunkInv}\ t\ \gamma\ nc\ \Phi\ \triangleq$$
$$\exists ac.\ \left(\boxed{\bullet\, ac}^{\gamma}\ *\ \begin{cases} \exists f.\ t \mapsto \mathit{White}\ f\ *\ \{\$nc\}\ f\ ()\ \{\Phi\}\ *\ \$ac \\ \lor\ \exists v.\ t \mapsto \mathit{Black}\ v \end{cases}\right)$$

γ is a ghost location, which we allocate at the same time as the thunk t. It holds elements of the authoritative monoid $\textsc{Auth}(\mathbb{N}, \max)$ [12]. The variable nc, for "necessary credits", is the cost of the suspended computation: it appears in the precondition of f. The variable ac, for "available credits", is the number of credits that have been paid so far. The disjunction inside the invariant states that:

- either the thunk is white, in which case we have ac credits at hand;
- or the thunk is black, in which case we have no credits at hand, as they have been spent already.

The predicate $\mathsf{isThunk}\ t\ n\ \Phi$ is then defined as follows:

$$\mathsf{isThunk}\ t\ n\ \Phi \triangleq$$
$$\exists \gamma, nc.\ \left(\boxed{\circ (nc - n)}^{\gamma} * \mathsf{NaInv}(\mathit{ThunkInv}\ t\ \gamma\ nc\ \Phi) \right)$$

The non-authoritative assertion $\boxed{\circ (nc - n)}^{\gamma}$ inside $\mathsf{isThunk}\ t\ n\ \Phi$, confronted with the authoritative assertion $\boxed{\bullet\, ac}^{\gamma}$ that can be obtained by acquiring the invariant, implies the inequality $nc - n \leq ac$, therefore $nc \leq ac + n$. That is, the credits paid so far (ac) plus the credits that remain to be paid (n) are sufficient to cover for the actual cost of the computation (nc). In particular, in the proof of *force*, we have a 0-debit thunk, so $nc \leq ac$ holds. In the case where the thunk is white, this means that the ac credits that we have at hand are sufficient to justify the call $f\ ()$, which requires nc credits.

The final aspect that remains to be explained is our use of $\mathsf{NaInv}(\cdots)$, an Iris "nonatomic invariant". Indeed, in this proof, we cannot rely on Iris' primitive invariants. A primitive invariant can be acquired only for the duration of an atomic instruction [12]. In our implementation of thunks (Figure 5), however, we need a "critical section" that encompasses several instructions. That is, we must acquire the invariant before dereferencing t, and (in the case where this thunk is white) we cannot release it until we have marked this thunk black. Fortunately, Iris provides a library of "nonatomic invariants" for this very purpose. (This library is used in the RustBelt project [10] to implement Rust's type `Cell`.) This library offers separate ghost operations for acquiring and releasing an invariant. Acquiring an invariant consumes a unique token, which is recovered when the invariant is released: this guarantees that an invariant cannot be acquired twice, or in other words, that two threads cannot be in a critical section at the same time. The unique token involved in this protocol is the one that we expose to the end user as "the thunderbolt".

8 Application: Union-Find in Iris$^{\$⧗}$

As an illustration of the use of both time credits and time receipts, we formally verify the functional correctness and time complexity of an implementation of the Union-Find data structure. Our proof [17] is based on Charguéraud and Pottier's work [4]. We port their code from OCaml to HeapLang, and port their proof from Separation Logic with Time Credits to Iris$^{\$⧗}$. At this point, the proof exploits just Iris$^{\$}$, a subset of Iris$^{\$⧗}$. The mathematical analysis of Union-Find, which represents a large part of the proof, is unchanged. Our contribution lies in the fact that we modify the data structure to represent ranks as machine integers instead of unbounded integers, and exploit time receipts in Iris$^{\$⧗}$ to establish the absence of overflow. We equip HeapLang with signed machine integers whose bit width is a parameter w. Under the hypothesis $\log \log N < w - 1$, we are able to prove that, even though the code uses limited-width machine integers, no overflow can occur in a feasible time. If for instance N is 2^{63}, then this condition boils down to $w \geq 7$. Ranks can be stored in just 7 bits without risking overflow.

As in Charguéraud and Pottier's work, the Union-Find library advertises an abstract representation predicate $\mathsf{isUF}\,D\,R\,V$, which describes a well-formed, uniquely-owned Union-Find data structure. The parameter D, a set of nodes, is the domain of the data structure. The parameter R, a function, maps a node to the representative element of its equivalence class. The parameter V, also a function, maps a node to a payload value associated with its equivalence class. We do not show the specification of every operation. Instead, we focus on *union*, which merges two equivalence classes. We establish the following Iris$^{\$\blacktriangleright}$ triple:

$$\left.\begin{array}{r} \log\log N < w - 1 \\ x \in D \\ y \in D \end{array}\right\} \quad \Rightarrow \quad \begin{array}{c} \{\mathsf{isUF}\,D\,R\,V \;*\; \$(44\alpha(|D|) + 152)\} \\ \mathit{union}\;(x, y) \\ \left\{\lambda z.\;\begin{array}{l}\mathsf{isUF}\,D\,R'\,V' \;* \\ z = R(x) \vee z = R(y)\end{array}\right\}_{\$\blacktriangleright} \end{array}$$

where the functions R' and V' are defined as follows:[12]

$$(R'(w), V'(w)) = \begin{cases} (z, \quad V(z)) & \text{if } R(w) = R(x) \text{ or } R(w) = R(y) \\ (R(w), V(w)) & \text{otherwise} \end{cases}$$

The hypotheses $x \in D$ and $y \in D$ and the conjunct $\mathsf{isUF}\,D\,R\,V$ in the precondition require that x and y be two nodes in a valid Union-Find data structure. The postcondition $\lambda z.\ \ldots$ describes the state of the data structure after the operation and the return value z.

The conjunct $\$(44\alpha(|D|) + 152)$ in the precondition indicates that *union* has time complexity $O(\alpha(n))$, where α is an inverse of Ackermann's function and n is the number of nodes in the data structure. This is an amortized bound; the predicate isUF also contains a certain number of time credits, known as the potential of the data structure, which are used to justify *union* operations whose actual cost exceeds the advertised cost. The constants 44 and 152 differ from those found in Charguéraud and Pottier's specification [4] because Iris$^{\$\blacktriangleright}$ counts every computation step, whereas they count only function calls. Abstracting these constants by using O notation, as proposed by Guéneau *et al.* [7], would be desirable, but we have not attempted to do so yet.

The main novelty, with respect to Charguéraud and Pottier's specification, is the hypothesis $\log\log N < w - 1$, which is required to prove that no overflow can occur when the rank of a node is incremented. In our proof, N and w are parameters; once their values are chosen, this hypothesis is easily discharged, once and for all. In the absence of time receipts, we would have to publish the hypothesis $\log\log n < w - 1$, where n is the cardinal of D, forcing every (direct and indirect) user of the data structure to keep track of this requirement.

For the proof to go through, we store n time receipts in the data structure: that is, we include the conjunct $\blacktriangleright n$, where n stands for $|D|$, in the definition of the invariant $\mathsf{isUF}\,D\,R\,V$. The operation of creating a new node takes at least one

[12] This definition of R' and V' has free variables x, y, z, therefore in reality must appear inside the postcondition. Here, it is presented separately, for greater readability.

step, therefore produces one new time receipt, which is used to prove that the invariant is preserved by this operation. At any point, then, from the invariant, and from the basic laws of time receipts, we can deduce that $n < N$ holds. Furthermore, it is easy to show that a rank is at most $\log n$. Therefore, a rank is at most $\log N$. In combination with the hypothesis $\log \log N < w - 1$, this suffices to prove that a rank is at most $2^{w-1} - 1$, the largest signed machine integer, and therefore that no overflow can occur in the computation of a rank.

Clochard *et al.* [5, §2] already present Union-Find as a motivating example among several others. They write that "there is obviously no danger of arithmetic overflow here, since [ranks] are only obtained by successive increments by one". This argument would be formalized in their system by representing ranks as either "one-time" or "peano" integers (in our terminology, clocks or snapclocks). This argument could be expressed in Iris$^{\$⧗}$, but would lead to requiring $\log N < w - 1$. In contrast, we use a more refined argument: we note that ranks are logarithmic in n, the number of nodes, and that n itself can never overflow. This leads us to the much weaker requirement $\log \log N < w - 1$, which means that a rank can be stored in very few bits. We believe that this argument cannot be expressed in Clochard *et al.*'s system.

9 Discussion

One feature of Iris and HeapLang that deserves further discussion is concurrency. Iris is an evolution of Concurrent Separation Logic, and HeapLang has shared-memory concurrency. How does this impact our reasoning about time? At a purely formal level, this does not have any impact: Theorems 1, 2, 3 and their proofs are essentially oblivious to the absence or presence of concurrency in the programming language. At a more informal level, though, this impacts our interpretation of the real-world meaning of these theorems. Whereas in a sequential setting a "number of computation steps" can be equated (up to a constant factor) with "time", in a concurrent setting, a "number of computation steps" is referred to as "work", and is related to "time" only up to a factor of p, the number of processors. In short, our system measures work, not time. The number of available processors should be taken into account when choosing a specific value of N: this value must be so large that N computation steps are infeasible even by p processors. With this in mind, we believe that our system can still be used to prove properties that have physical relevance.

In short, our new program logics, Iris$^{\$}$, Iris$^{⧗}$, and Iris$^{\$⧗}$, tolerate concurrency. Yet, is it fair to say that they have "good support" for reasoning about concurrent programs? We believe not yet, and this is an area for future research. The main open issue is that we do not at this time have good support for reasoning about the time complexity of programs that perform busy-waiting on some resource. The root of the difficulty, already mentioned during the presentation of thunks (§7.1), is that one thread can fail to make progress, due to interference with another thread. A retry is then necessary, wasting time. In a spin lock, for instance, the "compare-and-set" (CAS) instruction that attempts to acquire the

lock can fail. There is no bound on the number of attempts that are required until the lock is eventually acquired. Thus, in Iris$^{\$}$, we are currently unable to assign *any* specification to the *lock* method of a spin lock.

In the future, we wish to take inspiration from Hoffmann, Marmar and Shao [9], who use time credits in Concurrent Separation Logic to establish the lock-freedom of several concurrent data structures. The key idea is to formalize the informal argument that "failure of a thread to make progress is caused by successful progress in another thread". Hoffmann *et al.* set up a "quantitative compensation scheme", that is, a protocol by which successful progress in one thread (say, a successful CAS operation) must transmit a number of time credits to every thread that has encountered a corresponding failure and therefore must retry. Quite interestingly, this protocol is not hardwired into the reasoning rule for CAS. In fact, CAS itself is not primitive; it is encoded in terms of an $\mathsf{atomic}\ \{\ldots\}$ construct. The protocol is set up by the user, by exploiting the basic tools of Concurrent Separation Logic, including shared invariants. Thus, it should be possible in Iris$^{\$}$ to reproduce Hoffmann *et al.*'s reasoning and to assign useful specifications to certain lock-free data structures. Furthermore, we believe that, under a fairness assumption, it should be possible to assign Iris$^{\$}$ specifications also to coarse-grained data structures, which involve locks. Roughly speaking, under a fair scheduler, the maximum time spent waiting for a lock is the maximum number of threads that may compete for this lock, multiplied by the maximum cost of a critical section protected by this lock. Whether and how this can be formalized is a topic of future research.

The axiom $⧖\, N \Rrightarrow_\top \mathsf{False}$ comes with a few caveats that should be mentioned. The same caveats apply to Clochard *et al.*'s system [5], and are known to them.

One caveat is that it is possible in theory to use this axiom to write and justify surprising programs. For instance, in Iris$^{⧗}$, the loop "*for* $i = 1$ *to* N *do* () *done*" satisfies the specification $\{\mathsf{True}\}\ —\ \{\mathsf{False}\}$: that is, it is possible to prove that this loop "never ends". As a consequence, this loop also satisfies every specification of the form $\{\mathsf{True}\}\ —\ \{\Phi\}$. On the face of it, this loop would appear to be a valid solution to every programming assignment! In practice, it is up to the user to exhibit taste and to refrain from exploiting such a paradox. In reality, the situation is no worse than that in plain Iris, a logic of partial correctness, where the infinite loop "*while true do* () *done*" also satisfies $\{\mathsf{True}\}\ —\ \{\mathsf{False}\}$.

Another important caveat is that the compiler must in principle be instructed to never optimize ticks away. If, for instance, the compiler was allowed to recognize that the loop "*for* $i = 1$ *to* N *do* () *done*" does nothing, and to replace this loop with a no-op, then this loop, which according to Iris$^{⧗}$ "never ends", would in reality end immediately. We would thereby be in danger of proving that a source program cannot crash unless it is allowed to run for centuries, whereas in reality the corresponding compiled program does crash in a short time. In practice, this danger can be avoided by actually instrumenting the source code with *tick*() instructions and by presenting *tick* to the compiler as an unknown external function, which cannot be optimized away. However, this seems a pity, as it disables many compiler optimizations.

We believe that, despite these pitfalls, time receipts can be a useful tool. We hope that, in the future, better ways of avoiding these pitfalls will be discovered.

10 Related work

Time credits in an affine Separation Logic are not a new concept. Atkey [1] introduces them in the setting of Separation Logic. Pilkiewicz and Pottier [19] exploit them in an informal reconstruction of Danielsson's type discipline for lazy thunks [6], which itself is inspired by Okasaki's work [18]. Several authors subsequently exploit time credits in machine-checked proofs of correctness and time complexity of algorithms and data structures [4,7,21]. Hoffmann, Marmar and Shao [9], whose work was discussed earlier in this paper (§9), use time credits in Concurrent Separation Logic to prove that several concurrent data structure implementations are lock-free.

At a metatheoretic level, Charguéraud and Pottier [4] provide a machine-checked proof of soundness of a Separation Logic with time credits. Haslbeck and Nipkow [8] compare three program logics that can provide worst-case time complexity guarantees, including Separation Logic with time credits.

To the best of our knowledge, affine (exclusive and persistent) time receipts are new, and the axiom $⧖ N \Rrightarrow_\top \mathsf{False}$ is new as well. It is inspired by Clochard *et al.*'s idea that "programs cannot run for centuries" [5], but distills this idea into a simpler form.

Our implementation of thunks and our reconstruction of Okasaki's debits [18] in terms of credits are inspired by earlier work [6,19]. Although Okasaki's analysis assumes a sequential setting, we adapt it to a concurrent setting by explicitly forbidding concurrent operations on thunks; to do so, we rely on Iris nonatomic invariants. In contrast, Danielsson [6] views thunks as a primitive construct in an otherwise pure language. He equips the language with a type discipline, where the type *Thunk*, which is indexed with a debit, forms a monad, and he provides a direct proof of type soundness. The manner in which Danielsson inserts *tick* instructions into programs is a precursor of our tick translation; this idea can in fact be traced at least as far back as Moran and Sands [16]. Pilkiewicz and Pottier [19] sketch an encoding of debits in terms of credits. Because they work in a sequential setting, they are able to install a shared invariant by exploiting the anti-frame rule [20], whereas we use Iris' nonatomic invariants for this purpose. The anti-frame rule does not rule out reentrancy, so they must detect it at runtime, whereas in our case both concurrency and reentrancy are ruled out by our use of nonatomic invariants.

Madhavan *et al.* [15] present an automated system that infers and verifies resource bounds for higher-order functional programs with thunks (and, more generally, with memoization tables). They transform the source program to an instrumented form where the state is explicit and can be described by monotone assertions. For instance, it is possible to assert that a thunk has been forced already (which guarantees that forcing it again has constant cost). This seems analogous in Okasaki's terminology to asserting that a thunk has zero debits,

also a monotone assertion. We presently do not know whether Madhavan *et al.*'s system could be encoded into a lower-level program logic such as Iris$^{\$}$; it would be interesting to find out.

11 Conclusion

We have presented two mechanisms, namely time credits and time receipts, by which Iris, a state-of-the-art concurrent program logic, can be extended with means of reasoning about time. We have established soundness theorems that state precisely what guarantees are offered by the extended program logics Iris$^{\$}$, Iris$^{⧗}$, and Iris$^{\$⧗}$. We have defined these new logics modularly, by composing Iris with a program transformation. The three proofs follow a similar pattern: the soundness theorem of Iris is composed with a simulation lemma about the tick translation. We have illustrated the power of the new logics by reconstructing Okasaki's debit-based analysis of thunks, by reconstructing Clochard *et al.*'s technique for proving the absence of certain integer overflows, and by presenting an analysis of Union-Find that exploits both time credits and time receipts.

One limitation of our work is that all of our metatheoretic results are specific to HeapLang, and would have to be reproduced, following the same pattern, if one wished to instantiate Iris$^{\$⧗}$ for another programming language. It would be desirable to make our statements and proofs generic. In future work, we would also like to better understand what can be proved about the time complexity of concurrent programs that involve waiting. Can the time spent waiting be bounded? What specification can one give to a lock, or a thunk that is protected by a lock? A fairness hypothesis about the scheduler seems to be required, but it is not clear yet how to state and exploit such a hypothesis. Hoffmann, Marmar and Shao [9] have carried out pioneering work in this area, but have dealt only with lock-free data structures and only with situations where the number of competing threads is fixed. It would be interesting to transpose their work into Iris$^{\$}$ and to develop it further.

References

1. Atkey, R.: Amortised resource analysis with separation logic. Logical Methods in Computer Science **7**(2:17) (2011)
2. Birkedal, L.: Lecture 11: CAS and spin locks. `https://iris-project.org/tutorial-pdfs/lecture11-cas-spin-lock.pdf` (Nov 2017)
3. Brookes, S., O'Hearn, P.W.: Concurrent separation logic. SIGLOG News **3**(3), 47–65 (2016)
4. Charguéraud, A., Pottier, F.: Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. Journal of Automated Reasoning (2017)
5. Clochard, M., Filliâtre, J.C., Paskevich, A.: How to avoid proving the absence of integer overflows. In: Verified Software: Theories, Tools and Experiments. Lecture Notes in Computer Science, vol. 9593, pp. 94–109. Springer (2015)

6. Danielsson, N.A.: Lightweight semiformal time complexity analysis for purely functional data structures. In: Principles of Programming Languages (POPL) (2008)
7. Guéneau, A., Charguéraud, A., Pottier, F.: 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, pp. 533–560. Springer (2018)
8. Haslbeck, M.P.L., Nipkow, T.: 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, pp. 155–171. Springer (2018)
9. Hoffmann, J., Marmar, M., Shao, Z.: Quantitative reasoning for proving lock-freedom. In: Logic in Computer Science (LICS). pp. 124–133 (2013)
10. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: securing the foundations of the Rust programming language. PACMPL **2**(POPL), 66:1–66:34 (2018)
11. Jung, R., Krebbers, R., Birkedal, L., Dreyer, D.: Higher-order ghost state. In: International Conference on Functional Programming (ICFP). pp. 256–269 (2016)
12. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming **28**, e20 (2018)
13. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In: Principles of Programming Languages (POPL). pp. 637–650 (2015)
14. Krebbers, R., Jung, R., Bizjak, A., Jourdan, J.H., Dreyer, D., Birkedal, L.: The essence of higher-order concurrent separation logic. In: European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 10201, pp. 696–723. Springer (2017)
15. Madhavan, R., Kulal, S., Kuncak, V.: Contract-based resource verification for higher-order functions with memoization. In: Principles of Programming Languages (POPL). pp. 330–343 (2017)
16. Moran, A., Sands, D.: Improvement in a lazy context: An operational theory for call-by-need. In: Principles of Programming Languages (POPL). pp. 43–56 (1999)
17. Mével, G., Jourdan, J.H., Pottier, F.: Time credits and time receipts in Iris. `https://gitlab.inria.fr/gmevel/iris-time-proofs` (Oct 2018)
18. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1999)
19. Pilkiewicz, A., Pottier, F.: The essence of monotonic state. In: Types in Language Design and Implementation (TLDI) (2011)
20. Pottier, F.: Hiding local state in direct style: a higher-order anti-frame rule. In: Logic in Computer Science (LICS). pp. 331–340 (2008)
21. Zhan, B., Haslbeck, M.P.L.: Verifying asymptotic time complexity of imperative programs in Isabelle. In: International Joint Conference on Automated Reasoning (2018)

A Time Credit Initialization

This appendix provides a sketch of the proof of Lemma 6.

The ghost state that we allocate, and the invariant that we impose, must allow expressing the intuitive idea that the assertion $\$n$ represents the exclusive ownership of an n-unit "share" of the value currently stored in the counter c. This

is an archetypal example of what can be done with Iris, so, assuming familiarity with Iris, the construction is not difficult.

The first step is to allocate a ghost location γ, whose content is an element of the "authoritative monoid" $\textsc{Auth}(\mathbb{N}, +)$ [12]. This step gives us the ability to express two forms of assertions about γ. The "nonauthoritative" assertion $\boxed{\circ m}^{\gamma}$ represents a contribution of m units to γ, whereas the "authoritative" assertion $\boxed{\bullet n}^{\gamma}$ represents a guarantee that the sum of all contributions to γ currently in existence is at most n. We initialize the ghost location γ with the value n, so that, initially, we hold the conjunction $\boxed{\bullet n}^{\gamma} * \boxed{\circ n}^{\gamma}$.

The second step is to install an Iris invariant that ties the ghost location γ and the physical counter c together. Our invariant is as follows:

$$\exists n \quad \left(c \mapsto n * \boxed{\bullet n}^{\gamma} \right)$$

This invariant asserts that, at all times, the value n that is stored in the counter c is at least as high as the sum of all (ghost) contributions $\boxed{\circ m}^{\gamma}$ in existence. By installing this invariant, we lose the points-to assertion $c \mapsto n$ and the authoritative assertion $\boxed{\bullet n}^{\gamma}$. In return, once installed, an Iris invariant becomes shared: every thread can access the invariant for the duration of an atomic instruction. This is exploited (below) in the proof that $tick_c$ satisfies its specification.

The previous two steps, together, form the view shift that appears in the statement of Lemma 6. The next step in the proof of the lemma is to provide a witness for the existential quantification over \$. Our concrete definition of the Iris predicate \$ is as follows:

$$\$n \triangleq \boxed{\circ n}^{\gamma}$$

That is, a time credit is just a (ghost) contribution to γ. The nonauthoritative assertion $\boxed{\circ n}^{\gamma}$, which we still have at hand at this point, allows us to justify the conjunct $\$n$ in the right-hand side of the view shift.

There remains to justify the conjunct *TCIntf* (\$) $tick_c$ in the right-hand side of the view shift. It is easy to check that the three axioms of time credits are satisfied. The law $\$(n_1 + n_2) \equiv \$n_1 * \$n_2$, in particular, follows immediately from the basic law $\boxed{\circ (n_1 + n_2)}^{\gamma} \equiv \boxed{\circ n_1}^{\gamma} * \boxed{\circ n_2}^{\gamma}$. The last point that remains to be checked is the fact that $tick_c$, whose code appears in Figure 4, satisfies the specification of *tick*, which appears in Figure 1. There, the invariant is opened once, to justify the read operation $!\,c$. The precondition \$1 is just $\boxed{\circ 1}^{\gamma}$, which, together with the invariant, guarantees that the result k of this read operation is nonzero. This implies that *oops* (in this case, *crash*) is not executed. The invariant is then opened again, to justify the CAS operation $\texttt{CAS}(c, k, k-1)$. In the case where this operation succeeds, a frame-preserving update is performed so as to change the conjunction $\boxed{\bullet k}^{\gamma} * \boxed{\circ 1}^{\gamma}$ into just $\boxed{\bullet (k-1)}^{\gamma}$ and re-establish the invariant.

B Time Receipt Initialization

This appendix provides a sketch of the proof of Lemma 7.

The ingredients are the same as in our earlier proof of Lemma 6 (§A). This time, we allocate two ghost locations γ and δ, whose values inhabit the monoids $\textsc{Auth}(\mathbb{N}, +)$ and $\textsc{Auth}(\mathbb{N}, \max)$, respectively. We install an invariant that ties this ghost state with the physical state of the counter c:

$$\exists n \quad \Big(c \mapsto (N - n - 1) \ * \ \boxed{\bullet\, n}^{\gamma} \ * \ \boxed{\bullet\, n}^{\delta} \ * \ n < N \Big)$$

Then, we provide concrete definitions for the Iris predicates $\blacklozenge$ and $\boxtimes$:

$$\blacklozenge\, m \triangleq \boxed{\circ\, m}^{\gamma} \qquad \boxtimes\, m \triangleq \boxed{\circ\, m}^{\delta}$$

In short, the counter c stores the number of steps that can still be taken before tick_c executes $\mathit{loop}\,()$ and diverges. This number is always of the form $N - n - 1$, where n is the number of steps that have already been taken. The above invariant guarantees that, at all times, the following properties hold:

1. n is at least the sum of all exclusive receipts $\blacklozenge\, m$ currently in existence;
2. n is at least the value m of any persistent receipt $\boxtimes\, m$ currently in existence.
3. n is less than N.

In light of the above invariant and definitions, it is straightforward to check that the interface *TRIntf* ($\blacklozenge$) ($\boxtimes$) tick_c (Figure 3) is satisfied.

For instance, to check that the law $\boxtimes\, N \Rrightarrow_{\top} \mathsf{False}$ holds, it suffices to open the invariant, confront the assertions $\boxed{\circ\, N}^{\delta}$ and $\boxed{\bullet\, n}^{\delta}$ to obtain $N \leq n$, and combine this inequality with the inequality $n < N$ so as to derive a contradiction.

The fact that tick_c (Figure 4) satisfies the specification in Figure 3 is also easily verified. In contrast with the case of time credits, where k could not be zero and *oops* was never executed, in the present case, k *can* be zero, and *oops* *can* be executed. Fortunately, here, *oops* is *loop*, whose postcondition is False, so, in the case where k is zero, tick_c trivially satisfies its specification. In the case where k is nonzero, we get $N - n - 1 > 0$, that is, $n + 1 < N$, so that, after decrementing c, which amounts to incrementing n, we are able to re-establish the invariant.