# The Essence of Monotonic State

Alexandre Pilkiewicz François Pottier
INRIA
{Alexandre.Pilkiewicz,Francois.Pottier}@inria.fr

## Abstract

We extend a static type-and-capability system with new mechanisms for expressing the promise that a certain abstract value evolves monotonically with time; for enforcing this promise; and for taking advantage of this promise to establish non-trivial properties of programs. These mechanisms are independent of the treatment of mutable state, but combine with it to offer a flexible account of "monotonic state".

We apply these mechanisms to solve two reasoning challenges that involve mutable state. First, we show how an implementation of thunks in terms of references can be assigned types that reflect time complexity properties, in the style of Danielsson (2008). Second, we show how an implementation of hash-consing can be assigned a specification that conceals the existence of an internal state yet guarantees that two pieces of input data receive the same code if and only if they are equal.

***Categories and Subject Descriptors*** F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

***General Terms*** Languages, Theory

***Keywords*** types, capabilities, specification, hidden state, monotonic state, amortized complexity, type-based complexity-checking, thunks, hash-consing

## 1. Introduction

This paper presents novel type-theoretic mechanisms and techniques for exploiting *monotonicity* in establishing properties of programs that manipulate mutable, heap-allocated data.

***Two traditional modes of dealing with state*** How do the type systems in the literature deal with mutable state? Do they allow the type of mutable data to evolve over time? How do they keep track of this type? How do they deal with aliasing? Although there is a large variety of such systems, two modes seem to prevail:

1. *invariable, duplicable types; uncontrolled aliasing;*

   In most mainstream programming languages, including Java, Haskell, and ML, types are *invariable:* the type of a mutable object is fixed at allocation time, and cannot change with time. In return for this lack of expressiveness comes a gain in flexibility: type information can be *duplicated,* and aliasing can remain *uncontrolled,* without risking unsoundness.

2. *variable, linear types; controlled aliasing;*

   There are systems where the type of a mutable object is permitted to *vary,* in an arbitrary way, during the object's lifetime. The price to pay for this expressiveness is that type information must be *linear,* and aliasing must be *controlled.*

The traditional representatives of the latter mode are systems of *linear types.* By requiring that there exist at most one pointer to an object, these systems conflate the control of ownership and the control of aliasing. Their modern descendants [3, 4, 29] offer greater flexibility by using *linear capabilities* to control ownership and *regions* to control aliasing. In these systems, there may exist multiple pointers to an object. Each object, however, must belong to a region, and access to each region is governed by a capability. A capability can be thought of as a unique token that represents the ownership of a region.

It is sound for the two modes to co-exist: the literature presents numerous systems that mix them. They are most often both viewed as primitive. More economically, the second author has argued in earlier work [24] that the former mode (invariable, uncontrolled objects) can be implemented in terms of the latter mode (variable, controlled objects), with the help of a primitive mechanism for *hiding* a capability within a lexical scope.

This mechanism, known as the *anti-frame rule,* makes it possible to organize the implementation of a mutable object in such a way that aliasing and ownership are controlled within the implementation, but need not be controlled outside of it. As far as clients are concerned, objects appear to be ordinary (stateless, duplicable) values. Our implementations of monotonic counters (§2) and thunks (§5) are carried out in this style.

***Monotonic state*** The above two modes of dealing with state can be informally summarized in the semblance of a security policy, that is, by answering the following two questions:

> Who is permitted to *change* the type of an object?
>
> Who is permitted to *know* the type of an object?

Here, to "know" the type of an object means to record this type at some point in time, and to later use the object at the type that was recorded. The answers are, very roughly, as follows:

1. In the first mode, *nobody* is allowed to change the type of an object; *everybody* is allowed to know it.
2. In the second mode, *only the owner* is allowed to change or know the type of an object.

The purpose of this paper is to study a third mode, which strikes a different compromise. It has, at its heart, a notion of *monotonicity.* This mode can be described, again very roughly, as follows:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*TLDI'11,* January 25, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0484-9/11/01...$10.00

3. Only the owner of an object is allowed to change its type, and, furthermore, *only in a monotonic manner,* so that types get "better", in a certain sense, with time. In return for this constraint, *everybody* is allowed to record the type of an object, with the understanding that, by the time this information is exploited, the object may well have a "better" type than the one that was recorded.

The choice of the word "better" is meant to suggest some ordering relation. There is no connection between this ordering and the standard subtype ordering: we do not require types to become more precise with time. The choice of an appropriate ordering, on a case-by-case basis, is in the hands of the programmer.

In this third mode, which we dub "monotonic state", changing the type of an object is a restricted operation: control of ownership and aliasing is still required. However, recording and exploiting the type of an object are unrestricted operations: to a certain extent, this control is relaxed.

***Contributions*** The existence of this interesting compromise has been pointed out in the literature (§7), so it does not constitute, in itself, a contribution of this paper. Our contribution is twofold:

> *Contribution 1.* We study monotonic state in a standard type-theoretic setting and explain it in terms of a handful of primitive mechanisms.

The new mechanisms that we propose are simple and elegant (so we claim, that is!), because they are concerned purely with the essence of the interplay between monotonicity and linearity.

In order to define these mechanisms, we need, as our substrate, a type system equipped with capabilities (both linear and non-linear ones) and with logical assertions (which we view as a particular species of non-linear capabilities). In order to present meaningful applications, we need more, namely: higher kinds, regions, mutable state, and hidden state. This list of extra features is perhaps intimidating, but is in principle independent of the topic of this paper.

As is evident from the verbosity of our code (§4), our calculus is intended to serve as a kernel language, into which more palatable surface languages equipped with some form of monotonicity can be encoded.

> *Contribution 2.* As a non-trivial application, we show how Danielsson's system [6] for analyzing the complexity of pure, lazy programs can be programmed up as a library in our imperative, call-by-value setting.

We show how a simple implementation of suspensions, or thunks, can be ascribed a signature that allows a client of the "thunk" abstraction to reason about amortized execution time. This was a challenge because thunks must be ascribed precise types (the type *thunk* carries an integer index, which represents a time bound), yet there must be no control of ownership or aliasing over thunks.

Our result provides a foundational explanation of Danielsson's system: while Danielsson gives a direct proof of the soundness of his system, we encode it into a richer, lower-level system, which allows reasoning about the amortized complexity of imperative ML programs.

***Caveat emptor*** We do not, at this time, have a proof of type soundness for the full type system exploited in this paper. We build on two earlier papers by Charguéraud and Pottier [3] and by Pottier [24]. In as-yet-unpublished work, the second author has established the soundness of the combination of these papers: this took a 20000 line Coq proof. In order to justify the ideas presented in this paper, this proof must be extended with support for embedding logical assertions within capabilities, support for fates and predictions, and support for time credits. A generic treatment of monotonicity is built into the current machine-checked proof, and is exploited in order to justify singleton and group regions. It is hoped that this machinery can also justify fates and predictions.

```
1 let mk () = ref 0          – allocates a fresh counter
2 let read r = !r            – reads a counter
3 let inc r = r := !r + 1; r – increments a counter and returns it
```

**Figure 1.** An implementation of monotonic counters

```
1 type sc : VAL
2 val mk : unit → sc
3 val read : sc → int (0 ≤ .)
4 val inc : sc → sc
```

**Figure 2.** A signature for simple counters

```
1 type tc : SNG → ℤ → CAP
2 val mk : unit → ∃σ. ([σ] ∗ tc σ 0)
3 val read : ∀i, σ. [σ] ∗ tc σ i → int i ∗ tc σ i
4 val inc : ∀i, σ. [σ] ∗ tc σ i → [σ] ∗ tc σ (i + 1)
```

**Figure 3.** A signature for tracked counters

```
1 type mc : ℤ → VAL
2 val mk : unit → mc 0
3 val read : ∀i. mc i → int (i ≤ .)
4 val inc : ∀i. mc i → mc (i + 1)
```

**Figure 4.** A signature for monotonic counters

***Road map*** The paper begins with a challenge (§2). After presenting a 3-line implementation of *monotonic counters* (Figure 1) as well as a plausible signature for them (Figure 4), we ask: *in which known type systems can it be checked that this code satisfies this signature?* We argue that the answer appears to be: *none.* In order to address this challenge, we suggest introducing new type-theoretic mechanisms, called *fates* and *predictions*. After informally presenting these mechanisms (§3), we explain how, in combination with a number of orthogonal, pre-existing features, they allow monotonic counters to be type-checked (§4). As a more striking application, we show that they allow transporting Danielsson's analysis of thunks [6] from a purely functional, call-by-need language to an imperative, call-by-value setting (§5). We also present an application to hash-consing (§6). We conclude with a discussion of related work (§7). A summary of the core of our system (§A) appears in an appendix.

## 2. A challenge: monotonic counters

A *monotonic counter* is an integer counter that offers two operations: *read*, which returns the current value of a counter, and *inc*, which increments a counter, and returns its argument. (The reason for this design choice becomes apparent later on.) A constructor function, *mk*, allows creating fresh counters. An untyped implementation of monotonic counters appears in Figure 1. It is trivial: a counter is just an integer reference.

Monotonic counters pose a simple, yet challenging problem. In the following, we present three natural signatures for them, which correspond to the three modes of dealing with state that were reviewed in the introduction (§1). While the implementation of monotonic counters satisfies the first two signatures, it is not clear how to argue that it also satisfies the last one.

***Parenthesis: indexed types*** In order for the three signatures to differ in interesting ways, we need precise types: that is, we need a

type to be able to express an assertion about the integer value of a counter. To this end, we allow types to be parameterized with integer indices, in the style of Xi's Dependent ML [32].

Let us briefly review what this means. We write $\mathbb{Z}$ for the kind of integer indices. We use a singleton type, *int* $i$, whose unique inhabitant is the integer value $i$. The addition operator has type $\forall ij.\mathit{int}\ i \to \mathit{int}\ j \to \mathit{int}\ (i + j)$. The traditional, unparameterized type *int* can be viewed as sugar for $\exists i.\mathit{int}\ i$.

We also use the type *int* $(i \leq .)$, whose inhabitants are the integer values *greater than or equal to* $i$. This type can be defined as $\exists j.(\mathit{int}\ j * \langle i \leq j \rangle)$, where we use *existential quantification* over an integer index $j$ and a *conjunction* of a type, *int* $j$, and a *proposition* about indices, $\langle i \leq j \rangle$. We note that, if $i \leq j$ holds, then *int* $(j \leq .)$ is a *subtype* of *int* $(i \leq .)$. This fact intuitively reflects the set-theoretic inclusion $[j, \infty) \subseteq [i, \infty)$. Here, it can be derived from the definition of *int* $(i \leq .)$.

***A signature for simple counters*** In the first signature (Figure 2), the operations *read* and *inc* have simple types. The type *sc* of *simple counters* is abstract: it is internally defined as *ref* $(\mathit{int}\ (0 \leq .))$, where *ref* is ML's reference type constructor. (We write VAL for the kind of ordinary types, that is, types that classify values. Other kinds appear later on.) This definition encodes the invariant that the value of a counter is a nonnegative integer. This allows *read* to have codomain *int* $(0 \leq .)$ rather than just *int*.

Because *sc* is an abstract type, the reference *r* is accessible only through *read* and *inc*. This guarantees that counters are monotonic, that is, their value can only grow with time. However, this is only an *informal* guarantee. The type-checker is unaware of this property, which it can neither check nor exploit.

This signature is imprecise: it does not reflect the fact that *inc* increments its argument. In the setting of an ML-like type system, despite the availability of indexed types, this seems to be the best signature that one can express *and implement.*

***A signature for tracked counters*** In a type-and-capability system [3, 4, 29], access to a reference is governed by a linear capability, so that type-varying updates (also known as *strong updates*) are sound. In Charguéraud and Pottier's notation [3], for instance, a counter inhabits a *singleton region* $\sigma$. The type of the reference *r* is written $[\sigma]$, read "at $\sigma$", and means that *r* is the single inhabitant of the region $\sigma$. Access to *r* is governed by a capability of the form $\{\sigma : \mathit{ref}\ (\mathit{int}\ i)\}$, where the integer index $i$ represents the current state of the counter. This capability must be presented when the reference is read or written. A *read* operation returns an identical capability. An *inc* operation returns an updated capability, $\{\sigma : \mathit{ref}\ (\mathit{int}\ (i + 1))\}$. This is a strong update.

In such a system, a signature for *tracked counters* can be defined and implemented (Figure 3). There, $\mathit{tc}\ \sigma\ i$ is an *abstract capability*, analogous to an *abstract predicate* in separation logic, which hides the fact that the implementation of a counter is an integer reference. It is internally defined as $\{\sigma : \mathit{ref}\ (\mathit{int}\ i)\}$. (We write SNG for the kind of singleton regions and CAP for the kind of capabilities.)

The function *mk* has codomain $\exists\sigma.([\sigma] * \mathit{tc}\ \sigma\ 0)$. This means that *mk* returns: (i) a region $\sigma$; (ii) a value, of which nothing is known, except it inhabits $\sigma$; and (iii) a capability $\mathit{tc}\ \sigma\ 0$. This capability guarantees that *the inhabitant of* $\sigma$ *is a counter in state 0.* At the same time, it represents the *ownership* of this counter, that is, the right to pass this counter as an argument to *read* and *inc*.

Accordingly, the functions *read* and *inc* require not only a value of type $[\sigma]$, but also a capability $\mathit{tc}\ \sigma\ i$, which serves as a proof of ownership and indicates that the counter is initially in state $i$. Out of this, *read* produces a pair of *the integer* $i$ and an unmodified capability, while *inc* produces a pair of an unmodified value and an updated capability $\mathit{tc}\ \sigma\ (i + 1)$.

This interface is strong: thanks to capabilities, the state of a counter is *tracked* in a precise manner. Unfortunately, there is a price to pay: this interface imposes restrictions on aliasing and ownership. The fact that $\mathit{tc}\ \sigma\ i$ is a *linear* capability means that every counter must have a unique owner. This effectively restricts the use of tracked counters to linear data structures, that is, data structures without sharing.

***A signature for monotonic counters*** Is it intuitively sound to make an assertion about the value of a counter without imposing any restriction on aliasing or ownership? Yes. Because the value of a counter *increases* with time, it is possible for a client to maintain a sound *under-approximation* of it. This is permitted by the signature in Figure 4, where the type *mc* of *monotonic counters* is now integer-indexed.

What is the intuitive meaning of the type $\mathit{mc}\ i$? Certainly, the index $i$ cannot reflect the *exact* internal state of the counter, but must represent a *lower bound*. Indeed, if *x* has type $\mathit{mc}\ i$, then, after an application of *inc* to *x*, the variable *x* still has type $\mathit{mc}\ i$, even though the internal state of the counter *x* has just changed. More deeply, if some variable *y* also has type $\mathit{mc}\ i$, then, after this application, *y* still has type $\mathit{mc}\ i$. Yet, because *x* and *y* may be aliases, the state of the counter *y* could have just changed as well. So, the intuitive interpretation of $\mathit{mc}\ i$ is:

> *mc* $i$ *is a type of monotonic counters whose internal state is at least* $i$.

It is straightforward, although not quite trivial, to *informally* convince oneself that this signature is sound. Consider a counter *x*, of type $\mathit{mc}\ i$, where $i$ represents a lower bound on *x*'s internal state $j$, that is, $i \leq j$ holds. Invoking *read* returns $j$, which has type *int* $(i \leq .)$, as advertised. Invoking *inc* updates the internal state to $j + 1$, of which $i + 1$ is a lower bound, so it is sound for *inc* to advertise a return type of $\mathit{mc}\ (i + 1)$. Furthermore, $i$ remains a lower bound of $j + 1$, so it is sound to continue using $x$ at type $\mathit{mc}\ i$.

An unusual feature of this signature is that, even though *inc* returns its argument, it ascribes a *more precise* type to its result than to its argument! (To permit such a type refinement is the reason why we decided that *inc* should return its argument.) This allows keeping track of a lower bound on the internal state of a counter. For example, *read (inc (mk ()))* has type *int* $(1 \leq .)$. This may seem a tiny achievement; yet, such a feature is essential in our encoding of Danielsson's analysis of thunks. There (§5), the function *pay* is analogous to *inc*, in that it also returns its argument at a better type.

The signature of monotonic counters is stronger than the simple signature of Figure 2. (Indeed, the latter can be implemented in terms of the former, by defining *sc* as $\exists i.(\mathit{mc}\ i * \langle 0 \leq i \rangle)$.) On the other hand, it is incomparable with the signature of tracked counters (Figure 3). The latter allows keeping exact track of the state of a counter, while the former only allows keeping track of a lower bound. On the other hand, the latter comes with restrictions on aliasing and ownership, while the former does not.

***The challenge*** ML, extended with indexed types, allows writing down the signature in Figure 4, and allows clients of this signature to be written and type-checked. However, it does not allow checking that the code in Figure 1, augmented with a suitable definition of *mc*, satisfies this signature. Neither do the type-and-capability systems that we are aware of. In the following, we extend such a system with new mechanisms (§3) that address this challenge (§4).

## 3. Fates and predictions

Perhaps the most obvious approach to addressing the challenge would be to build monotonicity directly into references, by making *monotonic references* a primitive notion. We quickly abandoned this approach, however, for several reasons.

First, monotonicity is sometimes not a property of a single reference cell, but of a composite data structure. For example, in our application to hash-consing (§6), a hash table encodes a mathematical function whose graph grows with time (with respect to set-theoretic inclusion, $\subseteq$). A primitive notion of a monotonic reference would not support these non-trivial applications.

Second, requiring monotonicity to hold between *any two points* in time would be too inflexible. It is desirable to allow monotonicity to be *temporarily* violated, as long as this remains in some sense *unobservable* from the outside. This is again evident in advanced applications (§6): because updates to a complex data structure are not atomic, monotonicity does not make sense while an update is in progress. Again, perhaps one could build this flexibility into a set of *ad hoc* typing rules for monotonic references; here, however, it is obtained via the anti-frame rule.

***Fates*** Instead of monotonic references, we introduce *monotonic ghost variables,* or *fates,* for short. A fate can be thought of as a mutable memory location that does not exist at runtime and whose value can only grow with time. A fate is controlled by a linear capability, just as if it did exist in the runtime heap.

Fates are not tied to references or objects. Furthermore, as we will see, there is no need for fates to temporarily disobey monotonicity. For these reasons, the rules that govern fates are simple and lightweight. In a sense, fates distill the *essence* of monotonicity: they describe the interplay between linearity and monotonicity, and nothing else.

In order to express the fact that the value of a certain fate reflects the state of a certain reference, or of an entire data structure, one sets up an explicit *invariant*. The anti-frame rule [24] offers a mechanism for this purpose. The invariant, a capability, is invisible (and must hold) outside of a certain lexical scope, and is visible (and can be temporarily violated) within that scope. This approach, it turns out, addresses all three previously mentioned issues.

***Types and laws*** Because a fate does not exist at runtime, it does not contain a *programming language* value (e.g., a machine integer, a $\lambda$-abstraction, a memory location, etc.), but a *mathematical* value (e.g., an integer, a set of integers, a function of integers to sets of integers, etc.). In other words, the *type* $\mathbf{T}$ of a fate is a type of the ambient logic. Any sound logic that is powerful enough for the intended application can serve as the ambient logic. For instance, in the case of monotonic counters, Presburger arithmetic would be sufficiently expressive and would permit decidable type-checking. For more advanced examples, the ambient logic could be the Calculus of Inductive Constructions, so that our type-checker ships proof obligations to the Coq proof assistant. (In this paper, for the sake of uniformity, we place ourselves in this case.) A soundness proof for our system should in principle be independent of the choice of an ambient logic.

A fate of type $\mathbf{T}$ must be equipped with a *law* $\mathbf{R}$, that is, a preorder over $\mathbf{T}$. The law defines what it means for the value of a fate to *grow*. Each fate can have its own type $\mathbf{T}$ and law $\mathbf{R}$, which are fixed when it is created. A law need not be a total order: some applications of fates involve partial orders (§6).

In our running example, a fate is used to reflect the value of a monotonic counter. Its type is $\mathbb{Z}$, the type of the mathematical integers; its law is the ordering $\leq$ over $\mathbb{Z}$. For simplicity, we fix this special case in the following explanations.

***Creating a fate*** Since fates do not exist at runtime, none of the primitive operations over fates has a runtime effect. These operations can be thought of as type annotations, and are erased before the program is executed. More precisely, we will view them as *subsumption axioms*. For instance, the creation of a fresh fate is permitted by the following axiom:

$$\emptyset <: \exists\varphi.\{\varphi : i\}$$

Such an axiom means that the capability on the left-hand side can be transformed into the capability on the right-hand side. Here, out of nothing, one obtains a fresh *fate* $\varphi$, together with a *capability*, written $\{\varphi : i\}$, which represents both the *ownership* of the fate $\varphi$ and the *knowledge* that the current value of the fate is $i$. (Here, $i$, the initial value of the fate, can be any element of $\mathbb{Z}$.) This capability is *linear:* a fate has at most one owner.

This axiom can be compared with the type of the primitive operation for allocating a fresh reference [3]:

$$\mathit{ref} : \tau \longrightarrow \exists\sigma.([\sigma] * \{\sigma : \mathit{ref}\,\tau\})$$

When provided with an initial value of type $\tau$, this operation allocates a fresh reference cell in the heap, and returns: (i) a singleton region $\sigma$; (ii) a memory location, of type $[\sigma]$, the single inhabitant of this region; (iii) a capability $\{\sigma : \mathit{ref}\,\tau\}$, which represents both the ownership of the region and the knowledge of its type. Creating a fate is analogous to allocating a reference, in that it creates a fresh name and produces a capability. It differs in that no runtime values are involved, and the heap is unaffected.

***Updating a fate*** The owner of a fate is free to update it at any time, provided the new value is provably related to the current value. This is expressed as follows:

$$\{\varphi : i\} * \langle i \leq j\rangle <: \{\varphi : j\}$$

That is, the value of the fate can be changed from $i$ to $j$, provided the proposition $i \leq j$ holds. If $\mathbf{P}$ is a proposition of the ambient logic, then $\langle\mathbf{P}\rangle$ is a *duplicable* capability, a witness for $\mathbf{P}$. This axiom is analogous to the *strong update* of a reference [3], in that a capability is consumed and a potentially different capability is produced. It differs in that no runtime values are involved and updates are required to be monotonic. The capability $\langle i \leq j\rangle$ on the left-hand side can be thought of as a *proof obligation*.

***Tying a fate to a piece of runtime state*** The internal state of a monotonic counter consists of an integer reference, which inhabits a region $\sigma$, and of a fate $\varphi$. How do we express the fact that the value of the fate is kept synchronized with the content of the reference? The answer is simple: the capabilities that respectively govern the reference and the fate must *share* an integer index. The composite capability:

$$\exists i.(\{\sigma : \mathit{ref\ int}\ i\} * \{\varphi : i\})$$

not only represents the ownership of both the reference and the fate, but also indicates that they share a common value $i$. By existentially abstracting over $i$, we make this capability suitable for use as an *invariant* that remains true even as the counter is incremented. In the following (§4), this invariant is *hidden*, so that it is invisible to a client of the monotonic counter abstraction. It is visible only within the implementation of monotonic counters, where it can be temporarily broken, provided it is restored before control is returned to the client. One cannot forever escape one's fate!

***Making predictions*** We are still missing a piece of the puzzle. A capability $\{\varphi : i\}$ represents, *at the same time,* the ownership of a fate and an assertion about its value. In other words, so far, only the owner of a fate can assert a proposition about its value. However, in our planned application to monotonic counters, a client must be allowed to assert that the state of a counter is at least $i$, for a certain integer $i$, even though the client does not own the counter. How could we solve this difficulty?

This is where monotonicity comes into play. Because a fate is constrained to evolve in a monotonic manner, its current value serves as a *prediction* of its future values: if the current value is $i$, it is safe

to assert that any future value $j$ satisfies $i \leq j$. *Making* such a prediction requires knowledge of the current value $i$, which in turn requires ownership of the fate. However, once such a prediction is made, it can never be contradicted by updating the fate, so it remains valid forever. For this reason, a prediction can be considered a *duplicable capability,* separate of the linear capability that governs of the fate. A prediction is created as follows:

$$\{\varphi : i\} <: \{\varphi : i\} * \langle\varphi : i\rangle$$

We write $\langle\varphi : i\rangle$ for the prediction that *the value of $\varphi$ will always be at least $i$.* It can also be understood as an *observation* of some state of $\varphi$ that is sufficient to guarantee that the value of $\varphi$ will always be at least $i$. Both points of view are useful, so, in the following, we make use of both of the words "prediction" and "observation". The above axiom states that the owner of a fate can, at any time, produce an observation of the fate's current state or, equivalently, a prediction of its future states.

How does this solve the difficulty with which we were faced? A prediction $\langle\varphi : i\rangle$ is *non-linear*. It does not represent the ownership of the fate $\varphi$, yet it does represent an assertion about its state. So, *it is now possible to make an assertion about a piece of state that one does not own.* Our implementation of monotonic counters (§4) retains ownership of the fate, but creates predictions that it passes to its clients.

***Exploiting predictions*** Predictions—at least in our system!—are true and remain so forever. As a result, comparing an old prediction with the present state allows knowledge to be gained: one learns that the present state conforms to what was predicted. This is stated as follows:

$$\langle\varphi : i\rangle * \{\varphi : j\} <: \langle i \leq j\rangle * \{\varphi : j\}$$

If it was once predicted that the state would always be at least $i$, and if the present state is $j$, then $i \leq j$ must hold. In other words, exploiting a prediction produces a new logical fact, which can later be used in a proof.

As explained earlier, our implementation of monotonic counters (§4) uses the internal invariant: $\exists i.(\{\sigma : \mathit{ref\ int}\ i\} * \{\varphi : i\})$. Once unpacked, this becomes: $\{\sigma : \mathit{ref\ int}\ j\} * \{\varphi : j\}$, where a fresh integer index $j$ represents the current state of the reference and the fate. Now, imagine that a client presents us with an old prediction, of the form $\langle\varphi : i\rangle$. This prediction must have been created earlier within the implementation of monotonic counters, handed to a client, carried around for a while by the client, and is now being presented back to us. By exploiting it, we learn $i \leq j$, that is, we learn that the current state is as good as or superior to what the client expects. In its absence, nothing would be known about the current state, since $j$ is just an abstract integer index.

Since the reference has type *ref int* $j$, reading it yields a value of type *int* $j$, which, thanks to the proposition $i \leq j$, is a subtype of *int* $(i \leq .)$. In other words, the *read* operation produces a value of type *int* $(i \leq .)$, provided the client hands it the prediction $\langle\varphi : i\rangle$.

***Weakening predictions*** A prediction can be weakened to one that permits more numerous potential futures:

$$\langle i \leq j\rangle * \langle\varphi : j\rangle <: \langle\varphi : i\rangle$$

This is used in the implementation of monotonic counters (§4). In the *inc* operation, after the reference and the fate have been updated from $j$ to $j + 1$, the following capability is available:

$$\{\sigma : \mathit{ref\ int}\ (j + 1)\} * \{\varphi : j + 1\}$$

At this point, we wish to create a new prediction, based on the new state, and return it to the client. So, we construct the prediction $\langle\varphi : j + 1\rangle$. This prediction is valid; however, it cannot be returned to the client, because it mentions $j$, a variable that was introduced by unpacking our existentially quantified invariant. The client knows nothing about $j$, the true current state of the counter; it only knows about $i$, the value that it has observed in the past. Thus, we *weaken* the prediction $\langle\varphi : j + 1\rangle$ by changing it into $\langle\varphi : i + 1\rangle$. This is valid, because we know $i \leq j$, which implies $i + 1 \leq j + 1$. The weakened prediction can now be handed back to the client. In summary, *inc* produces the prediction $\langle\varphi : i + 1\rangle$, provided the client hands it the prediction $\langle\varphi : i\rangle$.

***Joining predictions*** If it has been predicted that the value of $\varphi$ will remain above $i$ and above $j$, then its (unknown) current value must be a common upper bound of $i$ and $j$. Thus, it is safe to produce a new prediction of *some* common upper bound $k$ of $i$ and $j$. Ownership of the fate is not required.

$$\langle\varphi : i\rangle * \langle\varphi : j\rangle <: \exists k.(\langle i\ \mathbf{R}\ k\rangle * \langle j\ \mathbf{R}\ k\rangle * \langle\varphi : k\rangle)$$

The need for this axiom arises when $\mathbf{R}$ is not a total order. For instance, it plays a key role in our application to hash-consing (§6). There, it is not even the case that every two elements $i$ and $j$ admit a common upper bound with respect to $\mathbf{R}$. Even in this case, the axiom is sound, and all the more useful.

## 4. Application: monotonic counters

We now put everything together and explain how to typecheck monotonic counters. The code, which appears in Figure 5, consists of four definitions, for *mc*, *mk*, *read*, and *inc*.

***Surface syntax*** Whereas this paper is concerned only with type checking in a core calculus, our illustrative examples are expressed in a plausible but informal sugared syntax, with some degree of inference of types and capabilities. In particular, the keywords **let fate**, **set fate**, **make**, and **exploit** are used to create (and name) a fate, update a fate, make (and weaken) a prediction, and exploit a prediction, respectively. We use **let cap** to define an abbreviation for a capability. We use **pack cap** and **unpack cap** to introduce and eliminate existentially quantified capabilities. We use **got cap** to assert that a certain capability is held: this is a machine-checkable comment. The construct "**hide** $I = C$ **outside of** $t$" , proposed by the second author in earlier work [24], has the double effect of introducing $I$ as an abbreviation for the capability $C$ within the term $t$, and of making the capability $I$ invisible outside of the term $t$. None of these constructs has a runtime effect: they are used by the type-checker only. In addition, we use ordinary **pack** and **unpack** constructs to introduce and eliminate existential types.

***Definition*** What is, *really,* a monotonic counter with index $k$? According to the definition of *mc* (lines 1–5), it is:

> for *some* abstract notion of an observation of an integer (line 2),
>
> a pair of two functions, or *methods*, where the *read* method (line 3) accepts an observation of *any* integer $i$ and produces an integer value that is no less than $i$, and the *inc* method (line 4) expects an observation of *any* integer $i$ and produces an observation of $i + 1$,
>
> packaged together with an observation of $k$ (line 5).

The parameter $k$ occurs *only* in the last component. It does not occur in the type of the methods! In other words, over the lifetime of a monotonic counter, observations of the counter in various states are created, but the vector of methods remains unchanged.

Because *obs* is a non-linear capability, *mc* $k$ is a non-linear type, as desired. (We write DCAP for the kind of non-linear, or *duplicable*, capabilities. It is a sub-kind of CAP.) This is a key point.

***Construction*** How does one construct a monotonic counter? Let us now review the definition of *mk* (lines 7–47).

```
1  type mc k =                                  – the type of monotonic counters
2    ∃obs : ℤ → DCAP.
3      ((∀i. unit ∗ obs i → int (i ≤ .)) ×
4       (∀i. unit ∗ obs i → unit ∗ obs (i + 1))) ∗
5      obs k
6
7  let mk : unit → mc 0 =
8  λ().
9    let fate φ : FATE ℤ (≤) = 0 in
10   got cap { φ : 0 };                         – we own the fate, in state 0
11   let cap obs i = ⟨ φ : i ⟩ in               – a notation for observations
12   make obs 0;                                – make an initial observation
13   let σ, (r : [σ]) = ref 0 in                – allocate a fresh reference
14   got cap { σ: ref int 0 };                  – we own the reference, in state 0
15
16   let methods :                              – build a vector of methods
17     (∀i. unit ∗ obs i → int (i ≤ .)) ×
18     (∀i. unit ∗ obs i → unit ∗ obs (i + 1)) =
19
20     hide I =                                 – I is visible only to the methods
21       ∃j. ({σ : ref int j} ∗ {φ : j})
22     outside of
23     pack cap I;                              – this establishes the invariant
24
25     let read : ∀i. unit ∗ obs i ∗ I → int (i ≤ .) ∗ I =
26     λ().
27       let j = unpack cap I in
28       got cap obs i ∗ {σ : ref int j} ∗ {φ : j};
29       exploit obs i;                         – this yields i ≤ j
30       let c : int j = !r in
31       let c : int(i ≤ .) = c in              – a subsumption step
32       pack cap I;
33       c
34     and inc : ∀i. unit ∗ obs i ∗ I → unit ∗ obs (i + 1) ∗ I =
35     λ().
36       let j = unpack cap I in
37       got cap obs i ∗ {σ : ref int j} ∗ {φ : j};
38       exploit obs i;                         – this yields i ≤ j
39       r := !r + 1;
40       set fate φ := j + 1;                   – a monotonic update
41       got cap obs i ∗ {σ : ref int (j + 1)} ∗ {φ : j + 1};
42       make obs (i + 1);                      – permitted, since i + 1 ≤ j + 1
43       pack cap I                             – the witness is j + 1
44     in
45     (read, inc)                              – this is the vector of methods
46   in got cap obs 0;                          – still got that initial observation
47   pack methods as mc 0
48
49 let read : ∀k. mc k → int (k ≤ .) =
50 λ(c : mc k).
51   let obs, methods = unpack c in
52   let (read, _) = methods in
53   got cap obs k;
54   read ()
55
56 let inc : ∀k. mc k → mc (k + 1) =
57 λ(c : mc k).
58   let obs, methods = unpack c in
59   let (_, inc) = methods in
60   got cap obs k;
61   inc ();
62   got cap obs (k + 1);
63   pack methods as mc (k + 1)
```

**Figure 5.** Monotonic counters

In the prologue (lines 9–14), a fresh fate $\varphi$ and reference $r$ are created. The notation *obs* $i$ is introduced for an observation of the fate $\varphi$ in state $i$, and an initial observation *obs* 0 is made.

Next, the methods that read and increment the counter are defined (lines 16–45). There are *two views* of these methods: an internal view, where an invariant $I$ occurs in the type of the methods, and an external view, where $I$ no longer appears. The invariant $I$, defined on line 20, requires $\varphi$ and $r$ to share a common (but unspecified) value $j$. It is immediately established (line 23).

The methods *read* and *inc* are first defined *within* the scope of the **hide** construct. As a result, they have access to $I$, and must preserve it: the capability $I$ appears in their argument and result types (line 25 and line 34). This is the *internal view* of the methods.

Next, a pair of *read* and *inc* is constructed (line 45) and returned outside of the **hide** construct. It is bound, there, to the name *methods* (line 16). The effect of **hide** [24] is to consume the capability $I$ and to remove the four occurrences of $I$ in the types of *read* and *inc*, so that $I$ does not appear in the type of *methods* (line 16). This is the *external view* of the methods.

The body of *read* is polymorphic in the integer index $i$, which appears in a *client-provided* observation *obs* $i$ (line 25), and in the integer index $j$, which is obtained by unpacking the existentially quantified invariant $I$ (line 27). In short, $i$ represents some past state of the counter, which was observed by the client, while $j$ represents its current state. As explained earlier, exploiting these facts yields the proposition $i \leq j$. This proposition is used to justify a subsumption step that converts the type of $c$, the current value of the counter, from *int* $j$ to *int* $(i \leq .)$ (line 31).

Similarly, the body of *inc* is polymorphic in $i$ and $j$, and exploits a client-provided observation to establish $i \leq j$. The reference $r$ and the fate $\varphi$ are incremented. Between these updates, $r$ is in state $j + 1$, while $\varphi$ is still in state $j$. This is fine: we are free to break the invariant $I$, provided it is restored before control is returned to the client. This is done on line 43. There, the use of **pack cap** causes us to *forget* that the current state is $j + 1$. *Before* forgetting this precious information, we make an observation of $j + 1$, and immediately weaken it to an observation of $i + 1$ (line 42). This weakening step is valid because $i + 1 \leq j + 1$ provably holds. In the end, the observation *obs* $(i + 1)$ is returned to the client. This is made explicit in the type of *inc* (line 34).

Because *read* and *inc* are polymorphic in $i$, they adapt to the level of knowledge that the client has acquired, and is able to exhibit. A greater value of $i$ means a stronger observation is provided by the client, and, accordingly, a stronger result is returned by *read* and *inc*. The current state $j$ does not (and cannot) occur in the types of *read* and *inc*, since it is existentially quantified within $I$.

In the epilogue (lines 46–47), we package up the *methods* vector together with the initial observation *obs* 0 and abstract away the definition of *obs*. In so doing, we not only obtain a value of type *mc* 0, as desired, but also ensure that $\varphi$ does not escape its scope: recall that *obs* $i$ is just a notation for $\langle \varphi : i \rangle$.

***Access*** This concludes our explanation of *mk*. The rest is boilerplate: there remains to define *read* and *inc* functions that satisfy the desired signature (Figure 4). Let us comment on *inc* (lines 56–63); *read* is analogous. Unpacking a monotonic counter object of type *mc* $k$ (line 58) yields an abstract observation constructor *obs*, a *methods* vector, and an observation *obs* $k$ (line 60). We extract the *inc* method (line 59) and invoke it (line 61). This invocation is legal, because we hold *obs* $k$, and produces a new observation *obs* $(k + 1)$ (line 62). By packing the methods vector together with this improved observation, we construct a new monotonic counter object, which, this time, has type *mc* $(k + 1)$. Yet, in a type-erasure interpretation, this new object is the unchanged argument!

Our definition of *mc* $k$ as an existential type, as well as our implementations of *read* and *inc*, follow the pattern of Pierce and

Turner's encoding of objects [23]. One difference is that Pierce and Turner's purpose was to avoid hidden mutable state (which, following Reynolds, they call *procedural abstraction*), in favor of purely functional objects and type abstraction: so, they used existential quantification *over the state*. Our purpose, on the contrary, is to explain procedural abstraction in the presence of monotonicity: so, we use existential quantification *over an observation* of the state.

Broadly speaking, we met the challenge that we set for ourselves: the code in Figure 5 satisfies the signature of Figure 4 and has the desired semantics. We did fail in one aspect, though: we modified the original code. In Figure 1, a counter was just an integer reference. In Figure 5, a counter is a pair of methods, which encapsulate an integer reference. At present, we do not know how to type-check the code in the absence of this encapsulation layer.

## 5. Application: thunks

We now explain why we are interested in thunks and how their study leads to a problem that is solved by our treatment of monotonic state. One of our long-term interests is to develop a type system that allows time complexity assertions to be expressed and checked. We would like it to be as close to a standard type system as possible, and we would like it to be able to encode the classic notions of *credits* and *debits,* which we now recall.

***Credits*** Tarjan [31] introduced the *banker's method* for deriving *amortized* time complexity bounds. The approach relies on the notion of a *credit.* One posits that, in order to perform just one step of computation, the machine requires, and consumes, one credit. At the beginning of its execution, the program is supplied with a number of credits, say $n$. Credits can serve as function arguments, as function results, and can be stored within data structures. Because there is no way to duplicate a credit or to create a credit out of nothing, the number of steps that the program can take must be bounded by $n$. Because credits can be stored and only later retrieved for consumption, this method leads to *amortized* complexity bounds.

One of the simplest examples of amortization is the reversal of a singly-linked list. The actual time complexity of reversal is linear in the length of the list. However, one can "pre-pay" for one reversal by artificially increasing the cost of the *cons* operation and storing one excess credit in each cell. Then, reversing an entire list requires just one credit, because the credit found in each cell pays for the next recursive call. In short, reversal has constant amortized time complexity: its cost has been amortized over the calls to *cons*.

A serious limitation of this method is that, because credits must not be duplicated, any data structure that contains credits must itself be "single-threaded", that is, linear. For instance, the above "pre-paid lists" must be linear: if such a list were carelessly shared, one could reverse it several times and incorrectly pretend that each reversal operation has constant amortized cost.

***Debits*** To address this limitation, Okasaki [22] proposed a modified version of the banker's method that allows data structures to be shared. The idea is to replace credits with debits: because it is sound for a debit to be duplicated, a debit-based analysis does not come with linearity restrictions.

Okasaki's approach is based on primitive suspensions, also known as *thunks*. When one wishes to execute a certain computation, instead of providing up front enough credits to run this computation, one creates a suspension, at a constant immediate cost. The cost of the computation must then be paid for, possibly in several increments, before the suspension can be forced.

In this approach, suspensions can be freely shared. This can cause a thunk to be paid for more than once: this is a waste, but leads to a sound approximation. This can also cause a thunk to be forced more than once, which is sound as well, because, thanks to memoization, the computation is performed just once, and the credits that have been accumulated are spent just once.

```
1 type thunk: ℕ → VAL → VAL
2 val mk: ∀n, α. (unit ∗ n$ → α) ∗ 1$ → thunk n α
3 val pay: ∀n, p, α. thunk n α ∗ p$ ∗ 1$ → thunk (n - p) α
4 val force: ∀α. thunk 0 α ∗ 1$ → α
```

**Figure 6.** A signature for thunks

Okasaki's approach has been formalized and proved correct by Danielsson [6], in the setting of a purely functional, lazy programming language.

***Credits as capabilities*** We equip an imperative, call-by-value programming language with a type-and-capability system that directly supports Tarjan's approach to complexity analysis, and that, via an encoding of thunks as references, also supports Okasaki and Danielsson's approach.

Credits are static entities, which do not exist at runtime. In other words, credits are capabilities. We introduce a new primitive capability: if $n$ has kind $\mathbb{N}$, then $n\$$ has kind CAP: it is a linear capability and represents $n$ credits. Capabilities can serve as function arguments, as function results, can be stored within data structures, and can form hidden invariants: as a result, so can credits.

We posit the subtyping axiom $(n + p)\$ \equiv n\$ * p\$$, where $n$ and $p$ are in $\mathbb{N}$. (If they were in $\mathbb{Z}$, this axiom would be unsound, as it would allow creating credits out of thin air.) (We write $\equiv$ for subtyping, both ways.)

To ensure that credits represent an actual measure of the computation cost, the type system must be modified in one other way: the typing rule for function applications must be amended so that every call consumes one credit. This is standard: in Hehner's approach [12], every recursive function call steps the clock; in Crary and Weirich's system [5], every function call steps the clock; in Danielsson's system [6], every function definition must be "ticked".

In a functional programming language, without primitive loop forms, function calls are the only non-trivial source of time complexity. Furthermore, time credits cannot be manufactured. As a result, one can prove that, up to a constant factor, the number of reduction steps that can be taken by the program is limited by the number of credits that are initially made available to it. For instance, one can prove the following statement: "if $p$ is a program such that $\vdash p : \forall n, \mathit{int}\ n * n\$ \rightarrow \mathit{unit}$ holds, then $p$ has worst-case linear time complexity: that is, there exists a constant $k$ such that, for every integer $n$, running the program $p\ n$ takes at most $kn$ steps." This theorem shows that the system derives correct complexity claims about complete programs. There follows that the claims that are derived about program components (open terms) must be correct as well, in some sense. The latter are "amortized" complexity claims in the sense that they eventually lead to a correct worst-case complexity assessment for all complete programs.

A type-and-capability system, equipped with credits-as-capabilities and with the rule that function calls consume one credit, is able to encode amortized time complexity analyses in the style of Tarjan. Such a system, however, has the limitation that data structures that contain credits must be linear. Thunks, in particular, do contain credits: as partial payments are made, more credits are stored; when the thunk is forced, the stored amount drops to zero. Thus, it seems that thunks must be linear. This is a problem: in order to reproduce Okasaki and Danielsson's analyses, it is essential to allow thunks to be freely shared.

This is where our treatment of monotonic state comes in: using the "monotonic counters" coding pattern, a call-by-value version of Danielsson's thunks can be implemented as a library.

```
1  type thunk n α =                          – the type of thunks
2    ∃obs : ℕ → DCAP.
3      ((∀n, p. unit ∗ obs n ∗ p$ → unit ∗ obs (n - p)) ×
4       (unit ∗ obs 0 → α)) ∗
5      obs n
6
7  type state n α =                          – internal state of a thunk:
8    | White ((unit ∗ n$ → α) ∗ 1$)          – not yet evaluated
9    | Gray (unit)                           – being evaluated
10   | Black (α)                             – evaluated
```

**Figure 7.** Thunks: internal type definitions

***A signature for thunks*** A signature for thunks appears in Figure 6. A thunk is parameterized with its cost $n$ (a "debit") and with its result type $\alpha$ (line 1). The type *thunk n α* has kind VAL: a thunk is an ordinary value, and can be duplicated without restriction. A thunk of cost $n$ is created out of a computation of cost $n$, that is, a function of *unit* to $\alpha$ that consumes $n$ credits (line 2). At any time, a thunk of cost $n$ can be partially paid for (line 3). This consumes $p$ credits, and produces a thunk of cost $n - p$. (This is subtraction in $\mathbb{N}$.) At runtime, *pay* is a no-op and returns its argument. Finally, a thunk can be forced when it has been paid for, that is, when its cost is zero (line 4).

Each of *mk*, *pay*, and *force* require (and consume) one credit. This effectively means that a call to one of these functions costs two credits, one of which pays for the call, the other of which is passed to the function. This does not have deep significance: the point is that these operations have (amortized) constant time cost.

***Implementation*** The definition of the type *thunk* closely follows the pattern that was introduced for monotonic counters. According to Figure 7, a thunk of cost $n$ is:

> for some abstract notion of an observation of a natural integer (line 2),
>
> a pair of a *pay* method (line 3), which accepts an observation of any integer $n$, as well as a number of credits $p$, and returns an observation of $n - p$; and a *force* method (line 4), which expects an observation of 0 and produces a value of type $\alpha$;
>
> packaged together with an observation of $n$ (line 5).

Under the hood, a thunk is implemented as a hidden reference to a state, which is one of *White*, *Gray*, or *Black* (lines 7–10). We build on an earlier encoding of thunks by the second author [24], which explained how to exploit a hidden reference but did not include a time complexity aspect. Here, we introduce a hidden "piggy bank", where credits are inserted by *pay*, and that is broken by the first call to *force*.

The implementation of thunks appears in Figure 8. In the prologue (lines 3–8), we introduce a fate $\varphi$ over the natural integers. Its value *decreases* with time and represents the number of credits that remain to be paid. It is initially set to the cost of the suspended computation. We also allocate a reference $r$, which initially holds the color *White*, the suspended computation *userfun*, and one credit, which we later use to pay for the call to *userfun*.

The invariant $I$ (line 14) is a conjunction of three capabilities, which respectively govern the reference, the piggy bank, and the fate. These capabilities share two integer indices. The index *nc*, for *necessary credits,* is the number of credits required to force the thunk. Its value is initially $n$ (line 17), and drops to zero when the thunk is first forced. The index *ac*, for *available credits,* is the number of credits in the piggy bank. The index *ac* is increased when a payment is made, and diminished by $n$ when the thunk is first forced. It is worth noting that *ac* does not, by itself, exhibit

```
1  let mk : ∀n, α. (unit ∗ n$ → α) ∗ 1$ → thunk n α =
2  λ(userfun : unit ∗ n$ → α).
3    let fate φ : FATE ℕ (≥) = n in          – a decreasing fate
4    got cap {φ : n};
5    let cap obs i = ⟨φ : i⟩ in
6    make obs n;
7    let σ, (r: [σ]) = ref (White userfun) in
8    got cap {σ: ref (state n α)};           – this uses up our 1$
9
10   let methods :
11     (∀n, p. unit ∗ obs n ∗ p$ → unit ∗ obs (n - p)) ×
12     (unit ∗ obs 0 → α) =
13
14     hide I = ∃nc, ac.
15       {σ : ref ((state nc α) ⊗ I) } ∗ ac$ ∗ {φ : nc - ac}
16     outside of
17     pack cap I;                           – the witnesses are n and 0
18
19     let pay : ∀n, p. unit ∗ obs n ∗ p$ ∗ I
20                 → unit ∗ obs (n - p) ∗ I =
21     λ().
22       let nc, ac = unpack cap I in
23       got cap
24         obs n ∗
25         {σ : ref ((state nc α) ⊗ I)} ∗
26         (ac + p)$ ∗                       – our new, combined credit
27         {φ : nc - ac};
28       exploit obs n;                      – this yields n ≥ nc - ac
29       set fate φ := nc - (ac + p);        – a monotonic update
30       make obs (n - p);                   – uses n - p ≥ nc - (ac + p)
31       pack cap I                          – witnesses: nc and ac + p
32
33     and force : unit ∗ obs 0 ∗ I → (α ⊗ I) ∗ I =
34     λ().
35       let nc, ac = unpack cap I in
36       got cap
37         obs 0 ∗
38         {σ : ref ((state nc α) ⊗ I)} ∗
39         ac$ ∗
40         {φ : nc - ac};
41       exploit obs 0;                      – this yields nc - ac = 0
42       match !r with                       – whence ac ≥ nc
43         | White (userfun : unit ∗ nc$ ∗ I → (α ⊗ I) ∗ I) →
44             r := Gray ();
45             got cap
46               {σ : ref ((state 0 α) ⊗ I)} ∗
47               (nc + 1)$ ∗                 – the necessary credit
48               (ac - nc)$ ∗                – any leftover credit
49               {φ : 0};
50             pack cap I;                   – witnesses: 0 and ac - nc
51             got cap (nc + 1)$ ∗ I;
52             let v : α ⊗ I = userfun () in
53             let _, _ = unpack cap I in
54             r := Black v;
55             pack cap I;                   – witnesses: 0 and 0
56             v
57         | Gray () → fail
58         | Black v → pack cap I; v         – witnesses: nc and ac
59     in (pay, force)
60 in
61 got cap obs n;
62 pack methods as thunk n α
```

**Figure 8.** Thunks

monotonic behavior. The difference $nc - ac$, which represents the amount that remains to be paid, does: it decreases with time.

For lack of space, the reason why $I$ is recursively defined, as well as the meaning of the tensor $\cdot \otimes I$, are not explained. As far as this paper is concerned, these aspects can be ignored. The reader is referred to the second author's earlier paper on hidden state [24].

The *pay* method (lines 19–31) stores the $p$ credits that it receives as an argument in the piggy bank (line 26), updates the fate so as to reflect a decrease in the amount that remains to be paid (line 29), and publishes a new observation (line 30). This method has absolutely no runtime effect: its type erasure is $\lambda().()$. If the type system supported user-defined *coercions* (a feature that we have not included!), the *pay* method, as well as the external *pay* function (Figure 6, line 3), could be declared as coercions. This would have several benefits: (i) an application of *pay* would be considered a coercion application, as opposed to a function call, so it would cost zero credits, as opposed to two; (ii) an application of *pay* would generate no code whatsoever; (iii) more importantly, this would allow *pay* to be used under a linear, covariant context, a feature that is present in Danielsson's system [6, §11]. Coercions, if included in the system, could also serve to expose the fact that the type *thunk* $n\,\alpha$ is covariant in $n$ and in $\alpha$.

The *force* method exploits the observation *obs* 0 (line 37) to determine that the available credit *ac* exceeds the necessary credit *nc* (line 41). This allows *nc* credits to be taken out of the piggy bank. Furthermore, in the first branch of the **match** construct, deconstructing *White* makes one extra credit is available. Thus, in total, we have $nc + 1$ credits (line 47). These credits are required to call *userfun* (lines 51–52): one credit is consumed by the call itself, while the other *nc* credits are passed to *userfun*, which consumes them. Note that, prior to invoking *userfun*, the invariant $I$ must be re-established (line 50). This is possible, in spite of the fact that *nc* credits have just been taken out of the piggy bank, because the thunk is now colored gray.

The instruction *r := Black v* (line 54) requires unpacking $I$ before the update (line 53) and re-packing it afterwards (line 55).

The implementation of the external functions *pay* and *force* (whose types appear in Figure 6, lines 3–4) is omitted. As before, they are just wrappers in the style of Pierce and Turner [23]. Because this wrapping involves a function call, the external versions of *pay* and *force* require one credit, whereas the internal methods do not.

## 6. Application: hash-consing

We now present an application to the specification of hash-consing.

***A challenge*** Let us fix a type *data*. A hash-consing facility usually takes the form of a function of type *data* → *data*, with the informal specification that the images of $d_1$ and $d_2$ are physically equal if and only if $d_1$ and $d_2$ are equal. Here, however, in order to avoid dealing with physical equality, we consider a slightly more basic interface, with equivalent expressive power. We view a *hash-consing function* as a function of type *data* → *int*, with the informal specification that the integer hashes associated with $d_1$ and $d_2$ are equal if and only if $d_1$ and $d_2$ are equal.

Hash-consing is typically implemented using a mutable data structure (say, a hash table) that maps data to integers. When some datum $d$ is presented, one checks whether $d$ already is in the domain of the table: if not, the table is extended with a binding of $d$ to some fresh integer. At this point, a binding of $d$ to some integer $i$ must exist in the table, and $i$ is returned.

The challenge is to check that this implementation is correct with respect to a signature that does not reveal the existence of an internal state, does not impose any linearity restrictions, yet is strong enough to encode the above informal specification.

***An ideal signature*** What is an ideal signature for hash-consing? We wish to claim that *hash* implements an injective mathematical function from data to integers. This is done by the signature in Figure 9, where *h* is declared to be such a function (lines 1–2), and *hash* is declared to implement *h* (line 3). There, *h* is a function in the ambient logic, while *hash* is a programming language function. We write *data* both for the (unparameterized) type of data in the ambient logic and for the (indexed) type of data in the programming language. We stick with the indexed-type approach that we have used throughout this paper, although a notation in the style of Hoare, without indexed types and with pre- and post-conditions, would arguably be more palatable.

```
1 logic val h: data → ℤ
2 logic property: injective h
3 val hash: ∀d. data d → int (h d)
```

**Figure 9.** An ideal (unattainable?) signature for hash-consing

```
1 type φ: FATE ifmap ⊆
2 val hash: ∀d. data d → ∃i. int i ∗ ⟨φ : [d ↦ i]⟩
```

**Figure 10.** A novel, pragmatic signature for hash-consing

It is very likely that this ideal signature is sound: as far as clients are concerned, everything is consistent with the illusion that *hash* has no side effect and implements some *fixed* injective mathematical function *h*.

Unfortunately, it is unclear how to argue that an imperative implementation satisfies this signature. The mutable table is initially empty and is populated only as the program is executed. As a result, it seems impossible to *statically* provide a definition of the mathematical function *h*. In fact, in two distinct program runs, the mutable table might hold distinct contents!

***A solution*** An informal explanation why hash-consing works is that the hash table holds a map of data to integer codes that remains injective at all times and *can only grow with time*, so that whatever facts the client observes about this map remain true forever. Of course, it is straightforward to model this situation using fates and observations.

We write *ifmap* for the (ambient logic) type of injective finite maps of data to integers. We write $[d \mapsto i]$ for the singleton map that maps $d$ to $i$. We write $\subseteq$ for map inclusion.

In the implementation of hash-consing that follows, we create a fate, say $\varphi$, of kind FATE *ifmap* $(\subseteq)$, and set up a hidden invariant that relates the content of the mutable table with this fate. By doing so, we state (and we must prove) that the table holds an injective map and grows with time.

An additional key idea is that, contrary to our earlier examples (§4, §5), the existence of $\varphi$ is not hidden. Instead, it is exposed in the signature (Figure 10, line 1). This allows *hash* to be specified as follows: when passed a datum $d$, *hash* produces an integer $i$, *together with a prediction* of the singleton map $[d \mapsto i]$ (Figure 10, line 2). This can be understood as a guarantee that the binding of $d$ to $i$ is in the map, now and forever.

This signature does not involve any linear entities, so *hash* can be used as if it were side-effect-free. In particular, it can be invoked by multiple clients without requiring them to cooperate with one another, as would be the case if the use of *hash* was governed by a linear capability.

Before showing how a typical imperative implementation can satisfy this new signature, let us first find out how expressive it is. Does it express the property that the integer hashes associated with $d_1$ and $d_2$ are equal if and only if $d_1$ and $d_2$ are equal? Yes—here is how. Imagine that a client invokes *hash*, at two arbitrary points in time, with respective arguments $d_1$ and $d_2$. She receives, in exchange, two predictions $\langle\varphi : [d_1 \mapsto i_1]\rangle$ and $\langle\varphi : [d_2 \mapsto i_2]\rangle$.

```
1  type htbl: SNG → VAL                    – type of a hash table
2  type cht: SNG → fmap → CAP              – capability over a hash table
3
4  val create:
5    unit → ∃ρ. (htbl ρ ∗ cht ρ ∅)
6  val add: ∀d i ρ m.
7    data d × int i × htbl ρ ∗ cht ρ m →
8    unit ∗ cht ρ (m[d ↦ i])
9  val find: ∀d ρ m.
10   data d × htbl ρ ∗ cht ρ m →
11   ((∃i. int i ∗ ⟨[d ↦ i] ⊆ m⟩) + (unit ∗ ⟨d ∉ dom(m)⟩))
12   ∗ cht ρ m
```

**Figure 11.** A strong specification of hash tables

```
1  type lock : CAP → VAL                   – locks are duplicable
2  val newlock: ∀γ. unit → lock γ          – initially in a locked state
3  val lock: ∀γ. lock γ → unit ∗ γ         – locking yields γ
4  val unlock: ∀γ. lock γ ∗ γ → unit       – unlocking consumes γ
```

**Figure 12.** A signature for locks

By joining these predictions (§3), she obtains the existence of an *injective* finite map $m$ such that $[d_1 \mapsto i_1] \subseteq m$ and $[d_2 \mapsto i_2] \subseteq m$ both hold. The fact that $m$ is a map allows her to show that $d_1 = d_2$ implies $i_1 = i_2$, while the fact that $m$ is injective allows her to prove that $i_1 = i_2$ implies $d_1 = d_2$. Thus, she has $d_1 = d_2 \iff i_1 = i_2$, as desired.

In summary, whereas the ideal signature of Figure 9 claims that *hash* implements a fixed injective mathematical function, the pragmatic signature of Figure 10 claims that it implements an injective mathematical function whose graph may be constructed at runtime, and may grow with time. The latter signature is more general than the former. It represents a relaxed definition of the concept of a *pure* function. It is, to the best of our knowledge, a contribution of this paper.

***Implementation*** Let us now see how this signature can be implemented in our setting. Before we look at the code, we need a couple of building blocks.

The first building block is an implementation of hash tables. An arbitrary implementation will do, as long as it satisfies the signature in Figure 11. This is a completely standard specification of hash tables: it is in no way specific to our application to hash-consing. It relies on a logic type *fmap* of finite (but not necessarily injective) maps of data to integers. It publishes an abstract type and an abstract capabillity: *htbl* $\rho$ (line 1) is the type of a pointer to a hash table in region $\rho$, while *cht* $\rho$ $m$ is a capability over such a hash table. The capability *cht* $\rho$ $m$ plays a dual role: it represents the ownership of the hash table and encodes the assertion that the contents of the table corresponds to the finite map $m$. This is analogous to an abstract predicate in separation logic [20]. The function *create* creates a fresh region $\rho$ containing an empty hash table (line 5). It returns a pair of a pointer to the table and a capability for the table. A call to *add* $(d, i, h)$ (lines 7–8) updates the table $h$ with a binding of $d$ to $i$. It requires a capability *cht* $\rho$ $m$ and produces an updated capability *cht* $\rho$ $(m[d \mapsto i])$, so as to reflect the update at the logical level. A call to *find* $(d, h)$ (lines 10–12) either returns an integer $i$ together with a proof that $d$ is associated with $i$ in the table, or produces a proof that $d$ is not in the domain of the table. In either case, *find* returns the capability over the unchanged hash table (line 12).

The second building block is an implementation of locks. In contrast with our earlier examples, here, we will not use the anti-frame rule directly. The anti-frame rule is too restrictive [26] in that it would require us to restore the hidden invariant before calling the hash table functions *add* and *find*. However, by restoring the invariant, we would lose the capabillity over the hash table, so we would no longer be able to call *add* or *find*! A solution to this problem is to introduce a dynamic check, and the most elegant way of doing so is to rely on an implementation of *locks*. A signature for locks appears in Figure 12. Our locks are analogous to dynamically allocated locks in separation logic [11, 13, 21]. Unlocking consumes a capability $\gamma$ (line 4), which can later be recovered by locking (line 3). The capability $\gamma$ is fixed at lock creation time (line 2). The type *lock* $\gamma$ is duplicable. Yet, the capability is never duplicated, because the *lock* operation can fail (in a sequential setting) or block (in a concurrent setting). In a sequential setting, locks can be implemented in terms of the anti-frame rule: all it takes is a hidden reference to a Boolean flag. In a concurrent setting, locks can be viewed as primitive.

```
1  logic type ifmap = {m : fmap | injective m}
2
3  let mkhash : unit → ∃φ : FATE ifmap ⊆.
4      ∀d. data d → ∃i. int i ∗ ⟨φ : (d ↦ i)⟩ =
5  λ().
6    let fate φ : FATE ifmap ⊆ = ∅ in
7    let ρ, (h : htbl ρ) = create () in
8    let σ, (r : [σ]) = ref 0 in
9    got cap {φ: ∅} ∗ cht ρ ∅ ∗ {σ: ref int 0};
10   let cap I = ∃c : int. ∃m : ifmap.
11     {φ: m} ∗ cht ρ m ∗
12     {σ: ref int c} ∗
13     ⟨∀i, i ∈ codom(m) → i < c⟩ in
14   pack cap I;                          – witnesses: 0 and ∅
15   let l : lock I = newlock () in
16   unlock l;                            – this consumes I
17
18   let hash : ∀d. data d → ∃i. int i ∗ ⟨φ : (d ↦ i)⟩ =
19   λd.
20     let () = lock l in                 – this yields I
21     let c, m = unpack cap I in         – open it up
22     match find (d, h) with
23     | Left i →                         – we have ⟨[d ↦ i] ⊆ m⟩
24       make ⟨φ: [d ↦ i]⟩;               – permitted: we have {φ : m}
25       pack cap I;                      – witnesses: c and m
26       unlock l;                        – this consumes I again
27       pack i
28     | Right () →                       – we have ⟨d ∉ dom(m)⟩
29       let c = !r in
30       add (d, c, h);                   – we have cht ρ (m[d ↦ c])
31       r := c +1;                       – we have {σ: ref int (c + 1)}
32       set fate φ := m[d ↦ c];
33       got cap ⟨∀i, i ∈ codom(m[d ↦ c]) → i < c + 1⟩;
34       make ⟨φ: [d ↦ c]⟩;
35       pack cap I;                      – witnesses: (c+1)
36       unlock l;                        – and m[d ↦ c]
37       pack c
38   in
39   pack hash                            – witness: φ
```

**Figure 13.** Implementation of hash-consing

Let us now turn towards the actual implementation of hash-consing (Figure 13).

We first define the type *ifmap* of *injective* finite maps on top of the type *fmap* of finite maps (line 1). We use Coq's subset notation. An injective map is a pair of a map $m$ and a proof that $m$ is injective. In the following, we adopt the informal convention that an *ifmap* object can be used where an *fmap* object is expected, and vice-versa; in the latter case, a proof obligation is generated.

The function *mkhash* (lines 3–39) is in charge of creating a new instance of the hash-consing facility. Its result type corresponds to the signature of Figure 10. Its code starts by creating a new initially empty increasing fate $\varphi$ over injective finite maps (line 6), an empty hash table $h$ (line 7) and an integer reference $r$ (line 8). The internal invariant is defined immediately thereafter (lines 10–13). The invariant states that the fate and the table are in a common state $m$, an injective finite map. It further states that the integer value $c$ stored in the reference $r$ is an upper bound for the codomain of $h$: that is, $c$ is the next available integer code. The invariant initially holds (line 14). (This generates the proof obligation $\forall i,\ i \in \mathrm{codom}(\emptyset) \rightarrow i < 0$.) The invariant is hidden using a fresh lock $l$. The lock $l$ is captured in the closure of the function *hash*, which is permitted because $l$ has duplicable type *lock I*.

Lines 18–39 show the actual hash-consing function, *hash*. We retrieve the invariant $I$ out of the lock and unpack it (Lines 20–21). Then, we check whether $d$ is already in the domain of $h$. If so (lines 23–27), we produce the observation $\langle \varphi : [d \mapsto i] \rangle$. This is permitted because we have the logical proposition $\langle [d \mapsto i] \subseteq m \rangle$ as well as the capability $\{\varphi : m\}$. Otherwise (lines 28–37), we update $h$ so that the datum $d$ is now mapped to $c$, the current value of the reference $r$. The reference $r$ is then incremented so that its value remains an upper bound for the codomain of $h$. We then check that we have the capabilities needed to repack $I$ (lines 32–34). The first one (line 32), namely $\{\varphi : m[d \mapsto c]\}$, generates two proof obligations. First, we must prove that $m[d \mapsto c]$ is an injective map: this holds because $m$ is injective and $c$ is not in the codomain of $m$. Second, we must prove that $m$ is a subset of $m[d \mapsto c]$: this holds because $d$ is not in the domain of $m$. The second one (line 33) represents a logical proposition that we must prove, namely $\forall i,\ i \in \mathrm{codom}(m[d \mapsto c]) \rightarrow i < c + 1$. This follows from the equality $\mathrm{codom}(m[d \mapsto c]) = \mathrm{codom}(m) \cup \{c\}$ and from the hypothesis $\forall i,\ i \in \mathrm{codom}(m) \rightarrow i < c$. The last capability (line 34) is an observation. This line produces two trivial proof obligations: first, we must check that $[d \mapsto c]$ is injective; second, we must prove $[d \mapsto c] \subseteq m[d \mapsto c]$.

Although the concrete syntax used in in our code snippets is admittedly informal, we believe that we have explained fairly precisely how a type-and-capability system, equipped with fates and predictions, and supplemented with an expressive logic, can be used to give a specification of a hash-consing facility. This specification conceals the existence of an internal state, yet guarantees that two pieces of input data receive the same code if and only if they are equal. This is an original result: to the best of our knowledge, no such specification of hash-consing has been presented before.

## 7. Related work

***Ghost state and history constraints*** A fate is a ghost variable that comes with a built-in temporal property: the sequence of its values forms an increasing chain with respect to a certain preorder $\mathbf{R}$. In combination with an ordinary invariant ("the values of reference $r$ and fate $\varphi$ coincide"), this allows expressing a temporal assertion about the state ("the value of reference $r$ must grow with time"). The idea of introducing ghost variables in order to reduce temporal reasoning to present-time reasoning is not new; see, for instance, Schneider [28, chapter 7].

Liskov and Wing [18] associate a *history constraint*—a predicate over pairs of visible states—with a class definition. This idea has been implemented in the Larch/C++ [15] and JML [16] specification languages. Unfortunately, there seems to exist no clear account of how history properties are verified. Our understanding is that the tools check that the pre- and post-state of every method are related by the history constraint. This is a necessary condition but, in the presence of callbacks, not a sufficient one. Furthermore, these systems offer no way of exploiting a history constraint to establish a new logical fact.

Fähndrich and Leino [10] note that, if the state of an object is constrained to evolve in a monotonic manner, then it is sound to make an assertion about this object, even in a system that does not control aliasing or ownership. We take inspiration from this idea: a prediction represents an assertion about an entity (namely, a fate) that one does not own. Fähndrich and Leino require every field update to be monotonic (that is, to preserve every property that might be known of the object). We adopt a simpler and more expressive approach: our fates and predictions are independent of the treatment of mutable state. While the update of a fate is required to be monotonic with respect to a fixed law $\mathbf{R}$, there is a priori no restriction on updates of references.

Leino and Schulte [17] extend the Spec# program verification system with *history invariants* in order to verify a version of the subject-observer pattern. Their approach is sound in the presence of callbacks and re-entrancy. Furthermore, there is a benefit to declaring history invariants. In the basic Spec# methodology, an invariant associated with object $o_1$ may refer to an object $o_2$ only if $o_1$ owns $o_2$ ($o_2$ is a "transitive rep object" of $o_1$). Leino and Schulte relax this restriction and allow this also when $o_2$ is declared to be a "subject" of $o_1$ and the invariant associated with $o_1$ is stable under the history invariant associated with $o_2$. Again, this expresses the idea that, provided updates to $o_2$ are monotonic, it is sound for $o_1$ to make an assertion about $o_2$, even though $o_1$ does not own $o_2$.

***Regions*** Most type-and-capability calculi use *regions* as a mechanism for assigning a name to a single value or to a set thereof [3, 4, 29]. Regions involve a form of monotonicity, which manifests itself in two ways: (i) once a region name is allocated, it exists forever; (ii) the population of a region can only grow with time. Thus, if a value $v$ inhabits a region $\rho$, then this fact holds forever. For this reason, the type $[\rho]$ of the inhabitants of region $\rho$ can safely be considered duplicable. This is the same reason why predictions are considered duplicable in the present paper.

Whereas regions denote sets of runtime values, fates take values in some type $\mathbf{T}$ of the ambient logic. This design decision plays an important part in the simplicity of the meta-theory of fates, but makes it impossible to define regions in terms of fates.

***Concurrency*** Fates and predictions are sound in both sequential and concurrent settings. The implementations of monotonic counters and thunks presented in this paper exploit the anti-frame rule, which is sound only in a sequential setting [24]. In a concurrent setting, one would instead use dynamically allocated locks, which hold and hide a capability [11, 13].

***Deny-guarantee*** There is a strong connection between our work and deny-guarantee reasoning [7]. (We are grateful to Hongseok Yang for bringing this to our attention.) In fact, it is possible to sketch an informal encoding of fates and predictions in terms of stable deny-guarantee assertions.

First, the information that a fate variable $\varphi$ has kind FATE $\mathbf{T}$ $\mathbf{R}$ corresponds to a deny on any update of $\varphi$ that does not respect $\mathbf{R}$. That is, no thread is allowed to perform a non-monotonic update. In our system, this information carried in the kind is duplicable, so it should be encoded as a "duplicable deny". This can be expressed in terms of Dodds *et al.*'s "fractional denies" via existential quantification over a fraction: $\exists a > 0.(a)\mathit{deny}$.

Second, a prediction $\langle \varphi : i \rangle$ corresponds to an assertion that the value of $\varphi$ is at least $i$, that is, $i\ \mathbf{R}\ \varphi$. Such an assertion carries no permission, hence is duplicable, as required. Furthermore, this assertion, once conjoined with the deny that forbids non-monotonic updates, is stable.

Last, a fate ownership token $\{\varphi : i\}$ corresponds to a conjunction of the assertion $i = \varphi$ and a full permission over all monotonic

updates of $\varphi$. Again, once conjoined with the deny that forbids non-monotonic updates, this assertion is stable.

One can informally check that this interpretation validates all of the axioms that govern fates and predictions (§3).

In summary, like deny-guarantee, fates and predictions allow reasoning about state changes. They are less ambitious and simpler in several ways: (i) they do not permit interference between threads; (ii) the law that governs a fate is fixed at allocation time, and, for this reason, fractional permissions are not needed; (iii) fates hold logical values, as opposed to runtime values. Fates and predictions could perhaps be viewed as an interesting "design pattern" in a programming language equipped with deny-guarantee reasoning.

***Relational models of monotonic state*** Ahmed, Dreyer, and Rossberg [1] consider a call-by-value $\lambda$-calculus with general references, and endow it with a possible worlds model in which the relational interpretation of a type may grow with time. They use this model to prove certain pairs of programs contextually equivalent, but also (often) to establish facts about a single program, such as the fact that a certain dynamic check is redundant. For this purpose, type-based approaches are applicable and, perhaps, offer better potential for integration in a programming language design.

When restricted to a unary setting, Ahmed *et al.*'s system exhibits strong analogies with Charguéraud and Pottier's calculi [3, 24] as well as with the present paper. Roughly speaking, an island corresponds to a piece of state that is hidden via the anti-frame rule; the island's population, a set of values, grows with time, and corresponds to a fate; and the fixed law that relates the population with a store relation corresponds to an invariant that is imposed via the anti-frame rule and ties a fate to a piece of runtime state.

In Ahmed *et al.*'s motivating example [1, Figure 1], type abstraction is used to protect an extensible table implementation. Integer indices into the table are passed to the client at an abstract type $t$, so the client cannot forge indices. As a result, every index must be in the domain of the table (which grows with time) and no bounds check is necessary. Type generativity guarantees that distinct table instances give rise to distinct instances of the abstract type $t$.

Our system also allows proving that no bounds check is necessary; is amenable to mechanical checking; and (perhaps surprisingly) is able to disclose the fact that table indices are just integers.

How does this work? The problem is essentially a simplified version of hash-consing, so our approach is similar to that described earlier (§6). With a table instance, we associate a fate $\varphi$, ranging over sets of integers, and whose law is set inclusion. As an internal invariant, we assert that $\varphi$ represents the domain of the table. Then, we define a "valid table index" as a pair of an integer index $i$ and an observation that $i$ is in the domain of the table: that is, we define the type *index* $\varphi$ as $\exists i.(\mathit{int}\ i * \langle \varphi : \{i\}\rangle)$. This type plays the role of $t$ in Ahmed *et al.*'s paper.

When supplied by the client with a value of type *index* $\varphi$, we confront the observation $\langle \varphi : \{i\}\rangle$ with the current state of $\varphi$, a capability of the form $\{\varphi : I\}$, where the set of integers $I$ represents the current domain of the table. This yields $i \in I$, which guarantees that the index is within bounds.

In Ahmed *et al.*'s "irreversible state change" example [1, §5.5], the challenge is to prove that $!x$ evaluates to 1, even though the unknown function $f$ might (via a re-entrant call) affect $x$:

$$\textbf{let}\ x = \mathit{ref}\,0\ \mathit{in}\ \lambda f.(x := 1; f(); !x)$$

This is proved by introducing a fate to express the fact that the value of $x$ grows with time, and by using the anti-frame rule to hide the existence of the cell $x$ and of its fate, together with the invariant that $x$ is 0 or 1. The update $x := 1$ is provably monotonic, since 1 is the greatest permitted value for $x$. An observation of the fate in state 1 is created before the call to $f()$ and is exploited, after the call, to establish that $x$ still holds the value 1.

Dreyer *et al.* [9] revisit Ahmed *et al.*'s framework and introduce so-called "state transition systems" to model the way in which properties of local state evolve over time. In their most basic form, state transition systems have "public transitions" only. In that case, they are just pre-orders, and allow reasoning about monotonic hidden state much in the same way as we do, as illustrated by the above examples. In their most general form, state transition systems also have "private transitions", which offer a way of taking advantage of the well-bracketing of function calls and returns. The second author's unpublished generalized anti-frame rule [25] stems from a similar motivation, and also allows exploiting well-bracketing. The basic anti-frame rule requires a fixed invariant, so a piece of hidden state is typically governed by a single fate throughout its lifetime. In contrast, the generalized anti-frame rule accepts a parameterized invariant, and uses universal quantification to express well-bracketing, so a piece of hidden state can be usefully tied to different fates at different points in time. Roughly speaking, dynamically allocating a new fate and tying it to the state corresponds to creating a fresh instance of a state transition system, or, in Dreyer *et al.*'s approach, to taking a private transition. In summary, the two approaches are closely related. Ours differs in that it is expressed as a type system and (we believe) is presented in a more orthogonal fashion. In particular, the rules that govern fates and predictions are independent of those that govern hidden state.

Sumii's environmental bisimulations [30] also involve a form of monotonicity, as the set of values that are accessible to the environment grows with time. We are unfortunately unable to offer a more informed comparison of the two approaches. The examples that Sumii presents are borrowed from Ahmed *et al.* and can be dealt with using our type- and assertion-based approach.

***Type-based complexity analysis*** Although *automated* time complexity analysis is a research field of its own, the development of expressive type systems that can *check* user-provided time complexity assertions has received surprisingly little attention.

The type systems by Dornic *et al.* [8], Reistad and Gifford [27], and Crary and Weirich [5] annotate function types with a worst-case cost. The type-and-capability system that we have sketched (§5) is significantly more expressive. This is evidenced, we hope, by our encoding of Okasaki and Danielsson's analysis of thunks.

The idea that a *space credit* is a linear entity, which can be passed around and stored, is not new: Hofmann [14] uses it quite elegantly to keep track of heap space usage. The idea that a *time credit* is a linear entity, which can similarly be passed around and stored, is explicit in Tarjan's work [31], and is formalized, for instance, by Atkey [2]. Atkey's system is analogous to ours in its motivation and in its treatment of credits. It is less expressive in several ways: because it lacks a treatment of hidden state and of monotonicity, we believe that it does not allow an encoding of Danielsson's thunks.

## References


[1] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 340–353, January 2009.

[2] Robert Atkey. Amortised resource analysis with separation logic. In *European Symposium on Programming (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 85–103. Springer, 2010.

[3] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming (ICFP)*, pages 213–224, September 2008.

[4] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 262–275, January 1999.

[5] Karl Crary and Stephanie Weirich. Resource bound certification. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 184–198, January 2000.

[6] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2008.

[7] Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *European Symposium on Programming (ESOP)*, volume 5502 of *Lecture Notes in Computer Science*, pages 363–377. Springer, March 2009.

[8] Vincent Dornic, Pierre Jouvelot, and David K. Gifford. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems*, 1(1):33–45, 1992.

[9] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *ACM International Conference on Functional Programming (ICFP)*, pages 143–156, September 2010.

[10] Manuel Fähndrich and Rustan Leino. Heap monotonic typestates. In *International Workshop on Alias Confinement and Ownership (IWACO)*, July 2003.

[11] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. Technical Report MSR-TR-2007-39, Microsoft Research, September 2007.

[12] Eric C. R. Hehner. *Abstractions of Time*, pages 191–210. Prentice Hall, 1994.

[13] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *European Symposium on Programming (ESOP)*, volume 4960 of *Lecture Notes in Computer Science*, pages 353–367. Springer, April 2008.

[14] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[15] Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In *Formal Methods (FM)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer, January 1999.

[16] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. *JML Reference Manual*, May 2008.

[17] K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In *European Symposium on Programming (ESOP)*, volume 4421 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2007.

[18] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[19] Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in system $F^\circ$. In *Workshop on Types in Language Design and Implementation (TLDI)*, pages 77–88, January 2010.

[20] Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable ADTs in Hoare type theory. In *European Symposium on Programming (ESOP)*, volume 4421 of *Lecture Notes in Computer Science*, pages 189–204. Springer, March 2007.

[21] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, May 2007.

[22] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[23] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[24] François Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 331–340, June 2008.

[25] François Pottier. Generalizing the higher-order frame and anti-frame rules. Unpublished, July 2009.

[26] François Pottier. Three comments on the anti-frame rule. Unpublished, July 2009.

[27] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. In *ACM Symposium on Lisp and Functional Programming (LFP)*, pages 65–78, 1994.

[28] Fred B. Schneider. *On Concurrent Programming*. Springer, 1997.

[29] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming (ESOP)*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer, March 2000.

[30] Eijiro Sumii. A complete characterization of observational equivalence in polymorphic lambda-calculus with general references. In *Computer Science Logic*, volume 5771 of *Lecture Notes in Computer Science*, pages 455–469. Springer, September 2009.

[31] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[32] Hongwei Xi. Dependent ML: an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007.

## A. Quick reference

We very briefly summarize how to equip a type-and-capability system in the style of Charguéraud and Pottier [3] with fates and predictions. In order to do this, the system must have existential quantification; a distinction between duplicable and linear capabilities; and logical assertions (viewed as duplicable capabilities). Everything else (products, sums, functions, regions, references, recursive types, hidden state) is orthogonal and described elsewhere [3, 24].

The kinds are listed in Figure 14. There, the judgement $\vdash_{\text{CIC}}$ is the typing judgement of the ambient logic, which we take to be the Calculus of Inductive Constructions. In this and the following figures, uses of $\vdash_{\text{CIC}}$ correspond to proof obligations, which, in an implementation, would be shipped to an external theorem prover. Metavariables in bold face stand for objects of the ambient logic: typically, $\mathbf{T}$ ranges over types, $\mathbf{R}$ over preorders, $\mathbf{t}$ over terms, $\mathbf{P}$ over propositions.

The syntax of types and capabilities appears in Figure 15. They are presented as a single syntactic category of *objects* $o$. A kind assignment system (Figure 16) allows telling which objects are types and which are capabilities; which objects are linear and which are duplicable; etc. This kind-driven approach to linearity has been independently studied by Mazurak *et al.* [19].

A kinding environment $K$ maps type variables to kinds. We write $\lfloor K \rfloor_{\text{CIC}}$ for the restriction of $K$ to bindings of the form $\alpha :: \mathbf{T}$. Value types (VAL) form a subset of computation types (CMP); similarly, duplicable capabilities (DCAP) form a subset of capabilities (CAP). The conjunction $(*)$ of a type and a capability is duplicable if and only if both conjuncts are duplicable. The ownership of a fate is considered a linear capability, whereas predictions and logical assertions are duplicable capabilities.

The typing judgements and typing rules are exactly as in Charguéraud and Pottier [3]. The delta with this previous work lies in the following subtyping axioms. The axioms that govern fates and predictions appear in Figure 17. They are subject to the implicit side condition that $\varphi$ has kind FATE $\mathbf{T}\ \mathbf{R}$. The axioms that govern propositions appear in Figure 18. Last, in order to allow duplicable capabilities to be copied, we add the axiom $D <: D * D$, where $D$ has kind DCAP.

$$
\begin{array}{rcll}
\kappa & := & \textsc{val} & \textit{a (duplicable) value type} \\
& | & \textsc{cmp} & \textit{a (linear) computation type} \\
& | & \textsc{dcap} & \textit{a duplicable capability} \\
& | & \textsc{cap} & \textit{a linear capability} \\
& | & \mathbf{T} & \textit{a logical value (an index)} \\
& | & \textsc{fate}\ \mathbf{T}\ \mathbf{R} & \textit{a fate}
\end{array}
$$

$$
\frac{\vdash_{\textsc{cic}}\ \mathbf{T}\ :\ \texttt{Type}}{\mathbf{T}\ \text{is well-formed}}
\qquad\qquad
\frac{\begin{array}{c}\vdash_{\textsc{cic}}\ \mathbf{T}\ :\ \texttt{Type} \\ \vdash_{\textsc{cic}}\ \mathbf{R}\ :\ \mathbf{T} \to \mathbf{T} \to \texttt{Prop} \\ \vdash_{\textsc{cic}}\ \mathbf{R}\ \text{is a preorder}\end{array}}{\textsc{fate}\ \mathbf{T}\ \mathbf{R}\ \text{is well-formed}}
$$

**Figure 14.** Kinds

$$
\begin{array}{rcll}
o & := & \alpha & \textit{variable} \\
& | & o \to o & \textit{arrow} \\
& | & o * o & \textit{conjunction} \\
& | & \exists \alpha :: \kappa . o & \textit{existential quantification} \\
& | & \emptyset & \textit{null capability} \\
& | & \{o : o\} & \textit{ownership of a fate} \\
& | & \langle o : o \rangle & \textit{prediction (observation)} \\
& | & \langle \mathbf{P} \rangle & \textit{logical proposition} \\
& | & \mathbf{t} & \textit{logical value (index)}
\end{array}
$$

**Figure 15.** Types and capabilities

$$
\frac{\alpha :: \kappa \in K}{K \vdash \alpha\ ::\ \kappa}
\qquad
\frac{\begin{array}{c}K \vdash o_1\ ::\ \textsc{cmp} \\ K \vdash o_2\ ::\ \textsc{cmp}\end{array}}{K \vdash o_1 \to o_2\ ::\ \textsc{val}}
\qquad
\frac{\begin{array}{c}K \vdash o_1\ ::\ \kappa \\ K \vdash o_2\ ::\ \textsc{dcap} \\ \kappa \in \{\textsc{val}, \textsc{dcap}\}\end{array}}{K \vdash o_1 * o_2\ ::\ \kappa}
\qquad
\frac{\begin{array}{c}K \vdash o_1\ ::\ \kappa \\ K \vdash o_2\ ::\ \textsc{cap} \\ \kappa \in \{\textsc{cmp}, \textsc{cap}\}\end{array}}{K \vdash o_1 * o_2\ ::\ \kappa}
\qquad
\frac{K,\ \alpha :: \kappa_1 \vdash o\ ::\ \kappa_2}{K \vdash \exists \alpha :: \kappa_1 . o\ ::\ \kappa_2}
\qquad
\frac{}{K \vdash \emptyset\ ::\ \textsc{dcap}}
$$

$$
\frac{\lfloor K \rfloor_{\textsc{cic}} \vdash_{\textsc{cic}}\ \mathbf{t}\ :\ \mathbf{T}}{K \vdash \mathbf{t}\ ::\ \mathbf{T}}
\qquad
\frac{\begin{array}{c}K \vdash o_1\ ::\ \textsc{fate}\ \mathbf{T}\ \mathbf{R} \\ K \vdash o_2\ ::\ \mathbf{T}\end{array}}{\begin{array}{c}K \vdash \{o_1 : o_2\}\ ::\ \textsc{cap} \\ K \vdash \langle o_1 : o_2 \rangle\ ::\ \textsc{dcap}\end{array}}
\qquad
\frac{\lfloor K \rfloor_{\textsc{cic}} \vdash_{\textsc{cic}}\ \mathbf{P}\ :\ \texttt{Prop}}{K \vdash \langle \mathbf{P} \rangle\ ::\ \textsc{dcap}}
\qquad
\frac{K \vdash o\ ::\ \textsc{val}}{K \vdash o\ ::\ \textsc{cmp}}
\qquad
\frac{K \vdash o\ ::\ \textsc{dcap}}{K \vdash o\ ::\ \textsc{cap}}
$$

**Figure 16.** Kind assignment

$$
\begin{array}{ll}
\textsc{fate-create} & \emptyset <: \exists \varphi . \{\varphi : \mathbf{t}\} \\
\textsc{fate-update} & \{\varphi : \mathbf{t}_1\} * \langle \mathbf{t}_1\ \mathbf{R}\ \mathbf{t}_2 \rangle <: \{\varphi : \mathbf{t}_2\} \\
\textsc{obs-create} & \{\varphi : \mathbf{t}\} <: \{\varphi : \mathbf{t}\} * \langle \varphi : \mathbf{t} \rangle \\
\textsc{obs-weaken} & \langle \varphi : \mathbf{t}_2 \rangle * \langle \mathbf{t}_1\ \mathbf{R}\ \mathbf{t}_2 \rangle <: \langle \varphi : \mathbf{t}_1 \rangle \\
\textsc{obs-exploit} & \{\varphi : \mathbf{t}_2\} * \langle \varphi : \mathbf{t}_1 \rangle <: \{\varphi : \mathbf{t}_2\} * \langle \mathbf{t}_1\ \mathbf{R}\ \mathbf{t}_2 \rangle \\
\textsc{obs-join} & \langle \varphi : \mathbf{t}_1 \rangle * \langle \varphi : \mathbf{t}_2 \rangle <: \\
& \exists \alpha_3 :: \mathbf{T} . (\langle \mathbf{t}_1\ \mathbf{R}\ \alpha_3 \rangle * \langle \mathbf{t}_2\ \mathbf{R}\ \alpha_3 \rangle * \langle \varphi : \alpha_3 \rangle)
\end{array}
$$

**Figure 17.** Subtyping axioms: fates

$$
\begin{array}{ll}
\emptyset \equiv \langle \texttt{True} \rangle & \\
\langle \mathbf{P}_1 \rangle * \langle \mathbf{P}_2 \rangle \equiv \langle \mathbf{P}_1 \wedge \mathbf{P}_2 \rangle & \\
\langle \exists \alpha : \mathbf{T} . \mathbf{P} \rangle \equiv \exists \alpha :: \mathbf{T} . \langle \mathbf{P} \rangle & \\
\langle \mathbf{P}_1 \rangle <: \langle \mathbf{P}_2 \rangle & \text{if } \vdash_{\textsc{cic}}\ \mathbf{P}_1 \Rightarrow \mathbf{P}_2
\end{array}
$$

**Figure 18.** Subtyping axioms: propositions