

The essence of monotonic state

Alexandre Pilkiewicz¹ and François Pottier²

¹ INRIA, alexandre.pilkiewicz@inria.fr

² INRIA, 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”. To demonstrate their use, we present a simple yet challenging example, namely monotonic integer counters. We then 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). This offers a foundational explanation of Danielsson’s system and, at the same time, extends it to a calculus with mutable state. Last, we sketch an application to 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.

Terminology: types versus properties. The type system discussed in this paper allows stating logical assertions about data, which serve, for instance, as function pre-conditions, post-conditions, and as object invariants. Thus, types are not just coarse descriptions of the layout of data in memory: they can be fine-grained descriptions of properties of data. Although we stick with the word “type”, one could read, interchangeably, “property of data”, or “assertion about 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*. Roughly speaking, changing the type of a mutable memory location is sound only if there exists just one copy of this type.

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 [1,2,3] 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. In principle, one could view them both as primitive. More economically, the second author has argued in earlier work [4] that the former mode (invariable, uncontrolled objects) can be implemented in terms of the latter mode (variable, controlled objects), provided a primitive mechanism is provided for *hiding* a capability within a certain lexical scope. This mechanism, known as the *anti-frame rule*, can be understood as a way of making the type of an object invariant, and, in return, relaxing the ownership and aliasing restrictions for this object. The anti-frame rule plays an important role in the present paper.

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:

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. Mutable state, it turns out, is not involved at all in this theory of monotonicity.

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 [5] 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.

```

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

```

Fig. 1. An implementation of monotonic counters

```

1 type sc : VAL
2 val mk : unit → sc
3 val read : sc → int ( $0 \leq .$ )
4 val inc : sc → sc

```

Fig. 2. A signature for simple counters

Caveat emptor. We have not, at this time, written a proof of type soundness. We believe that, using a standard syntactic technique, it would be moderately difficult to establish the soundness of the core of our system (§A). Scaling this proof up to the full system used in our examples (§4–§6), however, would be difficult, due to the size and complexity of the system, quite independently of the features introduced in this paper. There has been recent progress in semantic techniques for validating program logics equipped with higher-order frame and anti-frame rules [6]. We plan to investigate whether these techniques could be used to build a model of our type-and-capability calculus.

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 [5] from a purely functional, call-by-need language to an imperative, call-by-value setting (§5). After sketching an application to hash-consing (§6), we give a formal definition of the core of our system (§A), and conclude with discussions of related and future work (§7, §8).

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.

```

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)

```

Fig. 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)

```

Fig. 4. A signature for monotonic counters

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 [7].

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

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

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 $\text{ref } (\text{int } (0 \leq .))$, 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 $\text{int } (0 \leq .)$ 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 one can do.

A signature for tracked counters. In a type-and-capability system [1,2,3], 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* σ . The type of the reference r is written $[\sigma]$, read “at σ ”, and means that r is the single inhabitant of the region σ . Access to r is governed by a capability of the form $\{\sigma : \text{ref}(\text{int } i)\}$, 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, $\{\sigma : \text{ref}(\text{int } (i + 1))\}$. This is a strong update.

In such a system, a signature for *tracked counters* can be defined and implemented (Figure 3). There, $tc\ \sigma\ i$ 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 $\{\sigma : \text{ref}(\text{int } i)\}$. (We write SNG for the kind of singleton regions and CAP for the kind of capabilities.)

The function *mk* has codomain $\exists\sigma.([\sigma] * tc\ \sigma\ 0)$. This means that *mk* returns: (i) a region σ ; (ii) a value, of which nothing is known, except it inhabits σ ; and (iii) a capability $tc\ \sigma\ 0$. This capability guarantees that *the inhabitant of σ 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 $tc\ \sigma\ i$, 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 $tc\ \sigma\ (i + 1)$.

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 $tc\ \sigma\ i$ 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 *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 *mc* i , then, after an application of *inc* to x , the variable x still has type *mc* i , even though the internal state of the counter x has just

changed. More deeply, if some variable y also has type $mc\ i$, then, after this application, y still has type $mc\ i$. 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 $mc\ i$ 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 $mc\ i$, 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 $mc\ (i + 1)$. Furthermore, i remains a lower bound of $j + 1$, so it is sound to continue using x at type $mc\ i$.

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 $\exists i.(mc\ i * (0 \leq i))$.) 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, but 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, enforcing the requirement that every update to a reference is monotonic is not a simple task. This requires the current value of the reference to be known with certainty, so that the new value can be proved greater than the current value. (For instance, the very simple instruction $r := succ !r$ requires proving that the function *succ* does not write r .) This in turn requires some restriction over aliasing and ownership—but our very purpose is to get rid of such restrictions! Therefore, it seems that monotonic references should alternate between two modes: a controlled mode, with access restrictions and precise knowledge of the value, and an uncontrolled mode, without access restrictions and with approximate knowledge of the value. Perhaps one could come up with *ad hoc* typing rules that support this alternation between two modes. Here, however, it is obtained for free, as an effect of using the anti-frame rule.

Third, 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 [4] 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 λ -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. The ambient logic could be, for instance, the Calculus of Inductive Constructions, so that our type-checker ships proof obligations to the Coq proof assistant.

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* φ , together with a *capability*, written $\{\varphi : i\}$, which represents both the *ownership* of the fate φ 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]:

$$ref : \tau \rightarrow \exists \sigma. ([\sigma] * \{\sigma : ref \tau\})$$

When provided with an initial value of type τ , this operation allocates a fresh reference cell in the heap, and returns: (i) a singleton region σ ; (ii) a memory location, of type $[\sigma]$, the single inhabitant of this region; (iii) a capability $\{\sigma : ref \tau\}$, 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 σ , and of a fate φ . 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:

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

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 turns 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 φ will always be at least i* . Equivalently, it could be understood as an *observation* that *the value of φ once was 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 φ , 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. In real life, predictions rarely come true: it can be fun to compare an old prediction with the present time and to find out how different they are. Here, however, predictions *are* true and remain so forever. Comparing an old prediction with the present state is no longer a source of laughs, but becomes an *instructive* activity. When one is *reminded* of an old prediction, which possibly one had forgotten, one *learns* that the present state conforms to what was predicted. This can be 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. Ownership of the fate is retained. 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: $\exists i. (\{ \sigma : \text{ref int } i \} * \{ \varphi : i \})$. Once unpacked, this becomes: $\{ \sigma : \text{ref int } j \} * \{ \varphi : j \}$, 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 $\text{ref int } j$, reading it yields a value of type $\text{int } j$, which, thanks to the proposition $i \leq j$, is a subtype of $\text{int } (i \leq .)$. In other words, the *read* operation produces a value of type $\text{int } (i \leq .)$, 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:

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

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 a fate has been observed both in state i and in state j , then its current state must be a common upper bound of i and j , so it is safe to produce a new observation 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. In our application to hash-consing (§6), for instance, it is not even the case that every two elements i and j admit a common upper bound with respect to \mathbf{R} . This axiom plays a key role there.

4 Application: monotonic counters

We now put everything together and explain how to typecheck monotonic counters. The code, which appears in Figures 5 and 6, 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) an observation, and exploit an observation, 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 [4], 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).

```

1 type mc k = – the type of monotonic counters
2    $\exists \text{obs} : \mathbb{Z} \rightarrow \text{DCAP}.$ 
3    $(\forall i. \text{unit} * \text{obs } i \rightarrow \text{int } (i \leq .)) \times$ 
4    $(\forall i. \text{unit} * \text{obs } i \rightarrow \text{unit} * \text{obs } (i + 1)) *$ 
5   obs k
6
7 let mk : unit  $\rightarrow$  mc 0 =
8  $\lambda().$ 
9   let fate  $\varphi : \text{FATE } \mathbb{Z} (\leq) = 0$  in
10  got cap {  $\varphi : 0$  }; – we own the fate, in state 0
11  let cap obs i =  $\langle \varphi : i \rangle$  in – a notation for observations
12  make obs 0; – make an initial observation
13  let  $\sigma, (r : [\sigma]) = \text{ref } 0$  in – allocate a fresh reference
14  got cap {  $\sigma : \text{ref int } 0$  }; – we own the reference, in state 0
15
16  let methods : – build a vector of methods
17   $(\forall i. \text{unit} * \text{obs } i \rightarrow \text{int } (i \leq .)) \times$ 
18   $(\forall i. \text{unit} * \text{obs } i \rightarrow \text{unit} * \text{obs } (i + 1)) =$ 
19
20  hide I = – I is visible only to the methods
21   $\exists j. (\{\sigma : \text{ref int } j\} * \{\varphi : j\})$ 
22  outside of
23  pack cap I; – this establishes the invariant
24
25  let read :  $\forall i. \text{unit} * \text{obs } i * I \rightarrow \text{int } (i \leq .) * I =$ 
26   $\lambda().$ 
27    let j = unpack cap I in
28    got cap obs i *  $\{\sigma : \text{ref int } j\} * \{\varphi : j\};$ 
29    exploit obs i; – this yields  $i \leq j$ 
30    let c : int j =  $!r$  in
31    let c : int( $i \leq .$ ) = c in – a subsumption step
32    pack cap I;
33    c
34  and inc :  $\forall i. \text{unit} * \text{obs } i * I \rightarrow \text{unit} * \text{obs } (i + 1) * I =$ 
35   $\lambda().$ 
36    let j = unpack cap I in
37    got cap obs i *  $\{\sigma : \text{ref int } j\} * \{\varphi : j\};$ 
38    exploit obs i; – this yields  $i \leq j$ 
39    r :=  $!r + 1$ ;
40    set fate  $\varphi := j + 1$ ; – a monotonic update
41    got cap obs i *  $\{\sigma : \text{ref int } (j + 1)\} * \{\varphi : j + 1\};$ 
42    make obs (i + 1); – permitted, since  $i + 1 \leq j + 1$ 
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

```

Fig. 5. Monotonic counters

```

48 let read :  $\forall k. mc\ k \rightarrow int\ (k \leq \cdot) =$ 
49  $\lambda(c : mc\ k).$ 
50 let obs, methods = unpack c in
51 let (read,  $\_$ ) = methods in
52 got cap obs k;
53 read ()
54
55 let inc :  $\forall k. mc\ k \rightarrow mc\ (k + 1) =$ 
56  $\lambda(c : mc\ k).$ 
57 let obs, methods = unpack c in
58 let ( $\_$ , inc) = methods in
59 got cap obs k;
60 inc ();
61 got cap obs (k + 1);
62 pack methods as mc (k + 1)

```

Fig. 6. Monotonic counters (cont.)

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).

In the prologue (lines 9–14), a fresh fate φ and reference r are allocated. The notation $obs\ i$ is introduced for an observation of the fate φ 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 φ 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** [4] 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 φ are incremented. Between these updates, r is in state $j + 1$, while φ 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 φ 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 55–62); *read* is analogous. Unpacking a monotonic counter object of type *mc* k (line 57) yields an abstract observation constructor *obs*, a *methods* vector, and an observation *obs* k (line 59). We extract the *inc* method (line 58) and invoke it (line 60). This invocation is legal, because we hold *obs* k , and produces a new observation *obs* ($k + 1$) (line 61). 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 exactly the pattern of Pierce and Turner’s encoding of objects [8]. One important difference, though, 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* itself. 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 Figures 5 and 6 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

The way in which we were led to the study of monotonic state was rather indirect. Our initial, long-term goal was to develop a type-based time complexity analysis, that is, a type system that allows time complexity assertions to be expressed and checked.

Credits. Tarjan [9] 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 [10] 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 zero immediate cost. The suspension must then be paid for, possibly in several increments, before it can be forced.

In this approach, suspensions can be freely shared. This can lead to paying for a thunk twice, which is a waste but remains sound. This can also lead to forcing a single thunk twice, which is sound as well, because, thanks to memoization, all but the first attempt to force a thunk have zero actual cost.

Okasaki’s approach has been formalized and proved correct by Danielsson [5], 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 must be static entities, which do not exist at runtime, and must be linear. This is exactly what capabilities are. Thus, we introduce a new primitive capability: if n has kind \mathbb{N} , then $n\mathcal{S}$ has kind CAP: it is a capability that 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 $(n+p)\mathcal{S} \equiv n\mathcal{S} * p\mathcal{S}$, 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 [11], every recursive function call steps the clock; in Cray and Weirich’s system [12], every function call steps the clock; in Danielsson’s system [5], every function definition must be “ticked”. In the code that follows, we choose to ignore this aspect and use a normal, zero-cost application rule, as this helps focus on the key idea, namely the use of monotonic state. It would be possible to modify the code so that it is well-typed in the presence of the “costly application” rule.

A type-and-capability system, equipped with credits-as-capabilities and with a “costly application” rule, 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. A thunk *is* such a data structure: as partial payments are made, more credits are stored; when the thunk is forced, the stored amount drops to zero. Furthermore, thunks *must not* be subject to linearity restrictions. As a result, such a system seems unable to express thunks.

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.

A signature for thunks. A signature for thunks appears in Figure 7. A thunk is parameterized with its cost n (a “debit”) and with its result type α (line 1). The type $\text{thunk } n\alpha$ 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 α that consumes n credits (line 2). At any time, a

```

1 type think:  $\mathbb{N} \rightarrow \text{VAL} \rightarrow \text{VAL}$ 
2 val mk:  $\forall n, \alpha. (\text{unit} * n\$ \rightarrow \alpha) \rightarrow \text{think } n \alpha$ 
3 val pay:  $\forall n, p, \alpha. \text{think } n \alpha * p\$ \rightarrow \text{think } (n - p) \alpha$ 
4 val force:  $\forall \alpha. \text{think } 0 \alpha \rightarrow \alpha$ 

```

Fig. 7. A signature for thunks

```

1 type think  $n \alpha =$            – the type of thunks
2    $\exists \text{obs} : \mathbb{N} \rightarrow \text{DCAP}.$ 
3    $((\forall n, p. \text{unit} * \text{obs } n * p\$ \rightarrow \text{unit} * \text{obs } (n - p)) \times$ 
4      $(\text{unit} * \text{obs } 0 \rightarrow \alpha)) *$ 
5      $\text{obs } n$ 
6
7 type state  $n \alpha =$            – the internal state of a thunk:
8   | White  $(\text{unit} * n\$ \rightarrow \alpha)$  – not yet evaluated
9   | Gray  $(\text{unit})$            – in the course of being evaluated
10  | Black  $(\alpha)$            – evaluated

```

Fig. 8. Thunks: internal type definitions

think 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).

Implementation. The definition of the type *think* closely follows the pattern that was introduced for monotonic counters. According to Figure 8, 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 α ; 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 [4], which explained how to exploit a hidden reference but did not include a time complexity aspect. Here, we add a hidden “piggy bank”, where credits are inserted by *pay*, and that is broken by *force*.

The implementation of thunks appears in Figures 9 and 10. In the prologue (lines 3–8), we introduce a fate φ 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* and the suspended computation *userfun*.

```

1 let  $mk : \forall n, \alpha. (unit * n\$ \rightarrow \alpha) \rightarrow think\ n\ \alpha =$ 
2  $\lambda(userfun : unit * n\$ \rightarrow \alpha).$ 
3 let  $fate\ \varphi : FATE\ \mathbb{N}\ (\geq) = n\ \mathbf{in}$  - a decreasing fate
4 got cap  $\{\varphi : n\};$ 
5 let cap  $obs\ i = \langle \varphi : i \rangle\ \mathbf{in}$ 
6 make  $obs\ n;$ 
7 let  $\sigma, (r : [\sigma]) = ref\ (White\ userfun)\ \mathbf{in}$ 
8 got cap  $\{\sigma : ref\ (state\ n\ \alpha)\};$ 
9
10 let  $methods :$ 
11  $(\forall n, p. unit * obs\ n * p\$ \rightarrow unit * obs\ (n - p)) \times$ 
12  $(unit * obs\ 0 \rightarrow \alpha) =$ 
13
14 hide  $I =$ 
15  $\exists nc, ac. \{\sigma : ref\ ((state\ nc\ \alpha) \otimes I)\} * ac\$ * \{\varphi : nc - ac\}$ 
16 outside of
17 pack cap  $I;$  - the witnesses are n and 0
18
19 let  $pay : \forall n, p. unit * obs\ n * p\$ * I$ 
20  $\rightarrow unit * obs\ (n - p) * I =$ 
21  $\lambda().$ 
22 let  $nc, ac = \mathbf{unpack\ cap}\ I\ \mathbf{in}$ 
23 got cap
24  $obs\ n *$ 
25  $\{\sigma : ref\ ((state\ nc\ \alpha) \otimes I)\} *$ 
26  $(ac + p)\$ *$  - our new, combined credit
27  $\{\varphi : nc - ac\};$ 
28 exploit  $obs\ n;$  - this yields  $n \geq nc - ac$ 
29 set fate  $\varphi := nc - (ac + p);$  - a monotonic update
30 make  $obs\ (n - p);$  - uses  $n - p \geq nc - (ac + p)$ 
31 pack cap  $I$  - witnesses:  $nc$  and  $ac + p$ 

```

Fig. 9. Thunks

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. It 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 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 [4].

```

32 and force : unit * obs 0 * I → ( $\alpha \otimes I$ ) * I =
33  $\lambda()$ .
34 let nc, ac = unpack cap I in
35 got cap
36 obs 0 *
37 { $\sigma$  : ref ((state nc  $\alpha$ )  $\otimes$  I)} *
38 ac$ *
39 { $\varphi$  : nc - ac};
40 exploit obs 0; - this yields nc - ac = 0
41 match !r with - whence ac ≥ nc
42 | White (userfun : unit * nc$ * I → ( $\alpha \otimes I$ ) * I) →
43 r := Gray ();
44 got cap
45 { $\sigma$  : ref ((state 0  $\alpha$ )  $\otimes$  I)} *
46 nc$ * - the necessary credit
47 (ac - nc)$ * - any leftover credit
48 { $\varphi$  : 0};
49 pack cap I; - witnesses: 0 and ac - nc
50 got cap nc$ * I;
51 let v :  $\alpha \otimes I$  = userfun () in
52 got cap I;
53 r := Black v;
54 v
55 | Gray () → fail
56 | Black v → v
57 in (pay, force)
58 in
59 got cap obs n;
60 pack methods as think n  $\alpha$ 

```

Fig. 10. Thunks (cont.)

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 7, line 3), could be declared as coercions. This would ensure that *pay* has no runtime cost whatsoever, and, more importantly, would allow *pay* to be used under a linear, covariant context, a feature that is present in Danielsson’s system [5, §11]. Coercions could also serve to expose the fact that the type *think n* α is covariant in *n* and in α .

The *force* method exploits the observation *obs* 0 (line 36) to determine that the available credit *ac* exceeds the necessary credit *nc* (line 40). This allows *nc* credits to be taken out of the piggy bank and consumed in the call to *userfun* (lines 50–52). Note that, prior to invoking *userfun*, the invariant *I* must be re-

established (line 49). 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 implementation of the external functions *pay* and *force* (whose types appear in Figure 7, lines 3–4) is not shown. As before (§4), they are just wrappers in the style of Pierce and Turner [8].

6 Application: hash-consing

We now sketch 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 $data \rightarrow data$, 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 $data \rightarrow int$, 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.

```

1 logic val h: data  $\rightarrow$   $\mathbb{Z}$ 
2 logic property: injective h
3 val hash:  $\forall d. data\ d \rightarrow int\ (h\ d)$ 

```

Fig. 11. An ideal signature for hash-consing

```

1 type  $\varphi$ : FATE ifmap  $\subseteq$ 
2 val hash:  $\forall d. data\ d \rightarrow \exists i. int\ i * \langle \varphi : [d \mapsto i] \rangle$ 

```

Fig. 12. A novel, pragmatic signature for hash-consing

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 11, 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.

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. In the following, 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, one creates a fate, say φ , of kind FATE *ifmap* (\subseteq), and sets up a hidden invariant that relates the content of the mutable table with this fate. (This requires checking that the table remains injective and grows with time.) Now, contrary to our earlier examples (§4, §5), the existence of φ is not hidden. Instead, it is exposed in the signature (Figure 12, 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 12, 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.

Is it possible to argue that a typical imperative implementation satisfies this new signature? We believe so, although we lack space to provide details. (In particular, the prediction $\langle \varphi : [d \mapsto i] \rangle$ is produced by weakening an observation of the entire table down to a singleton map.)

How expressive is the new signature? 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 $\langle \varphi : [d_2 \mapsto i_2] \rangle$. By joining these predictions (§3), she obtains the existence of an injective finite map *m* such that both $[d_1 \mapsto$

$i_1] \subseteq m$ and $[d_2 \mapsto i_2] \subseteq m$ hold. This is sufficient to prove the propositions $d_1 = d_2$ and $i_1 = i_2$ logically equivalent, as desired.

In summary, whereas the ideal signature of Figure 11 claims that *hash* implements a fixed injective mathematical function, the pragmatic signature of Figure 12 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.

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 φ 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 [13, chapter 7].

Liskov and Wing [14] 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 [17] put forth the idea 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 [18] extend the Spec# program verification system with *history invariants*, as well as with a few other ad hoc features, in order to verify a version of the subject-observer pattern. History invariants are checked in a way that 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 is permitted to 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 .

In Leino and Müller’s experimental programming language Chalice [19], every object o carries a ghost field that contains a snapshot of the heap [20]. A snapshot of the current heap is written to this field whenever certain operations (“fork”, “acquire”, “release”) are applied to o . It is possible to refer to this field in specifications, so that one can assert, for instance, “when o was last acquired, $o.x$ was 0”. Chalice supports history invariants, which it internally translates down to assertions about heap snapshots. Again, this is an instance of introducing ghost variables in order to enable temporal reasoning.

Relational models of monotonic state. Ahmed, Dreyer, and Rossberg [21] consider a call-by-value λ -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, our type-based approach is applicable and, perhaps, offers better potential for integration in a programming language design.

In Ahmed *et al.*’s motivating example [21, 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 integer index must be in the domain of the table (which grows with time), so that no bounds check is necessary. In addition, type generativity is used to guarantee that distinct table instances give rise to distinct instances of the abstract type t .

Ahmed *et al.*’s technique allows (manually) proving that no bounds check is necessary. Our type system allows this as well, 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 φ , ranging over sets of integers, and whose law is set inclusion. As an internal invariant, we assert that φ 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* φ as $\exists i.(int\ i * \langle \varphi : \{i\} \rangle)$. This type plays the role of t in Ahmed *et al.*’s paper.

When supplied by the client with a value of type *index* φ , we confront the observation $\langle \varphi : \{i\} \rangle$ with the current state of φ , 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.

Type abstraction is not essential. Even though the type *index* φ is not abstract, the client cannot forge table indices: because he does not own φ , he cannot

produce observations of the form $\langle \varphi : \{i\} \rangle$. *Type generativity* is still required: each table instance must have its own fate. This is permitted by existential quantification over a fate. The constructor function that creates a fresh table instance and produces a pair of *insert* and *lookup* functions has type:

$$unit \rightarrow \exists \varphi.((string \rightarrow index \varphi) \times (index \varphi \rightarrow string))$$

If desired, this type can be converted to the more abstract $unit \rightarrow \exists \alpha.((string \rightarrow \alpha) \times (\alpha \rightarrow string))$, which corresponds to Ahmed *et al.*'s signature.

In Ahmed *et al.*'s “irreversible state change” example [21, §5.5], the challenge is to prove that $!x$ evaluates to 1, even though the unknown function f might (via a re-entrant call) have a side effect on x :

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

This is easily 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. Ahmed *et al.*'s more challenging examples [21, §5.6, §5.7] can be dealt with using the second author's generalized anti-frame rule [22], either in isolation, or in combination with fates.

Type-based complexity analysis. Although the *automated* time complexity analysis of programs 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.* [23], Reistad and Gifford [24], and Crary and Weirich [12] annotate function types with a worst-case cost. The type-and-capability system that we have sketched (§5) is significantly more expressive. By viewing a time credit as a capability, which can be passed to and returned from a function, stored in a data structure, or made part of a hidden invariant, and by offering flexible mechanisms for reasoning about aliasing, ownership, and monotonicity, we allow non-trivial complexity properties to be checked. This is evidenced, we hope, by our encoding of Okasaki and Danielsson's analysis of thunks. We intend to describe this type-based complexity analysis system in greater depth in a future paper.

The idea that a *space credit* is a linear entity, which can be passed around and stored, is not new: Hofmann [25] 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 [9]. Its use in the setting of a powerful type-and-capability system, however, appears to be new.

8 Future work

Future work includes proving type soundness for this foundational calculus of monotonicity; developing a plausible surface language; and further studying the application of type-and-capability systems to the verification of amortized time complexity assertions.

References

1. Cray, K., Walker, D., Morrisett, G.: [Typed memory management in a calculus of capabilities](#). In: ACM Symposium on Principles of Programming Languages (POPL). (January 1999) 262–275
2. Smith, F., Walker, D., Morrisett, G.: [Alias types](#). In: European Symposium on Programming (ESOP). volume 1782 of Lecture Notes in Computer Science, Springer (March 2000) 366–381
3. Charguéraud, A., Pottier, F.: [Functional translation of a calculus of capabilities](#). In: ACM International Conference on Functional Programming (ICFP). (September 2008) 213–224
4. Pottier, F.: [Hiding local state in direct style: a higher-order anti-frame rule](#). In: IEEE Symposium on Logic in Computer Science (LICS). (June 2008) 331–340
5. Danielsson, N.A.: [Lightweight semiformal time complexity analysis for purely functional data structures](#). In: ACM Symposium on Principles of Programming Languages (POPL). (January 2008)
6. Schwinghammer, J., Yang, H., Birkedal, L., Pottier, F., Reus, B.: [A semantic foundation for hidden state](#). Submitted (October 2009)
7. Xi, H.: [Dependent ML: an approach to practical programming with dependent types](#). Journal of Functional Programming **17**(2) (2007) 215–286
8. Pierce, B.C., Turner, D.N.: [Simple type-theoretic foundations for object-oriented programming](#). Journal of Functional Programming **4**(2) (April 1994) 207–247
9. Tarjan, R.E.: [Amortized computational complexity](#). SIAM Journal on Algebraic and Discrete Methods **6**(2) (1985) 306–318
10. Okasaki, C.: [Purely Functional Data Structures](#). Cambridge University Press (1999)
11. Hehner, E.C.R. In: [Abstractions of Time](#). Prentice Hall (1994) 191–210
12. Cray, K., Weirich, S.: [Resource bound certification](#). In: ACM Symposium on Principles of Programming Languages (POPL). (January 2000) 184–198
13. Schneider, F.B.: [On Concurrent Programming](#). Springer (1997)
14. Liskov, B., Wing, J.M.: [A behavioral notion of subtyping](#). ACM Transactions on Programming Languages and Systems **16**(6) (1994) 1811–1841
15. Leavens, G.T., Baker, A.L.: [Enhancing the pre- and postcondition technique for more expressive specifications](#). In: Formal Methods (FM). volume 1709 of Lecture Notes in Computer Science, Springer (January 1999) 1087–1106
16. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M.: [JML Reference Manual](#). (May 2008)
17. Fähndrich, M., Leino, R.: [Heap monotonic tpestates](#). In: International Workshop on Alias Confinement and Ownership (IWACO). (July 2003)
18. Leino, K.R.M., Schulte, W.: [Using history invariants to verify observers](#). In: European Symposium on Programming (ESOP). volume 4421 of Lecture Notes in Computer Science, Springer (2007) 80–94

19. Leino, K.R.M., Müller, P.: [A basis for verifying multi-threaded programs](#). In: European Symposium on Programming (ESOP). volume 5502 of Lecture Notes in Computer Science, Springer (March 2009) 378–393
20. Leino, K.R.M.: Personal communication (February 2009)
21. Ahmed, A., Dreyer, D., Rossberg, A.: [State-dependent representation independence](#). In: ACM Symposium on Principles of Programming Languages (POPL). (January 2009) 340–353
22. Pottier, F.: [Generalizing the higher-order frame and anti-frame rules](#). Unpublished (July 2009)
23. Dornic, V., Jouvelot, P., Gifford, D.K.: [Polymorphic time systems for estimating program complexity](#). ACM Letters on Programming Languages and Systems 1(1) (1992) 33–45
24. Reistad, B., Gifford, D.K.: [Static dependent costs for estimating execution time](#). In: ACM Symposium on Lisp and Functional Programming (LFP). (1994) 65–78
25. Hofmann, M.: [A type system for bounded space and functional in-place update](#). Nordic Journal of Computing 7(4) (2000) 258–289

A Formalization

We now briefly present a core type-and-capability system equipped with fates and predictions. For the sake of clarity, the system is reduced to a bare minimum. It has existential quantification; a distinction between duplicable and linear capabilities; and logical assertions (viewed as duplicable capabilities). These features are required. Everything else (products, sums, regions, references, hidden state) is orthogonal and described elsewhere [3,4].

Following Charguéraud and Pottier [3], we use an untyped λ -calculus (Figure 13) equipped with a call-by-value operational semantics (omitted).

The kinds are listed in Figure 14. There, the judgement \vdash_{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_{cic} correspond to proof obligations, which, in an implementation, would be shipped to an external theorem prover.

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, which are capabilities, etc. 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.

By convention, we write τ for value types, χ for computation types, D for duplicable capabilities, C for arbitrary (hence, presumably linear) capabilities. 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 typing judgements and typing rules (Figures 17 and 18) are borrowed from Charguéraud and Pottier [3]. We write Δ for a duplicable typing environment, which maps type variables to objects of a kind other than **CMP** and **CAP**. We write Γ for an arbitrary typing environment, which maps type variables to objects of any kind.

The subtyping axioms that govern fates and predictions are summarized in Figure 19. They are subject to the implicit side condition that φ has kind **FATE T R**. The subtyping axioms that govern propositions appear in Figure 20.

The rules that make subtyping a congruence, as well as some administrative rules concerning conjunction and existential quantification [3], are not shown. Last, the subtyping axiom $D <: D * D$, which allows copying a duplicable capability, is added.

$$\begin{array}{l} \text{Values } v := x \mid \mu f. \lambda x. t \\ \text{Terms } t := v \mid v t \end{array}$$

Fig. 13. Syntax of the untyped calculus

$$\begin{array}{l} \kappa := \text{VAL} \quad a \text{ (duplicable) value type} \\ \quad | \text{CMP} \quad a \text{ (linear) computation type} \\ \quad | \text{DCAP} \quad a \text{ duplicable capability} \\ \quad | \text{CAP} \quad a \text{ linear capability} \\ \quad | \mathbf{T} \quad a \text{ logical value (an index)} \\ \quad | \text{FATE } \mathbf{T} \mathbf{R} \quad a \text{ fate} \end{array}$$

$$\frac{\vdash_{\text{CIC}} \mathbf{T} : \text{Type}}{\mathbf{T} \text{ is well-formed}} \qquad \frac{\begin{array}{l} \vdash_{\text{CIC}} \mathbf{T} : \text{Type} \\ \vdash_{\text{CIC}} \mathbf{R} : \mathbf{T} \rightarrow \mathbf{T} \rightarrow \text{Prop} \\ \vdash_{\text{CIC}} \mathbf{R} \text{ is a preorder} \end{array}}{\text{FATE } \mathbf{T} \mathbf{R} \text{ is well-formed}}$$

Fig. 14. Kinds

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

Fig. 15. Types and capabilities

| | | | |
|--|--|--|--|
| $\frac{\alpha :: \kappa \in K}{K \vdash \alpha :: \kappa}$ | $\frac{K \vdash o_1 :: \text{CMP} \quad K \vdash o_2 :: \text{CMP}}{K \vdash o_1 \rightarrow o_2 :: \text{VAL}}$ | $\frac{K \vdash o_1 :: \kappa \quad K \vdash o_2 :: \text{DCAP} \quad \kappa \in \{\text{VAL}, \text{DCAP}\}}{K \vdash o_1 * o_2 :: \kappa}$ | $\frac{K \vdash o_1 :: \kappa \quad K \vdash o_2 :: \text{CAP} \quad \kappa \in \{\text{CMP}, \text{CAP}\}}{K \vdash o_1 * o_2 :: \kappa}$ |
| $\frac{K, \alpha :: \kappa_1 \vdash o :: \kappa_2 \quad \kappa_2 \in \{\text{VAL}, \text{DCAP}, \text{CAP}, \text{CMP}\}}{K \vdash \exists \alpha :: \kappa_1.o :: \kappa_2}$ | $\frac{}{K \vdash \emptyset :: \text{DCAP}}$ | $\frac{[K]_{\text{CIC}} \vdash_{\text{CIC}} \mathbf{t} : \mathbf{T}}{K \vdash \mathbf{t} :: \mathbf{T}}$ | |
| $\frac{K \vdash o_1 :: \text{FATE } \mathbf{TR} \quad K \vdash o_2 :: \mathbf{T}}{K \vdash \{o_1 : o_2\} :: \text{CAP} \quad K \vdash \langle o_1 : o_2 \rangle :: \text{DCAP}}$ | $\frac{[K]_{\text{CIC}} \vdash_{\text{CIC}} \mathbf{L} : \mathbf{Prop}}{K \vdash \langle \mathbf{L} \rangle :: \text{DCAP}}$ | $\frac{K \vdash o :: \text{VAL}}{K \vdash o :: \text{CMP}}$ | |
| $\frac{K \vdash o :: \text{DCAP}}{K \vdash o :: \text{CAP}}$ | | | |

Fig. 16. Kind assignment

| | |
|---|--|
| $\frac{\text{VAR} \quad x : \tau \in \Delta}{\Delta \vdash x : \tau}$ | $\frac{\text{FIX} \quad \Delta, f : \chi_1 \rightarrow \chi_2, x : \chi_1 \Vdash t : \chi_2}{\Delta \vdash \mu f. \lambda x. t : \chi_1 \rightarrow \chi_2}$ |
|---|--|

Fig. 17. Type assignment: values

| | | | |
|---|---|---|--|
| $\frac{\text{VAL} \quad \Delta \vdash v : \tau}{\Delta \Vdash v : \tau}$ | $\frac{\text{APP} \quad \Delta \Vdash v : \chi_1 \rightarrow \chi_2 \quad \Delta, \Gamma \Vdash t : \chi_1}{\Delta, \Gamma \Vdash vt : \chi_2}$ | $\frac{\text{SUB} \quad \Gamma \Vdash t : \chi_1 \quad \chi_1 <: \chi_2}{\Gamma \Vdash t : \chi_2}$ | $\frac{\text{*}-\text{INTRO (FRAME)} \quad \Gamma \Vdash t : \chi}{\Gamma, x : C \Vdash t : \chi * C}$ |
| $\frac{\text{*}-\text{ELIM} \quad \Gamma, x_1 : o, x_2 : C \Vdash t : \chi}{\Gamma, x_1 : o * C \Vdash t : \chi}$ | $\frac{\exists\text{-INTRO} \quad \Gamma \Vdash t : [\alpha \rightarrow o]\chi}{\Gamma \Vdash t : \exists \alpha. \chi}$ | $\frac{\exists\text{-ELIM} \quad \Gamma_1, \alpha :: \kappa, x : \chi_1, \Gamma_2 \Vdash t : \chi_2}{\Gamma_1, x : \exists \alpha :: \kappa. \chi_1, \Gamma_2 \Vdash t : \chi_2}$ | |

Fig. 18. Type assignment: terms

FATE-CREATE $\emptyset <: \exists \varphi :: \text{FATE } \mathbf{T} \mathbf{R}. \langle \varphi : \mathbf{t} \rangle$
 FATE-UPDATE $\langle \varphi : \mathbf{t}_1 \rangle * \langle \mathbf{t}_1 \mathbf{R} \mathbf{t}_2 \rangle <: \langle \varphi : \mathbf{t}_2 \rangle$
 OBS-CREATE $\langle \varphi : \mathbf{t} \rangle <: \langle \varphi : \mathbf{t} \rangle * \langle \varphi : \mathbf{t} \rangle$
 OBS-WEAKEN $\langle \varphi : \mathbf{t}_2 \rangle * \langle \mathbf{t}_1 \mathbf{R} \mathbf{t}_2 \rangle <: \langle \varphi : \mathbf{t}_1 \rangle$
 OBS-EXPLOIT $\langle \varphi : \mathbf{t}_2 \rangle * \langle \varphi : \mathbf{t}_1 \rangle <: \langle \varphi : \mathbf{t}_2 \rangle * \langle \mathbf{t}_1 \mathbf{R} \mathbf{t}_2 \rangle$
 OBS-JOIN $\langle \varphi : \mathbf{t}_1 \rangle * \langle \varphi : \mathbf{t}_2 \rangle <:$
 $\exists \alpha_3 :: \mathbf{T}. (\langle \mathbf{t}_1 \mathbf{R} \alpha_3 \rangle * \langle \mathbf{t}_2 \mathbf{R} \alpha_3 \rangle * \langle \varphi : \alpha_3 \rangle)$

Fig. 19. Subtyping axioms: fates

$\emptyset \equiv \langle \text{True} \rangle$
 $\langle \mathbf{P}_1 \rangle * \langle \mathbf{P}_2 \rangle \equiv \langle \mathbf{P}_1 \wedge \mathbf{P}_2 \rangle$
 $\langle \exists \alpha : \mathbf{T}. \mathbf{P} \rangle \equiv \exists \alpha :: \mathbf{T}. \langle \mathbf{P} \rangle$
 $\langle \mathbf{P}_1 \rangle <: \langle \mathbf{P}_2 \rangle \quad \text{if } \vdash_{\text{cic}} \mathbf{P}_1 \Rightarrow \mathbf{P}_2$

Fig. 20. Subtyping axioms: propositions