

# Types for complexity-checking

François Pottier

May 20th, 2010



I would like to talk about

- how to use types for “*complexity-checking;*”
- how to do this in an expressive programming language, such as ML (with *state*) or Haskell (with *suspensions*).

What is complexity-checking?

- *not* automated complexity analysis (which seems to apply to small functional programs or term rewrite systems);
- *not* implicit computational complexity (which relates typed programming languages and complexity classes);
- it consists in exploiting a type discipline to *check explicit complexity claims* provided by the programmer.

# Complexity-checking: *easy* or *hard*?

Complexity-checking is *hard* in the sense that it demands a lot of information about the program:

- *types* in the simplest sense (e.g., “this is a mutable binary tree”);
- *aliasing* and *ownership* information (e.g., “at any point in time, only one pointer to this binary tree is retained”);
- *logical properties* of data (e.g., “this binary tree is balanced”).

## Complexity-checking: *easy* or *hard*?

On the other hand, I would like to claim that complexity-checking is (relatively) *easy* if one's starting point is a type system (or proof system) that keeps track of this information.

The basic idea, following Tarjan [1985], is to extend the system with *time credits*.

Time credits do not exist at runtime, but appear in types, and are used to control the asymptotic run time of the code.

The recipe is as follows:

- ① Enforce the rule that *credits cannot be created or duplicated*.
- ② Enforce the rule that *every elementary computation step consumes one credit*. (In fact, in the absence of loop forms, it is enough for just function calls to consume one credit.)
- ③ Allow credits to be *passed to* and *returned by* functions.
- ④ Allow credits to be *stored* within data, including mutable data.

Rules 1 and 2 ensure that the total number of steps taken by the program is bounded, up to a constant factor, by the number of credits that are initially made available to it.

With a reasonable compiler, one step in the operational semantics is executed in constant time by the machine code version of the program.

Thus, the number of credits that is made available to the program bounds its *worst-case asymptotic time complexity*.

## Time credits—types are complexity assertions

Allowing credits to serve as function arguments and results (point 3) is required for expressiveness.

As a consequence of it, the complexity of a function can be read off its type. Here are some examples:

- |   |   |
|---|---|
| $\text{int} * 2\$ \rightarrow \text{int}$                                 | — constant time                         |
| $\forall n, \text{int} \times \text{int } n * n\$ \rightarrow \text{int}$ | — linear time in the parameter $n$      |
| $\forall n a, \text{list } n a * 2n\$ \rightarrow \text{int}$             | — linear time in the length of the list |

By construction, the system is *compositional*.



## Time credits—amortized complexity

Viewing credits as data (point 4) does not affect the end-to-end guarantee: the initial number of credits remains a bound on the program's worst-case asymptotic time complexity.

It does, however, change the interpretation of types, which must now be viewed as *amortized* complexity assertions. Credits can be stored for later use, retrieved when needed, and this is not visible in the types.

## Time credits—amortized complexity

Here is the classic example of a FIFO queue, implemented as a pair of lists. Elements are enqueued into the front list, and dequeued out of the back list. Dequeuing may require reversing the elements of the front list and moving them to the back list, a *linear time* operation.

The queue offers this abstract interface:

<code>new_queue:</code>	$\forall a, \text{unit} \rightarrow \text{queue } a$	— constant time
<code>enqueue:</code>	$\forall a, a \times \text{queue } a * 1\$ \rightarrow \text{queue } a$	— constant time
<code>dequeue:</code>	$\forall a, \text{queue } a \rightarrow \text{option } a \times \text{queue } a$	— constant time

Internally, the front list stores one credit together with each element:

$$\text{queue } a = \text{list } (a * 1\$) \times \text{list } a$$

In this example, because credits are not duplicable, the type `queue` inherits this property. These queues are *single-threaded*.

In the rest of this talk, I propose to:

- give an overview of the type-theoretic machinery that I use;
- return to complexity-checking and sketch an analysis of Haskell's *suspensions*.

- A type-checker's armory
  - Affinity
  - Capabilities
  - Regions
  - Other forms of capabilities
- When credits explain debits: an analysis of suspensions
- Conclusion
- Bibliography

## A challenge: reasoning about state

Programs without state, are relatively easy to reason about, because *properties of data are stable*: any logical property that holds now also holds into the future.

Programs that manipulate a heap of mutable objects are much more difficult to reason about: if a property of an object (or group thereof) holds now, how do I guarantee that it still holds at a certain point in the future?

Type system designers have offered answers that rely on a number of technical tools:

- *affinity* ensures the unique ownership of mutable state;
- distinguishing *values* versus *capabilities* enables flexible ownership policies;
- *regions* help keep track of which *capabilities* govern which objects, and can be used to record may-alias information.

In the following, I review these concepts.

(Note: I tend to say “*linearity*” for “affinity.”)

Affinity

How can I soundly make an assertion whose validity depends on the current state of a mutable object, or group thereof?

For instance, one might wish to assert:

- “this reference holds an integer;”
- “this reference holds an even integer;”
- “this group of objects forms a forest.”



The danger is to permit a state change by someone who is not aware of the assertion, and might break it.

A natural solution is to posit that:

- only the owner of an object can *write* it;
- only the owner of an object can *make an assertion* about it;
- an object has *at most one owner*.

This ensures that, when an object is written, all existing assertions about it are at hand. They are invalidated, and (if desired) new assertions about the object are made.

In short, this *affine ownership* discipline is sound and permits *strong updates*.

For simplicity, we posit that only the owner of an object can read it. Because only the owner can make an assertion about an object, a read by a non-owner would produce a value whose type and logical properties are unknown. This would make it useless.

What concrete form do these ideas take? For instance, one could extend a traditional affine type discipline, in the style of Barber's DILL, with *affine references*.

The three primitive operations would be:

$$\begin{aligned} \text{ref} &: \tau \rightarrow \text{ref } \tau \\ ! &: \text{ref } !\tau \rightarrow !\tau \times \text{ref } !\tau \\ := &: \text{ref } \tau_1 \times \tau_2 \rightarrow \text{ref } \tau_2 \end{aligned}$$

Here, a value of type  $\text{ref } \tau$  is the *address* of the reference, but it also represents the *ownership* of the reference and the *assertion* that the reference currently holds a value of type  $\tau$ .

Reading involves duplication, and is restricted to duplicable types.

Writing involves loss, which is fine in an affine system.

Writing allows strong updates.

The affine type system of the previous slide ensures that:

- there is at most one *use* of certain variables;
- there is at most one *pointer* to an object;
- there is at most one *owner* per object.

Here, only the third goal is of interest.

The second restriction is undesirable. It stems from the fact that we have conflated the *pointer* to the object and the *token of ownership* of the object. Only the latter need be affine!

# Capabilities

The address of a reference is just a *value*. It is duplicable.

The token of ownership of a reference is a *capability*. It is affine.

Like a value, a capability can be passed to a function, returned by a function, or can be a component of a value. However, capabilities do not exist at runtime.

By distinguishing values and capabilities, we recognize that reachability and ownership are separate concepts; we allow multiple pointers to an object; and we create a flexible *ownership transfer* mechanism.

Here is an example in a concurrent setting.

Imagine that the address of a memory buffer is shared between two threads. When the first thread is done filling the buffer, it sends a signal to the second thread, which starts processing it.

We can view the “signal” function as consuming a capability for the buffer, and the “wait” function as producing a capability for the buffer, so that, even though no data is transferred, the *ownership of the buffer is transferred* when the two threads synchronize.

My paper with Charguéraud [[2008](#)] presents a type & capability calculus, where every value is duplicable and every capability is affine.

An affine value can be reconstructed, if desired, as a pair of an unrestricted value and of the capability that governs it.



# Type & capability systems versus type & effect systems

Every value is duplicable. In particular, functions are duplicable.

Capabilities, on the other hand, are affine. This implies that a function closure must not capture a capability.

Thus, a function that wishes to read and/or write some object must request a capability for this object as an argument, and usually returns it (otherwise, it is lost).

In summary, *a function's side effects are advertised in its type.*  
Type-and-capability systems subsume type-and-effect systems.

# Type & capability systems versus type & effect systems

Type-and-capability systems are more expressive than type-and-effect systems.

While many functions require a capability and return it, which corresponds to advertising a side effect, some functions:

- require nothing and *produce a capability*; this is the case of memory allocation, lock acquisition, message reception, etc.; or
- *consume a capability* and return nothing; this is the case of memory de-allocation, lock release, message emission, etc.

# Show us a capability, will you?

What concrete form do these ideas take?

I can't answer right away, because I am missing one ingredient.

We have separated values and capabilities, but we do need to keep track of the connection between them. That is, we must be able to answer the question: *which capability governs which object?*

What is needed is a means for a capability to refer to an object. This is usually achieved using either a form of dependency or using regions.

Regions

A *singleton region* is a name for a value.

A value  $v$  has type  $[\sigma]$  (“at  $\sigma$ ”) when  $v$  is the unique inhabitant of the region  $\sigma$ .

This does not reveal anything about the structure of  $v$ : is it a pair, a function, a reference, ...?

This information is carried by the *capability* that controls the region.

For instance, the capability  $\{\sigma : \text{ref } \tau\}$  represents the *ownership* of the region  $\sigma$  and carries the *assertion* that its inhabitant is the address of a reference cell that currently contains a value of type  $\tau$ .

Regions are *monotonic* in the sense that the set of their inhabitants can only grow: if  $v$  inhabits  $\sigma$  now, then this is true forever.

This is required for soundness: because values are duplicable, once we have published the information that  $v$  has type  $[\sigma]$ , there is no way to revoke it.

On the other hand, because capabilities are affine, the information carried by the capability *can* change with time.

The three primitive operations for dealing with references are:

$$\begin{aligned} \text{ref} &: \tau \rightarrow \exists \sigma. ([\sigma] * \{\sigma : \text{ref } \tau\}) \\ ! &: [\sigma] * \{\sigma : \text{ref } \tau\} \rightarrow \tau * \{\sigma : \text{ref } \tau\} \\ := &: ([\sigma] \times \tau_2) * \{\sigma : \text{ref } \tau_1\} \rightarrow \text{unit} * \{\sigma : \text{ref } \tau_2\} \end{aligned}$$

Here, a value of type  $[\sigma]$  is the *address* of the reference. A capability  $\{\sigma : \text{ref } \tau\}$  represents the *ownership* of the reference and the *assertion* that the reference currently holds a value of type  $\tau$ .

Reading involves duplication, and is restricted to duplicable types. Every value type  $\tau$  is duplicable.

Writing allows strong updates.

A *group region*  $\rho$  is a name for a set of values.

There is just one capability for a group region, which represents the ownership of the region and of its inhabitants.

For instance,  $\{\rho : \text{ref int}\}$  represents the ownership of a group region that is populated with integer references. This capability does *not* permit strong updates.



*Other forms of capabilities*

## Logical assertions are capabilities

Logical assertions, such as  $(n' = n + 1)$ , where  $n$  and  $n'$  are type-level natural integers, can be viewed as capabilities.

These capabilities are *duplicable*: a purely logical assertion that is true now is true forever.

A function can require such a capability as an argument, or return one as a result: thus, we re-discover *pre- and post-conditions*.

Logical implication is embedded in the type system, which becomes a proof system. For instance,  $(n = 2n')$  can be turned into  $(n \text{ is even})$ .

Duplicable capabilities *can* be captured by closures: a function is allowed to exploit the logical properties that hold at its definition site.

Time credits can be viewed as *capabilities*.

Just like the capabilities that govern regions, they are affine: credits cannot be duplicated.

Just like the capabilities that govern regions, they can be passed to functions, returned by functions, stored within pairs, stored within references, etc.

Let us view one credit  $1\$$  as a primitive capability.

The function application rule is modified to consume one credit:

$$\frac{\Delta \vdash v : \chi_1 \rightarrow \chi_2 \quad \Delta, \Gamma \vdash t : \chi_1}{\Delta, \Gamma, 1\$ \vdash (v t) : \chi_2}$$

The form  $n\$$ , where  $n$  could be a variable, can be defined. Its properties, such as  $(n_1 + n_2)\$ \equiv n_1\$ * n_2\$$ , can be derived.

Equipped with this type & capability system, extended with logical assertions and time credits, are we ready to go off and check the complexity of interesting programs?

- in principle, yes, I claim so;
- but there remains to implement and show that this is usable! this is by no means trivial;
- furthermore, in some situations, more weaponry is needed.

In the rest of the talk, I wish to concentrate on the last point.

- A type-checker's armory
  - Affinity
  - Capabilities
  - Regions
  - Other forms of capabilities
- When credits explain debits: an analysis of suspensions
- Conclusion
- Bibliography

## An apparent limitation of Tarjan's approach

As pointed out by Tarjan, credits must not be duplicated. For this reason, *Tarjan's amortized data structures are single-threaded.*

In the type & capability system, any data structure that contains credits must be governed by an affine capability.

For instance, we could offer the following interface to an imperative version of the FIFO queue shown earlier: [◀ back](#)

`new_queue:`  $\forall a, \text{unit} \rightarrow \exists \sigma. ([\sigma] * \{\sigma : \text{queue } a\})$

`enqueue:`  $\forall a \sigma, a \times [\sigma] * \{\sigma : \text{queue } a\} * 1\$ \rightarrow \text{unit} * \{\sigma : \text{queue } a\}$

`dequeue:`  $\forall a \sigma, [\sigma] * \{\sigma : \text{queue } a\} \rightarrow \text{option } a * \{\sigma : \text{queue } a\}$

Yet, Okasaki [1999] pointed out that, in a purely functional language with *lazy evaluation*, it is possible to design a variety of data structures that are *persistent, shared*, and nevertheless enjoy interesting *amortized* complexity bounds.



Okasaki suggested reasoning in terms of *debits*, which can be safely duplicated, instead of credits, which must not be duplicated.

His approach was recently explained by Danielsson [2008] in terms of a type system for complexity-checking.

Danielsson's system is extremely simple. It involves neither affinity nor regions: there is *no control of ownership or aliasing*.

Danielsson introduces a primitive type of *thunks*.

*think n a* is the type of a suspended computation whose result has type *a* and whose (amortized) cost is at most *n*.

Because *n* represents a *cost* (a debit, in Okasaki's terms), as opposed to a credit, it is sound to duplicate (a pointer to) a thunk.

Every attempt to force the thunk will then appear to cost *n*, even though in reality, thanks to memoization, only the first attempt has non-zero cost.

Technically, *thunks* are equipped with the following operations:

return:  $\forall a, a \rightarrow \text{thunk } 0 a$

bind:  $\forall m n a \beta, \text{thunk } m a \rightarrow (a \rightarrow \text{thunk } n \beta) \rightarrow \text{thunk } (m + n) \beta$

tick:  $\forall n a, \text{thunk } n a \rightarrow \text{thunk } (n + 1) a$

pay:  $\forall m n a, \text{thunk } n a \rightarrow \text{thunk } m (\text{thunk } (n - m) a)$

*return* and *bind* construct suspended computations.

*tick* consumes one credit; it should be used at every function call.

*pay* allows paying ahead of time so as to decrease the cost of a *thunk*. Its soundness depends upon memoization!

Internally, *thunk*  $n a$  is just  $a$ : this is a phantom type.

I would like to recast Danielsson's system in terms of a strict language, where thunks are built and forced explicitly.

In the type & capability system, equipped with time credits, thunks should offer the following operations:

$$\begin{aligned} \text{mk: } & \forall na, (\text{unit} * n\$ \rightarrow a) \rightarrow \text{thunk } n a \\ \text{pay: } & \forall npa, \text{thunk } n a * p\$ \rightarrow \text{thunk } (n - p) a \\ \text{force: } & \forall a, \text{thunk } 0 a \rightarrow a \end{aligned}$$

Can we *implement* this as a library? This would allow explaining debits in terms of credits.

In the type & capability system presented so far, I (informally) claim that the answer is *negative*.

There are two important reasons why this is so...

## In need of more machinery: problem one

Recall the types of the functions that construct and evaluate thunks:

$$\begin{aligned} \text{mk: } & \forall a, (\text{unit} * n\$ \rightarrow a) \rightarrow \text{thunk } n a \\ \text{force: } & \forall a, \text{thunk } 0 a \rightarrow a \end{aligned}$$

These functions accept a value and return a value: *they do not require or produce any capabilities*. In other words, *these functions claim to have no side effect*. Yet, we know that they must have an effect: `mk` allocates a reference, which `force` reads and writes.

Put another way: because `thunk n a` is a duplicable type, a thunk does not have a unique owner. Yet, its implementation involves a reference, and every reference must have a unique owner.

## In need of more machinery: solution one

A solution is to extend the type & capability system with a new rule, which allows a capability to be *available* within a certain boundary and *hidden* outside of it.

Such a capability becomes an *invariant* that holds whenever the boundary is crossed.

See [[Pottier, 2008](#)].

## In need of more machinery: problem two

Recall the type of the function that pays for a thunk:

$$\text{pay: } \forall n p a, \text{thunk } n a * p\$ \rightarrow \text{thunk } (n - p) a$$

The idea is that credits are accumulated in the thunk, whose apparent cost decreases.

This mechanism is sound only because *the number of credits that remain to be paid can only decrease with time*, and because the parameter  $n$  in “thunk  $n a$ ” is an upper approximation of this number. Yet, nothing in the type system allows expressing or exploiting these properties.



## In need of more machinery: solution two

A solution is to extend the type & capability system with *fates*: (ghost) memory locations whose value must evolve monotonically with time.

Updating a fate involves a *proof obligation*: the update must be monotonic.

In return, confronting the current state of a fate with an *observation* of a previous state yields *new information*: these two states must be in the ordering relation.

See [[Pilkiewicz and Pottier, 2009](#)].

In short, the type & capability system, extended with time credits, hidden state, and monotonic state, *allows implementing thunks as a library*, with the desired interface.

- A type-checker's armory
  - Affinity
  - Capabilities
  - Regions
  - Other forms of capabilities
- When credits explain debits: an analysis of suspensions
- Conclusion
- Bibliography

I have encoded a very simple and elegant system — Danielsson's — into a significantly more complex, lower-level system — a type & capability calculus.

I view this as a testimony of the *expressiveness* of the latter.

I believe that *type-based complexity-checking* is an interesting direction.

The general-purpose layer of the system still requires much work:

- prove the entire system sound;
- design a palatable surface language;
- implement the system, in connection with a theorem prover.




Some complexity-specific aspects also deserve investigation:

- study the “big  $O$ ” notation.

Several authors have proposed type disciplines that include time or space credits. They are not surveyed in these slides; see [[Pilkiewicz and Pottier, 2009](#)].




- A type-checker's armory
  - Affinity
  - Capabilities
  - Regions
  - Other forms of capabilities
- When credits explain debits: an analysis of suspensions
- Conclusion
- Bibliography

(Most titles are clickable links to online versions.)

-  Charguéraud, A. and Pottier, F. 2008.  
*Functional translation of a calculus of capabilities.*  
In *ACM International Conference on Functional Programming (ICFP)*.  
213–224.
-  Danielsson, N. A. 2008.  
*Lightweight semiformal time complexity analysis for purely functional data structures.*  
In *ACM Symposium on Principles of Programming Languages (POPL)*.
-  Okasaki, C. 1999.  
*Purely Functional Data Structures.*  
Cambridge University Press.



## Bibliography]Bibliography

-  Pilkiewicz, A. and Pottier, F. 2009.  
*The essence of monotonic state.*  
Manuscript.
-  Pottier, F. 2008.  
*Hiding local state in direct style: a higher-order anti-frame rule.*  
In *IEEE Symposium on Logic in Computer Science (LICS)*. 331–340.
-  Tarjan, R. E. 1985.  
*Amortized computational complexity.*  
*SIAM Journal on Algebraic and Discrete Methods* 6, 2, 306–318.