Types for complexity-checking

François Pottier

January 31, 2011

Contents

The traditional use of types

Traditionally,

- types are simple *descriptions of data* (think: function types, algebraic data types);
- types are used to guarantee *memory safety* ("well-typed programs do not wrong").

This is true in ocaml, Haskell, Java, and C#, for instance.

The traditional use of types: example

In short, the type of "*rev*" asserts that "*rev*" maps a list to a list.

```
val rev_append: ∀a. list a → list a → list a
let rec rev_append xs1 xs2 =
  match xs1 with
  | [] →
      xs2
  | x1 :: xs1 →
      rev_append xs1 (x1 :: xs2)

val rev: ∀a. list a → list a
let rev xs =
  rev_append xs []
```

I would like to suggest how types can be used for *"complexity-checking"*.

That is, I would like the compiler to *check explicit, programmer-supplied time complexity assertions*, such as: "*rev* operates in linear time with respect to the length of its input".

Not in this talk!

This talk is *not* about:

- automated complexity analysis;
- worst-case execution time (WCET) analysis;
- implicit computational complexity.

The first two are concerned with *inferring* the asymptotic complexity or actual execution time of a single program.

The last is concerned with designing a programming language where all well-typed programs lie within a certain complexity *class*.

Why do this?

Why seek *machine-checked* complexity guarantees? Is it not overkill?

- manual analyses are often incorrect;
- complexity guarantees are not much harder to obtain than correctness guarantees, which today are already required in certain application areas.

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");
- *logical properties* of data (*e.g.*, "this binary tree is balanced").
- *aliasing* and *ownership* information (*e.g.*, "at any point in time, only one pointer to this binary tree is retained");

We need not just type systems, but *type-and-proof systems.*

Complexity-checking: easy or hard?

On the other hand, *if* one's starting point is such a type-and-proof system, *then* (I claim) complexity-checking is conceptually relatively *easy*.

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

In this talk, I will illustrate several complexity analysis techniques that can be formalized using standard type-theoretic technology together with time credits.

Contents

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.

They can be viewed as *capabilities* — permissions to spend one unit of time.

I will write 1$ for one credit.

The basic 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.

See, for instance, Crary and Weirich [2000].

Example: list reverse

Let us look in detail at "*rev*".

In order for a *type* to express the claim that "*rev* operates in linear time with respect to the length of its input", we need:

- a way for a type to refer to the length of a list;
- the ability for function types to indicate how many credits are expected and returned.

Example: list reverse

I will use an algebraic data type of lists indexed by their length.

type *list a n* **where**
[] : $\forall\, a,\ list\ a\ 0$
(::) : $\forall\, a\ n,\ a \times list\ a\ n \longrightarrow list\ a\ (n + 1)$

This could also be written using explicit logical assertions:

type *list a n* **where**
[] : $\forall\, a\ n,\ \langle\ n = 0\ \rangle \longrightarrow list\ a\ n$
(::) : $\forall\, a\ m\ n,\ a \times list\ a\ m * \langle\ n = m + 1\ \rangle \longrightarrow list\ a\ n$

A capability $\langle n = m + 1 \rangle$ can be thought of as a proof of the assertion $n = m + 1$. It is erased at runtime.

See, for instance, Xi [2007].

Example: list reverse

Equipped with indexed lists, we are able to express length information within types:

```
val rev_append: ∀ a m n. list a m → list a n → list a (m + n)
let rec rev_append xs1 xs2 =
  match xs1 with
  | [] →                                  – we may assume: m is 0
      xs2                                 – we must prove: n = m + n
  | x1 :: xs1 →                           – we may assume: m is m' + 1
      rev_append xs1 (x1 :: xs2)          – must prove: m' + (n + 1) = m + n

val rev: ∀ a m. list a m → list a m
let rev xs =
  rev_append xs []                        – we must prove: m + 0 = m
```

The compiler keeps track of which logical assertions hold at each point and emits proof obligations.

Example: list reverse

Let us now move to a system where a function call costs one credit. This forces us to request credits as arguments:

```
val rev_append: ∀ a m n. list a m * m$ ⟶ list a n ⟶ list a (m + n)
let rec rev_append xs1 xs2 =
  match xs1 with
  | [] ⟶
      xs2
  | x1 :: xs1 ⟶                          — we have m' + 1 credits; one pays
      rev_append xs1 (x1 :: xs2)         — for the call, the rest is passed on

val rev: ∀ a m. list a m * (m + 1)$ ⟶ list a m
let rev xs =
  rev_append xs []
```

These types can be read as *worst-case time complexity assertions.*

Informal correctness argument

How do we know that the system is sound?

1. credits can be moved around, but not created or duplicated; furthermore, each β-reduction step costs one credit; so, the number of β-reduction steps that a program can take is bounded by the number of credits that are initially supplied to it.
   — credits count function calls
2. up to a constant factor, the number of steps that a program takes is bounded by the number of β-reduction steps that it takes.
   — at the source level, it is enough to count function calls
3. a reasonable compiler produces machine code that simulates a reduction step in constant time.
   — at the machine level, it is still enough to count function calls

There is no free lunch

It can be difficult to express complexity assertions about complex code.

For instance, this specification of "*map*" is valid but not satisfactory:

val *map*: $\forall a\ b\ n.\ (a \to b) \times list\ a\ n * 2n\$ \to list\ b\ n$

It states (roughly) that "*map f*" has linear time complexity if "*f*" has constant time complexity. This is a restrictive assumption.

There exist better specifications, but they are much more complex.

This simplistic system *does* not support the big-O notation.

Note how "*rev xs*" costs $m + 1$ credits, while "*rev_append xs []*" only costs m credits.

In principle, *existential types* offer a solution. After "*rev*" is defined, it can be wrapped up as follows:

val *rev*: $\exists k_1\ k_2.\ \forall a\ m.\ list\ a\ m * (k_1\ m + k_2)\$ \longrightarrow list\ a\ m$

Contents

A simply-typed FIFO queue

Here is a classic implementation of a FIFO queue in terms of two singly-linked lists:

```
let put (Q (front, rear)) x =
  Q (front, x :: rear)                          — insert into rear list

let get (Q (front, rear)) =
  match front with
  | x :: front →                                — extract out of front list
      Some (x, Q (front, rear))
  | [] →                                        — if front list is empty,
      match rev_append rear [] with             — reverse rear list,
      | x :: rear →
          Some (x, Q (rear, []))                — and make it the front list
      | [] →
          None                                  — if both lists are empty, fail
```

How might we type-check this?

A length-indexed FIFO queue

We define a type of length-indexed queues:

$$\textbf{type}\ \mathit{queue}\ a\ n\ \textbf{where}$$
$$Q : \forall a\ nf\ nr\ n,\ \mathit{list}\ a\ nf \times \mathit{list}\ a\ nr * \langle\ n = nf + nr\ \rangle \longrightarrow \mathit{queue}\ a\ n$$

We could but do not wish to disclose nf and nr in the queue API, because they are implementation details. Only their sum is meaningful with respect to the queue abstraction.

Worst-case analysis of the FIFO queue

We are now able to carry out this analysis:

```
val put: ∀a n. queue a n × a ⟶ queue a (n + 1)
let put (Q (front, rear)) x =
  Q (front, x :: rear)                          — no function call: zero cost

val get: ∀a n. queue a n * (n + 1)$ ⟶ option (a × queue a (n - 1))
let get (Q (front, rear)) =                     — assume: n = nf + nr
  match front with                              — where nf and nr are unknown
  | x :: front ⟶ Some (x, Q (front, rear))
  | [] ⟶
      match rev_append rear [] with             — cost: nr + 1 credits
      | x :: rear ⟶ Some (x, Q (rear, []))
      | [] ⟶ None
```

The best upper bound for *nr* in terms of *n* is *n* itself. Thus, we conclude that "*get*" has worst-case linear time complexity.

Towards an amortized analysis of the FIFO queue

This analysis is sound, but pessimistic.

One would like to argue that reversal is costly but infrequent, so that its cost, *"averaged over a sequence of operations"*, is cheap.

Put another way, each element is moved only once from the front list into the back list, so the cost of reversal *per element inserted* is constant.

Is there a *sound* way of formalizing these arguments?

The answer lies in Tarjan's theory of *amortized complexity* [1985].

We augment our basic recipe 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.
3. Allow credits to be passed to and returned by functions.
4. Allow credits to be *stored* within data.

Rule 4 is new. Rule 1 is unchanged, but takes on new meaning and becomes more difficult to enforce (see ff. slides).

We would like the cost of reversal to be paid for *in advance.*

Thus, we need to accumulate as many credits as there are elements in the rear list.

We define a type of "rich" queues:

type *rqueue a n* **where**
Q : ∀ *a nf nr n*,
list a nf × *list a nr* ∗ ⟨ *n* = *nf* + *nr* ⟩ ∗ *nr*$ ⟶ *rqueue a n*

Amortized analysis of the FIFO queue

We are then able to carry out this analysis:

```
val put: ∀a n. rqueue a n × a * 1$ ⟶ rqueue a (n + 1)
let put (Q (front, rear)) x =    — deconstructing Q yields nr credits
  Q (front, x :: rear)           — constructing Q costs nr + 1 credits

val get: ∀a n. rqueue a n * 1$ ⟶ option (a × rqueue a (n - 1))
let get (Q (front, rear)) =                          — yields nr credits
  match front with
  | x :: front ⟶ Some (x, Q (front, rear))           — costs nr credits again
  | [] ⟶
      match rev_append rear [] with                  — costs nr + 1 credits
      | x :: rear ⟶ Some (x, Q (rear, []))           — costs zero credits
      | [] ⟶ None
```

Both "*put*" and "*get*" appear to have *constant time complexity.*

Informal correctness argument

Of course, a single call to "get" can take linear time in reality, so these types cannot be naïvely interpreted as worst-case time complexity bounds in the usual sense.

Nevertheless, the informal correctness argument that was sketched earlier remains valid. The complexity bounds that the type system derives *for a complete program* are correct *in the worst case.*

These types are *amortized* worst-case time complexity bounds.

A catch

Not every sequence of n queue operations has $O(n)$ cost.

Sequencing n "*put*" operations, yielding a queue of length n, and n "*get*" operations, yielding in the end an empty queue, takes time $O(n)$, as predicted by the type system.

However, sequencing n "*put*" operations, yielding a queue q of length n, then performing n times the operation "*get q*", takes time $O(n^2)$.

What *does* the type system predict in this case?

The system views the second sequence as *ill-typed* and rejects it.

Recall that *credits are affine* — they must not be duplicated.

Furthermore, a queue contains credits, as per our definition:

type *rqueue a n* **where**
Q : ∀ *a nf nr n*,
list a nf × *list a nr* ∗ ⟨ *n* = *nf* + *nr* ⟩ ∗ *nr*$ ⟶ *rqueue a n*

There follows that *queues are affine.* If a queue could be used twice, the credits in it would be used twice as well.

The hard general rule is, *a data structure that contains an affine component must itself be affine.*

A queue can be used *at most once*. In particular, a call to "*put*" or "*get*" *consumes* its argument queue, which becomes invalid, and produces a new queue, which itself can be used at most once.

Thus, a *sequence* of n "*get*" operations is well-typed, but an attempt to repeatedly perform the operation "*get q*" is ill-typed.

Thus, amortization is no magic bullet: it comes with an affinity restriction.

Contents

A pity!

This seems a pity: these FIFO queues are immutable data structures, hence they are *persistent*. Yet, they cannot be shared, or their nice complexity properties are lost.

Is there a way of preserving persistence, while getting rid of the affinity restriction?

The root of the problem seems to be that invoking "*get q*" twice may cause a reversal operation to be performed twice... Can this be helped?

Okasaki [1996] has shown that *memoization*, also known as *lazy evaluation*, offers a solution.

The idea is to contruct a *thunk* (a delayed computation) for the reversal operation and to take advantage of the fact that a thunk is only evaluated once, even if its value is demanded multiple times.

The time credits contained in a thunk are only spent once, even if the thunk is forced multiple times. A thunk is *a non-affine data structure that contains affine time credits*, something not normally permitted.

Planning ahead for memoization

The idea is deceptively simple: making effective use of memoization is in reality quite difficult.

The challenge is to create costly thunks *sufficiently long* before they are demanded.

Okasaki's banker's queue: principle

In the queue example, waiting until the front list is empty to reverse the rear list is *too late.* We must produce the first element of the reversed list *now.* Suspending the computation of the reversal would not help: we would have to force this suspension right away.

We must plan ahead and set up delayed reversal operations long before their result is required.

The idea is to *schedule* — but *not perform* — a reversal as soon as the rear list grows too large — say, larger than the front list.

Okasaki's banker's queue: data structures

Here is Okasaki's code, in ocaml (no fancy types).

```
type 'a stream =
  'a cell Lazy.t   — computation is suspended

and 'a cell =
  | Nil
  | Cons of 'a × 'a stream

type 'a queue = {
  lenf: int;       — length of front list
  f: 'a stream;    — lazy front list
  lenr: int;       — length of rear list
  r: 'a list;      — rear list
}                  — invariant: lenf ≥ lenr
```

Okasaki's banker's queue: basic functions

```
let put q x =
  check { q with                    — rebalance
    lenr = q.lenr + 1;
    r = x :: q.r;                   — insert into rear list
  }

let extract q =
  match extract q.f with            — extract out of front list (forcing a thunk)
  | None →                          — if front list is empty...
      None                          — ...then rear list is empty too
  | Some (x, f) →
      Some (x,
        check { q with              — rebalance
          f = f;
          lenf = q.lenf - 1;
        }
      )
```

Okasaki's banker's queue: rebalancing

The function "*check*" reverses the rear list if it has become too long.

```
let check ({ lenf = lenf ; f = f; lenr = lenr; r = r } as q) =
  if lenf ≥ lenr then
    q                          — invariant holds; nothing to do
  else {                       — we have lenf + 1 = lenr
    lenf = lenf + lenr;
    f = f ++ rev r;            — re-establish invariant
    lenr = 0;
    r = [];
  }
```

We are creating a costly thunk, "*rev r*", but it will be forced only after all elements of "*f*" have been extracted, so its cost is constant per extracted element. *(Do you trust this argument? You shouldn't.)*

Correctness?

This argument was quite subtle and *very informal!*

It *does not* rely on affinity! Queues can be freely aliased.

Okasaki presents it in a much better and more precise way than I did. Yet, his argument is still informal.

Can we machine-check this argument?

Two questions arise:

- What are formal rules for reasoning about thunks?
  — [Danielsson, 2008]
- Can they be deduced from the basic rules that govern credits?
  — [Pilkiewicz and Pottier, 2011]

Types for thunks: creation

What should be the type of the operation that creates a thunk?

We need to be able to set up thunks, or schedule computations, *before* we have enough credits to pay for these computations.

This suggests that creating a thunk should cost just one credit, regardless of the cost of the suspended computation.

Necessarily, the type of the thunk should reflect how many credits *remain to be paid.* Okasaki calls these *debits.*

val *mk*: $\forall\, n\ a.\ (\mathit{unit} * n\$ \longrightarrow a) \longrightarrow \mathit{thunk}\ n\ a$

Types for thunks: forcing

What should be the type of the operation that forces a thunk?

It would be unsound to force a thunk before the debt has been paid off, that is, before the required time credits have been supplied.

Thus, only thunks *with zero debits* may be forced.

Thanks to memoization, the credits that have been supplied are only spent once, even if the thunk is forced multiple times.

val *force*: $\forall\, a.\ \mathit{thunk}\ 0\ a \to a$

Types for thunks: paying

Thunks are created with non-zero cost, but only zero-cost thunks can be forced. How can we ever force a thunk?

The answer is, we must *pay* ahead of time. By spending credits, we decrease the apparent cost of a thunk.

This is a coercion — it does nothing at runtime and costs zero credits. It is really just a hint for the type-checker.

coercion *pay*: $\forall\, a\ n\ p.\ \mathit{thunk}\ n\ a * p\$ \longrightarrow \mathit{thunk}\ (n - p)\ a$

The principle of anticipated payment can take other forms, such as:

coercion *anticipate*:
$\forall\, a\ m\ n\ p.$
$\mathit{thunk}\ m\ (\mathit{thunk}\ (n + p)\ a) \longrightarrow \mathit{thunk}\ (m + p)\ (\mathit{thunk}\ n\ a)$

In a chain of thunks, *debits can be moved towards the front* — so they will be paid earlier.

This idea can be used to explain how, in "*f ++ rev r*", the linear cost of "*rev r*" can be anticipated and considered an extra unit cost for each element of "*f*".

Thus, roughly speaking, we are able to maintain the invariant that each thunk in the front stream has cost one.

Expressiveness?

Are these rules expressive enough?

Yes, it seems. They are reformulations of Danielsson's rules [2008], who was able to type-check Okasaki's queue.

With some effort.

Correctness?

Are these rules sound?

Yes. Danielsson gives a direct soundness proof for a type system where these rules are *axioms*.

Reduction to simpler rules?

Instead of viewing these rules as axioms, can we *deduce* them from the basic rules that govern credits?

That is, can we *define* the type "*thunk n a*", and can we *implement* "*mk*", "*force*", and "*pay*"?

After all, in plain ocaml, thunks can be implemented in terms of references. Can we justify that this implementation is well-typed in our complexity-type-system?

Reduction to simpler rules?

Yes, we can [Pilkiewicz and Pottier, 2011], but we need a more flexible treatment of affinity than I suggested so far.

Hidden state and monotonic state

Two issues arise:

- thunks act as containers for credits, yet they must not be affine;
  — we need hidden state
- in order to ensure that "*n*" in "*thunk n a*" is a correct upper bound for the actual debt, we must express and exploit the fact that the debt can only decrease with time.
  — we need monotonic state

Hidden state and monotonic state

The tools that Pilkiewicz and I offer [2011] are somewhat complex, but are independent of complexity-checking. They are of *general interest* for reasoning about programs with state.

In fact, analogous tools — as well as even more powerful ones — are being developed in concurrent separation logic — see e.g. Dodds *et al.*'s POPL 2011 paper.

Contents

Take-home message

In short, there are bad news and good news:

- *reasoning about programs is in general hard*, especially in the presence of state, and requires advanced tools (capabilities, affinity, logical assertions, hidden and monotonic state, ...);
- *reasoning about time complexity is not much harder!*

Perhaps your favorite system for proving programs could be easily modified to support time credits? Think about it!

Challenges ahead

Further challenges include:

- performing realistic case studies;
- understanding big-O notation;
- reasoning about space complexity in the presence of GC.

Contents

Bibliography I

(Most titles are clickable links to online versions.)

Crary, K. and Weirich, S. 2000.
Resource bound certification.
In *ACM Symposium on Principles of Programming Languages (POPL)*. 184–198.

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. 1996.
Purely functional data structures.
Tech. Rep. CMU-CS-96-177, School of Computer Science, Carnegie Mellon University. Sept.

Bibliography]Bibliography


Pilkiewicz, A. and Pottier, F. 2011.
The essence of monotonic state.
In *Workshop on Types in Language Design and Implementation (TLDI)*.


Tarjan, R. E. 1985.
Amortized computational complexity.
SIAM Journal on Algebraic and Discrete Methods 6, 2, 306–318.


Xi, H. 2007.
Dependent ML: an approach to practical programming with dependent types.
Journal of Functional Programming 17, 2, 215–286.