FORMAL VERIFICATION OF ASYMPTOTIC COMPLEXITY BOUNDS FOR OCAML PROGRAMS

Armaël Guéneau
supervised by François Pottier & Arthur Charguéraud

November 12, 2015

COUNTING PROGRAM STEPS WITH TIME CREDITS

Time complexity can be formalized in separation logic, thanks to *time credits*.

Example of specification:

$$\{\ \texttt{UF N D R} \star \$\,(3 * (\texttt{alpha N}) + 6)\ \}$$

$$\texttt{union x y}$$

$$\{\ \lambda \texttt{z} \Rightarrow \texttt{UF N D (fun w} \Rightarrow \texttt{If R w = R x} \vee \texttt{R w = R y then z else R w)} \star [\texttt{z = R x} \vee \texttt{z = R y}]\ \}$$

Amortized cost for `union`: $3 \times \alpha(N) + 6$.

Counting credits explicitly quickly becomes impractical, compared to using the “$O()$” notation:

- $n^2 \times m + 3nm + 3n + 6m + 5\log(n) + 2\log(m) + 5\log(n)\log(m) + 8$ instead of $O(n^2 \times m)$
- Specifications using explicit credits count are not modular
- Credits count are to be considered up to a constant factor anyway

OUR CONTRIBUTION

We present “CFML+credits+big-Os”, an extension of “CFML+credits” which formalizes (in Coq) the big-O notation, to be used in program specifications.

OUTLINE OF THIS TALK

Formalizing big-Os: challenges and proposed solutions

Proof automation

Case studies

FORMALIZING BIG-OS: CHALLENGES AND PROPOSED SOLUTIONS

BIG-OS TEXTBOOK DEFINITION

Recall the standard textbook definition for “$O()$”:

$$f \in O(g) \equiv \exists c, \exists n_0, \forall n \geqslant n_0, |f(n)| \leqslant c \times |g(n)|$$

Why is this not trivial to formalize?

CHALLENGE 1: BINDING VARIABLES

We often informally write "f is $O(n^2)$".

However $O()$ is a relation on functions, not expressions.

$\Rightarrow$ We should write "f is $O(\lambda n.n^2)$" instead.

How do we handle cost functions with multiple parameters?

```
let fill_rect n m =
  for j = 1 to m do
    for i = 1 to n do
      draw_pixel i j
    done
  done
```

Concrete cost:

$$\begin{aligned} f(n, m) &= m \times (1 + n) + 1 \\ &= m \times n + m + 1 \end{aligned}$$

Is `fill_rect` $O(\lambda(n, m).m \times n)$?

- If n **and** m go to infinity, then indeed $f(n, m) \in O(\lambda(n, m).m \times n)$

What about the asymptotic cost of “`fill_rect 0 m`”?

- Concrete cost: $f(0, m) = m + 1$
- Clearly not $O(\lambda m.m \times 0) = O(0)$

$\Rightarrow$ We cannot reuse the previous asymptotic bound

CHALLENGE 2: GOING TO INFINITY

- Big-O bounds are proved for one given notion of "going to infinity"
- There are multiple, non-equivalent ones

$\Rightarrow$ Let the user choose, while keeping a lightweight notation for the common cases.

CHALLENGE 2 SOLUTION: *FILTERS*, A FORMAL NOTION OF "GOING TO INFINITY"

A filter on a set `A`:

- is of type (`A` → `Prop`)→ `Prop`, named `filter A`;
- represents the set of neighborhoods of infinity;
- must satisfy additional properties, bundled in a `Filter` Coq class.

E.g. the standard filter on $\mathbb{Z}$ is:

```
Definition towards_infinity_Z : filter Z :=
  fun (P : Z → Prop) ⇒ ∃x0, ∀x, x0 ⩽ x → P x
```

CHALLENGE 2 SOLUTION: *FILTERS*, A FORMAL NOTION OF "GOING TO INFINITY"

"*O*()" definition parameterized by a filter *ultimately*:

```
Definition dominated
        (ultimately: filter A)
        (f g : A → Z) :=
  ∃c, ultimately (fun x ⇒ norm (f x) ≤ c * norm (g x)).
```

We use Coq typeclasses to allow the filter to be inferred in standard cases.

CHALLENGE 2 SOLUTION: FILTERS ON $\mathbb{Z}^2$

What were the filters involved in our `fill_rect` example?

- "Both components go to infinity":

```
Definition towards_infinity_ZZ :=
  fun (P: Z*Z → Prop) ⇒
    ∃P1 P2, towards_infinity_Z P1 ∧
            towards_infinity_Z P2 ∧
    ∀x1 x2, P1 x1 → P2 x2 → P (x1, x2)
```

- "The first component is fixed to `x0`, the second goes to infinity":

```
Definition towards_infinity_xZ (x0: Z) :=
  fun (P: { p: Z*Z | fst p = x0 } → Prop) ⇒
    towards_infinity_Z (fun y ⇒ P (x0, y))
```

CHALLENGE 3: EXISTENTIAL QUANTIFICATIONS

“The cost of `p` is $O(g)$” hides an additional existential quantification.

“The cost of `p` is $O(g)$” is in fact “*there exists* a cost function f st. $f \in O(g)$ and running `p(n)` takes $f(n)$ steps”.

- Convenient informal notation
- But more error prone: some incorrect proofs are harder to detect syntactically

```
let rec loop n =
  if n <= 0 then () else loop (n-1)
```

Lemma (incorrect)

The asymptotic complexity of `loop` *is* $O(1)$.

Proof.

(flawed, but not so obviously). By induction on n,

- $n \leqslant 0$: `loop` terminates in $O(1)$;
- $n \geqslant 1$: the cost of `loop(n)` is the cost of `loop(n-1)` plus $O(1)$. By induction, the cost of `loop(n-1)` is $O(1)$. $O(1) + O(1) = O(1) \Rightarrow$ total cost of $O(1)$.

The mistake: an invalid quantifier permutation.

- “*there exists* a cost function f st. *for all* n, …”, is not
- “*for all* n, *there exists* a cost function f …”.

The explicit cost function must be instantiated *before* entering the induction.

Coq is able to reject this kind of incorrect reasoning; the challenge is to keep a lightweight presentation.

Challenge 3 (imperfect) solution

We define `SpecO`, in order to write specifications using big-Os:

```
Definition SpecO (ultimately: filter A)
  (g: A → Z) (spec: (A → Z) → Prop)
:=
  ∃(f: A → Z), dominated _ f g ∧ spec f.
```

$$\forall n, \{\$\ (3 * n^2 + 2 * n + 5) \star H\}\ t(n) \{Q\}$$

becomes

$$\mathtt{SpecO}\ \_\ (\lambda n \Rightarrow n^2)(\lambda F \Rightarrow \forall n, \{\$\ F\ n \star H\}\ t(n)\{Q\})$$

Remark: arguments of the cost function do not have to be the arguments of the program.

Example: specification for `List.length`

$\forall$`l`,
`SpecO _` $(\lambda$`n` $\Rightarrow$ `n`$)$ $(\lambda$`F` $\Rightarrow$
  $\{\$$ `F` $($`length l`$)\}$ `List.length l` $\{\lambda$`n` $\Rightarrow [$`n` $=$ `length l`$]\})$

CHALLENGE 3 (IMPERFECT) SOLUTION

It does not cover all usages though, e.g. quantifying over a class of filters for the same cost function.

$\exists$(`f`: `A` $\rightarrow$ `Z`),
  ($\forall$ `x0`, `dominated` (`towards_infinity_xZ` `x0`) `f` `g`) $\wedge$
  `spec f`

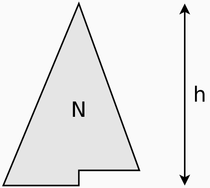
$\Rightarrow$ More general version of `SpecO` parameterized by any relation on f, g.

CHALLENGE 4: MONOTONIC COST FUNCTIONS

Paper proofs assume extensively that cost functions are non-decreasing.

CHALLENGE 4: MONOTONIC COST FUNCTIONS

Example:



$$h \leqslant \log(N) + 1$$

$$\{\$\ \texttt{F}(\texttt{h})\}\ \mathrm{p}\ \{\} \quad \longrightarrow \quad \{\$\ \texttt{G}(\texttt{N})\}\ \mathrm{p}\ \{\}$$

$$F \in O(\lambda h.h) \qquad\qquad G \in O(\lambda N.\log(N))$$

$$\texttt{G}(\texttt{N}) := \texttt{F}(\texttt{log}(\texttt{N}) + 1)$$

⇒ We need to prove `F(h)` ⩽ `G(N)`.

⇒ We need `F` to be non-decreasing.

```
Definition SpecO (ultimately: filter A) le
  (g: A → Z) (spec: (A → Z) → Prop)
:=
  ∃(f: A → Z),
    (∀ x, 0 ⩽ f x) ∧
    monotonic _ _ f ∧
    dominated _ f g ∧
    spec f.
```

challenge 5: o(0) and undesirable side-conditions

We would like to have:

"if f is $O(g)$, then $f + c$ is also $O(g)$ (with c a constant)".

Yet, this is false for $g = 0$.

CHALLENGE 5: O(0) AND UNDESIRABLE SIDE-CONDITIONS

We would like to have:

"if f is $O(g)$, then $\lambda n.\log(f(n))$ is $O(\lambda n.\log(g(n))$".

Yet, this is false for $g = 1$ and $f \geqslant 2$.

Alternative notion of $O()$: `idominated`.

- Matches `dominated` on the interesting cases: when costs functions go to infinity;
- Handles more pathological cases.

```
Definition idominated
      (ultimately: filter A) (leA: A → A → Prop)
      (f g : A → Z)
:=
  ultimately (monotonic_after leA leZ g) ∧
  ((bounded _ f ∧ bounded _ g) ∨dominated _ f g).
```

The following lemmas are now true:

```
idominated _ _ f g →
idominated _ _ (fun n ⇒ c + f n) g
```

```
idominated _ _ f g →
idominated _ _ (fun x ⇒ Z.log2 (f x))
               (fun x ⇒ Z.log2 (g x))
```

We also adapt `SpecO` to use `idominated` in place of `dominated`.

PROOF AUTOMATION

Goal-directed tactics to solve / simplify `idominated`, `monotonic`, `monotonic_after` goals.

Able to prove or simplify automatically goals involving $+, \times, \log, \hat{}$.

```
Goal idominated _ _
  (fun n ⇒ 5 * Z.log2 (3 * n + 2) + 8) Z.log2.
  Proof. idominated_Z_auto; math. Qed.
```

PROOF AUTOMATION

Auxiliary tactics to deal with η-equivalence for n-ary functions (still imperfect).

- We have to reason modulo η-equivalence.
  - $O(\log)$ vs $O(\lambda n.\log(n))$
  - $f \in O(h) \Rightarrow g \in O(h) \Rightarrow \lambda n.f(n) + g(n) \in O(h)$
- Not automatic on n-ary (uncurried) functions.
  - They are of the form $\lambda p.\text{let } (n, m) \ = \ p \text{ in } \ldots$
  - $f \in O(h) \Rightarrow g \in O(h) \Rightarrow$
    $\lambda p.(\text{let } (n, m) \ = \ p \text{ in } f((n, m)) + g((n, m))) \in O(h)$

WIP: a set of tactics to elaborate the cost function through the proof.

```
========================
 SpecO _ _ (λn ⇒ n) (λF ⇒ spec F)
```

```
xcfO (fun n ⇒ 3 * n + 12).              xcfO. [...]
[...]                               →   add_credits (λn ⇒ 1). [...]
                                        add_credits (λn ⇒ 2 * n).
                                        [...]
```

CASE STUDIES

We used the resulting library to formalize two non-trivial data structures:

- Dynamic Arrays, an imperative structure with amortized $O(1)$ costs;
- Binary Random Access Lists, a purely functional data structure with $O(\log n)$ costs, parameter transformation and filters on $\mathbb{Z}^2$.

Why a parameter transformation and filters on $\mathbb{Z}^2$?

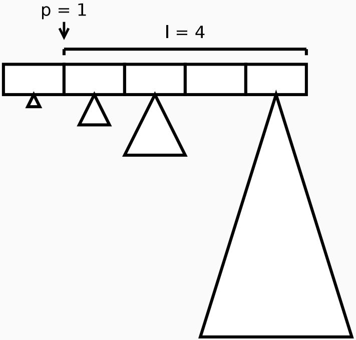


Figure 1: Induction for `lookup` and `update`

CONCLUSION: SOME NUMBERS

- Binary Random Access Lists:
  - Code: 80 lines, proof: 630 lines
  - Whole complexity analysis (credits + big-Os): $\simeq$ 40%
  - Reasoning on big-Os: $\simeq$ 25%
- Dynamic Arrays:
  - Code: 95 lines, proof: 520 lines
  - Whole complexity analysis (credits + big-Os): $\simeq$ 50%
  - Reasoning on big-Os: $\simeq$ 6%)
- Size of the library: $\simeq$ 2300 lines of Coq
  - `dominated`, `idominated` (definition, lemmas, tactics): 1260 lines
  - Filters (definitions, instances): 730 lines
  - Monotonicity (tactics): 250 lines
  - `SpecO` (definition, lemmas, tactics): 70 lines