

Formal Verification of Asymptotic Complexity Bounds for OCaml Programs

MPRI M2 Internship Report, March-August 2015

Armaël Guéneau, supervised by François Pottier and Arthur Charguéraud

Inria Paris-Rocquencourt

General context

Program verification covers a wide range of techniques, frameworks and tools, interested in proving various properties about real world programs. Beyond basic safety properties (e.g. memory safety), program verification can be used to establish full functional correctness, meaning that, for every input, the program always behaves as expected. Yet, full functional correctness does not capture all the desired properties of a program. In particular, it does not give any form of guarantee on the execution time of the program. How useful would a mechanically-correct program be, if it does not deliver its output within reasonable amount of time?

Estimating the real-life execution time of a program can be quite difficult given the complexity and underspecification of modern hardware, operating systems or even compilers. Without going that far, researchers have investigated the possibility of formally establishing asymptotic complexity bounds for programs. For example, some tools, such as RAML by Hoffmann and Hofmann [10], are capable of automatically inferring asymptotic bounds, but only for restricted classes of programs. Other lines of work (Danielsson's THUNK library [7], Charguéraud's CFML [4,3]) allow to establish execution cost bounds for arbitrary complex programs, thanks to the support of a proof assistant. However, they do not support reasoning on asymptotic complexity using Landau's big-O notation.

Problem studied

In a recent publication, Charguéraud and Pottier [5] extend earlier work on CFML with the notion of *time credits* in order to allow establishing bounds on the execution costs. They applied their approach to verify an OCaml implementation of Tarjan's union-find algorithm. The proof covers two aspects: first, functional correctness, but also asymptotic complexity. For example, one of the results of the paper is that the `link` function runs in $\alpha(n) + 2$ elementary steps, where α is the inverse of the Ackermann function.

Working with explicit cost functions, as opposed to using the big-O notation, although it was manageable for the union-find data structure, quickly becomes impractical in general. Cost functions such as $n^2 + 3n \log(n) + 4n + 5$ are already hard to read, while we could simply write $O(n^2)$ instead. The problem gets even worse when multiple variables are involved, when one needs to write $n^2 \times m + 3nm + 3n + 6m + 5 \log(n) + 2 \log(m) + 5 \log(n) \log(m) + 8$ instead of just $O(n^2 \times m)$.

Reasoning with big-Os is also more modular: when proving a complex program that uses various auxiliary functions, big-O bounds allow the implementation details (and the exact cost) of an auxiliary function to change, as long as its asymptotic bound remains the same.

Another interest of formalizing big-Os is that it enables us writing and proving high-level theorems about asymptotic complexity, like the very handy Master Theorem [6]. The master theorem gives asymptotic bounds for broad classes of cost functions that satisfy a recurrence relation. For example, if T is a cost function satisfying $T(n) \leq T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + a \log(n) + b$, then the master theorem can be applied, and tells us that T admits a $O(n)$ bound.

To sum up, using big-O notation brings at least four major benefits: more concise bounds, reasoning closer to paper proofs, improved modularity, and convenient high-level theorems.

Proposed contributions

During my internship, I developed a Coq library to formalize the big-O notation, I extended CFML to integrate big-O notations in specifications, and I considered two significant case studies to validate the approach. If Landau’s big-Os are not particularly reputed to be a complex mathematical object, we will see that they do hide some subtle aspects, and tend to be manipulated in quite informal ways, as described in section 1. The resulting tool, namely CFML with credits extended with support for big-O notation, appears to be the first proposal of a tool that allows to establish asymptotic bounds on the execution cost, for arbitrarily-complex programs.

Arguments supporting their validity

To demonstrate the features and usage of our library, we selected some OCaml programs that we proved correct, as case studies. After basic introductory examples, we study the implementation of two data structures, which constitute self-contained but real-world examples. Each structure features a particular challenge, allowing us to demonstrate the strengths of our library on these aspects. More precisely, we present (1) a basic example of constant-time function manipulations (in 4.1); (2) a basic example of recursive program with exponential running time (in 4.2); (3) implementation of dynamic arrays, featuring amortized $O(1)$ costs (in 4.3); (4) implementation of binary random access lists, with $O(\log(n))$ costs (in 4.4).

These case studies validate our work as they present, first, concise specifications, thanks to appropriate notations. A specification which was using e.g. an explicit cost “ $3 \log(n) + 2$ ” can now introduce a $O(\log(n))$ abstract cost function F , and use “ $F(n)$ ” instead. Secondly, our specifications compose well, justifying big-Os modularity. Finally, the proofs are of reasonable size, compared to the proofs of functional correctness only: adding the complexity analysis does not increase the total size much.

Summary

During the course of this work, we had to overtake the following challenges: (1) understand the implicit assumptions hidden behind the big-O notation; (2) formalize the definition of big-O in Coq, including these implicit assumptions; (3) develop concise notations for specifications to include big-O expressions (which is challenging due to the fact that the underlying constant needs to be quantified outside of the specification); (4) develop lemmas and tactics for automating to a large degree the process of manipulating concrete cost expressions and consume time credits; (5) carry out several case studies to demonstrate the interest and practicality of the approach—successfully addressing the challenge described by the internship proposal.

Our formalization also tackles various subtleties and difficulties inherent to the manipulation of big-Os, which are shed to light in the “Challenges” section (section 1).

Future work

In future work, we would like to improve the degree of automation, possibly through the development of a specific Coq plugin. In particular, we would like to enhance the degree of inference achieved by the tool, in order to reduce the number of places where the user needs to provide concrete cost functions explicitly.

Besides, our current work does not use the master theorem. In future work, we wish to prove the master theorem (the work of Drmota and Szpankowski [9] offers interesting prospects) and demonstrate its practical interest through the verification of recursive algorithms and data structures.

1 Challenges

How are big-Os defined in the first place, in paper proofs? Consider a program function p that expects as argument a natural number n . Let f be the *concrete cost function* for p , that is, such that $f(n)$ describes the number of execution steps performed when running $p(n)$. Common practice is to write that p is $O(g(n))$, as a lightweight notation to mean that the cost function of p , namely f , satisfies $f \in O(g)$. Recall that “ $f \in O(g)$ ”, following the standard definition, stands for $\exists c, \exists n_0, \forall n \geq n_0, f(n) \leq c \times g(n)$.

Formalizing this definition in a proof assistant may appear to be only a matter of writing down the right definitions and lemmas. This is partly true, but it also appears that the big-O notation is often used in a quite informal way: formalizing it presents in fact multiple challenges. In this section, we describe these challenges, which our formal development will need to tackle.

1.1 Challenge 1: binding variables

Informal big-O bounds are often written without explicitly binding the variable(s): we tend to write “ p is $O(n^2)$ ”. In reality, the $O()$ relation is defined on functions, not expressions with unbound variables. Therefore, a first step is writing “ p is $O(\lambda n. n^2)$ ” instead.

1.2 Challenge 2: existential quantifications

A second problem is that the big-O notation hides a quantification on the concrete cost function. When we write “ p is $O(\lambda n. n^2)$ ”, we actually mean: “there exists a concrete cost function f such that $f \in O(\lambda n. n^2)$ and the execution of $p(n)$ takes exactly $f(n)$ steps”. Even though it might be hidden behind definitions and notation, the quantification of f must somehow appear in the formal definition, outside of the Hoare triple describing the semantics of p .

Actually, if such syntactic sugar comes handy, it tends to make some wrong paper proofs harder to detect syntactically. To illustrate this point, consider the program below and an obviously-incorrect asymptotic complexity claim for it.

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

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

Proof. (flawed, but not so obviously).

By induction on n :

- when $n \leq 0$, the call to `loop` terminates in $O(1)$, there fore the cost is $O(1)$;
- when $n \geq 0$, the cost of `loop(n)` is the cost of `loop(n-1)` plus $O(1)$. By induction hypothesis, the cost of `loop(n-1)` is $O(1)$. Since $O(1) + O(1) = O(1)$, we conclude that the total cost is $O(1)$. \square

The syntax of big-Os does not make obvious that the proof is wrong: here, it is because of an invalid quantifier permutation, between the universal quantification on n , and the hidden existential quantification on the cost function, which must be instantiated *before* entering the induction. The circular definition attempted here boils down to an incorrect usage of reasoning rules about quantifiers, which would be rejected by a formal proof system. Cormen et al. [6] also mention this kind of mistakes when reasoning inductively with big-Os.

So, the first challenge is to clarify the location of the quantifiers associated with the use of big-Os in specifications, while nevertheless retaining as much as possible a lightweight presentation, in particular avoiding an explicit quantification on the concrete cost function.

1.3 Challenge 3: monotonic cost functions

Paper proofs assume extensively that cost functions are non-decreasing, without mentioning it explicitly. For example, let $p(n)$ be an OCaml program which calls some auxiliary function aux , of exact cost f_{aux} . A common pattern is calling aux on smaller data, but wanting to express its cost depending on the main parameters: e.g. `let p n = ... ; aux k ; ...`, where $k \leq n$. We would like to be able to promptly conclude that, as $k \leq n$ and by monotonicity of f_{aux} , the cost of $aux\ k$ is bounded by $f_{aux}(n)$, which is more useful for p 's specification as n is a parameter of p , while k is an internal variable.

Sometimes however, exact cost functions are not non-decreasing. As a very simple example, consider `loop'` defined as follows: `let loop' n = if n = 2 then loop 10 else loop n`. Because of a special case for the input $n = 2$, the cost function is not non-decreasing anymore, even if it presents the same asymptotic behavior. Similar perturbations can be added to make the cost function non-monotonic even asymptotically, without changing its asymptotic behavior.

As a consequence, we need to modify the interpretation of “ p is $O(g)$ ”, into “there exists a *non-decreasing* function f such that $f \in O(\lambda n.n^2)$ and the execution of $p(n)$ takes *no more than* $f(n)$ steps”.

To conclude, it appears that we do not want to existentially quantify on the concrete cost function, but instead on an upper-bound of this cost function, that presents the same asymptotic behavior, and is non-decreasing. This way, we can assume in the interpretation of big-Os that all the cost functions are non-decreasing, which simplifies the reasoning and matches the paper proofs.

1.4 Challenge 4: additive constants

In paper proofs, the following lemma is often implicitly used: if f is $O(g)$, then $f + c$, where c is a constant, is also $O(g)$. For example, since $3n^2 + 2n + 5$ is a $O(n^2)$ then $3n^2 + 2n + 5$ plus some $O(1)$ is also a $O(n^2)$.

Yet, such a lemma is false, in the general case. More precisely, it does not hold for $g = 0$. This seems like a fake problem: in practice, we do not use $O(0)$ bounds. Therefore, we somehow need to formalize the fact that when we write $f \in O(g)$ in the context of program verification, we wish to capture the assumption that g is nonnegative (i.e. $\forall n, g(n) > 0$).

1.5 Challenge 5: “going to infinity” with multiple parameters

A final subtlety of big-Os is that they implicitly requires some notion of “going towards infinity”. This is straightforward for cost functions with one parameter (with domain \mathbb{N} or \mathbb{Z}), and often inlined in the textbook definition (e.g. from Cormen et al. [6]):

$$O(g(n)) = \{f(n) \mid \exists c \geq 0, \exists n_0, \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$

However, we may want to generalize a bit, and handle cost functions with multiple parameters (e.g., “ $f(m, n)$ is a $O(g(m, n))$ ”), seen as functions with domain \mathbb{N}^2 or \mathbb{Z}^2 . A typical example is the complexity of graph algorithms, that often depends on both the number of vertex and edges. In this case, which notion of “going to infinity” should be used is not obvious. Actually, there is no definitive answer: all are not equivalent, and choosing one will depend, between other things, of later use of the specification. To illustrate this claim, consider the following program, which fills a rectangle of height n and width m .

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

The exact cost function for `fill_rect` is $f(n, m) = m \times (1 + n) + 1 = m \times n + m + 1$. In practice, we quickly deduce that `fill_rect` runs in $O(m \times n)$. Taking $g(n, m) = m \times n$, it is indeed true that $f = O(g)$ if we require both n and m to go to infinity.

Sometimes though, because e.g. the bound we establish is for a sublemma, we will want to fix n afterwards, and still have $f(n, m)$ be a $O(g(n, m))$. This is false in our example! For $n = 0$, in one hand, we have an exact cost equal to $0 \times m - m + 1 = m + 1$, i.e. $O(m)$; in the other hand, the asymptotic cost given by g , $O(g(0, m))$ is equal to $O(0 \times m) = O(0)$, i.e. 0. This is clearly wrong, as a $O(m)$ cost is not zero or even a constant. To be able to fix one parameter, we need another notion of “going to infinity”, that is not equivalent to the previous one. In this setting, a valid bound for f is e.g. $g'(m, n) = m \times n + m$.

This motivates the need for a formalized notion of “going to infinity”. Such a notion exists: mathematical *filters*—but we still need to understand how to adapt it to formalize the big-Os with multiple variables we encounter. The challenge here is to allow for the user to specify which filter is associated with each use of the big-O notation, yet keeping a lightweight notation for the typical case where a canonical filter is used.

2 Background: the CFML tool

Our Coq formalization is based on the work of Charguéraud and Pottier [5]: it extends the CFML tool they used for their proof. As a consequence, in the following sections, I present (1) the base CFML tool [4,3], useful to prove functional correctness of programs; (2) “CFML+credits”, the extension of CFML used for the proof of Union-Find [5] which allows to prove specifications about algorithmic complexity, using a mechanism of “time credits”.

2.1 CFML: proving functional correctness of OCaml programs

Charguéraud and Pottier’s machine-checked proof of Union-Find relies on the tool CFML [4,3], which is based on higher-order Separation Logic [13] and *characteristic formulae* [2]. The characteristic formula of an OCaml term t is a logic formula $\llbracket t \rrbracket$, which describes the semantics of t . For any precondition H and postcondition Q , if the logical proposition $\llbracket t \rrbracket H Q$ can be proved, then the Separation Logic triple $\{H\} t \{Q\}$ holds. The characteristic formula $\llbracket t \rrbracket$ can be generated by CFML given the term t . It can then be used, in Coq, to prove formally a specification for t .

Heap predicates Separation Logic heap predicates have type $\text{Heap} \rightarrow \text{Prop}$, and describe a part of the heap. We define the fundamental heap predicates as in Charguéraud’s paper [2], where h denotes a heap, H a heap predicate, and P a Coq logical proposition. A heap is a finite map from memory locations to values, $h_1 \perp h_2$ asserting that h_1 and h_2 have disjoint domains, and $h_1 \uplus h_2$ denoting their disjoint union.

$$\begin{aligned} [] &\equiv \lambda h. h = \emptyset \\ [P] &\equiv \lambda h. h = \emptyset \wedge P \\ H_1 \star H_2 &\equiv \lambda h. \exists h_1 h_2. h_1 \perp h_2 \wedge h = h_1 \uplus h_2 \wedge H_1 h_1 \wedge H_2 h_2 \\ \exists x. H &\equiv \lambda h. \exists x. H h \end{aligned}$$

In standard Separation Logic, another heap predicate is added, of the form $l \hookrightarrow v$, which describes a heap with a single memory cell at location l containing the value v .

$$l \hookrightarrow v \equiv \lambda h. h = (l \mapsto v)$$

Specifications In CFML, a specification for an OCaml term t is of the form $\llbracket t \rrbracket H Q$. If $\llbracket t \rrbracket H Q$ can be proved (in Coq), it implies that starting in a state that satisfies the heap predicate H , t reduces in a finite time to a value v , such that the final state satisfies the heap predicate $Q v$. Which implies that the Separation Logic triple $\{H\} t \{Q\}$ holds: H is the precondition, describing the state of the memory before running t , and Q the postcondition, describing both the value produced by t and the state of the memory afterwards.

More details on the generation of the characteristic formula $\llbracket t \rrbracket$ and the correspondence between OCaml values and OCaml terms can be found in the paper describing CFML [3]. One thing that can be noted is that the $\llbracket t \rrbracket$ characteristic formula, generated by the CFML tool, has the same shape and structure as the OCaml program. This will be useful to read the goals produced by the CFML tactics.

CFML tactics CFML’s Coq library provides a number of tactics that allow the user to walk through the program, by progressively refining the “characteristic formula” part of the goal, in order to prove a specification. This is essentially what an automated Verification-Condition generator does, but in our case we are able to use arbitrary complex lemmas between each step, or e.g. reason by induction.

As a consequence, during a proof of a $\llbracket t \rrbracket H Q$ specification, our main goal (excluding auxiliary side subgoals) will be of the form $f H' Q'$, with f being a subterm of the $\llbracket t \rrbracket$ characteristic formula, smaller as the proof progresses.

CFML tactics match the standard reasoning rules used to derive Separation Logic Hoare triples. Therefore, making the proof progress is often only a matter of applying the tactic corresponding to the head constructor of the current formula f , then proving the side subgoals that ensue.

To sum up, CFML provides a characteristic formula generator, and a Coq library that allows to write Separation Logic specification, and prove them using standard reasoning rules on programs thanks to Coq tactics. Let us illustrate this by proving functional correctness for a simple program: the `incr` function, that increments a reference.

```

let incr r =
  r := !r + 1

```

```

Parameter incr_cf :
  tag tag_top_fun Label_default
  Body incr r =>
    (LetApp _x4 := ml_get r in
     App ml_set r (_x4 + 1);)

```

We show the OCaml code for `incr`, alongside the characteristic formula that CFML produces from it. In practice, the formula is generated in a file that the user doesn’t need to read, but only import at the beginning of his proof.

A specification for `incr` is written as follows, meaning that as a precondition, `r` must be point to some integer `i`; after running `incr`, `r` must point to `i+1`:

```

Lemma incr_spec :
  Spec incr r |R>> ∀(i: int),
  R (r ≈ i) (# r ≈ (i+1)).

```

“`#H`” is syntactic sugar for “`fun (_,unit) => H`”: `incr` returns `()`. The `Spec` notation allows to abstract from the exact characteristic formula provided for `incr`. The `|R>>` piece of syntax is actually a binder, `R` being the bound variable, that will be instantiated with the characteristic formula. Once we feed it with the one we obtained, the goal becomes of the form $\llbracket \text{incr} \rrbracket (r \approx i) (\#r \approx (i+1))$. `Spec` also acts as a binder for the arguments of the specified function: “`Spec f x |R>>R H Q`” is equivalent to “ $\forall x, \text{App } f \ x \ H \ Q$ ”, which corresponds to “ $\forall x, \{H\} \text{App } f \ x \ \{Q\}$ ” in term of a Hoare triple.

The `xcf` tactic automatically introduces the characteristic formula.

```

Proof.
  xcf. intros.

```

The resulting goal is as follows:

```

r : loc
i : int
=====
(LetApp _x4 := ml_get r in
  App ml_set r (_x4 + 1);) (r ~> i * []) (# r ~> (i + 1))

```

We make the proof progress by applying tactics that match the head constructor of the characteristic formula part: here, it’s a `LetApp`; we use the `xapps` tactic. If the goal started with a `If`, we would have used `xif`, etc. We need another `xapp` after that, for the `App` constructor. Finally, what remains to be proven is a “heap implication”, stating that the final state matches the post-condition.

```

=====
#r ~> (i + 1) * [] ▶ #r ~> (i + 1) * ∃H', H'

```

The `hsimpl` tactic is able to prove it automatically, by instantiating an existential variable. Figure 1 holds the complete proof of `incr`’s specification.

2.2 CFML with credits: establishing concrete cost bounds

In order to assess the asymptotic time complexity of OCaml programs, Charguéraud and Pottier extend CFML, introducing *time credits*. A time credit is a resource that represent the right to perform one step of computation. Time credits are heap resources: they can be required as a part of the pre or postcondition, and interestingly, they can be stored for later consumption, thus allowing amortized complexity analyses.

Characteristic formulae are now instrumented to consume one time credit at each function call. We rely on (and admit) the following property of the OCaml compiler: if one ignores the cost of garbage collection, counting the number of function calls and for/while loop iterations performed by the source program is an accurate measure, up to a constant factor, of the number of machine instructions executed by the compiled program.

Therefore, the (amortized) time complexity for a function consists, up to a constant factor, in the number of credits required in its precondition.

Heap predicates Definitions of heap predicates need to be adapted a bit: a “heap” is now a couple of a map (from locations to values) and a integer: the number of available credits. We also add a new fundamental heap predicate $\$n$, describing a heap with exactly n credits.

$$\begin{aligned}
\$n &\equiv \lambda(m, c). m = \emptyset \wedge c = n \\
l \hookrightarrow v &\equiv \lambda(m, c). m = (l \mapsto v) \wedge c = 0 \\
(m_1, c_1) \perp (m_2, c_2) &\equiv m_1 \perp m_2 \\
(m_1, c_1) \uplus (m_2, c_2) &\equiv (m_1 \uplus m_2, c_1 + c_2) \\
\emptyset:\text{Heap} &\equiv (\emptyset:\text{Store}, 0)
\end{aligned}$$

The definitions for $[P]$, $H_1 \star H_2$ and $\exists x. H$, are unchanged.

Tactics A new `xpay` tactic is added, which “pays” for a time credit when required by a function call: more precisely, it consumes and removes from the current precondition a time credits, in order to justify a function call, or entering a loop.

With this extension, the specification and proof for `incr` becomes:

```

Lemma incr_spec :
  Spec incr r |R>> ∀(i: int),
    R ($ 1 * r ~> i) (# r ~> (i+1)).
Proof.
  xcf. intros.
  xpay.
  xapps. xapp. hsimpl.
Qed.

```

We now have to give one credit (\$1) in the precondition of `incr`, as it performs one step of computation. The credit does not appear in the postcondition: it is consumed by the function. In the proof, we use the `xpay` tactic to justify that we are indeed able to pay for the computation step.

3 CFML with credits and big-Os: asymptotic complexty bounds

Our development extends “CFML + time credits”, adding asymptotic reasoning and big-Os: it could be summarized as “CFML + time credits + big-Os”. In this section I present the new definitions and notions introduced in our extension, providing the following features:

- Given an explicit cost function (a “number of credits”), allow to prove an asymptotic big-O bound for it. The cost function may be defined after other functions asymptotically bounded by big-Os;
- Allow various manipulations on big-O bounds: composition, parameter transformation, etc.

3.1 Filters and textbook big-O definition

As motivated earlier, a formal notion generalizing “going to infinity” in a set that can be \mathbb{Z} , \mathbb{Z}^2 , ... is needed. We reuse the notion of *filter* from the literature. A similar notion is used in the real analysis Coq library Coquelicot [1], as a generic tool to unify the various notions of convergence in \mathbb{R} . In our case, we are only interested in convergence to infinity, in sets of the form \mathbb{Z}^k .

Informally, a filter for a given set describes a way to tend to infinity in this set. On \mathbb{Z} , the one obvious filter that corresponds to the textbook definition of “ O ” will work in any situation. However, as illustrated above, on \mathbb{Z}^2 (i.e. for functions with two parameters), we will be able to define various (not equivalent) filters, and clearly state which one should be used when writing a $O()$.

On a set A , a filter has type $(A \rightarrow \text{Prop}) \rightarrow \text{Prop}$, which can be interpreted either as a set of subsets of A (the sets of neighborhoods of infinity), or as a set of predicates on A (the set of properties that hold when going to infinity). The `Filter` class bundles additional properties that must be satisfied, like e.g. stability by intersection. This definition allows us to write `ultimately P`, when `ultimately` is of type `filter A`, meaning that P is true at some point “when going towards infinity”.

```

Definition filter A := (A → Prop) → Prop.
Class Filter {A : Type} (ultimately : filter A) := {
  (* A filter must be nonempty. *)
  filter_nonempty: ∃P, ultimately P;

  (* A filter does not have the empty set as a member. *)
  filter_member_nonempty: ∀P, ultimately P → ∃a, P a;

  (* A filter is closed by inclusion and by intersection. *)
  filter_closed_under_intersection:
    ∀P1 P2 P : set A,
    ultimately P1 → ultimately P2 → (∀ a, P1 a → P2 a → P a) →
    ultimately P
}.

```

The textbook definition of big-O can now be generalized for any functions f, g of type $A \rightarrow Z$, given a filter `ultimately` of type `filter A`. We impose Z as the codomain of f and g for practical reasons, even if we could be a bit more generic. Also for practical reasons, `norm` is not actually a norm, but equals to zero on negative values (it is defined as the Coq function `Z.to_nat`).

```

Definition dominated {A} ultimately {@Filter A ultimately} (f g : A → Z) :=
  ∃c, ultimately (fun x ⇒ norm (f x) ≤ c * norm (g x)).

```

Intuitively, “dominated ultimately f g ” means “ f is a $O(g)$, for the filter *ultimately*”. Surprisingly enough, this single notion is not sufficient to have convenient and composable proofs, as soon as the exact cost functions are abstracted away (we want to have explicit bounds, but abstract cost function, for modularity reasons). As pointed by challenges 3 and 4 (sections 1.3, 1.4), additional aspects must be taken in account in our $O()$ definition: monotonicity of the cost function, and special handling of $O(0)$ bounds.

3.2 Compatibility between filter and preorder, monotonicity

To address challenge 3 (section 1.3, we require our cost functions to be monotonic, with respect to some preorder (we do not need an order). For things to work well, some compatibility property is needed, between the preorder le (of type $A \rightarrow A \rightarrow \text{Prop}$) and the filter F (of type $\text{filter } A$).

The `FilterOrder` Coq class describes the compatibility property:

```
Class FilterOrder {A: Type}
  (F: filter A) (le: binary A)
  {Filter: Filter F} {O: Preorder le} :=
{
  filter_order_compatibility :  $\forall x: A, F (\text{fun } y \Rightarrow le\ x\ y)$ 
}.

```

Intuitively, le is compatible with F if increasing respectively to le goes towards infinity. Additionally, we can now define in a generic way a filter from a given preorder, such that they are compatible. This is not possible in the general case, however a sufficient condition is that le admits an upper bound for each pair of elements. Given a preorder le that satisfies this property, its “canonical filter” is defined as:

```
Definition canonical_filter : filter A :=
  fun (P : A  $\rightarrow$  Prop)  $\Rightarrow$   $\exists x0, \forall x: A, le\ x0\ x \rightarrow P\ x$ .

```

We are able to define an instance of the `FilterOrder` class for `canonical_filter`.

```
Definition fo_canonical_of_order : FilterOrder canonical_filter le.

```

The standard filter for \mathbb{N} and \mathbb{Z} is actually defined thanks to `canonical_filter`, and corresponds to the textbook notion of divergence to infinity.

Monotonicity predicates are defined in a standard way: `monotonic leA leB f` assess that f (of type $A \rightarrow B$) is monotonic regarding to relations leA and leB (respectively of type $A \rightarrow A \rightarrow \text{Prop}$ and $B \rightarrow B \rightarrow \text{Prop}$). `monotonic_after leA leB f a0` assess that f is monotonic for values greater than $a0$. Combined with a filter predicate, `monotonic_after` provides a notion of asymptotic monotonicity.

```
Definition monotonic A B (leA : A  $\rightarrow$  A  $\rightarrow$  Prop) (leB : B  $\rightarrow$  B  $\rightarrow$  Prop) (f : A  $\rightarrow$  B) :=
   $\forall a1\ a2,$ 
  leA a1 a2  $\rightarrow$  leB (f a1) (f a2).

```

```
Definition monotonic_after A B (leA: A  $\rightarrow$  A  $\rightarrow$  Prop) (leB: B  $\rightarrow$  B  $\rightarrow$  Prop)
  (f: A  $\rightarrow$  B) (a0: A) :=
   $\forall a1\ a2,$ 
  leA a0 a1  $\rightarrow$  leA a1 a2  $\rightarrow$  leB (f a1) (f a2).

```

3.3 Custom big-O definition

To address the second issue, detailed in challenge 4 (section 1.4), we define an variant of `dominated`, dubbed `idominated`, which allows more convenience lemmas that are used in practice. `idominated _ _ f g` unifies the cases where g is equal to 0 where the ones where it is a $O(1)$: in these cases, we are only interested in knowing that both f and g are $O(1)$. In the other cases—in which we are most interested in practice—`idominated` corresponds to `dominated`. We also require g to be asymptotically monotonic.

Definition `idominated`

```

{A} ultimately leA {Filter: Filter ultimately} {O: Preorder leA}
{FO: FilterOrder ultimately leA}
(f g : A → Z) :=
ultimately (monotonic_after leA le g) ∧
((bounded _ f ∧ bounded _ g) ∨ dominated _ f g).

```

The `bounded` predicate is equivalent as “being a $O(1)$ ”. Basically, `idominated` handles more “pathologic cases” while implying `dominated` on the interesting ones. As a consequence, the following lemma is now true for any constant `c`: `idominated _ _ f g ⇒ idominated _ _ (λn ⇒ c + f n)g`. Lemmas about `idominated` are more simple to use and get rid of multiple side conditions; as a drawback proving them is more technical (e.g. we proved for internal use the lemma stating that a monotonic unbounded function tends to infinity, in our setting of filters compatible with preorders).

Remark Another definition of `idominated` that comes in mind is to require `g` to be ultimately greater than zero. This seems a reasonable request, however it doesn’t compose very well: `idominated _ _ f g` does not imply `idominated _ _ (λn ⇒ Z.log2 (f n))(λn ⇒ Z.log2 (g n))`, because `Z.log2 (g n)` can be equal to zero for all `n` if `g n` is always equal to 1.

3.4 Custom specification predicate

A new “specification with big-Os” predicates wraps it all, quantifying existentially on the exact cost function to abstract it.

Definition `Spec0` (`ultimately: filter A`) `leA` (`g: A → Z`)

```

(spec: (A → Z) → Prop) :=
∃(f: A → Z),
(∀ x, 0 ≤ f x) ∧
monotonic _ _ f ∧
idominated _ _ f g ∧
spec f.

```

`Spec0` is a convenient wrapper to state a specification using a big-O: given an asymptotic bound `g`, it quantifies existentially on an explicit cost function `f`, and bundle the necessary facts about `f`:

- `f` needs to be positive, as it represent a number of time credits;
- As justified in section 3.1, we require `f` to be monotonic;
- `f` should be a big-O of `g`, using our custom `idominated` definition;
- Finally, the `spec f` specification must hold. Usually, `spec f` is of the form `Spec .. |R>> ..`, i.e. a standard CFML specification, that can use `f` in its precondition to require time credits in a modular way.

`Spec0` beginning by `∃f, ...` means that, to prove a `Spec0` goal, the user must more or less give a precise cost function right away. As it is now, in order to “guess” the exact expression of the cost function, the usual method is to look at the OCaml program, introduce the abstract cost functions for auxiliary functions, and remember that each function call consumes a credit.

An additional lemma, `Spec0_of_Spec0_after`, allows to prove a `Spec0` specification and side sub-goals only for values greater than a given bound. The lemma constructs a new cost function, equal to the user-provided one on this subdomain, and equal to zero elsewhere.

3.5 Additional lemmas, instances and tactics

These definitions come with various lemmas, instances and automated tactics to assist the user in her proofs. Some of them will be mentioned and described in the case studies, section 4.

4 Case studies

We illustrate the usage of our extension of CFML through various examples, presenting different challenges, of increasing complexity. Basic examples give a first grasp of the most useful new tactics, the proof of dynamic arrays involves amortization analysis over a mutable structure, and the proof of binary random access list will demonstrate composition of big-Os, product filters, and parameter transformations.

4.1 A first simple example: `incr`

We start by proving a very simple specification: asymptotic complexity for the `incr` function.

```
Lemma incr_spec :
  Spec01 (fun F =>
    Spec incr r |R>> ∀(i: int),
      R ($ F tt * r ~ i) (# r ~ (i+1))).
```

`Spec01` is `Spec0`, specialized with a filter on domain `unit`, useful for $O(1)$ cost functions that do not depend on any parameter. Recall that the `F` bound in the specification is a name for the abstract cost function of `incr` (which is here a $O(1)$), abstracting from the precise cost.

Just like we used the `xcf` tactic for goals starting with `Spec`, we use here the `xcf0` tactic. It takes an argument: the expression of the cost function. Note that the domain of the cost function is indeed `unit`, and that we provide a concrete expression for the cost function, before beginning the proof: credit count is explicit in the proof, and only abstracted afterwards.

```
Proof.
  xcf0 (fun (_:unit) => 1).
```

`xcf0`, applied on a `Spec0` goal, instantiates the existential quantification, and tries to prove the side subgoals (positivity, monotonicity, domination) using automated tactics. In this very simple example, all automated tactics succeed and the goal becomes the same as before:

```
=====
Spec incr r |R>> ∀(i: int),
  R ($ 1 * r ~ i) (# r ~ (i+1)).
```

From this point, the proof is the same as before.

```
Lemma incr_spec :
  Spec01 (fun F =>
    Spec incr r |R>> ∀(i: int),
      R ($ F tt * r ~ i) (# r ~ (i+1))).
```

```
Proof.
  xcf0 (fun (_:unit) => 1).
  xcf.intros.xpay.
  xapps.xapp.hsimpl.
Qed.
```

4.2 A basic example: `mktree`

A slightly more interesting example is the proof of the `mktree` recursive function, which builds a complete tree of depth n in $O(2^n)$ time.

```
type 'a tree =
  | Node of 'a tree * 'a tree
  | Leaf of 'a

let rec mktree (depth: int) (x: 'a): 'a tree =
  if depth <= 0 then Leaf x
  else Node (mktree (depth - 1) x,
             mktree (depth - 1) x)
```

A specification for proving `mktree` complexity uses `Spec0`, unsurprisingly, requiring credits from a cost function dominated by $(\lambda n \Rightarrow 2^n)$.

Lemma `mktree_spec` :

```
Spec0 (fun n => 2 ^ n) (fun F =>
  Spec mktree (depth: int) (x: a) |R>>
  0 <= depth ->
  R ($ F depth) (fun (t: tree a) => [])).
```

Note that this specification is not as expressive as it could be: it does not require the returned tree to be complete, as we are only interested in the complexity analysis.

The domain of our cost function `F` is `Z`. However, we are only interested in non negative values, as is `depth`. We apply `Spec0_of_Spec0_after` to allow us to prove our specification only for these values.

Proof.

```
applies @Spec0_of_Spec0_after 0.
xcf0 (fun n => 2 ^ (n + 1) - 1).
```

The cost function is then introduced using `xcf0`: building a tree of depth 2^n takes exactly $2^{n+1} - 1$ steps. Aside from the specification, `xcf0` produces two side subgoals that have not been proved automatically:

- The cost function is non-negative.

```
=====
  vx : int, 0 <= x -> 0 <= 2 ^ (x + 1)%I - 1
```

Easily proved thanks to the `pow2_pos` auxiliary lemma.

```
- intros. forwards~: pow2_pos (x+1).
```

- The cost function is a $O(2^n)$.

```
=====
  idominated_towards_infinity_Z le (fun n : int => 2 ^ (n + 1)%I - 1)
  (fun n : int => 2 ^ n)
```

We can use a bit of automation here: the `idominated_Z_auto` tactic is able to solve automatically simple goals of the form `idominated _ _ f g`, when `f` and `g` are composed of `+`, `*`, `Z.log` and `Z.pow`. Even if it does not succeed at solving the goal, it is often able to make some progress, for example by removing unnecessary constants.

In our case, applying `idominated_Z_auto` leads to two subgoals:

- `towards_infinity_Z (fun x : int => 0 <= x)`: `towards_infinity_Z` is the standard filter on `Z`. This subgoal asks for a rank from which values are greater than zero: as one can check by unfolding `towards_infinity_Z`, a simple `exists~ 0`. proves the goal.
- `towards_infinity_Z (monotonic_after le le (fun n : int => 2 ^ n))`: just as `idominated_Z_auto` is an automated tactic to solve or simplify `idominated` goals, the `monotonic_Z_auto` tactic tries to solve or simplify monotonicity goals.

In this case, calling the tactic proves the subgoal automatically.

```
- idominated_Z_auto~. exists~ 0. monotonic_Z_auto.
```

The last quirk is due to `Spec0_of_Spec0_after`. The goal uses an abstract cost function `F'` instead of `F`, and we are given a proof that $\forall x, 0 \leq x \rightarrow F' x = F x$. In practice, we just add this proof to our context and rewrite it when needed.

```
intros F' eqF'.
```

The rest of the proof uses standard CFML tactics, plus:

- `xpay` when presented to a `Pay; ...` goal;
- `csimpl` when presented to a heap implication involving credits: turns the goal into an (in)equality between the quantities of credits.

The complete proof can be seen in [Figure 2](#).

4.3 Dynamic arrays

We formalized an OCaml implementation of dynamic arrays (that grow and shrink according to number of items it stores, as in Cormen et al. [6]) using CFML with time credits. The proof illustrates how an amortized complexity analysis can be performed in this setting.

Definition A dynamic array is an array-like data structure that supports constant time random access (`get` and `set`), plus amortized constant time `push` and `pop` operations, which respectively add and remove an element at the end of the array. The memory shape of the structure is a standard array, of size equal or greater than the number of elements stored in it: there may be pre-allocated but unused memory cells at the end of the array. We name *capacity* the current size of the whole in-memory array: it represents the maximum number of elements that can be stored before having to allocate a new, bigger array.

`push` and `pop` do not always perform in constant time: from time to time, a new array (bigger or smaller) needs to be allocated, and the structure contents copied from the old array into the new one. However, it is possible to amortize the cost of these reallocations, by carefully choosing the size of the allocated arrays, and deciding when to reallocate. This is a standard analysis, detailed on paper in Cormen et al. [6], which we formalize in Coq.

The key technique is to store time credits in the heap, for later consumption. The heap predicate `hinv` characterizing a dynamic array is defined as describing a standard array, for some capacity, plus some logical invariants, *plus a certain amount of time credits*.

```

Definition hinv A {IA:Inhab A} (L:list A) size data (default:A) D n b_min t :=
  t ~> RecDynArray default size data
  * data ~> Array D
  * [inv L size D n b_min]
  * $(potential size n).

```

```

Definition DynArray (A:Type) {IA:Inhab A} (L:list A) (t:dyn_array A) :=
  ∃size data default D n, t ~> hinv L size data default D n true.

```

The exact amount of credits is defined by a `potential` function, which depends on the capacity of the array and the number of elements stored in it.

```

Definition potential size n := Z.abs (size - 2^(n + 1)) * op_cst.

```

A specification for `push` is then:

```

Lemma push_spec : ∀A,
  Spec01 (fun F =>
    Spec push (t:dyn_array A) (x:A) |R>>
      ∀{IA:Inhab A} (L:list A),
      R (t ~> DynArray L * $F tt) (# t ~> DynArray (L&x))).

```

Apart from the credits stored in the structure, `push` only requires a constant number of credits to run: if more time credits are needed to resize the array, they will be provided by the `t ~> DynArray L` part of the heap.

Specifications using big-Os are not particularly challenging in this example, though. As all functions of the API run in constant time or amortized constant time, only $O(1)$ cost functions are involved.

4.4 Binary Random Access Lists

Our most interesting and challenging case study is the formalization of binary random access lists, a purely functional data structure, well described by Okasaki [12]. Its proof is particularly interesting

as it involves some subtle reasoning with big-Os, that proved to be a challenge when designing the library.

In a first part we present in more details the “binary random access list” data structure, recalling the invariants involved and justifying informally the correctness of the implementation. In a second part, we describe key difficulties of the proof, and how they are handled using our CFML library extension.

Binary Random Access Lists: the OCaml implementation A “binary random access list” is a functional data structure, which features usual list operations (`cons` and `uncons` for adding and removing an element at the head of the list), but also random access `lookup` and `update`. That is, one can add or remove an element at the head of the list, but also modify or query the i^{th} element of the structure.

These four operations perform in worst-case $O(\log(n))$ steps, where n is the number of elements stored in the structure. Let us see step by step how it is implemented in OCaml. Figure ?? contains the source code for the complete implementation.

Type definitions, implicit invariants A binary random access list is a list of binary trees: we first define a `tree` OCaml type.

```
type 'a tree = Leaf of 'a | Node of int * 'a tree * 'a tree
```

Notice that only the leaves store elements. Nodes contain an integer corresponding to the number of elements stored (in the leaves) in the tree, which makes a `size` function trivial to implement:

```
let size = function
  | Leaf x -> 1
  | Node (w, _, _) -> w
```

Now, a binary random access list is a list of either a tree, either nothing. We consequently define an `tree option` type, here dubbed `digit`.

```
type 'a digit = Zero | One of 'a tree
```

The name `digit` comes of the similarity between a binary random access list and a list of bits, representing an integer—adding an element at the head being similar to incrementing the integer, etc. We’ll see more of that later.

Finally, we define the type for the whole structure: the binary random access list.

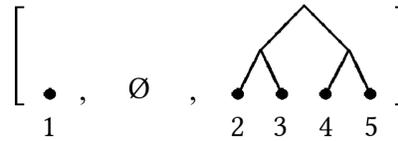
```
type 'a rlist = 'a digit list
```

A valid binary random access list should satisfy some additional invariants:

- Trees are complete trees – a tree of depth d always has 2^d leaves;
- Any `rlist` contains trees of increasing depth starting at some depth p : if the i^{th} cell (indexing from 0) contains a tree (is of the form `One t`), then this tree has depth $p + i$;
- A complete binary random access list is a `rlist` with starting depth p equal to 0: its first tree, if present, is only a leaf.

To sum up, a binary random access list is a list, which stores in its i^{th} cell either nothing, or a complete binary tree of depth i .

As an example, a binary random access list storing the sequence 1, 2, 3, 4, 5 can be represented as:



Binary random access lists look like integers in binary As mentioned before, a binary random access list is in fact quite similar to an integer represented in binary, i.e. as a list of bits.

Actually, if one erases the trees from the implementation (`type 'a digit = Zero | One of 'a tree` becomes `type digit = Zero | One`), one obtains an implementation of integers, represented as a list of bits (least significant bit at the head of the list); the `cons` operation being incrementing, `uncons` decrementing.

Incrementing an integer consists in looking at the least significant bit; if it's a zero, turning it into a one; if it's a one, turning it into zero, and recursively continuing to increment, starting with the next bit.

“Consing” to a binary random access list is similar, except that we have to handle a bit more information: the elements of the structure, stored in the trees.

Instead of adding 1, we add a tree `t` (more precisely, a digit `One t`): if the first element of the list is `Zero`, we turn it into `One t`. If it's a `One t'`, we turn it into `Zero`, and recursively continue, but with a new tree: `Node (size t + size t', t, t')`. This corresponds to the `link` operation, which combines two trees of depth d into one of depth $d + 1$.

```
let link t1 t2 = Node (size t1 + size t2, t1, t2)
```

The OCaml implementation follows:

```
let rec cons_tree t = function
  | [] -> [One t]
  | Zero :: ts -> One t :: ts
  | One t' :: ts -> Zero :: cons_tree (link t t') ts

let cons x ts =
  cons_tree (Leaf x) ts
```

The `uncons` operations follows the same idea: it is similar to decrementing, except that we also have to invert the `link` operation to obtain trees of smaller depth.

```
let rec uncons_tree = function
  | [] -> raise Empty
  | [One t] -> (t, [])
  | One t :: ts -> (t, Zero :: ts)
  | Zero :: ts ->
    match uncons_tree ts with
    | Node (_, t1, t2), ts' -> (t1, One t2 :: ts')
    | _ -> assert false

let head ts =
  match uncons_tree ts with
  | (Leaf x, _) -> x
  | _ -> assert false

let tail ts =
  let (_, ts') = uncons_tree ts in
  ts'
```

Unconsing a `rlist` of starting depth p always returns a tree of depth p (or fails if the list is empty). In particular, as the binary access list manipulated by the user always starts with depth 0, we can assume in the implementation of ‘head’ that the unconsed tree is a leaf.

Random access Once we know how to build a structure following the invariants we stated, thanks to `cons` and `uncons`, it is quite easy to implement random access lookup and update.

The idea is to walk through the list; faced to a `Node (w, l, r)` tree, we know how much elements it contains: it is exactly `w`. Knowing the index i of the element we want to visit, we can compare it to `w` to know whether we should explore this tree (if $i < w$), or continue walking the list.

```
let rec lookup i = function
  | [] -> raise (Invalid_argument "lookup")
  | Zero :: ts -> lookup i ts
```

```

| One t :: ts ->
  if i < size t
  then lookup_tree i t
  else lookup (i - size t) ts

```

If we have to walk through the tree, we can also do this without performing an exhaustive exploration: by comparing the index to half the size of the tree, we can decide whether we should explore the left or right subtree.

```

let rec lookup_tree i = function
| Leaf x -> if i = 0 then x else raise (Invalid_argument "lookup")
| Node (w, t1, t2) ->
  if i < w/2
  then lookup_tree i t1
  else lookup_tree (i - w/2) t2

```

The `update` function works in a similar fashion.

Specification and complexity analysis of Binary Random Access Lists The complete Coq script for the proof can be [observed online](#).

Invariants of the structure We begin the Coq formalization by expressing explicitly the implicit invariants of the algorithm, mentioned in the previous section.

Knowing that a random access list structure satisfies these invariants is key for the complexity proof: it gives us information about its size, and the size of its subtrees. Then, to know that our structures actually satisfy these invariants, we need to prove functional correctness of the OCaml code, i.e. prove that the functions do not break the invariants of the structure.

Consequently, our Coq proof is twofold: it proves both functional correctness, and algorithmic complexity.

Predicates CFML automatically generates the Coq counterpart of the OCaml datatypes, `tree a` and `rlist a`. We start the proof by defining three predicates `btree`, `inv` and `Rlist`, that make explicit the invariants of the structure.

First, a `btree` predicate. `btree n t L` means that the `t` is a complete (binary) tree of depth `n` which contains the sequence of elements in `L`.

```

Inductive btree : int -> tree a -> list a -> Prop :=
| btree_nil : ∀x,
  btree 0 (Leaf x) (x::nil)
| btree_cons : ∀p' n t1 t2 L1 L2 L',
  btree p t1 L1 ->
  btree p t2 L2 ->
  p' = p+1 ->
  n = 2^p' ->
  L' = L1 ++ L2 ->
  btree p' (Node n t1 t2) L'.

```

Then, an `inv` predicate: the invariant for the whole structure. `inv p ts L` means that `ts` is a `rlist` of complete trees of increasing depth, starting with depth `p`. `L` is the sequence of elements represented by `ts`. `ts` being a well-formed binary random access list corresponds to the case where `p` is equal to 0. It is useful to consider the cases where `p` is non-zero, though: reasoning by induction on `ts` will lead to such cases.

```

Inductive inv : int -> rlist a -> list a -> Prop :=
| inv_nil : ∀p,
  p ≥ 0 ->

```

```

    inv p nil nil
  | inv_cons : ∀p (t: tree a) ts d L L' T,
    inv (p+1) ts L →
    L' ≠ nil →
    p ≥ 0 →
    (match d with
    | Zero ⇒ L = L'
    | One t ⇒ btree p t T ∧ L' = T ++ L
    end) →
    inv p (d :: ts) L'.

```

Finally, the `Rlist` predicate corresponds to the $p = 0$ case: it describes a complete well-formed binary random access list.

Definition `Rlist (s: rlist a) (L: list a) := inv 0 s L.`

Bounds Given structures verifying these invariants, we can deduce additional properties, in particular:

Lemma `length_correct : ∀t p L,`
`btree p t L → length L = 2p.`

Lemma `ts_bound_log : ∀ts p L,`
`inv p ts L → length ts ≤ Z.log2 (2 * (length L) + 1).`

These lemmas will be key for proving our log complexity bounds, and constitute in fact our only mathematical analysis for this library.

cons_tree: a first proof Let us jump directly to the proof of the (internal) `cons_tree` function.

```

let rec cons_tree (t: 'a tree) = function
| [] -> [One t]
| Zero :: ts -> One t :: ts
| One t' :: ts -> Zero :: cons_tree (link t t') ts

```

```

and link t1 t2 = Node (size t1 + size t2, t1, t2)

```

`cons_tree t ts` adds a new tree `t` to the `rlist ts`. It may recursively walk through the list, calling `link` (the process is very similar to incrementing an integer represented as a list of bits).

As `link` runs in constant time, `cons_tree` performs $O(|ts|)$ operations. Moreover, we showed earlier that $|ts| = O(\log(|L|))$ where `L` is the list of elements contained in `ts`. Therefore, `cons_tree` performs “in $O(\log(n))$ ” (we want to eventually express the complexities depending on the number of elements in the structure; here “ n ”).

Our formal proof follows this two-step informal reasoning: first we prove a $O(|ts|)$ complexity, reasoning by induction on `ts` to follow the flow of the OCaml program; then we use our `ts_bound_log` lemma to deduce a logarithmic bound depending on the number of elements stored in `ts`.

cons_tree’s auxiliary specification We therefore prove an auxiliary specification, as our first step. Let us walk through the proof.

Lemma `cons_tree_spec_aux :`
`Spec0 (fun n ⇒ n) (fun F ⇒`
`Spec cons_tree (t: tree a) (ts: rlist a) |R>>`
`∀p T L, btree p t T → inv p ts L →`
`R ($ F (length ts) (fun ts' ⇒ [inv p ts' (T++L)]))`

To prove a `Spec0` goal, one must start by providing an explicit cost function. In this case however, we do not provide one right away: as `cons_tree` calls the `link` function, its cost function depends on `link`'s one. We need to unpack `link`'s specification in order to access its (abstract) cost function. We also use `Spec0_of_Spec0_after` to restrict the domain to non negative values (as is `length ts`).

Proof.

```
destruct link_spec as (link_cost & link_cost_nonneg & ? & ?).
applies @Spec0_of_Spec0_after 0.
specialize (link_cost_nonneg tt). (* Help the automated tactics. *)
xcf0 (fun n => 1 + (1 + (link_cost tt)) * n).
```

Our cost function is still relatively simple, so the additional goals (monotonicity, domination, ...) are automatically proven by `xcf0`. The rest of the proof (proving the specification by induction) does not present new difficulties. Figure 3 shows the complete proof.

cons_tree's main specification The main specification can then use a cost function in $O(\log(|L|))$, L being the list of elements in the structure.

Lemma `cons_tree_spec` :

```
Spec0 Z.log2 (fun F =>
  Spec cons_tree (t: tree a) (ts: rlist a) |R>>
  ∀p T L, btree p t T → inv p ts L →
  R ($ F (length L)) (fun ts' => [inv p ts' (T++L)])).
```

The proof is simple: we first reuse the cost function of the previous lemma `cons_tree_spec_aux`, feeding it with a sufficient number of credits, as justified by the `ts_bound_log` lemma (“ $|ts| \leq \log(2 \times |L| + 1)$ ”).

Proof.

```
destruct cons_tree_spec_aux
  as (cons_tree_cost & cost_pos & cost_mon & cost_dom & cons_tree_spec).
xcf0 (fun n => cons_tree_cost (Z.log2 (2 * n + 1))).
```

This time, we have to prove some additional goals by hand, produced by `xcf0`.

– Monotonicity

```
– applies @monotonic_comp. monotonic_Z_auto.
```

We first apply `monotonic_comp`: our cost function is monotonic as composition of two monotonic functions. `applies` includes a bit of automation, so the fact that `cons_tree_cost` is monotonic (present in the context) is automatically used. Remains to prove that `fun n => Z.log2 (2 * n + 1)` is monotonic: `monotonic_Z_auto` solves it automatically.

– Domination

```
– applies @idominated_transitive. applies @idominated_comp cost_dom.
  monotonic_Z_auto. monotonic_Z_auto. simpl. idominated_Z_auto.
```

Our initial goal is `idominated _ _ (fun n => cons_tree_cost (Z.log2 (2 * n + 1)))Z.log2`. We cannot directly apply a composition lemma; however we know that `cons_tree_cost` is $O(n)$: we first invoke transitivity of `idominated`, then apply a composition lemma.

The remaining goals are proved automatically, either by `monotonic_Z_auto` or `idominated_Z_auto` (here, `idominated_Z_auto` proves

```
idominated _ _ (fun n => Z.log2 (2 * n + 1))Z.log2).
```

We can finally prove the specification itself:

```
– xweaken. do 4 intro. intro spec. intros. xgc; [xapply spec ]; csimpl.
  { apply cost_mon. apply ts_bound_log. }
```

The proof consists in a weakening of `cons_tree_spec_aux`, plus the following facts:

- `ts_bound_log`: $|ts| \leq \log(2 \times |L| + 1)$
- `cons_tree_cost` is monotonic: needed to apply `ts_bound_log` inequality under `cons_tree_cost`.

lookup: how to deal with multiple parameters As illustrated by challenge 5 (section 1.5), things get tricky when the cost function depends on multiple parameters. More precisely, the user has to specify which notion of “going to infinity” she’s intending, by choosing the right filter for the domain (e.g. $\mathbb{Z} \times \mathbb{Z}$ for a cost function with two parameters).

Proving a specification for the `lookup` function involves precisely this kind of difficulty.

```
let rec lookup i = function
| [] -> raise (Invalid_argument "lookup")
| Zero :: ts -> lookup i ts
| One t :: ts ->
  if i < size t
  then lookup_tree i t
  else lookup (i - size t) ts
```

We prove a `lookup i ts` specification by induction on `ts`. During the induction we have two parameters: $|ts|$, and the depth `p` of `ts`’s first tree (matching an `inv p ts L` invariant).

The respective status of these two parameter differs, though. Once the proof by induction done, we’ll want, as for `cons_tree`, express the cost function depending on $|L|$. $|ts|$ will tend to infinity with $|L|$, but `p` will be fixed to 0, as `lookup` is only supposed to be called on well-formed random access lists from the user point of view.

When proving `cons_tree`, we did not have to provide any filter: the standard filter for \mathbb{Z} was inferred. Here, we proceed as follows:

- We establish on paper a first asymptotic bound of $O(p + |ts|)$;
- We provide a filter `towards_infinity_xZ p` on (a subset type of) $\mathbb{Z} * \mathbb{Z}$, which makes its second component tend to infinity, while the first is fixed to `p` (`p` is a parameter of the filter);
- We prove an intermediate specification using this filter, for any fixed `p`. Note that unfortunately, we cannot use `Spec0` to state our specification: to get a provable and useful specification, we need to quantify over `p` “in the middle of `Spec0`”.

The result is a quite ugly intermediate specification, unfortunately; the result of unfolding `Spec0` and quantifying over `p` in the middle.

```
Lemma lookup_spec_ind :
  ∃(F: Z * Z → Z),
  (∀ m n, 0 ≤ m → 0 ≤ n → 0 ≤ F (m, n)) ∧
  (∀ (p: Z),
    0 ≤ p →
    monotonic (fixed_fst_le le p) le (fun p ⇒ F (proj1_sig p)) ∧
    idominated (FO := fo_towards_infinity_xZ p) _ _
      (fun p ⇒ F (proj1_sig p))
      (fun p ⇒ let '(m, n) := proj1_sig p in m + n)) ∧
  Spec lookup (i:int) (ts: rlist a) |R>>
  ∀p L, inv p ts L → ZInbound i L →
  R ($F (p, length ts)) (fun x ⇒ [ZNth i L x]).
```

The proof (by induction) has the same spirit as these shown before—basically applying `monotonic_*` and `idominated_*` lemmas—just more involved.

- Finally, we prove a nicer top-level specification for `lookup`:

```
Lemma lookup_spec :
  Spec0 Z.log2 (fun F ⇒
```

```

Spec lookup (i: int) (ts: rlist a) |R>>
  ∀L, Rlist ts L → ZInbound i L →
  R ($F (length L)) (fun x ⇒ [ZNth i L x]).

```

After instantiating the cost function of our intermediate `spec lookup_spec_ind` as `lookup_spec_cost`, and projecting the monotonicity and domination properties with $p = 0$, we provide the following cost function:

```

xcf0 (fun n ⇒ lookup_spec_cost (0, Z.log2 (2 * n + 1))).

```

As `fun (m,n) ⇒ lookup_spec_cost (m, n)` is a $O(m + n)$, by composition, our cost function is indeed a $O(\log(n))$. We conclude by weakening.

To sum up, the two key aspects here were: (1) choosing the right filter, adapted for later usage of the auxiliary specification, (2) managing to write a specification using this filter, which was not obvious.

Our custom filter here was `towards_infinity_xZ p` (for any `p`), a filter on the set `FixedFst p = { x : Z * Z | fst x = p }`, a subset of `Z * Z`. (symmetrically, there exists a `towards_infinity_Zx p` filter for any `p`). `towards_infinity_xZ p` let the second component of pairs of the domain go to infinity, while the first component is (by definition) fixed and equal to `p`. By quantifying universally on `p` we get what we want: we are able to fix `p` to any value afterwards, and still have the second component growing to infinity.

The fact that we could not use `Spec0` to write our auxiliary specification is a bit confusing at first. It is however the only way to express what we wanted: the inductive proof of `lookup_spec_ind` requires an induction hypothesis generic in `p`, so we cannot quantify over `p` before `CFSpec0`. We cannot quantify over different `p` in the definition of the filter either, as we want to eventually fix it: the second component must be able to go to infinity for a given, fixed `p`.

5 Related Work

There is a relatively extensive literature on (semi-)automatic inference of big-O complexity bounds. The Resource Aware ML project (RAML) takes the approach of a type system denoting resource usage (time and space resources): Hofmann and Jost [11] describe automatic inference of linear bounds for higher-order functional programs, a work extended by Hoffmann and Hofmann [10] to handle amortized polynomial bounds, though for first-order programs only. In a recent work, Danner, Licata and Ramyaa [8] present a framework where big-O bounds for higher-order functional programs can be automatically deduced, for programs defined over an inductive datatype. Asymptotic bounds are expressed depending on a programmer-specified notion of size of the given datatype. As these frameworks focus on automation, they are limited in the bounds they can infer and the programs they can analyze.

In the same line of work as CFML extended with credits [5], where a proof assistant is used to enable proving arbitrarily complex specifications, Danielsson's THUNK library [7] allows proving amortized time bounds for lazy Agda programs expressed in a cost monad, with a symmetric concept of *debts*. However, cost annotations are also explicit, and do not use big-O notation.

The Coquelicot Coq real analysis library [1] formalizes Landau's small-o notation using filters, however their scope is restricted to real analysis functions. We believe that we are the first to combine a formalization of Landau's asymptotic notation and a program verification framework, in order to prove asymptotic bounds with big-Os for arbitrarily complex programs.

6 Conclusion

This internship was very interesting, and the people of the Gallium team very nice, as always. I'm looking forward to continuing working on this topic!

References

1. Boldo, S., Lelay, C., Melquiond, G.: [Coquelicot: A User-Friendly Library of Real Analysis for Coq](#). Mathematics in Computer Science p. 22 (Jun 2014)
2. Charguéraud, A.: [Characteristic formulae for the verification of imperative programs](#). In: International Conference on Functional Programming (ICFP). pp. 418–430 (2011)
3. Charguéraud, A.: [Characteristic formulae for the verification of imperative programs](#) (2012), to appear in HOSC
4. Charguéraud, A.: [Characteristic Formulae for Mechanized Program Verification](#). Ph.D. thesis, Université Paris 7 (2010)
5. Charguéraud, A., Pottier, F.: Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In: Proceedings of the 6th Conference on Interactive Theorem Proving (ITP 2015). Lecture Notes in Computer Science, Springer (Aug 2015)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: [Introduction to Algorithms \(Third Edition\)](#). MIT Press (2009)
7. Danielsson, N.A.: [Lightweight semiformal time complexity analysis for purely functional data structures](#). In: Principles of Programming Languages (POPL) (2008)
8. Danner, N., Licata, D.R., Ramyaa, R.: [Denotational cost semantics for functional languages with inductive types](#). CoRR abs/1506.01949 (2015)
9. Drmota, M., Szpankowski, W.: [A master theorem for discrete divide and conquer recurrences](#). J. ACM 60(3), 16:1–16:49 (Jun 2013)
10. Hoffmann, J., Hofmann, M.: [Amortized resource analysis with polynomial potential](#). In: European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 6012, pp. 287–306. Springer (2010)
11. Hofmann, M., Jost, S.: [Static prediction of heap space usage for first-order functional programs](#). In: Principles of Programming Languages (POPL). pp. 185–197 (2003)
12. Okasaki, C.: [Purely Functional Data Structures](#). Cambridge University Press (1999)
13. Reynolds, J.C.: [Separation logic: A logic for shared mutable data structures](#). In: Logic in Computer Science (LICS). pp. 55–74 (2002)

7 Annex of figures

```

Lemma incr_spec :
  Spec incr r |R>>  $\forall(i: \text{int}),$ 
    R (r  $\rightsquigarrow$  i) (# r  $\rightsquigarrow$  (i+1)).
Proof.
  xcf. intros.
  xapps. xapp. hsimpl.
Qed.

```

Fig. 1. Specification and proof for incr

```

Lemma mktree_spec :
  Spec0 (fun n  $\Rightarrow$  2 ^ n) (fun F  $\Rightarrow$ 
    Spec mktree (depth: int) (x: a) |R>>
    0  $\leq$  depth  $\rightarrow$ 
    R ($ F depth) (fun (t: tree a)  $\Rightarrow$  [])).
Proof.
  applys @Spec0_of_Spec0_after 0.
  xcf0 (fun n  $\Rightarrow$  2 ^ (n + 1) - 1).
  - intros. forwards: pow2_pos (x+1).
  - idominated_Z_auto. exists0. monotonic_Z_auto.
  - intros F' eqF'.
    xinduction (fun (depth: int) (x: a)  $\Rightarrow$  Z.to_nat depth).
    xcf. intros depth x spec_ind depth_pos. rewrites eqF'.
    forwards: pow2_pos depth.
    xpay. csimpl. rew_pow 2 depth.
    xif. xret. csimpl.
    xapps; try rewrites eqF'. math_lia. csimpl; rew_pow 2 depth.
    xapps; try rewrites eqF'. math_lia. csimpl; rew_pow 2 depth.
    xret. csimpl.
Qed.

```

Fig. 2. Proof of asymptotic complexity for mktree

Proof.

```

destruct link_spec as (link_cost & link_cost_nonneg & ? & ?).
applies @Spec0_of_Spec0_after 0.
specialize (link_cost_nonneg tt). (* Help the automated tactics. *)
xcf0 (fun n => 1 + (1 + (link_cost tt)) * n).
intros F' eqF'.
xinduction (fun (t:tree a) (ts:rlist a) => LibList.length ts).
xcf. intros ? ts. introv IH Rt Rts. rewrites~ FeqF'.
inverts Rts.
- xpay. csimpl. xgo; hsimp; constructors~.
- { xpay. csimpl. simpl_nonneg~.
  xmatch.
  - xret; hsimp; constructors~; subst; splits~.
  - unpack; subst. xapps~.
    { csimpl. rew_length. math_nia. }
  intros. xapps~.
  { rewrites~ FeqF'. csimpl; rew_length; math_nia. }
  intros. xret.
  { hsimp. constructors~. rew_list~. } }

```

Qed.

Fig. 3. Specification and proof for `cons_tree`