A formal proof of Borodin-Trakhtenbrot's Gap Theorem

Andrea Asperti

Department of Computer Science and Engineering – DISI
University of Bologna
asperti@cs.unibo.it

Abstract. In this paper, we discuss the formalization of the well known Gap Theorem of Complexity Theory, asserting the existence of arbitrarily large gaps between complexity classes. The proof is done at an abstract, machine independent level, and is particularly aimed to identify the minimal set of assumptions required to prove the result (smaller than expected, actually). The work is part of a long term *reverse complexity* program, whose goal is to obtain, via a reverse methodological approach, a formal treatment of Complexity Theory at a comfortable level of abstraction and logical rigor.

1 Introduction

The Gap Theorem, first proved by Boris Trakhtenbrot in 1964 [33] and independently rediscovered eight years later by Allan Borodin [12], is a major theorem of Complexity Theory stating the existence of arbitrarily large gaps in the hierarchy of complexity classes. More explicitly, given a computable function g representing an increase in computational resources, one can effectively find a recursive function t such that the complexity classes with boundary functions t and $g \circ t$ are identical. In Borodin's words [12] "no matter how much better one computer may seem compared to the other, there will be a t such that the set of functions computable in time t is the same for both computers".

The Gap Theorem is a typical example of an "abstract" complexity result, that is a fact that can be proved without any reference to concrete computational models. Actually, our main motivation for addressing the formalization of this theorem was to derive, along a reverse methodological approach, a minimal set of logical assumptions sufficient to entail the result. The work is part of a larger "reverse complexity" program, outlined in [2], that applies the methodology of reverse mathematics [20, 30] to Complexity Theory, reconstructing from proofs the basic notions and assumptions underlying the major results of this field. The final, long term goal would be to obtain a formal, axiomatic treatment of Complexity Theory at a *comfortable* level of abstraction and mathematical rigor, reviving under a new perspective and through an innovative methodological approach the old quest for a machine-independent theory of complexity; we refer the reader to [2] for a short historical survey and a more exhaustive discussion of the Reverse Complexity program.

Mechanical devices such as proof assistants and interactive theorem provers play a major role in our program, not only to check the formal correctness of the resulting theory, but as actual *drivers* of the research. In fact, the reverse methodology presupposes a deep and frequent refactoring of the formalization, playing with different axiomatizations, improving the readability and maintainability of the code, or reducing its complexity: it is natural to expect to be supported by automatic devices along this process. As we already observed in [4], the situation is similar to the role of type checkers in software development, that are not simply meant to discriminate good programs from bad ones: type checkers are essential drivers of the development phase, and major tools for the deployment of lightweight, adaptive software methodologies, requiring frequent modifications and refactoring. This interactive exploitment of proof checkers, more than their batch usage as oracles to discriminate between correct and wrong arguments is, in our opinion, the new and challenging frontier of interactive provers.

The formalization of the Gap Theorem discussed in this paper was done with the assistance of the Matita Interactive Theorem Prover [7]. Matita is a light implementation of the Calculus of Inductive Construction developed and maintained at the University of Bologna. We do not have enough space to describe here the syntax of Matita's script language, so we shall omit formal proofs. We only wish to remark that Matita is a constructive system, and all proofs in this paper are constructive.

The development itself is accessible (and executable!) through the web interface of Matita [6] at the following url: http://matita.cs.unibo.it/matitaweb.shtml. An offline version can be downloaded at http://www.cs.unibo.it/ asperti/gap.tar.

The structure of the paper is the following. In the next section we shall start giving a rigorous formulation of the gap theorem and the original proof of Borodin [12], discussing it as well as other, later versions of the proof [15, 35, 17]. Section 3 is devoted to a brief review of the main modules of the Matita library that will be required for the formalization of the result, and in particular: bounded quantification, big operators and minimization, iteration, and a bit of combinatorics. In Section 4, we introduce the axiomatic setting that we shall use for the proof, essentially based on the existence of a suitable function (intuitively) playing the role of Kleene's T-predicate. Section 5 contains the formal proof, as well as the computation of an interesting and apparently original upper bound for the gap operator. Conclusions are discussed in Section 6.

2 Borodin's proof of the Gap Theorem

The gap theorem can be stated and proved without any reference to a concrete computational model. The typical setting adopted for expressing and proving it is Blum's abstract complexity framework [11], that applies to time, space, and many other reasonable complexity measures.

We write $f(x)\downarrow$ to express that the *partial* function f is defined for input x.

Definition 1. *(Blum [11]) A pair $\langle\varphi, \Phi\rangle$ is a* computational complexity measure *if φ is a principal effective enumeration of all partial recursive functions and Φ*

satisfies the following axioms:

$$(a)\ \varphi_i(\boldsymbol{n})\downarrow \leftrightarrow \Phi_i(\boldsymbol{n})\downarrow$$
$$(b)\ \textit{the predicate } \Phi_i(\boldsymbol{n}) = m \textit{ is decidable}$$

We adopt the convention that $\Phi_i(\boldsymbol{n}) = \infty$ if $\Phi_i(\boldsymbol{n})\uparrow$; in particular, the relation $\Phi_i(\boldsymbol{n}) > n$ also holds when $\Phi_i(\boldsymbol{n})$ is undefined.

Blum's axioms are very weak and very general, and nevertheless they are sufficient to prove a large number of interesting results in Complexity Theory. In particular, they proved to be a convenient setting to investigate the order structure of complexity classes under set theoretic inclusion [12, 26], their recursive presentability and the computational quality of such a presentation [25, 34, 24]. It is important to observe that, from a strictly formal point of view, Blum's "axioms" do not provide a real axiomatization, since they rely on the delicate notion of *computable function*. The fact that φ is a *principal effective enumeration* (see e.g.[28]) of all partial recursive functions is used in an essential way in most proofs based on Blum's axioms, usually by an invocation of Church Thesis. For this reason, Blum's axioms are not easy to use in a strictly formal framework, urging us to look for a more convenient and possibly more primitive axiomatisation.

Borodin's proof of the gap theorem is very concise and elegant, so we report his original argument here; we just slightly rephrased it for notational reasons, and retouched some bounds in order to get a more elegant formalization.

Theorem (Gap Theorem). Let $\langle\varphi, \Phi\rangle$ be a complexity measure, g a nondecreasing recursive function such that $\forall x. x \leq g(x)$. Then there exists a nondecreasing recursive function t such that, for sufficiently large n,

$$\Phi_i(n) \leq t(n) \qquad \text{or} \qquad \Phi_i(n) > g \circ t(n)$$

PROOF. Define t as follows:

- $t(0) = 1$,
- $t(n+1) = \mu k \geq t(n)\{\forall i \leq n.[\Phi_i(n) \leq k \text{ or } \Phi_i(n) > g(k)]\}$

Then:

1. for any n, k exists, since forall $i \leq n$ if $\Phi_i(n)\uparrow$ then $\forall k.\Phi_i(n) > g(k)$, and if $\Phi_i(n)\downarrow$ then $\exists k.\Phi_i(n) \leq k$.
2. k can be found recursively, since Φ is a complexity measure and thus $\Phi_i(n) \leq k$ and $\Phi_i(n) > g(k)$ are recursive predicates.
3. t satisfies the theorem, since $n \geq i$ implies that either $\Phi_i(n) \leq t(n)$ or $\Phi_i(n) > g \circ t(n)$.

QED.

An arbitrarily large t can be found to satisfy the conditions of the gap theorem, by taking k larger than $\max\{r(n), t(n)\}$ (for a suitable function r) in the definition of $t(n+1)$.

2.1 Discussion

The first problem in formalizing the previous proof in a proof system like Matita is due to the definition of t, that is formulated by means of general (unbounded) minimization. In general, this kind of recursive functions cannot be directly expressed in the Calculus of Constructions, and you should resort to an indirect encoding, by means of a suitable predicate, that is not particularly elegant. A second problem is point 1. in the proof, that seems to use *tertium non datur* on a semidecidable predicate, namely if $\Phi_i(n)\downarrow$ or not.

Luckily, as already pointed out by [35] (see also [17]) the existence of k can be proved in a more constructive way, and this will also induce a more constrained (primitive recursive) definition of t.

The general idea is relatively simple. Suppose we wish to find a k larger than a base value b, such that (for given i and n)

$$\Phi_i(n) \leq k \text{ or } \Phi_i(n) > g(k) \tag{1}$$

If $\Phi_i(n) \leq b$ then we take $k = b$ (note that the test $\Phi_i(n) \leq b$ is decidable!); otherwise we check if $\Phi_i(n) < g(b)$: if the answer is yes, we take $k = g(b)$ and otherwise we again take $k = b$ (the interesting point is not the decidability of equation (1), that is obvious, but the fact that we can put an upper bound to the search for a k solving the equation).

The previous reasoning can be iterated over all $i \leq n$: in particular, in the interval between b and $g^{n+1}(b)$ there must exist at least one k such that

$$\forall i \leq n. \Phi_i(n) \leq k \text{ or } \Phi_i(n) > g(k)$$

Suppose that at least j functions terminate within $b_j \leq g^j(b)$; if no other function terminates within $g(b_j)$ we are done; otherwise we take $b_{j+1} = g(b_j) \leq g^{j+1}(b)$ and go on. Since the number of terminating functions increases at each iteration, we shall eventually stop after $n+1$ steps.

Stated in a different way, let us consider the intervals $[g^i(b), g^{i+1}(b)[$ for $0 \leq i \leq n$ and all functions with index $j < n$ such that $\Phi_j(n) \leq g^{n+1}(b)$. We have at most n functions to distribute over $n+1$ intervals, so at least one interval must remain empty.

An interesting consequence of the previous reasoning, that apparently has never been emphasized by any author, is that we can compute an explicit upper bound u for t. In particular, let $\sigma(n) = \sum_{i \leq n} i = n \cdot (n+1)/2$; then, for any n,

$$t(n) \leq g^{\sigma(n)}(1) \leq g^{n^2}(1)$$

(see Section 5.2 for the simple proof).

3 Preliminaries

In this section, we shall discuss some of the background material we need for our development: bounded quantification 3.1, big operators and minimization 3.2, iteration 3.3 and a few combinatorial results 3.4.

Most of the results in this section are absolutely standard; we present them for the sake of completeness, in order to provide a self-contained description of the formalization, fixing names and notations.

In the rest of the article, all parts inside round boxes are Matita code; all proofs are skipped, but they are really simple.

3.1 Bounded quantification

We need to exploit a small library of results about bounded quantification.
A proposition P is decidable if $P \vee \neg P$ is provable:

```
definition decidable : Prop → Prop :=λA:Prop. A ∨ ¬ A.
```

It is trivial to prove that decidable propositions are closed with respect to logical connectives and bounded quantification:

```
lemma decidable_not: ∀P. decidable P → decidable (¬ P).

lemma decidable_or: ∀P,Q. decidable P → decidable Q → decidable (P∨ Q).

lemma decidable_forall: ∀P. (∀i.decidable (P i)) → ∀n.decidable (∀i. i < n → P i).

lemma decidable_exists: ∀P. (∀i.decidable (P i)) → ∀n.decidable (∃i. i < n ∧ P i).
```

On a decidable predicate we have the usual duality properties we know from classical logic, and in particular:

```
lemma not_exists_to_forall: ∀P,n.
  ¬ (∃i. i < n ∧ P i) → ∀i. i < n → ¬ P i.

lemma not_forall_to_exists: ∀P,n. (∀i.decidable (P i)) →
  ¬ (∀i. i < n → P i) → (∃i. i < n ∧ ¬ (P i)).
```

3.2 Big operators and minimization

Matita's library offers a well developed module on big operators, that has been described in some detail in [5].

A big operator is a higher-order construction that is supposed to iterate a function F over all elements in a given range, combining the results with an operator op; a nil value is returned when the range is empty. The range, the function F, the operator op and the value nil are all explicit parameters of the big operator.

Matita's notation is relatively standard ([10]), and has the following shape:

```
  \big[op,nil]_{ range description } F
```

The range description gives a name to the iteration variable and fixes the domain over which this variable is supposed to range. The elements in the range are supposed to be enumerated (that is not a limitation, considering that the range must be finite), hence the range is specified as an interval $i \in [a, b]$ where a is the lower bound and b is the upper bound (both included in the range). In case the lower bound is 0, the simpler notation $i \leq b$ can also be used. The variable i whose name can obviously be chosen by the user, is bound by the notation, and it usually occurs free in F.

The range can be further restricted specifying an additional boolean predicate, acting as a filter. For instance, the following notation represents the product of all primes less or equal to n

```
\big[times,1]_{p ≤n | primeb p} p
```

In this paper, we shall use big operators to iterate boolean functions over finite domains; for instance, the notation

```
\big[andb,true]_{i < n} (b i).
```

expresses the boolean conjunction of all $(b\ i)$ for all i less than n.

Minimization is essentially a big operator where we iterate the binary minimum function `min` on all elements in a given range enjoying a suitable predicate; the only problem is the definition of a default `nil` element. A relatively natural choice, in case we found no element in the range $[a, b[$ matching the test, is to return b:

```
definition Min :=λa,b,f.\big[min,b]_{i ∈ [a,b] | f i} i.
```

Although the definition is elegant, the possibility to exploit results on big operators for proving properties of Min poses some problems, in this case. The point is that the lemmas on big operations are hierarchically organized according to the algebraic structure associated with the operator. In the case of the minimum, we have associativity and commutativity, but we do not have a (generic) neutral element (see also [10] for the discussion of a similar problem relative to maximization on real numbers), so we have only access to very basic results.

For minimization we shall use the following ad hoc notation:

```
  μ_{ i ∈ [a,b] } p
```

to express the minimum element in the range $[a, b]$ that satisfies p (and returns the successor of b is no such element is found).

The main results about minimization that we shall exploit are the following: under the assumption that there exists an element in the range $[a, b]$ that satisfies f, then the minimum m satisfies f and moreover it is not greater than b (as a matter of fact, the definition of the function t of the gap theorem does not exploit minimality, but only existence).

```
lemma f_min_true: ∀f,a,b.
  (∃i. a ≤i ∧ i ≤b ∧ f i = true) →f (μ_{i ∈[a,b]} (f i)) = true.
```

```
lemma min_up: ∀f,a,b.
  (∃i. a ≤ i ∧ i ≤ b ∧ f i = true) → μ_{i ∈[a,b]}(f i) ≤ b.
```

3.3 Iteration

We shall need to consider progressive intervals of the kind $[g^i(b), g^{i+1}(b)[$, that requires a simple higher-order iterator:

```
let rec iter (A:Type[0]) (g:A→ A) n a on n :=
match n with
  [O ⇒ a
  |S m ⇒ g (iter A g m a)].
```

The notation $g^i(b)$ is hence a shorthand for `(iter nat g i b)`.
For the proof of the gap theorem we only need the following result:

```
lemma le_iter: ∀g,a. (∀x. x ≤ g x) → ∀i. a ≤ g^i a.
```

A few more simple lemmas about composition and monotonicity are used for computing an upper bound of the gap operator:

```
lemma iter_iter: ∀A.∀g:A→ A.∀a,b,c. g^c (g^b a) = g^(b+c) a.

lemma monotonic_iter: ∀g,a,b,i. (monotonic ? le g) → a ≤ b →
  g^i a ≤ g^i b.

lemma monotonic_iter2: ∀g,a,i,j. (∀x. x ≤ g x) → i ≤ j → g^i a ≤ g^j a.
```

The question mark in `monotonic_iter` is an *implicit parameter*, that is an argument automatically filled in by the type inference algorithm (in this case, `nat`).

3.4 A bit of combinatorics

The final ingredient we need for the proof of the gap theorem is a bit of combinatorics. The only delicate point in the definition of t is the termination of the minimization. The general idea is to consider a succession of $n + 1$ disjoint intervals $[r_i, r_{i+1}[$ for $0 \leq i \leq n$; then, we consider a set of at most n values to distribute over them (expressing the resources required by a machine with index $i < n$ to terminate on a specific input). Since we have strictly less items than intervals, one of the interval $[r_k, r_{k+1}[$ must remain empty, that gives the desired k. This is essentially a variant (an inverse form) of the *Pigeonhole principle* (also know as Dirichlet's drawer principle), that states that if n items are put into m pigeonholes where $n > m$, then at least one pigeonhole must contain more than one item.

A simple way to formalize the principle is by considering lists of natural numbers. Given a list l we shall denote with $|l|$ the length of l, and we shall write $x \in l$ to express that x is an element f the list. Let us consider a list l of *distinct* numbers in the interval $[0, n[$; then, obviously, $|l| \leq n$. The interesting point is that

$$|l| = n \leftrightarrow \forall i. i < n \to i \in l$$

This is expressed by the following notions and results in the library of Matita. The `unique` predicate express the fact that the list has no duplicates:

```
let rec unique A (l: list  A) on l :=
  match l with
  [ nil  ⇒ True
  | cons a  tl  ⇒ ¬ a ∈tl ∧ unique A tl].
```

Then, we can prove the following results (the proofs are not entirely straightforward, but these basic combinatorial principles belong by now to the folklore of interactive proving, so we do not discuss them).

```
lemma length_unique_le: ∀n,l.
  unique ? l  → (∀x. x ∈ l → x < n) → |l| ≤ n.

lemma eq_length_to_mem_all: ∀n,l.
  |l| = n → unique ? l → (∀x. x ∈ l → x < n) → ∀i. i < n → i ∈ l.

lemma lt_length_to_not_mem: ∀n,l.
  unique ? l  → (∀x. x ∈ l → x < n) → |l| < n → ∃i. i < n ∧ ¬ (i ∈ l).
```

4 Kleene's predicate

The starting point of our axiomatization is Kleene's predicate, that we shall represent with a function U with the following type:

```
axiom U: nat → nat → nat → option nat.
```

The intuitive idea is that

$$U\, i\, x\, r = \begin{cases} \text{Some } y & \text{if program } i \text{ on input } x \text{ returns } y \text{ with resource bound } r \\ \text{None} & \text{otherwise} \end{cases}$$

You should think of U as some agent performing the execution of the program, and checking that it respects the given resource bounds. The only assumption we make about U is about its "monotonicity" with respect to the amount of resources at our disposal:

```
axiom monotonic_U: ∀i,n,m,y. n ≤ m →
  U i x n = Some ? y → U i x m = Some ? y.
```

From the previous axiom we easily conclude that U is single valued:

```
lemma unique_U: ∀i,x,n,m,yn,ym.
  U i x n = Some ? yn → U i x m = Some ? ym → yn = ym.
```

We say that the computation of program x on input y terminates with resource bound r (notation: $\langle i, x\rangle \downarrow r$) if there exists y such that $U\,i\,x\,r = \text{Some } y$:

```
definition terminate :=λi,x,r. ∃y. U i x r = Some ? y.
```

It is straightforward to prove that the previous notion of (bounded) termination is decidable:

```
lemma terminate_dec: ∀x,i,n. ⟨x,i⟩ ↓ n ∨ ¬ ⟨x,i⟩ ↓ n.
```

In order to define the gap operator, we need a boolean version of the termination test:

```
definition termb :=λi,x,t.
  match U i x t with [None ⇒ false |Some y ⇒ true].
```

It is easy to prove that `termb` *reflects* `terminate` in the sense of [21]:

```
lemma termb_true_to_term: ∀i,x,t. termb i x t = true → ⟨i,x⟩ ↓ t.

lemma term_to_termb_true: ∀i,x,t. ⟨i,x⟩ ↓ t → termb i x t = true.
```

Exploiting the decidability of termination and the closure properties of section 3.1 it is easy to prove that that the test used in the definition of the gap function is decidable too:

```
lemma decidable_test : ∀n,x,r,r1.
       (∀i. i < n → ⟨i,x⟩ ↓ r ∨ ¬ ⟨i,x⟩ ↓ r1) ∨
       (∃i. i < n ∧ (¬ ⟨i,x⟩ ↓ r ∧ ⟨i,x⟩ ↓ r1)).
```

5 The proof of the gap theorem

Let us define the following predicate `gapP n x g r` expressing that for all programs up to n, there is a gap between r and $g\,r$ on input x:

```
definition gapP :=λn,x,g,r. ∀i. i < n → ⟨i,x⟩ ↓ r ∨ ¬ ⟨i,x⟩ ↓ g r.
```

The important fact is that, for any b, g, n, x we can always find a r in the interval between b and $g^n\,b$ such that `(gapP n x g r)`:

```
lemma upper_bound: ∀g,b,n,x. (∀x. x ≤ g x) →
  ∃r.b ≤ r ∧ r ≤ gˆn b ∧ gapP n x g r.
```

For the proof, we pass through the following auxiliary lemma

```
lemma upper_bound_aux: ∀g,b,n,x. (∀x. x ≤ g x) → ∀k.
  (∃j.j < k ∧
    (∀i. i < n → ⟨i,x⟩ ↓ g^j b ∨ ¬ ⟨i,x⟩ ↓ g^(S j) b)) ∨
  ∃l. |l| = k ∧ unique ? l ∧ ∀i. i ∈ l → i < n ∧ ⟨i,x⟩ ↓ g^k b .
```

This is proved by induction on k. At the inductive step $k0$ we reason by cases on the inductive hypothesis: we already found our j or we have a list of programs terminating with bound $g^{k0}\, b$ on input x. In the first case, we are done. In the other case we reason by cases on `decidable_test n x (g^k0 b) (g^(S k0) b))`. In the first case, we can take $j = k0$, and otherwise we have a program i that does not terminate in $g^{k0}\, b$ but terminates in $g^{k0+1}\, b$ on input x, and we add i to the list l.

Starting from `upper_bound_aux` it is now easy to prove `upper_bound`. The idea is to proceed by cases on `(upper_bound_aux g b n x Hg n)`, where `Hg` is the hypothesis that g is increasing. In case we have a j, we take $r = g^j\, b$ and we conclude easily. Otherwise, we have a list of programs terminating with bound $g^n\, b$ on input x. Since the list has length n, by property `eq_length_to_mem_all` all programs up to n must appear in this list, and we can just take $r = g^n\, b$.

5.1 The gap operator

The first step for defining the gap operator is to express the gap predicate `gapP` as a computable boolean function; a simple approach is to use big operators to encode bounded quantification:

```
definition gapb :=λn,x,g,r.
  \big[andb,true]_{i < n} ((termb i x r) ∨ ¬ (termb i x (g r))).
```

It is straightforward to prove that `gapb` reflects the gap predicate `gapP`, and in particular:

```
lemma gapb_true_to_gapP : ∀n,x,g,t.
  gapb n x g t = true → ∀i. i < n → ⟨i,x⟩ ↓ t ∨ ¬ (⟨i,x⟩ ↓ (g t)).

lemma gapP_to_gapb_true : ∀n,x,g,r.
  (∀i. i < n → ⟨i,x⟩ ↓ r ∨ ¬ (⟨i,x⟩ ↓ (g r))) → gapb n x g r = true.
```

It is now easy to define the gap operator as a higher-order function parametric in g:

```
let rec gap g n on n :=
  match n with
  [ O ⇒ 1
  | S m ⇒ let b :=gap g m in μ_{k ∈[b,g^n b]} (gapb n n g k)
  ].
```

From `upper_bound` it is easy to derive an analogous upper bound for gapb:

```
lemma upper_bound_gapb: ∀g,m. (∀x. x ≤ g x) →
  ∃r.gap g m ≤ r ∧ r ≤ g^(S m) (gap g m) ∧ gapb (S m) (S m) g r = true.
```

Then, using property `f_min_true` we easily conclude:

```
lemma gapS_true: ∀g,m. (∀x. x ≤ g x) → gapb (S m) (S m) g (gap g (S m)) = true.
```

and from the previous result we derive the expected behaviour of gap operator, in the general case:

```
theorem gap_theorem: ∀g,i.(∀x. x ≤ g x)→ ∃k.∀n.k < n →
   ⟨i,n⟩ ↓ (gap g n) ∨ ¬ ⟨i,n⟩ ↓ (g (gap g n)).
```

We just instantiate k with i and proceed by cases on i.

5.2 An upper bound

We conclude this section providing a simple upper bound for *gap g*, namely, for any n

$$\text{gap } g\, n \leq g^{\sigma(n)}(1) \leq g^{n^2}(1)$$

where $\sigma(n) = \sum_{i \leq n} i = n \cdot (n+1)/2$.

```
let rec sigma n :=
  match n with
  [ O ⇒ 0 | S m ⇒ n + sigma m ].

lemma gap_bound: ∀g. (∀x. x ≤ g x) → (monotonic ? le g) →
  ∀n.gap g n ≤ g^(sigma n) 1.
```

The proof is a simple induction on n. If $n = 0$ both sides are equal to 1. In the inductive case:

$$\begin{aligned}
\text{gap}\, g\, (S\, n) &\leq g^{(S\, n)}(\text{gap}\, g\, n) && \text{by min\_up using upper\_bound\_gapb} \\
&\leq g^{(S\, n)}(g^{\sigma(n)}1) && \text{by induction hypothesis} \\
&= g^{(S\, n + \sigma(n))}1 && \text{by iter\_iter} \\
&= g^{\sigma(Sn)}1 && \text{by definition of sigma}
\end{aligned}$$

It is worth observing that if g is primitive recursive, than (gap g) is too, and not too far away from g in the elementary hierarchy.

Many authors (see e.g Papadimitriou [27]) note the “fantastically fast growth” of the gap function (without providing an explicit bound), but after all it is no *so* scary (at least, compared to the enormous complexity of other logical problems [19]). Of course, the growth-rate of the function has little to do with its ability to create a gap: its upper bound $g^{\sigma(n)}1$ is a (space and time) constructible function, hence the hierarchy theorems apply and it does not define any gap. The really surprising fact is that in a relatively small interval as that comprised between $g(n)$ and $g^{\sigma(n)}1$ we can find a function with such a strange behaviour as (gap g).

6 Conclusions

In this paper, we presented a formalization in the Matita Interactive Theorem Prover of Borodin-Trakhtenbrot's Gap Theorem of Computational Complexity. The work is part of a huge program of formal revisitation of Complexity Theory, that we call *reverse complexity*, based on the application of methodologies typical of *reverse mathematics* [20, 30], consisting in a backward reconstruction from proofs of the basic notions and assumptions underlying the main results of the field.

The final goal is to understand, at a suitable level of abstraction and logical rigor, what *really matters* for a foundational investigation of Complexity, since we know that the details of the different, specific computational models are largely uninfluential.

The need for a better understanding of the logical grounds of complexity theory is testified by a long series of works aimed to provide machine-independent characterizations, spanning from the old works of Blum [11], to the recent field of Implicit Computation Complexity (see [8], and the bibliography therein), passing through a multitude of systems defined by controlling different aspects of the computation: explicit bounds on the growth rate of functions [14, 13], the logical power required for proving termination [18], the use and replication of computational resources [9]. See also [16] for a modern treatment of bounded arithmetical systems and an investigation of proof complexity from the point of view of computational complexity.

Even in the relatively simple case of the Gap Theorem, the reverse methodology was instructive, allowing us to clarify that the full power of Blums' abstract framework is not required for this proof. In particular, there is no need to refer to a *principal enumeration* of partial recursive functions, that would be a difficult notion to characterize at an abstract level.
We only postulated the existence of a function U, intuitively playing the role of Kleene's T'-predicate, but avoiding any explicit reference to a system of computable functions; we just assumed U to be monotonic:

```
axiom U: nat → nat → nat → option nat.

axiom monotonic_U: ∀i,n,m,y. n ≤ m →
  U i x n = Some ? y → U i x m = Some ? y.
```

The U function seems to provide an interesting starting point for many different investigations. For instance, exploiting the idea embodied in Kleene's normal form, we can easily axiomatize the existence of an interpreter (universal machine):

```
axiom universal: ∃u.∀i,x,y.
  ∃n. U u ⟨i,x⟩ n = Some y ↔ ∃m.U i x m = Some y.
```

In [3], we proved that any indexed set of partial functions that is closed under composition, contains all projections, an interpreter, and satisfies the s-m-n theorem of Recursion Theory is algorithmically complete, that is, it enumerates all computable functions. So, adding a few more axioms, we get a natural, abstract theory of computable functions. Morevoer, following the ideas outlined in [1], we can integrate the closure conditions on the class of computable functions by suitable complexity conditions, obtaining an interesting formal framework to address complexity theory.

Even more interestingly, we can investigate weaker logical frameworks, corresponding to system of subrecursive functions. For instance, for many interesting results of Complexity Theory, you do not need the existence of a full interpreter, but just the possibility to perform a restricted form of *bounded* interpretation. This is for instance the case of the well known hierarchy theorems of computational complexity [22, 31], whose formalization was investigated in [2]. The relation between full and bound interpretation from the point of view of Complexity Theory seems to be an argument worth to be further investigated too.

The new, major milestone in our program is however to provide a suitable, abstract axiomatization of the so called "reachability method". The general idea is to consider the graph of all possible configurations of the computational device, reducing the existence of a computation to a reachability problem in such a graph. Time bounds the dimension of the graph, and in turn the dimension of each configuration bounds the number of possible distinct nodes in the graph, allowing to establish the main relations between time and space. This is largely indepedent from any specific computational device, and it seems important to identify the right *abstract* setting underlying the previous ideas, paving the way to a reverse investigation of the well known theorems of Savitch [29] and Immerman-Szelepcsényi [23, 32].

References

1. Andrea Asperti. The intensional content of Rice's theorem. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), January 7-12, 2008, San Francisco, California, USA*, pages 113–119. ACM, 2008.
2. Andrea Asperti. Reverse complexity. In *Submitted for publication*, 2013.
3. Andrea Asperti and Agata Ciabattoni. Effective applicative structures. In *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, Proceedings*, volume 953 of *Lecture Notes in Computer Science*, pages 81–95. Springer, 1995.
4. Andrea Asperti, Herman Geuvers, and Raja Natarajan. Social processes, program verification and all that. *Mathematical Structures in Computer Science*, 19(5):877–896, 2009.
5. Andrea Asperti and Wilmer Ricciotti. A proof of Bertrand's postulate. *Journal of Formalized Reasoning*, 5(1):37–57, 2012.
6. Andrea Asperti and Wilmer Ricciotti. A web interface for Matita. In *Proceedings of Intelligent Computer Mathematics (CICM 2012), Bremen, Germany*, volume 7362 of *Lecture Notes in Artificial Intelligence*. Springer, 2012.
7. Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita interactive theorem prover. In *Proceedings of the 23rd International Conference on Automated Deduction (CADE-2011), Wroclaw, Poland*, volume 6803 of *LNCS*, 2011.

8. Patrick Baillot, Jean-Yves Marion, and Simona Ronchi Della Rocca (eds.). Special issue on implicit complexity. *ACM Transactions on Computational Logic*, 10(4), 2009.
9. Stephen Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 283–293. ACM, 1992.
10. Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In *TPHOLs*, pages 86–101, 2008.
11. Manuel Blum. A machine-independent theory of the complexity of recursive functions. *J. ACM*, 14(2):322–336, 1967.
12. Allan Borodin. Computational complexity and the existence of complexity gaps. *J. ACM*, 19(1):158–174, 1972.
13. P. Clote and G. Takeuti. On the computational complexity of algorithms. *Annals of Pure and Applied Logic*, 56(1-3):73–117, 1992.
14. A. Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, Amsterdam, 1964.
15. Robert L. Constable. The operator gap. *J. ACM*, 19(1):175–183, 1972.
16. Stephen Cook and Phuong Nguyen. *Logical Foundations of Proof Complexity*. Cambridge University Press, 2010.
17. Nigel J. Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.
18. D.Leivant. A foundational delineation of computational feasibility. In *Proceedings of the sixth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 2–11. IEEE, 1991.
19. Harvey Friedman. Some decision problems of enormous complexity. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 2–12. IEEE Computer Society, 1999.
20. Harvey Friedman and Stephen G. Simpson. Issues and problems in reverse mathematics. *Contemporary Mathematics*, 257:127–143, 2000.
21. Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.
22. J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transaction of the American Mathematical Society*, 117:285–306, 1965.
23. Neil Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, 1988.
24. L. H. Landweber and E. L. Robertson. Recursive properties of abstract complexity classes. *Journal of ACM*, 19(2):296–308, 1972.
25. Forbes D. Lewis. Unsolvability considerations in computational complexity. In *Proceedings of the snd Annual ACM Symposium on Theory of Computing (STOC), May 4-6, 1970, Northampton, Massachusetts, USA*, pages 296–308. ACM, 1970.
26. Edward M. McCreight and Albert R. Meyer. Classes of computable functions defined by bounds on computation. In *Proceedings of the 1st Annual ACM Symposium on Theory of Computing (STOC), May 4-6, 1992, Victoria, British Columbia, Canada*, pages 79–88. ACM, 1969.
27. Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
28. Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press, 1987.
29. Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970.

30. Stephen G. Simpson. *Subsystems of second order arithmetic*. Cambridge University Press, 2009.
31. R. E. Stearns, J. Hartmanis, and P. M. Lewis. Hierachies of memory limited computations. In *Proceedings of the 6th Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1965)*, pages 179–190. focs, 1965.
32. Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Inf.*, 26(3):279–284, 1988.
33. Boris Trakhtenbrot. Turing computations with logarithmic delay. *Algebra and Logic*, 3(4):33–48, 1964.
34. Paul R. Young. Toward a theory of enumeration. *Journal of ACM*, 16(2):328–348, 1969.
35. Paul R. Young. Easy constructions in complexity theory: gap and speed-up theorems. *Proceedings of A.M.S.*, 37(2):555–563, 1973.