The Speedup Theorem in a Primitive Recursive Framework

Andrea Asperti
Department of Computer Science and Engineering DISI, University of Bologna
andrea.asperti@unibo.it

Abstract

Blum's speedup theorem is a major theorem in computational complexity, showing the existence of computable functions for which no optimal program can exist: for any speedup function r there exists a function f^r such that for any program computing f^r we can find an alternative program computing it with the desired speedup r. The main corollary is that algorithmic problems do not have, in general, a *inherent* complexity.

Traditional proofs of the speedup theorem make an essential use of Kleene's fix point theorem to close a suitable diagonal argument. As a consequence, very little is known about its validity in subrecursive settings, where there is no universal machine, and no fixpoints. In this article we discuss an alternative, formal proof of the speedup theorem that allows us to spare the invocation of the fix point theorem and sheds more light on the actual complexity of the function f^r.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical Verification; D.2.4 [*Software/Program Verification*]: Correctness proofs; F.1.3 [*Complexity Measures and Classes*]: Complexity hierarchies, Machine independent complexity; F.3.3 [*Studies of Program Constructs*]: Type Structure, Program and recursion schemes; F.4.1 [*Mathematical Logic*]: Lambda calculus and related systems, recursive function theory

Keywords Speedup, Primitive Recursion, Machine independent complexity, Matita

1. Introduction

Each computational problem can be solved by an infinite number of different programs. Given some *complexity measure*, counting the amount of computational resources (such as time or space) required by the different computations, one is obviously interested to find, if possible, an *optimal* program, that is a program with minimal complexity. Blum's speedup theorem [11] proves that, no matter how complexity is measured, there are problems admitting no optimal solution. As a consequence, a computable function does not have in general a *inherent* computational complexity (unless expressed as an aggregate of the complexities of all its programs).

CPP '15, January 13–14, 2015, Mumbai, India.

http://dx.doi.org/10.1145/2676724.2693178

The speedup theorem is traditionally expressed in Blum's abstract framework [10]. We write $f(x)\downarrow$ to express that the *partial* function f is defined for input x, and $f(x)\uparrow$ otherwise.

DEFINITION 1. *(Blum [10]) 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:*

$$\begin{array}{ll}(a) & \varphi_i(\vec{n})\downarrow\leftrightarrow\Phi_i(\vec{n})\downarrow\\ (b) & \textit{the predicate } \Phi_i(\vec{n})=m \textit{ is decidable}\end{array}$$

As traditional in this setting, we adopt the convention that $\Phi_i(\vec{n})=\infty$ if $\Phi_i(\vec{n})\uparrow$; in particular, the relation $\Phi_i(\vec{n})>m$ also holds when $\Phi_i(\vec{n})$ is undefined. We use the acronym a.e. as an abbreviation of *almost everywhere*, that is for all but a *finite* number of values.

THEOREM 2. *(Speedup theorem [Blum [11]]) For any speedup function r there exists a computable function f such that for any $\varphi_i=f$ there exists $\varphi_j=f$ such that*

$$r(\Phi_j(x))\leq\Phi_i \qquad a.e.$$

The proof of the speedup theorem is traditionally split in two parts, proving first a slightly simpler *pseudo speedup theorem* where we do not require that the function $f'=\varphi_j$ computed by the faster program is precisely f but that it is just *almost equal* to it (notation: $f\approx f'$). Since the computation on a finite number of inputs does not affect the asymptotic complexity of functions, we obtain the speedup theorem as a simple corollary.

In this article, we shall merely focus on the pseudo speedup theorem. It is probably worth to anticipate that the proof is far from trivial: for instance, in his well known introduction to recursion function theory, Cutland ([15], pag.219) observes that *the proof of this theorem is probably the most difficult in this book*.

Another important point to remark is that we *shall not* work with Blum's abstract framework, simply because it is *not suitable* for a *formal* development of complexity theory. The point is that Blum's "axioms" do not provide a real axiomatization, since they rely on the delicate notion of *computable function*. In particular, the fact that φ is a *principal effective enumeration* (see e.g.[22]) 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's Thesis. This notion is quite difficult to express in formal terms, and would require an early commitment to a specific computational model. Our methodology, instead, was to start from an analysis of the algorithms involved in the proof to derive, through a reverse engineering process, the natural model of computation inside which we could comfortably work at a formal level.

Essentially, we shall preserve as much as possible of Blum's abstract framework (that is just an alternative way to express Kleene's T predicate), dropping the requirement that φ is a principal enumeration of *all* recursive functions, and adding suitable closure conditions as required by the algorithms in the proof. As we shall see, in

the case of the speedup theorem, the most natural framework seems to be provided by the class of primitive recursive functions.

As a byproduct of the proof we obtained an axiomatic framework for expressing and proving complexity properties of primitive recursive functions that seems to have an interest in its own.

The formalization has been conducted with the assistance of the Matita Interactive Theorem Prover [7, 8]. Matita is based, similarly to Coq, on the Calculus of Inductive Constructions; in particular, its *constructive* nature helped us to emphasize a few delicate issues in the proof of speedup theorem, discussed in the article (see Section 4.3 and the conclusions).

In the following sections, all parts inside frames are excerpts of Matita scripts. The whole development is available at `http://www.cs.unibo.it/~asperti/speedup.tar`.

In the current state of the development we largely (ab)used axioms as a way to declare the abstractions parametrizing the proof. In the end, when the axiomatic framework will become more stable, we plan to pack axioms inside suitable "algebraic" theories.

2. Outline of the proof

In this section, we give a quick outline of our proof of the speedup theorem. It is worth to stress again that our approach is an *original revisitation* of the traditional proofs, that is not only more suitable for a formal development, but also sheds more light on the actual complexity of the involved algorithms. We shall make a comparison with the customary approach in the conclusion of this article.

Let φ_i be an enumeration of (computable) functions. We shall write $\{i \odot x\} \downarrow t$ to express the fact that program i terminates its computation on input x with resource bound t (that is an intensional property of program i, and not an extensional property of φ_i).
Let h be a binary computable function; we define a family $g_i^h(x)$ of functions such that

- (cond.1) $g_i^h(x) = g_0^h(x)$ almost everywhere
- (cond.2) if $g_0^h(x) = \varphi_i$, then for no $x > i$, $\{i \odot x\} \downarrow h(i{+}1, x)$,

Moreover, we shall also prove that

- (cond.3) for any r, there exists h^r such that the complexity of computing $g_i^{r \circ h^r}(x)$ is less than $h^r(i, x)$.

Then, we are done. Suppose indeed that $f = g_0^{r \circ h^r}$ is computed by some program φ_i. Then, by (cond.2) the complexity of $\varphi_i(x)$ is definitely larger than $r(h^r(i+1, x))$, but by (cond 3.) and (cond.1) $g_{i+1}^{r \circ h^r}(x)$ computes an almost equal function with complexity $h^r(i+1, x)$ (as already pointed out by Young [27], here we are making an implicit use of the smn theorem).

The rest of the paper is devoted to the definition of $g_i^h(x)$ and the proof of conditions 1-3.

3. Basic Framework and notation

The starting point of our axiomatization is Kleene's T predicate, that is the *decidability of bounded interpretation*. Since we work in a constructive setting, this can be simply achieved by axiomatising the existence of a function U with the following type:

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

The intuitive meaning is that

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

The only assumption we make about U is that it is "monotonic" 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.
```

We recall that the question mark, in Matita, stands for an *implicit parameter*, that is a term that the type inference algorithm should be able to infer by itself. Similarly ... can be used to express an *arbitrary* number of implicit parameters.

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: $\{i \odot x\} \downarrow r$) 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: ∀i,x,n. {i ⊙ x} ↓ n ∨ ¬{i ⊙ x} ↓ n.
```

In order to define the family of functions g, 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 [16]:

```
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.
```

It is also convenient to have a function returning the result of computation as a natural number instead of an option

```
definition out :=λi,x,t.
  match U i x t with [ None ⇒ 0 | Some z ⇒ z].
```

Given a "partial" function f : nat $\to$ option nat we say that i is a code for f if $U\,i\,x$ is definitely equal to $f\,x$:

```
definition code_for :=λf,i.∀x.∃n.∀m. n ≤ m → U i x m = f x.
```

Let us also observe that we can always regard a total function f : nat $\to$ nat as a partial function of type nat $\to$ option nat via the following, obvious transformation:

```
definition total :=λf.λx:nat. Some nat (f x).
```

To conclude this section, it is worth to observe that, as weak as it can appear, this basic framework is already sufficient to prove the gap theorem of computational complexity [2, 12].

4. The family g

The idea behind the definition of the function $g_u^h(x)$ is to make it different from any function φ_i, $u \le i < x$, such that the computation $\varphi_i(x)$ terminates with complexity $h(i+1, x)$ (but it doesn't for any input smaller than x, i.e. i has not been "cancelled" already). This is enough to ensure condition 2 of the outline, and condition 1 will follow easily. For condition 3, we shall need to study the complexity of a program computing g.

4.1 Big operators

The definition of the speedup function involves bounded minimization μ and the computation *Max* of a maximum element in a given range. Both are simple examples of so called big-operators [9].

A big operator is a higher-order construct iterating a function F over all elements in a given range, and combining the results with an operator op. A default value nil is returned when the range is empty.

Matita's notation for big operators has the following shape (see [5] for more details):

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

In this article, we shall mostly work with numerical ranges of the form $i \in [a, b]$ or $i \in [a, b[$ where a and b are the lower and upper bound of an interval of natural numbers (with the upper bound b respectively included and excluded from the range; similarly for the lower bound). The variable i may occur free in F, and is bound by the operator. The range can be further restricted specifying an additional boolean predicate, acting as a filter.

For instance, minimization can be expressed as a big operator where we combine elements in a given range enjoying a filter predicate `p` via the binary minimum operator `min`. In particular, we adopt the following notations:

```
μ_{i ∈ [a,b] | p i} = \big[min,S b]_{i ∈ [a,b] | p i} i.

Max_{i ∈[a,b] | p i}(f i) = \big[max,0]_{i ∈ [a,b] | p i}(f i).
```

4.2 The formal definition of g

Using minimization and Max we can give a very simple, formal definition of the speedup function:

```
definition  min_input :=λh,i,x.
  μ_{y ∈ ]i,x] } (termb i y (h (S i) y)).

definition  g :=λh,u,x.
  S (Max_{i ∈[u,x[ | min_input h i x = x} (out i x (h (S i) x))).
```

In the literature, if $min\_input\, h\, i\, x = x$ it is customary to say that i *is cancelled at stage* x; the range of Max in g is hence the set

$$C_u^h(x) = \{i \in [u, x[\mid i \text{ is cancelled at stage } x\}$$

While the explicit definition of $C_u^h(x)$ definitely helps in the discursive exposition of the proof, at the formal level it is better to implicitly describe it via its boolean characteristic function, used in the range of big operators.

4.3 Basic properties of g_u^h

For any $x \leq u$ the set $C_u^h(x)$ is empty, so $g_u^h(x) = 1$:

```
lemma le_u_to_g_1 : ∀h,u,x. x ≤ u → g h u x = 1.
```

The first relevant fact is the following *cancellation property*:

> for any u, there exists an index n_u such that, for any $x > n_u$ no $i < u$ belongs to $C_0^h(x)$.

Indeed, consider an i in the interval $[0, u[$. If, for some x, $\varphi_i(x)$ terminates with complexity $h\,(S\,i)\,x$ then there will exists a minimum input n_i verifying such a property; if we define n_u as the maximum of all n_i, i will be eventually cancelled before n_u and hence cannot belong to $C_0^h(x)$.
The important corollary is that for $x > n_u$, $C_0^h(x) = C_u^h(x)$ and hence

$$g_0^h(x) \approx g_u^h(x)$$

that is our first condition on g.

Unfortunately, the previous proof of the cancellation property is not constructive: the search for a minimum n_i such that $\varphi_i(n_i)$ terminates with complexity $h\,(S\,i)\,n_i$ can potentially diverge and no upper bound can be provided in advance.

Luckily, we can content ourselves with a slightly weaker property (that is essentially a double negation variation of the previous one): what we can prove constructively is that, for any u the following property *is absurd*:

$$P(u) = \forall n_u \exists x > n_u \exists i < u, i \in C_0^h(x)$$

The proof is by induction on u. If $u = 0$ the result is trivial since we have no $i < 0$. At the inductive step, we need to prove that $\neg P(u) \rightarrow \neg P(u+1)$, that by contraposition reduces to $P(u+1) \rightarrow P(u)$. Take n_u; by $P(u+i)$ instantiated with $n_{u+1} = n_u$, we know that there exists $x > n_u$ such that for some $i < S\,u, i \in C_0^h(x)$. If $i < u$ we are done; if $i = u$ we apply again $P(u+1)$ starting from $n_{u+1} = x$, that is enough to cancel i too.

Formally, the property can be expressed in the following form:

```
lemma eventually_cancelled : ∀h,u.¬ ∀nu.∃x.
 nu < x ∧
 max_{i ∈ [0,u[ | min_input h i x = x} (out i x (h (S i) x)) ≠ 0.
```

We must also revisit the notion of being almost equal. In a constructive setting, the basic relation is not equality but *apartedness*, and two objects are defined to be equal when they cannot be taken apart. Accordingly, two objects are almost equal when it is false that they can be eventually taken apart:

```
definition  almost_equal :=λf,g: nat → nat.
  ¬ ∀nu.∃x. nu < x ∧ f x ≠ g x.
```

Henceforth, we shall use the notation $f \approx g$ to express the fact that two functions are almost equal in the sense of the previous definition.

Using lemma `eventually_cancelled` and basic decomposition properties of big operators it is then easy to prove our first condition for g:

```
lemma condition_1: ∀h,u.g h 0 ≈ g h u.
```

Let us now tackle the second condition:

```
lemma condition_2: ∀h,i.
  code_for  ( total  (g h 0))  i  → ¬ ∃x. i<x ∧ {i ⊙ x} ↓  h (S i) x.
```

Suppose i is a program computing g_0^h, and let us suppose that there exists $x > i$ such that $\varphi_i(x)$ terminates with complexity $h\,(Si)\,x$. Then, there must exist a minimum x_0 satisfying this property, that implies that $i \in C_0^h(x_0)$. By definition of g, we must have $g_0^h(x_0) > \varphi_i(x_0)$, contradicting the fact that i was a program for g_0^h. The formal proof is not much longer, and essentially analogous.

5. The complexity of g, informally

Let us consider again the definition of g:

```
definition  min_input :=λh,i,x.
  μ_{y ∈ ]i,x] } (termb i y (h (S i) y)).

definition  g :=λh,u,x.
  S (max_{i ∈[u,x[ | min_input h i x = x} (out i x (h (S i) x))).
```

The complexity of computing g is bound by the sum, for $u \leq i < x$, of the complexity of computing *(min_input h i x)* plus the complexity of computing *(out i x (h (S i) x))*. In turn, the complexity of

(min_input h i x) is bound by the sum, for $i < y \leq x$, of the complexity of *(termb i y (h (S i) y))*.

If h is a constructible function (see section 6.3), the complexity of computing both *(out i x (h (S i) x))* and *(termb i x (h (S i) x))* is bound by the complexity of running an interpreter for i on input x with a resource bound of *(h (S i) x)*: let us call *sU(i,x,h (S i) x)* such a complexity.
So, the complexity of $g_u^h(x)$ does only depend on the sum of the values of *sU(i,y,h (S i) y)* for $u \leq i < x$ and $i < y \leq x$ (see Figure 1).

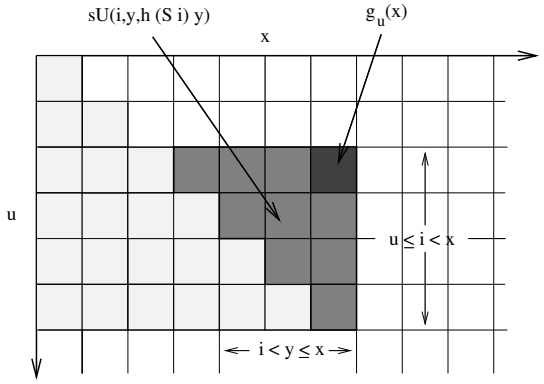

Figure 1. Complexity of $g_u^h(x)$ w.r.t. $sU(i, y, h\,(S\,i)\,y)$

The complexity of $g_u^h(x)$ has hence a behaviour similar to that described in (see Figure 2): the complexity of $g_u^h(x)$ is increasing in x and decreasing in u. At the end, we are interested to instantiate

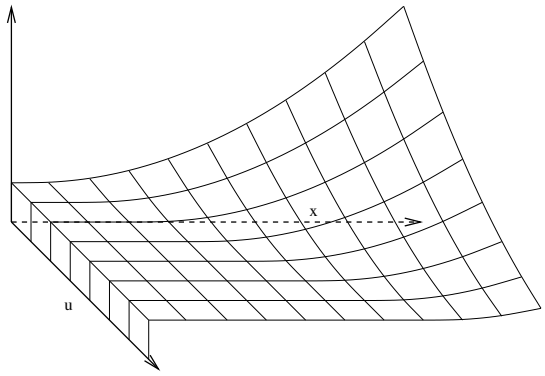

Figure 2. Complexity behaviour of $g_u^h(x)$

h with an upper bound to the complexity $g_u^h(x)$ (since g itself is not recursive, this will not be difficult).
We may assume that *sU* is monotonic in all its arguments[1]. Supposing h is antimonotonic in its first argument and monotonic in the second one, we obtain the following upper bound for the complexity of $g_u^h(x)$:

$$\sum_{u \leq i < x} \sum_{i < y \leq x} sU(i, y, h\,(S\,i)\,y) \leq (x-u)^2 \cdot sU(x, x, h\,(S\,u)\,x) \quad (1)$$

[1] Recall that *sU* is a complexity function: since we are interested in the asymptotic behaviour, we may e.g. suppose that, on in input x, it is defined as the maximum complexity for all input less or equal to x.

Let now r be an arbitrary, monotonic, increasing and constructible function, and let us consider the following recursive function:

$$h^r(u,x) = \begin{cases} 1 & \text{if } x \leq u \\ (x-u)^2 \cdot sU(x, x, r(h^r(u+1, x))) & \text{otherwise} \end{cases}$$

Then, by equation 1 the complexity of $g_u^{r \circ h^r}(x)$ is less than

$$(x-u)^2 \cdot sU(x, x, r(h^r\,(S\,u)\,x)) = h^r(u,x)$$

that is our third condition.

The previous proof has been slightly simplified for the sake of clarity. The point that was somewhat overlooked is the complexity of $g_u^h(x)$ when $u \leq x$, that we assumed to be constant. While this is sensible for certain complexity measures such as space, for others it can depends on inputs. This is for instance the case for time complexity, where we usually require the program to *consume* its inputs, and hence to have at least a complexity proportional to their dimension.

In our formal framework we shall assume to have a *minimal structural complexity MSC* expressing this cost. Intuitively, you can think of *MSC* as the complexity of the identity program; typically, all basic primitive operations will be assumed to have this complexity.

In the case of time, MSC is essentially a size measure, that is a logarithmic function; in the case of space, $MSC \in O(1)$. Instead of fixing a specific measure, however, it is more interesting to look for the abstract properties required for this function.

6. The formal complexity framework

6.1 Pairing

It is convenient to have at our disposal a primitive method for packing together multiple outputs into a single one. This means we need a pairing function mapping two natural numbers a and b into a natural number $\langle a, b \rangle$. We assume pairing is a bijection, with projections called fst and snd.

For instance, using pairs we can naturally rephrase U in the following way, avoiding the option in output:

```
definition pU : nat → nat → nat → nat :=
  λi,x,r.⟨termb i x r,out i x r⟩.

lemma pU_vs_U_Some : ∀i,x,r,y.
  pU i x r = ⟨1,y⟩ ↔ U i x r = Some ? y.

lemma pU_vs_U_None : ∀i,x,r.
  pU i x r = ⟨0,0⟩ ↔ U i x r = None ?.
```

We shall also use the pairing function to avoid working with n-ary functions, that would be an annoying complication in a formal setting.

The existence of the pairing function and its properties are assumed axiomatically. The reader may wonder why we did not give a more concrete definition. The point is that later on (see Section 6.5) we shall be forced to make complexity assumptions on pairing operations, and it does not make much sense to attribute an abstract complexity measure to a concrete encoding.

6.2 Complexity Classes

We shall define complexity classes in terms of the asymptotic behavior of functions, so we need a small library of results dealing with the traditional operators (big-O, small-o, etc.) used to characterize functions according to their growth rates. In this article, we shall only use the big-O operator, defined as follows:

```
definition O: relation (nat→nat) :=λf,g.
  ∃c.∃n0.∀n. n0 ≤ n → g n ≤ c∗ (f n).
```

It is easy to develop a small library of results, expressing the typical properties of these operators.

A delicate issue in the formalization of Complexity Theory is the choice between expressing complexity in terms of inputs or in terms of their size. We already investigated the latter approach in [3]; also in order to compare the two approaches we shall follow the first route in this article [2].

For any complexity function s, we define the complexity class $C\,s$ as the collection of all programs that terminate their computation in $O(s)$:

```
C :=λs,i . ∃c.∃a.∀x.a ≤ x → ∃y. U i x (c∗(s x)) = Some ? y.
```

Similarly, we say that a total function f : nat $\to$ nat is in CF s if there exists a program $i \in C\,s$ such that i is a code for (total f) (see Section 3 for the definition of total).

```
definition CF :=λs,f . ∃i . code_for ( total f) i ∧ C s i .
```

If a function f is computable in $O(h)$ we expect that both the input x and the output $f(x)$ have a minimal structural complexity bound by $O(h)$ as well:

```
axiom MSC_in: ∀f,h. CF h f → ∀x. MSC x ≤ h x.
axiom MSC_out: ∀f,h. CF h f → ∀x. MSC (f x) ≤ h x.
```

Moreover, we assume MSC is monotonic, less than the identity, and distributes over pairing (this latter axiom can be probably spared with)

```
axiom MSC : nat → nat.
axiom MSC_le: ∀n. MSC n ≤ n.
axiom monotonic_MSC: monotonic ? le MSC.
axiom MSC_pair: ∀a,b. MSC ⟨a,b⟩ ≤ MSC a + MSC b.
```

It is easy to prove that CFs is extensional and monotonic:

```
lemma ext_CF : ∀f,g,s. (∀n. f n = g n) → CF s f → CF s g.
lemma ext_CF_l: ∀f,s1,s2. (∀n. s1 n = s2 n) → CF s1 f → CF s2 f.
lemma O_to_CF: ∀s1,s2,f. O s2 s1 → CF s1 f → CF s2 f.
```

6.3 Constructibility

A function is said to be constructible w.r.t some complexity measure when the complexity of the computation is manifest in the (size of the) result. More formally, a function f is called space-constructible if there is a program that given an input of length n returns an output of length $f(n)$ running in $O(f(n))$ space; the definition of time-constructible function is analogous, changing space with time.

Avoiding to talk about the size of inputs, the formal definition is even simpler: f is constructible if, for all n, $f(n)$ can be computed in $O(f(n))$.

```
definition constructible :=λs. CF s s .
```

Constructible functions play an essential role when used as bounds for running an interpreter: the interpretation can be run in time $O(f)$ only if the bound itself can be computed in $O(f)$.

A typical example of a function that is not time constructible is the logarithmic function, since the time required for reading the input is already linear in its dimension. Similarly no decision algorithm is computed by a constructible function.

[2] It is worth to observe that the choice between the two paradigms is also related to the issue of the arity of functions. The point is that from the size of $\langle a, b\rangle$ we can deduce nothing interesting about the size of a and b. Expressing complexity in terms of the size of inputs has, among others, the disadvantage to force working with n-ary functions.

It is worth to observe that given any computable function f we may always find a computable function g such that $f \in O(g)$ and g is constructible. In particular, the complexity order of a computable function is usually expressed by a *constructible* bound.

These are a couple of simple lemmas we may prove on constructible functions.

```
lemma constr_comp : ∀s1,s2. constructible s1 → constructible s2 →
  (∀x. x ≤ s2 x) → constructible (s2 ∘ s1 ).

lemma ext_constr: ∀s1,s2. (∀x.s1 x = s2 x) → constructible s1 →
constructible s2.
```

6.4 Smn-theorem

The Smn-theorem of computability theory (see [22]) says that any instance of a computable function obtained by fixing some input n is still computable, and that a program computing the instance can be found effectively as a function n.

It is natural to extend the theorem taking complexity into account. As a matter of fact, the complexity of the instance cannot be larger than the complexity of the source program (see e.g. [1, 27]).

In our framework, this can be stated in a particularly simple and elegant way:

```
axiom smn: ∀f,s. CF s f → ∀x. CF (λy.s ⟨x,y⟩) (λy.f ⟨x,y⟩ ).
```

6.5 Complexity of primitive constructs

We shall assume the following complexities for the primitive structural operations:

```
axiom CF_id: CF MSC id.
axiom CF_comp_fst: ∀h,f. CF h f → CF h (fst ∘ f ).
axiom CF_comp_snd: ∀h,f. CF h f → CF h (snd ∘ f ).
axiom CF_comp_pair: ∀h,f,g. CF h f → CF h g → CF h (λx. ⟨f x,g x⟩ ).
axiom CF_eqb: ∀h,f,g. CF h f → CF h g → CF h (λx.eqb (f x) (g x )).
```

For composition we have

```
axiom CF_comp: ∀f,g,sf,sg,sh. CF sg g → CF sf f →
  O sh (λx. sg x + sf (g x)) → CF sh (f ∘ g ).
```

6.6 Primitive Recursion

The computations of μ and Max are based on big operators, that are a particular case of primitive recursion. The primitive recursion scheme is defined in the following way, where m should be understood as a *vector* of parameters):

```
let rec prim_rec (k,h: nat → nat) n m on n :=
  match n with
  [ O ⇒ k m
  | S a ⇒ h ⟨a,⟨prim_rec k h a m, m⟩⟩].

definition unary_pr :=λk,h,x. prim_rec k h ( fst x) (snd x ).
```

Supposing that k can be computed in $O(s_k)$ and h can be computed in $O(s_h)$ we expect to be able to compute $unary\_pr\ k\ h$) with a complexity, in time, that grows as the following function:

```
let rec prim_rec_compl (k,h,sk,sh: nat → nat) n m on n :=
  match n with
  [ O ⇒ sk m (∗ cost of k ∗)
  | S a ⇒ prim_rec_compl k h sk sh a m + (∗ cost of rec. call ∗)
          sh ⟨a, ⟨prim_rec k h a m, m⟩⟩]. (∗ cost of h ∗)
```

This can be stated by the following axiom, expressed in unary format:

axiom CF_prim_rec_gen: $\forall$k,h,sk,sh,sh1. CF sk k $\to$ CF sh h $\to$
O sh1 (λx. fst (snd x) +
sh $\langle$ fst x,$\langle$unary_pr k h $\langle$ fst x,snd (snd x)$\rangle$,snd (snd x)$\rangle\rangle$) $\to$
CF (unary_pr sk sh1) (unary_pr k h).

6.7 Arithmetic primitives

Primitive recursion provides a convenient way to express the flow structure of iterative programs, but we cannot write efficient arithmetic programs without resorting to an efficient representation of integers. Even in this case, the efficiency of primitive recursive algorithms suffer by well know limitations [14, 19], mostly due to the exceeding sequentialization imposed by high-level programming constructs.

For this reason, we shall hence suppose to have at our disposal a sufficiently large set of arithmetic primitives, computable with minimal structural complexity:

axiom CF_compS: $\forall$h,f. CF h f $\to$ CF h (S $\circ$ f).
axiom CF_le: $\forall$h,f,g. CF h f $\to$ CF h g $\to$ CF h (λx. leb (f x) (g x)).
axiom CF_eqb: $\forall$h,f,g. CF h f $\to$ CF h g $\to$ CF h (λx.eqb (f x) (g x)).
axiom CF_max: $\forall$h,f,g. CF h f $\to$ CF h g $\to$ CF h (λx. max (f x) (g x)).
axiom CF_plus: $\forall$h,f,g. CF h f $\to$ CF h g $\to$ CF h (λx. f x + g x).
axiom CF_minus: $\forall$h,f,g. CF h f $\to$ CF h g $\to$ CF h (λx. f x $-$ g x).

In order to prove the constructibility of the complexity bound of the function h, we shall also need the arithmetical product. This function cannot be computed with minimal structural complexity; we assume the usual square bound:

axiom CF_times: $\forall$f,g,h. CF h f $\to$ CF h g $\to$
CF (λx.h x + MSC (f x) $*$ MSC (g x)) (λx.f x $*$ g x).

By the previous axiom, it is easy to obtain the following result, that is usually simpler to use:

lemma CF_times1: $\forall$f,g,sf,sg. CF sf f $\to$ CF sg g $\to$
CF (λx.sf x $*$ sg x) (λx.f x $*$ g x).

6.8 If then else

Primitive recursion embeds definition by cases, and hence the "if then else" construct.

Exploiting the obvious coercion form bool to nat, we can easily prove the following lemma:

lemma if_prim_rec: $\forall$b:nat $\to$ bool. $\forall$f,g:nat $\to$ nat.$\forall$x:nat.
if b x **then** f x **else** g x = prim_rec g (f $\circ$ snd $\circ$ snd) (b x) x.

More interestingly, our complexity assumptions for primitive recursion allow us to derive the following complexity bound for the if then else:

lemma CF_if: $\forall$b:nat $\to$ bool. $\forall$f,g,s. CF s b $\to$ CF s f $\to$ CF s g $\to$
CF s (λx. **if** b x **then** f x **else** g x).

The proof is not complex, but requires a clever exploitation of the fact that the result of b is bounded.

6.9 Minimization and Max

Using primitive recursion we can also encode all big operators, and in particular the operations of minimization and Max of Section 4.1. As in the case of the "if then else", we can then derive complexity bounds for these constructs in our axiomatic framework. In particular, we have been able to prove the following results:

lemma CF_max: $\forall$a,b.$\forall$p:nat $\to$ bool.$\forall$f,ha,hb,hp,hf,s.
CF ha a $\to$ CF hb b $\to$ CF hp p $\to$ CF hf f $\to$
O s (λx.ha x + hb x +
($\sum$_{i $\in$[a x ,b x[}
(hp $\langle$i,x$\rangle$ + hf $\langle$i,x$\rangle$ + max_{i $\in$ [a x, b x [}(hf $\langle$i,x$\rangle$)))) $\to$
CF s (λx.max_{i $\in$[a x,b x[| p $\langle$i,x$\rangle$ }(f $\langle$i,x$\rangle$)).

lemma CF_mu: $\forall$a,b.$\forall$f:nat $\to$ bool.$\forall$sa,sb,sf,s.
CF sa a $\to$ CF sb b $\to$ CF sf f $\to$
O s (λx.sa x + sb x +
$\sum$_{i $\in$[a x ,S(b x)[}
(sf $\langle$i,x$\rangle$ + MSC $\langle$b x $-$ i,$\langle$S(b x),x$\rangle\rangle$)) $\to$
CF s (λx.μ_{i $\in$[a x,b x] }(f $\langle$i,x$\rangle$)).

In both cases, it is convenient to consider more relaxed but simpler bounds. In our proof of the speedup theorem, we exploited the following variants:

lemma CF_max2: $\forall$a,b.$\forall$p:nat $\to$ bool.$\forall$f,ha,hb,hp,hf,s.
CF ha a $\to$ CF hb b $\to$ CF hp p $\to$ CF hf f $\to$
O s (λx.ha x + hb x +
(b x $-$ a x)$*$max_{i $\in$ [a x, b x [}(hp $\langle$i,x$\rangle$ + hf $\langle$i,x$\rangle$)) $\to$
CF s (λx.max_{i $\in$[a x,b x[| p $\langle$i,x$\rangle$ }(f $\langle$i,x$\rangle$)).

lemma CF_mu4: $\forall$a,b.$\forall$f:nat $\to$ bool.$\forall$sa,sb,sf,s. ($\forall$x.sf x $>$ 0) $\to$
CF sa a $\to$ CF sb b $\to$ CF sf f $\to$
O s (λx.sa x + sb x +
(S(b x) $-$ a x)$*$Max_{i $\in$[a x ,S(b x)[}(sf $\langle$i,x$\rangle$)) $\to$
CF s (λx.μ_{i $\in$[a x,b x] }(f $\langle$i,x$\rangle$)).

6.10 Complexity of bounded interpretation

The final ingredient is the possibility to perform *bound* interpretation *inside* the system, that essentially amount to the *internalization* of the function U.

More precisely, let us consider the unary version of the function pU of Section6.1:

definition pU_unary :=λp. pU (fst p) (fst (snd p)) (snd (snd p)).

We simply require that pU is computable with some complexity sU.

axiom sU : nat $\to$ nat.
axiom CF_U : CF sU pU_unary.

We suppose sU to be mononotic in all its arguments (as we already observed, this is a natural requirement for any complexity function).

axiom monotonic_sU: $\forall$i1,i2,x1,x2,s1,s2.
i1 $\leq$ i2 $\to$ x1 $\leq$ x2 $\to$ s1 $\leq$ s2 $\to$
sU $\langle$i1,$\langle$x1,s1$\rangle\rangle$ $\leq$ sU $\langle$i2,$\langle$x2,s2$\rangle\rangle$.

Moreover, we expect sU to grow more than linearly in the resource bound:

axiom sU_le: $\forall$i,x,s. s $\leq$ sU $\langle$i,$\langle$x,s$\rangle\rangle$.

It is easy to prove that sU is also the complexity of the (unary versions of) $termb$ and out:

lemma CF_termb: CF sU termb_unary.
lemma CF_out: CF sU out_unary.

7. The complexity of g

We have now all the machinery we need to tackle the formal analysis of the complexity of the speedup function. The details are not very interesting: what was interesting was the backward reconstruction process that, with the assistance of an interactive prover, allowed us to build the previous axiomatic framework from the problem we were meant to solve. The current structure of the development still reflects this small step backward activity[3].

The main result is the following. Let us consider the unary version of g:

```
definition unary_g :=λh.λux. g h (fst ux) (snd ux).
```

Then, supposing that h is a constructible binary function antimonotonic in its first argument and monotonic in the second one, the complexity of *unary_g* is *sg*, where *sg* satisfies the following equation (c is just a function sufficiently large to cover the complexity of structural operations):

```
definition c :=λx.(S (snd x−fst x))∗MSC ⟨x,x⟩.

definition sg :=λh,x.
  let a := fst x in
  let b :=snd x in
  c x + (b−a)∗(S(b−a))∗sU ⟨x,⟨snd x,h (S a) b⟩⟩.

lemma compl_g11 : ∀h.
   constructible (λx. h (fst x) (snd x)) →
  (∀n. monotonic ? le (h n)) →
  (∀n,a,b. a ≤b →b ≤n →h b n ≤h a n) →
    CF (sg h) (unary_g h).
```

8. Closing the argument

To close the argument, we need to instantiate the function h of g with a *constructible* upper bound of its complexity; more precisely, we are looking for (a constructible upper bound to) a fix point of *sg*.

We can easily define such a function using primitive recursion; the bound we provide is not particularly tight; we just aimed to a function for which we could easily prove constructibility. Here is the actual definition:

```
  let rec h_of_aux (r:nat →nat) (d,b:nat) on d : nat :=
  match d with
  [ O ⇒ ⟨⟨b,b⟩,⟨b,b⟩⟩
  | S d1 ⇒ let c :=⟨d1,⟨h_of_aux r d1 b,b⟩⟩ in
            (S c)∗(S(S c))∗sU ⟨c,⟨b,r (h_of_aux r d1 b)⟩⟩ ].

definition h_of :=λr,p.
  let m :=max (fst p) (snd p) in
  h_of_aux r (snd p − fst p) m.
```

It is easy to prove that, if r is a monotonic function and for any n, $n \leq r(n)$, then *(h_of r)* is antimonotonic in the first argument and monotonic in the second one. Moreover, if *r* is constructible, *(h_of r)* is constructible too.

Putting together all these results, it is easy to conclude the following lemma:

```
lemma speed_compl: ∀r:nat →nat.
  (∀x. x ≤r x) →monotonic ? le r → constructible r →
  CF (h_of r) (unary_g (λi,x. r(h_of r ⟨i,x⟩ ))).
```

Using *smn*, we get as a corollary that

```
lemma speed_compl_i: ∀r: nat →nat.
  (∀x. x ≤r x) →monotonic ? le r → constructible r →
  ∀i. CF (λx.h_of r ⟨i,x⟩) (λx.g (λi,x. r(h_of r ⟨i,x⟩)) i x).
```

We can now prove the pseudo speedup theorem that we state in the following way: for any function r that is monotonic, constructible and for any n, $n \leq r(n)$, there exists a function f such that if the complexity of f is s_f, there exists another function f' such that $f' \approx f$ and f' has a complexity $s_{f'}$ where $r \circ s_{f'} \in O(s_f)$:

```
theorem pseudo_speedup:
∀r: nat →nat. (∀x. x ≤r x) →monotonic ? le r → constructible r →
  ∃f.∀sf. CF sf f →∃f',sf'. f ≈f' ∧CF sf' f' ∧O sf (r ∘ sf').
```

For the proof, we just take

$$f = g\ (\lambda i, x.r\ (h\_of\ r\ \langle i, x\rangle))\ 0)$$

and

$$f' = g\ (\lambda i, x.r\ (h\_of\ r\ \langle i, x\rangle))\ (S\ i)$$

By lemma *speed_compl_i* the complexity of f' is then

$$sf' = \lambda x.h\_of\ r\ \langle S\ i, x\rangle$$

and the proof that $r \circ sf' \in O(sf)$ is trivial.

9. Conclusions

Our proof has been inspired by the simplified version of Blum's proof proposed by Young [27]. The main difference is that in Young's proof the function $g_u(x)$ is defined in terms of *its own complexity*; quoting from [27]:

> *We will also assume that it is legitimate to define a function recursively, not just from its earlier values, but also from its earlier run times. Intuitively, this amounts to assuming that if we used a program to calculate the value of a function at an early argument, we can know the resources used in the computation even if we do not explicitly know the entire program used for computing the function. Formally of course, one must use the recursion theorem or some other means to validate such an argument.*

A more formal version of Young's argument, making an explicit use of the recursion theorem, can be found in [15] (see also [25]). However, the recursion theorem is a quite heavy tool of computability theory, imposing - among other things - to work in a general recursive setting.
The proof presented in this article is based on two key observations:

- we do not really need to work with the actual complexity of $g_u(x)$: any upper bound h to such a complexity will do the job;
- we can abstract the definition of $g_u(x)$ w.r.t. this function h, and instantiate it later according to the complexity of g.

This approach has several advantages. First of all, it makes no use of the fixed point theorem, hence providing information about speedup phenomena in subrecursive settings. For instance, if r is primitive recursive, $g_0^{r\circ(h^r)}$ is too, hence primitive recursive functions do not have, in general, a inherent computational complexity. Up to our knowledge, the only other article addressing a similar problem is [18], where the result is only stated (with no proof) in terms of *space* complexity, referring to the Ph.D. Thesis of Ritchie [21] for the complexity analysis of the Turing machine implementing the speedup function.

Another technical advantage of our proof is that the termination of $g_u(x)$ is not an issue, while it becomes delicate when making use of fix points (in [15], termination takes a good part of the proof).

[3] So, it can appear a bit strange at first glance.

The price to pay is a detailed investigation of the complexity of $g_u^y(x)$; however, this provides interesting information on the complexity of the functions that can be effectively sped-up (in particular, one could still wonder about speedup phenomena *below* the complexity of $g_u^y(x)$). Finally, our development (in contrast with [18]) proves that this complexity analysis can be done at a comfortable level of abstraction, avoiding the need to dig with Turing Machines.

The work presented in this article is part of a large program of formal revisitation of Complexity Theory aiming to a synthetic systematization of the field, particularly oriented to machine verification. Our approach is based on a reverse methodology [3], aiming at reconstructing from proofs of known results in Complexity Theory the basic, abstract notions and assumptions underlying them; the approach can be essentially compared with the work of defining an application programming interface starting from the investigation of the set of services that are supposed to be offered by the application.

The methodology has been already applied to the hierarchy theorems [3], and the gap theorem [2]. The speedup function, discussed in this article, was a particuarly challenging test-bench, due to the complexity of the algorithms involved in the proof.

A minor issue still deserves a more detailed investigation. Using our complexity results for the if-then-else, and in particular the *CF_if* lemma of section 6.8, it is easy to prove that any pair of almost equal functions have the same complexity (and hence that the pseudo-speedup theorem entails the speedup theorem). However, if we start from the weak notion of "almost equal" of section 4.3, there seems to be no way to provide a *constructive* proof of the previous result. On the other side, as we already discussed, working with the traditional notion, we did not see a way to constructively prove the pseudo-speedup theorem. In conclusion, the whole proof of the speed-up theorem, as far as we can see at present, does not appear to be constructive.

Possible extensions of the work presented in this article consist in studying the possibility to decompose the U function in a sequence of more elementary transition steps (along the lines of [13]); this seems an important preliminary step to define the *reachability graph* among configurations, that plays a major role in many important results of Complexity Theory such as the theorems of Savitch [23] and Immerman-Szelepcsényi [17, 24].

Finally, our axiomatic framework must be eventually validated by some concrete computational model. Our syntetic approach naturally complements the more traditional, concrete approach to (formal) computability and complexity, starting from the definition and analysis of specific models of computation, that was recently initiated by many different authors [4, 6, 20, 26].

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. A formal proof of borodin-trakhtenbrot's gap theorem. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2013.

[3] Andrea Asperti. Reverse Complexity. In *Submitted for publication*, 2013.

[4] Andrea Asperti and Wilmer Ricciotti. Formalizing Turing Machines. In *Logic, Language, Information and Computation - 19th International Workshop, WoLLIC 2012, Buenos Aires, Argentina*, volume 7456 of *Lecture Notes in Computer Science*, pages 1–25, 2012.

[5] Andrea Asperti and Wilmer Ricciotti. A proof of Bertrands's postulate. *Journal of Formalized Reasoning*, 5(1):37–57, 2012.

[6] Andrea Asperti and Wilmer Ricciotti. A formalization of Multi-tape Turing Machines. *Theor. Comput. Sci.*, to appear, 2015.

[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] Andrea Asperti, Wilmer Ricciotti, and Claudio Sacerdoti Coen. Matita tutorial. *Journal of Formalized Reasoning*, 7(2):91–199, 2014.

[9] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In *TPHOLs*, pages 86–101, 2008.

[10] Manuel Blum. A Machine-Independent Theory of the Complexity of Recursive Functions. *J. ACM*, 14(2):322–336, 1967.

[11] Manuel Blum. On Effective Procedures for Speeding Up Algorithms. *J. ACM*, 18(2):290–305, 1971.

[12] Allan Borodin. Computational Complexity and the Existence of Complexity Gaps. *J. ACM*, 19(1):158–174, 1972.

[13] Bruno Buchberger. Certain decompositions of Gödel numbering and the semantics of programming languages. In *International Symposium on Theoretical Programming, 1972*, volume 5 of *Lecture Notes in Computer Science*, pages 152–171. Springer, 1974.

[14] Loïc Colson. About primitive recursive algorithms. *Theor. Comput. Sci.*, 83(1):57–69, 1991.

[15] Nigel J. Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.

[16] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[17] Neil Immerman. Nondeterministic Space is Closed Under Complementation. *SIAM J. Comput.*, 17(5):935–938, 1988.

[18] Albert R. Meyer and Patrick C. Fischer. Computational speed-up by effective operators. *J. Symb. Log.*, 37(1):55–68, 1972.

[19] Yiannis N. Moschovakis. On primitive recursive algorithms and the greatest common divisor function. *Theor. Comput. Sci.*, 1-3(301):1–30, 2003.

[20] Michael Norrish. Mechanised Computability Theory. In *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, volume 6898 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011.

[21] D. M. Ritchie. *Program structure and computational complexity*. Ph.D. thesis, Harward University, Division of Engineering and Applied Physics, 1968.

[22] Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press, 1987.

[23] Walter J. Savitch. Relationships Between Nondeterministic and Deterministic Tape Complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970.

[24] Róbert Szelepcsényi. The Method of Forced Enumeration for Nondeterministic Automata. *Acta Inf.*, 26(3):279–284, 1988.

[25] Peter van Emde Boas. Ten years of speedup. In *Mathematical Foundations of Computer Science 1975, 4th Symposium, Mariánské Lázne, Czechoslovakia, September 1-5, 1975, Proceedings*, volume 32 of *Lecture Notes in Computer Science*, pages 13–29. Springer, 1975.

[26] Jian Xu, Xingyuan Zhang, and Christian Urban. Mechanising Turing Machines and Computability Theory in Isabelle/HOL. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013.

[27] Paul R. Young. Easy constructions in complexity theory: gap and speed-up theorems. *Proceedings of A.M.S.*, 37(2):555–563, 1973.