On notation systems for natural numbers and polynomial time computations

Konrad Zdanowski

University of Cardinal Stefan Wyszyński, Warsaw

Numbers and Truth, Gothenburg, 2012

Outline

Outline

Motivations

- When we compute in the Turing model of computations, we do it on finite words.
- Therefore, while talking about computability of certain number theoretic function, we need to fix a notation system.
- A number theoretic function is computable only with respect to a given notation.

Motivations

- Do we have intended notations and how we could distinguished them?
- What is so special about binary or decimal notation?
- How we can construct a notation for a given computational task?

Outline

- I assume a familiarity with basic notions of computability and complexity.
- In particular, I will use a polynomial time class of computations as an explanation for feasibility (Edmond's thesis).
- However, the results are not restricted to this class only. One can take an arbitrary sufficiently robust complexity class.

Definition 1

- *A notation σ is a function from ω to the set of finite words over a fixed finite alphabet.*
- *The word $\sigma(n)$ is a name for n in σ.*
- *By $|u|$ we denote the length of a word u.*

We always require that the image of σ is a computable set.

Example

- We all are familiar with decimal and binary notations.
- A one which is well known is the unary notation: I, II, III, IIII, IIIII, IIIIII, IIIIIII, IIIIIIII, IIIIIIIII, IIIIIIIIII, ...

Definition 2

Two notations σ, τ are recursively (polynomially) isomorphic if there exist recursive (polynomial) functions $F^{\sigma}_{\tau}, F^{\tau}_{\sigma}$ such that for each n

$$F^{\sigma}_{\tau}(\sigma(n)) = \tau(n),$$

$$F^{\tau}_{\sigma}(\tau(n)) = \sigma(n).$$

We will call functions $F^{\sigma}_{\tau}, F^{\tau}_{\sigma}$ as translations.

Example

- The binary and decimal notations are polynomially isomorphic.
- The binary and unary notations are recursively isomorphic but not polynomially.
- A function translating bin(n) into unary(n) would have to write down a word which is of length n so exponential in the length of bin(n) which is $\log(n+1)$.

Definition 3

For a relation $R \subseteq \omega^r$ we define

$$R^\sigma = \{(\sigma(n_1), \ldots, \sigma(n_r)) : (n_1, \ldots, n_r) \in R\}.$$

Definition 4

By σRec we denote the set of functions computable with respect to σ that is

$$\sigma Rec = \{f : \mathbb{N}^k \to \mathbb{N} : \text{the function } f^\sigma \text{ is computable.}\}$$

By σPoly we denote the set polynomially computable functions with respect to σ:

$$\sigma Poly = \{f : \mathbb{N}^k \to \mathbb{N} : f^\sigma \text{ is polynomial time computable.}\}$$

Example

- The usual number theoretic polynomial time is identified with binPoly.
- unaryPoly $=$ binTime$(2^{O(n)})$.
- The class binTime$(2^{O(n)})$ is much bigger than binPoly! Why don't we use the unary notation?

Definition 5

A notation σ is dense if there exists C such that for any $n \in \omega$,

$$|\sigma(n)| \leq \log(n)^C.$$

Usually we require something more:

$$|\sigma(n)| \leq C \log(n).$$

The density property is a form of effectiveness of σ.

- We usually want our notations to be space efficient. If we would need space of size n to store a number n we would quickly ran out of memory.
- We do not want a function to be not efficiently computable just because we cannot store its output (or even inputs) for some small values.

A natural way to compare between notations is to compare between the set of number theoretic functions one can compute using them.

Definition 6

Let $\tau \leq_{Rec} \sigma$ *if* $\tau Rec \subseteq \sigma Rec$.

Definition 7

Let $\tau \leq_{Poly} \sigma$ *if* $\tau Poly \subseteq \sigma Poly$.

Outline

- We defined notations as functions from ω.
- Can we get more restrictions on notations and compare them?
- In reality, we do distinguish between notations, we favour some notations over the other ones.

Outline

Example – unary notation

In some situation the best notation to use is the unary one.

- When we teach children counting we use fingers or apples which may be thought as a kind of unary notations.
- When we are in prison and we count on the wall the days of our stay, we use the unary notation: I, II, III, . . . , II. . .II, . . .

Example – unary notation

Why the unary notation is suitable when being imprisoned?

- We want to compute nothing more than the successor function.
- We do not have costly space limits.
- We can easily carve a stroke on a wall, but it may be hard to erase it. This is why the binary notation may be less efficient in this situation.

Example

- Imagine we want to compute the following set of functions on natural numbers: the exponentiation function 2^a and, for $a = (a_k \dots a_0)_{\text{bin}}$ and $b = (b_k \dots b_0)_{\text{bin}}$,

$$a +_{\text{bit}} b = (\max(a_k, b_k) \dots \max(a_0, b_0))_{\text{bin}},$$

$$a \times_{\text{bit}} b = (\min(a_k, b_k) \dots \min(a_0, b_0))_{\text{bin}}.$$

Example

$$a +_{\text{bit}} b = (\max(a_k, b_k) \dots \max(a_0, b_0))_{\text{bin}},$$
$$a \times_{\text{bit}} b = (\min(a_k, b_k) \dots \min(a_0, b_0))_{\text{bin}}.$$

- We could consider writing $\emptyset$ for 0, $\{a\}$ for 2^a, $a \cup b$ for $a +_{\text{bit}} b$ and $a \cap b$ for $a \times_{\text{bit}} b$. Let me call this notation st (set theoretical).
- We can identify $+_{\text{bit}}$, $\times_{\text{bit}}$ with set theoretic operations by the isomorphism $\text{hf} \colon V_\omega \to \mathbb{N}$:

$$\text{hf}(x) = \sum_{y \in x} 2^{\text{hf}(x)}.$$

Example

- If we would be interested in computing functions: exp, $+_{\mathrm{bit}}$, $\times_{\mathrm{bit}}$ then the intended notation would be the one inherited from set theory.
- What is our intended notation depends on what we want to do with it. What we want to compute.

Example

- What is our intended notation depends also on how we compute.
- Let us recall a residue notation for natural number. A residue notation for a given number n is a sequence of residues of this number modulo some fixed sequence $n_0, n_1, n_2, \ldots$.
- If we have a higly parallel model of computation we may prefer a residue notation.
- In fact such a notation is used in algorithms for fast multiplication.

- I will adapt usual complexity notions when comparing between notations.
- In this situation, we do have costly space limits, so from now on when talking about complexity, I will consider only dense notations ($|\sigma(n)| \leq \log(n)^c$).
- Consequently, let us restrict the ordering $\leq_{\mathsf{Poly}}$ to dense notations only.

Outline

In "Acceptable Notation" (1982), Shapiro considers conditions under which a notation may be called acceptable.

Theorem 8 (Shapiro)

Only finite variations of constant functions or the identity function are computable in any notation.

Theorem 9 (Shapiro)

For any function F the following are equivalent

- *there exists a notation σ such that $F \in \sigma Rec$,*
- *there exists a permutation T such that $T^{-1}FT \in binRec$.*

Theorem 10 (Shapiro)

There exists a function which is not computable in any notation (in fact any function hard for a Σ^0_2 complete set).

So, what notations should be called acceptable?

Theorem 11 (Shapiro)

The following are equivalent:

- $x + 1 \in \sigma Rec$,
- σ *is recursively isomorphic with bin,*
- $\sigma Rec = binRec$.

- It is enough to have a successor function computable in σ to get binRec $\subseteq \sigma$Rec.
- There is no notation σ such that binRec $\subsetneq \sigma$Rec. The notation bin is maximal with respect to recursive relations.
- Let us observe that we can have less. There is a notation σ such that σRec $\subsetneq$ binRec.

Shapiro's principle: An acceptable notation would be any notation σ such that $x + 1 \in \sigma\mathrm{Rec}$.

Pro:
We do want this weak requirement and it already fixes the class of computable functions.
Contra:
Any two notations satisfying the above are recursively isomorphic but they can still differ significantly. The isomorphism between the notation st and bin is computable in exponential time but we would hardly say that st is an intended notation for natural numbers.

Outline

- Now, we will compare notations with respect to computational complexity.
- We assume from now on, that all notations we consider are dense.
- We use $\leq_{\mathsf{Poly}}$ to compare between notations.

- The usefullnes of a notations depends also on our model of computation.
- The notation of residues modulo prime numbers is e.g. convienient for paraller computations.
- I will assume the classical Turing model but we should remember that this is a rough approximations of a real world situation.

Theorem 12

There is no dense notation τ, such that binPoly $\subsetneq \tau$Poly.

Corollary 13

The notation bin is a maximal element among dense notations with respect to $\leq_{Poly}$.

From the proof we get also the following.

Theorem 14

For any dense notation τ, if $\tau Poly = binPoly$ then τ and bin are polynomially isomorphic.

- There is no notation which would allow us to compute more functions in polynomial time. In fact, in some notations we may compute less.
- However, there are notations which allow us to compute in polynomial time different set of functions.
- Any two dense notations σ, τ such that $\sigma\text{Poly} = \tau\text{Poly} = \text{binPoly}$ are polynomially isomorphic.

Outline

Theorem 15

There is no dense notation σ, such that binPoly $\subsetneq \sigma$Poly.

Let σ be a dense notation such that binPoly $\subseteq \sigma$Poly.

Let $x\%2$ be the remainder of x modulo 2 and let $\lfloor x \rfloor$ be the floor function.

Lemma 16

Let $x+1, x2, x\%2, \lfloor \frac{x}{2} \rfloor \in \sigma$Poly. The translations F^{σ}_{bin} and F^{bin}_{σ} are polynomially computable in $\log(n)$ *(so polynomially in $|bin(n)|$).*

We consider only the case of $F^{\sigma}_{\text{bin}}(x)$:

$W \leftarrow \epsilon$
while $x \neq \sigma(0)$ **do**
  if $x\%^{\sigma}2 = \sigma(0)$ **then**
    $W \leftarrow 0^\frown W$
  else
    $W \leftarrow 1^\frown W$
  end if
  $x \leftarrow \lfloor \frac{x}{2} \rfloor^{\sigma}$
end while
return W

The **while** loop is repeated $\log(n)$ many times. Each iteration is polynomial in $\max\{|\sigma(k)| : k \leq n\} \leq \text{poly}(\log(n))$.

- The presented algorithm does not need to be polynomial in $|\sigma(n)|$.
- If $\sigma(n)$ is much shorter than $\log(n)$ then we may have not enough time to iterate the loop $\log(n)$ many times.

Definition 17

A dense notation σ is regular if there exists C such that for all n,

$$|\sigma(n)| \geq \log(n)^{1/C}.$$

Though natural it is, we do not impose this requirement on notations.

Let us observe that the regularity follows from an intuitive property that for any $n \leq k$, $|\sigma(n)| \leq |\sigma(k)|$.

- If σ is regular then the presented translation would be polynomial in $|\sigma(n)|$.
- With the above assumption we can carry out both translations in polynomial time.

Lemma 18

Assume that σ is regular and $x+1, x2, \lfloor \frac{x}{2} \rfloor, x\%2 \in \sigma Poly$. Then $\sigma Poly = binPoly$.

Proof.

Under the above assumptions σ and bin would be polynomially isomorphic. Thus σPoly $=$ binPoly. □

Lemma 18

Assume that σ is regular and $x+1, x2, \lfloor \frac{x}{2} \rfloor, x\%2 \in \sigma Poly$. Then $\sigma Poly = binPoly$.

Proof.

Under the above assumptions σ and bin would be polynomially isomorphic. Thus σPoly $=$ binPoly. □

Now, it is enough to prove that if binPoly $\subseteq \sigma$Poly and σ is dense then it has to be regular.

Lemma 19

Let σ be a dense notation such that binPoly $\subseteq \sigma$Poly. Then σ is regular.

Assume, by contradiction, that for all C there are infinitely many n, such that

$$|\sigma(n)| \leq \log(n)^{1/C}.$$

We will show that this gives a form of an impossible speedup for binPoly.

Hard functions

Let us consider the family of functions $H_K(x)$ with the following properties:

- each $H_K(n)$ is computable in time $\log(n)^{K+1}$ (in binary).
- for each machine M working in time $n^{O(K)}$, there are only finitely many $x \in \mathbb{N}$ such that

$$M(x) = H_K(x).$$

Hard functions

Let $\Gamma(x)$ be a recursive function generating a sequence of all machines $M_0, M_1, \ldots$ working in time $\log(n)^K$.

We can define $H_K(x)$ as follows:

simulate first $\log(x)$ steps of Γ
let $M_0, \ldots, M_r$ be machines generated by Γ
for $i = 0 \to r$ **do**
  compute $y_i = M_i(x)$
end for
return the first y such that $y \notin \{y_0, \ldots, y_r\}$

Speedup

- Let us fix K such that translations between $\sigma(n)$ and $\mathrm{bin}(b)$ can be performed in time $\log(n)^K$.
- By $\mathrm{binPoly} \subseteq \sigma\mathrm{Poly}$, let us take S such that $H_K(x)$ is computable in σ in time $|x|^S$.
- By the assumption that σ is not regular let $a_0, a_1, \ldots$ be such that $|\sigma(a_i)| \leq \log(a_i)^{1/S}$.

Speedup

$H_K(\sigma(n)) \in \mathrm{TIME}(|\sigma(n)|^S), |\sigma(a_i)| \leq \log(a_i)^{1/S}$.

Let us consider the following algorithm computing $H_K(\mathrm{bin}(a))$ in binary:

let $y \leftarrow F_\sigma^{\mathrm{bin}}(\mathrm{bin}(a))$
let $z \leftarrow H_K^\sigma(y)$
return $F_{\mathrm{bin}}^\sigma(z)$

The working time of the above algorithm is

$$\log(a)^K + |F_\sigma^{\mathrm{bin}}(\sigma(a))|^S + \log(a)^K.$$

If $a = a_i$ then we need only

$$\log(a_i)^K + (\log(a_i)^{1/S})^S + \log(a_i)^K \in O(\log(a_i)^K).$$

Contradiction

- By assuming that σ is not regular we constructed an algorithm which computes H_K in binary in time $\log(n)^K$ on infinite sets of inputs a_i, $i \in \omega$.
- Thus, we get a desired contradiction with the hardness of H_K.

There is no better regular notation σ than bin (up to polynomial equivalence).

Theorem 20

If σ regular and $x2, \lfloor \frac{x}{2} \rfloor, x\%2, x+1 \in \sigma Poly$ then $\sigma Poly = binPoly$.

Corollary 21

Assume that a dense notation σ is not regular. Then there exists a function F such that $F \in binPoly$ and $F \notin \sigma Poly$.

Theorem 22

Assume $x2, \lfloor \frac{x}{2} \rfloor, x\%2, x+1 \in \sigma Poly$ *for a dense notation* σ*. If there exists* $F \in \sigma Poly \setminus binPoly$ *then for each* C *for infinitely many* a*,*

$$|F(a)| \geq \log(a)^C.$$

The only reason that F is not in binPoly is that it grows too fast.

Outline

We have many natural and useful notations:

- set theoretical notation st,
- prime powers notation, a number $a = p_0^{a_0} \cdots p_k^{a_k}$ is written as $(\mathrm{bin}(a_0), \ldots, \mathrm{bin}(a_k))$,
- ...

Some functions outside binPoly are computable in constant or linear time in the above notations.

None of these notations is regular so there are functions in binPoly which are not polynomial in the above notations.

Caveat emptor

- What is so special in the class binPoly? It is a robust class but do we need all these functions to be effectively computable on natural numbers?
- We have independent recursion theoretic characterizations of binPoly (Cobham, Bellantoni and Cook) but the recursion schemes heavily depends on the structure of bin notation.

- The above situation holds not only for polynomial time computations.
- One can show such results, e.g., about quasi–linear time, $\mathrm{TIME}(n(\log(n)^{O(1)}))$.

  In this case regularity condition will become stronger: $|\sigma(n)| \geq \varepsilon \log(n)$.

Outline

Outline

- Starting from Cobham we have many algebraic characterizations of ptime functions.
- They are constructed for the binary notation for natural numbers.
- One of the most elegant was given by Bellantoni and Cook.

- All functions $f(\bar{x}; \bar{y}))$ in B.–C. algebra have their variables splited into two sets:
  - safe variables, to the left of the semicolon,
  - normal variables, to the right of the semicolon.
- Ptime recursion is allowed only for safe variables.

The basic functions of B.–C. algebra are:

- a constant 0 function,
- projection functions $\pi^{n,m}(x_1, \ldots, x_n; x_{n+1}, \ldots, x_{n+m}) = x_j$, for $1 \leq j \leq n + m$,
- two successors $s_i(; a) = ai$, for $i \in \{0, 1\}$,
- predecessor $p(; 0) = 0$ and $p(; ai) = a$,
- conditional $c(; x, y, z) = \begin{cases} y & \text{if } x = 0 \mod 2, \\ z & \text{otherwise.} \end{cases}$

New functions are constructed by

- safe composition as $f(\bar{x};\bar{y}) = h(\bar{r}(\bar{x};);\bar{t}(\bar{x};\bar{y}))$, for $h, \bar{r}, \bar{t}$ already defined.
- predicative recursion on notation

$$\begin{aligned} f(0,\bar{x};\bar{z}) &= g(\bar{x};\bar{z}), \\ f(yi,\bar{x};\bar{z}) &= h_i(y,\bar{x};\bar{z},f(y,\bar{x};\bar{z})), \quad \text{for } i \in \{0,1\}. \end{aligned}$$

Outline

- One can define a similar algebra for the set theoretic notation (or so it seems).
- It would give a nice, algebraic characterization of what the polynomial time computation is in the universe of (heredirarily finite) sets.

In defining such recursion schemes one should first be aware of the following

- a given set x may be written down in various forms, e.g., $\{\emptyset, \{\emptyset\}\}$ or $\{\{\emptyset\}, \emptyset\}$,
- there is a need to normalize our notation for sets,
- the normalization should be computable in ptime.

Normalizing notations

- An algorithm normalizing notations should, for a given set x, make a list of all sets in the least transitive closure of x and then order them with respect to numbers they denote,
- From now on we assume that a set is given in a normalized notation $x = \{x_n, \ldots, x_0\}$ with $\mathrm{hf}(x_n) > \cdots > \mathrm{hf}(x_0)$.

An HF–algebra for ptime

Basic functions:

- a constant function $\emptyset$,
- projections $\pi_j^{n,m}(x_1, \ldots, x_n; x_{n+1}, \ldots, x_{n+m}) = x_j$, for $1 \leq j \leq n + m$,
- a weak sum function $s(; x, y) = x \cup \{y\}$,
- a conditional function

$$c(; x, y, z) = \begin{cases} y & \text{if } x = \emptyset, \\ z & \text{otherwise.} \end{cases}$$

An HF–algebra for ptime

We allow

- safe composition,
- predicative recursion:

$$\begin{aligned} f(\emptyset, \bar{x}; \bar{p}) &= g(\bar{x}; \bar{p}), \\ f(y \cup \{y_0\}, \bar{x}; \bar{p}) &= h(y, y_0, \bar{x}; f(y, \bar{x}; \bar{y}), f(y_0, \bar{x}; \bar{p}), \bar{p}), \end{aligned}$$

for functions $g(\bar{x}; \bar{p})$ and $h(z_1, z_2, \bar{x}; w_1, w_2, \bar{p})$.

Some basic functions

For a set $x = \{x_n, \ldots, x_0\}$ we can define its least significant element, $\mathrm{lse}(x) = x_0$, and its most significant element, $\mathrm{mse}(x) = x_n$, as follows:

$$\begin{aligned} \mathrm{lse}(\emptyset;) &= \emptyset, \\ \mathrm{lse}(x \cup \{x_0\};) &= x_0, \end{aligned}$$

and

$$\begin{aligned} \mathrm{mse}(\emptyset;) &= \emptyset, \\ \mathrm{mse}(x \cup \{x_0\}; y) &= c(; x, x_0, \mathrm{mse}(x)). \end{aligned}$$

Then, let $\mathrm{pred}(x) = \{x_n, \ldots, x_1\}$ and $\mathrm{pred}(\emptyset) = \emptyset$.

Some basic functions

If we identify $\bot$ with $\emptyset$ and $\top$ with $\{\emptyset\}$ then we can get equality between two sets with the following scheme:

$$\begin{aligned} =(\emptyset; y) &= c(; y, \emptyset, \{\emptyset\}), \\ =(x \cup \{x_0\}; y) &= \wedge(; = (x; \mathsf{pred}(y)), =(x_0; \mathsf{lse}(; y)). \end{aligned}$$

Similarily we can get an “in” predicate:

$$\begin{aligned} \in(\emptyset; x) &= \emptyset, \\ \in(y \cup \{y_0\}; y) &= \vee(; \in(y; x), =(; y_0, x). \end{aligned}$$

Some basic functions

To get a polynomial rate of growth we can define the cartesian product. Firstly, we define a sum of two sets:

$$\begin{aligned} \cup(\emptyset; y) &= y, \\ \cup(x \cup \{x_0\}; y) &= s(; \cup(x; y), x_0). \end{aligned}$$

Then, we define $\hat{\times}(x; a) = \{a\} \times x$ as

$$\begin{aligned} \hat{\times}(\emptyset; a) &= \emptyset, \\ \hat{\times}(x \cup \{x_0\}; a) &= \cup(; \hat{\times}(x; a), \{\{x_0, a\}, \{x_0\}\}). \end{aligned}$$

Now, we can define a cartesian product $x \times y$ by recursion on x:

$$\begin{aligned} \times(\emptyset; y) &= \emptyset, \\ \times(x \cup \{x_0\}; y) &= \cup(; \times(x; y), \hat{\times}(; y, x_0)). \end{aligned}$$

- Having defined all the above functions, next one may define a "length" function. This may be, e.g.,

$$l(x) = y = \{\ldots\{\emptyset\}\ldots\},$$

  where the number of brackets in x and y are the same.
- The above function may be used as a "clock" function in a simulation of a computation, after we translate the input set x into a representation of a input tape.
- Since we have all the set theoretic objects like finite functions etc., the whole construction may be carried on in a more natural way than in the case of binary notation.

Some questions

- Can we modify this algebra in such a way that it would be well defined for arbitrary sets? A natural candidate for a recursion scheme instead of

$$f(y \cup \{y_0\}, \bar{x}; \bar{p}) = h(y, y_0, \bar{x}; f(y, \bar{x}; \bar{y}), f(y_0, \bar{x}; \bar{p}), \bar{p}),$$

  would be

$$f(y, \bar{x}; \bar{p}) = h(\bar{x}; \{f(z) : z \in y\}, \bar{p}).$$

- Can we extend the algebra for hereditarily finite sets with the power set operation, $\mathcal{P}(x)$? In particular, do we have a notation normalizing algorithm in this case?

Outline

Conclusions I

- Under relatively weak requirement that a notation σ is regular and allows to compute some simple functions ($x2, x+1, \ldots$) we have its polynomial isomorphism with the binary notation.
- We showed for any dense notation that we cannot have more functions effectively computable in σ than in binPoly. (But we may have different functions effectively computable.)

Conclusions I

If we assume

- a notation should be dense,
- a notation should allow us to compute all polynomially computable functions

then we may restrict the class of intended notations to those polynomially isomorphic with bin.

Conclusions II

- Many questions may be posed for notations other than binary.
- In some cases such notations allow to capture different concepts as rudimentary relations, least fixed point logic in finite models for hereditarily finite sets, etc.
- Maybe we should investigate notations in general, e.g., give some lower bounds for computing addition and mulitiplication with respect to any dense notation.

Thank you.