Classes of Algorithms: Formalization and Comparison

Serge Grigorieff
LIAFA, CNRS & Université Paris 7
seg@liafa.jussieu.fr

Pierre Valarcher
LACL, Université Paris-Est
pierre.valarcher@u-pec.fr

May 18, 2012

Abstract

We discuss two questions about algorithmic completeness (in the operational sense). First, how to get a mathematical characterization of the classes of algorithms associated to the diverse computation models? Second, how to define a robust and satisfactory notion of primitive recursive algorithm? We propose solutions based on Gurevich's Abstract State Machines.

Contents

1 Introduction

• From Church Thesis to Gurevich' Sequential Thesis

Since Gurevich introduced Abstract State Machines (aka Evolving Algebras) in the 80's [Gur91, Gur99], there is now a formal mathematical notion of small-step (i.e. non parallel) algorithm working in discrete time.

This answers a question raised by Turing in his celebrated 1936 paper:

> *The real question at issue is "What are the possible processes which can be carried out in computing a [real] number?" [Tur36], page 20*

and Donald Knuth's concern for the question in his famous books [1]:

> *The notion of an algorithm is basic to all of computer programming, so we should begin with a careful analysis of this concept.*
>
> (The Art of Computer Programming, vol. 1, p.1, §1.1 "Algorithms")

As a formalization of an intuitive notion, there can be no formal proof that ASMs (Abstract State Machines) truly formalize what they are intended to. Exactly as there can be no formal proof that the mathematical notion of recursive functions truly formalizes the intuitive notion of computable function.

Nevertheless, as is well-known, all the approaches to the notion of computability considered up to now have been proved to be equivalent to or included in that of recursive function. This carries water for the famous Church Thesis. The same holds with ASMs. The algorithms of all (small-step, discrete time) computation models considered up to now have been proved to be faithfully emulated by ASMs. This has lead to Gurevich' Sequential Thesis [Gur99] which is to algorithms what is Church Thesis to computable functions.

• Classes of algorithms associated to computation models

In §2 we propose a formalization of classes of algorithms associated to computation models in terms of classes of ASMs defined by purely mathematical conditions on the ASM framework (namely, fixing the static background and the vocabulary of the dynamic foreground, cf. detailed explanations in §2) *with no condition on ASM programs except typing constraints.*

This last requirement may seem to be quite a challenge. Indeed, there is a small price to fulfill this requirement: computation models have to be closed under constant speedups. We consider that this last condition as a normalization which removes contingencies while preserving the core ideas of the computation model. Though the reader may not buy that such a closure property is a virtue in itself, we expect him to admit that it is a small price for a truly mathematical characterization.

• Algorithmic gaps in computation models

For a computation model, Turing completeness does not ensure algorithmic completeness: though all computable functions are obtained, maybe some algorithms are missing.

In §3.1 we recall how resource complexity may reveal algorithmic incompleteness. In §3.2, 3.3 we recall Colson's remarkable theorem about the limitations of primitive recursive definitions as algorithms for primitive recursive functions. This result is the motivation for Definition 5.2 in §5.

• **Operational equivalence**

In §4, we discuss operational equivalence of computation models, i.e. equality of the associated classes of algorithms.

First, in §4.2, we recall a recent result [FG10] about the operational completeness of lambda calculus. In §4.3, 4.4, we consider an imperative programming language for algorithms associated to (the inherently functional) Gödel system T.

• **What is a primitive recursive algorithm?**

The algorithmic gaps discovered by Colson (cf. §3.2) show that the obvious answer to that question is totally unsatisfactory. In §5, we recall the solution proposed in [Val08] and describes an imperative programming language $\textsc{Loop}_{\texttt{halt}}$ which is algorithmically complete with respect to the simplest natural class of primitive recursive algorithms Apra.

The rest of the paper is a dialog between the authors and Yuri Gurevich's imaginary student Quisani.

2 Computation models as simple classes of ASMs

2.1 The problem

Q: I heard that you showed that usual sequential-time non-parallel computation models correspond to very simple classes of Abstract State Machines.

A: Yes, this appears in [GV10, GV12a]. Recall that the Gurevich Sequential Thesis [Gur99] ensures that any sequential time small step algorithm is modeled step by step by an ASM.

Q: "Small step algorithm": do you mean that a bounded amount of work is performed at each step?

A: Yes, this is what it means. The simplest way to associate a class of ASMs to a computation model is as follows. Consider all ASMs which model step by step the algorithms associated to the machines or programs of that computation model. Alas, this correspondence is not effective at all: in general, it is undecidable to test if an ASM models a particular algorithm.

Q: A usual phenomenon. Maybe you can take a subclass of that.

A: Yes. In fact, this is what is usually done. For each particular machine or program of the computation model, one devises a particular ASM which models step by step the associated algorithm. This gives a simple and effective class

of ASMs. In particular, since it is effective, this class is necessarily a proper subclass of the previous largest one.

Q: Ok, but the obtained class has no intrinsic ASM characterization.

A: You are right. Indeed, given a computation model $\mathcal{C}$, what we do is to define a class $\mathcal{M}$ of ASMs which has an intrinsic ASM definition and gives exactly the same algorithms as machines or programs in the computation model $\mathcal{C}$.

2.2 Relaxing the time unit

A: But let us start with a word of caution. This approach does not work with the computation models as they are usually considered. We have to generalize them by letting the time unit be chosen arbitrarily.

Q: What does this mean?

A: Consider a Turing machine $\mathcal{M}$. It is usual to consider its instantaneous descriptions at times 0, 1, 2, ... up to time t if $\mathcal{M}$ halts at time t. Instead of that, fix some $k \geq 1$ and consider the instantaneous descriptions at times 0, k, $2k$, ... up to time $\lceil \frac{t}{k} \rceil k$ if $\mathcal{M}$ halts at time t. Looking at $\mathcal{M}$ this way, we get another machine $\mathcal{M}^{(k)}$ which "runs k times faster". The generalization of the model we consider is to add all $\mathcal{M}^{(k)}$'s, $k \geq 1$, for every $\mathcal{M}$.

Q: Relaxing the time unit... Is this generalization so benign? Don't you loose some important feature of a computation model?

A: Choosing a time or space unit carries some arbitrariness. An elementary action or piece of code can always be seen as a family of more atomic ones. On the opposite, some groups of several elementary actions or pieces of code can be considered as what is truly significant.

For instance, a transition of a Turing machine is considered as one computation step. But, you could view it as six computation steps corresponding to different actions of the machine: 1) read the scanned cell, 2) decide how to overwrite its contents, 3) overwrite the scanned cell, 4) decide how to move the head, 5) move the head, 6) change state.

Another example with space rather than time. It is common to say that computers work in base 2. But bits are usually grouped eight by eight into bytes. So computers can also be said to work in base 256.

We view the considered generalization of computation models as a normalization which removes contingencies and preserves the core features of the models.

Q: Hum... What is a contingency? What is a core feature?

A: Some examples will clarify our claim.

2.3 ASMs in a nutshell

A: But, first, let us briefly recall how Gurevich devised ASMs according to his analysis of the constituents of an algorithm. Let us consider only sequential-

time small-step algorithms. Gurevich [Gur99, Gur12] views an algorithm as a transition system and points four basic constituents.

(1) The data structures involved in the algorithm. They constitute the multisort domain of the ASM.

(2) The operations or relations over these data structures which are given for free: the so-called static framework. Observe that these operations and relations are typed. Together, the multisort domain and the associated static framework constitute the background of the computation.

(3) A dynamic vocabulary to name the elements and functions over the data structures which form the foreground of the computation. In the conventional programming, this role is played by the variables. The foreground is dynamic: it may vary at each step of the execution of the algorithm. This dynamic vocabulary is also typed.

A state of the algorithm is the static framework over the multisort domain, augmented with interpretations of the symbols of the dynamic vocabulary.

(4) The fourth ingredient of an algorithm is the program which determines the state change, i.e. the evolution of the dynamic foreground.

The run of an ASM halts if and when either some special value of a dynamic symbol is obtained or the ASM has reached a fixed point, so that two successive states are identical.

Q: So, the algorithm does not tell anything about how the values of static functions are obtained. This departs from the a priori intuition that algorithms tell everything about what they do.

A: Static functions have to be considered as oracles. The algorithm decomposes its global input/output action into a succession of atomic steps. These atomic steps are the limit of the algorithmic process: they use static functions totally ignoring their operationality.

Q: Thus, *algorithms are intrinsically oracular!*

A: Right. This is an essential feature.

For instance, with Turing machines, moving the head, reading the scanned symbol, changing state are static operations given for free. The machine knows how to do that though there is nothing about that in the transition table. However, as anyone can experience, if you want to write a program to emulate a Turing machine in any programming language, there will be lines of code devoted to these operations... As another example, with Blum-Shub-Smale machines [BSS89], one can even check for free if a real number is zero or not. A test known to be non-computable since 1954 (Rice [Ric54]) but given for free in the static framework!

Q: So, there is no absolute notion of algorithm.

A: Well, there is a smallest "effective" notion of algorithm: the one obtained by restricting to computable backgrounds. This means to restrict to computable data structures and to require all static functions to be computable.

Q: Going back to the brief overview of ASMs, what are ASM programs?

A: An ASM program is built with conditionals and updates. Updates are of the form $f(\tau_1, \ldots, \tau_k) := \tau$ where f is a dynamic symbol and $\tau_1, \ldots, \tau_k, \tau$ are any terms. These are in fact ground terms: there is no variable hence all terms are ground. The meaning of such an update is as follows: let $a_1, \ldots, a_k, a$ be the values of the τ_i and τ terms computed at step t; then at step $t+1$, the new value of f at $a_1, \ldots, a_k$ is a.

Q: I remember that when I first learned about ASMs, I was really surprised that there is no explicit loop instruction in ASM programs.

A: It is the run of the ASM, obtained by applying again and again the ASM program, which constitutes kind of a meta loop.

2.4 Intrinsic ASM characterization of computation models

A: Let us go back to computation models and ASMs. Our idea is as follows:

> *Usual computation models are obtained by fixing the three first constituents of ASMs: namely, the data structures, the static framework and the dynamic vocabulary.* As for the fourth constituent, the ASM program, there is no constraint except for the type constraint in the syntactic construction of terms.

Q: Oh! You require the program to be subject to no constraint but typing restrictions. All emulating programs I ever saw code a lot of information about the emulated computation model: they are very particular programs! How is it possible not to discriminate among ASM programs? You are setting yourself such a high bar that it seems impossible to jump over it.

A: This is possible if you relax the time unit.

Q: How does this work for Turing machines?

A: Consider Turing machines with one tape infinite in both directions. There are three data types involved: an infinite set of states Q, an infinite set of letters A and the set $\mathbb{Z}$ of integers to denote the cells on the tape.

Q: Sorry, there are only finitely many states and letters!

A: Yes, but arbitrarily large. So let them be infinite sets. Finiteness will come as a side effect of the ASM program: being a finite object, it will mention only finitely many states and letters.

Q: Nice trick.

A: The static framework for one-tape Turing machines is as follows. Infinitely many nullary function symbols to denote the diverse states and letters and two unary function symbols to denote the successor and predecessor functions on $\mathbb{Z}$. That's all.

Q: Nullary function symbols? You mean constant symbols.

A: Yes. But we shall also deal with dynamic symbols and "dynamic constant" sounds quite awkward. This is why the term "nullary function" is preferred.

Q: So you consider an infinite static vocabulary.

A: Not a big deal. Any ASM program will use only a finite part of this infinite vocabulary.

The dynamic vocabulary consists of three symbols σ, π and γ. The two first ones are nullary function symbols of respective types Q and $\mathbb{Z}$, they are used to denote the current state and position of the head. The last one is a unary function symbol of type $\mathbb{Z} \to A$, it is used to denote the current contents of the tape.

Q: Ok, any Turing machine obviously corresponds to such an ASM. Now, given such an ASM, how do you get a Turing machine?

A: Look at what the program can express. You can update the current state σ to any new value. The position of the head π can also be updated to a new value. But this new value has to be defined by an ASM term of type $\mathbb{Z}$. Look at the static and dynamic vocabulary: there is only one way to get a term of type $\mathbb{Z}$, by applying the successor and predecessor functions to the dynamic constant π. Thus, you can move the head a bounded number of cells, say m, left or right. You can also update the dynamic function γ, henceforth modifying the contents of the current tape. But this can be done at a bounded number of places around the position of the head, say at distance $\leq n$. These bounds m, n depend on the length of terms in the ASM program. Thus, in one step of the ASM you get what is done in $m + 4n$ steps of an appropriate Turing machine.

Q: So this ASM simulates some Turing machine considered with a $m+4n$ times larger time unit. This is why you extend the computation model by relaxing the time unit. By the way, why $4n$?

A: You may have to modify the contents of the n cells right of the head and the n cells left of the head. So, n steps to modify cells on one side of the head, n steps to move back the head and n steps to modify cells on the other side of the head. Finally, at most $n + m$ steps are necessary to move the head to the new scanned cell. With some no move transitions, this last phase requires exactly $n + m$ steps.

2.5 Slight background modification, big consequences

Q: A question. There is no static symbol for the integer 0.

A: No, and this is no accident. If you add a symbol 0 then the update $\pi := 0$ resets the head at cell 0 in one step. Thus, we would model "reset Turing machines". Which is another computation model.

Q: Right. Is it as easy for any computation model as it is for Turing machines?

2.6 ASM reserve and types with Cartesian product ranges

A: For multi-tape Turing machines, for RAMs, for grammars, for automata, etc. it does work with no real difficulty. But there is a problem with Kolmogorov-Uspensky machines and with Schönhage Storage Modification Machines. Their tapes are graphs which can grow.

Q: So you have to appeal to some "reserve set" to pick a new vertex.

A: Here comes the problem. When you pick a new element in the reserve, you have to do two actions: add the picked element to the graph and remove it from the reserve.

Q: Easy to tell in an ASM program. What is the problem?

A: The problem is that this is a constraint on programs: if the program adds a fresh node then it should also remove it from the reserve. But our approach puts no constraint on ASM programs except typing!

Q: How do you get around the problem?

A: We generalize typing. We want two simultaneous actions. In other words, we want to fire two functions at the same time. A solution is to introduce types with Cartesian product ranges. Instead of considering two functions $f : C \to D$ and $g : C \to E$ we consider a function $(f, g) : C \to D \times E$.

Q: Are you kidding? This is the same thing!

A: Not for the way we use it. First, given f and g separately, you can fire f and g on different arguments whereas if you fire (f, g) then you must fire f and g on the same argument. Second, there are far fewer possibilities of composition with a function of type $C \to D \times E$ than with two functions, one of type $C \to D$ and the other one of type $C \to E$.

Let us detail the case of Schönhage machines. The data structures are Q, A for states and letters, an abstract infinite set X for the nodes of the graph-tape (current nodes and "reserve" nodes) and the data structure $\mathcal{P}_{<\omega}(X)$ (the family of finite subsets of X) for the current set of nodes. There are static constants for states and letters and a static function $new : \mathcal{P}_{<\omega}(X) \to X \times \mathcal{P}_{<\omega}(X)$ with Cartesian product range such that, for any $Z \in \mathcal{P}_{<\omega}(X)$, $new(Z)$ is of the form $(a, Z \cup \{a\})$ where $a \notin Z$. The dynamic symbols are σ of type Q for the current state, π of type X for the current position of the head, U of type $\mathcal{P}_{<\omega}(X)$ for the current set of nodes of the graph-tape, γ of type $X \to A$ for the labels of the nodes (with a default value for nodes not in U) and $f_1, \ldots, f_k$ of type $X \to X$ for the partial functions giving the nodes pointed by a given node.

Now, since U is the unique term of type $\mathcal{P}_{<\omega}(X)$, any use of *new* will involve U in an update $(\tau, U) := new(U)$ where τ is either π or a composition of the f_i's applied to π. Such an update simultaneously picks a fresh element from the reserve (which is the difference set $X \setminus U$) and removes it from the reserve, as wanted.

Q: In conventional programming languages like Java, one would not need such function pairs since a new object is created and then is worked upon during the same computation step.

A: Yes, this generalization of typing is done for ASM programming in order to fulfill our ASM characterization of computation models with no constraints on program except for the type constraints in the syntactic construction of terms.

Q: So the price to pay for this ASM characterization of computation models is
1) the extension of computation models by relaxing the time unit,
2) the generalization of typing.

A: Exactly. In our opinion, 1) is forgetting contingencies and 2) is reviving a forgotten (or underrated) feature of functions: coarity, i.e. the number of items constituting the output.

3 Turing completeness vs algorithmic completeness

Q: What are the tools to prove that the step-by-step behaviour of an algorithm is different from that of any algorithm in a given class of algorithms?

A: Tools to reveal operational gaps in computation models and programming languages, even in Turing complete ones?

3.1 Resource complexity theory reveals operational gaps

A: For sure, the simplest tool is computational complexity. For instance, some problems are much harder to solve with one-tape Turing machines than with two-tapes Turing machines. An example is palindrome recognition which requires quadratic time with one tape Turing machine but is obviously solvable in linear time with two tapes.

Q: Yes, with one-tape there is no way but sweep back and forth through the tape to check that a first group of letter coincides with the last one, then a second group of letters next to the first one coincides with the penultimate one and so on. One can do this for groups of one, two or more letters but necessarily uniformly bounded groups. So we get time about $2n+2(n-k)+2(n-2k)+\ldots$, which is $\Omega(n^2)$ indeed.

A: Yet the formal proof that the idea works involves some technique. Crossing sequences in Hennie's original proof [Hen65], or Kolmogorov complexity in Wolfgang J. Paul's very elegant proof [Pau79].

By the way, this result has been extended to multidimensional one-tape Turing machines with a time lower bound $\Omega(n^2/\log(n))$, cf. [Bie03].

Q: Such resource complexity lower bounds appear with a lot of different models. The more powerful the computation model, the more high-performing are the algorithms it implements to solve a fixed problem.

A: Sure. In ASM terms, your observation can be rephrased: with more powerful (static) background and richer dynamic vocabulary, ASM programs simulate step-by-step more sophisticated and high-performing algorithms.

Q: Is there such a substantial difference between the static backgrounds and dynamic vocabularies of one-tape and two-tape Turing machines?

A: Let us see. With two-tape Turing machines, there are two copies of $\mathbb{Z}$, say $\mathbb{Z}^{(\varepsilon)}$, $\varepsilon = 0, 1$, one for each tape, and on each copy, the successor and predecessor operations. This is where lies the difference: one or two copies of the structure $(\mathbb{Z}, x \mapsto x+1, x \mapsto x-1)$. And for each such copy $\mathbb{Z}^{(\varepsilon)}$ there is a symbol π_ε of type $\mathbb{Z}^{(\varepsilon)}$ for the position of the head, and a symbol $\gamma_\varepsilon : \mathbb{Z}^{(\varepsilon)} \to A$ for the current contents of the tape.

Q: Is this redundancy really necessary?

A: Yes! An example, different but similar to the above, illustrates the strength of duplication of structures in the ASM framework. Consider one counter machines. Modeled as ASMs, we have the sort Q for states, the sort $\mathbb{N}$ for contents of the counter, the static structure $(\mathbb{N}, x \mapsto x+1, x \mapsto x-1)$ and two dynamic constants: σ for the current state and γ for the current contents of the counter. Observe that γ has type $\mathbb{N}$. Consider now two-counter machines. The ASM model is analog with two copies $(\mathbb{N}^{(\varepsilon)}, x \mapsto x+1, x \mapsto x-1)$ and $\gamma^{(\varepsilon)}$.

Now, the duplication of $(\mathbb{N}, x \mapsto x+1, x \mapsto x-1)$ and γ witnesses a spectacular classical result: the halting problem is decidable for one-counter machines, but it is undecidable for two-counter machines!

3.2 Operational gaps and primitive recursive functions

Q: By the way, what about intentional behavior of programming languages? A subject initiated by Loïc Colson in the 1990's.

A: The primary goal of his research was to look at the algorithms associated to programs in some programming languages. The question was to find operational properties of such algorithms computing very simple and usual functions.

In particular, in [Col89], L. Colson considers the particular primitive recursive function *min* which computes the minimum of two non negative integers. Very elementary function, isn't it?

Q: Sure. A simple way to compute this function is to decrement x, y in parallel as long as none is 0. This runs in time $O(min(x, y))$.

A: The natural algorithm that you describe is based on a definition of the function *min* by the following equations:

$min(0, y) = 0 \quad , \quad min(x, 0) = 0 \quad , \quad min(x + 1, y + 1) = min(x, y) + 1.$
Though these equations constitute an inductive definition of *min*, this is not a primitive recursive definition since the induction proceeds with two variables simultaneously.

Colson looks at computations which use exclusively the equations involved in a primitive recursive definition of min. He proves that each such computation takes time $O(x)$ or $O(y)$ to compute $\min(x, y)$. When x and y are not of the same magnitude, this does not match the previous $O(min(x, y))$ time algorithm.

Q: What does it mean that a computation uses exclusively the equations involved in a primitive recursive definition?

A: The equations are considered as reduction rules and the computation is a succession of applications of these rewriting rules. Formally, Colson considers PR combinators which formalize primitive recursive definitions of functions.

(i) The basic PR combinators are $\mathtt{0}$, $\mathtt{S}$, $\pi_{k,i}$ (where $k, i \in \mathbb{N}$, $1 \leq i \leq k$). They have respective types $\mathbb{N}$, $\mathbb{N} \to \mathbb{N}$ and $\mathbb{N}^k \to \mathbb{N}$ and represent the integer 0, the successor function over $\mathbb{N}$ and the projections $\mathbb{N}^k \to \mathbb{N}$.

(ii) If c is a PR combinator of type $\mathbb{N}^k \to \mathbb{N}$ and $c_1, \ldots, c_k$ are PR combinators of type $\mathbb{N}^\ell \to \mathbb{N}$ then $\langle c; c_1, \ldots, c_k \rangle$ is a PR combinator of type $\mathbb{N}^\ell \to \mathbb{N}$. If $c, c_1, \ldots, c_k$ represent functions $g : \mathbb{N}^k \to \mathbb{N}$, $h_1, \ldots, h_k : \mathbb{N}^\ell \to \mathbb{N}$ then $\langle c; c_1, \ldots, c_k \rangle$ represents the function $\vec{x} = (x_1, \ldots, x_\ell) \mapsto g(h_1(\vec{x}), \ldots, h_k(\vec{x}))$.

(iii) Given two PR combinators c and d of respective types $\mathbb{N}^k \to \mathbb{N}$ and $\mathbb{N}^{k+2} \to \mathbb{N}$, we get another PR combinator $\mathtt{rec}_{c,d}$ of type $\mathbb{N}^k \to \mathbb{N}$. If c and d represent functions $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$ then $\mathtt{rec}_{c,d}$ represents the function $f : \mathbb{N}^{k+1} \to \mathbb{N}$ such that $f(0, \vec{y}) = g(\vec{y})$ and $f(x + 1, \vec{y}) = h(x, f(x, \vec{y}), \vec{y})$.

Q: Well, this describes PR combinators and their denotational semantics. But, how do you compute with a combinator?

A: To get the operational semantics, we define reduction rules:

(i) $(\pi_{k,i}; t_1, \ldots, t_k) \rightsquigarrow t_i$,

(ii) $\mathtt{rec}_{c,d}(\mathtt{0}, \vec{t}) \rightsquigarrow c(\vec{t})$, $\mathtt{rec}_{c,d}(\mathtt{S}(u), \vec{t}) \rightsquigarrow d(t, \mathtt{rec}_{c,d}(u, \vec{t}), \vec{t})$,

(iii) $c \rightsquigarrow c$ for every c,

(iv) if $u \rightsquigarrow u'$, $t_1 \rightsquigarrow t'_1, \ldots, t_k \rightsquigarrow t'_k$ then $\langle u; t_1, \ldots, t_k \rangle \rightsquigarrow \langle u'; t'_1, \ldots, t'_k \rangle$ and $\mathtt{rec}_{c,d}(u, t_1, \ldots, t_k) \rightsquigarrow \mathtt{rec}_{c,d}(u', t'_1, \ldots, t'_k)$.

To any combinator c representing a function $f : \mathbb{N}^k \to \mathbb{N}$, we associate a nondeterministic algorithm which works as follows: on input $(a_1, \ldots, a_k) \in \mathbb{N}^k$, iteratively apply reduction rules to the PR combinator $\langle c; S^{(a_1)}(\mathtt{0}), \ldots, S^{(a_k)}(\mathtt{0}) \rangle$ (which has type $\mathbb{N}$) until no reduction rule applies. It is routine to check that we then get a PR combinator of the form $\mathtt{S}^{(n)}(\mathtt{0})$ where $n = f(a_1, \ldots, a_k)$.

Q: So, a confluence property holds for this calculus.

A: Yes. Now, a given PR combinator may contain many redexes, i.e. , there may be many sub-combinators to which a reduction can be applied. Thus,

there are many computations to get the value of the function f represented by the PR combinator c on an input $(x_1, \ldots, x_k) \in \mathbb{N}^k$. In particular, there are computations corresponding to a call-by-name or a call-by-value strategy or any mixing, possibly a random one, of these strategies. What Colson proves is that every one of these computations takes time $O(x_1)$ or $\ldots$ or $O(x_k)$.

3.3 Denotational semantics reveals operational gaps

Q: Could you recall the argument in Colson's proof?

A: Colson uses a non-standard denotational semantics, namely *lazy natural integers*. Formally, this model contains two kinds of objects: for every $n \in \mathbb{N}$, there is the integer $S^{(n)}(0)$ and the "in progress" integer $S^{(n)}(\bot)$ which stands for "being $\geq n$ ". There is also a limit element $S^{(\infty)}(\bot)$. The ordering relations are reduced to $S^{(n)}(\bot) \leq S^{(n+k)}(\bot) \leq S^{(\infty)}(\bot)$ and $S^{(n)}(\bot) \leq S^{(n+k)}(0)$.

Q: Yes, I remember. Colson processes the PR combinators on such lazy integers. And the core argument was an "ultimate obstinacy" result: on the pair $(S^{(\infty)}(\bot), S^{(\infty)}(\bot))$, the computation of *min* gets stuck, decrementing the same argument again and again. This prevents any alternation between arguments.

A: As it is, the proof can be viewed as non-constructive since it involves an infinite object $S^{(\infty)}(\bot)$. A constructive variant is given in [Coq92].

Q: On lazy integers, what are the functions computed by PR combinators?

A: Looking on the sole true integers $S^{(n)}(0)$, PR combinators obviously compute all primitive recursive functions. Now, on the $S^{(n)}(\bot)$'s, they give exactly the subclass of primitive recursive functions obtained with definitions by inductions with null base case [Val08], i.e. inductions of the form

$$\begin{cases} f(\bot, S^{(y_1)}(\bot), \ldots) &= \bot \\ f(S^{(x+1)}(\bot), S^{(y_1)}(\bot), \ldots) &= h(S^{(x)}(\bot), f(S^{(x)}(\bot), S^{(y_1)}(\bot), \ldots), S^{(y_1)}(\bot), \ldots) \end{cases}.$$

Q: I see. The base case involves $\bot$, i.e. $S^{(0)}(\bot)$, as the induction input. Since $\bot$ means no information at all, when the induction input has value $\bot$, it is reasonable that the function gets value $\bot$.

How far have such arguments been pushed?

A: If the data is enriched with lists, the argument breaks down. In fact, René David [Dav93] proves that the best algorithm for the *min* function can be obtained from a primitive recursive definition of *min* involving recursion over lists. Going to second-order primitive recursive definitions, i.e. working at level 1 of Gödel system T, one can also obtain the best algorithm for the *min* function, cf. [Col89]. Let us stress that this requires a call-by-name strategy. On the opposite side, [CF98] proves that call-by-value strategies for higher order primitive recursive definitions of *min*, (i.e. working at any level of Gödel system T) do not allow to get the best algorithm for *min*.

Q: Could you remind me of system T ?

A: Instead of considering primitive recursive definitions for the sole functions $\mathbb{N}^k \to \mathbb{N}$, extend such definitions to functionals. Formally, you consider a typed lambda calculus with types obtained from the basic type *nat* via the type constructor $A, B \mapsto (A \to B)$. Augment this calculus with

(1) constants $\mathtt{0}$, $\mathtt{S}$ and $\mathtt{rec}_{\alpha,\beta}$ (for integer zero, the successor function over $\mathbb{N}$ and the recursion operator defining functions of type $\alpha \to \beta$).

(2) Besides the usual beta reduction, consider new reduction rules for each $\mathtt{rec}_{\alpha,\beta}$ such that, for any terms t, u, n with respective types $\alpha \to \beta$, $\mathbb{N} \to ((\alpha \to \beta) \to (\alpha \to \beta))$ and $\mathbb{N}$,
$$\mathtt{rec}_{\alpha,\beta}\ t\ u\ 0 \rightsquigarrow t \quad , \quad \mathtt{rec}_{\alpha,\beta}\ t\ u\ (sn) \rightsquigarrow u\ n\ (\mathtt{rec}_{\alpha,\beta} t u n)$$

With the usual notations for functions, the above reductions correspond to a higher order definition by induction: $H(0) = t\ ,\ H(n+1) = u(n, H(n))$ where H has type $\mathbb{N} \to (\alpha \to \beta)$.

Q: Please, forget the recursion combinators... What about an example expressed with higher order primitive recursive definitions of functionals?

A: Sure. Let us show how the best algorithm for *min* is second-order primitive recursive. Consider the following second-order primitive recursive definitions of functionals $G : \mathbb{N}^{\mathbb{N}} \times \mathbb{N} \to \mathbb{N}$ and $F : \mathbb{N} \to \mathbb{N}^{\mathbb{N}}$:

$$\left\{ \begin{array}{lcl} G(f,0) & = & 0 \\ G(f,y+1) & = & f(y)+1 \end{array} \right. , \quad \left\{ \begin{array}{lcl} F(0) & = & \lambda y\ .\ 0 \\ F(x+1) & = & \lambda y\ .\ G(F(x),y) \end{array} \right.$$

These equations yield $\left\{ \begin{array}{lcl} F(0)(y) & = & 0 \\ F(x+1)(0) & = & G(F(x),0) = 0 \\ F(x+1)(y+1) & = & G(F(x),y+1) = F(x)(y)+1 \end{array} \right.$.

Letting $\varphi(x,y) = F(x)(y)$, we have $\left\{ \begin{array}{lcl} \varphi(0,y) & = & 0 \\ \varphi(x+1,0) & = & 0 \\ \varphi(x+1,y+1) & = & \varphi(x,y)+1 \end{array} \right.$.

These are exactly the equations for the best algorithm for the *min* function!

Q: Very nice definition. And beyond the *min* function?

A: Recall Stein's algorithm for the greatest common divisor of two integers. To take full benefit of the binary representation of integers in computers, Stein considers the following recursive equations for the gcd:

$$\begin{array}{lcll} \gcd(x,y) & = & 0 & \textit{if}\ \ min(x,y) = 0 \\ \gcd(2x,2y) & = & 2\gcd(x,y) & \textit{if}\ \ min(x,y) > 0 \\ \gcd(2x,2y+1) & = & \gcd(x,2y+1) & \textit{if}\ \ x > 0 \\ \gcd(2x+1,2y) & = & \gcd(2x+1,y) & \textit{if}\ \ y > 0 \\ \gcd(2x+1,2y+1) & = & \gcd(|x-y|, \min(2x+1,2y+1)) & \end{array}$$

Stein's algorithm has time complexity $O(\log(x)+\log(y))$. Moschovakis [Mos03] proves that this complexity cannot be matched by any algorithm based on primitive recursive definitions of the gcd and using a call-by-value strategy. In fact,

he gets an $O(x+y)$ lower bound for call-by-value computations even if addition, subtraction, division by 2, parity and order comparison are basic functions in such primitive recursive definitions.

4 Operational equivalence of classes of algorithms

4.1 A new problem

A: Besides all these new complexity results, a new problem emerged around the behavior of programs and especially functional programs. Instead of looking for missing algorithms in some computation model, we try to compare the families of algorithms associated to programming languages. Are they equal? Is one of them larger than the other? In this way, one can compare the operational strength of programming languages.

4.2 Lambda calculus and ASMs

Q: By the way, I remember Yuri told me that Kolmogorov-Uspensky machines and Schönhage Storage Modification Machines enriched with a pairing function encode all ASMs.

A: Yes. Such a pairing function allows to encode any data structure on the set of nodes, so that by adding a static framework on the set of nodes, you get any ASM static framework. And any dynamic vocabulary can be encoded by the dynamic successors of nodes in these models. You could also consider hyper-machines à la Kolmogorov-Uspensky or à la Schönhage: replace graph tapes by k-uniform hypergraph tapes for some $k \geq 3$.

Q: What is a k-uniform hypergraph?

A: An undirected graph can be seen as a pair (V,E) where V is a set of nodes and E is a collection of node sets of cardinality 2. A k-uniform hypergraph is a generalization where E is a collection of node sets of cardinality k.

Q: So we get two computation models which are operationally equivalent to ASMs. Well, adding a pairing function on the set of nodes seems to be an ad hoc device which was never considered seriously. Is there any other computation model which gives all sequential time small step algorithms?

A: Yes, lambda calculus. This is done in [FG10].

Q: How is that possible? Beta reduction is such a low level operation! Some consider it as an instruction on the level of an assembly language. Moreover, though all countable data structures can be encoded in lambda calculus, such encodings are time consuming: an unbounded number of beta reductions are necessary to mimic any basic operation in data structures.

A: Concerning your first objection, it is true that beta reduction is a very crude operation. But remember that the time unit is not really an intrinsic notion.

In fact, the simulation of one step of an ASM corresponds to a succession of k beta reductions in lambda calculus for some fixed k which depends only on the simulated ASM.

As for the second objection, there is a fairness argument: the static framework is for free in ASMs, hence it should also be for free in lambda calculus. This leads us to add constants in lambda calculus to represent elements of the data structures and the diverse static operations. And also, of course, to add a new kind of reductions. For instance, with the $\mathbb{N}$ data structure, if constants a, b, c represent integers m, n, p and constant d represents a function $f : \mathbb{N}^2 \to \mathbb{N}$ such that $f(m, n) = p$ then there is a reduction $d(a, b) \rightsquigarrow c$.

Q: Adding constants in lambda calculus is a much studied topic.

A: Yes, but what is done in [FG10] is very peculiar. There is no variable involved in the new reductions, only ground terms with no lambda.

Q: Do you count these new reductions in your simulation?

A: As you wish. If you want to count them, this is no problem. There is a fixed ℓ (depending on the simulated ASM) such that one step of the ASM run corresponds to exactly k beta reductions plus ℓ "new" reductions.

Q: So lambda calculus is operationally equivalent to ASMs.

A: No. In lambda calculus, a beta reduction is not always a small step action since the number of redexes in a lambda term can be arbitrarily large! Beta reduction is a parallel action which goes beyond small step algorithms. Thus, the operational strength of lambda calculus lies somewhere between that of small step ASMs and that of parallel ASMs.

Q: When you simulate an ASM step by a fixed amount of successive beta reductions and "new" reductions, do these beta reductions have small step action or do they reduce arbitrarily large number of redexes simultaneously?

A: They reduce a bounded number of redexes. So they do have small step action. No problem, the simulation is fair.

4.3 The imperative language LOOP$^\omega$

A: Non-Turing complete programming languages can also be compared relative to their operational strength. The first such result [CLV06] is relative to the LOOP language of Ritchie and Meyer [MR76] and to T_0, the level 0 of system T (which is equivalent to classical primitive recursion, extensionally as well as intentionally [Col91]). LOOP and T_0 have the same operational strength. One step in any one of these languages can be simulated by one step in the other language.

Q: How do you prove that?

A: Well, this result has been extended in [CPV09] to the whole system T and to an extension LOOP$^\omega$ of the programming language LOOP with first-class procedures (true closures) and mutable procedural variables (aka function pointers). We shall rather present this extended result.

Q: So you consider an imperative language LOOP$^\omega$. That is exciting. Is it a usual one?

A: LOOP$^\omega$ can easily be written using C# syntax (where anonymous first-class procedures are called delegates [ISO, 2003]) and then compiled with a C# compiler.

In LOOP$^\omega$, instructions are as follows: block of commands, bounded loop, assignment, incrementation, decrementation and procedure call. In Backus-Naur notation, this can be written

$$\begin{array}{rrcl} (\textit{command}) & c & ::= & \{s\} \\ & & | & \texttt{for}\ y\ := 1\ \texttt{to}\ e\ \{s\} \\ & & | & y := e \mid \texttt{inc}(y) \mid \texttt{dec}(y) \\ & & | & p(\vec{e};\vec{y}) \end{array}$$

In a sequence, one can declare mutable variables and constant variables:

$$\begin{array}{rrcl} (\textit{sequence}) & s & ::= & \epsilon \\ & & | & c;\ s \\ & & | & \texttt{cst}\ y\ =\ e;\ s \\ & & | & \texttt{var}\ y\ :=\ e;\ s \end{array}$$

In the declaration of a procedure, arguments are of two kinds : readable only (`in`) and writable only (`out`). This distinction is similar to ADA syntax and semantics:

$$(\textit{anonymous procedure}) \quad a \quad ::= \quad \texttt{proc}(\texttt{in}\ \vec{y};\texttt{out}\ \vec{z})\ \{s\}$$

Q: Looks like usual imperative programming languages. Do you have some examples of programs?

A: Sure. Here is an a LOOP$^\omega$ program that computes the Ackermann function.

```
proc (in m: int, n: int; out r: int) {                          1
    proc next(in y: int, n: int; out p: int) {                  2
        p:=y;                                                   3
        inc(p);                                                 4
    }p                                                          5
    var g: proc (in int; out int);                              6
    g:=next;                                                    7
    for i:= 1 to m {                                            8
        cst h = g;                                              9
        proc aux(in y: int; out p: int) {                       10
            h(1,p);                                             11
            for j:= 1 to y {                                    12
                h(p; p);                                        13
            }p;                                                 14
        }p;                                                     15
        g:=aux;                                                 16
    }p;                                                         17
    g(n; r);                                                    18
}r;                                                             19
```

Q: A few comments would be welcome...

A: Recall Ackermann's equations:

$$\begin{cases} Ack(0,n) &= n+1 \\ Ack(m+1,0) &= Ack(m,1) \\ Ack(m+1,n+1) &= Ack(m,Ack(m+1,n)) \end{cases} .$$

Introduce $A(x) = \lambda y.Ack(x,y)$ and observe that $A(m)$ is defined by induction on m in such a way that the induction step defines $A(m+1) = \lambda n.Ack(m+1,n)$ by induction on n using function $A(m)$.

All procedures in the program are functional so that we shall write $g(x) = y$ instead of $g(x,y)$ and the same with h and *aux*. Procedure *next* in lines 2-5 is the successor function. In case $m = 0$, the loop in lines 8-17 vanishes. Hence the output is $g(n;r)$ (line 18), i.e. the successor function (line 7). Thus, we get the basic case of the induction giving $A(0)$. For $m > 0$, we argue by induction on m and assume that the loop in lines 8-17 makes g be the function $A(m)$ after m iterations of the loop. Then an $m+1$-th iteration defines a function *aux* which takes value $A(m)(1) = Ack(m,1)$ on 0 (line 11). Which is the base case of the induction on n to get $Ack(m+1,n)$. For $n > 0$, we argue by induction and assume that the loop in lines 12-14 makes *aux* be the function $\lambda n.Ack(m+1,n)$ after n iterations of the loop. So that the value of p is $aux(n) = Ack(m+1,n)$. Then an $n+1$-th iteration makes $aux(n+1)$ equal to $A(m)(Ack(m+1,n)) = Ack(m,Ack(m+1,n)) = Ack(m+1,n+1)$ (cf. line 13).

Q: Of course, you can similarly program the best time algorithm for the *min* function in LOOP$^\omega$. Just follow Colson's definition of *min* at level 1 of system T.

A: No, no. In T, you get the best algorithm for *min* with the call-by-name strategy. And Loop$^\omega$ only simulates the call-by-value strategy.

Let us mention that, as far as we know, Loop$^\omega$ is the first total imperative language in which the Ackermann function can be programmed.

Back to the definition of Loop$^\omega$. Of course, some constraints are imposed by the type system [CPV09] and allow us to give a simple operational semantics in terms of transition systems which we prove to be equivalent to the natural semantics. Also, every variable has a default value which depends only on its type.

Another operational semantics is given through the translation of a program of Loop$^\omega$ into a term of system T. This translation preserves the operational semantics. This is how we obtain a step-by-step simulation.

4.4 Translating Loop$^\omega$ programs into system T terms

First, let us write $\texttt{let}\ (x_1,\dots,x_n) = u\ \texttt{in}\ t$ as an abbreviation for the redex $\lambda(x_1,\dots,x_n).t\ u$. The intuition behind the translation is as follows: if $\vec{x}$ denotes the variables of the environment then the block $\{c_1;\dots;c_n\}$ is translated into

$$\texttt{let}\ \vec{x} = c_1^*\ \texttt{in}\dots\ \texttt{let}\ \vec{x} = c_n^*\ \texttt{in}\ \vec{x}$$

where the * operation is defined below.

We may be more precise, cf. [CPV09], and annotate blocks by the sets of variables that get modified. For instance, the block $\{\texttt{inc}(x);\texttt{inc}(x)\}$ is annotated $\{\texttt{inc}(x);\texttt{inc}(x)\}_x$.

The translation is defined as follows.

Definition 4.1. *The stars e^* and $\{s\}^*_{\vec{x}}$ of an expression e and a block $\{s\}_{\vec{x}}$ in a term of system T are defined by mutual induction:*

$\bar{n}^*$	*is*	$S^n(0)$
y^*	*is*	y
$\{\}^*_{\vec{x}}$	*is*	$\vec{x}$
$(\texttt{proc}(\texttt{in}\ \vec{y};\texttt{out}\ \vec{z})\{s\}_{\vec{z}})^*$	*is*	$\lambda\vec{y}.\{s\}^*_{\vec{z}}[\vec{z}_0/\vec{z}]$
where $\vec{z}_0$ denotes the default value for each type of z variables		
$\{\texttt{var}\ y := e;\ s\}^*_{\vec{x}}$	*is*	$\{s\}^*_{\vec{x}}[e^*/y]$
$\{\texttt{cst}\ y = e;\ s\}^*_{\vec{x}}$	*is*	$\texttt{let}\ y\ =\ e^*\ \texttt{in}\ \{s\}^*_{\vec{x}}$
$\{y := e;\ s\}^*_{\vec{x}}$	*is*	$\texttt{let}\ y\ =\ e^*\ \texttt{in}\ \{s\}^*_{\vec{x}}$
$\{\texttt{inc}(y);\ s\}^*_{\vec{x}}$	*is*	$\texttt{let}\ y\ =\ \texttt{succ}(y)\ \texttt{in}\ \{s\}^*_{\vec{x}}$
$\{\texttt{dec}(y);\ s\}^*_{\vec{x}}$	*is*	$\texttt{let}\ y\ =\ \texttt{pred}(y)\ \texttt{in}\ \{s\}^*_{\vec{x}}$
$\{p(\vec{e};\vec{z});\ s\}^*_{\vec{x}}$	*is*	$\texttt{let}\ \vec{z}\ =\ p^*\ \vec{e}^*\ \texttt{in}\ \{s\}^*_{\vec{x}}$
$\{\{s_1\}_{\vec{z}};\ s_2\}^*_{\vec{x}}$	*is*	$\texttt{let}\ \vec{z}\ =\ \{s_1\}^*_{\vec{z}}\ \texttt{in}\ \{s_2\}^*_{\vec{x}}$
$\{\texttt{for}\ y :=\ 1\ \texttt{to}\ e\ \{s_1\}_{\vec{z}};\ s_2\}^*_{\vec{x}}$	*is*	$\texttt{let}\ \vec{z}\ =\ \texttt{rec}(e^*,\vec{z},[\vec{z},y]\{s_1\}^*_{\vec{z}})$ $\texttt{in}\ \{s_2\}^*_{\vec{x}}$

Observe that in the imperative program some instructions cost one computation step but other ones, namely, declaring a variable and managing a loop

counter, do not cost any computation step. The instructions that do cost are mapped into `let` expressions.

The following theorem ensures that the translation gives a step by step simulation.

Theorem 4.2. *For any well-typed state (c, μ) (where c is a program and μ is the environment), if $\vec{x} = dom(\mu)$ then*

$$(c, \mu) \to (c', \mu') \quad \textit{implies} \quad c^*[\mu(\vec{x})^*] \rightsquigarrow c'^*[\mu'(\vec{x})^*]$$

where $\to$ denotes one step reduction of the Loop$^\omega$ *language and $\rightsquigarrow$ denotes one step reduction in the system T language.*

Q: Does your imperative language capture all functions of system T ?

A: Yes, this is the first result. Let us recall a major result concerning system T : functions over $\mathbb{N}$ that are definable in this system correspond exactly to functions that are provably total in first-order Peano arithmetic (see Schütte's book [Sch77]). More precisely, consider the syntactic hierarchy of fragments T_n of system T associated to type order where $o(nat) = 0$ and $o(A \to B) = \max(o(B), 1 + o(A))$. Then the class of functions representable in T_n is identical to the class of functions provably recursive in the fragment of Peano arithmetic where induction is restricted to Σ_{n+1} sentences. In particular, T_0 corresponds to the class of primitive recursive functions. We define a similar hierarchy of fragments Loop$_n$ of Loop$^\omega$ and we show that both translations relate programs of Loop$_n$ and terms of T_n. The particular case $n = 0$ shows that functions representable in a language with higher-order procedures but without procedural variables (which is a sublanguage of Loop$_0$) are primitive recursive. This corollary generalizes a previous result presented in [CLV06] where Meyer and Ritchie's Loop language was translated into T_0.

Now, the fundamental result of [CPV09] is that all properties about complexity of programs in system T with call-by-value are retrieved in our Loop$^\omega$ language, so *they are algorithmically equivalent.*

Q: Of course, call-by-value is sine qua non.

A: Sure. Recall that the best algorithm for *min* can be obtained in system T with call-by-name.

Q: So, this also corroborates results about system T such as the fact that the best algorithm for *min* is not captured by Loop$^\omega$ (which was proved in [CF98] for system T with call-by-value).

Well, the striking point is that the two languages you describe, namely Loop$^\omega$ and system T with call-by-value not only compute the same class of functions but also the same class of algorithms.

5 What is a primitive recursive algorithm?

5.1 Primitive recursive running time

A: Let us consider the class of primitive recursive functions and the question "What is a primitive recursive algorithm?".

Q: There is an answer with Turbo ASMs [BS03].

A: The Turbo approach to define primitive recursive ASMs constrains the syntax of ASM programs to be a clone of primitive recursive definitions. And this constraint has far reaching consequences. In fact, what Colson proves can be transferred to this approach. In particular, the best algorithm for the *min* function is missing. Now, *we want these missing algorithms to be considered as primitive recursive algorithms.*

The solution proposed in [Val08] is based on a classical theorem about primitive recursive functions:

Theorem 5.1. *A function f is primitive recursive if and only if there exists a Turing machine which computes f in time bounded by a primitive recursive function.*

Q: One of my teachers used to call it an analog of Lebesgue's dominated convergence theorem...

A: This theorem ensures the equivalence of two conditions. The first one is about the function itself whereas the second one is about how to compute that function hence about algorithms.

Q: Turing machines are not a very good model for algorithms. I guess your next step is to replace them by Abstract State Machines.

A: Yes, you guessed right! The above statement which is a theorem involving Turing machines will now become a definition involving ASMs.

Definition 5.2. *An algorithm is said to be primitive recursive if and only if it corresponds to an ASM which halts in primitive recursive time.*

It's very simple! Based on the identification of algorithms to ASMs. And it captures the best algorithm for the *min* function.

Q: I presume that the output of an ASM run is the value of some dynamic symbol α when the ASM halts. Let $t(input)$ be the running time. Do you require the ASM to halt exactly at time $t(input)$ or do you consider the value of α at time $t(input)$? In the first case, the ASM halts "naturally" whereas in the second case you just ignore subsequent states.

A: There is no difference: just add $t(input)$ as a static function! A simple counter will ensure halting in due time. This is easy to be done in an ASM program. Nevertheless, we shall prefer the approach which ignores subsequent states. The reason is that we do not want to spoil the clarity of ASM programs by the management of a counter to get halting at the proper time.

Q: I see a problem with Definition 5.2. Every function f admits a primitive recursive algorithm: just put f itself in the background of the ASM! Moreover, such an ASM computes any value of f in exactly one step.

A: Right. There are trivial algorithms which carry no operational contents. This is the price to pay to the oracular nature of algorithms: oracles are what is in the background. You just considered the trivial case where the oracle is the function itself.

Now, there is more to answer your observation. Taking the background into consideration, we can define a notion of primitive recursive algorithm relative to a fixed background.

Q: Well, well. So, the most natural way of defining primitive recursive functions is inadequate to capture a reasonable notion of primitive recursive algorithm. You base the notion of primitive recursive algorithm on primitive recursive bounds on the length of runs of ASMs.

A: Runs and length of runs are at the root of the formalization given by ASMs: the first ASM postulate is that algorithms are discrete time transition systems!

Q: Which makes your approach an intrinsic one.

5.2 Basic arithmetical primitive recursive algorithms

Q: So, you have defined from scratch a class of algorithms to compute primitive recursive functions (cf. Definition 5.2). This class is large enough to contain all known natural algorithms to get these functions using a reasonably simple background. You consider this class as the natural candidate for the desired notion of primitive recursive algorithm. Now, the definition makes no reference to any programming language. An interesting question then arises: find a programming language that implements this class of algorithms.

A: In this approach to primitive recursive algorithms, we are led to consider a computation of an algorithm as a triple $(\mathcal{A}, I, c)$ where $\mathcal{A}$ is an abstract state machine, I is an initial state and c is a function that represents the length of the part of the ASM run that we consider (ignoring all subsequent states).

Q: Oh! I heard about something like that thirty years ago, a program is equivalent to the pair of a sequential function and a computation strategy for it. As I remember this was called *sequential algorithms on concrete data structures* (cf. [BC82]). I think it's not sufficient to consider similar pairs. You have to require that static functions and initial interpretations of dynamic symbols be primitive recursive. Else, you would capture non-primitive recursive functions! Wouldn't you?

A: Sure. Now, there is another point to consider. To focus on programming languages, we have to compare a theoretical set of algorithms with the set of algorithms of a programming language. So, in fact, we have to look at the expressive power (in term of algorithms) of control structures in programming languages.

Q: On that matter, no doubt that ASMs have the smallest set of control structures: the conditional and nothing else.

A: Though there is no loop instruction in ASMs, it is no problem to simulate the successive steps of a loop by the successive steps of the ASM run. In fact, the ASM run is a big loop which somehow externalizes all loop instructions. Indeed, this externalization is one of the main ideas of ASMs.

Q: Simultaneous updates are allowed with ASMs, a rather uncommon feature in imperative programming languages.

A: There are two ways to cope with simultaneous updates. First, by simply allowing them in the programming language. But, as you said, this would badly depart from "real" programming languages. So we exclude simultaneous instructions (or evaluations) in our programming language and keep, as is usual, completely sequentialized computations. Now, the number of simultaneous updates is bounded. Though we cannot simulate step-by-step, we can simulate one ASM step by a fixed number of computation steps of the program. Again, we appeal to a flexible time unit.

Q: I see still another problem related to the ASM background. To balance the poorness of the control structure, ASMs allow powerful data structures: first order logical data structures. And this is not allowed in most programming languages. Of course, programming languages can emulate such data structures but there is a price for the emulation!

A: You are right. In order to stay close to real programming languages, we restrict the family of data structures of ASMs to the ones usually allowed in programming languages. This leads to the following definitions.

Definition 5.3. *An ASM is* $\mathbb{N}$*-typed if its multisort domain is reduced to* $\{0,1\}$ *and* $\mathbb{N}$*, that is Booleans and integers.*

Now, the (static) background of an $\mathbb{N}$-typed ASM consists of some Boolean and integer elements and some functions with Boolean and integer inputs and outputs. And its dynamic vocabulary contains constant (i.e. arity zero) and function symbols to represent such elements and functions.

This is still too much to match any existing programming language. First, as mentioned in §5.1, we shall fix a reasonably simple background on data structures $\{0,1\}$ and $\mathbb{N}$. We shall also introduce another constraint on the dynamic vocabulary: no function symbol of arity ≥ 1.

Definition 5.4. *An* $\mathbb{N}$*-typed ASM A is called* basic arithmetical *if*

i. its static framework is reduced to constants, Boolean operations on $\{0,1\}$ *and the predecessor and successor functions on* $\mathbb{N}$*,*
ii. and its dynamic vocabulary contains only nullary symbols.

Q: This is a very restricted family of ASMs.

A: Since static functions in an ASM are evaluated for free, as concerns resource complexity, the restriction to basic arithmetical ASMs may be seen as a reasonable choice. But, first, let us give a basic arithmetical ASM program for the best algorithm to compute $min(m, n)$:

$$\left|\begin{array}{llll} \texttt{if} & x = 0 & \texttt{then} & res := n \\ \texttt{if} & x \neq 0 \wedge y = 0 & \texttt{then} & res := m \\ \texttt{if} & x \neq 0 \wedge y \neq 0 & \texttt{then} & \left|\begin{array}{l} x := x - 1 \\ y := y - 1 \end{array}\right. \end{array}\right.$$

Q: So there are four dynamic constants: m, n contain the inputs and are not modified, x, y take values m, n in the initial state and are the core of the computation.

A: Yes.

Q: Basic arithmetical ASMs seem to be very close to the Loop language.

A: It is true that a single computation step of basic arithmetical ASMs is a very rudimentary action. Now, the run of the ASM is an external loop which is an unbounded loop, not a bounded one like those in the Loop language. This makes basic arithmetical ASMs far more powerful than Loop. In fact, extensionally, basic arithmetical ASMs are Turing complete!

Q: How do you see that?

A: Simulate any two counter machine $\mathcal{C}$. Consider static constants of type $\mathbb{N}$ to encode the finitely many states of $\mathcal{C}$. Use three dynamic constants of type $\mathbb{N}$ to encode the state and the contents of the counters.

Q: It seems that basic arithmetical ASMs are exactly multicounter machines.

A: No. Counter machines are not able to compare the contents of two counters in a single computation step. Such a comparison requires a computation involving parallel decrementations very similar to the best computation of the *min* function. On the opposite, an equality test between two dynamic constants is an elementary instruction for basic arithmetical ASMs. So, basic arithmetical ASMs are operationally more powerful than multicounter machines.

Q: So you are going to define a class of primitive recursive algorithms using basic arithmetical ASMs.

A: Yes, we consider a variant of Definition 5.2.

Definition 5.5. *Basic arithmetical primitive recursive algorithms (in short* Apra*) are the algorithms associated to basic arithmetical ASMs which halt in time bounded by a primitive recursive function.*

Thus, a basic arithmetical primitive recursive algorithm can be viewed as a pair (A, f) where A is a basic arithmetical ASM and f is a primitive recursive function which bounds the lengths of the runs of A.

Q: Of course, due to Theorem 5.1, halting in primitive recursive time is equivalent to halting in time bounded by a primitive recursive function.

A: Sure. However, you may ignore the exact halting time and merely know a primitive recursive bound.

5.3 APRA and the imperative programming language $\text{LOOP}_{\texttt{halt}}$

Q: Why is APRA a reasonable class?

A: We hope to convince you that this class is the good one. Let us show you some programming language that satisfactorily implements APRA. Recall that we really care about programming languages.

We start with the the sub-language LOOP of LOOP^{ω} with no high-order procedural variables. It is similar to the Meyer-Ritchie *Loop* language [MR76] which extensionally expresses only primitive recursive functions. Add to this language LOOP a command to force the program to halt:

$$c \ ::= \ \ldots \ | \ \texttt{halt;}$$

This gives an imperative language $\text{LOOP}_{\texttt{halt}}$.

Q: Is `halt` really an earth-shaking command?

A: Recall that the best time algorithm for *min* cannot be programmed in LOOP (cf. [CLV06] for a direct proof). But we have seen an $\mathbb{N}$-typed ASM program for that *min* algorithm. Now, here is a $\text{LOOP}_{\texttt{halt}}$ program for it.

```
if m=0 then min := 0 and halt;                1
if n=0 then min := 0 and halt;                2
x := m;                                       3
y := n;                                       4
for i := 1 to m do                            5
        x := x - 1;                           6
        y := y - 1;                           7
        if y=0 then min := n and halt;        8
end for;                                      9
min := m;                                    10
```

Q: Nice and so simple. So `halt` adds new algorithms. Now, how do $\text{LOOP}_{\texttt{halt}}$ and APRA operationally compare?

A: The easy direction is to associate a basic arithmetical ASM to a $\text{LOOP}_{\texttt{halt}}$ program. Attach dynamic constants of type $\mathbb{N}$ to the variables in the program, including the counter variables of loop instructions and the counter for the line currently executed. For instance, the above $\text{LOOP}_{\texttt{halt}}$ for the *min* function becomes the following ASM program (where $\|$ separates the instructions in a block):

if	$c = 1 \wedge m = 0$	*then*	$(min := 0 \parallel c := 11)$	*else*	$c := 2$	1
if	$c = 2 \wedge n = 0$	*then*	$(min := 0 \parallel c := 11)$	*else*	$c := 3$	2
if	$c = 3$	*then*	$(x := m \parallel c := 4)$			3
if	$c = 4$	*then*	$(y := n \parallel c := 5)$			4
if	$c = 5 \wedge i = m$	*then*	$c := 10$	*else*	$c := 6$	5
if	$c = 6$	*then*	$(x := x - 1 \parallel c := 7)$			6
if	$c = 7$	*then*	$(y := y - 1 \parallel c := 8)$			7
if	$c = 8 \wedge y = 0$	*then*	$(min := n \parallel c := 11)$	*else*	$c := 9$	8
if	$c = 9$	*then*	$(i := i + 1 \parallel c := 5)$			9
if	$c = 10$	*then*	$(min := m \parallel c := 11)$			10

Observe that when $c = 11$ the ASM program does nothing. Thus, the ASM assignment $c := 11$ corresponds to the `halt` command.

Q: Ok, it is an easy exercise to devise an ASM program such that the obtained basic arithmetical ASM simulates step by step the execution of any $\text{LOOP}_{\texttt{halt}}$ program. So every algorithm associated to some $\text{LOOP}_{\texttt{halt}}$ program is in APRA. What about the converse inclusion?

A: Let A be a basic arithmetical ASM with ASM program π_A and running time bounded by a primitive recursive function f. To get a $\text{LOOP}_{\texttt{halt}}$ program which captures the operational contents of A, we proceed in three main steps: 1) define a shell program, 2) get a core program, 3) "inject" the shell program into the core program.

To get the shell program, we first translate the ASM program π_A into a $\text{LOOP}_{\texttt{halt}}$ program. Then we duplicate the variables associated to the ASM dynamic constants. These new variables will represent the state before an update. Thus, comparing old and new values, we can detect a fixed point of the ASM run.

Q: But there is nothing but conditionals and updates in π_A. So this is quite a degenerate LOOP program!

A: Yes, this is a sequence of conditionals and updates that mimic the π_A program. This is, in fact, the best simulation we can manage since we simulate simultaneous updates by sequences. Thus, we loosen the step by step simulation: for some fixed k, one ASM step is simulated by $\leq k$ steps of the LOOP program. We can also get exactly k steps if we want.

Q: So far, there is still no loop. And what you get is not a very attractive program.

A: We agree, this is not the usual way to construct programs (for the time being!). But we just want to show that this programming language is expressive enough as concerns primitive recursive algorithms.

Now, we introduce the core program.

Recall that the running time of the ASM run is bounded by some primitive recursive function f. The core program is any LOOP program with running time greater than or equal to f. There are such core programs: since f is primitive

recursive there exists a LOOP program that computes f; complete this program with a loop bounded by the value of f just computed.

Q: The running time of the core program can be far much greater than f since f may not have good implementations in LOOP.

A: This is where the magic command `halt` comes in!

We "inject" the shell program into the core program. This means that at each step of the core program we also execute the shell program to mimic one step of the run of the given basic arithmetical ASM. Using the duplicated variables, we know when the ASM run halts. Besides the shell program we also inject in the core program some conditional instruction

$$\texttt{if}\ \ C\ \texttt{then halt}$$

where condition C uses the variables associated to the ASM (plus their duplicates) to check whether the simulated ASM run halts or not.

Q: Summing up, the core program has a running time long enough so that the shell program can be iteratively executed a number of times at least equal to the ASM running time. And the command `halt` allows to stop the execution of the core program at the right time. Now, how do you define this "injection" of the shell program into the core program?

A: Let P be a LOOP program and Q be a $\text{LOOP}_{\texttt{halt}}$ program. We define the insertion $P[Q]$ of Q in P by induction on the length of P:

P	$\equiv$	$x := e;\ com_s$
$P[Q]$	$\equiv$	$Q\ x := e;\ com_s[Q]$
P	$\equiv$	$\texttt{if}\ e\ \texttt{then}\ com_1\ \texttt{else}\ com_2\ \texttt{endif};\ com_s$
$P[Q]$	$\equiv$	$Q\ \texttt{if}\ e\ \texttt{then}\ com_1[Q]\ \texttt{else}\ com_2[Q]\ \texttt{endif};\ com_s[Q]$
P	$\equiv$	$\texttt{loop}\ expr_{int}\ \texttt{do}\ com\ \texttt{endloop};\ com_s$
$P[Q]$	$\equiv$	$Q\ \texttt{loop}\ expr_{int}\ \texttt{do}\ com[Q]\ Q\ \texttt{endloop};\ com_s[Q]$

Q: Semicolons are missing after Q in your table.

A: No. In fact, Q represents a piece of code ending with a semicolon!

Q: Ok. You do not obtain exact operational simulation. The environment of the $\text{LOOP}_{\texttt{halt}}$ program is richer than that of the simulated basic arithmetical ASM.

Q: Right. We obtain the following result.

Theorem 5.6. *Let g be a function from $\mathbb{N}^k$ to $\mathbb{N}$ and A be an* APRA *algorithm which computes g in time bounded by a primitive recursive function f. Then there is a LOOP_{halt} program that simulates A and runs in time $O(f)$.*

This leads to consider escape commands as good control structure for programming languages (see [Rob95] for a pedagogical discussion on that subject). Besides `halt`, one can consider kind of "escape from a loop" or "escape from a k-nested loop" or introduce more sophisticated control structures such that exception (see [ADA02] for instance).

5.4 Functional implementation of APRA

Q: Well, what you showed me makes good use of the fact that the LOOP language is close to ASMs with the update (or assignment) notion. But what about a functional implementation of APRA?

A: Following [CLV06] and [CPV09], we extract from Gödel system T the core of the functional language that simulates the class APRA. The fragment of system T which captures APRA is exactly the language induced by the definition of *primitive recursion with variable parameters* (see [Pét67]) with a mixed call-by strategy (by value on β–reduction and by-name for `rec` and `if` reductions).

Q: Hum... A functional language with two kinds of reductions. Do you know any language which truly shares such a feature?

A: In fact, in each usual functional language designers have added a way to use the opposite reduction strategy. For instance, Haskell for call-by-need and Ocaml for call-by-value for instance. This is no coincidence.

Q: So, you start from a LOOP_{halt} program and you translate it to a system T term?

A: No. Roughly speaking, a halting command does not go well with functional programming nor with a mathematical model. The simulation relies on another control structure related to the bounded loop itself that can be called *conditional bounded loop* and denoted by LOOPC.

Q: This kind of control structure has been used in PL/1.

A: This control structure can also be viewed as a conditional `exit` from a loop which can appear anywhere among the instructions inside the loop.

For convenience, we shall consider *down-to* bounded loops: loops which go downto 1.

$$\textit{command} \quad ::= \quad ... \mid \texttt{for}\ x_i := t\ \texttt{downto}\ 1\ \texttt{onlyIf}\ b\ \ \{s\}$$

The informal operational semantics is "loop until 1 is reached or b is false".

Before entering the translation, let us mention the definition of expressions in LOOPC :

$$\begin{array}{lcl} e & ::= & e_{int} \mid e_{bool} \\ e_{int} & ::= & \bar{n} \mid x \mid x-1 \mid x+1 \\ e_{bool} & ::= & \texttt{true} \mid \texttt{false} \mid x_i = x_j \mid \neg b \mid b_1 \wedge b_2 \end{array}$$

The desired translation is as follows.

Definition 5.7. *The translation (denoted by $^+$) of a* LOOPC *program with variables $\vec{x} = (x_1, \ldots, x_n)$ into a term in system T is defined by induction on*

expressions and commands as follows:

$$
\begin{array}{lll}
\bar{n}^* & \textit{is} & S^n(0) \\
\overline{\texttt{true}}^+ & \textit{is} & \texttt{true} \\
\overline{\texttt{false}}^+ & \textit{is} & \texttt{false} \\
x_i^+ & \textit{is} & x_i \\
(x_i+1)^+ & \textit{is} & \texttt{succ}(x_i) \\
(x_i-1)^+ & \textit{is} & \texttt{pred}(x_i) \\
\{\}^+ & \textit{is} & \vec{x} \\
\{x_i = e;\ s\}^+ & \textit{is} & \texttt{let}\ x_i\ =\ e^+\ \texttt{in}\ \{s\}^+ \\
\{c;\ s\}^+ & \textit{is} & \texttt{let}\ \vec{x}\ =\ c^+\ \texttt{in}\ \{s\}^+ \\
 & & \textit{if } c \textit{ is not an assignment} \\
(\texttt{if}\ b\ \{s_1\}\ \texttt{else}\ \{s_2\})^+ & \textit{is} & \texttt{if}(b^+, \{s_1\}^+, \{s_2\}^+) \\
(\texttt{for}\ x_i\ \texttt{in}\ e\ \texttt{downto}\ 1\ \texttt{onlyIf}\ b\ \ \{s\})^+ & \textit{is} & \\
\multicolumn{3}{l}{\qquad \texttt{rec}_{N,N}(e^+, \lambda\vec{x}.\vec{x},\ [y, x_i]\lambda\vec{x}.\texttt{if}(b^+, \texttt{let}\ \vec{x} = \{s\}^+\ \texttt{in}\ y\ \vec{x},\ \vec{x}))\ \ \vec{x}}
\end{array}
$$

Q: I don't understand, your translation of the conditional bounded loop gives a term in the system T at level 1 (due to $\texttt{rec}_{N,N}$). Is the target language beyond primitive recursion?

A: A priori, you're right! But if you analyze the term you will see that it doesn't use the full power of level one of system T. In fact, the translated term encodes a primitive recursive schema called primitive recursion with variable parameters which has the form

$$
\begin{array}{lcl}
f(0, \vec{y}) & = & g(\vec{y}) \\
f(x+1, \vec{y}) & = & h(x, f(x, j(x, \vec{y})), \vec{y})
\end{array}
$$

where g, h and j are also defined by primitive recursion with variable parameters.

Q: This looks much like the second-order primitive recursion to get the best algorithm for *min*, cf. §3.3.

A: This recursion schema offers a new control structure for the functional language (of primitive recursion). Though it is not expressible as such in system T, using level 1 of system T, we can mimic it. Now, observe that we do not use level 1 for its extensional power but we use it for its operational power which goes beyond that of level 0 even for primitive recursive functions.

Q: Ok, but in system T with call-by-value strategy, you mentioned earlier that there is no way to obtain the good algorithm for the *min* function!

A: Yes, with such kind of language call-by-value is an handicap. Other strategies may enlarge the algorithmic expressive power. The good strategy is the one that reduces `rec` by-name and `let` by-value.

Q: So, your result is similar to that of Theorem 4.2 relative to the translation of Loop$^\omega$ into system T. One reduction step in LoopC is simulated by one reduction step in T. Is that correct?

A: In fact, the simulation is only lockstep: one step in LoopC is translated by one or two steps in the `rec` term.

Theorem 5.8. *If* $\langle c, (\vec{x}, \vec{n})\rangle \rightarrow \langle c_1, (\vec{x}, \vec{n_1})\rangle$ *then* $c^+[\vec{n}^+/\vec{x}] \rightsquigarrow^{\leq 2} c_1^+[\vec{n_1}^+/\vec{x}]$ *where* $\rightarrow$ *denotes one reduction step of the* LoopC *language and* $\rightsquigarrow^{\leq i}$ *denotes a sequence of at most* i *reduction steps in the system* T *language.*

Acknowledgments.
Many thanks to Yuri Gurevich for numerous illuminating discussions. Many thanks too to Charles R. Wallace and Benjamin Rossman for suggestions and corrections to the first draft of this paper.

References

[ADA02] *Consolidated ADA Reference Manual: Language and Standard Libraries*, Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[APV] Ph. Andary, B. Patrou, and P. Valarcher. A representation theorem for primitive recursive algorithms, *Fundamenta Informaticae*, 107(4): 313-330 (2011).

[BC82] Gérard Berry and Pierre-Louis Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20(3):265–321, 1982.

[Bie03] Therese C. Biedl, Jonathan F. Buss, Erik D. Demaine, Martin L. Demaine, Mohammad Taghi Hajiaghayi and Tomás Vinar. Palindrome recognition using a multidimensional tape. *Theoretical Computer Science*, 302(1-3):475–480, 2003.

[BSS89] Lenore Blum, Mike Shub and Steve Smale. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bulletin American Mathematical Society*, 21(1):1–46, 1989.

[BS03] E. Börger and R. Stärk, *Abstract state machines: a method for high-level system design and analysis*, Springer-Verlag, 2003.

[Bou07] Olivier Bournez, Manuel L. Campagnolo, Daniel S. Graça and Emmanuel Hainry. Polynomial differential equations compute all real computable functions on computable compact intervals. *Journal of Complexity*, 23(3):157–166, 2007.

[BD95] Stephen Brookes and Denis Dancanet. Sequential algorithms, deterministic parallelism, and intensional expressiveness. *22nd Symposium on Principles of Programming Languages (POPL'95)*, 13–24, 1995.

[Col89] Loïc Colson. About primitive recursive algorithms. *Theoretical Computer Science*, 83(1):57–69, 1991. Preliminary version in Proceedings ICALP 1989, *Lecture Notes in Computer Science*, 372:194–206, 1989.

[Col91] Loïc Colson. Représentation intensionnelle d'algorithmes dans les systèmes fonctionnels. *Thèse de doctorat*, Université Paris 7, 1991.

[Col96] Loïc Colson. A unary representation result for system T. *Annals of Mathematics and Artificial Intelligence*, 16:385–403, 1996.

[CF98] Loïc Colson and Daniel Fredholm. System T, call-by-value and the minimum problem. *Theoretical Computer Science*, 206:301–315, 1998.

[Coq92] Thierry Coquand. Une preuve directe du thérorème d'ultime obstination. *Comptes Rendus de l'Académie des Sciences*, Série I, 314:389–392, 1992.

[CLV06] Tristan Crolard, Samuel Lacas, and Pierre Valarcher. On the expressive power of the Loop language. *Nordic Journal of Computing*, 13(1-2):46–57, 2006.

[CPV09] Tristan Crolard, Emmanuel Polonowski and Pierre Valarcher. Extending the loop language with higher-order procedural variables. *ACM Transactions on Computational Logic* 10(4):1–37 (2009)

[Dav93] René David. Un algorithme primitif récursif pour la fonction Inf. *Comptes Rendus de l'Académie des Sciences*, Série I, 317:899–902, 1993.

[Dav94] René David. The Inf function in the system F. *Theoretical Computer Science*, 135:423-431, 1994.

[Dav01] René David. On the asymptotic behaviour of primitive recursive algorithms. *Theoretical Computer Science*, 266(1-2):159–193, 2001.

[Dav03] René David. Decidability results for primitive recursive algorithms. *Theoretical Computer Science*, 300(1-3):477–504, 2003.

[FG10] Marie Ferbus and Serge Grigorieff. ASMs and Operational Algorithmic Completeness of Lambda Calculus. *Fields of Logic and Computation*, Nachum Dershowitz, Wolfgang Reisig editors. Lecture Notes in Computer Science 6300:301–327, Springer, 2010.

[GV10] Serge Grigorieff and Pierre Valarcher. Evolving multialgebras unify all usual sequential computation models. *STACS 2010*, 301–327, Jean-Yves Marion, Thomas Schwentick, editors, 2010.

[GV12] Serge Grigorieff and Pierre Valarcher. Functionals using Bounded Information and the Dynamics of Algorithms. *LICS 2012*.

[GV12a] Serge Grigorieff and Pierre Valarcher. Computation models and ASM frameworks. *In preparation*.

[Gur91] Yuri Gurevich. Evolving algebras: an attempt to discover semantics. *Bulletin of EATCS*, 43:264–284, June 1991.

[Gur99] Yuri Gurevich. The Sequential ASM Thesis. *Bulletin of EATCS*, 43:264–284, February 1999.

[Gur12] Yuri Gurevich. What Is an Algorithm? *SOFSEM*, Lecture Notes in Computer Science, 7147:31–42, 2012.

[Hen65] F. C. Hennie. One-Tape, Off-Line Turing Machine Computations. *Information and Control*, 8(6):553–578, 1965.

[1] Kurt Schütte. *Fundamental algorithms.* The Art of Computer Programming, volume 1. Addison-Wesley, 1968.

[MR76] Albert R. Meyer and Dennis M. Ritchie. The complexity of loop programs *Proc. 22nd National ACM Conference*, 465–470, 1976.

[MV09] David Michel and Pierre Valarcher. A total functional programming language that computes APRA. *Studies in Weak Arithmetic, CSLI Lecture Notes*, 196:1–19, Stanford, 2009.

[Mos03] Yiannis Moschovakis. On primitive recursive algorithms and the greatest common divisor function. *Theoretical Computer Science*, 301 (1-3):1–30, 2003 .

[Pau79] Wolfgang J. Paul. Kolmogorov Complexity and Lower Bounds. *Proc. Fundamentals of Computation Theory (FCT)*, Budach, L., editor, pages 325–334, Berlin/Wendisch-Rietz Akademie-Verlag, 1979.

[Pét67] Rózsa Péter. *Recursive Functions.* Academic Press, 1967.

[Ric54] H. G. Rice. Recursive real numbers. *Proceedings American Mathematical Society*, 5:784–791, 1954.

[Rob95] Eric S. Roberts. Loop exits and structured programming: reopening the debate. *SIGCSE'95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer Science Education*, 268–272, 1995.

[Rog67] Hartley Rogers Jr. *Theory of recursive functions and effective computability.* McGraw-Hill, 1967.

[Sch77] Kurt Schütte. *Proof theory.* Springer, 1977.

[Tur36] Alan Mathison Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proceedings London Math. Soc., series 2*, 42:230–265, 1936. Correction ibid. 43, pp 544-546 (1937).

[Val08] Pierre Valarcher. A complete characterization of primitive recursive intensional behaviours. *Theoretical Informatics and Applications*, 42(1):62–82, 2008.