A Short Introduction to Implicit Computational Complexity

Ugo Dal Lago

Dipartimento di Scienze dell'Informazione, Università di Bologna
Mura Anteo Zamboni 7, 40127 Bologna, Italy
dallago@cs.unibo.it

Abstract. These lecture notes are meant to serve as a short introduction to implicit computational complexity for those students who have little or no knowledge of recursion theory and proof theory. They have been obtained by enriching and polishing a set of notes the author wrote for a course (on the same subject) he gave at ESSLLI 2010. These notes are definitely *not* meant to be comprehensive nor exhaustive, but on the other hand much effort has been done to keep them self-contained.

1 Introduction

While *computability* theory studies the boundary between what can and what cannot be computed by an algorithm without putting any specific constraint on the behavior of the machine which execute it, *complexity* theory refine the analysis by exploring the class of computable functions and classifying them. The classification is based on the amount of *resources* algorithms computing those functions require when executed by simple, paradigmatic machines like Turing machines. Resources of interest can be computation time or space[1], while bounds are not expressed by fixed, absolute constants but rather as a function of the size of the input to the algorithm. This way algorithms are allowed to consume larger and larger amounts of resources when the size of the input increases.

A complexity class can be defined as the collection of all those functions which can be computed by an algorithm working within resource bounds of a certain kind. As an example, we can form the class FP of those functions which can be computed in *polynomial time*, i.e. within an amount of computation time bounded by a polynomial on the size of the input. As another example, we can form the class FLOGSPACE of those functions which can be computed in logarithmic space. Complexity theory has developed grandly since its inception [10,7] and nowadays an enormous number of complexity classes are studied, even if not so much is known about the relations between them. Traditionally, complexity classes are studied by carefully analyzing the combinatorial behavior of machines rather than the way algorithms are formulated, namely programs.

[1] Other resources, such as communication, are of some interest, but we will not deal with them in these notes.

Starting from the early nineties, researchers in mathematical logic and computer science have introduced *implicit* characterizations of the various complexity classes, starting from FP and later generalizing the approach to many other classes. By implicit, we here mean that classes are not given by constraining the amount of resources a *machine* is allowed to use, but rather by imposing linguistic constraints on the way *algorithms* are formulated. This idea has developed into an area called *implicit computational complexity*, which at the time of writing is very active, with annual thematic workshops specifically devoted to it.

The purpose of these lecture notes is giving the reader the possibility of understanding *what* implicit computational complexity is, the typical *flavour* of the results obtained in it and *how* these results are proved. There is no hope of being exhaustive here, while some effort has been done to keep the presentation as self-contained as possible: no prior knowledge of computability and complexity is required, as an entire section (Section 3) is devoted to giving all the needed preliminaries.

2 Mathematical Preliminaries

Throughout these lecture notes, we will use the language of mathematics. However, not much is assumed about the reader's knowledge. In this section, we recall some concepts and notation that we will use throughout the course.

Sets will be denoted with meta-variables like A, B, C. The *cartesian product* of two sets A and B is the set $A \times B$ of all pairs whose first component is in A and whose second component is in B: $A \times B = \{(a, b) \mid a \in A, b \in B\}$. A *binary relation* on A and B is a any subset of the cartesian product $A \times B$. Similarly, one can define the cartesian product $A_1 \times A_2 \times \ldots \times A_n$ and the corresponding notion of *n-ary relation*. Given a set A, A^n stands for the cartesian product

$$\underbrace{A \times A \times \ldots \times A}_{n \text{ times}}.$$

The set of *natural numbers* (i.e. of the non-negative integers) is $\mathbb{N}$.

There are very few concepts in mathematics which are more pervasive and ubiquitous than the notion of a function. Functions are fundamental objects of study in mathematical analysis, number theory and algebra, just to give some examples. A *partial function* from a set A to a set B is a binary relation on A and B such that for every $a \in A$ there is *at most* one $b \in B$ such that $(a, b) \in f$. If this condition is satisfied, we write $f : A \rightharpoonup B$. A partial function $f : A \rightharpoonup B$ is said to be *total* or simply a *function* if for every $a \in A$ it is guaranteed that $(a, b) \in f$ for some b. If this condition is satisfied, we write $f : A \rightarrow B$. If $f : A \rightharpoonup B$ is a partial function and $a \in A$, $f(a)$ denotes the element $b \in B$ such that $(a, b) \in f$, provided it exists; notice, however, that if f is total, then $f(a)$ is guaranteed to exist.

A function $f : A \rightarrow B$ is said to be *injective* if for every $a, b \in A$, $f(a) = f(b)$ implies $a = b$. On the other hand, f is said to be *surjective* if for every $b \in B$ there is $a \in A$ such that $f(a) = b$. A function which is both injective and surjective is said to be *bijective*. If A is a finite set, i.e. a set with a finite number n of elements, $|A|$ is simply n. If $f : A \rightarrow B$ is bijective, then A and B are said to be *in bijective correspondence* or to *to have the same cardinality*, and we write $|A| = |B|$.

A class of binary relations which are very useful in mathematics and computer science are order relations. Before introducing them, however, one is required to give some preliminary notions about binary relations. Let R be a binary relation on A. Then:

- R is said to be *reflexive* if $(a, a) \in R$ for every $a \in A$. R is said to be *irreflexive* if $(a, a) \notin R$ for every $a \in A$;
- R is said to be *symmetric* if $(b, a) \in R$ whenever $(a, b) \in R$. R is said to be *antisymmetric* if $(b, a) \in R$ and $(a, b) \in R$ imply $a = b$;
- R is said to be *transitive* if $(a, c) \in R$ whenever both (a, b) and (b, c) are in R.

This way we can define orders of various kinds:

- A *strict order* is a binary relation which is irreflexive and transitive.
- A *partial order* is a binary relation which is reflexive, antisymmetric and transitive.
- A *quasi order* is a binary relation which is reflexive and transitive.

An *alphabet* is a just a finite nonempty set, denoted with meta-variables like Σ, Υ, Φ. Elements of an alphabet Σ are called *symbols* and are usually denoted with meta-variables like c, d, e. A *string* or a *word* over an alphabet Σ is a finite, ordered, possibly empty sequence of symbols from Σ. Given an alphabet Σ, the *Kleene's closure* Σ^* of Σ is the set of all words over Σ, including the empty word ε. Words are ranged over by meta-variables like W, V.

Lemma 1. *For every alphabet Σ, $|\Sigma^*| = |\mathbb{N}|$.*

A *language* over the alphabet Σ is a subset of Σ^*. Among the formally definable languages there are *programming languages*, that is to say languages whose elements are strings which are meant to be *programs*. Programs will be denoted with meta-variables like P, R, Q.

One way of defining a language $\mathcal{L}$ is by saying that $\mathcal{L}$ is the smallest set satisfying some closure conditions, formulated as a set of productions. For example, the language $\mathcal{P}$ of *palindrome* words[2] over the alphabet $\{\mathtt{a}, \mathtt{b}\}$ can be defined as follows

$$W ::= \varepsilon \mid \mathtt{a} \mid \mathtt{b} \mid \mathtt{a}W\mathtt{a} \mid \mathtt{b}W\mathtt{b}.$$

Among the elements of $\mathcal{P}$, there are `aba` or `baab`, but clearly not `ab`. Interesting languages are the following:

- The language $\mathcal{N}$ of all sequences in $\{\mathtt{0}, \ldots, \mathtt{9}\}^*$ denoting natural numbers. It will be ranged over by meta-variables like N, M. Given N, $[\![N]\!] \in \mathbb{N}$ is the natural number denoted by N.
- The language $\mathcal{B}$ of all binary strings, namely $\{\mathtt{0}, \mathtt{1}\}^*$.

[2] A word is said to be palindrome if it is equal to its reverse.

3 Computability and Complexity: An Introduction

The concept of a function can be considered as one of the most basic, yet powerful, way to give meaning to computation, i.e. to capture *what* programs do. Any program is interpreted by the function mapping any possible value the program can take as *input* from the environment to the specific value the program produces as an *output*, if any. This way, every program is interpreted as a function, which however can be partial whenever the program does not terminate in correspondence to certain inputs. If a function f is the interpretation of a program P, then f is said to be the *function computed by* f, or simply *computable*. The notion of a computable function can be formalized in many different ways. In this section, three possibilities will be analyzed and some arguments will be given as for why they are all equivalent.

A function f is computable whenever the process of computing its value $f(x)$ from x is *effective*, meaning that it can be turned into an algorithm. There is no guarantee, however, about the amount of *resources* such an algorithm requires. Some algorithms can be very inefficient, e.g. they can require an enormous amount of time to produce their *output* from their *input*. Sometimes, this inefficiency is due to poor design. But it can well happen that the function f intrinsically requires a great amount of resources to be computed. The field of *computational complexity* studies precisely this kind of issues, aiming at classifying functions based on the amount of resources they require to be computed. Computational complexity will be briefly introduced in Section 3.5 below.

3.1 Counter Machines

Programs are ways to communicate one's intention to machines, which then follow the *instructions* the program is made with. One of the simplest machines one can think of are counter machines, which consist in some finite *counters*, each capable of containing a natural number. Counters can be manipulated by incrementing the natural number they contain or by testing the content of any counter. Counter machines are *programmed* by so-called counter programs, which consist in a finite sequence of instructions.

The language $\mathcal{C}$ of *counter programs* is defined as follows:

$$\begin{aligned} P &::= I \mid I\texttt{,}P \\ I &::= \texttt{inc(}N\texttt{)} \mid \texttt{jmpz(}N\texttt{,}N\texttt{)} \end{aligned}$$

where N ranges over $\mathcal{N}$ and thus stands for any natural number in decimal notation, e.g. `12`, `0`, `67123`. In other words, a counter program is a sequence

$$I_1\texttt{,}I_2\texttt{,}\ \dots\ \texttt{,}I_n$$

where each I_i is either $\texttt{inc(}N\texttt{)}$ or $\texttt{jmpz(}N\texttt{,}M\texttt{)}$. An example of a counter program is the following one:

$$\texttt{jmpz(0,4),inc(1),jmpz(2,1)} \qquad (1)$$

The instructions in a counter programs are meant to modify the content of some registers $R_0, R_1, \ldots$, each containing a natural number. More specifically:

- The instruction `inc(`N`)` increments by one the content of R_n (where $n = [\![N]\!]$ is the natural number denoted by N). The next instruction to be executed is the following one in the program.
- The instruction `jmpz(`N`,`M`)` determines the content n of $R_{[\![N]\!]}$ and
  - If n is positive, decrements $R_{[\![N]\!]}$ by one; the next instruction is the following one in the program.
  - If it is zero, the next instruction to be executed is the $R_{[\![M]\!]}$-th in the program.

Whenever the next instruction to be executed cannot be determined following the two rules above, the execution of the program terminates.

As an example, consider the counter program (1). It is easy to realize that what it does is simply copying the content of R_0 into R_1, provided the initial content of R_n is 0 whenever $n > 1$. The execution of (1) when the initial value of R_0 is 4 can be summarized as in the following table:

Current Instruction	R_0	R_1	R_2
`jmpz(0,4)`	4	0	0
`inc(1)`	3	0	0
`jmpz(2,1)`	3	1	0
`jmpz(0,4)`	3	1	0
`inc(1)`	2	1	0
`jmpz(2,1)`	2	2	0
`jmpz(0,4)`	2	2	0
`inc(1)`	1	2	0
`jmpz(2,1)`	1	3	0
`jmpz(0,4)`	1	3	0
`inc(1)`	0	3	0
`jmpz(2,1)`	0	4	0
`jmpz(0,4)`	0	4	0

If the instruction `jmpz(0,4)` is executed in a configuration when R_0 contains 0, the counter program stops, because the would-be-next instruction (the fourth) does not exist.

Even if the instruction set of counter programs is very limited, one can verify that many arithmetic operations can be implemented, including addition, multiplication, exponentiation, etc. Moreover, programs can be sequentially composed, allowing for more and more complicated operations to be implemented. But how can one formally *define* what a counter program actually compute?

The partial function $[\![P]\!]_{\mathcal{C}} : \mathbb{N} \rightarrow \mathbb{N}$ computed by a counter program P is defined by letting $[\![P]\!]_{\mathcal{C}}(n)$ be the content of the register R_0 after executing P (provided the execution terminates). When the computation starts, R_0 is set to n and all the other registers $R_1, R_2, \ldots$ are assumed to contain the natural number 0.

Now, consider the program P defined as follows:

```
inc(1),
jmpz(0,4),
jmpz(2,1),
jmpz(1,9),
jmpz(0,7),
jmpz(2,4),
inc(0),
jmpz(2,4)
```

The function $[\![P]\!]_{\mathcal{C}}$ is easily seen to be

$$[\![P]\!]_{\mathcal{C}}(n) = \begin{cases} 1 \text{ if } n \text{ is even} \\ 0 \text{ if } n \text{ is odd} \end{cases}$$

Indeed, the first three instructions allow to swap the contents of R_0 and R_1. The other ones alternatively set R_0 to 1 and to 0 a number of times equal to the content of R_1. In other words, the number of times R_0 is "switched" equals the input.

Of course, one can easily define nonterminating counter programs. A trivial example is the program R defined as `jmpz(1,1)`. The function $[\![R]\!]_{\mathcal{C}}$ is the (partial!) function which is everywhere undefined. The set of partial functions from $\mathbb{N}$ to itself which can be computed by counter programs is denoted CC. One may wonder whether CC is a definitive answer to the question of what should be *the* set of computable functions given that, after all, counter programs are very simple. For the moment, let us analyze another notion of a program.

3.2 Turing Machines

Turing machines [23] are probably the most influential computational model. In their simplest formulation, they consist of a *tape*, subdivided into infinitely many contiguous cells, each containing a symbol from a finite alphabet. Computation consists in modifying the content of the tape by moving a *head* around it which can read and write symbols from and into cells. We here present Turing machines in a slightly nonstandard language theoretic manner, instead of the usual automata theoretic way.

The language $\mathcal{T}$ of *Turing programs* is defined as follows:

$$\begin{aligned} P &::= I \mid I\texttt{,}P \\ I &::= \texttt{(}N\texttt{,}c\texttt{)}\ \texttt{>}\ \texttt{(}N\texttt{,}a\texttt{)} \mid \texttt{(}N\texttt{,}c\texttt{)}\ \texttt{>}\ \texttt{stop} \end{aligned}$$

where c ranges over the *input alphabet* Σ which includes a special symbol $\Box$ (called the *blank symbol*), and a ranges over the alphabet alphabet $\Sigma \cup \{\texttt{<}, \texttt{>}\}$. Intuitively, an instruction $\texttt{(}N\texttt{,}c\texttt{)}\ \texttt{>}\ \texttt{(}M\texttt{,}a\texttt{)}$ tells the machine to move to state M when the symbol under the head is c and the current state is N. Moreover, if a is in Σ the symbol under the head is changed to a, while if a is `<`, then, the symbol c is left unchanged but the head moves one position leftward. Similarly when a is `>`. Finally, the instruction `(`N`,`c`) > stop` tells the machine to simply stop whenever in state N and faced with the symbol c.

We will restrict our attention to *deterministic* Turing programs, namely those Turing programs such that no two instructions have exactly the same left-hand-side. Determinism guarantees that the behavior of the Turing program is well-defined given the content of the tape and the internal state.

An example of a (deterministic) Turing program on $\{\mathtt{0},\mathtt{1}\}$ is R:

```
(0,□) > (1,>),
(1,0) > (2,1),
(1,1) > (2,0),
(2,0) > (1,>),
(2,1) > (1,>),
(1,□) > stop
```

Formally, the computation of a Turing program P on Σ is best described by way of the notion of a *configuration*, which is an element of $\mathcal{F}(P) = \Sigma^* \times \Sigma \times \Sigma^* \times \mathcal{M}$ where $\mathcal{M}$ is the (finite!) subset of those elements of $\mathcal{N}$ which appear in the program (we assume $\mathtt{0}$ is always in $\mathcal{M}$). Intuitively:

- The first component is the portion of the tape lying on the left of the head, in reverse order.
- The second component is the symbol the head is reading.
- The third component is the portion of the tape lying on the right of the head.
- The fourth is the *state*.

The configuration $(\varepsilon, \Box, W, \mathtt{0})$ is said to be the *initial configuration* for W. A configuration is *terminating with output* V if it is in the form (X, c, W, N), `(`N`,`c`) > stop` is part of P, and V is obtained from X by erasing all instances of $\Box$. A binary relation $\xrightarrow{P}$ on $\mathcal{F}(P)$ can be defined easily, and induces a notion of reachability between configurations: D is *reachable* from C if there is a finite (possibly empty) sequence of configurations $E_1, \ldots, E_n$ such that

$$C = E_1 \xrightarrow{P} E_2 \xrightarrow{P} \ldots, \xrightarrow{P} E_n = D.$$

The partial function computed by a Turing program P on $\Sigma \cup \{\Box\}$ is the function $[\![P]\!]_{\mathcal{T}} : \Sigma^* \rightharpoonup \Sigma^*$ defined as follows: $[\![P]\!]_{\mathcal{T}}(W) = V$ if there is a terminating configuration with output V which is reachable from the initial configuration for W. Otherwise, $[\![P]\!]_{\mathcal{T}}(W)$ is undefined.

Consider, again as an example, the Turing program R defined above. And consider how it behaves when fed with the string `010`:

$$\begin{aligned}(\varepsilon, \Box, \mathtt{010}, \mathtt{0}) &\xrightarrow{R} (\Box, \mathtt{0}, \mathtt{10}, \mathtt{1}) \xrightarrow{R} (\Box, \mathtt{1}, \mathtt{10}, \mathtt{2})\\ &\xrightarrow{R} (\Box\mathtt{1}, \mathtt{1}, \mathtt{0}, \mathtt{1}) \xrightarrow{R} (\Box\mathtt{1}, \mathtt{0}, \mathtt{0}, \mathtt{2})\\ &\xrightarrow{R} (\Box\mathtt{10}, \mathtt{0}, \varepsilon, \mathtt{1}) \xrightarrow{R} (\Box\mathtt{10}, \mathtt{1}, \varepsilon, \mathtt{2})\\ &\xrightarrow{R} (\Box\mathtt{101}, \Box, \varepsilon, \mathtt{1}).\end{aligned}$$

All occurrences of `0` (respectively, of `1`) in the input string have been converted to `1` (respectively, to `0`). Generalizing a bit, one can be easily prove that $[\![P]\!]_{\mathcal{T}}$ is precisely the function which flips every bit in the input string.

The set of partial functions which can be computed by a Turing machine is denoted TC. Again, the computational model we have just introduced is very simple, so it is in principle debatable whether it corresponds to the intuitive notion of an effective procedure.

3.3 Computability

A basic, fundamental, question, which is at the heart of theoretical computer science, is the following: *can all functions between natural numbers be computed by programs?*. Or, equivalently but more concretely, *is every function from* $\mathbb{N}$ *to* $\mathbb{N}$ *the interpretation of some program?* The answer is negative, and is a consequence of the following lemma:

Lemma 2 (Cantor). *Both* $|\mathbb{N}| \neq |\mathbb{N} \to \mathbb{N}|$ *and* $|\mathbb{N}| \neq |\mathbb{N} \rightharpoonup \mathbb{N}|$.

Proof. The proof is by contradiction. Suppose that $\mathbb{N}$ and $\mathbb{N} \to \mathbb{N}$ are in bijective correspondence. This implies that $\mathbb{N} \to \mathbb{N}$ can be arranged in a sequence $f_0, f_1, f_2, \ldots$ in such a way that every function in $\mathbb{N} \to \mathbb{N}$ appears in the list. Then, define g as follows: $g(n) = f_n(n) + 1$. This means that g is different from all the functions in the sequence above. Indeed, if $g = f_m$, then $f_m(m)$ is equal to $g(m)$ which, by definition, is nothing but $f_m(m) + 1$. So, we have found a function which cannot be in the sequence above, namely g. And this contradicts the assumption on the equipotency of $\mathbb{N}$ and $\mathbb{N} \to \mathbb{N}$. One can prove similarly that $|\mathbb{N}| \neq |\mathbb{N} \rightharpoonup \mathbb{N}|$. $\square$

As an easy consequence, one get the following:

Proposition 1. *There are partial functions which are not computable whenever the notion of computability is formulated by way of languages, as in counter programs and Turing programs.*

Proof. Whenever the notion of a computable function is formulated by referring to languages, the set of computable function has at most countable cardinality (i.e. the same cardinality as $\mathbb{N}$), because the cardinality of any language of strings of symbols from a *finite* alphabet is itself at most countable, by Lemma 1. By Lemma 2, this cardinality is different from the cardinality of the set of all (partial or total) functions. As a consequence, there must be functions which do not correspond to any program. $\square$

The way we proved Proposition 1 is nonconstructive: one cannot directly extract from it an uncomputable function definition. Turing's original proof, instead, was constructive and based on the diagonalisation technique. Actually, there are many concrete, easily definable functions which can be proved *not* to be computable. An example is the halting function for counter programs: given (the encoding) of a counter program P and an input n for it, return 1 if $[\![P]\!]_{\mathcal{C}}$ is defined on n and 0 otherwise.

Proposition 1 tells us that there must be non-computable functions, independently from the underlying programming language. Another interesting question is in order: does the class of computable functions depend on the underlying programming language? Saying it another way: do different programming languages have different expressive powers in terms of the classes of functions they compute? Programming languages are usually designed to be as *expressive* as possible, i.e., to allow the programmer to compute as many functions as possible. Actually, the large majority of these languages have been proved to be equivalent. Counter and Turing programs are no exceptions. The class of functions computable by *any* programming language of this kind is usually referred to simply as the class of (partial) *computable functions*. The so-called Church-Turing thesis goes beyond saying that this set corresponds to the intuitive notion of an *effectively computable function*. Some programming languages, on the other hand, are specifically designed so as to be less powerful than others. In the next sections, we will analyze some of them

Notice that TC is a set of partial functions from Σ^* to Σ^*, while CC is a set of partial functions from $\mathbb{N}$ to $\mathbb{N}$. There are however canonical, effective ways to see a function in $\Sigma^* \rightharpoonup \Sigma^*$ as a function in $\mathbb{N} \rightharpoonup \mathbb{N}$. Modulo this correspondence, TC and CC can be proved to be the same class:

Theorem 1. $\mathsf{TC} = \mathsf{CC}$.

Proof (Sketch). The equality between two sets such as TC and CC both defined as the sets of all those functions computable by programs of a certain kind can be proved by showing that any program of the first kind can be turned into an *equivalent* program of the second kind. In this specific case, any Turing program P can be proved to have a "sibling" counter program $P\downarrow$ which computes the same function as P. This simulation can be carried out by allowing two registers like R_1 and R_2 to mimic the content of the left and right portions of the tape of a Turing machine. This way TC can be shown to be a subset of CC. The details are left to the reader, together with a proof of $\mathsf{CC} \subseteq \mathsf{TC}$. $\square$

3.4 Computability by Function Algebras

Machine models are definitely not the only way to characterize the set of computable function. In particular, there are other ways to do that based on *function algebras*. A function algebra consists of all those functions obtained by applying precisely defined constructions (any number of times) to a set of base functions. In this section we will analyze one of these function algebras, which is due to Kleene.

The following are the so-called *basic functions*:

- The function $z : \mathbb{N} \to \mathbb{N}$ is defined as follows: $z(n) = 0$ for every $n \in \mathbb{N}$.
- The function $s : \mathbb{N} \to \mathbb{N}$ is defined as follows: $s(n) = n + 1$ for every $n \in \mathbb{N}$.
- For every positive $n \in \mathbb{N}$ and for whenever $1 \leq m \leq n$, the function $\Pi^n_m : \mathbb{N}^n \to \mathbb{N}$ is defined as follows: $\Pi^n_m(k_1, \ldots, k_n) = k_m$.

Functions can be constructed from other functions in at least *three different ways*:

- Suppose that $n \in \mathbb{N}$ is positive, that $f : \mathbb{N}^n \to \mathbb{N}$ and that $g_m : \mathbb{N}^k \to \mathbb{N}$ for every $1 \leq m \leq n$. Then the *composition* of f and $g_1, \ldots, g_n$ is the function $h : \mathbb{N}^k \to \mathbb{N}$ defined as $h(\boldsymbol{i}) = f(g_1(\boldsymbol{i}), \ldots, g_n(\boldsymbol{i}))$.
- Suppose that $n \in \mathbb{N}$ is positive, that $f : \mathbb{N}^n \to \mathbb{N}$ and that $g : \mathbb{N}^{n+2} \to \mathbb{N}$. Then the function $h : \mathbb{N}^{n+1} \to \mathbb{N}$ defined as follows
$$\begin{aligned} h(0, \boldsymbol{m}) &= f(\boldsymbol{m}); \\ h(k+1, \boldsymbol{m}) &= g(k, \boldsymbol{m}, h(k, \boldsymbol{m})); \end{aligned}$$
is said to be defined by *primitive recursion* form f and g.
- Suppose that $n \in \mathbb{N}$ is positive and that $f : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$. Then the function $g : \mathbb{N}^n \rightharpoonup \mathbb{N}$ defined as follows:
$$g(m, \boldsymbol{i}) = \begin{cases} k \text{ if } f(0, \boldsymbol{i}), \ldots, f(k, \boldsymbol{i}) \text{ are all defined} \\ \qquad \text{and } f(k, \boldsymbol{i}) \text{ is the only one in the list being } 0. \\ \uparrow \text{ otherwise} \end{cases}$$
is said to be defined by *minimization* from f.

KC is the smallest class of functions which includes the basic functions and which is closed by composition, primitive recursion and minimization. In other words, KC contains all (and only) those functions which can be proved to be *partial recursive* (or simply *recursive*) by a finite number of applications of the rules of composition, primitive recursion and minimization, starting from basic functions. For every n, KC_n is the set of those n-ary partial functions on $\mathbb{N}$ which are in KC. The following are examples of functions in KC:

- The function *plustwo* : $\mathbb{N} \to \mathbb{N}$ defined as $\mathit{plustwo}(x) = x + 2$ is recursive: it can be obtained by composing s with itself;
- The function *succ3* : $\mathbb{N}^3 \to \mathbb{N}$ defined as $\mathit{succ3}(x, y, z) = z + 1$ is itself easily seen to be recursive, since it can be obtained by composing Π^3_3 and s;
- The function *add* : $\mathbb{N}^2 \to \mathbb{N}$ defined as $\mathit{add}(x, y) = x + y$ is recursive itself: it can be obtained by primitive recursion from Π^1_1 and *succ3*. Indeed:
$$\begin{aligned} 0 + m = m &= \Pi^1_1(m); \\ (1 + n) + m = 1 + (n + m) &= \mathit{succ3}(n, m, (n + m)). \end{aligned}$$

Surprisingly, the class of partial recursive function equals the class of computable functions:

Theorem 2. $\mathsf{KC} = \mathsf{CC}$.

Proof. The inclusion $\mathsf{KC} \subseteq \mathsf{CC}$ can be proved by showing that for every function $f \in \mathsf{KC}$ there is a counter program computing the same function. This can be done by induction on the *structure of the proof* of f being an element of KC, since this proof must be finite by definition. Conversely, $\mathsf{CC} \subseteq \mathsf{KC}$ can be proved by showing that any function f computable by a *Turing* machine is partial recursive, due to Theorem 1. This requires, in particular, to encode Turing machine configurations as natural numbers. □

A strict subclass PR of KC is the class of *primitive* recursive functions: it the smallest class of functions containing the basic functions and closed by composition, primitive recursion, *but not minimization*. All functions in PR are total, but not all total functions in KC are in PR (see [8] for more details).

3.5 Computational Complexity

From the beginning of the sixties, researchers started to be interested by a more *refined* notion of computability than the one we have looked at so far. More specifically, functions can be classified as being *efficiently computable* iff they are computable and, moreover, the amount of resources needed for their computation somehow remains *under control* when the size of their inputs grows. But how can we formally capture the idea above? One of the ways to do so consists in requiring that the *time* or *space* needed by some machine (computing the function) is bounded by a not-so-fast-growing function of the size of the input. But the next question is: how can we model the amount of time or space needed by a program? In these notes, we shall be concerned with time, only. A Turing program P on $\Sigma \cup \{\Box\}$ is said to *work in time* $f : \mathbb{N} \to \mathbb{N}$ iff for every initial configuration C for W, a final configuration D is reached in, at most, $f(|W|)$ computation steps, i.e., $C = E_1 \xrightarrow{P} \ldots \xrightarrow{P} E_n = D$, where $n \leq f(|W|)$. A Turing program P *works in polynomial time* if it works in time $f : \mathbb{N} \to \mathbb{N}$, where f is a polynomial.

One of the most interesting class is FP_Σ, the class of functions which are computable in polynomial time, namely the class of functions:

$$\{f : \Sigma^* \to \Sigma^* \mid f = [\![P]\!]_{\mathcal{T}} \text{ for some } P \text{ working in polynomial time}\}.$$

This class enjoys some basic properties which makes it quite robust (see, e.g. [1]). In particular, it is closed by composition: if $f, g \in \mathsf{FP}_\Sigma$, then $f \circ g$ is itself in FP_Σ. Moreover, this class is universally accepted as somehow capturing the intuitive notion of a "feasible" function, namely one which can be computed in an amount of time which grows in an acceptable way with respect to the size of the input.

Observe that FP_Σ has been defined by an explicit reference to both a machine model (Turing machines) and a class of bounds (polynomials). An interesting question is whether this class can be characterized, like CC, by way of function algebras, thus avoiding reference to explicit machine models, but also without any reference to polynomials. Implicit computational complexity aims precisely at this, namely at giving elegant, simple and machine-free characterizations of complexity classes.

4 Functional Programming and Complexity Classes

In this section, some characterizations of complexity classes based on functional programs and function algebras will be introduced. But while the latter have been already introduced in the form of an algebra KC characterizing computable functions, the former have not.

A signature $\mathcal{S}$ is a pair (Σ, α) where Σ is an alphabet and $\alpha : \Sigma \to \mathbb{N}$ assigns to any symbol c in Σ a natural number $\alpha(c)$, called its arity. The signature $\mathcal{S}_\mathbb{N}$ is defined as $(\{\mathtt{0}, \mathtt{s}\}, \alpha_\mathbb{N})$ where $\alpha_\mathbb{N}(\mathtt{0}) = 0$ and $\alpha_\mathbb{N}(\mathtt{s}) = 1$. Analogously, the signature $\mathcal{S}_\mathcal{B}$ is defined as $(\{\mathtt{0}, \mathtt{1}, \mathtt{e}\}, \alpha_\mathcal{B})$ where $\alpha_\mathcal{B}(\mathtt{0}) = \alpha_\mathcal{B}(\mathtt{1}) = 1$ and $\alpha_\mathcal{B}(\mathtt{e}) = 0$. Given two signatures $\mathcal{S}$ and $\mathcal{T}$ such that the underlying set of symbols are disjoint, one can naturally form the sum $\mathcal{S} + \mathcal{T}$.

Given a signature $\mathcal{S} = (\Sigma, \alpha)$, the set of *closed terms* over $\mathcal{S}$ is the smallest set of words $\mathcal{C}(\mathcal{S})$ satisfying the following closure property: if $\mathbf{f} \in \Sigma$ and $t_1, \ldots, t_{\alpha(\mathbf{f})} \in \mathcal{C}(\mathcal{S})$, then $\mathbf{f}\texttt{(}t_1, \ldots, t_{\alpha(\mathbf{f})}\texttt{)} \in \mathcal{C}(\mathcal{S})$. For example, the set of closed terms over the signature of natural numbers is:

$$\mathcal{C}(\mathcal{S}_\mathbb{N}) = \{\mathtt{0}, \mathtt{s(0)}, \mathtt{s(s(0))}, \ldots\}.$$

As another example, the set of closed terms over the signature of binary strings is:

$$\mathcal{C}(\mathcal{S}_\mathcal{B}) = \{\mathtt{e}, \mathtt{0(e)}, \mathtt{1(e)}, \mathtt{0(0(e))}, \ldots\}.$$

One can easily realize that there is indeed a natural correspondence between $\mathcal{C}(\mathcal{S}_\mathbb{N})$ (respectively, $\mathbb{N}$) and $\mathcal{C}(\mathcal{S}_\mathcal{B})$ (respectively, $\mathcal{B}$). Sometimes, one wants to form terms that contain variables: given a language of variables $\mathcal{L}$ distinct from Σ, the set of *open terms* $\mathcal{O}(\mathcal{S}, \mathcal{L})$ is defined as the smallest set of words including $\mathcal{L}$ and satisfying the closure condition above.

Suppose that $\Sigma = \{\mathbf{f}_1 \ldots, \mathbf{f}_n\}$ and that Υ is disjoint from Σ. The language of *functional programs* on $\mathcal{S} = (\Sigma, \alpha)$ and $\mathcal{T} = (\Upsilon, \beta)$ is defined as follows:

$$\begin{aligned} P &::= R \mid R\texttt{,}P \\ R &::= l \quad \texttt{->} \quad t \\ l &::= \mathbf{f}_1\texttt{(}p_1^1\texttt{,} \ldots \texttt{,}p_1^{\alpha(\mathbf{f}_1)}\texttt{)} \mid \ldots \mid \mathbf{f}_n\texttt{(}p_n^1\texttt{,} \ldots \texttt{,}p_n^{\alpha(\mathbf{f}_n)}\texttt{)} \end{aligned}$$

where the p_m^k range over $\mathcal{O}(\mathcal{T}, \mathcal{L})$, and t ranges over $\mathcal{O}(\mathcal{S} + \mathcal{T}, \mathcal{L})$. We here consider orthogonal functional programs only, i.e. we assume that no distinct two rules in the program are overlapping, that every variable appears at most once in the lhs of any rule, and that all variables occurring in the rhs also occur in the lhs. Given a functional program over $\mathcal{S}$ and $\mathcal{T}$:

- Its *constructor terms* are the words in $\mathcal{C}(\mathcal{T})$;
- Its *patterns* are the words in $\mathcal{O}(\mathcal{T}, \mathcal{L})$;
- Its *closed terms* are the words in $\mathcal{C}(\mathcal{S} + \mathcal{T})$;
- Its *terms* are the words in $\mathcal{O}(\mathcal{S} + \mathcal{T}, \mathcal{L})$.

How can we define the *evaluation* of functional programs? The simplest way to do that is probably defining a call-by-value notion of evaluation: given a term t, if t contains a subterm s in the form $\mathbf{f}(s_1, \ldots, s_n)$ where $s_1, \ldots, s_n$ are *constructor* terms, we can substitute (the occurrence of) s in t by the right hand side of the appropriate rule for $\mathbf{f}$, applied to the naturally defined substitution. In other words, we assume that reduction is call-by-value, i.e. that substitution triggering any reduction must assign constructor terms to variables. As an example, consider the following program over $\mathcal{S}_{\mathbb{N}}$ computing the addition of two natural numbers (in unary notation):

$$\begin{aligned}\texttt{add}(\texttt{0},x) &\texttt{ -> } x\\ \texttt{add}(\texttt{s}(x),y) &\texttt{ -> } \texttt{s}(\texttt{add}(x,y)).\end{aligned}$$

The evaluation of $\texttt{add}(\texttt{s}(\texttt{s}(\texttt{0})),\texttt{s}(\texttt{s}(\texttt{s}(\texttt{0}))))$ goes as follows:

$$\begin{aligned}\texttt{add}(\texttt{s}(\texttt{s}(\texttt{0})),\texttt{s}(\texttt{s}(\texttt{s}(\texttt{0})))) &\to \texttt{s}(\texttt{add}(\texttt{s}(\texttt{0}),\texttt{s}(\texttt{s}(\texttt{s}(\texttt{0})))))\\ &\to \texttt{s}(\texttt{s}(\texttt{add}(\texttt{0},\texttt{s}(\texttt{s}(\texttt{s}(\texttt{0}))))))\\ &\to \texttt{s}(\texttt{s}(\texttt{s}(\texttt{s}(\texttt{s}(\texttt{0}))))).\end{aligned}$$

Given a program P on $\mathcal{S}$ and $\mathcal{T}$, and a symbol $\mathbf{f}$, to which $\mathcal{S}$ attributes unary arity, the function $[\![P,\mathbf{f}\,]\!]_{\mathcal{T}}$ is defined as the partial function from $\mathcal{C}(\mathcal{T})$ to $\mathcal{C}(\mathcal{T})$ mapping any $t \in \mathcal{C}(\mathcal{T})$ to the term obtained by fully evaluating $\mathbf{f}(t)$. What's the class of functions which can be captured this way? Unsurprisingly, it coincides with the class of computable functions:

Theorem 3. *The set of functions from $\mathcal{C}(\mathcal{S}_{\mathbb{N}})$ to itself which can be computed by functional programs is exactly* KC.

Proof. For every $f \in \mathsf{KC}$, we can define a program P_f by induction on the proof that $f \in \mathsf{KC}$. Some interesting cases:

- Composition:

$$\mathbf{h}(x_1,\ldots,x_n) \texttt{ -> } \mathbf{g}(\mathbf{f}_1(x_1,\ldots,x_n),\ldots,\mathbf{f}_m(x_1,\ldots,x_n)).$$

- Minimization:

$$\begin{aligned}\mathbf{g}(x_1,\ldots,x_n) &\texttt{ -> } \mathbf{h}(\mathbf{f}(\texttt{0},x_1,\ldots,x_n),x_1,\ldots,x_n,\texttt{0});\\ \mathbf{h}(\texttt{0},x_1,\ldots,x_n,y) &\texttt{ -> } y;\\ \mathbf{h}(\texttt{s}(z),x_1,\ldots,x_n,y) &\texttt{ -> } \mathbf{h}(\mathbf{f}(\texttt{s}(y),x_1,\ldots,x_n),x_1,\ldots,x_n,\texttt{s}(y)).\end{aligned}$$

Conversely, one can prove that any program corresponds to a computable function by going through the characterization of computable functions based on Turing programs. □

Sometimes, it is useful to reason about the complexity of programs at the level of functional programs, without thinking at their implementation by machines, since the former are equipped with a very natural and simple notion of evaluation, with its associated notion of time, namely rewriting. In other words, one can easily define when a function can be computed by a functional program *in a certain amount of time*. What's the relation with complexity theory as we defined it in Section 3.5?

Theorem 4 (Invariance). *The set of functions from $\mathcal{C}(\mathcal{S}_{\Sigma})$ to itself which can be computed by functional programs in polynomial time are exactly the functions in* FP_{Σ}.

4.1 Safe Recursion

Finally, it is time to give an implicit characterization of the polynomial time computable functions. We here describe how the set of polynomial time computable functions can be characterized by way of a function algebra in the style of the one of general recursive functions (see Section 3.4). This result is due to Bellantoni and Cook [4] and is one of the earliest in implicit computational complexity.

The object of study here are *safe functions*, namely functions which are pairs (f, n), where $f : \mathcal{B}^m \to \mathcal{B}$ and $0 \leq n \leq m$. The number n identifies the number of *normal arguments* between those of f: they are the first n, while the other $m-n$ are *safe*. Following [4], we use semicolons to separate normal and safe arguments: if (f, n) is a safe function, we write $f(\boldsymbol{W}; \boldsymbol{V})$ to emphasize that the n words in $\boldsymbol{W}$ are the normal arguments, while the ones in $\boldsymbol{V}$ are the safe arguments.

First of all, we can form *basic safe functions*:

- The safe function $(e, 0)$ where $e : \mathcal{B} \to \mathcal{B}$ always returns the empty string ε.
- The safe function $(a_0, 0)$ where $a_0 : \mathcal{B} \to \mathcal{B}$ is defined as follows: $a_0(W) = \mathtt{0} \cdot W$.
- The safe function $(a_1, 0)$ where $a_1 : \mathcal{B} \to \mathcal{B}$ is defined as follows: $a_1(W) = \mathtt{1} \cdot W$.
- The safe function $(t, 0)$ where $t : \mathcal{B} \to \mathcal{B}$ is defined as follows: $t(\varepsilon) = \varepsilon$, $t(\mathtt{0}W) = W$ and $t(\mathtt{1}W) = W$.
- The safe function $(c, 0)$ where $c : \mathcal{B}^3 \to \mathcal{B}$ is defined as follows: $c(\varepsilon, W, V) = W$, $c(\mathtt{0}X, W, V) = W$ and $c(\mathtt{1}X, W, V) = V$.
- For every positive $n \in \mathbb{N}$ and for whenever $1 \leq m, k \leq n$, the safe function (Π^n_m, k), where Π^n_m is defined in a natural way.

Then, functions can be formed by simultaneous safe recursion on notation or by safe composition:

- Suppose that $(f : \mathcal{B}^n \to \mathcal{B}, m)$, that $(g_j : \mathcal{B}^k \to \mathcal{B}, k)$ for every $1 \leq j \leq m$, and that $(h_j : \mathcal{B}^{k+i} \to \mathcal{B}, k)$ for every $m+1 \leq j \leq n$. Then the *safe composition* of safe functions above is the safe function $(p : \mathcal{B}^{k+i} \to \mathcal{B}, k)$ defined as follows:
$$p(\boldsymbol{W}; \boldsymbol{V}) = f(g_1(\boldsymbol{W};), \ldots, g_m(\boldsymbol{W};); h_{m+1}(\boldsymbol{W}; \boldsymbol{V}), \ldots, h_n(\boldsymbol{W}; \boldsymbol{V})).$$
- Suppose that, for every $1 \leq i \leq j$ and for every $k \in \{0, 1\}$, the functions $(f^i : \mathcal{B}^n \rightharpoonup \mathcal{B}, m)$ and $(g^i_k : \mathcal{B}^{n+j+2} \rightharpoonup \mathcal{B}, m+1)$ are safe functions. Then for every $1 \leq i \leq j$, the functions $(h^i : \mathcal{B}^{n+1} \to \mathcal{B}, m+1)$ defined as follows
$$\begin{aligned} h^i(0, \boldsymbol{W}; \boldsymbol{V}) &= f(\boldsymbol{W}; \boldsymbol{V}); \\ h^i(\mathtt{0}X, \boldsymbol{W}; \boldsymbol{V}) &= g^i_0(X, \boldsymbol{W}; \boldsymbol{V}, h^1(X, \boldsymbol{W}; \boldsymbol{V}), \ldots, h^j(X, \boldsymbol{W}; \boldsymbol{V})); \\ h^i(\mathtt{1}X, \boldsymbol{W}; \boldsymbol{V}) &= g^i_1(X, \boldsymbol{W}; \boldsymbol{V}, h^1(X, \boldsymbol{W}; \boldsymbol{V}), \ldots, h^j(X, \boldsymbol{W}; \boldsymbol{V})); \end{aligned}$$
are said to be defined by *simultaneous safe recursion on notation* from (f^i, m) and $(g^i_k, m+1)$.

BCS is the smallest class of safe functions which includes the basic safe functions above and which is closed by safe composition and safe recursion. BC is the set of those functions $f : \mathcal{B} \to \mathcal{B}$ such that $(f, n) \in \mathsf{BCS}$ for some $n \in \{0, 1\}$.

The way we have defined basic functions and closure conditions are very similar to the corresponding notions we gave when defining partial recursive functions. The main differences are the absence of minimization and the class of basic functions, which is larger for BC. Moreover, BC is a class of functions on binary words, while KC is a class of (partial) functions on the natural numbers. Actually, BC is a slight variation on the algebra from [4]: safe recursion is more general here, i.e., it allows to generate more than one function simultaneously.

The rest of this section is devoted to proving that BC equals $\mathsf{FP}_{\{0,1\}}$, namely that BC captures the polynomial functions on binary strings precisely. The first step towards this goal consists in showing that the result of any function in BC cannot be too large compared to the size of the input:

Lemma 3. *For every* $(f : \mathcal{B}^n \to \mathcal{B}, m)$ *in* BCS*, there is a monotonically increasing polynomial* $p_f : \mathbb{N} \to \mathbb{N}$ *such that:*

$$|f(V_1,\ldots,V_n)| \leq p_f\left(\sum_{1\leq k\leq m} |V_k|\right) + \max_{m+1\leq k\leq n} |V_k|.$$

Proof. By induction on the structure of the proof that $(f,n) \in \mathsf{BCS}$.

- If f is a base function, then the appropriate polynomial p_f can be very easily found. As an example, is c, then p_c is the degenerate 0-ary polynomial 0.
- Suppose that the functions (f^j, n) (where $1 \leq j \leq i$ are obtained by simultaneous safe recursion from $(g^j, n-1), (h_0^j, n), (h_1^j, n)$. Let us consider the case where $i = 1$ and call f^1, g^1 and h_0^1, h_1^1 simply by f, g and h_0, h_1. Finally, let $p_f(m) = p_g(m) + m \cdot (p_{h_0}(m) + p_{h_1}(m))$. Then observe that:

$$\begin{aligned}
|f(\varepsilon, \boldsymbol{W}; \boldsymbol{V})| &= |g(\boldsymbol{W}; \boldsymbol{V})| \\
&\leq p_g\left(\sum_{1\leq m\leq n-1} |W_m|\right) + \max_{n\leq m\leq k} |V_m| \\
&\leq p_f\left(|\varepsilon| + \sum_{1\leq m\leq n-1} |W_m|\right) + \max_{n\leq m\leq k} |V_m|; \\
|f(\mathsf{0}\cdot X, \boldsymbol{W}; \boldsymbol{V})| &= |h_0(X, \boldsymbol{W}; \boldsymbol{V}, f(X, \boldsymbol{W}; \boldsymbol{V}))| \\
&\leq p_{h_0}\left(|X| + \sum_{1\leq m\leq n-1} |W_m|\right) \\
&\quad + \max\left\{\max_{n\leq m\leq k} |V_m|, |f(X, \boldsymbol{W}; \boldsymbol{V})|\right\} \\
&= p_{h_0}(i) + \max\left\{\max_{n\leq m\leq k} |V_m|, |f(X, \boldsymbol{W}; \boldsymbol{V})|\right\};
\end{aligned}$$

where

$$i = |X| + \sum_{1\leq m\leq n-1} |W_m|.$$

By induction hypothesis,

$$\begin{aligned}|f(\mathtt{0}\cdot X,\boldsymbol{W};\boldsymbol{V})| &\le p_g(i)+i\cdot(p_{h_0}(i)+p_{h_1}(i))+p_{h_0}(i)+\max_{n\le m\le k}|V_m|\\ &\le p_g(i+1)+(i+1)\cdot(p_{h_0}(i+1)+p_{h_1}(i+1))\\ &\quad+\max_{n\le m\le k}|V_m|\\ &=p_f(i+1)+\max_{n\le m\le k}|V_m|\\ &=p_f(|\mathtt{0}\cdot X|+\sum_{1\le m\le n-1}|W_m|)+\max_{n\le m\le k}|V_m|.\end{aligned}$$

This concludes the proof. □

Once we have the result above, we can prove that all functions in BC can be computed by a program working in polynomial time:

Theorem 5 (Polytime Soundness). $\mathsf{BC}\subseteq\mathsf{FP}_{\{0,1\}}$.

Polytime soundness implies that every function in BC can be computed in polynomial time by a Turing program. To reach our goal, however, we still need to prove that any function computable in polynomial time can be somehow "seen" as a function in BC. A first intermediate result consists in proving that all polynomials are in BC:

Lemma 4. *For every polynomial $p:\mathbb{N}\to\mathbb{N}$ with natural coefficients there is a safe function $(f,1)$ where $f:\mathcal{B}\to\mathcal{B}$ such that $|f(W)|=p(|W|)$ for every $W\in\mathcal{B}$.*

Proof. Define the following safe functions:

$$\begin{aligned}addaux(\varepsilon;W)&=W;\\ addaux(\mathtt{0}\cdot V;W)&=a_0(addaux(V;W));\\ addaux(\mathtt{1}\cdot V;W)&=a_1(addaux(V;W));\\ mult(\varepsilon,W;)&=W;\\ mult(\mathtt{0}\cdot V,W;)&=addaux(W;mult(V,W));\\ mult(\mathtt{1}\cdot V,W;)&=addaux(W;mult(V,W));\\ add(W,V;)&=addaux(W;V).\end{aligned}$$

Composing a finite number of times *add* and *mult*, we get the thesis. □

Once you know that any reasonable bound can be encoded, everything boils down to show that the transition function of a Turing program can be captured:

Theorem 6 (Polytime Completeness). $\mathsf{FP}_{\{0,1\}}\subseteq\mathsf{BC}$.

Proof. Given a Turing program P on $\{\mathtt{0},\mathtt{1},\square\}$, the four components of a configuration for P can all be encoded as a word in $\mathcal{B}$: the state and the symbol under the head will be both encoded by words of finite length, while the left and the right portions of the tape will be encoded by words of unbounded length. We can define the following functions by simultaneous recurrence:

- The function *left*$(W, V;)$ which returns the left portion of the tape after $|W|$ iterations, starting with input V.
- The function *right*$(W, V;)$ which returns the right portion of the tape after $|W|$ iterations, starting with input V.
- The function *head*$(W, V;)$ which returns the symbol under the head after $|W|$ iterations, starting with input V.
- The function *state*$(W, V;)$ which returns the state of the machine after $|W|$ iterations, starting with input V.

By Lemma 4, this implies the thesis. □

Corollary 1. $\mathsf{BC} = \mathsf{FP}_{\{0,1\}}$.

4.2 The Multiset Path Order

In the literature, many different methodologies have been introduced which somehow control the computational complexity of functional programs. In this section, we will describe of them which actually characterizes the class of primitive recursive functions, pointing to a refinement of it which is able to capture polytime computable functions.

A *multiset* M of a set A is a function $M : A \to \mathbb{N}$ which associates to any element $a \in A$ its *multiplicity* $M(a)$. In other words, a multiset is a set whose elements can "occur" more than once in the set. Given a sequence $a_1, \ldots, a_n$ of (not necessarily distinct) elements of A, the associated multiset is written as $\{\{a_1, \ldots, a_n\}\}$. Given a strict order $\prec$ on A, its multiset extension $\prec^m$ is a relation between multisets of A defined as follows: $M \prec^m N$ iff $M \neq N$ and for every $a \in A$ if $N(a) \prec M(a)$ then there is $b \in A$ with $a \prec b$ and $M(b) \prec N(b)$.

Lemma 5. *If $\prec$ is a strict order, then so is $\prec^m$.*

Let $\mathcal{S} = (\Sigma, \alpha)$ be a signature. A strict order $\prec_{\mathcal{S}}$ on Σ is said to be a *precedence*. Given a precedence $\prec_{\mathcal{S}}$, we can define a strict ordering on terms in $\mathcal{C}(\mathcal{S})$, called $\prec_{\mathsf{MPO},\mathcal{S}}$ as the smallest binary relation on $\mathcal{C}(\mathcal{S})$ satisfying the following conditions:

- $t \prec_{\mathsf{MPO},\mathcal{S}} \mathbf{f}(s_1, \ldots, s_n)$ whenever $t \prec_{\mathsf{MPO},\mathcal{S}} s_m$ for some $1 \leq m \leq n$;
- $t \prec_{\mathsf{MPO},\mathcal{S}} \mathbf{f}(s_1, \ldots, s_n)$ whenever $t = s_m$ for some $1 \leq m \leq n$;
- $\mathbf{f}(t_1, \ldots, t_n) \prec_{\mathsf{MPO},\mathcal{S}} \mathbf{g}(s_1, \ldots, s_m)$ if:
  - either $\mathbf{f} \prec_{\mathcal{S}} \mathbf{g}$ and $t_k \prec_{\mathsf{MPO},\mathcal{S}} \mathbf{g}(s_1, \ldots, s_m)$ for every $1 \leq k \leq n$.
  - or $\mathbf{f} = \mathbf{g}$ and $\{\{t_1, \ldots, t_n\}\} \prec^m_{\mathsf{MPO},\mathcal{S}} \{\{s_1, \ldots, s_m\}\}$.

A functional program P on $\mathcal{S}$ and $\mathcal{T}$ is said to *terminate by MPO* if there is a precedence $\prec_{\mathcal{S}+\mathcal{T}}$ such that for every rule l `->` t in the program P, it holds that $t \prec_{\mathsf{MPO},\mathcal{S}+\mathcal{T}} l$.

Theorem 7 (Hofbauer). *The class of functions computed by functional programs terminating by MPO coincides with* PR.

Proof. For every primitive recursive function f, it is easy to write a program which computes it and which terminates by MPO: actually, the required precedence can be easily defined by induction on the proof of f being primitive recursive. Consider, as an example, a rule we need when writing a functional program for a function f defined by primitive recursion from g and h:

$$\mathbf{f}(\mathtt{s}(x), y_1, \ldots, y_n) \quad \texttt{->} \quad \mathbf{h}(x, y_1, \ldots, y_n, \mathbf{f}(x, y_1, \ldots, y_n))$$

Suppose that $\mathbf{h} \prec \mathbf{f}$ and that $\mathtt{s} \prec \mathbf{f}, \mathbf{h}$. Then:

$$\begin{aligned}
x &\prec \mathtt{s}(x);\\
x &\prec \mathbf{f}(\mathtt{s}(x), y_1, \ldots, y_n);\\
\forall 1 \leq i \leq n. \qquad y_i &\prec \mathbf{f}(\mathtt{s}(x), y_1, \ldots, y_n);\\
\{\{x, y_1, \ldots, y_n\}\} &\prec^m \{\{\mathtt{s}(x), y_1, \ldots, y_n\}\};\\
\mathbf{f}(x, y_1, \ldots, y_n) &\prec \mathbf{f}(\mathtt{s}(x), y_1, \ldots, y_n);\\
\mathbf{h}(x, y_1, \ldots, y_n, \mathbf{f}(x, y_1, \ldots, y_n)) &\prec \mathbf{f}(\mathtt{s}(x), y_1, \ldots, y_n).
\end{aligned}$$

The proof that every functional program terminating by MPO computes a primitive recursive function is more difficult. □

Noticeably, a restriction of the multiset path order, called the light multiset path order, is capable of capturing the polytime functions [18].

5 The λ-calculus

Functional programs as we defined them in Section 4 are inherently *first-order*, that is to say functions can only take data as arguments, and not other functions. A somehow simpler, but very powerful, language can be defined which allows one to program *higher-order* functions, namely functions which can take other functions as arguments. We are here referring to the pure untyped λ-calculus [3,11], which is the subject of this section. It will be presented endowed with weak (that is, reduction cannot take place in the scope of an abstraction) call-by-value reduction.

Definition 1. *The following definitions are standard:*

- λ-terms *(or simply* terms*) are defined as follows:*

$$M ::= x \mid \lambda x.M \mid MM,$$

  where x ranges over a language $\mathcal{L}$. Λ denotes the set of all λ-terms.
- *Values are defined as follows:*

$$V ::= x \mid \lambda x.M.$$

- *The term obtained by* substituting *a term M for a variable x into another term N is denoted as $N\{M/x\}$.*

- *Weak call-by-value reduction is denoted by $\rightarrow_v$ and is obtained by closing call-by-value reduction under any applicative context:*

$$\frac{}{(\lambda x.M)V \rightarrow_v M\{V/x\}} \quad \frac{M \rightarrow_v N}{ML \rightarrow_v NL} \quad \frac{M \rightarrow_v N}{LM \rightarrow_v LN}$$

  Here M ranges over terms, while V ranges over values.
- *A* normal form *is a λ-term M which is irreducible, namely such that there is not any N with $M \rightarrow_v N$. A term M* has *a normal form iff $M \rightarrow_v^* N$. Otherwise, we write $M \uparrow$.*

Weak call-by-value reduction enjoy many nice properties, which are not all shared by strong reduction. One example is the following:

Lemma 6 (Strong Confluence). *For every M, if $M \rightarrow_v N$ and $M \rightarrow_v L$, then there is a term P such that $N \rightarrow_v P$ and $L \rightarrow_v P$.*

Despite its simplicity, weak call-by-value λ-calculus has a great expressive power. It's not immediately clear, however, how one could compute functions on the natural numbers by terms: how can we represent *natural numbers* themselves? There are various ways to encode data in the λ-calculus. For example, Scott's scheme allows to encode natural numbers in an easy and inductive way:

$$\ulcorner 0 \urcorner = \lambda x.\lambda y.x;$$
$$\ulcorner n+1 \urcorner = \lambda x.\lambda y.y\ulcorner n \urcorner.$$

A λ-term M is said to Scott-compute a function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ iff for every n, if $f(n)$ is defined and equals m, then $M\ulcorner n \urcorner \rightarrow_v^* \ulcorner m \urcorner$ and otherwise $M\ulcorner n \urcorner \uparrow$. SC is the class of functions which can be Scott-computed by a λ-term. As we've already said, the expressive power of the λ-calculus is enormous: it can compute all the general recursive functions:

Theorem 8. $\mathsf{SC} = \mathsf{CC}$.

Proof. The inclusion $\mathsf{CC} \subseteq \mathsf{SC}$ can be proved by giving, for every partial recursive function f, a λ-term M_f which computes f. This can be done by induction on the structure of the proof of f being an element of KC. Some examples:

- The successor function $s(x) = x + 1$ is computed by the λ-term

$$M_s = \lambda z.\lambda x.\lambda y.yz.$$

- If f is obtained by composing g and $h_1, \ldots, h_m$, then M_f is the λ-term

$$M_f = \lambda x_1. \ldots .\lambda x_n.M_g(M_{h_1}x_1 \ldots x_n) \ldots (M_{h_m}x_1 \ldots x_n).$$

- If f is defined by primitive recursion or minimization, then M_f makes use of fixpoint combinators (see [3] for more details).

The opposite inclusion $\mathsf{SC} \subseteq \mathsf{CC}$ can be proved by observing that λ-reduction is an effective procedure, which can be implemented by a Turing program. This concludes the proof. $\square$

The λ-calculus as we have defined it, in other words, can be seen as just another characterization of computability. If one is interested in characterizing smaller classes of functions, λ-calculus needs to be restricted, and this can be done in many different ways.

6 Further Readings

A nice and simple introduction to computability theory can be found in [8], while a more advanced and bibliographic reference is [21]. Computational complexity has been introduced in the sixties [7,10], and since then it has been generalized in many different directions. Good references are the introductory book by Papadimitriou [20] and the more recent one by Arora and Barak [1]. The first examples of truly implicit characterizations of complexity classes go back to Bellantoni and Cook [4] and Leivant [15], who introduced characterizations of complexity classes as function algebras. This approach has then been generalized to complexity classes different from $\mathsf{FP}_{\{0,1\}}$, like polynomial space functions [17] and logarithmic space functions [19]. Various restrictions on functional programs are possible, leading to characterizations of the polytime computable functions. One way to do that is by appropriately restrict path orders, like in LMPO [18]. Another is by the so-called *interpretation method*, like in polynomial interpretations [5]. The two approaches can then be mixed together to capture more programs, leading to quasi-interpretations [6]. The λ-calculus can be itself restricted in many different ways, obtaining characterizations of complexity classes. These include type systems [16,12] or logical characterizations in the spirit of linear logic [9,14,22].

References

1. Arora, S., Barak, B.: Computational Complexity: A Modern Approach. Cambridge University Press, New York (2009)
2. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge University Press (1998)
3. Barendregt, H.P.: The Lambda Calculus – Its Syntax and Semantics. North-Holland (1984)
4. Bellantoni, S., Cook, S.A.: A new recursion-theoretic characterization of the polytime functions. Computational Complexity 2, 97–110 (1992)
5. Bonfante, G., Cichon, A., Marion, J.-Y., Touzet, H.: Algorithms with polynomial interpretation termination proof. Journal of Functional Programming 11(1), 33–53 (2001)
6. Bonfante, G., Marion, J.-Y., Moyen, J.-Y.: Quasi-interpretations a way to control resources. Theoretical Computer Science 412(25), 2776–2796 (2011)
7. Cobham, A.: The intrinsic computational difficulty of functions. In: Bar-Hillel, Y. (ed.) Logic, Methodology and Philosophy of Science, Proceedings of the 1964 International Congress, pp. 24–30. North-Holland (1965)
8. Cutland, N.: Computability: An Introduction to Recursive Function Theory. Cambridge University Press (1980)

9. Girard, J.-Y.: Light linear logic. Information and Computation 143(2), 175–204 (1998)
10. Hartmanis, J., Stearns, R.: On the computational complexity of algorithms. Transactions of the American Mathematical Society 117, 285–306 (1965)
11. Hindley, J.R., Seldin, J.P.: Lambda-Calculus and Combinators: An Introduction. Cambridge University Press (2008)
12. Hofmann, M.: Linear types and non-size-increasing polynomial time computation. Information and Computation 183(1), 57–85 (2003)
13. Kristiansen, L., Niggl, K.-H.: On the computational complexity of imperative programming languages. Theoretical Computer Science 318(1-2), 139–161 (2004)
14. Lafont, Y.: Soft linear logic and polynomial time. Theoretical Computer Science 318(1-2), 163–180 (2004)
15. Leivant, D.: Stratified functional programs and computational complexity. In: Proceedings of Twentieth Annual Symposium on Principles of Programming Languages, pp. 325–333 (1993)
16. Leivant, D., Marion, J.-Y.: Lambda Calculus Characterizations of Poly-Time. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 274–288. Springer, Heidelberg (1993)
17. Leivant, D., Marion, J.-Y.: Ramified Recurrence and Computational Complexity II: Substitution and Poly-Space. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 486–500. Springer, Heidelberg (1993)
18. Marion, J.-Y.: Analysing the implicit complexity of programs. Information and Computation 183(1), 2–18 (2003)
19. Neergaard, P.M.: A Functional Language for Logarithmic Space. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 311–326. Springer, Heidelberg (2004)
20. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1993)
21. Rogers, H.: The Theory of Recursive Functions and Effective Computability. MIT Press (1987)
22. Terui, K.: Light affine lambda calculus and polynomial time strong normalization. Archive for Mathematical Logic 46(3-4), 253–280 (2007)
23. Turing, A.: On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society 2, 230–265 (1937)