GIORGI JAPARIDZE

Arithmetics based on computability logic

Giorgi Japaridze

Villanova University,

800 Lancaster Avenue, Villanova, PA 19085, USA.

Institute of Philosophy of RAS,

12/1 Goncharnaya Str., Moscow, 109240, Russian Federation.

E-mail: giorgi.japaridze@villanova.edu

Abstract: This paper is a brief survey of number theories based on *computability logic* (CoL) — a game-semantically conceived logic of computational tasks of resources. Such theories, termed *clarithmetics*, are conservative extensions of first-order Peano arithmetic.

The first section of the paper lays out the conceptual basis of CoL and describes the relevant fragment of its formal language, with so called parallel connectives, choice connectives and quantifiers, and blind quantifiers. Both syntactically and semantically, this is a conservative generalization of the language of classical logic.

Clarithmetics, based on the corresponding fragment of CoL in the same sense as Peano arithmetic is based on classical logic, are discussed in the second section. The axioms and inference rules of the system of clarithmetic named **CLA11** are presented, and the main results on this system are stated: constructive soundness, extensional completeness, and intensional completeness.

In the final section two potential applications of clarithmetics are addressed: clarithmetics as declarative programming languages in an extreme sense, and as tools for separating computational complexity classes. When clarithmetics or similar CoL-based theories are viewed as programming languages, programming reduces to proof-search, as programs can be mechanically extracted from proofs; such programs also serves as their own formal verifications, thus fully neutralizing the notorious (and generally undecidable) program verification problem. The second application reduces the problem of separating various computational complexity classes to separating the corresponding versions of clarithmetic, the potential benefits of which stem from the belief that separating theories should generally be easier than separating complexity classes directly.

Keywords: Computability logic, Peano arithmetic, game semantics, constructive logic, intuitionistic logic, linear logic, interactive computability

For citation: Japaridze G. “Arithmetics based on computability logic”, *Logicheskie Issledovaniya* / *Logical Investigations*, 2019, Vol. 25, No. 2, pp. 61–74. DOI: 10.21146/2074-1472-2019-25-2-61-74

1. Computability logic (CoL): a formal theory of computability in the same sense as classical logic is a formal theory of truth

This talk is about number theories based on *computability logic*. I will be using the abbreviation CoL for the latter. So, first of all, what is computability logic? This is an approach introduced by myself some time ago, which I characterize as a formal theory of computability in the same sense as the classical logic is a formal theory of truth. Let us compare the two logics to see what this means.

In classical logic the central semantical concept is *truth*, formulas represent *statements*, and the main utility of classical logic is that it provides a systematic answer to the questions: 1) Is P (always) *true*? 2) Does *truth* of P (always) follow from *truth* of Q ?

Now, what is computability logic and how does it differ from this? Everything is the same, except that truth is replaced with computability. So the central semantical concept here is *computability*. Formulas represent not just statements as before, but *computational problems*, as computability is a property of computational problems. And the logic provides a systematic answer to the questions: 1) Is P (always) computable? 2) Does computability of P (always) follow from computability of Q ? Moreover, it provides answers to these two questions in a constructive sense. Namely, when it establishes that P is computable, it not only merely establishes existence of a computation (algorithm) for P , but rather also tells us exactly how to compute P ; similarly it tells us how to construct an algorithm for P from an algorithm for Q .

Things are naturally arranged in such a way that classical statements (predicates, propositions) are special cases of computational problems, and classical truth is a special, simplest case of computability. This eventually makes classical logic a conservative fragment of computability logic. Conservative fragment in the sense that the language of CoL is more expressive than that of classical logic, containing the latter just as a fragment, but if we restrict the language back to that of classical logic, CoL's semantics validates nothing less and nothing more than what the classical semantics does.

Anyway, first and foremost, we want to agree on our understanding of what *computational problems* are. If you go back to Church, computational problems are just functions to be computed. But for us the understanding of computational problems is more general. Namely, a computational problem is a zero-sum game between a machine, symbolically named $\top$, and its environment, symbolically named $\perp$. Functions are just special cases of such games, but otherwise here we have computational problems of arbitrary degrees of interactivity (the problem of computing a function is not very interactive, with only two steps involved: receiving an input and generating an output).

I don't specify or define exactly what is meant by a "machine", but let us understand it as an algorithm. So, instead of a machine you can always think of an interactive algorithm, some mechanical procedure to follow. We say a machine M *wins* (*computes*, *solves*) a game/problem G iff M wins G regardless of how the environment acts. Such an M is said to be an *algorithmic solution* (algorithmic *winning strategy*) for G . A problem/game is *computable* iff it has an algorithmic solution.

CoL's logical operators (connectives, quantifiers) represent operations on games. There is a whole zoo of operators, but in this presentation we only consider the modest fragment of it consisting of

$$\neg, \wedge, \vee, \rightarrow, \forall, \exists, \sqcap, \sqcup, \sqcap, \sqcup.$$

Here we see all operators of classical logic and something in addition to that. These operators, just like all operators of CoL for that matter, are operations on games. The classically-shaped operators generalize their classical counterparts, and do so in a conservative fashion. That is in the sense that such operators automatically retain their classical meanings when applied to statements that also happen to be legitimate sentences of classical logic.

A *run* of a game is a sequence of moves, each one prefixed with $\top$ or $\perp$ to indicate which player has made the move. I am not giving any formal definitions in this presentation, but of course they do exist.

Atomic sentences such as $2 + 2 = 4$ are moveless games, that is, games whose only legal run is the empty run $\langle \rangle$. Such a game is won by the machine if the sentence is true in the classical sense, and lost if false. The same can be seen to be the case for all formulas built from atoms using only $\neg, \wedge, \vee, \rightarrow, \forall, \exists$ once we define these operators.

So, (the empty run of) $2 + 2 = 4$ or $\forall x(x = x)$ is won by the machine, and $2 + 2 = 5$ or $0 = 1 \wedge 2 = 2$ is lost. This means the problem/game $2 + 2 = 4$ or $\forall x(x = x)$ is computable (by a machine that does nothing) while $2 + 2 = 5$ or $0 = 1 \wedge 2 = 2$ is incomputable.

The bottom line to remember here is that formulas of classical logic represent moveless games. Some may ask in frustration: "how can we call something a game if there are no moves in it?!" Well, let me remind you the situation with the concept of zero. The Romans did not have this concept, and the subsequent European tradition for a long time resisted the idea of accepting zero as a legitimate number. The attitude was that a number is supposed to represent a quantity, but zero means no quantity at all, so it is not a meaningful number. But now we know that we can not do much in mathematics without zero. Similarly, the above moveless sorts of games make perfect sense under CoL's approach.

Now let us look at the operations on games that we will be dealing with.

First comes the *choice conjunction* of two games A and B , written $A \sqcap B$ and read “ A chand B ”. This is a game where the first legal move is (only) by the environment, which should choose between the two available moves “*left*” and “*right*”. After that the game continues according to the rules of the chosen component A or B , respectively. This choice is not only a privilege of the environment, but also an obligation, because if the environment fails to make a move (choice) here, then it loses, with the machine correspondingly considered to be the winner.

Choice disjunction (“*chor*”) $A \sqcup B$ is similar, with the difference that here it is the machine which makes the first move/choice and which loses if no such move is made. So the roles of the machine and the environment are interchanged here.

The *choice universal quantification* of a game $A(x)$ is in fact a “big choice conjunction” in the sense that this game, written $\sqcap x A(x)$ and read “*chall* $x A(x)$ ”, is a game where the first legal move is, just like in the case of choice conjunction, by the environment, which, however, instead of choosing between *left* and *right*, simply chooses a natural number n , after which the game continues as $A(n)$. If no such move is made, then the environment loses.

Choice existential quantification $\sqcup x A(x)$ (“*chexists* $x A(x)$ ”) is similar, with the difference that, in it, it is the machine which makes the first move and which loses if no move is made.

Here is an example. $\langle \perp 6, \top \text{left} \rangle$ is a machine-won run of the game $\sqcap x (Even(x) \sqcup Odd(x))$, and the following sequence shows how it “modifies” the game:

$$\sqcap x (Even(x) \sqcup Odd(x)) \Rightarrow Even(6) \sqcup Odd(6) \Rightarrow Even(6).$$

According to this scenario, the environment chose 6 for x , after which the game continued as $Even(6) \sqcup Odd(6)$. In response, the machine made the move *left*, meaning that the left component was chosen, so the game continued as $Even(6)$. It also ended as $Even(6)$, because no further moves were made (or could have been made). The machine won as $Even(6)$ is a true proposition, and true propositions are automatically won by the machine.

Negation $\neg$, read as “*not*”, is a role switch operation: the moves and wins of the machine become those of the environment, and vice versa. More precisely: for a run Φ , let the *negative image* of Φ mean the result of changing all labels in Φ to their opposite. For instance, the negative image of $\langle \perp 6, \top \text{left} \rangle$ is $\langle \top 6, \perp \text{left} \rangle$. Then, given a game G , $\neg G$ is the game whose legal runs are the negative images of those of G , that is, the environment and the machine have interchanged their moves, and where a given player wins a given run of the

game iff the other player wins, in the sense of G , the negative image of that run.

Example: we have $\neg \sqcap x(Even(x) \sqcup Odd(x)) = \sqcup x(Odd(x) \sqcap Even(x))$. Indeed, consider the negation of $\sqcap x(Even(x) \sqcup Odd(x))$. Without a negation this is a universally quantified game, meaning that the environment makes the first move in it. But negation interchanges players' roles, so now it is the machine that can make the first move. That is why $\sqcap x$ becomes $\sqcup x$. For similar reasons, $\sqcup$ becomes $\sqcap$, *Even* becomes *Odd* ($= \neg Even$) and *Odd* becomes *Even*.

We can see that DeMorgan's laws remain valid with choice operators. In fact, CoL has several (4+) sorts of conjunctions, disjunctions and quantifiers, and DeMorgan's laws go through for all of them.

When applied to an atomic sentence S (or any moveless game for that matter), $\neg$ behaves exactly like classical negation, because the only legal run in games represented by atomic sentences is the empty run $\langle \rangle$, and the negative image of $\langle \rangle$ is the same $\langle \rangle$; so, only the winners are interchanged, and the machine wins this "run" of S (in other words, S is true) iff the environment wins it in game $\neg S$ (that is, S is false).

The same can be seen to be the case with all operators that look like operators of classical logic: $\wedge$, $\vee$, $\rightarrow$, $\forall$, $\exists$. When applied to moveless games, the behavior of such operators is exactly classical. Their new meanings coincide with their classical meanings, and this happens naturally/automatically rather than being "manually" postulated.

Whether S is moveless or not, we have $\neg \neg S = S$. The double negation principle remains valid, because interchanging the roles twice brings each player to its original role.

The *parallel conjunction* of games A and B , written $A \wedge B$ and read "*A pand B*", is a game where A and B are played simultaneously (no choice is made between the two), and where, in order to win, the machine needs to win in both components. To indicate in which component a given move is made, it should be prefixed with "*left*" or "*right*".

Parallel disjunction ("*por*") $A \vee B$ is similar, with the only difference that here winning in just one component is sufficient.

Example: $\langle \bot_{right.6}, \top_{left.6}, \bot_{left.left}, \top_{right.left} \rangle$ is a legal run of the game

$$\sqcup x(Odd(x) \sqcap Even(x)) \vee \sqcap x(Even(x) \sqcup Odd(x)).$$

The meaning of the first move $\bot_{right.6}$ is that the environment chooses 6 for x in the right $\vee$ -disjunct. As a result, the game turns it into

$$\sqcup x(Odd(x) \sqcap Even(x)) \vee (Even(6) \sqcup Odd(6)).$$

The move $\top left.6$ made by the machine in response signifies choosing the same number 6 for x in the left $\vee$ -disjunct, and the game correspondingly continues as

$$(Odd(6) \sqcap Even(6)) \vee (Even(6) \sqcup Odd(6)).$$

The next move $\perp left.left$ by the environment, whose meaning is choosing the left $\sqcap$ -conjunct in the left $\vee$ -disjunct, further brings the game down $Odd(6) \vee (Even(6) \sqcup Odd(6))$. The final move $\top right.left$ made by the machine turns it into $Odd(6) \vee Even(6)$. The machine wins because $Even(6)$ is true and thus it wins in one of the two $\vee$ -disjuncts.

Parallel implication (“*pimplication*”) $\rightarrow$ is simply understood as a standard abbreviation: $A \rightarrow B =_{df} \neg A \vee B$. Due to the prefix $\neg$, the players’ roles are interchanged in A , turning it into a *computational resource* that can be used by $\top$ in solving B . The computational intuition associated with this combination is the most interesting one as, intuitively, $A \rightarrow B$ can be seen to be the problem of *reducing* B to A . Here is an example. Consider the following two predicates:

- $Halts(x, y)$ = “Turing machine x halts on input y ”;
- $Accepts(x, y)$ = “Turing machine x accepts input y ”.

The renowned Halting problem as a decision problem can be written this way: $\sqcap x \sqcap y (Halts(x, y) \sqcup \neg Halts(x, y))$. In this game the environment chooses some particular values for x and y , and the machine should reply by telling whether x halts on y ($\top left$) or not ($\top right$). Similarly for the Acceptance problem $\sqcap x \sqcap y (Accepts(x, y) \sqcup \neg Accepts(x, y))$. Both problems are known to be undecidable. But the pimplication from one to the other can be shown to be computable.

$$\underbrace{\sqcap x \sqcap y (Halts(x, y) \sqcup \neg Halts(x, y))}_{\text{Halting problem}} \rightarrow \underbrace{\sqcap x \sqcap y (Accepts(x, y) \sqcup \neg Accepts(x, y))}_{\text{Acceptance problem}}$$

Reduction of the acceptance problem to the halting problem

The winning strategy here relies on reducing the consequent to the antecedent. Roughly it goes like this. While both players have initial legal moves in this game, the machine waits until the environment makes a choice of some

values m and n for x and y in the consequent (otherwise it loses) and thus brings the game down to

$$\Box x \Box y (Halts(x, y) \sqcup \neg Halts(x, y)) \rightarrow Accepts(m, n) \sqcup \neg Accepts(m, n).$$

Then the machine chooses the same m and n in the antecedent, so the game continues as

$$Halts(m, n) \sqcup \neg Halts(m, n) \rightarrow Accepts(m, n) \sqcup \neg Accepts(m, n).$$

Now, again, both players have legal moves: $\top$ in the consequent and $\perp$ in the antecedent, but it is wise for the machine to wait to let the environment move first. If the environment fails to make a move, the environment loses in the antecedent and thus the machine wins the overall game. So, assume the environment makes a move in the antecedent, telling whether m halts on n or not. We may assume that whatever the environment says here is true, or else it loses. If it says that m does not halt on n , then the machine chooses $\neg Accepts(m, n)$ in the consequent and wins because if m does not halt, then it does not accept either. Otherwise, if the environment says that m halts on n , the machine simulates the work of m on input n , and this simulation will show whether the halting was with acceptance or rejection. The machine correspondingly chooses between $Accepts(m, n)$ and $\neg Accepts(m, n)$ in the consequent and wins.

What we call *blind quantifiers* are $\forall$ (“*blall*”) and $\exists$ (“*blexists*”). $\exists$ is the DeMorgan dual of $\forall$ ($\exists = \neg \forall \neg$), so let us just look at the blind universal quantification $\forall x A(x)$. Unlike $\Box$, no move is associated with $\forall$, i.e., no value for x is specified (by either player); in order to win, the machine needs to play “blindly” in a way that guarantees success in $A(x)$ for every possible value of x .

As an example, consider the game

$$\forall x (Even(x) \sqcup Odd(x) \rightarrow \Box y (Even(x + y) \sqcup Odd(x + y))).$$

The following scenario unfolds according to the run $\langle \perp right.5, \perp left.right, \top right.left \rangle$: The environment chooses 5 for y in the consequent, bringing the game down to

$$\forall x (Even(x) \sqcup Odd(x) \rightarrow Even(x + 5) \sqcup Odd(x + 5)).$$

The machine has to tell whether $x + 5$ is even or odd. But it does not know x . Can it still determine the parity of $x + 5$? Yes, it can, if the machine just knows whether x is even or odd without otherwise knowing what particular number x is. And this information about the parity of x can be obtained from the antecedent, where the environment is obligated to move. By making the move $\perp left.right$ and bringing the game down to $\forall x (Odd(x) \rightarrow Even(x + 5) \sqcup$

$Odd(x + 5))$, the environment says that x is odd. But then $x + y$ is even, so the final move $\top right.left$, resulting in $\forall x(Odd(x) \rightarrow Even(x + 5))$, makes the machine the winner. Notice how $\forall$ persisted in the above sequence of games, and so did $\rightarrow$. Generally, the classical structure of a game when it evolves in this fashion will remain unchanged, all changes will happen exclusively in choice ($\sqcap, \sqcup, \sqcap, \sqcup$) components.

2. Clarithmetics: formal number theories based on CoL

In CoL, the standard concepts of time and space complexities are naturally and conservatively generalized to the interactive level, making them meaningful for games. Also, a new complexity measure — *amplitude complexity* — is introduced. It is concerned with the sizes of $\top$'s moves relative to the sizes of $\perp$'s moves.

Instead of computability-in-principle, we can now just as meaningfully speak about computability with limited resources (amplitude, space, time). For instance, polynomial space computability rather than just computability.

Example: each of the problems in the left column below is polynomial time computable. The right column tells us what this means in standard terms.

$\sqcap x \sqcap y (x = y \sqcup x \neq y)$	“=” is polynomial time decidable
$\sqcap x \sqcap y \sqcup z (z = x + y)$	“+” is polynomial time computable
$\sqcap x (\exists y (x = y \times y) \rightarrow \sqcup y (x = y \times y))$	“square root”, when exists, is polynomial time computable
$\sqcap x \sqcup y (p(x) \leftrightarrow q(y))$	p is polynomial time reducible to q
$(A \leftrightarrow B \text{ abbreviates } (A \rightarrow B) \wedge (B \rightarrow A))$	

Clarithmetics are formal number theories based on CoL in the same sense as **PA** (Peano arithmetic) is based on classical logic, usually in the language whose logical vocabulary is the one that we surveyed in the preceding section: $\neg, \wedge, \vee, \rightarrow, \forall, \exists, \sqcap, \sqcup, \sqcap, \sqcup$. The non-logical vocabulary is standard: $0, ', +, \times, =$ (x' means $x + 1$). We need some technical preliminaries before we proceed.

- By a *bound* we mean a *p*term (term or pseudoterm) for some monotone function (the term was coined by George Boolos). Pseudoterms are not really terms strictly speaking, but one can pretend that we do have special terms for them in the language. Nothing will change this way because

each formula with pseudoterms can be equivalently rewritten as one with only real terms. Terminologically we identify a pterm with the function it represents.

- A *boundclass* is a set of bounds closed under variable renaming.
- A *tricomplexity* is a triple $\mathbf{R} = (\mathbf{R}_{\text{amplitude}}, \mathbf{R}_{\text{space}}, \mathbf{R}_{\text{time}})$ of boundclasses satisfying certain *regularity* conditions. I will not show here those conditions, but they are natural. Suffice it to say that all naturally emerging tricomplexities are regular as long as $\mathbf{R}_{\text{amplitude}}$ is at least linear (contains all linear functions), $\mathbf{R}_{\text{space}}$ at least logarithmic, and $\mathbf{R}_{\text{time}}$ at least polynomial.
- An $\mathbf{R}$ *tricomplexity solution* of a given problem G is a solution for G whose amplitude complexity is upper-bounded by some bound from $\mathbf{R}_{\text{amplitude}}$, space complexity by some bound from $\mathbf{R}_{\text{space}}$, and time complexity by some bound from $\mathbf{R}_{\text{time}}$. In other words, this is a $\top$'s winning strategy/algorithm that runs in time from $\mathbf{R}_{\text{time}}$, in space from $\mathbf{R}_{\text{space}}$ and in amplitude from $\mathbf{R}_{\text{amplitude}}$.
- $|x|$ is the standard arithmetization of function “the length of the binary representation of x ”, i.e., of “ $\lceil \log_2(x+1) \rceil$ ”.
- $\text{Bit}(x, y)$ is the standard arithmetization of the predicate “the y 'th least significant bit of the binary representation of x is 1”, i.e., of “ $\lfloor x/2^y \rfloor \bmod 2 = 1$ ”.
- $\underline{s}$ (resp. $|\underline{s}|$) means a tuple $(s_1, \dots, s_n)$ (resp. $(|s_1|, \dots, |s_n|)$), where $s_1, \dots, s_n$ are variables.
- Let B be a boundclass. We say that a formula F is *B-bounded* iff every $\sqcap$ -subformula of F has the form $\sqcap x(|x| \leq b(|\underline{s}|) \rightarrow F)$ and every $\sqcup$ -subformula has the form $\sqcup x(|x| \leq b(|\underline{s}|) \wedge F)$, where $x, \underline{s}$ are pairwise distinct variables not bound by $\forall, \exists$ in F , and b is a bound from B .

A number of systems of clarithmetic were developed for various purposes, corresponding to various complexity classes. In this presentation we will look only at one system of clarithmetic, called **CLA11**. It is a scheme of clarithmetical theories rather than a particular theory, generating a particular theory **CLA11^R** per each tricomplexity $\mathbf{R}$.

We fix some tricomplexity $\mathbf{R} = (\mathbf{R}_{\text{amplitude}}, \mathbf{R}_{\text{space}}, \mathbf{R}_{\text{time}})$ to define the corresponding theory **CLA11^R**. The logical basis (axioms, rules) of such a theory is a certain axiomatization **CL12** of CoL, sound and complete in a very

strong sense. It is the logical basis of **CLA11^R** in the same sense as classical logic is a logical basis of Peano arithmetic. Having said that, here we shall only focus on the nonlogical postulates of **CLA11^R**.

The nonlogical *axioms* of **CLA11^R** are the following:

1. $\forall x \neg(0 = x')$
2. $\forall x \forall y (x' = y' \rightarrow x = y)$
3. $\forall x (x + 0 = x)$
4. $\forall x \forall y (x + y' = (x + y)')$
5. $\forall x (x \times 0 = 0)$
6. $\forall x \forall y (x \times y' = (x \times y) + x)$
7. The $\forall$ -closure of $F(0) \wedge \forall x (F(x) \rightarrow F(x')) \rightarrow \forall x F(x)$ for each $\Box, \sqcup, \sqcap, \sqcup$ -free formula F
8. $\Box x \sqcup y (y = x')$
9. $\Box x \sqcup y (y = |x|)$
10. $\Box x \Box y (Bit(x, y) \sqcup \neg Bit(x, y))$

Formulas (1)–(7) are nothing but the so-called Peano axioms, that is, axioms of standard **PA**. Of course, (7) is a scheme of axioms, generating infinitely many particular axioms. In addition, we have three extra-Peano axioms (8)–(10). The first one expresses the computability of the successor function. The second one expresses the computability of the logarithm function. The third one says that the question of telling whether the y 'th least significant bit of (the binary representation of) x is 1 or not is decidable.

Along with **CLA11^R** we also consider **CLA11^R + TA**, where **TA** stands for “Truth Arithmetic”. **CLA11^R + TA** differs from **CLA11^R** in that its axioms include all true sentences of classical arithmetic (rather than just Peano axioms) as additional axioms. Of course, this system, unlike **CLA11^R**, is no longer recursively axiomatizable.

On top of axioms, **CLA11^R** has the following two nonlogical *rules* of inference:

$$\bullet \quad \text{Induction} \quad \frac{F(0) \quad \Box x (F(x) \rightarrow F(x'))}{\Box x \leq b(|\underline{s}|) F(x)}$$

where x and $\underline{s}$ are pairwise distinct variables, $F(x)$ is an **R_{space}**-bounded formula, and b is a bound from **R_{time}**;

- *Comprehension*
$$\frac{\Box y(p(y) \sqcup \neg p(y))}{\sqcup |x| \leq b(|s|) \forall y < |x| (Bit(y, x) \leftrightarrow p(x))}$$

where x , y and s are pairwise distinct variables, $p(y)$ is a $\Box, \sqcup, \Box, \sqcup$ -free formula not containing x , and b is a bound from $\mathbf{R}_{amplitude}$.

As we see, the above induction rule, unlike its standard counterpart, has $\Box$ instead of $\forall$. The intuition here is that, if we know how to compute $F(0)$ and we also know, for all x with $x \leq b(|s|)$, how to reduce $F(x')$ to $F(x)$, then we know how to compute $F(x)$. A restriction is that $F(x)$ should be an $\mathbf{R}_{space}$ -bounded formula and the bound b should be a bound taken from $\mathbf{R}_{time}$.

The intuition associated with the comprehension rule is that, if you can decide the predicate p , then you can generate (hence, *chexists*) the number x such that, for any n , the n -th least significant bit of x is a 1 if and only if p is true of n . The decidable predicate p thus generates its own number in the sense that it tells us each bit of the binary representation of x . Again, we have a restriction here, according to which b should be a bound from $\mathbf{R}_{amplitude}$.

Now we are ready to state the main result below after couple of additional terminological conventions. By an *arithmetical problem* we mean a problem/game that is expressed by some formula of the language of $\mathbf{CLA11}^{\mathbf{R}}$. We say that such a problem is *representable* in $\mathbf{CLA11}^{\mathbf{R}}$ if it is expressed by some theorem of $\mathbf{CLA11}^{\mathbf{R}}$. Note that the same arithmetical problem can be expressed by many different formulas and, in order for the problem to qualify as representable, it is sufficient that just one of such formulas be provable.

Theorem 1. *For any tricomplexity $\mathbf{R}$, the following holds:*

- **Constructive soundness:** *For every theorem F of $\mathbf{CLA11}^{\mathbf{R}} + \mathbf{TA}$, the problem expressed by F has an $\mathbf{R}$ tricomplexity solution, and such a solution can be automatically extracted from a proof of F .*
- **Extensional completeness:** *Every arithmetical problem with an $\mathbf{R}$ tricomplexity solution is representable in $\mathbf{CLA11}^{\mathbf{R}}$.*
- **Intensional completeness:** *Every formula expressing a problem with an $\mathbf{R}$ tricomplexity solution is provable in $\mathbf{CLA11}^{\mathbf{R}} + \mathbf{TA}$.*

The $\mathbf{R}$ parameter of $\mathbf{CLA11}^{\mathbf{R}}$ can be turned in a mechanical, brute force, “canonical” way to obtain a sound and complete theory with respect to the target computational tricomplexity. For instance, for

linear amplitude + logarithmic space + polynomial time,

we can choose the $\mathbf{R}$ with

$\mathbf{R}_{amplitude} = \{\text{all terms built from } x \text{ using } 0, ', +\}$ — you can see that these are exactly the terms that describe all linear functions.

$\mathbf{R}_{space} = \{\text{all pterms built from } |x| \text{ using } 0, ', +\}$ — this can be seen as the set of pterms that express all logarithmic functions.

$\mathbf{R}_{time} = \{\text{all terms built from } x \text{ using } 0, ', +, \times\}$ — again, you can see that these are exactly those terms that express polynomial functions.

In a similar fashion, with very little effort, you can generate instances of **CLA11** for a huge variety of tricomplexities. All reasonable complexity triples can be captured this way, such as:

- *polynomial amplitude + polynomial space + polynomial time*
- *linear amplitude + linear space + quasipolynomial time*
- *linear amplitude + polynomial space + exponential time*
- *superexponential amplitude + elementary space + primitive recursive time*

... you just name it!

3. Potential applications of clarithmetics

The two main motivations for studying **CLA11** and clarithmetics in general are that (1) such theories can be seen as declarative programming languages in an extreme sense, and (2) they can potentially be used (perhaps, hopefully) as a tool for separating computational classes.

(1) **CLA11** as a declarative programming language

Let us say you want a program computing the integer square root function, such that this program runs in polynomial time, logarithmic space and linear amplitude. All you need to do to get such a program is this:

1. Instantiate the $\mathbf{R}$ parameter with the corresponding tricomplexity (cf. the end of Section 2.);
2. Write the formula $\Box x(\exists y(x = y \times y) \rightarrow \Box y(x = y \times y))$ expressing your goal in the language of $\mathbf{CLA11}^{\mathbf{R}}$;
3. Find a $\mathbf{CLA11}^{\mathbf{R}}$ -proof of this formula.

Once you find a proof, the compiler then extracts the sought program from it. With $\Box x(\exists y(x = y \times y) \rightarrow \Box y(x = y \times y))$ being in fact a specification of such a program, the proof automatically also serves as a formal verification of

the fact that the program meets its specification. This way, the notorious and generally undecidable problem of program verification is fully neutralized.

Other than that, the proof lines can be seen as (the best possible) comments for the corresponding steps of the program.

Furthermore, — this is from the realm of fiction at this point, but still — if we develop reasonably efficient theorem-provers, step 3 (finding the proof) can be delegated to the compiler, and the “programmer”’s job will be just to write the goal/specification $\Box x(\exists y(x = y \times y) \rightarrow \Box y(x = y \times y))$. All of us can thus qualify as programmers, because we all can very easily understand the language in which such a formula is written.

(2) **CLA11** as a tool for separating computational complexity classes.

The main open problems in the theory of computation are about separating various naturally emerging computational complexity classes, with the P versus NP problem being the best known example of problems of this kind. To show that two (tri)complexity classes **R1** and **R2** are not equal, it is sufficient to separate the two theories **CLA11**^{R1} and **CLA11**^{R2}. Separating theories should to be easier than separating complexities directly. After all, we have seen some very impressive success stories of separating theories, such as separating different versions of set theory or proving independence results. We do not have comparable success stories in separating complexities, which mostly has been just a fruitless struggle so far.

An extensive online survey of CoL and CoL-based applied theories can be found at

www.csc.villanova.edu/~japaridz/CL/.

Most relevant publications: [Japaridze 2015], [Japaridze 2016c], [Japaridze 2014], [Japaridze 2015], [Japaridze 2016a], [Japaridze 2016b], [Japaridze 2016c].

Acknowledgements. Paper is RAS Institute of Philosophy Seminar Talk, 14 November 2018 (See video <https://youtu.be/w9-1Xm8MMmQ>).

References

- Japaridze 2010 – Japaridze, G. “Towards applied theories based on computability logic”, *Journal of Symbolic Logic*, 2010, Vol. 75, pp. 565–601.
- Japaridze 2011 – Japaridze, G. “Introduction to clarithmetic I”, *Information and Computation*, 2011, Vol. 209, pp. 1312–1354.
- Japaridze 2014 – Japaridze, G. “Introduction to clarithmetic III”, *Annals of Pure and Applied Logic*, 2014, Vol. 165, pp. 241–252.
- Japaridze 2015 – Japaridze, G. “On the system CL12 of computability logic”, *Logical Methods in Computer Science*, 2015, Vol. 11, No. 3, Paper 1, pp. 1–71.

-
- Japaridze 2016a – Japaridze, G. “Build your own clarithmetic I: Setup and completeness”, *Logical Methods in Computer Science*, 2016, Vol. 12, No. 3, Paper 8, pp. 1–59.
- Japaridze 2016b – Japaridze, G. “Build your own clarithmetic II: Setup and completeness”, *Logical Methods in Computer Science*, 2016, Vol. 12, No. 3, Paper 12, pp. 1–62.
- Japaridze 2016c – Japaridze, G. “Introduction to clarithmetic II”, *Information and Computation*, 2016, Vol. 247, pp. 290–312.