Limit Computable Mathematics and Interactive Computation

Susumu Hayashi, Kobe University

September 17, 2001

Abstract

We are now investigating an "executable" fragment of classical mathematics for *testing formal proofs* to make formal proof developments less laborious. Several theories of execution of full classical proofs are known. In these theories, some kind of abstract values such as continuations, are necessary. It makes them illegible from computational point of view, although they are mathematically interesting. In contrast, we consider only a fragment of classical mathematics and give a simple and natural "computational" contents without such abstract values. The fragment appears to cover a rather large domain of practical mathematics. The point is that codes associated to proof by our method is *not* computable in Turing's sense, i.e., Δ^0_1, but "executable" in the sense of Gold's theory of machine learning, i.e., Δ^0_2. I will give a survey of this new executable mathematics LCM. I will also discuss a possible framework of "interactive computation" emerged from LCM research. It may serve a unified framework for practical scientific computing such as numerical analysis and problem solving in computer science such as verification via refinements. Many materials of this paper is complied from the manuscript [6].

1 Proof Animation: testing proofs

The formal development of proofs is becoming more and more practical thanks to advancements of hardware and software. However, the task is still very costly. There are several reasons why the task is so heavy. One of them and probably most serious one is mismatches between formalized notions and informal notions. There would not be ultimate solution for this problem. However, we introduced a notion of *Proof Animation* (PA) to make this problem less problematic [5]. In a sense, PA is a contrapositive of the good old proofs as programs notion "if a program is (extracted from) a proof checked, then it does not have bugs". For PA, the objective is not correct programs but checked proofs. Thus, we say "if a program extracted from a proof has a bug, then the proof does have a bug". The reader may wonder why a formal proof has a bug. Here proofs mean partial proofs under development and so they may have bugs in subgoals.

Note that even a finished and checked proof may have a bug. Proof checking do not detect wrong formalizations. Formalizations themselves can be wrong. In Viper project, bugs were found in *specifications* rather than implementation, and they were found only through conventional validation or testing methods (see [3]). In practice of formal development of proofs, it is not very rare that everything is proved but then one finds the conclusion just proved is short for actual requirements.

No *formal* metrics of correctness of formalizations would exist. Thus, quite often we do *not* know what are our real requirements at the very beginning of project. They are found only through the process refining our knowledge on the system under development. It is desirable to fix such requirements as early as possible. However, often they are found only in the very last stage of the project or even after the project is finished.

Formal proof developments (and formal software development) is a kind of problem solving and thus it would be an inevitable nature of the activity. Thus, we cannot and should not avoid some kind of empirical means in formalization of proofs, since "correctness" of specifications, definitions and goals are defined only through informal criteria. The notion of "proof animation" or "testing proofs" [5] is an example of such an informal means. The idea is finding bugs in proofs by testing programs associated to proofs.

In the talk, the method of proof animation will be illustrated by logic puzzles in [5] and [6]. We will not get into the details here, but point out that associated programs are actually useful to find bugs.

2 Limit Computable Mathematics

We can use Curry-Howard isomorphism to animate proofs as pointed out in [5] and the demo in the talk illustrates. However, it is applicable only to constructive proofs and many proofs are not constructive. Thus, we need a method for animation (execution) of such classical proofs. Since proof animation is a means to understand proofs through computations, such proof execution methods or interpretation must satisfies the following two criteria:

1. computational contents (programs) associated with proofs are legible,
2. association between proofs and programs is legible.

We call a proof execution method with these criteria *accountable.*

Unfortunately, almost all proof execution methods for classical logic are not accountable. Since there are many classical proofs, such as proofs of Banach-Tarski paradox, in which we cannot find computational contents, there might not be truly accountable proof execution methods for *full* classical mathematics. Then we should seek an alternative approach. Find a *fragment* F of classical mathematics such that proof execution for F is accountable and enough mathematics can be done in F.

Limit Computable Mathematics (LCM) was introduced as a candidate of such a fragment. LCM *is a fragment of classical mathematics whose Brouwer-Heyting-Kolmogorov-Kleene-interpretation is realized by limiting recursive functions rather than recursive functions.* Since, it is realized by limiting computable (limiting recursive) functions, it is called Limit Computable Mathematics. LCM may have some different formulations. Already three approaches [7, 1, 2] have been proposed, and there would be more. However, they are essentially the same idea which can be illustrated by the following simple non-constructive theorem.

Proposition 1 *If f is a computable function on natural numbers, then f has a minimum value, that is, $\exists x \forall y. f(x) \leq f(y)$ holds.*

This theorem is not constructive since there is no recursive function computing the answer y from f. However, we can always find the correct answer for y in finite time by a "computation." Imagine an agent is trying to compute the answer. First, it guesses $f(0)$ would be the answer. However, if it encounters a smaller value $f(i)$, it changes mind and guesses $f(i)$ would be the answer. It continues to try the remaining inputs in the same way and *never* stop. Since the numbers guessed are decreasing, it eventually encounter the right answer. After then, it will never change its mind, since it has already *learned* the right answer. However, it does not know when it got the right answer.

In the same vein, we can compute (identify) the consistency of any formal system in the limit. First, you bet to the consistency and say "the system is consistent", but you also start to check consistency by inspecting the conclusions of all possible proofs. If you find a conclusion is a contradiction, then you change your mind and say "I got a second thought. It's contradictory.". No matter of consistency of the system, you can say the right answer in a finite time. If the system is consistent, your first utterance is correct and you never change the answer. If the system is not consistent, your second utterance is correct and you never change what you said anymore. In both case, you would utter the correct answer in finite time. Of course, you never know when you utter it.

This is the fundamental paradigm of computational learning theory introduced by Gold [4]. All contemporary learning theory may be regarded as descendants of this seminal work. In the terminology of learning theory, the minimum value and consistency of formal systems are said *learned*. We now wish to apply Gold's idea to logic and Proof Animation by regarding the learning process as a kind of computation.

Kleene called a disjunctive statement $A \vee B$ is realized, if there is an algorithm deciding whether A or B is correct.[1] On the other hand, it is said a function $g(x,n)$ computes a value v *in the limit*, if $g(x,n)$ attains v for any n bigger than a number N. It is also said the sequence $g(x,0), g(x,1), g(x,2), \cdots$ converges to v and written as $v = \lim_n g(x,n)$. If $f(x) = \lim_n g(x,n)$ holds for a recursive function g, then the function f is called limiting recursive. These are the fundamental notions of algorithmic learning theory by Gold [4].

From the examples above, it would be clear that the law of excluded middle (LEM) $A \vee \neg A$ for any Π^0_1-statement A is realized in Kleene's sense by a limiting recursive function. Thus, Π^0_1-LEM is realizable by limiting recursive functions. We have shown in [7] that any classical proof whose LEM is restricted to Π^0_1-LEM and some other related principles, such as Δ^0_2-double negation elimination, are realiz-

[1] This explanation is not correct, when A and B contains disjunctions or existential quantifiers. If they contains such logical signs, the algorithm must return realizers of A and B as well.

able by limiting recursive functions and have pointed out that a large portion of mathematics needs only such a restricted non-constructive principles. A prime example was Hilbert's finite basis theorem, which was considered highly non-constructive by 19th century mathematicians and was even called "not mathematics but theology" by P. Gordan, a reputed algebraist of the time.

It is expected that LCM covers a very large part of practical mathematics for computer science and applied mathematics. (See [6] for more detailed discussions.) Then, we will be able to make proof animation practical.

Note that proof animation by LCM resembles Shapiro's algorithmic debugging, which is also based on algorithmic learning theory. By algorithmic debugging, Prolog programs are debugged. By LCM, (partial) proofs are debugged. Since executions of Prolog is a kind of proof building, it is likely that PA by LCM is related to algorithmic debugging. However, exact relationship has not been known.

Besides this possible relation to algorithmic debugging, it has been known that LCM is related to various research subjects including computability theory over real numbers (computable analysis), computer algebra, reverse mathematics, etc. (see [6]). We will briefly discuss them in the talk. This versatility would be implication of the universality of Gold's paradigm of limit computation. In the next section, we will examine a notion of interactive computation which is a generalization of Gold's paradigm.

3 Interactive computation

Limiting computation has been regarded as a paradigm of learning. However, I propose to regard it or its extension as a paradigm of practical computing. Practical computatings are often not completely automatic but some kind of problem solvings are involved. Thus, Gold's paradigm seems to fit better to the real world rather than Turing's.

To begin with, we compare Gold's paradigm of limiting computation and Turing's paradigm of computation. Turing machine may be regarded as a machine to keep computation forever. There is no structural reason to exclude infinitely long computations in the architecture of Turing machine. Finite automaton has a finite tape and the head runs in one direction. Thus, it inevitably stops. On the other hand, Turing machine has an infinite tape and the head moves to the both sides or may stay on the current cell. Turing machine is more natural to be conceived as a machine of non-stopping computation as actual computers. In this respect, the notion of the final states of Turing machine looks somehow artificial.

On the other hand, Gold's limiting recursive function is computed by a non-stopping Turing machine. It is convenient to think a machine with a separate output tape. Thus input is set on the main tape and it mainly computes on it, then outputs on the output tape. The machine never stop and does not need the notion of the final state. However, it is said that *the output* $c_1c_2\cdots c_n$ *is computed*, when the output persists on the output tape forever from a time on.

On these considerations, we may say as follows:

Turing: The machine runs to compute the answer forever. However, the machine can decide when the answer is obtained. Thus, it may stop when the machine get to know the answer is obtained.

Gold: The machine runs to compute the answer forever. The machine itself does not know when the correct answer is obtained, but eventually it encounters to a correct answer and the output becomes stationary. Thus, an Π^0_1-oracle can decide when the correct output is obtained.

These situation resembles practical scientific computations. Imagine that we are computing gcd of two integers by a program. We can program when the program should stop. However, for some scientific computation, we cannot program when it should stop. Imagine we are drawing a fractal image by a program. A fractal is an infinite being and we cannot complete the image in finite time. Thus the program is stopped by human on his/her decision, when he/she guesses the output image is enough saturated. A computation of fractal is done by a pair of non-stopping machine and an oracle telling when the machine should stop.

By generalizing these situations, a notion of interactive computation is introduced. *A system of interactive computation* consists of

computer: a non-stopping computation mechanism, e.g. non-stopping Turing Machine,

oracle: an oracle to decide when a require answer is obtained and (may stop), e.g., the mechanism itself, human being, Amida-

Butsu, God,...[2]

Interactive computation captures the following "computation paradigms":

1. Turing computation:

   computer: non-stopping Turing Machine

   oracle: the computer

2. Gold computation:

   computer: non-stopping Turing Machine

   oracle: Π^0_1-oracle

3. Fractal computation:

   computer: non-stopping graphics program

   oracle: human

4. Interactive proof protocol

   computer: Prover

   oracle: Verifier

5. Numerical analysis (simple case): The software (or algorithm) is fixed, but the precision is increased till a right approximated answer is obtained.

   computer: numerical analysis software

   oracle: human

Note that the fractal computation and the last two examples are not automatic computation but a sort of problem solving. In real scientific computations, sometimes even software or algorithm must be changed on the result of a run of the software. From theoretical point of view, all possible changes are recursive enumerable and can be automatized. However, it is not realistic. Thus, it is more natural to introduce nondeterminism. By allowing nondeterministic computer in interactive computation, we can modelize problem solving such as refinements in formal methods and real scientific computations.

This paradigm is rather broad and, without restricting the classes of computer and oracle, it would be an abstract nonsense. However, the framework must be helpful to understand computations in real life and the relationship of LCM to them.

[2]Butsu means Budda.

References

[1] Akama, Y.: Limiting Partial Combinatory Algebras: Towards Infinitary Lambda-calculi and Classical Logic, in Computer Science Logic, 15th International Workshop, CSL 2001, L. Fribourg (Ed.), Lecture Notes in Computer Science 2142, Springer, 2001.

[2] Berardi, S.: Classical logic as Limit Completion, -a constructive model for non-recursive maps-, submitted, 2001

[3] Cohn, A.: The notion of proof in Hardware Verification, Journal of Automated Reasoning, **5** (1989) 127–140

[4] Gold, E. M.: Limiting Recursion, The Journal of Symbolic Logic, **30** (1965) 28–48

[5] Hayashi, S., Sumitomo, R. and Shii, K.: Towards Animation of Proofs - Testing Proofs by Examples -, Theoretical Computer Science, to appear, available at PA/LCM homepage.

[6] Hayashi S, and Nakata M.: Towards Limit Computable Mathematics, invited talk at TYPES 2000 and submitted to its proceedings, available at PA/LCM homepage, 2001.

[7] Nakata, M. and Hayashi, S.: Realizability Interpretation for Limit Computable Mathematics, submitted, 2000