Theory of Computation

Now and forever

Jakob Grue Simonsen
Inaugural lecture
Department of Computer Science
Theory of computation concerns the mathematics governing computing in this universe, and in other universes you may dream of.

The lecture will cover why this is sometimes relevant to the audience, why it is often irrelevant, and why the latter is a good thing.

The final part of the lecture will concern the particular professorship inaugurated, and how I intend to use it, for the greater good.
PART I: THEORY OF COMPUTATION
Theory of computation concerns the mathematics governing computing in this universe, and in other universes you may dream of.

The lecture will cover why this is sometimes relevant to the audience, why it is often irrelevant, and why the latter is a good thing.

The final part of the lecture will concern the particular professorship inaugurated, and how I intend to use it, for the greater good.
The mathematics governing computing (in this universe, mostly)

- Multiple facets: physics; applied mathematics on hardware; numerical analysis; and many more.
- In computer science, “theory of computation” usually refers to a specific facet: the study of what computers can compute, and how efficiently they can compute it.
- Conversely: also the study of what computers cannot compute, or how inefficient they are sometimes forced to be.
But a computer is not merely what you think!

To your right are several kinds of logic gates. Your computers are (roughly) made of large numbers of these.

Logic gates are only defined via their input-output functions, not the substrate they are made from.

This property is called extensionality – we will return to it later.
As soon as the underlying principles are understood, many seemingly innocuous phenomena can be used to build computers.
Robust Soldier Crab Ball Gate

Yukio-Pegio Gunji
Yuta Nishiyama
Department of Earth and Planetary Sciences
Kobe University
Kobe 657-8501, Japan

Andrew Adamatzky
Unconventional Computing Centre
University of the West of England
Bristol, United Kingdom

Soldier crabs *Mictyris guinotae* exhibit pronounced swarming behavior. Swarms of the crabs are tolerant of perturbations. In computer models and laboratory experiments we demonstrate that swarms of soldier crabs can implement logical gates when placed in a geometrically constrained environment.

1. Introduction

All natural processes can be interpreted in terms of computations. To implement a logical gate in a chemical, physical, or biological spatially extended medium, Boolean variables must be assigned to disturbances, defects, or localization traveling in the medium. These traveling patterns collide and the outcome of their collisions are converted into resultant logical operations. This is how collision-based computers work [1, 2]. Now, classical examples of experimental laboratory unconventional computing include the Belousov-Zhabotinsky (BZ) medium and the slime mold of *Physarum polycephalum*. In BZ, excitable medium logical variables are represented by excitation waves that interact with each other in the geometrically constrained substrate or “free-space” substrate [3-6]. Slime mold is capable of solving many computational problems, including maze and adaptive networks [7, 8]. In the case of ballistic computation [9], slime molds implement collision computation when two slimes are united or avoid each other dependent on the gradient of attractor and inhibitor. We previously suggested that the slime mold logical gate is robust against external perturbation [2, 10]. To expand the family of unconventional spatially extended computers, we studied the swarming behavior of soldier crabs *Mictyris guinotae* and found that compact propagating groups of crabs emerge and endure under noisy external stimulation. We speculated that swarms can behave similarly to billiard balls and thus implement basic circuits of collision-based computing. The results of our studies are presented in this paper.

*Complex Systems*, 20 © 2011 Complex Systems Publications, Inc.
On a less expensive note:

• There are multiple *kinds* of computers all of which can, in general, *compute* things, but in physically very disparate ways.

• Some less exotic (non-crab, non-planet) examples: biological computers, quantum computers.

• All of them are subject to the same limits to *what* they can compute.

• And all of them have a notion of a *program*: a set of instructions that describe how a particular computation should be performed.
PART II
THE PATH OF NOW AND FOREVER
Theory of computation: what we actually do

We seek to prove mathematical statements -“theorems”- that ideally concern all these notions of computation.

The validity of these is eternal; and should hold for any computing device imaginable.

This goal is similar across all the branches of the theory of computation.

At DIKU variations of this goal are pursued by many people in quite different ways—go talk to your local algorithms or programming language researchers for a taste.

(If we can. Sometimes we must go after smaller targets.)

(but someone may come up with a generalization or a better result)
A venerable pursuit:

• The seminal paper of computer science as a field.

• Part of an early 20th century movement in metamathematics.

• In Turing’s time, a “computer” was usually a specialist in a back office working through an algorithm on paper.

• Hence, the modern device called “computer” was called “electronic computing machine” by Turing.

• Stated a time-honoured tradition of programs-as-data: programs can have other programs as input.
• The breakthroughs in the 1930s stood upon the shoulders of a history of using formal systems to reason about other formal systems.

• Operationalizing this process – roughly, finding out how to perform the computations needed to do so had been conceived by a number of scholars who devised what we today would see as proto-programming languages.

• More vague ideas about using systems of thought to reason about other systems of thought can be found much earlier – if you squint and are generous, the inklings can be traced back to (at least) the middle ages.
Computer Science is a *science* with a *history* spanning 80 years and a *historical roots* spanning centuries.

The fact that you and everyone else in the western world carry an advanced laboratory with which to *perform* such science does not make computer science subservient to your pleasure (or the pleasure of other sciences using computers).

*[We will help you, though. Just don’t be arrogant about it.]*
The poster child for an interesting theorem in Theory of Computation.

**Rice’s Theorem** (1953).

(Scott buckle up: we need to cover 63 more years of research in the rest of the talk.)
• Remember the circuits on the right?
• Circuits are *extensional*: the little tables on the right describe the *input-output function* of each circuit.
• A program is a concrete *implementation* of a particular circuit – there are many different ways of building the same circuit, and your way may be different from mine. But they all share the same input-output function.
• In general, programs on your computer will have infinitely many possible inputs. For example, 1,2,3, ... or any finite-length text in English.

(In practice, the actual inputs to your programs will always come from a finite set – because your computer has finite memory. But since we want to reason about all computers with different memory sizes, it is just easier to assume that all inputs are possible.)
We want programs to *decide* properties of other programs.

A property of programs is *decidable* if there exists a program $p$ that, when given any other program $q$, says "yes" if $q$ has the property, and says "no" otherwise.

(This can of course also be done for properties of many other things than just programs.)
A property $R$ of programs is said to be *extensional* if, for all programs $p$ and $q$: if $p$ has property $R$ and $q$ has the same input-output function as $p$, then $q$ also has property $R$.

That is, a property $R$ is extensional if we *cannot* use it to distinguish between programs that have the same input-output behavior.

The following properties are extensional:

- $p$ outputs the word “Shibboleth” on all inputs.
- On input $x$, $p$ outputs $x$.
- $p$ loops infinitely on input “2”.
- $p$ outputs “3” when given “1” as input.

The following properties are *not* extensional:

- $p$ runs for at most 20 seconds.
- $p$ contains 27 instructions.
• Rice’s Theorem: *If an extensional property of programs is decidable, then either all programs have the property, or no programs have the property.*

• So: There are *no* non-trivial extensional properties.

• Thus, for example no program can decide, when given another program q as input, whether that program outputs “3” on input “1”.

• The above will hold for what you already think of as a program (or “app” on your phone) using your favorite programming language on your favorite computer.

  *It will also* hold for all the other things we can regard as (general-purpose) computers – and regard as programs.

And it will also hold for the human “computer” in the back office.

• There are many more theorems like Rice’s.
So what do we actually, really do here?

• Rice’s Theorem was about *extensional* properties – hence concerns *what* is computed. This has been studied for more than 60 years.

• We mainly attempt to crack a harder nut: prove theorems about *intensional* properties – *how* things are computed.

• One specific example of this is *computational complexity*: the study of classes of problems that programs can *decide* on computers using *limited resources*. 
• Recall The Path of Now and Forever: we want results that hold for as many different kinds of “computers” and “programs” as possible.

• But different computers and different notions of program may differ in how efficiently they can perform certain operations.

• For this, and other reasons, the classes of limited resources one must consider must have so-called “closure properties” – performing some limited transformations on the resource bounds should result in a bound still in the class.

(The potential trouble already starts when considering non-exotic computers and different real-world programming languages)

The most famous example:

\( P \) is the set of finite bit sequences decidable by a (deterministic) program that runs in time bounded by a polynomial in the size of the input.

(Recall from high school that this is a polynomial: \( ax^2 + bx + c \).

So is \( 5x + 2 \) and \( 10x^{1000} + 1 \).
• There are a multitude of classes of problems that can be solved within different classes of limited resources.

• On your right: black text denotes classes of problems, and blue, red and pink(?) denotes logical characterizations of them.

• For many of these classes, we do not know whether they are distinct. The most famous problem being the “N = NP?” problem, unsolved for more than 40 years.

(Immerman, 1999)
So what do we actually, really, really do here?

- A: We prove new theorems on intentional properties for programs (example in a moment!)

- B: We devise new ways of characterizing classes of problems that can be solved with limited problems. The end goal is to separate these classes (remember: we do not know whether many of them are equal or not)

A characterization is typically a logic whose set of models of true sentences corresponds (in a formal way) to the class of problems.

Or a new programming language where the allowed programs can compute exactly the programs in the class.

Or a set of pre-existing algebraic constructs from mathematics.
Example: An *intensional* generalization of Rice’s Theorem

- We want to say that some *intensional* properties are undecidable.

- It is already known that the set of programs running in polynomial time is undecidable.

- However, many researchers try to under- or over-approximate the set of such programs by decidable properties.

- Over-approximation is intuitively: “this program runs in polynomial time (fast!) or runs slightly more slowly, but has some other desirable property”.

(Figure by Moyen, 2016)
An intensional theorem (I will not explain the technicalities, just the consequences):

• Any non-empty, partially extensional, decidable set is extensionally complete (Moyen + Simonsen 2016).

• Consequence I: Rice’s Theorem.

• Consequence II: Any decidable set containing all programs that run in polynomial time contains (infinitely many!) programs for computing any other computable function.

• Consequence III (more prosaic): any decidable set of programs containing all programs that never send any email must also contain a spambot.
Theory of computation concerns the mathematics governing computing in this universe, and in other universes you may dream of.

The lecture will cover why this is sometimes relevant to the audience, why it is often irrelevant, and why the latter is a good thing.

The final part of the lecture will concern the particular professorship inaugurated, and how I intend to use it, for the greater good.
Let’s play a game!

- You are gods. The alphas and the omegas, the beginnings and the ends.
- One day at the office, your deity-in-chief tells you the following:
  - There are two rules:
    1. Rule 1: If you see an “a” you may change it to “f(a)”.
    2. Rule 2: If you see an “f”, you may change it to “g”.

- Your deity-in-chief says: “please start with ‘a’. Then, from the top, use Rule 1 until you cannot use it any more.”

- He then says: “oh, and when you are done with that, please use Rule 2 from the top until you cannot use it any more.”
Rule 1: If you see an “a” you may change it to “f(a)”.
Rule 2: If you see an “f”, you may change it to a “g”.

• Your deity-in-chief says: please start with “a”. Then, from the top, use Rule 1 until you cannot use it any more.

\[
a \rightarrow f(a) \rightarrow f(f(a)) \rightarrow f(f(f(a))) \rightarrow \ldots
\]

After having spent eternity doing this, you have obtained an infinitely long string: \(f(f(f(f(f(\ldots))))))\)

• He then says: oh, and when you are done with that, please use Rule 2 from the top until you cannot use it any more.

\[
f(f(f(f(f(\ldots)))))) \rightarrow g(f(f(f(f(\ldots)))))) \rightarrow g(g(f(f(f(f(\ldots)))))) \rightarrow \ldots
\]

After having spent two eternities, you have obtained a new, infinitely long string: \(g(g(g(g(g(\ldots))))))\)
• The *length* of such infinite jobs is well-defined in set theory.

• Your boss could have been nastier and given you more rules and told you to do an infinity of infinite computations.

• Of course, if you had been clever, you could just have ignored your boss and used Rules 1 and 2 alternatingly --- and only worked for one eternity instead of two.

\[(\omega + \omega)\]  
\[(\omega \times \omega)\]  
\[(\omega)\]  
\[a \rightarrow f(a) \rightarrow g(a) \rightarrow g(f(a)) \rightarrow g(g(a)) \rightarrow g(g(f(a))) \rightarrow g(g(g(a))) \rightarrow \ldots\]
• But for every pantheon of gods there is a Prometheus.

(A Prometheus stole fire from Olympus and gave it to mankind.)

• A human can write programs that do the work for you (and yes, even continue computing after an infinity has passed).

(For logicians: This works for any ordinal smaller than the Church-Kleene ordinal.)

• And a human can even write a program that, no matter what rules and instructions you deity in chief gives you, will automatically transform computations that lasts any (even infinite!) number of eternities into one that only requires a single eternity.

(One program that on input any infinite computation satisfying simple convergence and computability requirements will output a computation of length omega)

• Let’s see it!
• This very much the world of mankind with actual computers.

• Standard natural constants such as pi or e have infinitely many digits, and natural algorithms for computing them can usually be run forever (and truncated whenever we have enough digits).

• In general, one may want to make several passes over infinite lists of digits. The program for “compression” that you have just seen automatically transform even an infinite number of such passes into a single pass.
Computing with Infinite Terms and Infinite Reductions

JEROEN KETEMA, Imperial College London
JAKOB GRUE SIMONSEN, University of Copenhagen

We define computable infinitary rewriting, introducing computability to the study of convergent, potentially infinite reductions over potentially infinite first-order terms.

We provide the first constructive proofs of two central results from infinitary rewriting: (a) composition of left-linear systems, and (b) confluence of orthogonal, non-collapsing systems. Thus, for (a) we show that there is a program that, given a computable reduction $S$ having computable ordinal length, will output a program computing a reduction of length at most $\omega$ beginning and ending in the same terms as $S$. For (b) we show that if the peak of a confluence diagram consists of computable reductions, then so does the valley, and there exists a program computing the valley when given the peak as input. As a corollary, we obtain a simple constructive proof of an infinitary Church-Rosser property.

BASEL implementations are provided for all constructions.

Categories and Subject Descriptors: F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—Lambda calculus and related systems

General Terms: Theory

Additional Key Words and Phrases: Infinitary term rewriting, computability, lazy data structures

ACM Reference Format:
DOI: 10.1145/0000000.0000000

1. INTRODUCTION

Infinitary term rewriting has been invented with the purpose of extending ordinary term rewriting to (i) enable description of structures that—in theory—are infinite (e.g., lazy data structures such as streams [Tucker and Zucker 1992] and sentences in infinitary logic [Barwise 1969]), and (ii) enable description of infinite reductions involving such structures (e.g., iterating through every element of an infinite list). In ordinary term rewriting, all terms and reductions are finite, hence issues concerning computability pertain only to more advanced properties such as deciding whether a term has a normal form. But in infinitary rewriting, the most basic concepts of interest—terms and reductions—may fail to be computable at all. In addition, the existing literature on infinitary rewriting contains proofs of the standard properties expected of rewriting systems—for example, that orthogonal non-collapsing infinitary rewriting systems are confluent—but these proofs are in general not constructive. This is in contrast to ordinary rewriting: for example, ordinary orthogonal term rewriting systems are confluent, and several proofs of this fact are constructive; thus, there is an algo

Prometheus (Jeroen Ketema).
Theory of computation concerns the mathematics governing computing in this universe, and in other universes you may dream of.

The lecture will cover why this is sometimes relevant to the audience, *why it is often irrelevant, and why the latter is a good thing*.

The final part of the lecture will concern the particular professorship inaugurated, and how I intend to use it, for the greater good.
• Theory of Computation was born from mathematics. But the influence also flows the other way.

• I have talked about dream-worlds, but these were grounded in actual computation in this universe.

• Mathematics has no such concerns. Advances in the theory of computation has influenced areas of pure mathematics that have adopted notions of computability inspired by, but distinct from the usual (computation within arithmetic universes, within toposes, and many others).

• (This is not the same as certain spurious ideas purporting to actually compute more than ordinary computers. Often, this is charlatanism.)

• What could make a young science more proud than to materially influence its parent?
PART III
FOR THE GREATER GOOD
Theory of computation concerns the mathematics governing computing in this universe, and in other universes you may dream of.

The lecture will cover why this is sometimes relevant to the audience, why it is often irrelevant, and why the latter is a good thing.

The final part of the lecture will concern the particular professorship inaugurated, and how I intend to use it, for the greater good.
Theory of computation:

*Indifferent* to humanity.

‘twould have been the same had humanity been different.

And it *will* be the same when our successors use computers. Or are computers themselves.

So as a *professor* of Theory of Computation, I truly could not care less about humans, or the limits of our feeble technology.

But:

Computer science is much more than theory.

And even some theory very much concerns itself with humans.

And I, as a *human being*, actually do care about humans.
Two completely different fields of research: Human-Computer Interaction and Information Retrieval.
• An amalgam of many different conversations I have heard at the CHI conference over the years:

“Your field lacks external validity, reproducibility and any kind of methodological rigor.”

(“You’re a bunch of pot-smoking hippies whose work has no bearing on anything except the actual people you observed while smoking the aforementioned pot”)

“Well, your field is completely decontextualized and therefore ultimately practically irrelevant.”

(“You did a lab experiment on 12 white, male, middle-class college students and did some dubious number cooking on how fast they could solve contrived exercises in MS Word. So what?”)

• Happens everywhere, also in mathematics:

``Admit! All lattice theory is trivial.’’

(shouted by a very senior mathematician at Gian-Carlo Rota in an M.I.T. hallway - N. K. Thakare in *mathematical reviews*, MR2351371)
Some lessons learned from years of collaboration and administration in a hypercompetitive, highly politicized environment:

- Scientists are quite overwhelmingly willing to foist their own criteria of success upon fields they know nothing about.

- The capacity of people to pass judgment on research fields they know nothing about is seemingly unbounded – even within sub-sub-fields of the same scientific fields.

- True cross-disciplinary research only happens when all parties understand at least the basics of each other’s fields.

Some advice to students:

And some advice to professors:

Be careful what you judge – your undergraduates likely have a broader knowledge of contemporary computer science than you do.

(so you tend to look very silly …)
The greater good:

- Fields of research can occasionally be insular with great success. *People* and *academic departments* usually *cannot*.

- I am a theorist *third*, a computer scientist *second*, and a scientist *first*.

- I will try to understand *what* you do and *why* it is done in the way you do it without passing judgment.

- I will relentlessly browbeat *you* to attempt to understand what your colleagues are doing and why they do it in the way they do.

- *(All of the above just comes down to retaining the intellectual curiosity that made us become scientists in the first place.)*
To collaborators from DIKU:

• The IR lab from the IMAGE Section (Brian, Casper, Christina, Niels, Yevgeny).

• The entire HCC section (Too many to mention. Except Sebastian who would never stop complaining if I did not mention him).

And to my family!

Special thanks to my local group in Theory of Computation and related subjects (and sorry guys, for only speaking about the easy stuff).
And thank you for coming!

Beer will be served outside the auditorium!