# *Landin's Thesis, Rogers' Isomorphism Theorem* and the Programmer's *Turing Thesis*

Manuel Bremer

§1 Church stated his thesis on the elucidation of the intuitive notion of computable functions by means of a formal explication originally in terms of the λ-calculus:

**(*Church's Thesis* 1) The effective calculable functions**
**are identical to the λ-definable functions.**

Church a little later more generally expressed his thesis as:

**(*Church's Thesis* 2) The effective calculable functions**
**are identical to the recursive functions.**[1]

This became then the more general thesis that everything that is computable is recursively computable, which then in combination with Turing's work, who showed (i) that the recursive functions are computable by his notional machines, and (ii) that there is a *Universal Machine* able to simulate any other machine, led to the

**(*Church-Turing Thesis*) Everything that is computable is**
**computable by a Turing-Machine.**

[1] Cf. Church, "An Unsolvable Problem of Elementary Number Theory. Preliminary Report" and "A note on the Entscheidungsproblem".

where “Turing-Machine” refers to a Deterministic Turing-Machine (DTM). Other models of effective computation turned out equivalent to Turing-Machines (TM), from Non-Deterministic Turing-Machines (NTMs) to PRAMs (*Programmable Random-Access Machines*), which are closer to the *von Neumann-Architecture* of most real computing machines.

§2 The λ-calculus (LC) is said to have laid the foundations not just of *Functional Programming* (FP) but also of programming in general, despite the TM model being a clearer model of a computing machine as a machine carrying out procedures. A TM actually proceeds in steps of computation. It can be programmed by a machine table. The PRAM improves on that in the model of (random) memory access and comes close to the *von Neumann-Architecture* of standard computing machines. Low level programming with Assembler (like) instruction sets takes only a small step from a simple PRAM to higher level programming. A paradigmatic higher level programming language like **C** takes then only a smaller step from Assembler and embodies the paradigm of *Imperative Programming* (IP) and computer memory management. This procedural approach seems distinct from the λ-calculus.[2]

In the 1960s/1970s programming was just burgeoning into a manifold of programming languages, **LISP** and **ALGOL** being two of the first programming languages with **LISP** being more functional than imperative and **ALGOL** being more imperative (leading to its proper imperative followers **PASCAL** and **C**), Peter Landin proposed that programming languages are just LC with added ‘syntactic sugar’ for easier comprehension and practical implementation.[3] This has been dubbed:

**(*Landin’s Thesis*)** **Programming Languages are λ-calculus sweetened with syntactic sugar.**

[2] Similar remarks apply to the low-level *byte code* instruction of a Virtual Machine (like the *Java Virtual Machine*, JVM), which resemble an extended Assembler (with security checks etc.).

[3] Cf. Landin, “A correspondence between ALGOL 60 and Church's Lambda-notation”.

λ-terms in LC and FP define functions, they give no instructions. Computation in FP occurs in evaluating them, starting from the β-reduction rule (applying the term in the scope to some arguments) and evaluating the resulting terms with respect to the involved function(s). Thus, for instance, in evaluating "λx(x + 1)4" β-reduction yields "4 + 1", and this is evaluated as an addition. Addition has to have been implemented somewhere as a procedure. The static impression λ-terms convey hides recurring on implemented (basic) procedures to evaluate λ-terms. In this way Declarative Programming (say, as Logical Programming in **PROLOG**) and Functional Programming (say, in **HASKELL**) rely on the procedures (i.e. Imperative Programming) of their (virtual) evaluation machines. The procedural semantics of a **PROLOG**-program assigns to the program the (unification involving) steps of the resolution algorithm to resolve the program (in contrast to a denotational semantics of the program). Resolution is a procedure. As machine procedures form the core of machine computation *the core* of machine computation is IP. As FP and IP are often contrasted as paradigms this contrast is justified with respect to the style of actual programming languages and corresponding code (say, in **HASKELL** or **PROLOG** in contrast to **C**). Nonetheless – especially given the paradigm contrast between IP and FP – procedures (i.e. IP) seem to be more than just 'syntactic sugar'.

In LC one can have **if then else**-like statements by an encoding that mimics the branching behaviour; e.g. if the Boolean value **true** is encoded "λxy.x", the value **false** as 'λxy.y", and branching **if then else** as "λbxy.bxy", then by β-reduction **if true t u → t**, because this is "(λbxy.bxy)(λxy.x)t u", accordingly **if false t u → u**. Thus, a code with branching can be expressed within LC. Numeral constants ('Church Numerals') can be introduced by λ-terms with iterated self-application. As also arithmetic operations can be introduced a branching code with arithmetic operations on numerals can be encoded into a – long and hard to read – λ-expression. LC can be understood as a programming language.

Translating such a λ-expression in 'ordinary' code with symbols "+", "if" etc. one can see the LC as the backbone of such symbolic code and the use of the easier to read symbols as 'syntactic sugar'.

§3 Thus, one might say, on the one hand, that symbolic code is 'syntactic sugar' for λ-expressions, but as λ-expressions are evaluated by implemented procedures which ultimately come down in a machine to Assembler or Virtual Machine operations, on the other hand symbolic code is closer related to what the machine actually does. (This is almost tautological for low level programming.)

It can be constructively shown that for every recursive program (say in **LISP** or **PROLOG**) there is an imperative, iterative **while**-program (say in **C**) without recursion that computes the same function. This can be systematically stated by reference to a theorem by Rogers[4]:

> **(*Rogers' Isomorphism Theorem*)**
>
> **For any two programming languages complete with respect to recursive functions there is a computable correct translation between their programs.**

This means that complete programming languages are equivalent in their computational power.

In the meta-theory of **C** a **C**-program is understood as an *Abstract State Machine*[5], the state of which depends on the program, the step of execution and the state of the data, which should remind one of TM machine tables and states of execution. One may instead of **while**-statements, which might be taken to presuppose an understanding of iteration, go back to the even simpler ingredients of **if**-statements and assignments (including assignments to a program counter, allowing for jumps to implement iteration). Defining an *Abstract State Machine* (ASM) by executing programs the step-wise ingredients are only **if**-statements with assignments (including updating a program counter), one may claim that they capture *all* sequential algorithms[6]:

> **(*Gurevich's Thesis*)** **Every sequential algorithm can be step-for-step simulated by an appropriate ASM.**

---

4 Cf. Kfoury & Moll & Arbib, *A Programming Approach to Computability*, pp. 115-116, 147-150.

5 Cf. Gustedt, *Modern C*, pp. 50-54.

6 Cf. Gurevich, *Sequential Abstract State Machines Capture Sequential Algorithms* and "A Natural Axiomatization of Computability and Proof of Church's Thesis".

The claim has been strengthened to the claim that, since ASMs are clearly TM-computable, a specification of the ASM amounts, in fact, since *vice versa* TM machine tables can be translated to ASM-programs, to *proving the Church-Turing Thesis*. This claim, however, rests on an *axiomatization* of 'sequential algorithm' which mobilizes our *intuitive understanding* of effective steps, again. Thus, *given* the axioms defining sequential effective algorithms one can prove their equivalence to Turing-computability, but the conceptual elucidation linking an intuitive concept to a formally explicit one has only been back-shifted to that axiomatization.[7]

Nonetheless, such arguments strengthen the case that any computable function or set is computable by a TM or (partial) recursive functions, and thus by *any programming language* rich enough to capture these.

§4 This leads to the working programmer's endorsement of rather *Turing's Thesis* (narrowly understood as elucidating the intuitive notion of computable function *best* by reference to TMs) than the original *Church's Thesis*. The notion of an algorithm can succinctly and comprehensively be stated in reference to Turing-Machines, which capture the idea of computational procedures (i.e. algorithms). As many a programming text book states succinctly: "At its heart, a program is a set of commands executed in sequence."[8] *Turing's Thesis* is closer to the programmer's approach.

Suppose a TM $M_1$ computes the decimal expansion of $\pi$. Should we say that $\pi$ is computable? In a *narrow* sense $\pi$ is not computable as we never compute $\pi$ (in its entirety). We only compute $\pi$ up to some digit. Computable in the narrow sense is the $n^{th}$ digit in the decimal expansion of $\pi$. This computation terminates. One may thus say that $\pi$ is computable (in principle).[9]

This gives us a notion of algorithm in the *narrow* sense:

---

7 Cf. Sieg, "Axioms for Computability: Do They Allow a Proof of Church's Thesis?".

8 Liberty, *Teach Yourself C++ in 21 Days*, p. 49.

9 Turing's classic paper "On Computable Numbers" deals with irrationals thus computable!

**(Algorithm 1)** An algorithm is:

- **Implementation neutral** (abstract)
- **Effective** in its individual steps (they do not require ingenuity)
- **Finite** (given finite input and a finite number of machine states in a finite time the algorithm terminates with a finite output in finite space *if* it terminates)

We see *implementation neutrality* in one algorithm being carried out by devices that may differ in their architecture and material. If 'program' is understood as 'program in one or the other programming language' then different programming languages provide different ways to implement one and the same algorithm. Therefore, we speak of 'Algorithms in C' etc.

We see *effectiveness* in the simple basic steps of the different paradigms of computability (e.g. moving the head of a Turing Machine or reading a register in a register/abacus machine). The individual steps are basic mechanical operations. In a computer they correspond to the basic instruction set or Assembler or Virtual Machine byte code.

*Partial* recursive functions are computable in the narrow sense. They correspond to TMs (as shown by Turing). If one strengthens the finitude constraint to

- **Finite** (given finite input and a finite number of machine states the algorithm *terminates* in a finite time with a finite output in finite space)

partial recursive functions are computable in that narrowest sense as well in as much as *there is* always an extensionally equivalent terminating total recursive function. [Because of the *Halting Problem* we will not always be able to identify that equivalent total function.]

In a *broad* sense $\pi$ may be said to be computable itself as $M_1$ computes it. $M_1$ is algorithmic in a broad sense as it is an infinite sequence of terminating sub-computations. This gives a notion of an algorithm in the *broad* sense.

**(Algorithm 2)** An algorithm is:

- **Implementation neutral**
- **Effective**
- **Finite in its sub-computations** (each of which given finite input and a finite number of machine states in a finite time terminates with a finite output in finite space)

Halting algorithms then are those that enter after finitely many of such sub-computations in a halting/acceptance state. Non-halting algorithms either stay in a non-halting state or consist of an infinite sequence of sub-computations.

The condition of effectiveness is fulfilled by decomposing complex instructions into basic instructions. Thus, it does not exclude that some of the steps of an algorithm are priorly defined algorithms themselves, and are not immediately recognizable as effective unless one has seen their decomposition. This corresponds to programming in a high-level programming language with modules and sub-routines, where the high-level program is then compiled to Assembler or other machine executable instructions (like the byte code of a Virtual Machine).

§5 Allowing for unbounded sub-computations violates the finitude constraint. Allowing unbounded sub-computations – *inter alia* – would turn the *Halting Problem* solvable. If some notional computing device is (said to be) 'hyper-computational' this means that it can solve problems *not* solvable by TMs. [10] This capacity depends – in the notional machines proposed so far – on: infinitely many states, infinite input, infinite time, space or infinite precision of measurements – all features beyond the intuitive concept of computation and algorithm which involves *finitude*! A special case are *Oracle Turing-Machines*, which resort to *ineffective* steps of computation (by consulting the oracle). All these notional machines *defy realization*. None of them refutes the *Church-Turing Thesis* or even the *Physical Church-Turing Thesis* (that a machine beyond the *Turing Limit* cannot be physically realized). In a finite, discrete universe infinite advice or precision (as required

[10] Cf. Burgin, *Super-recursive Algorithms*; Syropoulus, *Hypercomputation*; Copeland (Ed.), *Hypercomputation*; Siegelmann, *Neural Networks and Analog Computation*.

in some notional models of hypercomputation machines like *Infinite Advice* TMs) are *not available*, so that hypercomputation devices can neither be constructed nor do they occur naturally. There are physical limits preventing such hypercomputation.

§6 This concept of 'algorithm' and *Turing's Thesis* delimit at the same time the expressive and computational power of programming languages to the *Turing Limit*. Programming languages that implement LC or TMs inherit their meta-logical features:

(i) They can implement programs that do not return, because to be Turing-complete they have to implement the partial recursive functions.

(ii) Whether a program in such a language terminates or is correct (i.e. computes what it should compute) is not decidable in general.

(iii) The consistency of programs in general in such a language cannot be checked by a program of that language.

These properties correspond to the *Halting Problem (Church's Theorem)*, *Rice's Theorem*, *Gödel's 2nd Incompleteness Theorem*, and the non-enumerability (by diagonalization) of the total recursive functions.

Programming languages that enforce termination of *every* program can do so only at the cost of sometimes giving the wrong answer, and there will be (even) *total* functions which they cannot compute.

§7 The *Turing-Limit* or *Turing-Barrier* has consequence for claims of 'superintelligence' in debates about (strong) AI.[11] Meant in one way 'superhuman intelligence' or by 'intelligence explosion' might mean *computational acceleration and improved computational complexity measures*. Acceleration and improved computational complexity, and thus an increased complexity (in length and storage space) of the representations to be successfully processed, may allow to solve problems which could not be solved before, because of a computational time window of solvability human capabilities cannot catch up with in too short time spans. Theoretical these

[11] Cf. Kurzweil's *The Singularity Is Near* and Vinge: "The Coming Technological Singularity: How to Survive in the Post-Human Era".

problems are *still solvable* for a human or a TM given the *Turing Thesis*. A question to be considered is whether such *quantitative* changes add up in the development of future AI to some *qualitative* change. A proper qualitative change would be one to *hypercomputation*, but the step to hypercomputation machines seems questionable [cf. §5].

And one can add: How could superhuman intelligence, if not hypercomputation, ever *evolve* out of ordinary computability, as implemented in our programming languages, their corresponding computer architecture, and – supposedly – our brains? Even a supposed super intelligent agent cannot get to hypercomputation using self-improving *programming* in in programming languages like **C** or **Java**, as these are *within* the *Turing-Limit*.

Whereas reasoning beyond our logic faculty is – by definition – beyond our ken and imagination, it is nonetheless a stumbling block for any theory of 'intelligence explosion' or 'super intelligence' that we by no means can see how the laws of quantificational and propositional logic (the realm of the Turing-computable [cf. §9]) can ever be superseded. If they are not and there is a logical core to both types of intelligence, we are still in the realm of *Turing's Thesis*. In one sense of it the 'proofs' of *Turing's Thesis* [cf. §3] in principle are, by tying it to a minimal concept of algorithm, at the same time a *refutation* of any prediction of an arriving super intelligence. The support for *Turing's Thesis* seems overwhelming, and even if there are *notional* machines beyond the *Turing Limit*, the *Physical Turing Thesis* maintains justifiably that we can never built such machines. Any realistic understanding with respect to 'intelligence explosion' thus has to focus on *acceleration and increased representational complexity*. Humans could in this case still simulate the super intelligent reasoning *in principle*, but the qualifier "in principle" now points to the substantial difference between super intelligence and ordinary human intelligence. Humans will substantially lack in performance, thus rendering the 'in principle' available simulations or translations of the super intelligent reasoning futile. As problem solving and creativity depend on working memory, simultaneously weighing options and (probabilistic) anticipation, a super intelligence equipped with advanced working memory and calculation speed might *find* solutions where humans cannot, and may even *invent* solutions and techniques which though understandable – in principle – to humans would have taken too long to be discovered, if ever in terms of human history.

Increases in computing speed face a limit: the speed of light (given current physics is right in this respect). This means that the speed up given with any future AI will be *linear* with respect to

human processing speed. Put otherwise: provided linear slowdown the computations of the new intelligence should be computable by humans. The new intelligence does not compute functions which are incomputable (e.g., solve some variant of the *Halting Problem*). The new intelligence will not even be able to solve in manageable time (e.g., before the sun burns out) highly complex problems which are computable (i.e., problems in a computational complexity class like EXP or EXPSPACE), given a large enough instance of that problem. Given the very rough categorizations of computational complexity theory, it might very well be that the new intelligence falls into *the same complexity class* like human intelligence. Nonetheless, the difference can be extensive enough for all practical human matters. To put it somewhat vaguely consider the following: The 'average speed of the average city car' is a pretty vague notion. We face a huge variety in cars, and a huge range of their speeds or even average speeds. Nevertheless, in most races (i.e., apart from those consisting exhaustively of narrow bents) a race car will vastly outperform a city car, although the city car can also arrive at the destination of the race. There is an average speed of human communication – however vaguely this specifies a range of speeds. In case a human can no longer follow the communication speed of the new intelligence a disconnection, a hiatus in intelligence has occurred.

One may take this *lack of communication speed* as one's criterion of 'intelligence explosion' in at least a weak sense. One can imagine a kind of test, similar to the idea of the *Turing Test*. A machine or cyborg (or whatever) passes the test for 'super-intelligence' if we cannot follow its communication and explanations *in real time*, but understand their content in case of *linear slow down* well enough.[12] Given increased speed a new intelligence, supposedly also equipped with a hugely extended working memory, can survey the solution space and the solution attempts to a problem much quicker (at least by brute force) and, therefore, more comprehensively. If this happens, we have a 'singularity' in the weak sense: We may – in principle – understand and check the results delivered by the new intelligence, but we no longer discover them. In that sense discovery of the new intelligence was our last unaided discovery. New intelligence in the sense specified need not come about as a conscious program, an android, cyborg, or whatever fancy material or immaterial creature imagined. A human being *augmented* by access to computing devices of the critical

[12] An analogous argument can be run with respect to working memory space. Even if the new intelligence does not fall in another complexity class than human intelligence the difference may be significant enough for all practical purposes.

strength (i.e., devices which by themselves need not possess consciousness nor all aspects of human intelligence) may with these devices *be* a complex super-intelligent system. She can reason beyond the limits of non-augmented human beings. The bottleneck of this *weak intelligence explosion* and *weak singularity* rests in the interfaces hooking up the augmented human's brain to the computer equipment. Do not hold your breath in expectation.

§8 Thus, an AI program, although not really intelligent, may prove to be beyond our understanding by its speed up of computation time. It neither needs to compute beyond the *Turing-Barrier* nor compute with *exponential* speed up. Taking the perspective of programming (languages) seriously also motivates a more realistic attitude to complexity measures for programs and algorithms. The complexity classes of computational complexity theory (like O(log n), O(n), $O(n^2)$, $O(2^n)$ etc.) are coarse grained in neglecting constant linear factors, and focussing on complexity of arbitrary input size, as well as especially worst-case scenarios.

Folklore of computational complexity theory endorses what has been dubbed:

**(Cook's Thesis) Polynomial-time TM computability is a reasonable formalization of tractable computability.**

Linear factors and coefficients size, however, can make a huge difference in real programs: for realistic (often small) inputs an exponential time complexity might be more feasible than a polynomial with two-digit coefficients, and a polynomial more feasible than a linear complexity with four-digit constant linear factors. A program might be judged for its *average* rather than its worst-case complexity.

Time complexity of a program can be measure in counting the steps taken by the program (including accessing linear data structures). The *rationale* for *Cook's Thesis* lays in the robustness of polynomial program execution and reduction between TM and PRAM programs, which include up to some constant overhead programs in Assembler, byte code or **C**. This robustness would even hold if there were non-deterministic TMs and non-deterministic programs in Assembler, byte code or **C**. Polynomial space complexity shows similar robustness. This robustness does not hold for

linear-time. On the one hand, complexity theory contains *speed up theorems* (like *Blum's Speedup Theorem*) applying to TMs working in superlinear time, which cannot be transferred from *notional* TMs to *physically realized* imperative programs. On the other hand, one can prove (by diagonalization) that constant time factors matter for these programs[13], resulting in a *Linear-Time Hierarchy* for imperative time complexity $TIME^I$:

**(Linear-Time Hierarchy)** **There is a constant *b* such that for all $a \neq 0$, there is a set in $TIME^I(a^* b^* n)$ that is not in $TIME^I(a^* n)$.**

Linear speedup, therefore, can be highly relevant for programs (in AI or elsewhere) running on physical machines, which matches the working programmer's experience with optimization.

§9 Algorithms may be planned and exactly expressed in the format of *First Order Logic* (FOL). In his classical paper on computing machines and the 'Entscheidungsproblem' Turing constructively proves two sides of an equivalence (as a lemma on the way to prove that the *Halting Problem* is equivalent to the *Entscheidungsproblem*):

**(Turing's Lemma)**

**(i) If a TM M accepts some input $\alpha$, there is a FOL axiomatized theory T of the machine table of M such that $\vdash_T \alpha$ (for all und only the accepted input $\alpha$).**

**(ii) If there is a FOL axiomatized theory T such that $\vdash_T \alpha$, there is some TM M which accepts $\alpha$ (for all und only the derivable theorems $\alpha$).**

The second part means in the light of the *Church-Turing Thesis*: Every derivation of a FOL theory is intuitively computable, which may not be considered surprising.

[13] Cf. Jones, *Computability and Complexity from a Programming Perspective*, pp. 290-91.

The first part constitutes in the light of the *Church-Turing Thesis*, however, a much more substantial claim:

> **(Turing's Universal Logic Thesis)**
>
> **Everything that is computable can be captured by a derivation in a FOL theory.**

So, TMs and FOL (theories) are computationally equivalent. This is the backbone of the idea and claim that everything logical (in the narrow sense of being algorithmically computable) *can* be captured by FOL. In that sense FOL is the universal logic.[14]

Another thesis, *Hilbert's Thesis*, therefore claims that any cogent reasoning anywhere in mathematics can be given a FOL rendering.

§10 The *Turing Thesis* and thus the *Turing-Limit* as well as *Turing's Universal Logic Thesis* have an impact on symbolic AI (sometimes called "Good Old Fashioned AI" to contrast it with 'Machine Learning') directly and 'Machine Learning' indirectly, as both are programmed with standard programming languages (from classical **LISP** and **PROLOG** to nowadays **Python** etc.) on standard machines (PRAMs with von Neumann-architecture), and also have an impact on any *Computational Theory of Cognition*.

Given (i) *Turing's Thesis*, (ii) a *Representational Theory of Mind*, which plausibly claims that the mind can be characterized by symbol manipulation (of some sort or other), (iii) the arithmetization of formal languages, and (iv) *Montague's Thesis* (that each natural language is equivalent to some formal language)[15], we get a strong version of the *Computational Theory of Mind*:

---

[14] Whether a rendering in FOL is always most perspicuous is another matter, and that there are contexts (prominently inconsistent contexts) where a sub-logic needs to be used is well known.

[15] Cf. Montague, *Formal Philosophy*, p. 188.

**(Strong Computational Theory of Mind)**

Anything that could be computed or said (by a human person) can be computed or generated by a DTM.

This leads directly to the idea of a *mechanization of the mind* in *strong* versions of AI, which aim not only at simulation of individual faculties accompanied by intelligence in human persons, but a general-purpose AI (sometimes called "General Problem Solver"[16] or "Artificial General Intelligence"), which may even include those aspects of a person's mind which are not cognitive in a narrow sense, thus claiming:

**(Strong Artificial Intelligence)**

The mind can be simulated by a DTM.

To give *mechanism* some bite it has to claim that the mind is computable. Of course, in both theses the reference to DTMs could be replaced by one to a computationally equivalent but more efficient model like PRAMs. One might have serious doubts about these last two theses. In any case – even if for the sake of the argument we entertain them – controversies and results concerning Turing-computability [like in the preceding paragraphs] can be seen as having immediate effects in the philosophy of mind. Asking for the *Turing Limit* means asking whether there are other concepts of computability and whether a machine corresponding to such a concept of computability might compute functions which cannot be computed by a DTM. Since some claim the brain – which need not be identical to the mind – to be an analog net, one has also to consider the computing power of such nets. It turned out that the analog computational models proposed so far either cannot be realized by us, because they require *infinite precision* – if infinite network weighs ever occur in nature at all – in measurement or manipulation, or they are computationally equivalent to DTMs.

Maybe the brain is an analog device that *implements at some level* a TM.[17]

---

[16] Cf. Newell, *Unified Theories of Cognition*.

If the analog processes are not *cognitively penetrable,* however, they need not belong to a level of the mind that concerns cognition in the intuitive sense. As *Turing's Thesis* is not falsified by super-computability, since super-computability is not algorithmic in the usual sense, even if the brain was a super-computer, the mind need not be, so strong AI claims would not be falsified by that. Not only super-computability devices, but *also* Turing-computable networks, however, provide significant *speed up*. You might speculate whether this is a reason to assume that the mind is realized in such a structure.

[17] Cf. Dennett, *Consciousness Explained*, pp. 209-26.