Church's Thesis, 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.

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.

[1] Cf. Church, "An Unsolvable Problem of Elementary Number Theory. Preliminary Report" and "A note on the Entscheidungsproblem"; cf. Schulze, *Die These von Church*.

§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.

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.[2] This has been dubbed[3]

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

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**-

[2] Cf. Landin, "A correspondence between ALGOL 60 and Church's Lambda-notation".
[3] Cf. Trakhtenbrot, "Comparing the Church and Turing Approaches: Two Prophetical Messages".

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 operations, on the other hand symbolic code is closer related to what the machine 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]:

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

(*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.

§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). *Turing's Thesis* is closer to the programmer's approach.

Suppose a TM M_1 computes the decimal expansion of π. Should we say that π is computable? In a *narrow* sense π is not computable as we never compute π (in its entirety). We only compute π up to some digit. Computable in the narrow sense is the n^{th} digit in the decimal expansion of π. This computation terminates. One may thus say that π is computable (in principle).[5]

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

(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

[5] Turing's classic paper "On Computable Numbers" deals with irrationals thus computable!

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.
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 π 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. Allowing for unbounded sub-computations violates the finitude constraint.

§5 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. 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).
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.

§6 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 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:

(i) If a TM M accepts some input α, 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 α).

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

The first part means in the light of the *Church-Turing Thesis*:

(i) Everything that is intuitively computable can be captured by a derivation in FOL theory.

The second part means in the light of the *Church-Turing Thesis*:

(ii) Every derivation of a FOL theory is intuitively computable.

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.[6]

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

[6] 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.