To appear in "Church's Thesis after 70 Years" ed. A. Olszewski, Logos Verlag, Berlin, 2006.

Church's Thesis and Functional Programming

David Turner
Middlesex University, UK

The earliest statement of Church's Thesis, from Church (1936) p356 is

> *We now define the notion, already discussed, of an* **effectively calculable function of positive integers** *by identifying it with the notion of a recursive function of positive integers (or of a lambda-definable function of positive integers).*

The phrase in parentheses refers to the apparatus which Church had developed to investigate this and other problems in the foundations of mathematics: *the calculus of lambda conversion*. Both the Thesis and the lambda calculus have been of seminal influence on the development of Computing Science. The main subject of this article is the lambda calculus but I will begin with a brief sketch of the emergence of the Thesis.

The epistemological status of Church's Thesis is not immediately clear from the above quotation and remains a matter of debate, as is explored in other papers of this volume. My own view, which I will state but not elaborate here, is that the thesis is empirical because it relies for its significance on a claim about what can be calculated by mechanisms. This becomes clearer in Church's restatement of the thesis the following year, after he had seen Turing's paper, see below. For a fuller discussion see Hodges (this volume).

Three definitions of the effectively computable functions of the natural numbers (non-negative integers, hereafter N), developed nearly contemporaneously in the early to mid 1930's, turned out to be equivalent. Church (1936, quoted above) showed that his own theory of lambda definable functions yielded the same functions on $N^k \to N$ as the recursive functions of Herbrand and Gödel [Herbrand 1932, Gödel 1934]. This was proved independently by Kleene (1936).

A few months later Turing (1936) introduced his concept of logical computing machine (LCM) - a finite automaton with an unbounded tape divided into squares, on which it could move left or right and read or write symbols from a finite alphabet, in accordance with a specified state transition table. A central

result of the paper is the existence of a universal LCM, which can emulate the behaviour of any LCM whose description is written on its tape. In an appendix Turing shows that the numeric functions computable by his machines coincide with the lambda-definable ones.

In his review of Turing's paper, Church (1937) writes

> *there is involved here the equivalence of three different notions: computability by a Turing machine, general recursiveness ... and lambda-definability ... The first has the advantage of making the identification with effectiveness in the ordinary sense evident immediately ... The second and third have the advantage of suitability for embodiment in a system of symbolic logic.*

The Turing machine led, about a decade later, to the Turing/von-Neumann computer - a realization in electronics of Turing's universal machine, with the important optimization that (an initial portion of) the tape is replaced by a random access store. The concept of a programming language didn't yet exist in 1936, but the second and third notions were eventually to provide the basis of what we now know as functional programming.

2 The Halting Theorem

All three notions of computability involve *partiality* in an essential way. General recursion schemas yield the partial recursive functions, which may for some values of their arguments fail to produce a result. We will write their type as $N^k \to \overline{N}$. We have $\overline{N} = N \cup \{\bot\}$ where the undefined value $\bot$ represents non-termination[1]. The recursive functions are the subset that are everywhere defined. That this subset is not recursively enumerable is shown by a use of Cantor's diagonalization argument[2]. Since the partial recursive functions *are* recursively enumerable it follows that the property of being total (for a partial recursive function) is not recursively decidable.

By a separate argument it can be shown that the property for a partial recursive function of being defined at a specified value of its input vector is also not in general recursively decidable. Similarly, Turing machines may not halt and lambda-terms may have no normal form; and these properties are not, respectively, Turing-computable or lambda-definable, as is shown in each case by a simple argument involving self-application.

Thus of perhaps equal importance with Church's Thesis and which emerges from it is the Halting Theorem: given an arbitrary computation whose result is of type $\overline{N}$ we cannot in general decide if it is $\bot$. What is actually proven, e.g. of the *halts* predicate on Turing machines, is that it is not Turing-computable

[1]The idea of treating non-termination as a peculiar kind of value, $\bot$, is more recent and was not current at the time of Church and Turing's foundational work.

[2]The proof is purely constructive and doesn't depend on Church's Thesis: any effective enumeration, h, of computable functions in $N \to N$ is incomplete - it lacks $f(n) = h(n)(n)+1$.

(equiv not lambda-definable etc). It is by an appeal to Church's Thesis that we pass from this to the assertion that *halting* is not effectively decidable.

The three convergent notions (to which others have since been added) identify an apparently unique, effectively enumerable, class of functions of type $N^k \to \overline{N}$ corresponding to what is computable by *finite but unbounded means*. Church's identification of this class with effective calculability amounts to the conjecture that this is the best we can do.

In the case of the Turing machine the unbounded element is the tape (it is initially blank, save for a finite segment but provides an unlimited working store). In the case of the lambda calculus it is the fact that there is no limit to the intermediate size to which a term may grow during the course of its attempted reduction to normal form. In the case of recursive functions it is the minimalization operation, which searches for the smallest $n \epsilon N$ on which a specified recursive function takes the value 0.

The Halting Theorem tells us that unboundedness of the kind needed for *computational completeness* is effectively inseparable from the possibility of non-termination.

3 The Lambda Calculus

Of the various convergent notions of computability Church's lambda calculus is distinguished by its combination of simplicity with remarkable expressive power.

The lambda calculus was conceived as part of a larger theory, including logical operators such as implication, intended as an alternative foundation for mathematics based on *functions* rather than *sets*. This gave rise to paradoxes, including a version of the Russell paradox. What remained with the propositional part stripped out is a consistent theory of pure functions, of which the first systematic exposition is Church (1941)[3].

In the sketch given here we use for variables lower case letters: $a, b, c \cdots x, y, z$ and as metavariables denoting terms upper case letters: $A, B, C \cdots$. The abstract syntax of the lambda calculus has three productions. A term is one of

variable e.g. x

application AB

abstraction $\lambda x.A$

In the last case $\lambda x.$ is a *binder* and free occurrences of x in A become *bound*. A term in which all variables are bound is said to be *closed* otherwise it is *open*. The motivating idea is that closed term represent functions. The intended meaning of AB is the application of function A to argument B while $\lambda x.A$ is the function which for input x returns A. Terms which are the same except for renaming of bound variables are not distinguished, thus $\lambda x.x$ and $\lambda y.y$ are the same, *identity function*.

[3]In his monograph Church defines two slightly differing calculi called λI and λK, of these λK is now regarded as canonical and is what we sketch here.

In writing terms we freely use parentheses to remove ambiguity. We further adopt the conventions that application is left-associative and that the scope of a binder extends as far to the right as possible. For example $f\ g\ h$ means $(f\ g)h$ and $\lambda x.\lambda y.Ba$ means $\lambda x.(\lambda y.(Ba))$.

The calculus has only one essential rule, which shows how to substitute an argument into the body of a function:

$(\beta) \qquad (\lambda x.A)B \rightarrow_\beta [B/x]A$

Here $[B/x]A$ means substitute B for free occurrences of x in A. The smallest reflexive, symmetric, transitive, substitutive relation on terms including $\rightarrow_\beta$, written $\Leftrightarrow$, is Church's notion of $\lambda-$*conversion*. If we omit symmetry from the definition we get an oriented relation, written $\Rightarrow$, called *reduction*.

An instance of the left hand side of rule β is called a *redex*. A term containing no redex is said to be in *normal form*. A term which is convertible to one in normal form is said to be *normalizing*. There are non-normalizing terms, of which perhaps the simplest is $(\lambda x.xx)(\lambda x.xx)$. We have the cyclic

$$(\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (\lambda x.xx)(\lambda x.xx)$$

as the only available step.

The two most important technical results are

Church-Rosser Theorem If $A \Leftrightarrow B$ there is a term C such that $A \Rightarrow C$ and $B \Rightarrow C$. An immediate consequence of this is that the normal form of a normalizing term is unique[4].

Normal Order Theorem Stated informally: the normal form of a normalizing term can be found by repeatedly reducing its *leftmost redex*[5].

To see the significance of the normal order theorem consider the term

$$(\lambda y.z)((\lambda x.xx)(\lambda x.xx))$$

We have

$$(\lambda y.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta z$$

which is the normal form. But if we try to reduce the argument $((\lambda x.xx)(\lambda x.xx))$ to normal form first, we get stuck in an endless loop.

In general there are many ways of reducing a term, since it or one of its reducts may contain multiple redexes. The normal order theorem gives a sequential procedure, ***normal order reduction***, which is guaranteed to reach the normal form if there is one. Note that normal order reduction substitutes arguments into function bodies without first reducing any redexes inside the argument, which amounts to *lazy evaluation*.

[4]This means unique up to changes of bound variable, of course.

[5]In case of nested redexes, leftmost is usually defined as leftmost-outermost, although the theorem will still hold if we take leftmost-innermost.

A closed term of pure[6] λ-calculus is called a *combinator*. Note that any normalizing closed term of pure λ-calculus must reduce to an abstraction. Some combinators with their conventional names are:

$S = \lambda x.\lambda y.\lambda z.xz(yz)$

$K = \lambda x.\lambda y.x$

$I = \lambda x.x$

$B = \lambda x.\lambda y.\lambda z.x(yz)$

$C = \lambda x.\lambda y.\lambda z.xzy$

It is evident that λ-calculus has a rich collection of functions, including functions of higher type, that is whose arguments and/or results are functions, but since (at least closed) terms can denote *only* functions and never ground objects it remains to show how to represent data such as the natural numbers. Here are the Church numerals

$\overline{0} = \lambda a.\lambda b.b$

$\overline{1} = \lambda a.\lambda b.ab$

$\overline{2} = \lambda a.\lambda b.a(ab)$

$\overline{3} = \lambda a.\lambda b.a(a(ab))$

etc. $\cdots$

To understand this representation for numbers note the effect of applying a Church numeral to function f and object a:

$$\begin{array}{lcl} \overline{0}fa & \Leftrightarrow & a \\ \overline{1}fa & \Leftrightarrow & fa \\ \overline{2}fa & \Leftrightarrow & f(fa) \\ \overline{3}fa & \Leftrightarrow & f(f(fa)) \end{array}$$

The numbers are thus represented as *iterators*. It is now straightforward to define the arithmetic operations, for example

$\overline{+} = \lambda m.\lambda n.\lambda a.\lambda b.ma(nab)$

$\overline{\times} = \lambda m.\lambda n.\lambda a.\lambda b.m(na)b$

predecessor and subtraction are a little trickier, see Church (1941). We also need a way to branch on 0:

$\overline{zero} = \lambda a.\lambda b.\lambda n.n(Kb)a$

[6]Pure means using only variables and no proper constants, as λ-calculus is presented here.

We have

$$\begin{array}{rcll} \overline{zero}\ A\ B\ N & \Leftrightarrow & A, & N \Leftrightarrow \overline{0} \\ & \Leftrightarrow & B, & N \Leftrightarrow \overline{n+1} \end{array}$$

The master-stroke, which shows every recursive function to be λ-definable is to find a universal fixpoint operator, that is a term Y with the property that for any term F,

$$YF \Leftrightarrow F(YF)$$

There are many such terms, of which the simplest is due to H.B.Curry.

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

The reader may satisfy himself that we have $YF \Leftrightarrow F(YF)$ as required.

The beauty of λ-definability as a theory of computation is that it gives not only — assuming Church's Thesis — all computable functions of type $N \to N$ but also those of higher type of any finite degree, such as $(N \to N) \to N$, $(N \to N) \to (N \to N)$ and so on.

Moreover we are not limited to arithmetic. The idea behind the Church numerals is very general and allows any data type — pairs, lists, trees and so on — to be represented in a purely functional way. Each datum is encoded as a function that captures its *elimination operation*, that is the way in which information is extracted from it during computation. It is also possible to represent *codata*, such as infinite lists, infinitary trees and so on.

Part of the simplicity of the calculus lies in its considering only functions of a single argument. This is no real restriction since it is a basic result of set theory that for any sets A, B, the function spaces $(A \times B) \to C$ and $A \to (B \to C)$ are isomorphic. Replacing the first by the second is called *Currying*[7]. We have made implicit use of this idea all along, e.g. $\overline{+}$ is curried addition.

Solvability and non-strictness

A non-normalizing term is by no means necessarily useless. An example is Y, which has no normal form but can produce one when applied to another term. On the other hand $(\lambda x.xx)(\lambda x.xx)$ is irredeemable — there is no term and no sequence of terms to which it can be applied and yield a normal form.

Definition: a term T is *SOLVABLE* if there are terms $A_1, \cdots, A_k$ for some $k \geq 0$ such that $TA_1 \cdots A_k$ is normalizing. Thus Y is solvable because we have for example

$$Y(\lambda x.\lambda y.y) \Leftrightarrow (\lambda y.y)$$

whereas $(\lambda x.xx)(\lambda x.xx)$ is *unsolvable*.

An important result, due to Christopher Wadsworth (1976), is that a term is solvable if and only if it can be reduced to *head normal form*:

$$\lambda x_1 \cdots \lambda x_n.x_k A_1 \cdots A_m$$

[7]After H.B.Curry, although the idea was first used in Schönfinkel (1924).

the variable x_k is called the *head* and if the term is closed must be one of the $x_1 \cdots x_n$. If a term is solvable normal order reduction will reduce it to HNF in a finite number of steps[8].

All unsolvable terms are equally useless, so we can think of them as being equivalent and introduce a special term $\perp$ to represent them. This gives us an extension of $\Leftrightarrow$ for which we will use $\equiv$. The two fundamental properties of $\perp$, which follow from the definitions of unsolvability and head normal form, are:

$$\perp\ A \equiv \perp \tag{1}$$

$$\lambda x.\perp \equiv \perp \tag{2}$$

Introducing $\perp$ allows an ordering relation to be defined on terms with $\perp$ as least element and a stronger equivalence relation using limits which is studied in domain theory (see later). We make one further remark here.

Definition: a term A is *STRICT* if

$$A\ \perp \equiv \perp$$

and *non-strict* otherwise. A strict function thus has $\perp$ for a fixpoint and applying Y to it will produce $\perp$. So non-strict functions play an essential role in the theory of λ-definability – without them we could not use Y to encode recursion.

Combinatory Logic

Closely related to λ-calculus is combinatory logic, originally due to Schönfinkel (1924) and subsequently explored by H.B.Curry. This has meagre apparatus indeed — just application and a small collection of named combinators. These are defined by stating their reduction rule. In the minimal version we have two combinators, defined as follows

$S\ x\ y\ z\ \Rightarrow x\ z(y\ z)$

$K\ x\ y\ \Rightarrow x$

here x, y, z are metavariables standing for arbitrary terms and are used to state the reduction rules. Combinatory logic terms have no variables and are built using only constants and application:, e.g. $K(SKK)$.

A central result, perhaps one of the strangest in all of logic, is that every λ-definable function can be written using only S and K. Here is a start

$I\ = SKK$

The proof is by considering application to an arbitrary term. We have

$$SKKx \Rightarrow Kx(Kx) \Rightarrow x$$

as required.

[8]In the published version of the paper Wadsworth's result on solvablity and hnf was erroneously attributed to Corrado Böhm.

The definitive study of combinatory logic and its relationship to lambda calculus is Curry & Feys (1958). There are several algorithms for transcribing λ-terms to combinators and for convenience most of these use besides S, K, additional combinators such as B, C, I etc.

It would seem that only a dedicated cryptologist would choose to write other than very small programs directly in combinatory logic. However, Turner (1979a) describes *compilation* to combinators as an implementation method for a high-level functional programming language. This required finding a translation algorithm, described in Turner (1979b), that produces compact combinator code when translating expressions containing many nested λ-abstractions. The attraction of the method is that combinator reduction rules are much simpler than β-reduction, each requiring only a few machine instructions, allowing a fast interpreter to be constructed which carries out normal order reduction.

The paradox

It is easy to see why the original versions of λ-calculus and combinatory logic, which included properly logical notions, led to paradoxes. (Curry calls these theories *illative*.) The untyped theory is too powerful, because of the fixpoint combinator, Y. Suppose N is a term denoting logical negation. We have

$$YN \Leftrightarrow N(YN)$$

which is the Russell paradox. Even minimal logic, which lacks negation, becomes inconsistent in the presence of Y — implication is sufficient to generate the paradox, see Barendregt (1984) p575. Because of this Y is sometimes called *Curry's paradoxical combinator*.

Typed λ-calculi

The λ-calculus of Church (1941) is *untyped*: it allows the promiscuous application of any term to any other, so types arise only in the interpretation of terms. In a *typed λ-calculus* the rules of term formation embody some theory of types. Only terms which are well-typed according to the theory are permitted. The rules for β reduction remain unchanged, as does the Church-Rosser Theorem. Most type systems disallow self-application, as in $(\lambda x.xx)$, preventing the formation of a fixpoint combinator like Curry's Y. Typed λ-calculi fall into two main groups depending on what is done about this

(i) Add an explicit fixpoint construction to the calculus - for example a polymorphic constant Y of type schema $(\alpha \to \alpha) \to \alpha$, with reduction rule $YH \Rightarrow H(YH)$. This allows general recursion at every type and thus retains the computational completeness of untyped λ.

(ii) In the other kind of typed λ-calculus there is no fixpoint construct and *every term is normalizing*. This brings into play a fundamental isomorphism between programming and logic: the Propositions-as-Types principle.

This gives two apparently very different models of functional programming, which we discuss in the next two sections.

4 Lazy Functional Programming

Imperative programming languages, from the earliest such as FORTRAN and COBOL which emerged in the 1950's to current "object-oriented" ones such as C++ and Java have certain features in common. Their basic action is the assignment command, which changes the content of a location in memory and they have an explicit flow of control by which these state changes are ordered. This reflects more or less directly the structure of the Turing/von Neumann computer, as a central processing unit operating on a passive store. Backus (1978) calls them "von Neumann languages".

Functional[9] programming languages offer a radical alternative — they are descriptive rather than imperative, have no assignment command and no explicit flow of control — sub-computations are ordered only partially, by data dependency.

The claimed merits of functional programming — in conciseness, mathematical tractability, potential for parallel execution — have been argued in many places so we will not dwell on them here. Nor will we go into the history of the concept, other than to say that the basic ideas go back over four decades, see in particular the important early papers of McCarthy (1960), Landin (1966) — and that for a long period functional programming was mainly practised in imperative languages with functional subsets (LISP, Scheme, Standard ML).

The disadvantages of functional programming within a language that includes imperative features are two. First, you are not forced to explore the limits of the functional style, since you can escape at will into an imperative idiom. Second, the presence of side effects, exceptions etc., *even if they are rarely used*, invalidate important theorems on which the benefits of the style rest.

The λ-calculus is the most natural candidate for functional programming: it is computationally complete in the sense of Church's Thesis, it includes functions of higher type and it comes with a theory of λ-*conversion* that provides a basis for reasoning about program transformation, correctness of evaluation mechanisms and so on. The notation is a little spartan for most tastes but it was shown long ago by Peter Landin that the dish can be sweetened by adding a sprinkling of syntactic sugar[10].

[9] We here use functional to mean what some call *purely functional*, an older term for this is *applicative*, yet another term which includes other mathematically based models, such as logic programming, is *declarative*.

[10] The phrase *syntactic sugar* is due to Strachey, as are other evocative terms and concepts in programming language theory.

Efficient Normal Order Reduction

The Normal Order Theorem tells us that an implementation of λ-calculus on a sequential machine should use *normal order reduction*[11], otherwise it may fail to find the normal form of a normalizing term. This requires that arguments be substituted *unevaluated* into function bodies as we noted earlier. In general this will produce multiple copies of the argument, requiring any redexes it contains to be reduced multiple times. For λ-calculus-based functional programming to be a viable technology it is necessary to have an efficient way of handling this.

A key step was the invention of *normal graph reduction*, by Wadsworth (1971). In this scheme the term is held as a directed acyclic graph, and the result of β-reduction is that a *single copy* of the argument is retained, with the function body containing multiple pointers to it. As a consequence any redexes in the argument are reduced at most once.

Turner adapted this idea to graph reduction on S, K, I, etc. combinators, allowing a much simpler abstract machine. In Turner's scheme the graph may be cyclic, permitting a more compact representation of recursion. The reduction rule for the Y combinator, $Y\ H \Rightarrow H\ (Y\ H)$, creates a loop in the graph, increasing the amount of sharing. The combinators are a target code for a compiler for compilation from a high level functional language. Initially this was SASL (Turner 1976) and in later incarnations of the system, Miranda.

While using a set of combinators fixed in advance is a good solution if graph reduction is to be carried out by an interpreter, if the final target of compilation is to be native code on conventional hardware it is advantageous to use the λ-abstractions present (explicitly or implicitly) in the program source as the combinators whose reduction rules are to be implemented. This requires a source-to-source transformation called λ-*lifting*, Hughes (1983), Johnsson (1985). This method was first used in the compiler of LML, a lazy version of the functional subset of ML, written by Lennart Augustsson & Thomas Johnsson at Chalmers University in Sweden, around 1984. Their model for mapping graph reduction onto conventional hardware, the G machine, has since been further refined, leading to the optimized model of Simon Peyton Jones (1992).

Thus over a period of two decades normal order functional languages have been implemented with increasing efficiency.

Miranda

Miranda is a functional language designed by David Turner in 1983-6 and is a sugaring of a typed λ-calculus with a universal fixpoint operator. There are no explicit λ's — instead we have function definition by equations and local definitions with *where*. The insight that one can have λ-calculus without λ goes back to Peter Landin (1966) and his ISWIM notation. Neither is the user required to mark recursive definitions as such - the compiler analyses the call graph and inserts Y where it is required.

[11] Except where prior analysis of the program shows it can be avoided, a process known as strictness analysis.

The use of normal order reduction (aka *lazy evaluation*) and non-strict functions has a very pervasive effect. It supports a more mathematical style of programming, in which infinite data structures can be described and used and, which is most important, permits communicating processes and input/output to be programmed in a purely functional manner.

Miranda is based on the earlier lazy functional language SASL (Turner, 1976) with the addition of the system of polymorphic strong typing of Milner (1978). For an overview of Miranda see Turner (1986).

Miranda doesn't use Church numerals for its arithmetic — modern computers have fast fixed and floating point arithmetic units and it would be perverse not to take advantage of them. Arithmetic operations on unbounded size integers and 64bit floating point numbers are provided as primitives.

In place of the second order representation of data used within the pure untyped lambda calculus we have algebraic type definitions. For example

```
bool ::= False | True
nat ::= Zero | Suc nat
tree ::= Leaf nat | Fork tree tree
```

Introducing new data types in this way is in fact better than using second order impredicative definitions for two reasons: you get clearer and more specific type error messages if you misuse them — and each algebraic type comes with a principle of induction which can be read off from the definition. The analysis of data is by pattern matching, for example

```
flatten :: tree -> [nat]
flatten (Leaf n) = [n]
flatten (Fork x y) = flatten x ++ flatten y
```

The type specification of *flatten* is optional as the compiler is able to deduce this; `++` is list concatenation.

There is a rich vocabulary of standard functions for list processing, *map, filter, foldl, foldr,* etc. and a notation, called *list comprehension* that gives concise expression to a useful class of iterations.

Miranda was widely used for teaching and for about a decade following its initial release by Research Software Ltd in 1985-6 provided a *de facto* standard for pure functional programming, being taken up by over 200 universities. The fact that it was interpreted rather than compiled limited its use outside education, but several significant industrial projects were successfully undertaken using Miranda, see for example Major et. al. (1991) and Page & Moe (1993).

Haskell, a successor language designed by a committee, includes many extensions, of which the most important are *type classes* and *monadic input-output.* The language remains purely functional, however. For a detailed description see S. L. Peyton Jones (2003). Available implementations of Haskell include, besides an interpreter suitable for educational use, native code compilers. This makes Haskell a viable choice for production use in a range of areas.

The fact that people are able to write large programs for serious applications in a language, like Miranda or Haskell, that is essentially a sugaring of λ-calculus is in itself a vindication of Church's Thesis.

Domain Theory

The mathematical theory which explains programming languages with general recursion is Scott's domain theory.

The typed λ-calculus looks as though it ought to have a set-theoretic model, in which types denote sets and λ-abstractions denote functions. But the fixpoint operator Y is problematic. It is not the case in set theory that every function $f \epsilon A \to A$ has a fixpoint in A.

There is second kind of fixpoint to be explained, at the level of types. We can define recursive algebraic data types, like (we are here using Miranda notation):

```
big ::= Leaf nat  | Node (big -> big)
```

This appears to require a set with the property

$$Big \cong N + (Big \to Big)$$

which is impossible on cardinality grounds.

Dana Scott's domain theory solves both these problems. A domain is a complete partial order: a set with a least element, $\perp$, representing non-termination, and limits of ascending chains (or more generally of directed sets). The function space $A \to B$ for domains A, B, is defined to contain just the *continuous* functions from A to B and this is itself a domain. Continuous means preserving limits. The continuous functions are also monotonic (= order preserving). For a complete partial order, D, each monotonic function $f \epsilon D \to D$ has a least fixed point, $\bigsqcup_{n=0}^{\infty} f^n \perp$.

A plain set, like N can be turned into a domain by adding $\perp$, to get $\overline{N}$. Further, domain equations, like $D \cong \overline{N} + (D \times D)$, $D \cong \overline{N} + (D \to D)$ and so on, all have solutions. The details can be found in Scott (1976) or Abramsky & Jung (1994). This includes that there is a non-trivial[12] domain D_∞ with

$$D_\infty \cong D_\infty \to D_\infty$$

providing a semantic model for Church's untyped λ-calculus.

Domain theory was originally developed to underpin denotational semantics, Christopher Strachey's project to formalize semantic descriptions of real programming languages using a typed λ-calculus as the metalanguage (see Strachey, 1967, Strachey & Scott, 1971). Strachey's semantic equations made frequent use of Y to explain control structures such as *loops* and also required recursive type equations to account for the domains of the various semantic functions. It was during Scott's collaboration with Strachey in the period around 1970 that domain theory emerged.

Functional programming in non-strict languages like Miranda and Haskell is essentially programming directly in the metalanguage of denotational semantics.

[12]The one-point domain, with $\perp$ for its only element, if allowed, would be a trivial solution.

Computability at higher types, revisited

Dana Scott once remarked that λ-calculus is only an algebra, not a calculus. With domain theory and proofs using *limits* we get a genuine calculus, allowing many new results.

Studying a typed functional language with arithmetic, Plotkin (1977) showed that if we consider functions of higher type where we allow *inputs as well as outputs to be* $\perp$, there are computable functions which are not λ-definable. Using domain $\overline{B}$ where $B = \{True, False\}$, two examples are:

$Or \ \epsilon \ \overline{B} \to \overline{B} \to \overline{B}$ where $Or \ x \ y$ is $True$ if either x or y is $True$

$Exists \ \epsilon \ (N \to \overline{B}) \to \overline{B}$ where $Exists \ f$ is $True$ when $\exists i \epsilon N. f \ i = True$

This complete or parallel Or must interleave two computations, since either of its inputs may be $\perp$. $Exists$ is a multi-way generalization.

What we get from untyped λ-calculus, or a typed calculus with N and general recursion, are the *sequential functions*. To get all computable partial functions at every type we must add primitives expressing interleaving or concurrency. In fact just the two above are sufficient.

This becomes important for programming with *exact real numbers*, an active area of research. Martin Escardo (1996) shows that a λ-calculus with a small number of primitives including $Exists$ can express every computable function of analysis, including those of higher type, e.g. differentiation and integration.

5 Strong Functional Programming

There is an extended family of typed λ-calculi, all without Y or any other method of expressing general recursion, in which every term is normalizing. The family includes

simply typed λ-calculus — this is a family in itself

Girard's system F (1971), also known as the second order λ-calculus (we consider here the *Church-style* or explicitly typed version)

Coquand & Huet's calculus of constructions (1988)

Martin-Löf's intuitionist theory of types (1973)

In a change of convention we will use upper case letters $A, B, C \cdots$ for types and lower case letters $a, b, c \cdots$ for terms, reserving x, y, z, for λ-calculus variables (this somewhat makeshift convention will be adequate for a short discussion).

In addition to the usual conversion and reduction relations, $\Leftrightarrow, \Rightarrow$, these theories have a *judgement of well-typing*, written $a : A$ which says that term a has type A (which may or may not be unique).

All the theories share the following properties:

Church-Rosser If $a \Leftrightarrow b$ there is a term c such that $a \Rightarrow c$ and $b \Rightarrow c$.

Decidability of well-typing This what is meant by saying that a programming language or formalism is strongly typed (aka *staticly* typed).

Strongly normalizing Every well-typed term is normalizing and every reduction sequence terminates in a normal form.

Uniqueness of normal forms Immediate from Church-Rosser.

Decidability of $\Leftrightarrow$ on well-typed terms From the two previous properties — reduce both sides to normal form and see if they are equal.

Note that decidability of the well typing judgment, $a : A$, is not the same as *type inference*. The latter means that given an a we can find an A with $a : A$, or determine that there isn't one. The simply typed λ-calculus has type inference (in fact with most general types) but none of the stronger theories do.

The first two properties in the list are shared with other well-behaved typed functional calculi, including those with general recursion. So the distinguishing property here is *strong normalization*. Programming in a language of this kind has important differences from the more familiar kind of functional programming. For want of any hitherto agreed name, we can call it *strong functional programming*[13].

An obvious difference is that all evaluations terminate[14], so we do not have to worry about $\perp$. It is clear that such a language cannot be computationally complete — there will be always-terminating computable functions it cannot express (and one of these will be the interpreter for the language itself). It should not be inferred that a strongly normalizing language must therefore be computationally weak. Even simple typed lambda calculus, equipped with N as a base type and primitive recursion, can express every recursive function of arithmetic whose totality is provable in first order number theory (a result due to Gödel, 1958). A proposed elementary functional programming system along these lines, but including codata as well as data, is discussed in Turner (2004).

A less obvious but most striking consequence of strongly normalization is a new and unexpected interface between λ-calculus and logic. We show how this works by considering the simplest calculus of this class.

Propositions-as-Types

The simply typed λ-calculus (STLC) has for its types the closure under $\rightarrow$ of a set of base types, which we will leave unspecified. As before we use $A, B, C \cdots$ as variables ranging over types. We can associate with each closed term a type schema, for example

$$\lambda x.x : A \rightarrow A$$

[13] Another possible term is "total functional programming", although this has the disadvantage of encouraging the unfortunate term "total function" (redundant because it is part of the definition *function* that it is everywhere defined on its domain).

[14] This seems to rule out indefinitely proceeding processes, such as an operating system, but we can include these by allowing codata and corecursion, see eg Turner (2004).

The function $\lambda x.x$ has many types but they are all instances of $A \to A$, which is its *most general* type.

A congruence first noticed by Curry in the 1950's is that the types of closed terms in STLC correspond to tautologies of intuitionist propositional logic, if we read $\to$ as implication, e.g. $A \to A$ is a tautology. The correspondence is exact, for example $A \to B$ is not a tautology and neither can we make any closed term of this type. Further, the most general types of the combinators $\mathbf{s} = \lambda\mathbf{x}.\lambda\mathbf{y}.\lambda\mathbf{z}.\mathbf{xz}(\mathbf{yz})$ and $\mathbf{k} = \lambda\mathbf{x}.\lambda\mathbf{y}.\mathbf{x}$ are

$$\mathbf{s} \,:\, ((A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

$$\mathbf{k} \,:\, A \to (B \to A)$$

and these formulae are the two standard axioms for the intuitionist theory of implication: every other tautology in $\to$ can be derived from them by *modus ponens*. What is going on here?

Let us look at the rules for forming well-typed terms of simply typed λ.

$$\frac{(x : A)\quad b : B}{\lambda x.b : A \to B} \qquad\qquad \frac{c : A \to B \quad a : A}{c\ a : B}$$

On the left[15] we have the rule for abstraction, on the right that for application. If we look only at the types and ignore the terms, these are the introduction and elimination rules for implication in a natural deduction system. So naturally, the formulae we can derive using these rules are all and only the tautologies of the intuitionist theory of implication[16].

In the logical reading, the terms on the left of the colons provide witnessing information – they record how the formula on the right was proved. The judgement $a : A$ thus has two readings — that term a has type A, but also that proof-object or witness a proves proposition A.

The correspondence readily extends to the other connectives of propositional logic by adding some more type constructors to SLTC besides $\to$. The type of pairs, cartesian product, $A \times B$, corresponds to the conjunction $A \wedge B$. The disjoint union type, $A \oplus B$, corresponds to the disjunction $A \vee B$. The empty type corresponds to the absurd (or False) proposition, which has no proof.

This *Curry-Howard isomorphism* between types and propositions is jointly attributed to Curry (1958) and to W. Howard (1969), who showed how it extended to all the connectives of intuitionist logic including the quantifiers. It is at the same time an isomorphism between terminating programs and constructive (or intuitionistic) proofs.

[15]The left hand rule says that if from *assumption* $x : A$ we can derive $b : B$ then we can derive what is under the line.

[16]The *classical* theory of implication includes additional tautologies dependant on the law of the excluded middle — the leading example is $((A \to B) \to A) \to A$, Pierce's law.

The Constructive Theory of Types

Per Martin-Löf (1973) formalizes a proposed foundational language for constructive mathematics based on the isomorphism. The Intuitionist (or Constructive) Theory of Types is at one and the same time a higher order logic and a theory of types, providing for constructive mathematics what for classical mathematics is done by set theory. It provides a unified notation in which to write *functions, types, propositions* and *proofs*.

Unlike the *constructive set theory* of Myhill (1975), Martin-Löf type theory includes a principle of choice (not as an axiom, it is provable within the theory). It seems that the source of the non-constructivities of set theory is not the choice principle, which for Martin-Löf is constructively valid, but the *axiom of separation*, a principle which is noticeably absent from type theory[17] [18].

Constructive type theory is both a theory of constructive mathematics and a strongly typed functional programming language. Verifying the validity of proofs is the same process as type checking. Martin-Lof (1982) writes

> *I do not think that the search for high level programming languages that are more and more satisfactory from a logical point of view can stop short of anything but a language in which all of constructive mathematics can be expressed.*

There exist by now a number of different versions of the theory, including several computer-based implementations, of which perhaps the longest established is NuPRL (Constable et al. 1986).

An alternative *impredicative* theory, also based on the Curry-Howard isomorphism, is Coquand and Huet's *Calculus of Constructions* (1988) which provides the basis for the COQ proof system developed at INRIA.

6 Type Theory with Partial Types

Being strongly normalizing, constructive type theory cannot be computationally complete. Moreover we might like to reason about partial functions and general recursion using this powerful logic. Is it possible to somehow unify type theory with a constructive version of Dana Scott's domain theory?

In his PhD thesis Scott F. Smith (1989) investigated adding partial types to the type theory of NuPRL. The idea can be sketched briefly as follows. For each ordinary type T there is a partial type $\overline{T}$ of T-computations, whose elements include those of T and a divergent element, $\perp$. For partial types (only) there is a fixpoint operator, $fix : (\overline{T} \to \overline{T}) \to \overline{T}$. This allows the definition of general recursive functions.

[17] Note that Goodman & Myhill's (1978) proof that Choice implies Excluded Middle makes use of an instance of the Axiom of Separation. The title should be Choice + Separation implies Excluded Middle.

[18] The frequent proposals to "improve" CTT by adding a subtyping constructor should therefore be viewed with suspicion.

The constructive account of partial types is significantly different from the classical account given by domain theory. For example we cannot assert

$$\forall x : \overline{T}.\ x \in T \vee x = \bot$$

because constructively this implies an effective solution to the halting problem for $\overline{T}$. A number of intriguing theorems emerge. Certain non-computability results can be established *absolutely*, that is independently of Church's Thesis, see Constable & Smith (1988)[19]. Further, the logic of the host type theory is altered so that it is no longer compatible with classical logic — some instances of the law of the excluded middle, of the form $\forall x.P(x) \vee \neg P(x)$ can be disproved.

To recapture domain theory requires something more than $\overline{T}$ and fix, namely a *second order* fixpoint operator, FIX, that solves recursive equations in partial types. As far as the present author is aware, noone has yet shown how to do this within the logic of type theory. This would unify the two theories of functional programming. Among other benefits it would allow us to give within type theory a constructive account of the denotational semantics of recursive programming languages.

Almost certainly relevant here is Paul Taylor's *Abstract Stone Duality* (2002), a computational approach to topology. The simplest partial type is Sierpinski space, Σ, which has only one point other than $\bot$. This plays a special role in Taylor's theory: the open sets of a space X are the functions in $X \to \Sigma$ and can be written as λ-terms. ASD embraces both traditional spaces like the reals and Scott domains (topologically these are non-Hausdorff spaces).

CONCLUSION

Church's Thesis played a founding role in computing theory by providing a single notion of effective computability. Without this foundation we might have been stuck with a plethora of notions of computability depending on computer architecture, programming language etc.: we might have Motorola-computable *versus* Intel-computable, Java-computable *versus* C-computable and so on.

The λ-calculus, which Church developed during the period of convergence from which the Thesis emerged, has influenced almost every aspect of the development of programming and programming languages. It is the basis of functional programming, which after a long infancy is entering adulthood as a practical alternative to traditional ad-hoc imperative programming languages. Many important ideas in mainstream programming languages — recursion, procedures as parameters, linked lists and trees, garbage collectors — came by cross fertilization from functional programming. Moreover the main schools of both

[19]The paper misleadingly claims that among these is the Halting Theorem, which would be remarkable. What is in fact proved is the *extensional* halting theorem, which is already provable in domain theory, trivially from monotonicity. The real Halting Theorem is *intensional*, in that the halting function whose existence is to be disproved is allowed access to the internal structure of the term, by being given its Gödel number.

operational and denotational semantics are λ-calculus based and amount to using functional programming to explain other programming systems.

The original project from whose wreckage by paradox λ-calculus survived, to unify logic with an account of computable functions, appears to have been reborn in unexpected form, via the propositions-as-types paradigm. Further exciting developments undoubtedly lie ahead and ideas from Church's λ-calculus will continue to be central to them.

[last revised 18.04.2021]

REFERENCES

S. Abramsky, A. Jung "Domain theory", in S. Abramsky, D. M. Gabbay, T. Maibaum (eds) *Handbook of Logic in Computer Science, vol. III*, OUP 1994.

H. P. Barendregt *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, 1984.

A. Church "An Unsolvable Problem of Elementary Number Theory", *American Journal of Mathematics*, 58:345–363, 1936.

A. Church: Review of A M Turing (1936) "On computable numbers ...", *Journal of Symbolic Logic*, 2(1):42–43, March 1937.

A. Church *The calculi of lambda conversion*, Princeton University Press, 1941.

R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*, Prentice Hall, 1986.

Robert L. Constable, Scott F. Smith "Computational Foundations of Basic Recursive Function Theory", *Proceedings 3rd IEEE Symposium on Logic in Computer Science*, pp 360–371, (also Cornell Dept CS, TR 88–904), March 1988. This and other papers of the NuPRL group can be found at *http://www.nuprl.org*.

T. Coquand, G. Huet "The Calculus of Constructions", *Information and Computation*, 76:95–120 (1988).

H. B. Curry, R. Feys *Combinatory Logic, Vol I*, North-Holland, Amsterdam 1958.

M. H. Escardo "Real PCF extended with existential is universal", eds. A. Edalat, S. Jourdan, G. McCusker, *Proceedings 3rd Workshop on Theory and Formal Methods*, IC Press, pp 13–24, April 1996. This and other papers of Escardo can be found at *http://www.cs.bham.ac.uk/∼mhe/papers/*.

J.-Y. Girard "Une extension de l'interpretation fonctionnelle de Gödel a l'analyse et son application a l'elimination des coupures dans l'analyse et la theorie des types", *Proceedings 2nd Scandinavian Logic Symposium*, ed. J. F. Fenstad, pp 63–92, North-Holland 1971. A modern treatment of System F can be found in — Jean-Yves Girard, Yves Lafont, Paul Taylor *Proofs and Types*, Cambridge University Press, 1989.

K. Gödel "On Undecidable Propositions of Formal Mathematical Systems", 1934 Lecture notes taken by Kleene and Rosser at the Institute for Advanced Study. Reprinted in M. Davis (ed.) *The Undecidable*, Raven, New York 1965.

K. Gödel "On a hitherto unutilized extension of the finitary standpoint", *Dialectica*, 12:280–287 (1958).

N. D. Goodman, J. Myhill "Choice Implies Excluded Middle", *Zeit. Logik und Grundlagen der Math*, 24:461, 1978.

J. Herbrand "Sur la non-contradiction de l'arithmetique", *Journal fur die reine und angewandte Mathematik*, 166:1–8, 1932.

Andrew Hodges "Did Church and Turing have a thesis about machines?", *this collection*.

J. Hughes "The Design and Implementation of Programming Languages", D. Phil. Thesis, University of Oxford, 1983 (Published by Oxford University Computing Laboratory Programming Research Group, as Technical Monograph PRG-40, September 1984).

W. Howard (1969) "The Formulae as Types Notion of Construction", privately circulated letter, published in *To H. B. Curry, Essays on Combinatory Logic, Lambda Calculus and Formalism*, eds Seldin and Hindley, Academic Press 1980.

Thomas Johnsson "Lambda Lifting: Transforming Programs to Recursive Equations", *Proceedings IFIP Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, Sept. 1985 (Springer LNCS 201).

S. C. Kleene "Lambda-Definability and Recursiveness", *Duke Mathematical Journal*, 2:340–353, 1936.

P. J. Landin "The Next 700 Programming Languages", *CACM*, 9(3):157–165, March 1966.

John McCarthy"Recursive Functions of Symbolic Expressions and their Computation by Machine", *CACM*, 3(4):184–195, 1960.

F. Major, M. Turcotte, et al. "The Combination of Symbolic and Numerical Computation for Three-Dimensional Modelling of RNA", *SCIENCE*, 253:1255–1260, September 1991.

P. Martin-Löf "An Intuitionist Theory of Types - Predicative Part", in *Logic Colloquium 1973*, eds Rose and Shepherdson, North Holland 1975.

P. Martin-Löf "Constructive Mathematics and Computer Programming", in *Proceedings of the Sixth International Congress for Logic, Methodology and Philosophy of Science*, pp 153–175, eds Cohen, Łos, Pfeiffer & Podewski) North Holland 1982. (Also in *Mathematical Logic and Programming Languages*, eds Hoare & Shepherdson, Prentice Hall 1985, pp 167–184.)

R. Milner "A Theory of Type Polymorphism in Programming", *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

J Myhill "Constructive set theory", *Journal of Symbolic Logic*, 40(3):347–382, Sep 1975.

Rex L. Page, Brian D. Moe "Experience with a large scientific application in a functional language" in *proceedings ACM Conference on Functional Programming Languages and Computer Architecture*, Copenhangen, June 1993.

S. L. Peyton Jones "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine", *Journal of Functional Programming*, 2(2):127–202, April 1992.

S. L. Peyton Jones *Haskell 98 language and libraries: the Revised Report*, Cambridge University Press, 2003, also published in *Journal of Functional Programming*, 13(1), January 2003. This and other information about Haskell can be found at *http://haskell.org*.

G. Plotkin "LCF considered as a programming language", *Theoretical Computer Science*, 5(1):233–255, 1977.

Moses Schönfinkel (1924) "Über die Bausteine der mathematischen Logik" translated as "On the Building Blocks of mathematical logic", in van Heijenoort *From Frege to Gödel — a source book in mathematical logic 1879–1931*, Harvard 1967.

Dana Scott "Data Types as Lattices", *SIAM Journal on Computing*, 5(3):522–587 (1976).

Scott F. Smith "Partial Objects in Type Theory", Cornell University Ph.D. Thesis, 1989.

Christopher Strachey "Fundamental Concepts in Programming Languages", originally notes for an International Summer School on computer programming, Copenhagen, August 1967, published in *Higher-Order and Symbolic Computation*, Vol 13, Issue 1/2, April 2000 — this entire issue is dedicated in memory of Strachey.

Dana Scott, Christopher Strachey "Toward a mathematical semantics for computer languages", *Oxford University Programming Research Group Technical Monograph PRG-6*, April 1971.

Paul Taylor "Abstract Stone Duality", privately circulated, 2002 — this and published papers about ASD can be found at *http://www.cs.man.ac.uk/∼pt/ASD/*.

A. M. Turing "On computable numbers with an application to the Entscheidungsproblem", *Proceedings London Mathematical Society, series 2*, 42:230–265 (1936), correction 43:544–546 (1937).

D. A. Turner "SASL Language Manual", *St. Andrews University, Department of Computational Science Technical Report*, 43 pages, December 1976.

D. A. Turner (1979a) "A New Implementation Technique for Applicative Languages", *Software-Practice and Experience*, 9(1):31–49, January 1979.

D. A. Turner (1979b) "Another Algorithm for Bracket Abstraction", *Journal of Symbolic Logic*, 44(2):267–270, June 1979.

D. A. Turner "An Overview of Miranda", *SIGPLAN Notices*, 21(12):158–166, December 1986. This and other information about Miranda† can be found at *http://miranda.org.uk*.

D.A.Turner "Total Functional Programming", *Journal of Universal Computer Science*, 10(7):751–768, July 2004.

C. P. Wadsworth "The Semantics and Pragmatics of the Lambda Calculus", D.Phil. Thesis, Oxford University Programming Research Group, 1971.

C. P. Wadsworth "The Relation between Computational and Denotational Properties for Scotts D_∞ Models of the Lambda-Calculus", SIAM Journal on Computing 5(3):488–521 (1976).

† Miranda is a trademark of Research Software Limited.