Compute

I Introduction

1. It is very sad that the following is not explained before doing any serious mathematics. It took us years to find this out ourselves, and it unearths a very latent but very crippling uneasiness with any proof one writes or reads due to skepticism, which can now be remedied.

1. I compute because it is possible for me to physically exist and compute.
2. It is pointless to compute that I can compute (to formalize why I can compute) because the fact that I can compute justifies itself.
3. My ability to compute is a sequential process.
4. It matters not if I compute non-sense or not, contradiction or not, mathematics or not. Whatever I compute is computable.
5. If I so wish, I may compute rules and formalisms. To bootstrap them, I must choose a subset of actions I do for at least one of the symbols of the language, so that I can, when I compute, manipulate the formalism, compute with it.
6. Mathematics happens by computing of the kind mentioned in 5. Hence, it is predicated on the possibility of computation, which by 1. is possible because it happens and for no other real reason.
7. I can compute a formalism that abstracts the way I compute, but that again is predicated on 1. It cannot be used to justify 1, but it can be used to model it.
8. By 5. and 6. Mathematics is inescapably bootstrapped by semantic atom. This cannot be circumvented by basing it on a formal model of computation, because that itself, by 5. is bootstrapped by that same semantic atom.
9. By 8., when formalizing mathematics, per example, on a non-human computer, I must achieve a similar bootstrapping, otherwise 'nothing happens', the formal language is 'dead' since it cannot be manipulated.
10. This is easy though. The non-human machine exactly like me in 1., can compute because it is possible for it to exist and compute (this includes the possibility of it to consistently compute in a deterministic way).
11. By the choice in which I build the machine, it will have some form of intrinsic computing capability that will be used to bootstrap the formal system. Canonically, the simplest such capability is the the ability to concatenate bits. This is also called 'counting'. Everything follows from that.
12. Since everything follows from that, every sequence of computation can be attached to counting. What differentiates every such computation is what it attaches to the 1, to 2, and so on. The essential component of this difference (also called algorithm) is choice. The choice of algorithm.
13. The choice of algorithm is bootstrapped again by the possibility to exist and compute a finite sequence of symbols, symbols that are linked to the bootstrapping of the computer (human or not), allowing it to manipulate them.
14. By 13., the process as a whole of all computers (human or not) computing is hence inevitable, a computation which contains choice when seen from the point of view of the 'leaf' computer computing it,

the choice of algorithm for it coming from a 'mother' computer, and so on, until the root computer is reached: the universe providing the possibility of computing, where all choices have been turned into inevitabilities.

15. I share my computations such that they can be sensed (seen) by others who also seem to compute. I know this because I have a limited but useful model of what they are and what they do, and when I contrast it to my model of myself, I realize they also compute.
16. The probability of my existence and act of computation of 'laws of physics' is, based on computations of probability done by others and that I studied (sense) is infinitesimal but not zero.
17. Everything that is possible must happen, is inevitable.
18. By 10. and 11. My computation of the 'laws of physics' is physically inevitable even if quasi-infinitely improbable.

2. In formal mathematics, there are no 'definitions', there are only axioms. How does one pass from informal definitions to formal axioms? Due to our skepticism, we wish to mentally only work with axioms and not definitions.

3. In set-theoretical mathematics, the bootstrapping definitions (e.g of the naturals) are given by induction. If mathematics logic, definitions such as the one of 'wff' (when not done algebraically) are given recursively. Assuming one wants to write a formal theorem prover. How are these recursive definitions 'unpacked', 'used in theorems'?

4. After a lot of reading, a bit of exercises, lots of switching, it seems that we found our guiding bible in [@c34].

II Answers with Barendregt

5. (At [@c35, p.259]) we find a simple, clear and formal relation between proofs and types:

> One of the main applications of type theory is its connection with logic. For several logical systems L there is a type theory λL and a map translating formulas A of L into types [A] of λL such that □LA □ ΓA□λL M:[A],for some M, where ΓA is some context 'explaining' A. The term M can be constructed canonically from a natural deduction proof D of A. So in fact one has □L A, with proof D □ ΓA □λL [D]:[A], (1) where the map [ ] is extended to cover also derivations. For deductions from a set of assumptions one has Δ □L A, with proof D □ ΓA,[Δ] □λL [D] : [A].
>
> Curry did not observe the correspondence in this precise form. He noted that inhabited types in λ□, like A□A or A□B□A, all had the form of a tautology of (the implication fragment of) propositional logic. Howard [1980] (the work was done in 1968 and written down in the unpublished but widely circulated Howard [1969]), inspired by the observation of Curry and by Tait [1963], gave the more precise interpretation (1). He coined the term propositions-as-types and proofs-as-terms .
>
> On the other hand, de Bruijn independently of Curry and Howard developed type systems satisfying (1). The work was started also in 1968 and the first publication was de Bruijn [1970]; see also de Bruijn [1980]. The motivation of de Bruijn was his visionary view that machine proof checking one day will be feasible and important. The collection of systems he designed was called the Automath

> family, derived from AUTOmatic MATHematics verification. The type systems were such that the right hand side of (1) was efficiently verifiable by machine, so that one had machine verification of provability. Also de Bruijn and his students were engaged in developing, using and implementing these systems. Initially the Automath project received little attention from mathematicians. They did not understand the technique and worse they did not see the need for machine verification of provability.
>
> Also the verification process was rather painful. After five 'monk' years of work, van Benthem Jutting [1977] came up with a machine verification of Landau [1900] fully rewritten in the terse 'machine code' of one of the Automath languages. Since then there have been developed modern versions of proof-assistants family, like Mizar, COQ (Bertot and Casŕ eran [2004]), HOL, and Isabelle (Nipkow, Paulson, and Wenzel [2002b]), in which considerable help from the computer environment is obtained for the formalization of proofs. With these systems a task of verifying Landau [1900] took something like five months. An important contribution to these second generation systems came from Scott and Martin-Lo ̈f, by adding inductive data-types to the systems in order to make formalizations more natural.25 In Kahn [1995] methods are developed in order to translate proof objects automatically into natural language. It is hoped that in the near future new proof checkers will emerge in which formalizing is not much more difficult than, say, writing an article in TeX.

III Preliminary Answers with Barbanera

6. Can we bootstrap our 'computation of mathematics' using a functional language that can only do recursion so that we bootstrap basic mathematical logic such as wff formation and checking? It seems functional programming languages already have 'too many built-in types and operations'. We know that if we can do symbol concatenation we can have the naturals and vice versa, can we bootstrap with something that basic? Or will the author point us to Lambda Calculus already doing this in a much better way?

7. Lambda Calculus seems to be exactly what we were searching for. We decided that reducing the vocabulary allowed in a language is what makes it 'formal'. Since we see now computation as a necessary bootstrap, we would like a minimal system in which to compute on symbols. The lambda calculus is exactly that. In fact this is the reason all our previous short attempts at looking at it ended in bewilderment. The right motivation for this calculus must come from mechanising computation to do logic, starting with a minimal vocabulary. We read in [@c2]

> Conspicuously absent from the language of λ-terms are primitive types (numbers, characters, Booleans, etc), and any kind of aggregation (strings, pairs, vectors, etc). This is intentional. The pure λ-calculus is minimalist, and consists only of variables, applications and abstractions.

Having done this, comes the topic of 'lambda definability', which builds the natural numbers, etc. This is nicely and directly explained in [@c3], coming from Scheme and replacing built-in constants and operators by expressing them in term of others. A similar approach is demonstrated in [@c4]. On the other hand, [@c5] seems to be a nice, simple and pure introduction to the lambda-calculus.

8. It seems that we need to understand what this means [@c6] before we continue:

> Meaning of λ-terms: first attempt * The meaning of a λ-term is its normal form (if it exists). * All terms without normal forms are identified. This proposal incorporates such a simple and natural interpretation of the λ- calculus as a programming language, that if it worked there would surely be no doubt that it was the right one. However, it gives rise to an inconsistent theory! (see the above reference).

9. In our quest for the simplest bootstrapping of computation (bootstrapping itself by its possibility), we find this on wikipedia :

> Semi-Thue systems were developed as part of a program to add additional constructs to logic, so as to create systems such as propositional logic, that would allow general mathematical theorems to be expressed in a formal language, and then proven and verified in an automatic, mechanical fashion. The hope was that the act of theorem proving could then be reduced to a set of defined manipulations on a set of strings. It was subsequently realized that semi-Thue systems are isomorphic to unrestricted grammars, which in turn are known to be isomorphic to Turing machines. This method of research succeeded and now computers can be used to verify the proofs of mathematic and logical theorems.

> At the suggestion of Alonzo Church, Emil Post in a paper published in 1947, first proved "a certain Problem of Thue" to be unsolvable, what Martin Davis states as "...the first unsolvability proof for a problem from classical mathematics – in this case the word problem for semigroups." (Undecidable p. 292)

> Davis [ibid] asserts that the proof was offered independently by A. A. Markov (C. R. (Doklady) Acad. Sci. U.S.S.R. (n.s.) 55(1947), pp. 583–586

This [@c7] seems to be a good survey source on the topic. Sim-Thue systems are Turing-complete. But how do we pass from such a 'purely computational, almost anything goes' system to the lambda calculus to logic to mathematics? What does this mean?

> The notion of a semi-Thue system essentially coincides with the presentation of a monoid

IV Induction and Recursion with Taylor

(Taylor) is a golden. To look at recursion and induction, it seems a good solid and computationally grounded starting point is definition 2.5.1 (p.95) of the recursive paradigm.

V Curry-Howard with Soerensen et.

10. We need to understand this

As we saw in the preceding section every simply typable term has a normal form. In fact, one can effectively find this normal form by repeated reduction of the leftmost redex. (These results hold for both the a la Curry and a la Church systems). Therefore one can easily figure out whether two simply typable terms are β-equal: just reduce the terms to their respective normal-forms and compare them.

VI Treatise on Intuitionistic Type Theory

11. gold, TODO: add other intuit resources and how intuit improves on finitism.

VII Finitism with Tait

12. see highlights

VII.1 Actives Lesen, Versuch I

13. From the ability to compute, i.e. 'manipulate symbols using a small set of manipulations', we can arbitrarily manipulate.

14. We can try to restrict manipulation to something 'provably useful'.

15. The proof of usefulness is 'in the pudding', i.e. in putting the developed program (the code of its rules) as a whole to the test, and nothing else. This is *not* at this stage a 'mathematical proof' of the program's function or characteristics.

16. We wish for a system that can

1. 'calculate' i.e. do concrete (variable-less) calculations
2. 'compare results of calculations' i.e. an '=' whose calculation result is a T/F, although for arithmetic one can simply subtract and always just compare to '0'. But what is this needed for? It is needed to steer code paths, so what we really need is a conditional in the computation engine, but that is already there as a mini-program?
3. 'express math' i.e. form questions (variable-based) expressions that can be used to either 'fill in an unknown and then calculate' or 'find the unknown'. If calculation is a program (is it?) then
   - the first is an 'incomplete program' that becomes a 'calculate' program when complete (filled). This is 'just' text that can be manipulated by substituting literally the fills and then trivially 'compiled' and 'executed'.
   - the second is in its most basic level a search, a 'while' that tries all substitutions and completes if they are exhausted or the substitution satisfies. We will improve on this in the 'prove math' section, in which we hopefully can 'prove' methods that are better than searching.
4. 'prove things' about all of the above. Is this possible? Is it impossibly self-referential? Usually what is missing from the above is a 'calculation' to T/F on incomplete programs. Is this simply an added 'forall', or maybe the inclusion of 'types' that can be used in expressions (instead of variables), and the 'proof' is one because it only uses information about the 'type' and not the variable/instance?, about the 'arbitrary instance'?

5. All this sounds like an augmented calculator, where item is the regular arithmetic one.
6. Examples of proofs are the 'infinity of primes' or 'the commutativity of addition', which is eluded to in the first sentence by Tait. Arguably, this is an encouraging sign.
7. Probably, we will need a 'step-up' in language to do the proofs of the language, but maybe not always? Maybe we 'close the loop' by building up to prove what is down?

VII.2 Comments on Actives Lesen, Versuch I

17. Let us start with a very simple character manipulation system. How could we 'as soon as possible' prove things about it in it? Maybe by starting with something like an untyped 'wild' computational engine (semi-thue? While?) and introducing as soon as possible a 'type' which allows to compute with types, that is 'prove things about the computations of the system', per example about the commutativity of addition by proving on the 'type' 'natural'. This seems to be something not usually done, can we do it? We bypass the whole discussion about 'semantics' since we do not allow oursleves to bring in extra material and use it for 'proof'.

18. It seems that we are reinventing some of the history of lambda calculus. Our bootstrapping of typed from untyped (which is basically single-typed with type 'no-type') is indeed things are seen and bootstrapped as we read here: http://goodmath.scientopia.org/2012/02/29/programs-as-proofs-models-and-types-in-the-lambda-calculus/

> Once people really started to understand types, they realized that the untyped lambda calculus was really just a pathologically simple instance of the simply typed lambda calculus: a typed LC with only one base type.

> The semantics of lambda calculus are easiest to talk about in a typed version. For now, I'll talk about the simplest typed LC, known as the simply typed lambda calculus. One of the really amazing things about this, which I'll show, is that a simply typed lambda calculus is completely semantically equivalent to an intuitionistic propositional logic: each type in the program is a proposition in the logic; each β reduction corresponds to an inference step; and each complete function corresponds to a proof! Look below for how.

19. Since Semi-Thue and generally string-rewriting-systems are Turning complete, why are there not 'string rewriting' programming languages other than: Thue

20. an answer might be that one does something similar but with more complexity using term-rewriting which is:

> In term rewriting languages, terms (syntax trees) are the basic unit of state. A program consists of rules that match subterms and replace them with new subterms. Unlike our simple string rewriting example, term rewriting languages generally have complex pattern matching with variable binding, types, and even rewriting in the presence of equational theories.

as explained here: http://www.freefour.com/rewriting-as-a-computational-paradigm/

21. How would we define a type in Thue such that we can execute our program of proving things about the created type in the language (Thue) itself? Can we start with a binary digit type? then a binary string? and then prove that concatenation is commutative?

22. Can a 'restricted' 'bottom up' SRS be also Turing complete like 'Thue' while forcing by 'syntax error' the building up of our 'program'?

1. solve 'type primitives' by not allowing any 'undefined symbols'? An example is:
   - $0 > 0$
   - $1 > 1$
   - ab > a ? $0 > 0$, $1 > 1$, $00 > 0$, $10 > 1$, ... (??) (fully explicit will never work)
   - $\tau a >$???

23. From the above it seems that the 'pattern matching' part of SRS is 'annoying', as is the need to bootstrap types. Of course it can be done (?) But why not take something off-the-shelf that already does this? Our main complaint is that typed lambda seem to carry a lot of baggage, but a simple look at the list above shows that this is probably necessary baggage and that if we manger to execute our idea 22, then we start creating some kind of single-typed lambda. Would a compromise be that we take single-typed or typed lambda and try to at least only allow to prove things about itself? per example proof of addition (or concatenation) commutativity? In essence, shouldn't we be happy with what's there in terms of models (Turing, Lambda, etc.) not minding the 'seemingly large' baggage? We probably should.

24. It is still interesting to look at the 'Turing tarpit' and once we know more try to create a certain 'foundational tarpit' that has minimal languages that can emulate simple-typed (or single-typed) lambda.

25. We should study the difference between single and simple typed lambda to some degree. A good way might be to start with a 'non-mathematical book' about untyped lambda. This books seems nice [@c11]. It also contrasts untyped lambda with Turing Machines.

VIII Notes on (Stuart 2013)

26. After reflection on (p.21), where we doubt that semantics are superfluous. A formal mathematical language is the possibility of proof (a choice of initial conditions, of code as data), where proof (constructed by something) can be checked against ('executed against') the language's formal inference system. A programming language (syntax) is the possibility of program (a choice of initial conditions, of code), where program can be checked against ('executed against') the 'computer' which is the 'inference system' that 'runs' the 'initial conditions'. So implicitly, the proof, the program are whatever can be 'executed' against the inference system syntactically. 'Semantics' as formal program 'inference rules' are nothing but syntax, but simply one level up (are they?, if yes against what are 'they' executed? against the reader?).

27. At (p.258). Possibly a good way to reliably think about UTMs or other self-compiled languages without feeling strange, is by seeing them as incomplete programs. If a computer (or UTM) loads the code of program that is a UTM(-1), the program does nothing, it actually is stuck in a state. To actually complete running the 'partial' 'not yet at a halting state' UTM-1, it needs to be appended with mode code (e.g from file or from other i-o). In that sense the full program is UTM-1 plus that piece of code. If the piece of code is just an EOF, the full program UTM-1+EOF (probably) just halts (exits). This may make it clear that there is no self-reference in a 'full' 'direct' sense, but only in an indirect one.

28. At (p.259). A different and maybe more basic argument on forever looping is that, for us we base everything on the accidentally available but self-justifying 'possibility to compute'. Already by that it is clear that, using the 'assumption of no Leprechaun', it is possible to compute at any time, hence at all times, and hence for all time,

hence, 'loop forever' or 'not halt', in fact, the halting of computation for a human is exactly their death. Without the assumption, 'loop forever' simply means continue computing (even if it means idly computing by staying in the same state) until the possibility is no more granted (by the universe). That then includes the death of all humans. So just as the ability to compute is self-bootstrapping and self-justifying, it also is all that is needed to 'prove' that it is inherently linked to looping forever, even if that happens on a 'chain' of computers, the lower ones spawning and dying likes leaves in a tree. The ultimate root lambda never fully evaluates.

29. At (p.262). What is it exactly that makes a human mind intuitively at first refuse to believe this page, or the 'quines' here: https://en.wikipedia.org/wiki/Quine_(computing) . This is also good motivation for hooking into Kleene's recursion theorem:

> Quines are possible in any Turing complete programming language, as a direct consequence of Kleene's recursion theorem.

30. At (p.262).

> Both Epigram and Charity could be considered total functional programming languages, even though they don't work in the way Turner specifies in his paper.

Charity looks very nice, what are coinductive data types already? http://pll.cpsc.ucalgary.ca/charity1/www/home.html . This sounds approachable without prerequisites:

> As a definition or specification, coinduction describes how an object may be "observed", "broken down" or "destructed" into simpler objects. As a proof technique, it may be used to show that an equation is satisfied by all possible implementations of such a specification.

(https://en.wikipedia.org/wiki/Coinduction)

31. At (p.262). How can one claim to understand computability, and hence mathematics, without understanding the difference between universal programming languages and total functional languages such as 'charity'? Total functional languages take us through a loop:

> Both Epigram and Charity could be considered total functional programming languages, even though they don't work in the way Turner specifies in his paper. So could programming directly in plain System F, in Martin-Löf type theory or the Calculus of Constructions.

Which brings us to the very interesting, and hopefully soon readable: https://en.wikipedia.org/wiki/Calculus_of_constructions and https://en.wikipedia.org/wiki/Lambda_cube . We must understand what exactly are the differences between the languages in the cube, and how they relate to untyped lambda.

This is also nice:

> Another outcome of total functional programming is that both strict evaluation and lazy evaluation result in the same behaviour, in principle

32. Inconsistency in naive set theory: if 'the' set contains itself is T, then 'the' set contains itself is F, then ... This is just a non-halting, forever-looping program, taken set theory as a 'function' and calculating the truth

value of 'the' set contains itself, that function turns out as non-total, the result of this 'input' is undefined. The closest we find to this idea is 'An Intro to Curry-Howard and Verification: Relating Logic and Machine Learning': http://users.ox.ac.uk/~{}magd3996/research/05-04-2017-NYC.pdf But in any case, isn't this idea nothing but a dilettante version of Curry-Howard?

33. One could see programs as compressors/decompressors. It is natural that we could sometime 'decompress' something to something of the same size, this is a prerequisite for a quine. The next prerequisite that the two somethings are literally equal. This does not seem at all undoable. What bothers the mind is that the naive version, the one without 'decompression', without 'encoding' and 'decoding', cannot be possible because there must be a 'command' to 'print' and then the size of the program in this naive version is the size of the argument to the command plus the size of the command, which is then more than the size of what is to be printed. But now it is easier to see why this is not impossible to fix. The examples are the quines.

34. A silly example of a language where quines are easy is one where 'non-programs' are reprinted automatically verbatim. Then all non-accepted programs are quines. Running the python interpreter in interactive mode, all strings or numbers are quines.

35. Potentially, a good next book to go from here is [@c12], up to Kleene's Recursion Theorem, after that, on the the Curry-Howard titled book

36. It turns out that in total functional programming, the implementation of 'mod' would be exactly our first version of it, which uses the worst case (but known and fixed) number of iterations. On wikipedia, we read:

> For example, quicksort is not trivially shown to be substructural recursive, but it only recurs to a maximum depth of the length of the vector (worst-case time complexity O(n2)). A quicksort implementation on lists (which would be rejected by a substructural recursive checker) is, using Haskell ...

and

> Some classes of algorithms that have no theoretical upper bound but have a practical upper bound (for example, some heuristic-based algorithms) can be programmed to "give up" after so many recursions, also ensuring termination.
>
> Another outcome of total functional programming is that both strict evaluation and lazy evaluation result in the same behaviour, in principle; however, one or the other may still be preferable (or even required) for performance reasons.

Given this, it seems sensible and less magical that such programming languages cannot compile themselves, because, as we noted, without knowing the 'input', the program to execute is incomplete, and we cannot find the worse case 'iteration' numbers. Finally

> In total functional programming, a distinction is made between data and codata—the former is finitary, while the latter is potentially infinite. Such potentially infinite data structures are used for applications such as I/O. Using codata entails the usage of such operations as corecursion. However, it is possible to do I/O in a total functional programming language (with dependent types) also without codata.

IX Branching form (Stuart 2013)

37. A good and pedestrian motivation for combinators can be found here http://web.archive.org/web/19970727171129/http://www.cs.oberlin.edu:80/classes/cs280/labs/lab4/lab40.html

> This lab comprises an introduction to the theory and practice of combinators. Combinators are a special class of higher-order functions which were invented to study the elimination of bound variables from languages of logic, arithmetic and programming. By eliminating bound variables, various formal properties of these languages can be studied independently of the usual problems associated with variable clashes and renamings. In particular, this leads to the possibility of an algebraic approach to programming and reasoning about programs, in which we can freely replace "equals with equals" in a direct, equational way. This contrasts with the usual approach to functional programming based on lambda calculus; recall from the Fall term how complicated it was to define correctly the operation of substitution on lambda terms so as to avoid variable clashes.

38. In http://fsl.cs.illinois.edu/images/d/d9/CS522-Spring-2008-02.pdf we read:

> Combinatory logic shows that bound variables can be entirely eliminated without loss of expressiveness. ... A good reference for these subjects is the book "The Lambda Calculus: Its Syntax and Semantics" by H.P. Barendregt (Second Edition, North Holland 1984). This book also contains a great discussion on the history and motivations of these theories.
>
> We will show how λ-calculus can be formalized as an equational theory. That means that its syntax can be defined as an algebraic signature (to enhance readability we can use the mix-fix notation); its proof system becomes a special case of equational deduction; and its reduction becomes a special case of rewriting (when certain equations are regarded as rewrite rules).
>
> Even though λ-calculus is a special equational theory, it has the merit that it is powerful enough to express most programming language concepts quite naturally. Equational logic is considered by some computer scientists "too general": it gives one "too much freedom" in how to define concepts; its constraints and intuitions are not restrictive enough to impose an immediate mapping of programming language concepts into it.
>
> A question addressed by many researchers several decades ago, still interesting today and investigated by many, is whether there is any simple equational theory that is entirely equivalent to λ-calculus. Since λ-calculus is Turing complete, such a simple theory may provide a strong foundation for computing.

In general, this pdf is a good 'short' of lambda, with focus on substitution and motivation for combinators. There is indeed a sort of graphical notation for lambda here: https://tromp.github.io/cl/diagrams.html linking to a thesis with five styles: [Visual Lambda Calculus] (http://bntr.planet.ee/lambda/work/visual_lambda.pdf). Is there anything similar for typed lambda?

39. The short on normal forms, and a motivation for Chuch-Rosser is this:

> We say that a lambda expression without redexes is in normal form, and that a lambda expression has a normal form iff there is some sequence of beta-reductions and/or expansions that leads to a normal form
>
> Q: Does every lambda expression have a normal form ? A: No, e.g.: (λz.zz)(λz.zz). Note that this should not be surprising, since lambda calculus is equivalent to Turing machines, and we know that a Turing machine may fail to halt (similarly, a program may go into an infinite loop or an infinite recursion).
>
> Q: If a lambda expression does have a normal form, can we get there using only beta-reductions, or might we need to use beta-expansions, too? A: Beta-reductions are good enough (this is a corollary to the Church-Rosser theorem, coming up soon!)
>
> Q: If a lambda expression does have a normal form, do all choices of reduction sequences get there? A: No. Consider the following lambda expression: (λx.λy.y)((λz.zz)(λz.zz)) This lambda expression contains two redexes: the first is the whole expression (the application of (λx.λy.y) to its argument); the second is the argument itself: ((λz.zz)(λz.zz)). The second redex is the one we used above to illustrate a lambda expression with no normal form; each time you beta-reduce it, you get the same expression back. Clearly, if we keep choosing that redex to reduce we're never going to find a normal form for the whole expression. However, if we reduce the first redex we get: λy.y, which is in normal form. Therefore, the sequence of choices that we make can determine whether or not we get to a normal form.
>
> Q: Is there a strategy for choosing beta-reductions that is guaranteed to result in a normal form if one exists? A: Yes! It is called leftmost-outermost or normal-order-reduction (NOR), and we'll define it below.

These are to be found at http://pages.cs.wisc.edu/~{}horwitz/CS704-NOTES/1.LAMBDA-CALCULUS.html#churchRosser which also seems to contain a nice way to the proof.

40. A very nice high-level overview relating many things that we are trying to study and relate is here http://www.rbjones.com/rbjpub/logic/cl/cl017.htm . This leads us to have a look at Curry's Paradox.

X Chasing Curry's Paradox, up to Stratified type for Lambda

X.1 Notes

41. We read

> The attempt to make a foundation for mathematics using the illative combinatory logic were blighted by Curry's paradox and semantic opacity. While systems avoiding this paradox have been devised, they are typically weak, and lack the kind of convincing semantic intuition found in first order set theory.

42. Wiki tells us there is no resolution for this paradox in lambda calculus (and also not in naive set theory): (https://en.wikipedia.org/wiki/Curry's_paradox). There are resolutions in some unrestricted languages, formal logic and set theory.

43. This page (http://www.goodmath.org/blog/2014/08/21/types-and-lambda-calculus/) is a must read, since it explains how

> Curry's paradox meant that the logicians working with λ-calculus had a problem that they needed to solve. And following the fashion of the time, they solved it using types. Much like ST type theory attempted to preserve as much of set theory as possible while fixing the inconsistency, typed λ-calculus tried to fix the inconsistency of λ-calculus. The basic approach was also similar: the typed λ-calculus introduced a stratification which made it impossible to build the structures that led to inconsistency.

> As an aside: this isn't really a problem in practice. The self-referential expressions that cause the Curry paradox turn into non-terminating computations. So they don't produce a paradox; they just don't produce anything. Logical inconsistencies don't produce results: they're still an error, instead of terminating with an inconsistent result, they just never terminate. Again, to the logicians at the time, the idea of non-termination was, itself, a deep problem that needed a solution

44. We read

> But to logicians like Haskell Curry, it read is "The statement (r r) implies y". Rendered into simple english, it's a statement like: "If this statement is true, then my hair is purple". It's purely a logical implication, and so even though we can't actually evaluate (r r), in logical terms, we can still say "This is a well-formed logical statement, so is it true?".

But this is a bit too vague, especially the 'logical interpretation'. Let us follow this page anyway and see if we can come back to this and compare to how exactly it is 'resolved' in typed lambda, maybe by looking at how the correspondence between proofs and programs disappears if the inconsistency re-appears in typed lambda.

45. Finally we can begin to answer this simple question: 'programs correspond to proofs: where does the "proofs" part come from, the programs coming from typed lambda?'. The answer, in a very fitting manner to our syntactic formal tendencies, is that the syntax of typed lambda and intuitionistic logic proofs is the same. Namely, for the typed lambda:

- type ::= primitive | function | (type)
- primitive ::= α | β | γ | ...
- function ::= type $\rightarrow$ type

46. Despite the author's good efforts, we still do not see the problem with

$$r = (\lambda x.((xx) \Rightarrow y))$$

even when interpreted in intuitionistic logic. We need to find out. Specifically, the problem with the fixed-point combinator and logic.

47. We need to find out about inhabited types, also from another, slightly more verbose source.

48. The two above items are what is to be done next, but now we have a reduced version (a knowledge mother node) of what happens and clearer questions. Maybe we are even ready for [@c9] up to page 72?

- Sadly, we have the feeling that [@c9] has the attitude of using mathematics to analyze the calculus and not vice versa, this might be incompatible with our program.
- Possibly, a longer, philosophically-laden, but detailed answer can be fount in [Treatise on Intuitionistic Type Theory].
- Looking at citations of [@c9], we find [Type Theory and Formal Proof: An Introduction] which seems to have the right focus (note that it uses the term constructive logic instead of intuitionistic logic).

X.2 Exercises with (Nederpelt and Geuvers 2014)

X.2.1 Introduction

49. Already the first chapter teaches us why untyped lambda is 'too wild', given the combinators at the end. Sadly, it does not clearly justify why the 'negative aspects' of untyped lambda mentioned at the end of the chapter are 'negative', this leads us to the following note.

50. This books seems very promising. Sadly, we find it talks very rarely about consistency. We would like to know directly why untyped lambda is inconsistent. This leads us to the following:

- 'A logic is inconsistent if it corresponds to a type system where every type is inhabited, i.e. you can prove any theorem.'. This means that the next priority is to understand 'inhabiting'. Luckily, our books focuses on this already at (p.50)
- The book [@c14] treats this directly at (p.71) and seems quite a nice book as well, especially when it comes to combinators. It also has a chapter explicitly dedicated to definitions!
- This page (http://brandon.si/code/the-gift-of-inconsistency/) provides a programmer's view.
- Curry's original articles [The Paradox of Kleene and Rosser] and [The Inconsistency of Certain Formal Logic] might be worth a read for their verbosity.
- This paper provides a very short version: https://pdfs.semanticscholar.org/4724/9f46c448462c37a50263ad29704100a42723.pdf
- This book might also be interesting: [Types and Programming Languages]. It mentions the paradox in a short but clear way at (p.273)
- A book that is supposed to be simple and has a specific chapter on the correspondence between lambda and logic is [@c15], the chapter is already at (p.74) and this book is hence worth looking at. Per example it says:

  > But it <lambda to intuitionistic logic correspondence> was viewed there as no more than a curiosity.

  However, the book requires prior contact with lambda calculus.
- Another (rather encyclopedic) book that talks directly about proofs as propositions and give at least a detailed example is [@c16], already at (p.25).

51. After a large loop, ending with our 'paradox.pdf', we come back to this book's exercises.

52. After coding tree visualisation of λ -terms in automah, along with binding arrows, it seems the article 'To Dissect a Mockingbird: A Graphical Notation for the Lambda Calculus with Animated Reduction' is not nonsense anymore so it is time to read it properly.

1. The first remark on this is that we finally understand the term 'application' once we add remember our 'terminology' (abstr=subst rule, appl=args) and our analysis that 'variables' in terms are misleading because they are not 'variables'. Specifically, in an application, the left 'variable' can very well be subsituted by an abstraction (a rule), upon which, if reduced, it indeed becomes an application of that rule to the application's right 'variable'. Once can of course substitute 'source code', that is, other 'abstractions'.
2. This is a subtle but very good remark:

   > For example, why is lab.a+b any better than plain old +?

3. Of course, this holds for almost everyone:

   > When I first understood the lambda calculus I felt that those arbitrary bound variables, like a,b,f and x above, just served to obscure what was really going on. They are only necessitated by the constraint of writing everything as a one-dimensional string of symbols. So I am going to introduce you to some of the strange characters that inhabit the world of combinators by using a two-dimensional notation that I developed. This notation started from the idea that, instead of the bound variables we could draw arrows connecting each lambda symbol to blank spaces in the expression where its variable would have appeared.

4. Finally reading the combinator [x. [x x]] makes sense, even by using the standard term 'apply to':

   > We can write lambda expressions like la.a the identity combinator, or la.aa the combinator that applies any combinator to itself, or lab.ba the combinator that reverses the order of application of two combinators.

5. This is a good one-linear about the relation between lambda and combinators:

   > All we need is function abstraction and application. "Abstracted from what and applied to what?", you may ask. Other such functions! Actually when they are not applied to anything except each other we refer to them as combinators rather than functions. This is called the pure lambda calculus.

6. The book 'laws of form' is intriguing. How does it related to our 'semantics-free' mathematics? does it point to a premature death of it?

   > This notation was developed as part of an attempt to place the lambda calculus on an even deeper foundation. Both the attempt (so far unsuccessful) and the notation were inspired by George Spencer-Brown's controversial book Laws of Form (1969).

   https://en.wikipedia.org/wiki/Laws_of_Form http://homepages.math.uic.edu/~{}kauffman/Laws.pdf
7. We are happy of validation of our complaint about the textbook order of introduction alpha before beta:

   > It is unfortunate that the primary form of reduction was named after the second letter of the Greek alphabet, beta (b). The form of conversion which was given the first letter, alpha (a), turns out to be merely an artifact of the particular method of representing these songs as one-dimensional strings of symbols in the textual lambda calculus.

8. Noting that replication is a part of beta-reduction 'hidden' when using textual lambda but apparent when using graphical lambda (along with substitution), here are Schoenfinkel's five birds that can create all birds

   > The Starling, shown in figure 20, is the last of Schönfinkel's original five combinators to be introduced. These are the Idiot bird, Kestrel, Cardinal, Bluebird and Starling, I, K, C, B, S (although Schönfinkel called some of them by different letters at the time). These five birds alone can easily generate all other birds. The Starling is the only one of these birds to perform replication.

> While it is remarkable enough that the five birds I, K, C, B, S can be used to derive all others, it is an even more remarkable fact, shown by Shönfinkel, that all other birds can be derived from the Starling and Kestrel alone, although the expressions involved can be enormous.
>
> Schönfinkel dreamed of another bird called the Jay from which the Starling and Kestrel (and so all birds) could be derived. It seems that the closest we can come to this is the quite complicated bird shown in figure 21. While not quite what Schönfinkel had in mind, it has been given the name Jay.
>
> The Jay was discovered in 1935 by J. Barkley Rosser and has the property that it can be used in association with the Idiot bird to derive all birds except those which ignore or eliminate one or more of their inputs (such as the Kestrel and Kite).

9. We have to find out what this means!

   > Many other sets of primitive combinators are possible and we could argue forever over which is more fundamental. In fact there is a direct relationship between bases for the theory of combinators and axiom schemes for implication logic, via a mapping called "Formulae as Types". See Hindley & Seldin (1986).

10. We finally have a chance for the Turing bird, and it is amazing!

    > If we cause a Turing bird (U) to hear its own song we end up with a bird called the Theta (Q) bird which is a fixed-point finding bird or simply a fixed-point bird. The Turing and Theta birds were discovered by Alan Turing in 1937. The Theta bird does not have a normal form. Figure 22 shows one possible representation obtained by performing a single beta reduction on UU and relocating the first U inside the remains of the second. This version of the Theta bird is at least in weak head normal form which means only that it has an outer box with an ear. A fixed-point bird has the amazing property that on hearing any bird b, it responds with a bird f of which b is fond. That is, it responds with a fixed-point of b. So if we want a solution for f in the recursive equation f = bf we need only write Qb (or Yb as we shall see later).

11. And the result of the existence of the Turing bird is of major importance to our quest of understanding recursive definitions:

    > The proof that fixed-point birds exist, is a very powerful result since it says that it is meaningful to define functions by means of recursive equations such as that for f above.
    >
    > The simplest fixed-point bird, the Why bird, is generally attributed to Haskell Curry around 1942 (see the notes on p185 of Curry, 1958). Perhaps the name refers to the incredulity of its discoverer, "Why does it exist?" or "Why does it work the way it does?".

12. And we are left with the last enigma that we also need to understand:

    > It is a fact of mathematical life (as shown by Kurt Gödel) that if we are to have something as powerful as a Why bird (a fixed-point bird) we must accept the risk of producing an Omega bird (a non-terminating bird). Our old friend the Mockingbird, with its ability to apply a bird to itself, is implicated in both.

13. Sadly, we are tending to see that our beloved [@c13] might be too hard on our tender skin, and we should start with the standard [@c31] since it actually seems that its authors to have a sense of humour, and the book is gentler, in fact, it looks like a perfect prerequisite for [@c13]. the bib references are 'c13' and 'c31' maybe this is 'providence'...

14. Intuitive explanations of the Y-combinator can additionally be found here: https://stackoverflow.com/questions/93526/what-is-a-y-combinator https://cs.stackexchange.com/questions/9604/clear-intuitive-derivation-of-the-fixed-point-combinator-y-combinator
15. Note that unlike the article, we could in fact build an animation library for lambda even without the author's special notation, simply using trees and arrows! Given a well functioning tikz (no overlap), this is a two-week project at most!

X.2.2 Notes

53. At (p.8,9). It is not explicitly stated, but the fact that for an application, the free variables are taken as a union instead of an intersection is due to the fact that the concept of free variables is there exactly to define closed λ -terms. The sense is that a closed term is such that no matter how we reduce it, one of the variables is never 'touched', 'substituted away', so we can never reduce to a 'result', to a 'variable-free' (a bit wrong, but that's because of lambda calculus mixing abstraction and application, substitution rules and arguments, almost encapsulating the operation semantics?) term. Maybe better is simply that 'no more arguments' need to be supplied, that it is a 'full program' 'not waiting for input', just like our idea that compilers are not 'full programs' since they cannot executed until the code to compile is provided. But note that the need for the definition underlines that computation must be done on the syntax to find that out is this a meaningful remark?. In any case, this is a 'dangling concept' for now. But with this thought, what is not?! Almost all concepts are dangling until closure by the theory's 'big theorems'. The problem may be that one is used, for elementary theories, for 'early applications' but in fact, for most cases, these 'applications' are merely illusions of applications. Everything is dangling until the 'big theorems'. This explains through another route the need for a mother structure. A new alternative that we now spot is using the subtle feeling of noticing the existence of a new syntax, that gives rise to new syntactic/inferential behavior, and being happy with it, seeing the 'big theorem application' as 'icing on the cake'. Although it is true that it can be difficult to spot the novelty in the syntactic behavior until exactly the 'big theorem', which usually underlines how interesting that behavior actually is.

54. Additionally to the above, we conclude that alpha-equivalence is here introduced too early. The point of this equivalence is that terms that will beta-reduce in exactly the same steps in terms of small-step operational semantics, or in other words, given a function that takes terms to terms that is invariant with respect to small-step ops of beta reduction, we need to find an algorithm that finds for two terms if they are so related. The design of the rules of alpha-eq is guided by this invariance, something that acts on two terms. So in a way, this is too early, although we understand how the textbook exposition puts it here and 'dangles' it.

55. By the note in [@c33] that untyped-lambda is consistent, we were confused. We have to differentiate between 'inconsistent' and 'logically inconsistent'.

1. untyped-lambda is consistent, as a calculus, in the sense of [@c33, p.24]

   > The lambda-calculus is consistent i.e lambda not |- true = false...

2. Untyped-lambda is a slimming down of a logically-laden lambda calculus that was dropped as a foundation of mathematics because of Kleene-Rosser

   The lambda calculus was introduced by mathematician Alonzo Church in the 1930s as part of an investigation into the foundations of mathematics.[7][8] The original system was shown to be logically inconsistent in 1935 when Stephen Kleene and J. B. Rosser developed the Kleene–Rosser paradox.[9][10]
3. A 'logically consistent' reworking of the slimmed down calculus was then the STLC:

> Subsequently, in 1936 Church isolated and published just the portion relevant to computation, what is now called the untyped lambda calculus.[11] In 1940, he also introduced a computationally weaker, but logically consistent system, known as the simply typed lambda calculus.[12]

4. The Kleene-Rosser paradox was used to find the simple Curry paradox, both affected untyped lambda 'when' used as a 'deductive lambda calculus'

   > Kleene and Rosser were able to show that both systems are able to characterize and enumerate their provably total, definable number-theoretic functions, which enabled them to construct a term that essentially replicates the Richard paradox in formal language. Curry later managed to identify the crucial ingredients of the calculi that allowed the construction of this paradox, and used this to construct a much simpler paradox, now known as Curry's paradox.
   >
   > Curry's paradox, and other paradoxes arise in Lambda Calculus because of the inconsistency of Lambda calculus considered as a deductive system. See also deductive lambda calculus.

5. So then we have to understand the difference between the lambda calculus as a deductive system and not as one. This is a good page directly treating the topic https://en.wikipedia.org/wiki/Deductive_lambda_calculus

Exercises

56. We finished the exercises of chapter 1, leading to an introductory understanding of untyped lambda calculus

57. We started the exercises of chapter 2, they mostly concern Church style typed lambda calculus $\lambda_{\rightarrow}$.

58. We read up to chapter 8 and heavily marked the book. In essence, our goal here would be the expository chapters on each kind of typed lambda calculus, forming the famous lambda cube, and culminating in the calculus of constructions, later modified into λ_d to allow more flexibility with definitions.

1. We find that after chapter 1, one cannot, as was possible in chapter 1, remove logic out of the motivations. For untyped lambda calculus is a model of computation. (Further details on all what is missing from chapter 1 can be found in the recommended 'Elements of the theory of computation'). Decisive steps for each following chapter have to be modified by proofs on the amount of expressivity of each typed version. But this is too advanced and ommitted from the book. We realize that although these lambda calculi neatly build up a strong enough calculus, by encoding logics, they are not 'necessary' in the sense that the logics 'can' be understood without them, although the calculi give a nice structure of how these calculi relate to each other. But at this stage, we feel that this is too specialized to require exercise solving. So this could be an early exit point from this dive, with a stack left unfinished. (We did learn the flag notation which we found nice). We also find that the 'additional reading' sections of the book are crucial as study continuations and not only as sections to read. 'Lambda Calculi with Types' might be a shorter route if we come back, and 'What Rests on What' can hopefully now be much more readable. We must remember that 'system-F' was the topic of Gerard's PhD and that for reasons of logic (even if computer programming polymorphism was another direction from a parallel discovery), so after this stage, logic becomes essenstial, and we might as well go back to pure logic books, eschewing the 'coding into lambda calculus' no matter how neat (keeping in mind what we learned) for a future time. We note that this dive was cause by note II.7 in 'First Exploitation' and by the date, it cost us 4 months. We solved exercise in multiple books and read countless, as a first dive into computer science, lambda calculus, type theory, more logic and automated proving.

X.2.3 Exercises with (Hindley and Seldin 2008)

X.2.3.1 Notes

59. Finally we found a way to memorise 'left/right associative' (not thanks to the author, but by paying attention). 'Associate left/right' means parentheses accumulate 'left/right'!

60. This increases our confidence in the author's style (when introducing substitution in terms):

> The details are rather boring, and the reader who is just interested in main themes should read only up to Definition 1.12 and then go to the next section. By the way, in Chapter 2 a simpler system called combinatory logic will be described, which will avoid most of the boring technicalities; but for this gain there will be a price to pay.

61. This is a very good, clear, terse and practical note about 'induction on', coupled with a definition of lgh:

> The phrase 'induction on M' will mean 'induction on lgh(M)'.

62. (At p.21). This is an unusually excellent short 'reading of' or 'interpretation' of a combinator.

> To motivate combinators, consider the commutative law of addition in arithmetic, which says (□x,y) x+y = y+x. The above expression contains bound variables 'x' and 'y'. But these can be removed, as follows. We first define an addition operator A by A(x,y) = x + y (for all x,y), and then introduce an operator C defined by (C(f))(x,y) = f(y,x) (for all f,x,y). Then the commutative law becomes simply A = C(A). The operator C may be called a combinator;

we can see how this is the right tool to 'talk about' the formal structure of function abstraction and application and formalize it on the next meta level. In the example, the C combinator captures the notion of commutativity in form, 'text on paper', 'computationally'.

63. We reached new territory, Boehm's theorem. Since this book is quite bad in the general setting, this reference helps well [@c32]. Here is the short of Boehm's theorem:

1. In λ-calculus, programs/λ-terms are constructed in a purely functional way, and there is no distinction be- tween programs and data: every program can be passed as the argument of another program. The evaluation mechanism, the so called β-rule, mimics the operation of replacing equally labeled formal place-holders with a specific λ-term, without looking at its actual structure nor if the places are for functions or for arguments. A λ-term is in normal form when it cannot be reduced any further, by the application of the β-rule, implicitly declaring the end of the evaluation process. Being the ending results of the evaluation of a λ-term (when this evaluation has an end), normal forms play the role of values and Bohm's Theorem ensures that two values are equal only if they are written in the same way.
2. This seems meaningless if one does not technically define denotation (which as we have seen in other notes about Bohm's is related to semantics):

   > In λ-calculus the normal form, if any, of a term is unique; in other words, the result of a computation is independent of the order in which the computational steps are applied. Because of this, normal forms can be seen as the "denotations" of λ-terms, or equivalently, as their primary meaning. Bo¨ hm's Theorem ensures that in λ-calculus every denotation of a λ-term (if we limit to programs/λ-terms that terminate) can be written in only one way: two syntactically distinct values/normal forms correspond in fact to two distinct denotations.

3. Such a property makes the λ-calculus an ideal mathematical model in which to interpret programs and in which to study their properties—in particular, their equivalence. Moreover, Bo ¨hm's Theorem ensures that the way in which we can separate two distinct λ-terms is internal to the calculus; it suffices to apply the distinct λ-terms to the same suitable sequence of inputs. The construction of such a sequence of inputs requires the determination of a set of combinatory operations on the tree structure of λ-terms that are at the basis the so-called Bo¨ hm out technique (see the next section). Proving his theorem, Corrado Bo¨ hm not only recognized the basic operations of the Bo¨ hm out technique, but also had the great intuition that such combinatory operations could be internalized into the λ-calculus by means of suitable λ-terms. Such a deep understanding of the computational mechanism behind the β-rule was a great breakthrough in the analysis of the basic computational mechanisms of programs and played a central role in the development of the mathematical studies of the semantics of programs (see below the section on the follow-up to Bo¨ hm's Theorem).
4. Knowing the above, we see that the author of [@c31] made a wise choice, and we stop this here.
5. Still, this is very useful to know:

   > The idea that an interesting computational system should have enough power to be able to speak about itself has played a central role in all the research of Corrado Bo¨ hm, not only in his studies on λ-calculus. In fact, in his thesis [Bo¨ hm, 1954], Corrado Bo¨ hm defined the first compiler that could be described in its own language (see also [Knuth and Pardo, 1980]). Since then, one of the main questions that guided Corrado Bo¨ hm in his research on computational systems was "how much of its meta-theory is contained into the system itself?" Therefore, when he started to think at the λ-calculus as a basis for defining programming languages (actually, he believes that the λ-calculus is THE programming language), one of the first questions that he tried to answer was if there was an internal way for studying the equality of λ-terms, and the answer was Bo¨ hm's Theorem.
   >
   > Bo¨ hm's Theorem says that the equational theory induced by the β-reduction is complete for its normal forms. In fact, trying to equate any pair of non η-equivalent normal λ-terms would correspond to equating the whole set of the normal λ-terms, forcing the collapse of the whole set of λ-terms into one point.

64. At (p.43), we read an authoritative statement that explains the marriage of lambda with logic in many texts. It also validates the which for a type-free foundation originally, but coming back to it after failure, while pursuing type-free as well:

> Both λ and CL were originally introduced as parts of strong systems of higher-order logic designed to provide a type-free foundation for all of mathematics. But the systems of Church and Curry in the 1930s turned out to be inconsistent. This led Church to turn to type theory; and he published a neat system of typed λ in 1940, [Chu40]. On the other hand, Curry was still attracted by the generality of type-free logic, and turned to analysing its foundations with great care, through a series of very general but very weak systems.

It is also nice how this validates our conclusion that understanding computation is essential:

> Until about 1960, λ and CL were only studied by a few small groups. But around that time, logicians working on computability began to ex- pand their interest from functions of numbers to functions of functions, and to ask what it meant for such a higher-order function to be com- putable.

Formal systems were devised to try to express the properties of higher-order computable functions precisely, and most of these were based on some form of applied λ or CL. The same question also inter- ested some of the leaders in the then-young subject of computer science, including John McCarthy in the late 1950s, who was one of the first advocates for the functional style of programming. McCarthy designed a higher-order programming language LISP which used a form of λ-notation.

From then on, interest in λ and CL began to grow. In 1969, the American logician Dana Scott, while designing a formal theory of higher-order computation, realized, to his own surprise, that he could build a model for pure untyped λ and CL using only standard set-theoretical concepts. Until then, untyped λ and CL had been seen as incompatible with generally-accepted set-theories such as the well-known system ZF. Scott's model changed this view, and many logicians and computer-scientists began to study his model, and other models which were invented soon after. (See Chapter 16 below, or [Bar84, Chapters 18–20].)

65. Interpreting lambda an 'operators' explained, this is golden:

Discussion 3.27 (Interpreting pure λ and CL) Up to now, this book has presented λ and CL as uninterpreted formal systems. However, these systems were originally developed to formalize primi- tive properties of functions or operators. In particular, I represents the identity operator, K an operator which forms constant-functions, and S a substitution-and-composition operator. But just what kind of operators are these? Most mathematicians think of functions as being sets of ordered pairs in some 'classical' set theory, for example Zermelo–Fraenkel set theory (ZF). To such a mathematician, I, K and S simply do not exist. In ZF, each set S has an identity- function IS with domain S, but there is no 'universal' identity which can be applied to everything. (Similarly for K and S.)

In many practical applications of CL or λ this question does not arise: as we shall see in Chapter 10 the rules for building the systems' terms may be limited by type-restrictions, and then instead of one 'univer- sal' identity-term I there would be a different term Iτ for each type- expression τ. Type-expressions would denote sets, and Iτ would denote the identity-function on the set denoted by τ.

But type-free systems also have their uses, and for these systems the question must still be faced: what kind of functions do the terms represent? One possible answer was explained very clearly by Church in the in- troduction to his book [Chu41]. In the 1920s when λ and CL began, logicians did not automatically think of functions as sets of ordered pairs, with domain and range given, as mathematicians are trained to do today. Throughout mathematical history, right through to computer science, there has run another concept of function, less precise at first but strongly influential always; that of a function as an operation-process (in some sense) which may be applied to certain objects to produce other objects. Such a process can be defined by giving a set of rules describing how it acts on an arbitrary input-object. (The rules need not produce an output for every input.) A simple example is the permutation-operation φ defined by φ(□x,y,z□) = □y,z,x□. Nowadays one would think of a computer program, though the 'operat-ion-process' concept was not originally intended to have the finiteness and effectiveness limitations that are involved with computation. From now on, let us reserve the word 'operator' to denote this impre- cise function-as-operation-process concept, and 'function' and 'map' for the set-of-ordered-pairs concept.

> Perhaps the most important difference between operators and func- tions is that an operator may be defined by describing its action with- out defining the set of inputs for which this action produces results, i. e. without defining its domain. In a sense, operators are 'partial functions'. A second important difference is that some operators have no restric- tion on their domain; they accept any inputs, including themselves. The simplest example is I, which is defined by the operation of doing noth- ing at all. If this is accepted as a well-defined concept, then surely the operation of doing nothing can be applied to it. We simply get II = I. Other examples of self-applicable operators are K and S; in formal CL we have KKxyz =w y, SSxyz =w yz (xyz), which suggest natural meanings for KK and SS. Of course, it is not claimed that every operator is self-applicable; this would lead to contra- dictions. But the self-applicability of at least such simple operators as I, K and S seems very reasonable.
>
> The operator concept lies behind programming languages such as ML, in which a single piece of code may be applied to many different types of inputs. Such languages are called polymorphic. It can be formalized in set theory if we weaken the axiom of foundation which prevents functions from being applied to themselves; see (1) in Remark 16.68. The operator concept can be modelled in standard ZF set theory if, roughly speaking, we interpret operators as infinite sequences of functions (satisfying certain conditions), instead of as single functions. This was discovered by Dana Scott in 1969; see Chapter 16. However, it must be emphasized that no experience with the operator concept, or sympathy with it, will be needed in the rest of this book.

66. We are a failure. [@c31], looking forward, seems too long, especially the chapters between the basics and type theory. We will try to use it as a reading companion to [@c13].

X.2.4 Prelude with (Stenlund)

67. At (p.15). Clearly, from now on, we should see logic and/or proof/deduction as 'simply' a special case of computation, possibly less 'expressive'.

X.2.5 History with (Cardone and Hindley 2006)

68. At (p.4).

> The von Neumann axioms eventually evolved, with many changes, including from a function base to a sets-and-classes base, into the nowadays well known Von-Neumann-Bernays-Godel system NBG. In that system the analogue of von Neumann's combinator axioms was a finite list of class-existence axioms, and the analogue of his combinatory completeness theorem was a general class-existence theorem. That system gave a finite axiomatization of set theory, in contrast to the Zermelo-Fraenkel system known nowadays as ZF. It did not, however, eliminate bound variables completely from the logic; unlike Schonfinkel, von Neumann did not have that as an aim.

69. What role does substitution play in PL?

> In propositional logic the rule of substitution is significantly more complex than modus ponens and the other deduction-rules, and at the time Curry began his studies the substitution operation was largely unanalyzed.

70. We are confirmed in our view of how the combinator completeness is seen as a game:

> He saw the problem of producing an A such that Ax1...xn converted to X, from the point of view of building X from the sequence x1,...,xn. First, A must remove from x1,...,xn all variables which do not occur in X; this can be done using K and B. Then A must repeat variables as often as they occur in X; this can be done using W and B. Next, A must re-arrange the variables into the order in which they occur in X; this can be done using C and B. Finally, A must insert the parentheses that occur in X, and this can be done using B (together with I or CKK).

71. Starting from the above, here is the impossible to discover process hidden in the removed scaffolding:

> But the details of Curry's algorithm were rather complex, and it was later re- placed by ones that were much simpler to describe. These algorithms were multi- sweep; i.e. they built [x1, . . . , xn].X by building first [xn].X, then [xn−1].[xn].X, etc., and their key was to build [x].X for all x and X by a direct and very simple induction on the number of symbols in X.
>
> The first to appear was in [Church, 1935, §3 p.278]. Church used just two basic combinators I and J, where JUXY Z = UX(UZY); the latter had been proposed by Rosser in 1933 [Rosser, 1935, p.128].9
>
> But the simple algorithm based on S and K that is used in textbooks today is due to Paul Rosenbloom [Rosenbloom, 1950, p.117]. He seems to have been the first person, at least in print, to have realised that by using S in the induction step for [x].X the proof of its main property ([x].X)Y = [Y/x]X was made trivial. This algorithm, with some variants, became generally accepted as standard.
>
> However, although neater to describe, the direct inductive algorithms were not so efficient to use: they produced considerably longer outputs than Curry's first one, and in the 1970s when the desire arose to use combinators in practical programming languages, and attention began to focus on efficiency, the algorithms that were invented then were more like Curry's original, see for example [Kearns, 1973; Abdali, 1976; Turner, 1979; Piperno, 1989]

72. This echos our thoughts about free self-application, where we think that in the passage from pure computation, where self application is totally unproblematic, by simply reusing a letter within the 'scope' of the same letter. We are happy to see that criticisms came from the semantic side, which at this point is to be fully ignored!

> A consequence of this interest in generality was that for Curry, as for Schonfinkel, every combinator was allowed to be applied to every other combinator and even to itself, in a way not commonly accepted for set-theoretic functions. This freedom was later criticized on semantic grounds by several other leading logicians, e.g. Willard V. Quine in [Quine, 1936a, p.88].

73. At. (p.6).

> However, semantic questions apart, by the end of the 1920s the combinator concept had provided two useful formal techniques: a computationally efficient way of avoiding bound variables, and a finite axiomatization of set theory.

74. It seems that 'the appearance of a paradox' is a common them in foundational systems. We read from this that there is a lot of boldness and walking long distances by 'tattonnement' involved in publishing such systems when in fact, one has no proof that they are consistent. Even more, the masters themselves of these systems work on them for years 'cluelessly' so to speak. On the other hand, it is often possible to modify a system that was suddenly realised paradoxical, so most work is not lost.

> Church's system as it appeared in [Church, 1932] was a type-free logic with unrestricted quantification but without the law of excluded middle. Explicit formal rules of λ-conversion were included. However, almost immediately after publication a contradiction was found in it. The system was revised a year later,
>
> Stephen Kleene and Barkley Rosser, and in a re- markable four years of collaboration this group made a series of major discoveries about both Church's 1933 logic and the underlying pure λ-calculus. Unfortunately for the 1933 logic, one of these discoveries was its inconsistency, as a variant of the Richard paradox was proved in the system, [Kleene and Rosser, 1935].
>
> In contrast, the pure λ-calculus, which had at first seemed shallow, turned out to be surprisingly rich. (Well, Kleene was surprised, anyway, [Kleene, 1981, p.54].)

75. We have no clue what this (fuss) is about, being or not being a term (or an active term) given the presence or absence of a free variable, we have to find out:

> (Church's name is often associated with the λI-calculus, the version of λ-calculus in which λx.M is only counted as a term when x occurs free in M. But he did not limit himself so strictly to this version as is often thought. In 1932 and '33 he allowed non-λI-terms λx.M to exist but not to be "active", i.e. he did not allow a term (λx.M)N to be contracted when x did not occur free in M, see [Church, 1932, pp. 352, 355]. In his 1940 paper on type theory, where consistency would have been less in doubt, although probably not yet actually proved, he was happy for non-λI-terms to be active [Church, 1940, pp. 57, 60]. Only in his 1941 book did he forbid their very existence [Church, 1941, p. 8].)

76. At (p.8). We read about the first fixed-point combinator.

> Incidentally, in the course of his doctoral work Turing gave the first published fixed-point combinator, [Turing, 1937b, term Θ]. This was seen as only having minor interest at that time, but in view of the later importance given to such combinators.

77. At (p.8). We have a good hint about the use of fixed-point combinators in relation to logic and a paradox, using some negation 'function' N :

> the accounts of Russell's paradox in [Church, 1932, p.347] and [Curry, 1934a, §5] both mentioned (λy.N (yy))(λy.N (yy)), where N represented negation,

78. At (p.9). Why is that? Is it because of 'total functional programming'?

> Note that although fixed-point combinators are often used in λ-defining recursion, they are not really necessary for recursion on the natural numbers; none was used in the λ-representation of the recursive functions in [Kleene, 1936].

79. It is not only us who dreamed of a type-free lambda that is consistent, Church did himself! (note yet another ghostly paradox and its fix):

> After the discovery of the Kleene-Rosser inconsistency, Church replaced his 1933 system by a free-variable logic based on the λδ-calculus, which he proved consistent by extending the Church-Rosser theorem. (In essence δ was a discrim- inator between normal forms; see [Church, 1935] or [Church, 1941, §20], and the review [Curry, 1937].) But the new system was too weak for much mathematics and seems to have played no role in Church's future work. In fact in the later 1930s Church retreated from the task of designing a general type-free logic to the much less ambitious one of re-formulating simple type theory on a λ-calculus base. In this he succeeded, publishing a smooth and natural system in [Church, 1940] that has been the foundation for much type-theoretic work since (see §5.1 and §8 below).

80. And yet another ten-year ghost paradox:

> The work of Curry through the 1930s continued his earlier study of the most basic properties of abstraction, universal quantification and implication in as general a setting as possible. He analysed the Kleene-Rosser proof of the inconsistency of Church's system, and by 1942 he had found in a logical system of his own a very simple inconsistency now called the Curry paradox [Curry, 1942b]. This showed combinatory completeness to be incompatible with unrestricted use of certain simple properties of implication in a very general setting.

81. Curry's F-combinator is very interesting to know of, and one sees how many years of trial and error went into this.

> He took a different approach from Russell and from the one that Church later took: he added to his combinator-based logic a functionality constant F, with the intention that an expression such as Fabf should mean approximately (□x)(x □ a □ fx □ b).21 This approach differed from Church's in that Curry's Fabf did not imply that the domain of f was exactly a, only that it included a. In Curry's functionality-theory a term could have an infinite number of types.

This is also a good first hint about the so-far unexplored polymorphic approach:

> Curry's approach was what we would now call polymorphic. In contrast, in a Church-style type-theory, for a term fα□β, its domain would be exactly α and its type would be uniquely α □ β.

The simplest types-as-propositions in one sentence is:

> The propositions-as-types correspondence was noticed by Curry in the 1930s. In its simplest form this is an isomorphism between the type-expressions assignable to combinators and the implicational formulas provable in intuitionistic logic.

At (p.10). We see that at least the birth of lambda was very tied to being using for logic and foundations. It is a good page to read to look at the slow seeds of the isomorphism, which slowly started slapping people in the face, a la sleepwalkers. Fitch's seems to be a type-free system, quite capable. Note that his second system took decades to develop. Also note that his doctoral system 'avoided Russellś paradox', so we feel that as before this is ghost hunting, first avoiding a known paradox, and then later trying to find a proof of consistency.

> In the 1930s another logician who made use of combinators was Frederic Fitch in Yale University. Fitch's work began with his doctoral thesis [Fitch, 1936], which described a type-free combinator-based logic that avoided Russell's paradox by for- bidding certain cases of abstraction [x1, ..., xn].X where X contained repeated vari- ables. He then developed over several decades a system CΔ, very different from the one in his thesis but still type-free and combinator-based, with unrestricted quantification and abstraction; see [Fitch, 1963]. The set of theorems of CΔ was not recursively enumerable, [Fitch, 1963, p.87], but the system was adequate for a considerable part of mathematical analysis and had a (non-finitary) consistency proof. He presented a further modified system in his student-level textbook on CL, [Fitch, 1974].

82. This quote is great, for who cares about semantics?!

> From the viewpoint of a semantics based on a standard set theory such as ZF or NBG, an advantage of algebraic logic is that it retains the restrictions of first-order logic, and therefore its semantic simplicity; as noted earlier, unrestricted type-free combinators are much harder to interpret in standard set theories.

83. What games are we playing here? Expand, ghost, retract, see how much mathematics can be expressed.

> But his system is more restricted than Tarski's, and less mathematics has been developed in it; see the review [Lercher, 1966].

84. We still don't understand the importance of 'reductions', but here is about reductions, lambda and CL, including the attractiveness of avoiding the complexity of bound variables:

> His purpose was to make later discussions of logic applicable equally to languages based on CL and on λ. Strong reduction indeed allowed this to be done to a significant extent, but its metatheory turned out to be complicated, and although simplifications were made by Roger Hindley and Bruce Lercher in the 1960s which produced some improvement, it eventually seemed that wherever strong reduction could be used, it would probably be easier to use λ instead of CL. Strong reduction was therefore more or less abandoned after 1972. However, in the 1980s a β-strong reduction was defined and studied by Mohamed Mezghiche, see [Mezghiche, 1984], and the idea of working with a combinatory reduction which is preserved by abstraction but avoids the complexity of bound variables is still tantalisingly attractive.

We read, that decades later, 'our' whim is still in the minds:

> Turning from pure to applied CL and λ: although type-theory made important advances in the 1950s, some logicians felt that type-restrictions were stronger than necessary, and that type-free higher-order logic was still worth further study. Most systems of such logic contain analogues of combinators in some form, but here we shall only mention those in which CL or λ were more prominent.
>
> Curry's work on applied systems (which he called illative, from the Latin "illa- tum" for "inferred") was published in the books [Curry and Feys, 1958] and [Curry et al., 1972]. He never gave up his interest in type-free logic, although one of his main themes of study at this time was the relation between types and terms.

85. Gentzen and type-theory, a novelty still in 1958:

> In his 1958 book with Robert Feys, perhaps the authors' most important contri- bution in the illative sections was to introduce Gentzen's techniques into type theory. As we mentioned in §4.2, this approach was not as obvious in 1958 as it is now.

86. These are good references to look at:

> The concept of partial function was also behind several substantial papers by Feferman from 1975 through the 1980s, in which he examined the possibilities for type-free systems, particularly as foundations for constructive mathematics and for category theory, and proposed several actual systems in which "partial" combinators played a role, see for example [Feferman, 1975a; Feferman, 1975b; Feferman, 1977; Feferman, 1984] and [Aczel and Feferman, 1980].
>
> An analysis of the foundations based closely on λ was made by Peter Aczel in an influential paper [Aczel, 1980].

87. More consistency ghosts:

> From 1967 onward, work on type-free illative systems was undertaken by Martin Bunder. He made a careful study of the inconsistencies that had arisen in past systems, and proposed several new systems that avoided these, including some in which all of ZF set theory can be deduced,

88. More untyped systems from Russia:

> In Moscow, independently of all the above, a series of type-free combinator- based systems was proposed by Alexander Kuzichev from the 1970s onward; see, for example, [Kuzichev, 1980; Kuzichev, 1983; Kuzichev, 1999] and the references therein.

89. CUCH is very interesting, although here too we see how long it took Bohm:

> These included Turing machines and Post and Thue systems, but the variety of these models of computation led B̈ohm in the following years to search for an inclusive formalism.
>
> But the main influence of Bo̧hm and his students on λ and CL was through discoveries about the pure systems. The presentation of CUCH in [B̈ohm, 1966] and [B̈ohm and Gross, 1966] set up a uniform language for formulating technical problems about the syntax of untyped λ and CL, and the solution of such problems dominated much of B̈ohm's later research activity

A bit later, this is also of interest:

> B̈ohm's aim of using CUCH as a real programming language led him to study various ways of coding data structures as λ-terms.

90. Dana Scott is dissatisfied:

In 1963 Dana Scott gave lectures on λ to an autumn seminar on foundations at Stanford [Scott, 1963]. These were partly motivated by "a certain lack of satisfac- tion with systems for functions suggested in papers of John McCarthy", with the hope "that the underlying philosophy will... lead to the development of a general programming language for use on computers"

91. So higher order functions are related to semantics:

Scott's next major contribution to the theory of λ was on the semantic side in 1969 and radically changed both the subject itself and its more general context, the study of higher-order functions; see §9.1 below.

92. Since we see 'semantics' as nothing but syntactics on a higher layer, we might find this interesting:

In 1971 appeared the influential thesis of Henk Barendregt, [Barendregt, 1971]. Part of its motivation was semantical, so we shall discuss it in §9.2 p.46, but its results and methods were syntactical and stimulated much work on syntactical properties of pure λ in the 1970s.

93. We meet monoids from a motivated point of view:

The quest for algebraic structure in combinatory logic led also to the study of the monoid structure of combinators.

94. It seems to have taken until 1968 for a work on 'pure' typed lambda without 'logic':

That thesis contained the first formulation of "pure" simply typed λ in isolation without the extra logical axioms and rules that the systems of Church and Curry had contained. Morris' formulation was what is nowadays called "Curry-style"; types were assigned to untyped pure λ-terms.

95. Since we do not care about models for now, we know that Boehm's theorem is something to ignore:

Morris' thesis also contained perhaps the first application of B¨ohm's Theorem, in a result which, although syntactical, later played a significant role in the model theory of λ.

96. Meanings of syntactically circular typed lambda expressions were indeed found:

Near the end of his thesis, Morris argued for the usefulness of allowing circular type-expressions in simply typed λ-calculus, like the solution of the type equation $\alpha = \alpha \to \beta$, remarking that "we have only a limited intuition about what kind of an object a circular type expression might denote (possibly, a set of functions, some of which have themselves as arguments and values)", [Morris, 1968, p.123]. The answer to this puzzle was to be found less than one year later by Dana Scott with his set-theoretic model for the λ-calculus, see §9.1 below.

97. The urge to shorten proofs exists in general, here is an example:

Ever since the original proof of the confluence of λβ-reduction in [Church and Rosser, 1936], a general feeling had persisted in the logic community that a shorter proof ought to exist. The work on abstract confluence proofs described in §5.2 did not help, as it was aimed mainly at generality, not at a short proof for λβ in particular.

98. So structural-induction comes from Tait:

> Tait's structural-induction method is now the standard way to prove confluence in λ and CL.

99. Reduction and programming:

> Returning to λβ: problems involved in making reduction more efficient began to attract attention in the 1970s, stimulated by programming needs. One such problem was that of finding a reduction of a term to its normal form that was optimal in some reasonable sense.

The same for substitution:

> Substitution was another topic which aroused interest from a computing point of view. Its correct definition had been settled years ago, but its efficient imple- mentation now became an important problem;

100. Although it does not matter syntactically (and we don't care about set-theoretic interpretations of lambda), here is a good bootstrapping about what is meant by 'type':

> there was little agreement as to what kind of entities types should be. Already in [Russell, 1903, §497] we find at least two ways in which types can be intended:
>
> (1) Types as ranges of significance of propositional functions: given a propositional function φ(x), there is a class of objects, the type of x, such that φ has a value whenever it is applied to a member of this type.
>
> (2) Types as sets: individuals form the lowest type of objects, then there are sets of individuals, sets of sets of individuals and so on, ad infinitum.
>
> The first interpretation of types is related to the theory of grammatical categories. The view of types as sets underlies, of course, the early developments in set theory, cf. [Quine, 1963, Ch. XI], but most notably from a λ-viewpoint it underlies the definition of the hierarchy of simple types in [Church, 1940] and much of the research on set-theoretical models of type theories.

101. At the moment, we also don't care about Lawvere's meaning of type, but we appreciate that this tells us that the keywords mentioned in the quote belong to 'semantics':

> (3) Types as objects in a category, especially a cartesian closed category, whose definition provides a general framework for a formulation of the set-theoretic- al bijection between functions of several arguments and unary functions with functions as values.

102. We limit ourselves to the intuitionistic (and here we already are within logic and not pure lambda) meaning of type:

> (4) Types as propositions: in either λ or CL, terms of a given type play the role of codes for proofs of (the proposition coded by) that type.

103. Categorical grammar is for now perplexing. But a useful point of view is that since it is intended to study natural language, it is again something that has to be restricted to be applied to mathematics. Just like pure computation (lambda) has to be restricted. That being said, this is telling:

> Writing x -> y to mean that every expression of category x has also category y, one can prove theorems like, for example, that xy -> z implies y -> x/z.

104. And indeed, here is a simplistic view of a restriction expressed in english:

> with the idea that "a theory of types is essentially a device for saying that certain combinations are not propositions", see [Curry, 1980, §9].

105. This is for far perplexing, but it seems approachable if one goes into a bit of detailed study:

> The grammatical interpretation of functionality was described in [Curry, 1961];47 it regarded the combination Fαβ as the category of those functors that take an argument of category α and give a result of category β, in the context of a formal system whose only operation is application. (For example, if nouns and noun-phrases have category N, then adjectives have category FNN, ad- verbs (as modifiers of adjectives) have category F(FNN)(FNN), and the suffix "-ly" (which changes adjectives to adverbs) has category F(FNN)(F(FNN)(FNN)).) The possibility of describing grammatical categories by means of types has also played a role in formalizing the syntax of logical theories within typed λ-calculi.

106. The 'Edinburgh Logical Framework' seems like a very important and noble thing to look at. It's usefulness is at the very least delimiting what is common and what is not between different logics and deductive frameworks!

107. In we read per example, and are happy about the use of canonical forms which we so like:

> The above syntax describes what we call canonical forms. Note what is not a canonical form: there is no syntactic way to apply a lambda-abstraction to an argument. Based on your past experience with programming languages, it may seem strange to define LF so that only canonical forms exist—we are not allowed to write down any programs that have any computation left to do. However, this restriction makes sense if you think about our methodology for representing object languages in LF. For example, we represent natural numbers with the following LF signature: nat : type. z : nat. s : nat -> nat. For this representation to be adequate, the only LF terms of type nat must be z, s z, s (s z), and so on. It is easy to see that non-canonical LF terms interfere with this encoding. For example, the LF term would have type nat, but it is not the representation of any informal natural number.

108. This is so far the closest to our wishful thoughts about even from the onset, being able to prove things about 'all naturals', by having a single object, not a natural but a 'type natural', that has properties, the proof being nothing but an definitional unpacking + deduction.

> The conclusion of the theorem then states a property that depends on the (free) name x, which can be instantiated to any specific object k provided that k is an integer and that we have a proof that k > 0.

continuing this quote leads us to finally some clarity about 'inhabited types' and the relation between proofs and types:

> In a line where the identifier a has the expression e of category C as definiens (in a context Γ), a has two possible interpretations: as the name of the object e of type C, or as the name of a proof e of a proposition P, provided that C can be interpreted as the type Proof(P) of proofs of P.60 Types are not formally identical to propositions; rather, to every proposition can be assigned a type, the type of its proofs, and asserting a proposition amounts to saying that the type of its proofs is inhabited [Bruijn,1970].

continuing this motivates us to check the mentioned reference:

> This paradigm of "proofs as objects" specializes to the constructive interpretation of logical constants following [Heyting, 1956] where, for example, a construction of an implication A □ B is a function that maps each construction of A to a construction of B. This is recognized in [Bruijn, 1994, p.205] to have been one of the clues to the interpretation of proof classes as types.

109. An incident where a program language gave a theoretical hint:

> It is worth remarking that the block structure of Algol led Landin to the formal analysis of that language by means of λ-calculus.

110. We find the reference to [@c18] and upon reading discover that de Bruijn simply is us! his itinerary through mathematics and his doubts are exactly hours, except that we were never forced to continue while having them, but acted on them immediately.

111. Automath is dead, long live automah. We hope to get a short path towards proofs-as-types reading automath's defunct (1999) manual.

112. A short look at https://softwarefoundations.cis.upenn.edu/current/Stlc.html reminds us of one problem. It uses 'Inductive' definitions. How are inductive types in general (and proofs about them) handled in typed lambda calculus? One introductory answer seems to be [@c19]. Indeed the answer lies in a typed version of fixed point combinators, but note the consequences. This note is **crucial**:

> Recursive types are an extremely expressive extension of the simply-typed Lambda- Calculus λ□. To see this, recall that in the untyped Lambda-Calculus, recursive functions can be defined with the help of the fixed point combinator fix = λf. (λx. f(xx))(λx. f(xx)). This term cannot be typed in λ□, as in order to do so, the sub-term x would need to have an arrow type whose domain is the type of x itself—something that does not exist in λ□. In a type system with recursive types, however, the required property is easily satisfied by the type μS. S □ T, and for every type T, a fixed point combinator can be typed as follows: fixT =λf :T □T. (λx:(μS.S □T).f(xx))(λx:(μS.S □T).f(xx)) fixT :(T□T)□T. This feature was first noticed by James Morris in 1968 (Morris 1968), one of the earliest authors to deal with recursive types in computer science. A corollary of the presence of a well-typed fixed point combinator for every type T is, that systems with recursive types do not have the strong normalisation property according to which every well-typed term should have a normal form. More specifically, they allow for infinitely many well-typed diverging functions: For every type T , a function divergeT can be defined and typed that diverges when applied to Unit:

> divergeT =λ->:Unit.fixT(λx:T.x). divergeT : Unit T. The existance of such functions also renders recursive type systems useless as logics in the Curry–Howard sense: The fact that every type T is inhabited translates to the statement that every proposition is provable.

This seems to have very good answers to the problematic exposed in the quote: https://cstheory.stackexchange.com/questions/22410. In fact it gives a good hook into the lambda cube, even mentions the 'charity' project and how it has a solution using special combinators that 'fold' and 'unfold' in a way that can even represent filters, this is important. We also remember [@c14] and its chapter on definitions, maybe there is an answer there. Indeed, (p.167) does, and this reinforces the **importance of this book**:

> Later in this book we will show that we can do without recursive definitions by making use of the descriptor ι, which gives a name to a uniquely existing entity (see Section 12.7). By not incorporating recursive definitions, we attain two goals: − we keep our system relatively simple; − we avoid the complications accompanying such recursive definitions, such as the necessity to show that each recursive definition represents a terminating algorithm with a unique answer for each input. An obvious drawback is that, in our case, a function like fac is not a program, so it cannot be executed. As a consequence, fac(3) = 6 requires a proof . For more examples of recursive definitions and of the way in which we embed these in our type-theoretic system, see Section 14.4 (about integer addition) and Section 14.11 (about integer multiplication).

Another possible source of answers is one of our trigger books: [@c8], with its second chapter 'Types and Induction'.

113. In our search to build automah on top of untyped lambda calculus, and then typed lambda calculus on top of that, we searched for semantics be it denotational or small-step of the latter with respect to the former. We found this insightful comment (https://en.wikipedia.org/wiki/Simply_typed_lambda_calculus#Operational_semantics):

> The presentation given above is not the only way of defining the syntax of the simply typed lambda calculus. One alternative is to remove type annotations entirely (so that the syntax is identical to the untyped lambda calculus), while ensuring that terms are well-typed via Hindley–Milner type inference

114. At this stage, we now understand why some of our starting books were overly cryptic. They were focusing on logic, and even though historically this might be how the lambda calculus and the combinators were born, the sober and non-esoteric (stolen from 'charity') path is by first studying those without logic, and only then adding logic to them. This applies to books such as [@c20], [@c8], [@c21], [@c22]. At the same time, it was extremely helpful to do all the exercises in the first part [@c23] of for it taught us the essential part of logic that are mentioned everywhere and saved us from going in loops.

115. Illative logic 'is' 'our naive' and Church and Curries dream [@c24]:

> One nice thing about the ILC is that the old dream of Church and Curry came true namely there is one system based on untyped lambda calculus or combinators on which logic hence mathematics can be based More importantly there is a combinatory transformation between the ordinary interpretation of logic and its propositions-as-types interpretation.

XI Reboot with ("An Overview of Type Theories")

116. This book is sobering, it foreshadows the realisation that pure STLC has been 'eaten' by algebra (category theory). We step back and look at our original goals. Even though we still would like to formally solve inductive definitions as they are in set theory, we feel it might be nice to go back to [@c26] as a survey, and search where the consistency of intuitionistic logic is proved, and let this take us back to (@c25, p.15), even though this again mixes pure lambda with logic. In automah, the system should be logic agnostic as De Bruijn envisaged. Maybe the most general system is simply that of boxes (Taylor), of 'rewriting systems', of 'two dimensional deductive notation'. The only thing that always stands in the way of this is 'did we find a system that is logically consistent?' not seen as a huge disaster, but simply as a question to be answered. And can we build mathematics on top, with standard ways of doing mathematics requiring at least some sort of consistency.

> In intuitionistic logic, the semantics is given by the Brouwer-Heyting-Kolmogorov interpretation: truth is identified with provability.

117. Mathematics is about proving. Informal mathematics is not mathematics. What needs to be proven about a logic is, if necessary, that it is consistent. It is OK to use a stronger system to prove something about a weaker sub-system. This is but an edge in a graph. It 'proves' something as opposing to proving that something is wrong about the weaker sub-system. And to emphasize, there is no need for 'semantics' in mathematics.

118. As a comment of (@c26, p.13) 'the meaning of meaning' we simply say there is no meaning, only rules of games, and records of playing them.

119. At (p.35) we read:

> Thesis 1. Any correct and complete assertion expressible in a certain part of intuitionistic type theory can be demonstrated by means of the inference rules justified in that part of intuitionistic type theory.

120. At (p.36) we read:

> Here I have used the phrase law of logic as more or less synonymous with inference rule ; this is clearly an oversimplification of how the phrase is usually understood, but it fits well with intuitionistic type theory.

121. We don't care about such informal meta justifications, because we do not care about the system to be used in automah, even inconsistent systems are welcome

> Note that this inference rule is well-formed since the presupposition of the conclusion, that A & B is a proposition, follows from the presuppositions of the premisses, i.e., that A and B are propositions

122. One more reason to ignore justifications is pages like (p.40).

123. We must still go back to (p.41-44) in the sense of them being the link that bootstraps the formal (IPC) from the informal. As we concluded, the 'problem' is that the world itself does not present itself to us formally, so there is always the first bootstrapping. This would be the post-goal, after having solved with automah the formal side.

124. Even if we only care about syntax, the use of ‘variables’ and ‘schematic letters’ is crucial, per example in the operational/denotational sense. At (p.48,49) it would be nice in automah to really control these ‘variables’ and treat them as nothing but schematic letters with ‘inference rules’ (meta) specifying how they are ‘transformed’/‘substituted’.

125. Probably, the best way to deal with definitions on the lower levels is not to have them ... Replace them with inference rules (like Hilbert systems?)

126. Given automah’s unbiased view, we even disagree with this:

> On the other hand, when they are employed for beings of reason (ens rationis), they are very problematic, since they presup- pose a kind of Platonic universe of ideas. I will call the definition of a complete form of assertion a meaning explanation. That is, in intuitionistic type the- ory, descriptive definitions are avoided and replaced by meaning explanations, thereby avoiding the existence problem.

This is not ‘problematic’, only that in set theory, the object as described by relations, may or may not exist as a set, and that is to be proven if judged necessary. As to the whole system being only consistent by ‘belief’, ‘interpolation’ and ‘practice’ and ‘probably patchable if not’, that is another question.

127. We even don’t care about extensional vs. intentional, we only care about this being implicitly expressed using the formal two-dimensional inference rules:

> There are two important distinctions to be made. First, with respect to the meanings of the terms, one should make a distinction between intensional and extensional notions of set, class, etc. If a concept is taken as standing for the totality denoted, then it is taken extensionally ; whereas if the identity of the concept itself is also taken into account, then it is taken intensionally.

The being said, this still seems meaningful:

> Thus, for a notion of set, class, etc. to be intensional, equality between sets has to mean something more than mere coincidence of the extensions of the concepts.

128. Out of all the other ’definitions’ this is the most sobre one. One can see this already by the choice of words. Surely, one can also build the ’mathematical’ versions of the others in this, for this is the essence of computation, free of semantics, to which the other ’definitions’ must reduce themselves sooner or later.

> A set A is defined by prescribing how a canonical element of A is formed as well as how two equal canonical elements of A are formed...There is no limitation on the prescription defining a set, except that equality between canonical elements must always be defined in such a way as to be reflexive, symmetric and transitive

Cantor’s on the other hand is a joke:

> By a ‘set’ we understand every collection M of definite well-distinguished objects m of our intuition or thought (which are called the ‘elements’ of M) to a whole.

Bishop's is also nice:

> A set is not an entity which has an ideal existence : a set exists only when it has been defined. To define a set we prescribe, at least implicitly, what we (the constructing intelligence) must do in order to construct an element of the set, and what we must do to show that two elements of the set are equal

The classical definition (not intuitionistic) was at least one devoid of meaning:

> Here a real definition of the notion of set is given up entirely and instead the containment relation ε is taken as primitive. In such an axiomatic formulation of set theory, it is difficult to say exactly what it means for something to be a set. By set, one is free to understand anything, as long as it satisfies the axioms ; or even more boldly, the notion of set is defined by the axioms. The path taken below, in attempting a real definition of the notion of set, is based on another philosophy, viz., that the definition comes first and that the axioms have to be valid in virtue of the definition.

129. The author's grand definition of a set is a joke, until it is formalized.

> A set is a universal concept, a first intention, collected to a whole, i.e., substantiated, where the objects falling under it, called elements of the set, are considered as individuals, and for which the definitional equality between objects satisfies the four requirements : that a definitum is always equal to its definiens, that two objects of the same form are equal if their parts are equal, that any object is equal to itself, and that two objects which equal a third object are equal to one another.

130. A simple example to hook into what the author means by canonical elements (e.g of a set):

> Only canonical sets and elements are defined by this definition. I say canonical to distinguish a term which immediately refers to a set or element from a term which refers only through computation, which, in this context, is called a noncanonical set or element. This distinction applies to expressions and to their meanings, but not to the objects denoted by the expressions.57 For example, 4 is a canonical decimal number, but 2 + 2 is a noncanonical decimal number.

131. A simple short explanation of impredicativity (including that it can be vague) is:

> Both Definition 3 and the above definition are predicative in the sense of Russell and Poincare. In Definition 3, predicativity is expressed by the phrase first intention and in the above definition by the phrase without reference to the totality of sets. A predicative definition is noncircular, or well-founded ; the notion defined is not in any way presupposed in its definiens. One could say that impredicativity in definition corresponds to a vicious circle, or petitio principii, in demonstration.59 The condition of predicativity is necessarily somewhat vague: some set forming operations are such that one has to make up one's own mind whether to accept them or not.

Note that by now we can understand how defining natural numbers using the successor is not circular: each natural, instance of type natural, is defined by referencing some other natural. The type natural in the exact sense, does not refer to the type natural!

132. This is very nice because it ties to our desire to express everything on top of pure computation by limiting the rules of the games using inference rules:

> The first two parts of the definition do not give rise to any inference rules. For example, there is no inference rule like [A ε set]|[it is defined what a ε el(A) means], simply because the conclusion is not of a form subject to type-theoretical treatment. Still, granted that A is a set, it must be defined what the forms of assertion a ε el(A) and a = b ε el(A) mean, for this particular set A.

133. At least one conclusion of (p.65) is that if one tries to bootstrap formal set theory using two-dimensional inference rules, one probably is already implicitly using types? Probably also necessarily explicitly? How does one limit comprehension? Also, probably the seemingly most basic notion of ‘membership’ can only be stated implicitly, per example through rules for union, etc. At least when we try not to use quantifier logic at this early stage?

134. We still don’t understand these seemingly meaningless:

> the assertions 1 = 1 ε el(B) and 0 = 0 ε el(B) are both meaning determining.

Until we understand this, we bypass this chapter and jump to ‘Reference and Computation’.

135. We should keep Dirlichet’s function in mind and the difference in attitude towards it (p.77), as we read this chapter and also in general. It hooks CPC, IPC, computability, decidability, canonical elements, all together.

136. This is interesting:

> The third feature of computation, which is easy to overlook, is that even though the value of the computation is not known beforehand, it is always known what type of value to expect : we can have a number valued computation, a Boolean valued computation, etc., but never simply a computation.

137. Luckily, also here, we do not care about the informal concept of ‘algorithm’:

> But we are of the opinion that these constructions were only introduced in order to provide a formal characterization of the informal concept of algorithm. Thus the concept itself was recognized as existing independently from this formal characterization and as preceding in time.

138. A short contrast between ‘function’, ‘program’, ‘algorithm’, although we are not sure about what is meant with ‘definiteness’:

> Recall the difference between program, function, and algorithm : they have the characteristics input, output, and exactness in common, function adding finiteness, program adding definiteness, and algorithm adding both.

But (p.81) tells us that we need to study an introduction to recursion theory.

139. We find this [@c28] puzzling, since what does defining the divergent function have to do with logic:

> In type theories such as those of Coq [5], Isabelle/HOL [15] or HOL [14], all functions must be total. In particular, all recursive functions must be provably terminating. If this was not the case, one could define the diverging function “let recfx = 1+fx”, and then simplify the equality “fx = 1+fx” towards “0 = 1”, which would be an obvious inconsistency. Thus, the definition of recursive functions must be restricted in some way.

Having said that, at some point we need to reach understanding of this:

> Type theories need to enforce some restrictions on recursive definitions in order to remain sound. Depending on the implementation, these restrictions may prevent the user from defining recursive functions as conveniently as in a functional programming language. This paper describes a fixed point combinator that can be applied to any functional. A fixed point equation can be derived for the recursive function produced, provided that all recursive calls are made on arguments that are smaller than the current argument, with respect to a decidable well-founded relation or a measure. The approach is entirely constructive, and does not require the user to program with dependent types. It supports partial functions, n-ary functions, mutual recursion, higher-order recursion and nested recursion. It has been implemented and experimented in Coq.

This is also nice:

> A decreasing-measure argument is a simple and powerful technique for establish- ing that a recursive function terminates on all input. The principle is to exhibit a particular function, called the measure, that computes the size of an argument, and to verify that all recursive calls are made on arguments of smaller size than the current argument.

140. We note that IPC will not 'deliver us' from diagonal arguments. At the same time there is hope since Lawvere's fixpoint theorem seems like an acceptable and digestible alternative. http://math.andrej.com/2007/04/08/on-a-proof-of-cantors-theorem/

141. The 'Munich Project CIP' (pdfs can be found with exactly this name, or with ISBNs 3540151877, 9783540151876) seems to be very close to some ideas we had [@c29].

142. As opposed to our decision above, can we skip studying recursion theory and get away with reading these very short introduction? we hope so.

- Chapter 4 of [@c27]
- First part of [@c30]

XII Recursion Theory with (Fernández 2009) and (*Randomness and Undecidability in Physics*)

143. Here is a hook:

> Primitive recursive functions play an important role in the formalisation of computability. Intuitively speaking, partial recursive functions are those that can be computed by Turing machines, whereas primitive recursive functions can be computed by a specific class of Turing machines that always halt. Many of the functions normally studied in arithmetic are primitive recursive. Addition, subtraction, multiplication, division, factorial, and exponential are just some of the most familiar examples of primitive recursive functions. Ackermann's function, which we will define in Section 4.1, is a well-known example of a non-primitive recursive function.

144. In which sense are division or exponential primitive recursive, i.e can be computed by Turing machines that always halt? It seems that what is meant is division on the naturals.

145. In which sense is then Ackerman's function not primitive recursive? We need to understand this. This seems helpful: https://math.stackexchange.com/questions/96483 and this https://www.quora.com/Why-cant-Ackermann-function-be-calculated-iteratively

146. Interestingly, we find in [@c30] a clear statement mirroring our motivations for automah. We read **Formal Systems correspond to Computable Processes** (p.30), and **The essence of inference. At least for a formalized theory, *is string processing*[391].. Let thus the inputs of the computation correspond to axioms. The program correspond to the rules of inference, and the output to proved theorems.** Now why is this not 'enough'?

147. The author of [@c30] surprises us in Chapter 2 by treating exactly our vague questions about the relation of proving to what he calls 'evolution determinism' on a continuum.

148. The author of [@c30] keeps surprising us. (p.124-126) are gold.

149. Maybe we should stop at (p.126) and write an article: 'The art of paradox using fixed point combinators' or 'The paradox creator s toolbox'. Since that seems to be what is intrinsically rubbing us. The feeling that we could never invent such paradoxes. We will spend a week on this project. We did this and the result is in 'paradox.pdf'.

XIII Miscellanea

150. There is a latent misconception that 'natural numbers' and 'natural'. Because of this, the reason we never understood ordinals is that we were looking at the set-theoretic construction through the lens of 'naturals numbers and then ...'. This is enough to disallow the right concept. Since set theory is used to subsume mathematics, it also subsumes all known number kinds. One should then simply look at how a specific set of numbers is encoded in set theory. 'Clearly', this does not exhaust all order, the ordinals take that job.

151. Natural numbers are data. The problem of bootstrapping numbers is not only mathematical. We cannot count up to six without already knowing what six is, somehow. Therefore in a minimal lambda calculus, we cannot recurse from six down without first having built six as an argument to count down from. Possibly a functional version of $S(S(...(0)..)$. In that sense, natural numbers are 'data'. That being said, because the lambda calculus has syntax, one can encode data into the syntax! This reminds vaguely of initial conditions of dynamical systems, with are the 'data' of the dynamical process (computation).

152. We arrive at the idea that textbooks, even the basic ones, already do more than we want. If we have a formal way to describe how we 'compute' proofs, very low level, including a formalization of the syntactic operations, bokos already prove things about that formal system, informally. We would like to withhold such proofs, simply describe the machinistic part based on informal axioms of machinistic leprechaun-free repeatability. Only after that do we proceed to develop a formal language in which to prove things about the formal syntactic procedures. This seems to be the most sane 'bootstrapping circularity' one could wish for.

153. In this context above, it could be that even natrual numbers are not needed 'directly', but an informal machinistic way to distinguish symbols from each other, in a deterministic manner. Recursive/inductive steps are handled by isolation and the no-Leprechaun assumption. Is this realistic?

154. Terms as Proofs. In [Type Theory and Homotopy, Steve Awodey] we read:

> Martin-Loef type theory is a formal system originally intended to provide a rigorous framework for constructive mathematics [ML75, ML98, ML84]. It is an extension of the typed λ-calculus admitting dependent types and terms. Under the Curry-Howard correspondence [How80], one identifies types with propositions, and terms with proofs; viewed thus, the system is at least as strong as second- order logic, and it is known to interpret constructive set theory [Acz74].

We would really like to see this correspondence. We find two good resources on this: [@c9] and [@c10]. In [@c9] we read:

> But there is much more to the isomorphism than this. For instance, it is an old idea—due to Brouwer, Kolmogorov, and Heyting, and later formalized by Kleene's realizability interpretation—that a constructive proof of an implication is a procedure that transforms proofs of the antecedent into proofs of the succedent; the Curry-Howard isomorphism gives syntactic representations of such procedures.

We would really like to see the syntactic representations. The nice thing about [@c9] is that it starts with type-free lambda calculus. We hope these two resources makes [@c8] (which is a goal) more approachable by acting as prerequisites. We hope that we know enough to understand the well-ordering and other ordering ideas presented in it, linking us to set theory.

155. defdefqed: The 'I can compute' can be seen in the computer, the physics is what makes it possible, proof can only be circular, the machine language is untyped! untyped lambda? to do 'logic' we move to typed! it is all there already... (the 'form' is only sensible once we have a type in mind see [Finitism] (p.8) ... float to int pointer cast)

156. Trees, books, proofs, stories, understanding, grows by scaling, not by starting and proceeding to finish. This is why we will not only explicitly (or maybe better implicitly) write our books this way, but also study books this way, by reducing them in scaling and not linearly to their seeds, and growing them again while injecting the added detail into the understanding that is then being grown in parallel.

Bibliography

Abramsky, Samson. "The Lazy Lambda Calculus." https://www.cs.ox.ac.uk/files/293/lazy.pdf.

"An Overview of Type Theories."

Barbanera, Franco. "Short Introduction to Functional Programming and Lambda Calculus." http://www.dipmat.unict.it/~barba/PROG-LANG/PROGRAMMI-TESTI/READING-MATERIAL/ShortIntroFPprog-lang.htm.

Barendregt, Henk. 1997. "The Impact of the Lambda Calculus in Logic and Computer Science." ***Bulletin of Symbolic Logic*** 3 (2). Cambridge University Press: 181–215.

Barendregt, Henk, Wil Dekkers, and Richard Statman. 2013. ***Lambda Calculus with Types***. Cambridge University Press.

Barendregt, HP. 1993. "Lambda Calculi with Types, Handbook of Logic in Computer Science (Vol. 2): Background: Computational Structures." Oxford University Press, Inc., New York, NY.

Bauer, Andrej, Martín Hötzel Escardó, and Alex Simpson. 2007. "Lecture Notes in Computer Science." Springer Berlin Heidelberg.

Blaheta, Don. "The Lambda Calculus, Notes." http://cs.brown.edu/courses/cs173/2002/Lectures/2002-10-28-lc.pdf.

Bruijn, Nicolaas Govert de. 1994. "Reflections on Automath." ***Studies in Logic and the Foundations of Mathematics*** 133. Elsevier: 201–28.

Cardone, Felice, and J Roger Hindley. 2006. "History of Lambda-Calculus and Combinatory Logic." ***Handbook of the History of Logic*** 5. Elsevier, Amsterdam, to appear: 723–817.

Charguéraud, Arthur. "Proof Pearl: A Practical Fixed Point Combinator for Type Theory."

Chiswell, Ian, and Wilfrid Hodges. 2007. ***Mathematical Logic***. Vol. 3. OUP Oxford.

Curry, Haskell Brooks. 1963. ***Foundations of Mathematical Logic***. Courier Corporation.

Dershowitz, Nachum, and Jean□Pierre Jouannaud. "REWRITE SYSTEMS." http://www.cs.tau.ac.il/~nachum/papers/survey-draft.pdf.

Fernández, Maribel. 2009. ***Models of Computation: An Introduction to Computability Theory***. Springer Science & Business Media.

Girard, Jean-Yves, Paul Taylor, and Yves Lafont. 1989. ***Proofs and Types***. Vol. 7. Cambridge University Press Cambridge.

Goldberg, Mayer. "The Lambda Calculus Outline of Lectures." http://www.little-lisper.org/website/files/lambda-calculus-tutorial.pdf.

Guerrini, Stefano, Adolfo Piperno, and Mariangiola Dezani-Ciancaglini. 2008. "Bohm's Theorem."

Hindley, J Roger. 1997. ***Basic Simple Type Theory***. Vol. 42. Cambridge University Press.

Hindley, J Roger, and Jonathan P Seldin. 2008. ***Lambda-Calculus and Combinators: An Introduction***. Vol. 13. Cambridge University Press Cambridge.

Hodel, Richard E. 2013. ***An Introduction to Mathematical Logic***. Courier Corporation.

Kfoury, Assaf J, Robert N Moll, and Michael A Arbib. 2012. ***A Programming Approach to Computability***. Springer Science & Business Media.

Kuhlmann, Marco. "Recursive Types." https://www.ps.uni-saarland.de/courses/seminar-ws02/RecursiveTypes.pdf.

Larson, Jim. "An Introduction to Lambda Calculus and Scheme." http://www.cs.unc.edu/~stotts/723/Lambda/scheme.html.

Nederpelt, Rob, and Herman Geuvers. 2014. ***Type Theory and Formal Proof: An Introduction***. Cambridge University Press.

Randomness and Undecidability in Physics.

Rojas, Raul. "A Tutorial Introduction to the Lambda Calculus." http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf.

Stenlund. ***Combinators, Lambda-Terms and Proof Theory***.

Stuart, Tom. 2013. ***Understanding Computation: From Simple Machines to Impossible Programs***. " O'Reilly Media, Inc."

Sørensen, Morten Heine, and Pawel Urzyczyn. 2006. ***Lectures on the Curry-Howard Isomorphism***. Vol. 149. Elsevier.

Taylor, Paul. ***Practical Foundations of Mathematics***.

"Treatise on Intuitionistic Type Theory."

Troelstra, Anne Sjerp, and Helmut Schwichtenberg. 2000. ***Basic Proof Theory***. Vol. 43. Cambridge University Press.