Proofs and Representations

Laura Pausini, Alice Pascoe*

November 29, 2012

Abstract

We outline a formal proof of the falsity of Church's thesis: that the effectively calculable functions are the lambda definable functions. We suggest an alternative formulation with far greater potential for non-deterministic reasoning.

Introduction

The intuitive reason why Church's thesis is false is the simple fact that whether or not a function is effectively calculable depends crucially upon what it is *supposed* to calculate. That is to say, it is the *intensional* definition of the function which tells us whether it is effectively calculable or not, and this intensional definition is only ever given in respect to the *denotational* semantics of the values it calculates. So in order for us to be able to *know* that we have effected a calculation, we need to know the meaning of the values which result. Any function which is so 'defined' that the meanings of the values it calculates are ambiguous is clearly not an effectively calculable function; for how could we possibly have known how to effect the calculation, and how should we know how to interpret the result? Clearly then, if we are to consider it to have been defined at all, a function must have a definite value for any definite values of its arguments, and its semantic or denotational definition should reflect this.

The Problem

First we present a very simple illustration which shows up some of the problems with interpreting lambda calculus in terms of syntactic normal forms. This leads on to the contradictory interpretation which we present in the section which follows this one.

*e-mail c/o iang@pobox.com.

The 'standard diagonalisation' produces a representational or *syntactic* fixed point μF of any lambda term F which takes a lambda representation $\ulcorner M \urcorner$ of the Gödel representation $\lceil M \rceil$ of arbitrary lambda terms M as its only argument. Thus for all terms F we can effectively present a term μF such that

$$\mu F = F \ulcorner \mu F \urcorner. \tag{1}$$

The fixed point is defined $\mu F \equiv W \ulcorner W \urcorner$ where $W \equiv \lambda x.F\,(\mathbf{A}\,x\,(\mathbf{R}\,x))$, and $\mathbf{A}$ and $\mathbf{R}$ are terms which satisfy

$$\mathbf{R}\,n = \ulcorner \mathbf{n} \urcorner \qquad \text{and} \qquad \mathbf{A} \ulcorner M \urcorner \ulcorner N \urcorner = \ulcorner M\,N \urcorner, \tag{2}$$

where $\mathbf{n}$ is the Church numeral representing the number represented by the argument n which is assumed to be a Church numeral. Crucially, the function represented by the lambda term $\mathbf{R}$ must produce the Gödel representation of numbers in *principal normal form,* which is to say they must use some standard numeric representation such as Church numerals, with standard binding variables, f and x, say, in that order, and they must always be in normal form. So we stipulate that the double Quine quotes denoting Gödel representations in the object language denote *only* Church numerals in this canonical form.

We use the very simple Gödel representation $\lceil M \rceil$ defined inductively on the structure of lambda terms M and N and variables $l \in \Sigma = \{a, b, c, \ldots\}$ indexed by the enumeration $i : \Sigma \to \{1, 2, 3, \ldots\}$

$$\begin{aligned} \lceil l \rceil &= 2^{i(l)} \\ \lceil \lambda l.M \rceil &= 2^{i(l)} \times 3^{\lceil M \rceil} \\ \lceil M\,N \rceil &= 3^{\lceil M \rceil} \times 5^{\lceil N \rceil}. \end{aligned}$$

For example, supposing $i(f) \mapsto 1$ and $i(x) \mapsto 2$ then

$$\lceil \mathbf{1} \rceil = \lceil \lambda f.\lambda x.f\,x \rceil = 2^1 \times 3^{2^2 \times 3^{3^2 \times 5^{2^2}}} = 2 \times 3^{4 \times 3^{5{,}625}}.$$

To formally define $\Lambda M.\ulcorner M \urcorner$ we define the following recursive functions on $\mathbb{N} = \{1, 2, 3, \ldots\}$

$$\begin{aligned} V_x &= 2^{i(x)} \\ L_x(m) &= V_x \times 3^m \\ A(m, n) &= 3^m \times 5^n. \end{aligned}$$

where $i(x)$ refers to the index number of the variables $x \in \Sigma$.

Now we can present the function $\mathbf{R}\,n = \ulcorner \mathbf{n} \urcorner$ defined in (2) by the primitive recursive function

$$R(n) = L_f(L_x(S(n))), \quad \text{where} \quad S(n) = \begin{cases} A(V_f, V_x) & \text{if } n = 1, \\ A(V_f, S(n-1)) & \text{if } n > 1. \end{cases} \tag{3}$$

Next we define a lambda term $\mathbf{R}$ which we can prove is denotationally equivalent to the recursive function R. That is, we need to be able to prove that $[\![\mathbf{R}\,\mathbf{n}]\!] = R(n)$ whenever $\mathbf{n} \equiv \lambda f.\lambda x.f^n\,x$.

We will assume the same lambda definitions of the arithmetic functions for multiplication and exponentiation which Paulson gives in his [13].

$$\mathbf{MUL} \equiv \lambda m.\lambda n.\lambda f.\lambda x.m\,(n\,f)\,x \qquad \mathbf{EXP} \equiv \lambda m.\lambda n.\lambda f.\lambda x.n\,m\,f\,x$$

Now we define lambda terms

$$\begin{aligned}\mathbf{V}_x &\equiv \mathbf{EXP}\ \mathbf{2}\ \mathbf{i}_x\\ \mathbf{L}_x &\equiv \lambda m.\mathbf{MUL}\ \mathbf{V}_x\,(\mathbf{EXP}\ \mathbf{3}\,m)\\ \mathbf{A} &\equiv \lambda m.\lambda n.\mathbf{MUL}(\mathbf{EXP}\ \mathbf{3}\,m)(\mathbf{EXP}\ \mathbf{5}\,n)\end{aligned}$$

to implement the operations in the definition of $\Lambda M.\ulcorner M \urcorner$ given above, where $x \in \Sigma$ and $\mathbf{i}_x$ denotes the Church numeral corresponding to the index $i(x)$ of the variable x in Σ. Then it easy to see that the following lambda-defines

$$\mathbf{R} \equiv \lambda n.\mathbf{L}_f\,(\mathbf{L}_x\,(n\,(\lambda x.\mathbf{A}\,(\mathbf{V}_f)\,x)\,\mathbf{V}_x)).$$

Now we define a metalanguage functional $\mu \equiv \Lambda F.\Lambda W = \lambda x.F\,(\mathbf{A}\,x(\mathbf{R}\,x)).W\ulcorner W \urcorner$ which denotes the representational fixed point of any object language lambda term F. Here we are using $\Lambda l = M.N$ to represent local binding of the metalanguage variable l to the expression M in N. We have presented μ as a metalanguage operator because it requires the Gödel representation of the function f and it is impossible to lambda define the function which computes this unless one already has a representation of the required function. In the metalanguage this is always available.

Equational reasoning in the denotation of the object language terms gives

$$\begin{aligned}\mu F &\equiv (\lambda x.F\,(\mathbf{A}\,x(\mathbf{R}\,x)))\ulcorner W \urcorner && \text{Defn. } \mu F\\ &= F\,(\mathbf{A}\ulcorner W \urcorner(\mathbf{R}\ulcorner W \urcorner)) && \beta\text{-conv.}\\ &= F\,(\mathbf{A}\ulcorner W \urcorner\ulcorner\ulcorner W \urcorner\urcorner) && (*)\ \text{Defn. } \mathbf{R} \text{ in (2)}\\ &= F\ulcorner W\ulcorner W \urcorner\urcorner && \text{Defn. } \mathbf{A} \text{ in (2)}\\ &\equiv F\ulcorner \mu F \urcorner. && \text{Defn. } \mu F\end{aligned} \tag{4}$$

The point $(*)$ is where we assume the principal normal form of Gödel encoded Church numerals $\Lambda N.\ulcorner N \urcorner$. Now assume we have a single-step *metacircular evaluator* term $\mathbf{\Sigma}_1\ulcorner M \urcorner = \ulcorner N \urcorner$ which, given a Church numeral m representing the Gödel representation $\ulcorner M \urcorner$ of a lambda term M, performs one beta conversion step, and returns the Church numeral $\ulcorner N \urcorner$ representing the Gödel representation of the resulting term N. If the term M is in normal form then it just returns the normal form. If the term m does not represent the Gödel representation of a lambda term then it returns just m. So $\mathbf{\Sigma}_1$ represents a total function on $\mathbb{N}$, and it is in fact primitive recursive.

Consider the syntatic fixed point $\mu\Sigma_1$ of Σ_1. By the equational reasoning in the denotation of terms, from the second line of (4), this reduces in one step to $\Sigma_1 N$, where N *effectively presents* a Church numeral $N = \ulcorner \mu\Sigma_1 \urcorner$. So $\mu\Sigma_1$ certainly has a normal form and indeed $\Sigma_1 \ulcorner \mu\Sigma_1 \urcorner = \ulcorner \Sigma_1 N \urcorner$. Then, still by equational reasoning in the denotation of terms, substituting $N = \ulcorner \mu\Sigma_1 \urcorner$ in the RHS of this, we obtain

$$\mu\Sigma_1 = \Sigma_1 \ulcorner \mu\Sigma_1 \urcorner = \ulcorner \Sigma_1 N \urcorner = \ulcorner \Sigma_1 \ulcorner \mu\Sigma_1 \urcorner \urcorner \tag{5}$$

which is impossible because these are each distinct Church numerals in normal form.

Now of course this does not contradict the Church-Rosser theorem: it is a *denotational semantic* inconsistency, not an operational inconsistency. But it is no different in kind to the 'contradictions' which are the conclusions of the standard computability no-go proofs such as the one Paulson presents so clearly in [13]. The difference is that here we have the semantic contradiction arising from a perfectly well-defined, well-typed primitive recursive function Σ_1.

Berry's Paradox Formalised

Berry's paradox was attributed by Russell as being due to G.G. Berry (1867–1928) who was a Librarian at the Bodleian Library in Oxford. It originally concerned the least undefinable ordinal. Informally, the paradox appears in the definition of the *smallest* number *not* definable in *less* than a certain amount of space, which happens to be *more* space than it takes to say this. This is impossible, of course.

It is worth thinking about the intuitive reason for the paradox, which is clear even informally. It is simply that, like all paradoxes, the definition refers impredicatively to itself. In particular the reference it makes is to the size of the definition itself, which can be done if one can find a certain *fixed point,* so that when the definition mentions its own size, it does so in a consistent manner. This is the essential source of the paradox: the sentence accurately describes some aspect of itself. Such sentences are called *Quine sentences* after the American logician W.V.O. Quine. The ordinal numbers are quine sentences because they denote their own names, but expressions like $1+2$ do not.

We will use a formal definition of Berry's paradox in terms of Gödel numbers of lambda terms which calculate Gödel numbers. Accordingly, we define the set R of Gödel representations of terms which are themselves calculations of the Gödel representations of terms. So for example, R includes the Gödel representation $\ulcorner \mathbf{L}_x \mathbf{V}_x \urcorner$ of a lambda term which calculates the Gödel representation $\ulcorner \lambda x.x \urcorner$. It also includes the Gödel representations $\ulcorner \mathbf{L}_f (\mathbf{L}_x (\mathbf{A} \mathbf{V}_f \mathbf{V}_x)) \urcorner$ and $\ulcorner \mathbf{R}\,\mathbf{1} \urcorner$ of terms which both calculate $\ulcorner \mathbf{1} \urcorner$ given as an example above. The set R also includes Gödel representations Quine terms which calculate their own Gödel numbers. For example, R is closed under $\lambda n.\mu n\,\mathbf{I}$, so there is no shortage of such terms which calculate their own Gödel representation.

Now the set B of *potential* Berry's numbers is any subset of R which is closed under Quine quotes in the sense that $\ulcorner N \urcorner \in B$ if and only if there is some $\ulcorner M \urcorner \in B$ such that $N = \ulcorner M \urcorner$. This includes not only Gödel representations like $\ulcorner \mathbf{Q} \urcorner$ of 'atomic' Quine terms $\mathbf{Q} = \ulcorner \mathbf{Q} \urcorner$, but also those of terms which calculate Gödel representations of Quine terms.

Thus we formally define the set B by induction as the least set of Gödel numbers $\ulcorner L \urcorner$ and $\ulcorner M \urcorner$ of lambda terms L and M closed under the rules

$$\mu\colon \frac{\Lambda \ulcorner L \urcorner . L = \ulcorner L \urcorner}{\ulcorner L \urcorner \in B} \qquad \nu\colon \frac{\Lambda \ulcorner L \urcorner . \ulcorner L \urcorner \in B \qquad \Lambda \ulcorner M \urcorner . M = \ulcorner L \urcorner}{\ulcorner M \urcorner \in B} \tag{6}$$

Now we can state Berry's paradox as the definition of the number calculated by a term in B which is the least such not calculable by any term less than or equal to *itself*. The absurdity is immediately apparent in that we are looking for a term in B not calculable by a term equal to itself, but the equality relation *is* the denotational device for terms in B because the only terms in B are terms which calculate the Gödel representations of terms in B, either their own or others. By choosing this as the domain we have restricted the Berry numbers to just those terms which are representable in space equal to themselves: in other words, to Quine terms. So there is no term in B not representable by a term equal to itself.

To prove this formally, let $\mathbf{I}_n \equiv \Lambda n . \mu \mathbf{n} \mathbf{I}$ so that $\mathbf{I}_n = \ulcorner \mathbf{I}_n \urcorner$ for all $n \in \mathbb{N}$ and let $\mathbf{I_A} \equiv \mathbf{I}_1$ and $\mathbf{I_B} \equiv \mathbf{I}_2$. Then we define a function $\mathbf{B}$ over the set $\{\ulcorner \mathbf{I_A} \urcorner, \ulcorner \mathbf{I_B} \urcorner\} \subseteq B$ which is the characteristic function of Berry's number

$$\mathbf{B} L M \ulcorner N \urcorner = \begin{cases} L & \text{if } N > L; \\ M & \text{otherwise.} \end{cases} \tag{7}$$

We say that $\mathbf{B}$ is lambda-definable because it is possible to write a total recursive function which, given the Gödel representation of any lambda term with a normal form, converts to a Gödel representation of that normal form. In the case of terms in B these normal forms always exist and are always Church numerals, so to decide $N > L$ it is sufficient to lambda define a term $\mathbf{\Omega}_B$ which, for $\ulcorner N \urcorner \in B$, effects $\Lambda \ulcorner N \urcorner . N$, the inverse operation of the Gödel representation $\Lambda N . \ulcorner N \urcorner$, and to compare the resulting Church numeral N with L in the metalanguage.

Thus motivated, we make the condition of *effective presentability* of any domain B, which is the existence of a lambda term $\mathbf{\Omega}_B$ which presents the terms M whose representations $\ulcorner M \urcorner$ are in B. This is a formal statement which, on the domain B, is equivalent to Church's thesis:

$$\Lambda \mathbf{\Omega}_B . \Lambda \ulcorner M \urcorner \in B . \mathbf{\Omega}_B \ulcorner M \urcorner = M. \tag{8}$$

We use the following terms from [13] defining logical and arithmetical constants

$$\mathbf{T} \equiv \lambda f . \lambda x . f \qquad \mathbf{F} \equiv \lambda f . \lambda x . x \qquad \mathbf{S} \equiv \lambda n . \lambda f . \lambda x . n f (f x)$$

Then we have the lambda defined predecessor function $\mathbf{P}$ in terms of the successor function $\mathbf{S}$ for Church numerals and a term representing $\mathbf{Q}\langle m,n\rangle = \langle \mathbf{S}\,m, m\rangle$

$$\mathbf{Q} \equiv \lambda p.\lambda f.f\,(\mathbf{S}(p\,\mathbf{T}))(p\,\mathbf{T}) \qquad\qquad \mathbf{P} \equiv \lambda n.(n\,\mathbf{Q}\,(\lambda f.f\,\mathbf{F}\,\mathbf{F}))\mathbf{F}$$

Now we define $\mathbf{B} \equiv \lambda l.\lambda m.\lambda n.(l\,\mathbf{P}\,(\mathbf{\Omega}_B\,n))(\mathbf{T}\,l)\,m$. Then let $\mathbf{\Gamma} \equiv \mathbf{B}\,\mathbf{I_A}\,\mathbf{I_B}$ so we have

$$\mathbf{\Gamma} \equiv \lambda n.(\mathbf{I_A}\,\mathbf{P}\,(\mathbf{\Omega}_B\,n))(\mathbf{T}\,\mathbf{I_A})\,\mathbf{I_B}.$$

which, given the condition (8), has the denotational semantics of $\mathbf{B}\,\mathbf{I_A}\,\mathbf{I_B}$ as defined in (7), that is

$$\mathbf{\Gamma}\ulcorner N \urcorner = \begin{cases} \mathbf{I_A}, & \text{if } ((\mathbf{\Omega}_B\ulcorner N\urcorner) - \mathbf{I_A}) \geq 1 \\ \mathbf{I_B}, & \text{otherwise.} \end{cases}$$

Now we consider the value of $\mu\mathbf{\Gamma} = \mathbf{\Gamma}\ulcorner\mu\mathbf{\Gamma}\urcorner$ and ask "Is $\ulcorner\mu\mathbf{\Gamma}\urcorner$ in B?" Because $\mathbf{I_B} > \mathbf{I_A}$ the lambda term $\mu\mathbf{\Gamma}$ is equal to $\mathbf{I_A}$ only if it is greater than $\mathbf{I_A}$, and equal to $\mathbf{I_B}$ only if it is less than $\mathbf{I_B}$ so $\mathbf{\Gamma}$ has contradictory semantics over the set B so it is not an effectively calculable function. Again using equational reasoning in the denotation of the object language terms:

$$\begin{aligned} \mu\mathbf{\Gamma} &= \mathbf{\Gamma}\ulcorner\mu\mathbf{\Gamma}\urcorner && \text{By (4)} \\ &\equiv (\lambda n.(\mathbf{I_A}\,\mathbf{P}\,(\mathbf{\Omega}_B\,n))(\mathbf{T}\,\mathbf{I_A})\,\mathbf{I_B})\ulcorner\mu\mathbf{\Gamma}\urcorner && \text{Defn. } \mathbf{\Gamma} \\ &= \mathbf{I_A}\,\mathbf{P}\,(\mathbf{\Omega}_B\ulcorner\mu\mathbf{\Gamma}\urcorner)(\mathbf{T}\,\mathbf{I_A})\,\mathbf{I_B} && \beta\text{-conv.} \\ &= \mathbf{I_A}\,\mathbf{P}\,\mu\mathbf{\Gamma}\,(\mathbf{T}\,\mathbf{I_A})\,\mathbf{I_B}. && \text{By (8)} \end{aligned}$$

But $\mu\mathbf{\Gamma} = \mathbf{\Gamma}\ulcorner\mu\mathbf{\Gamma}\urcorner$ by (4) so, given (8), the lambda definition of the function $\mathbf{\Gamma}$ must decide the value of $\mathbf{\Gamma}\ulcorner\mu\mathbf{\Gamma}\urcorner$ before it is determined. Clearly the semantic definition of $\mathbf{\Gamma}$, though lambda definable, does not effectively determine the values for every point in the domain.

Therefore over the set B of potential Berry's numbers, $\mathbf{B}$ is lambda-definable but it is not effectively calculable, so the lambda-definable functions are not the effectively calculable functions, therefore Church's thesis is false. Because the lambda calculus is deterministic, this is provable in arithmetic. Therefore any automated theorem proving system which assumes Church's thesis will be inconsistent, because it will derive a contradiction from assuming that $\mathbf{B}$ is effectively calculable.

The contradiction is only possible if we *know* the denotation of the numbers, and this then allows us to lambda-define the function $\mathbf{\Omega}_B$ which is an *effective presentation* of the function represented by a given Church numeral, which is taken to represent a lambda term over the set B defined by (6). But this knowledge is *accidental*, not essential to the lambda calculus or to arithmetic. If we did not know the Gödel representation then we would not be able to lambda define $\mathbf{\Omega}_B$ and we would not have the condition (8) of effective presentation. But the particular Gödel representation we choose is arbitrary, so it is only because of a coincidence between the syntax of representations and the object language presentation of the function $\mathbf{\Omega}_B$ that we have the possibility of producing unbounded recursion. But this correspondence is not

something that any object language lambda term could effectively detect; it is something that is essentially syntactical. But having restricted the universe of discourse to just those lambda terms (6) with representations in B, we *can* make the assumption of a particular Gödel representation. Then we have inconsistent denotation.

What Went Wrong?

Clearly the semantics of self-representation are not consistent. When we extend the domain of a language to include the interpretation of numbers as Gödel representations of sentences of the object language then we have the possibility of such *syntactic fixed points,* and they may result in either contradictory or non-contradictory denotational semantics. The fixed point Turing presents in [17, 18] is such a non-contradictory circular dependency in the denotational semantics of *a*-machines.

The reason we call these *syntactic* fixed points is that, although they appear to be denotational in nature, they are not. The denotation must always be the consistent operational semantics otherwise there would be no point in using formal systems at all. On closer inspection one invariably finds that the guaranteed consistency of the underlying formalism 'sorts out' the denotational inconsistencies and the result is always a perfectly consistent denotational *and* operational semantics. What one finds is that when the inconsistent denotational definitions occur, the corresponding lambda-definition turns out to represent *another* function, which *necessarily* is a total computable function. The precise sense in which we need to extend the semantics is in *depth* of representation. Any fixed point $\Lambda F.\mu F = F\ulcorner\mu F\urcorner$ always has consistent semantics once it has represented itself twice, i.e. at $F\ulcorner\mu F\urcorner = F\ulcorner F\ulcorner F\ulcorner\mu F\urcorner\urcorner\urcorner$. This semantics is capable of resolving non-determinism because the middle one of the three nested representations effectively contains any potential contradiction as a superposition.

So it is not *inconsistency* that is the cause of these denotational contradictions, but *incompleteness of the model,* in the sense that we extend the denotational semantics without extending the model accordingly and the result is that the model is incomplete in so far as it does not fully determine the semantics. The same thing appears to happen in Gödel's incompleteness theorem: the extended denotational semantics of Gödel representations of proofs as arithmetical formulæ produces undecidable propositions because the *model* is not able to distinguish between apparently contradictory semantics. What is the extended model? It is an entirely artificial, essentially *syntactic* domain—like the Herbrand base, but extended beyond just the ground terms. It consists in an enumerable set of well-formed sentences of the object language.

Why is Church's Thesis False?

Church's thesis states that the effectively calculable functions are the lambda definable functions, or something provably equivalent such as the functions computable by Turing *a*-machines. This is false, and for *two* reasons. Firstly, there are effectively calculable functions which are not lambda definable. One such is the function we presented as $\Lambda M.\ulcorner M \urcorner$ which calculates the Gödel representation of an arbitrary lambda term. This is not lambda definable because the semantics of the lambda calculus are 'up to' extensionality, so no lambda term can 'see' the syntactic structure of any lambda term. The crucial distinction to be made is that between the *presentation* of a function and its *representation*. The lambda calculus as an object language deals only with the presented functions. As a metalanguage however, it deals with representations as lambda terms which represent Gödel representations of lambda terms. The fixed point operator μ we presented is an example: though effectively calculable, it is not lambda definable, so it must always be a metalanguage term. But once *given* a representation of any presented lambda term, then we can effectively calculate its fixed point.

But even if we strengthen the thesis to say that the effectively calculable functions are just those lambda definable functions which are *lambda representable*, then there are lambda representable functions such as Σ_1 which are not lambda definable, but are nevertheless effectively calculable. We say they are not lambda definable because the lambda term that is supposed to represent the defined semantics does not in fact do that. This is what we demonstrated when we showed in (5) that $\mu\Sigma_1 = \ulcorner \Sigma_1 \ulcorner \mu\Sigma_1 \urcorner\urcorner$ is denotationally inconsistent. The reason is that *effective presentation* is not lambda representable because it necessarily involves knowing the meaning of representations: it cannot be decided by syntactic considerations alone, because the existence of syntactic fixed points like those produced by the operator μ will always defeat any such characterisation.

The problem for HOL [4], and indeed any other formal system based on a logical framework of intuitionistic type theory, is that the type of the domain does *not* in fact secure the semantics of functions from all possible contradiction, precisely because of the problem of effective presentation not being lambda representable. Whatever function we have lambda defined over one type has consistent semantics, not by virtue of the syntactic form of the function itself, but by virtue of the *domain* semantics. So when we consider the same syntactic form over a different type of domain then we do not necessarily have consistent semantics. The function Σ_1 for example, considered as an arithmetical function over the domain of the natural numbers $\mathbb{N}$ is a total function. Considered as a *functional* over the domain of numerical representations of arithmetic functions it does *not* have decidable semantics.

As soon as we extend the denotational semantics from the *presentation* to the *representation* of terms, we have indeterminacy, which is inconsistency as far as the unextended denotational semantics is concerned. But we can extend the denotational semantics by deep embedding and this can always be done consistently.

Incompleteness in Systems of Formal Proof

The idea of effective presentability is the foundation of Gödel's incompleteness theorems [6]. Here we represent negation by the upright Greek N, universal quantification by Π_1 and formula substitution by existential quantification as Σ_1. Gödel's statement of the conclusion[1] reads:–

> Let any primitive recursively defined class $\varkappa$ of formulæ be given. Then if the formal decision (from $\varkappa$) of the proposition-formula $\Pi_1(17, r)$ is also given, one could effectively present:
>
> 1. A proof of $\mathrm{N}(\Pi_1(17, r))$.
> 2. For any given n, a proof of $\Sigma_1(r, 17, \ulcorner n \urcorner)$.
>
> In other words, a formal decision for $\Pi_1(17, r)$ would imply the effective presentability of a proof of ω-inconsistency.

Therefore what the theorem produces is *not* a specific formula which cannot be proved: it is an effective method which, *given* some formal proof of the Gödel sentence, *could* produce a formal proof of contradiction, or a counter-example. That method Gödel calls an *effective presentation,* and it is *potential,* not actual.

Our use of $\mathbf{\Sigma}_1$ to represent single-step beta conversion is no accident. Gödel's incompleteness theorem has an isomorphic structure with our proof, where $\mathbf{\Sigma}_1$ is the operation which effects 'quasi diagonalisation' in Gödel's proof. So Gödel proves $\mathbf{\Sigma}_1 \ulcorner \mu\mathbf{\Sigma}_1 \urcorner = \ulcorner \mathbf{\Sigma}_1 \ulcorner \mu\mathbf{\Sigma}_1 \urcorner \urcorner$ where in (5) we prove $\mathbf{\Sigma}_1 \urcorner\urcorner \mu\mathbf{\Sigma}_1 \urcorner\urcorner = \urcorner\urcorner \mathbf{\Sigma}_1 \urcorner\urcorner \mu\mathbf{\Sigma}_1 \urcorner\urcorner\urcorner\urcorner$. Both proofs operate in essentially the same way: they extend the interpretation of the domain to include self-representation of functions on that domain, and then they produce a syntactic fixed point which is *apparently* operationally undecidable. However this is only if one *assumes* that the function in question has definite denotational semantics. In fact the formula actually *presented* is a representation of a *different* function on the same domain, but one with decidable denotational semantics.

Gödel effectively implements the primitive recursive single-step beta-reduction functional by using the rule of cut to represent variable binding, and then defining capture-avoiding substitution as a Σ_1^0 function—i.e. a primitive recursive arithmetic function with a single existential quantifier—on the domain $\ulcorner P \urcorner$ of numbers construed as Gödel representations of formal proofs P. On this domain the semantics of the function Σ_1 are undecidable, despite it being a well-defined primitive recursive arithmetic function on the domain $\mathbb{N}$ of numbers construed as, well ..., as *numbers* of course!

The contradiction hinges on the assumption of calculability, by that *particular* metamathematical proof system, of effective presentation; for there is no other way to interpret the denotational semantics of the *representations* of functions in order to get a contradiction with the semantics of the functions as they are *presented*.

[1] This is paraphrasing Martin Hirzel's English translation, which is the only one we have access to. It would be very good if a logician who is a fluent German speaker could, in the light of what we have written here, make a very careful English interpretation of Gödel's original text freely available.

Classes of Computable Functions

We use the symbol Λ to represent the lambda definable functions, with the subscript *rep* to indicate the lambda representable functions. And we use Γ to represent the effectively calculable functions. Then what we have shown, albeit only informally, is

$$\Lambda \subset \Lambda_{rep} \subset \Gamma. \tag{9}$$

Let Γ_π be the class of primitive recursive fuctions.Then let Γ_ι be the class of higher order recursive functionals of finite type I—those computable by Gödel's System **T**—and let Γ_ρ be the class of total recursive functions. Let Γ_α be the class of Turing *a*-machine computable functions, and let Γ_μ be the class of general recursive functions computable using Kleene's μ-operator for computing inverse functions. Finally, let Γ_γ be the class of *c*-machine computable functions—those computable by Turing machines equipped with an axiomatic choice operator. Then we have, by the established results of Church [3], Kleene, Turing [17, 18], Tait [16] and others, that $\Gamma_\pi \subset \Gamma_\iota \subset \Gamma_\rho \subset \Lambda = \Gamma_\mu = \Gamma_\alpha \subseteq \Gamma_\gamma$. Now define the class Γ_Γ of *Gödel representable* functions as the extensional equivalence class of the intersection of the all the classes of functions calculable using any formalism which is provably equivalent to *at least* the class Γ_ι of primitive recursive functionals of finite type I on $\mathbb{N}$. So we have $\Gamma_\iota \subseteq \Gamma_\Gamma$.

Then we formally state the *Gödel-Church-Turing thesis* as the inequality

$$\Gamma_\Gamma \subseteq \Gamma. \tag{10}$$

This means that the effectively calculable functions are *at least* the Gödel representable functions. Then we extend the *Gödel-Church-Turing hypothesis*

$$\Gamma_\gamma \subseteq \Gamma_\Gamma \tag{11}$$

which states that the Gödel representable functions are at least the functions computable by non-deterministic choice machines. Turing has shown that $\Gamma \subseteq \Gamma_\gamma$, so if we could prove (11) then we would have

$$\Gamma_\Gamma \subseteq \Gamma \subseteq \Gamma_\gamma \subseteq \Gamma_\Gamma$$

so $\Gamma = \Gamma_\Gamma$, and that would be a formal proof of the Gödel-Church-Turing thesis.

Many will at once think, as we did, that we would thereby have restricted the effectively calculable functions to the class Γ_ι and therefore that there are total recursive functions and general recursive functions which are not effectively calculable, and that this is therefore a contradiction. But if they think again then they will find that they are wrong.

What then of the inequality $\Gamma_\iota \subset \Gamma_\rho \subset \Lambda = \Gamma_\mu = \Gamma_\alpha \subseteq \Gamma \subset \Gamma_\gamma$ apparently established by Church, Kleene, Turing and others? This is only provable assuming the strong form of the Church-Turing thesis, and the consequent effective presentability of Gödel representations, which we have shown is false.

```
1  val present =
2      fn (Mu x) => x
3  val representation =
4      fn thing =>
5         thing (Mu thing)
6  val effectpresentation =
7      fn thing =>
8         fn repr'n =>
9            thing ((present repr'n) repr'n)
10 val presentation =
11     fn thing =>
12        fn repr'n =>
13           fn value =>
14              effectpresentation thing repr'n value
15 val effectself =
16     fn thing => representation (presentation thing)
17
18 val prefib = fn fib => fn n => if n<3 then n else fib(n-2)+fib(n-1)
19 val fib = effectself prefib
20 val r = fib 6
```

Figure 1: Unbounded recursion in the theory of polymorphic types

Type Polymorphism

Polymorphic type systems are described in some detail in [11]. The following effects general recursion in Standard ML using a *recursive datatype.* It comes from Harrison's [10] lecture notes[2].

```
datatype 'a mu = Mu of 'a mu -> 'a
```

This `datatype` declares a new *type constructor* called `Mu`, to which Standard ML assigns the type `'a Mu = fn: ('a mu -> 'a) -> 'a mu` which is to say that it is a *polymorphic function* of type $\forall\alpha.((\alpha)\texttt{mu} \to \alpha) \to (\alpha)\texttt{mu}$. This can be used to construct *type isomorphisms* such as $(\tau)\texttt{mu} \simeq (\tau)\texttt{mu} \to \tau$, which are *instances* of the polymorphic type $\forall\alpha.(\alpha)\texttt{mu}$. Here the variable τ is not universally quantified and this indicates that the type is some particular instance, though we don't care to say which one because it doesn't matter for now. The Standard ML function `fib n` in figure (1) constructs the Fibonacci numbers.

The function `present` defined on line 1 is called a *deconstructor*. As constructors construct values of the given type from values of some type, which may be this type or some other; the deconstructors effect the converse operation: they each take a value of the type and return some part of it. This deconstructor *presents* the function that was *represented* using the constructor. The *binder* `effectself` is essentially a Standard ML representation of *effective self presentation,* otherwise known as *recur-*

[2] §6.4.4, p. 72. *The Subtlety of Recursive Types.*

sion. But this cannot be done directly, because ML is a typed language. So we do it indirectly in *Looking Glass Land,* so to speak, through the binder of *reflection.*

The types of the values so constructed are all isomorphisms of $(\alpha)\texttt{mu} \simeq (\alpha)\texttt{mu} \to \alpha$ so ML prints their names as, say `'a mu` but they should be thought of as a recursive unfolding $((\ldots((\alpha)\texttt{mu} \to \alpha)\ldots \to \alpha) \to \alpha) \to \alpha$ of the isomorphism to an arbitrary and *indeterminate* depth. Thus we can use `Mu` as a *binder* in a fixed point combinator and so define a recursive function without using any of Standard ML's built-in type constructors `fun` and `val rec` for recursive functions. This is what the function `effectself` does. By the formal process described in [9], Standard ML infers the type $\forall\alpha.\forall\beta.((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta$ for `effectself`, which should be read from the outside in: so it takes a function of type $(\alpha \to \beta) \to \alpha \to \beta$ called a *pre-fixed point* and it returns a function which is a *least pre-fixed point,* and that means it is a *least fixed point* with the type $\alpha \to \beta$. So when we apply it to the pre-fixed point `prefib`, defined on line 18 which is a lambda term of type $(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int}$ then we get a term `fib` on line 19 which has a type $\texttt{int} \to \texttt{int}$ of the fixed point, which is a function which refers to *itself* by its argument `fib`. That is to say that the function which is the pre-fixed point has been *diagonalised.* So *reflection* is merely *representation* which is what happens on line 3, but *effecting one's self* or *autopoesis,* which is what happens on line 15, is something entirely different.

In detail: the function `presentation` does almost nothing: it is an *eta-expansion* of `effectpresentation` and its sole purpose is to prevent Standard ML's so-called *eager evaluation* from effecting the presentation of the function to itself *until* there is actualy something for it to do. This makes ML's evaluation of `effectpresentation` *lazy*. Without it the process would be an infinite regress of self presentation. The function `effectpresentation` does the work. It takes a *thing,* which is a function, and a representation of something, and it presents the thing with the self-representation of the thing represented. This is like the lambda term $W \equiv \Lambda F.\lambda x.F\,(\mathbf{A}\,(x\,(\mathbf{R}\,x)))$. Finally the function `representation` is the equivalent of $\Lambda F.W(F)^{\ulcorner}W(F)^{\urcorner}$. What makes all this possible is the *injection* of the set of functions on functions on $\mathbb{N}$ into the set of functions on $\mathbb{N}$, as witnessed by the type $\forall\alpha.\forall\beta.((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta$ of `effectself`. This is impossible if the set of all functions $F{:}\mathbb{N} \to \mathbb{N}$ is considered *extensionally,* but it can be effected by *intensional* representation of `x` by `Mu x`.

We have been using Martin-Löf's notation of *operators and arities* to describe abstract types where $(\alpha_1, \alpha_2, \ldots, \alpha_n)\texttt{op}$ denotes a *type operator* called `op`, of *arity* n, acting on types denoted by *type variables* $\alpha_1, \ldots, \alpha_n$. Thus the type constructor $(\alpha)\texttt{mu}$ is a function which has a special status as a constructor, not of *values*, but of *types*. Having defined this *type operator* we have a whole new *genus* of types $\forall\beta_1, \ldots, \beta_m.(\sigma)\texttt{mu}$, each of which is called a *generic instance* of the *polymorphic type scheme* $\forall\alpha.(\alpha)\texttt{mu} \simeq (\alpha)\texttt{mu} \to \alpha$ and has an associated recursive type constructor `Mu`. Substituting any concrete type τ for α in the *abstract type constructor* called $(\alpha)\texttt{mu}$ gives a new *concrete* type constructor `Mu` of type $((\tau)\texttt{mu} \to \tau) \to (\tau)\texttt{mu}$ for constructing *values* of the *type isomorphism* $(\tau)\texttt{mu} \simeq (\tau)\texttt{mu} \to \tau$. But instances need not be so-called *monotypes,* without any type variables, they may themselves be *whole* new polymorphic type-schemes, and these are the generic instances we referred to above. For example we

could have the polymorphic type-scheme $\forall\alpha.\forall\beta.(\alpha\times\beta)\,\mathtt{mu}\simeq(\alpha\times\beta)\,\mathtt{mu}\rightarrow\alpha\times\beta$. Generic instances are solely to enable *let-polymorphism,* as described in [10].

If a type-scheme τ is a generic instance of the type-scheme σ, then one writes the *order of the genera* as $\tau<\sigma$. So a generic instance or *species,* though *more complex* in structure, is *less abstract* than the simpler genus of which it is an instance. Thus in Milner's theory of type polymorphism, one says that type-schemes are *downward-closed* under generic instantiation. So each generic instance is derived by substituting a generic instance of another type-scheme for a universally quantified type variable in some generic instance. The *most* abstract genus is the *function-space* type constructor $\forall\alpha.\forall\beta.(\alpha,\beta)\rightarrow$, effected by the *binder* called *lambda.* Thus *every type is a function.*

An Historical Diversion

The following is from Quine's *Variables Explained Away* [14].

> As x increases, we are told, $2/x$ decreases. Since numbers never increase or decrease, such talk of variables must be taken metaphorically. The meaning of this example is of course simply the general statement that if $x>y$ then $2/x<2/y$. Indeed logicians and mathematicians nowadays use the word 'variable' mostly without regard to its etymological metaphor; they apply the word merely to the essentially pronominal letters 'x,' 'y,' etc., such as are used in making general statements and existence statements about numbers. A characteristic use of such letters is seen in the generality prefix 'every number x is such that,' followed by some sentence, usually of the conditional or 'if-then' form, containing the letter 'x.' Another characteristic use of such letters is seen in the existence prefix 'some (at least one) number x is such that.'

The 'pronomial' sense of the word *variable* is the sense in which it is a *pronoun* such as *he, she, they* or *it.* For example one might say "If any *animal* is pink then *it* is a pig." One might formalise this in a propositional calculus as "$\forall Animal.\mathrm{Pink}(Animal)\Rightarrow\mathrm{Pig}(Animal)$." Here the variable is a *pronoun* or *denoting phrase* which *binds* to the denotation. If the universe consisted entirely in pigs, we could express "Some *pigs* are such that *they* are baby, and *they* are pink" by writing "$\exists Pigs.\mathrm{Baby}(Pigs)\wedge\mathrm{Pink}(Pigs)$."

Let us now go back to the early 20th century and recall Russell's paradox which is supposed to have put paid to Frege's hopes of establishing a sound logical foundation for the theory of classes in his *Begriffschrift.* The problem was Frege's failure to distinguish between the properties or *attributes* of a thing and the *thing itself.* So by denoting classes by their characteristic predicates P, Russell was able to construct the paradox of the class R of classes which do not denote themselves from what perhaps seems to be an obscure, but harmless definition $R(P)=\neg P(P)$. Now this denotes *all* those classes which do *not* denote themselves, but this is itself a class, so consider the denotation of $R(R)=\neg R(R)$ which is contradictory. So Russell's predicate R can distinguish Russell's predicate R from itself, which is a sign that all is not well with

its sense of identity of the things in its semantic domain. It was this which led Russell to develop the ramified theory of types used in *Principia Mathematica.*

We cannot resist pointing out how apt are Quine's two most well-known *bons mots*, one of which is "to be is to be the value of a variable," Quine's other well-known slogan is "no entity without identity," This says that we ought be able to distinguish entities one from another, and in addition, we ought *not* to be able to distinguish them from *themselves.* Compare this with the definition of the ML function `effect-self` on line 15 on page 11. This is indeed what we demand of any denoting phrase: that it denote what it actually represents, just like the term $\mu\Sigma_1$ *doesn't!*

Here is a bit of computer science folklore which has somewhat dubious veracity, we think, however we would never let something as trivial as *truth* get in the way of a good story, so we repeat it at every opportunity. Legend has it that Curry's paradoxical lambda combinator $\mathbf{Y}$ has its origin in the encoding of Russell's paradox as a lambda term. Generalising the predicate to an arbitrary function f we get $\mathbf{Y} \equiv \lambda f.(\lambda x.f\,(x\,x))\,\lambda x.f\,(x\,x)$, and for all terms f we have $\mathbf{Y}\,f = f\,(\mathbf{Y}\,f)$, so $\mathbf{Y}$ is a fixed point combinator. Now as anyone who took the trouble to look up Harrison's [10] will have seen, the ML term `effectself` is almost the same thing, *viz.*

```
fun Y f =
   let val x =
       fn (repr'x as (Mu x)) =>
          fn v => (f (x repr'x)) v
   in x (Mu x)
   end
```

If we cease to distinguish between the thing itself and the representation of the thing then we use just `x` and we don't need `repr'x` or `Mu x`, and if we assume normal order reduction of lambda terms so that we don't need the `fn v =>...v` eta expansion, then the terms $\mathbf{Y}$ and `Y` are operationally identical. The existence of the fixed point combinators and general recursion in the untyped lambda calculus is what led Church to develop the simple theory of types [4] and so, via Scott's typed logic for computable functions, to Milner's theory of polymorphic types and the development of the ML language, and then on to the further development of *LCF* and *HOL* logics by Paulson, Gordon, Melham *et al*. Some of this history is sketched in Gordon's [8].

But let's step back still further, to 1886, when the third Earl Russell was just sixteen years old, and Lewis Carroll published a children's book [2], *The Game of "Logic"*. In the introduction he wrote:

> But . . . a remark has to be made—one that is rather important, and by no means easy to understand all in a moment: so please do read this *very* carefully.
>
> The world contains many *Things* (such as "Buns," "Babies," "Beetles," "Battledores," etc.); and these things posess *Attributes* such as "baked," "beautiful," "black," "broken," etc.: in fact, whatever can be "attributed to", that is "said to belong to", any Thing, is an Attribute.

> Whenever we wish to mention a Thing we use a *Substantive*: when we wish to mention an Attribute, we use an *Adjective*.
>
> People have asked the question "Can a Thing exist without any Attributes belonging to it?" It is a very puzzling question and I'm not going to try to answer it: let us turn up our noses, and treat it with contemptuous silence, as if it really wasn't worth noticing. But if they put it another way, and ask "Can an Attribute exist without any Thing for it to belong to?", we may say at once "No: no more than a baby could go on a railway-journey with no one to take care of it!" You never saw "Beautiful" floating about in the air, or littered about on the floor, without any Thing to *be* beautiful, now did you?
>
> And now what am I driving at, in all this long rigmarole? It is this. You may put "is" or "are" between the names of any two *Things* (for example, "some Pigs are fat Animals"), or between the names of two *Attributes* (for example "pink is light-red"), and in each case it will make good sense. But if you put "is" or "are" between the name of a *Thing* and the name of an *Attribute*, (for example, "some pigs are pink"), you do *not* make good sense, for how can a Thing *be* an Attribute?) unless you have an understanding with the person to whom you are speaking. And the simplest understanding would, I think, be this—that the substantive shall be supposed to be repeated at the end of the sentence, so that the sentence, if written out in full, would be "some Pigs are pink (Pigs)." And now the word "are" makes quite good sense.

In other words, $\exists Pigs.\text{Pink}(Pigs)$, or "to be (a Pink Pig, in this case) is to be the value of a variable, and consequently to have, as well as a *name,* at least *one* attribute." A little type-casting and a well-chosen binder go a long way! Let's go yet further back in time, to Greece, in around 350 BCE, when Aristotle [1] wrote:

> A term which is repeated in the premisses ought to be joined to the first extreme, not to the middle. I mean for example that if a syllogism should be made proving that there is knowledge of justice, that it is good, the expression 'that it is good' (or '*qua* good') should be joined to the first term. Let A stand for 'knowledge that it is good', B for good, C for justice. It is true to predicate A of B. For of the good there is knowledge that it is good. Also it is true to predicate B of C. For justice is identical with a good. In this way an analysis of the argument can be made. But if the expression 'that it is good' were added to B, the conclusion will not follow: for A will be true of B, but B will not be true of C. For to predicate of justice the term 'good that it is good' is false and not intelligible. Similarly if it should be proved that the healthy is an object of knowledge *qua* good, of goat-stag an object of knowledge *qua* not existing, or of man perishable *qua* an object of sense: in every case in which an addition is made to the predicate, the addition must be joined to the extreme.

This is probably more easily explained initially using a more down-to-earth context, such as the attributes of pigs, for example. The term that is repeated in the premisses is the binder or *middle term,* and in all these examples it appears as a pronoun. So

Quine may say with Lewis Carroll that "There is knowledge of *pigs*, that *they* are pink (pigs)" or more in line with Aristotle, he might say "There is knowledge of pigs *qua* pink things."

NOTA BENE: As witnessed by the above paragraph, formal logic is *not* a deterministic calculus of operations for *constructing* proofs or truths. Formal logic is an analysis of a *whole* argument by a process of *deconstruction* and the aim is *not* to establish truth, but to understand the reasons *why* something is necessarily or possibly true, false or contradictory. The denotation of a formal proof is *not* truth, it is *knowledge*.

Aristotle continues:

> The position of the terms is not the same when something is established without qualification and when it is qualified by some attribute or condition, e.g. when the good is proved to be an object of knowledge and when it is proved to be an object of knowledge that it is good. If it has been proved to be an object of knowledge without qualification, we must put as middle term 'that which is', but if we add the qualification 'that it is good', the middle term must be 'that which is something'. Let A stand for 'knowledge that it is something', B stand for 'something', and C stand for 'good'. It is true to predicate A of B: for *ex hypothesi* there is a science of that which is something, that it is something. B too is true of C: for that which C represents is something. Consequently A is true of C: there will then be knowledge of the good, that it is good: for *ex hypothesi* the term 'something' indicates the thing's special nature. But if 'being' were taken as middle and 'being' simply were joined to the extreme, not 'being something', we should not have had a syllogism proving that there is knowledge of the good, not that it is good, but that it is; e.g. let A stand for knowledge that it is, B for being, C for good. Clearly then in syllogisms which are thus limited we must take the terms in the way stated. Aristotle *Prior Analytics* I.38.

It is very clear that Aristotle's use of 'something' is in the pronomial sense of a variable binding, and it is also very clear that it is a *metalanguage* binding. In order to understand this it helps to know that Aristotle holds that demonstrative or *scientific* knowledge is only possible when the basic truths are 'appropriate' to the genus of that science, and that there must necesarily be more than one science. By 'appropriate' he means something very close to our modern use of the notion of well-typed. Then predication is *not* syntactically characterised because it has a different sense for every sense in which the predicate could be interpreted according to the type of the subject. Hence Aristotle's frequent use of the formula '*X qua Y*' to make explicit the particular sense of predication; this could be thought of as a kind of type annotation. Then the discusson here is about the way in which logical inference can cross the boundary between two types. Of particular interest is how we can come to know *first* that something exists in an unqualified sense, without having any specific knowledge regarding that thing. To anyone who has found the patience to read this far, this ought by now have begun to seem horribly familiar. When we wish to argue the existence of unqualified knowledge, we must predicate the *knowledge* of the thing that we wish to prove exists because we cannot prove something exists without mentioning that thing. For example, we cannot prove that the inhabitants of a type σ exist by

predicating something concerning their denotation alone, otherwise we could prove the existence of unicorns, goat-stags, and even completely implausible things such as the centre of mass of the Solar System, which is *in principle* unknowable.

To make this inference, according to Aristotle, we must *predicate knowledge* of the existence of the things themselves; i.e. we must state precisely the nature of our knowledge of *which* type σ is a substitution-instance in the particular sense in which we are here predicating something of it, be that its existence as pure being, its existence as a representation of a particular subtype of lambda representation, or as the pre-image of an element of some particular representing type under some particular representing function.

One may get many, many examples of these different kinds of representation from any textbook on Category Theory. Categories are *potential deconstructions.* Robin Milner once advised one of us: "You have to know a *bit* about Category Theory because when you make something they always want to know what Category it is." Unfortunately we have not been able to *follow* this advice. We find that we have a problem with Category Theory which is that "one can't see the trees for the forest." The reason is that, in almost all of the texts we've seen, with a few remarkable exceptions such as Rutten and Jacob's excellent introduction [15], the theory is presented as a construction, not an analysis. When someone starts explaining Category Theory, the audience invariably[3] begin desperately searching for the intuitive basis on which to interpret what they hear, and sometimes it seems as if the aim of a really top-class Category Theorist is to get right through her exposition without *ever* mentioning *anything* that could be given *any* kind of interpretation as a *model* of *any* kind. But it doesn't have to be like this; interpretation is *good*; if we can put something into our own words, then we know that we really understand that thing, and perhaps more importantly, it gives us, and others, an opportunity to *test* that understanding.

So the Category Theorists *shouldn't* be asking *us* to tell *them* what Category is the thing we have made before they admit it as a valid construction; rather, *they* should be *interpreting* our constructions in the language of Category Theory, and then *they* could tell *us* what Category these things are. Furthermore, we think we can quite directly explain why interpretation of the Categories is the *essence* of Category Theory. This is the source of the remarkable degree of generality of the results: they are a consequence of the fact that there can be *no* concrete denotation whatsoever for the language. So the only sense in which a Category could be given a meaning is in respect of its translation from one intensional syntactic representation to another. Thus Category Theory is purely abstract knowledge and its sole claim to existsence is as a syntactic or *formal* translation from one intensional language to another. Category Theory is based on the idea that mathematics is not actually the study of actual things, it is the study of their abstract properties. So we dispense with the things themselves entirely and just study their properties. One says then that an object X is only an object in so far as it has properties Y and Z. But how can one speak about the properties without mentioning the object? One can speak in parables. So one never

[3] An extrapolation based, we admit, on rather limited survey—with a sample-population of one.

ever actually mentions anything except by drawing correspondences with something else. What is the first object that all the correspondences started with? It turns out never to have existed, yet. One reckons that eventually everything will be understood in terms of categories, and then we won't need any actual objects anymore, and everything will be Categories. So it's the *converse* of the smile of the Cheshire Cat: we've seen the smile often enough, so we know the Cat is *around,* and will certainly show up in the *end.* In the mean time we just get on with the job. What's the job? It's the making of The Cheshire Cat, of course! The formal requirements of a morphism between categories are minimal. It follows that, if what we construct *isn't* some Category, then it quite simply *does not exist.* This makes Category Theory a very good candidate for the language to describe that which we earlier called the class Γ_Γ of Gödel representable functions. What is the formal basis for such a translation? *Knowledge* of what the Category *actually is.* And the more we are able to translate the representations of that knowledge from one formalism to another, the better will our understanding of those formalisms become.

Such *qualified universal knowledge* is a prerequisite for actual knowledge:

> If he did not in an unqualified sense of the term know the existence of this triangle, how could he know without qualification that its angles were equal to two right angles? No: clearly he knows not without qualification but only in the sense that he knows universally. If this distinction is not drawn, we are faced with the dilemma in the Meno: either a man will learn nothing or what he already knows. Aristotle *Posterior Analytics* I.1.

And *actual* knowledge is a pre-requisite for learning logic:

> Now a man cannot believe in anything more than in the things he knows, unless he has either actual knowledge of it or something better than actual knowledge. But we are faced with this paradox if a student whose belief rests on demonstration has not prior knowledge; a man must believe in some, if not in all, of the basic truths more than in the conclusion. Moreover, if a man sets out to acquire the scientific knowledge that comes through demonstration, he must not only have a better knowledge of the basic truths and a firmer conviction of them than of the connexion which is being demonstrated: more than this, nothing must be more certain or better known to him than these basic truths in their character as contradicting the fundamental premisses which lead to the opposed and erroneous conclusion. For indeed the conviction of pure science must be unshakable. Aristotle *Posterior Analytics* I.2.

From whence are we supposed to get this knowledge? From our apprehension of the *wholes* in the world around us; those things which are in themselves integral, entire, completed constructions with which we find ourselves *presented* when we pay attention to our immediate surroundings. But we need to know how to describe those things we perceive before we can demonstrate knowledge of them. The sense in which we know them is not unqualified, but *qualified universal knowledge.* In the process of making the potential constructions actual, we come to know them in an unqualified sense.

> Obviously, therefore, the potentially existing constructions are discovered by being brought to actuality; the reason is that the geometer's thinking is an actuality; so that the potency proceeds from an actuality; and therefore it is by making constructions that people come to know them (though the single actuality is later in generation than the corresponding potency). Aristotle. *Metaphysics.* IX.9

And therefore at the conclusion of a Geometric *problem,* we say "That which *was to be constructed.*" In Greek:

ὅπερ ἔδει ποιῆσαι.

Proving Theorems in Formal Systems

Peirce's Law is $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$ for any propositional variables P and Q. It is a tautology in classical propositional logic, and it is not provable in intuitionistic propositional logic. One can see immediately why it is not provable in intuitionistic logic; it is because, since it only appears once in the formula, the propositional variable Q is arbitrary and in intuitionistic propositional logic the only axiom which eliminates negation is called the *law of contradiction* which says of any propositional formula A, that $\neg A \Rightarrow (A \Rightarrow B)$ for any propositional formula B whatsoever. Now the truth of $P \Rightarrow Q$ is determined by both P and Q and if it is *false,* then we can prove *either* $(P \Rightarrow Q) \Rightarrow P$ or $(P \Rightarrow Q) \Rightarrow \neg P$. One says that the truth or falsity of Q is *material,* so we cannot decide this proposition.

In classical propositional logic on the other hand, we can use equational reasoning in the denotation of the terms and we have the equivalence $A \Rightarrow B = \neg A \vee B$ as well as de Morgan's law $\neg(A \vee B) = \neg A \wedge \neg B$ and the double-negation law $\neg\neg A = A$, so we deduce that $(P \Rightarrow Q) \Rightarrow P = \neg(\neg P \vee Q) \vee P = (P \wedge \neg Q) \vee P$. Then whether Q is true or false is *immaterial* because if Q is false then we have $(P \wedge \mathsf{T}) \vee P = P \vee P = P$ and if Q is true then we have $(P \wedge \mathsf{F}) \vee P = \mathsf{F} \vee P = P$ so the truth of $(P \wedge \neg Q) \vee P$ is determined entirely by the truth of P and certainly $P \Rightarrow P$, so we know that whatever the values of P or Q, the formula $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$ is always true. But we did not need to know whether either were *actually* true or false to be able to deduce this. The double-negation elimination equivalence was crucial to the inference.

This is the defining characteristic of intuitionistic propositional logic: that the consequences of deductions are determined entirely from the axioms and rules of deduction of the proof system, but in the example we just gave of a deduction in classical logic the "path of truth" through the deductive system was *indeterminate* in so far as we never actually knew *which* of the two cases was *necessary.*

The relation of necessity or *logical consequence* appears in Kripke's models of intuitionistic logic. For propositional logic these are very simple. A set ϕ of propositional formulæ is modeled by a structure called a *Kripke frame,* which is a finite set K_ϕ of *nodes* $k, k', \ldots$ which each determine the truth values of a certain subset $k \subseteq \phi$

of formulæ, and a *partial* order $\leq_K$ which expresses the relation "determines" in the sense that for any $k, k' \in K$ then $k \leq_K k'$ if and only if k' determines the truth values of every propositional expression A which the node k determines. Note however that nodes do *not* determine the values of propositional variables P, Q, R, etc. These are essentially *indeterminate,* but the relation $\leq$ preserves whatever they actually may be. So for all nodes $k, k' \in K$ with $k \leq k'$, and all propositional variables P ocurring in ϕ, if k determines P then k' also determines P. Thus, once a propositional variable has been determined it is determined for ever after.

Writing $k\Delta_K A$ to denote "$k \in K$ determines A" the relation of logical consequence is extended over a set ϕ of propositional formulæ as follows. For all frames K_ϕ, all nodes $k \in K_\phi$ and all formulæ $A, B \in \phi$

1. $k\Delta_K A \wedge B$ if $k\Delta_K A$ and $k\Delta_K B$.
2. $k\Delta_K A \vee B$ if $k\Delta_K A$ or $k\Delta_K B$.
3. $k\Delta_K A \Rightarrow B$ if, for all $k' \geq k$, if $k'\Delta_K A$ then $k'\Delta_K B$.
4. $k\Delta_K \neg A$ if for no $k' \geq k$ does $k'\Delta_K A$.

It is not hard to see that this relation preserves consistency in the sense that no node k determines both A and $\neg A$ for any proposition $A \in \phi$, and also that it is monotonic so that $k'\Delta_K A$ if $k\Delta_K A$ for any $k \leq k'$.

One may profitably think of the entire set $\mathcal{U}$ of Kripke frames $K \in \mathcal{U}$ as providing a model of causality or *time,* with each frame K an *observer space* which is a partial order on events with those events at time $k' \in K_\phi$ all logical consequences of events at earlier times $k \leq k' \in K_\phi$. Time is a partial order because *in principle,* one does not necessarily have sufficient information to determine the relative order of any two given events. For example, there is a necessarily finite maximum rate of propagation of the relation of logical consequence, whence the joke "Time is what stops everything from happening all at once." One cannot know what this rate is however, because it is that very measure by which we judge time itself, so we necessarily have $\mathrm{d}t/\mathrm{d}t = 1$. All judgements of the relative order of events are *logical* deductions; all time measurements being ultimately physical measurements of some kind based on some physical theory of causality. Thus it would be logically impossible, or literally *absurd,* to measure the rate of the passage of time, because those physical theories by which the measurement is judged are necessarily based on an assumption of the constant finite rate of propagation of logical inference.

Thus each finite frame K_ϕ represents some coherent body of knowledge about physical facts, and the partial order on the nodes $k \in K$ represents the temporal order of events in that frame K. Because the rate of propagation of effects from their causes is finite, there are so-called *space-like-separated* frames, $K_1, K_2 \in \mathcal{U}$ say, in which events have no temporal relation so that events determined at times $k_1 \in K_1$ and $k_2 \in K_2$ have no relation of logical consequence. We say they are *coincidental* or *chance*

events because one cannot say that either one determines the other, even though at some later stage of physical evolution in both frames there may be shared judgements $P \in k'_1 \cap k'_2 \neq \varnothing$ in nodes $k'_1 \in K_1$ and $k'_2 \in K_2$ such that all events in $K_1 \cup K_2$ may be placed in temporal relation with one another establishing a total order.

Thus far everything we have said about the universe $\mathscr{U}$ of Kripke frames is independent of any particular formal system of deduction. Note that for any proposition A, the leaf nodes of any finite frame K each either determine A or $\neg A$, simply because by definition the leaf nodes k in any frame K are such that there exist no nodes $k' \in K$ where $k' > k$, and therefore by condition (4), if k does not determine A then k determines $\neg A$ by default or *vacuously*. Thus the leaf nodes of any finite Kripke frame K_ϕ are all classical models of the theory ϕ.

Kripke showed that a proposition A is intuitionistically provable if and only if it is determined by the $\leq_K$-least element k_0 of *every* finite frame K. That is to say that the only formulæ provable in intuitionistic propositional logic are those tautologies that are *absolute,* in the sense that they are completely mechanically determined by the axioms and rules of deduction of the theory. Any propositional formula A contains finitely many propositional variables P, Q, etc., so it is possible to enumerate a finite set Γ of Kripke frames $K_i \in \Gamma$, the nodes of which determine every propositional variable in every possible order. Such a set constitutes a minimum which would necessarily contain a counter-example K with a $\leq_K$-least element $k_0 \in K$ which does not determine A, if one exists.

This gives a completely mechanical method for determining intuitionistic propositional truth. Given an arbitrary proposition A, one simply tests the finite set of Kripke frames for counter-examples and if there are none then one is assured that the recursively enumerable set of formal proofs will include a proof of A. But this is a much more complex affair than determining the tautologies in classical propositional logic where, for a formula with n propositional variables one simply enumerates the set of all 2^n possible combinations of values of those variables. If the proposition holds in all cases then it is a tautology.

In a *Hilbert-style* deduction system a *proof* is an ordered sequence of formulæ, each of which is one of a set of *axioms,* or is derived by a *rule of inference* from two other formulæ ocurring earlier in the same list. The two antecedent formulæ are called the *premisses* of the inference and the consequent is called the *conclusion.* The proof is said to *prove* the final formula in the list. Thus a proof may be *converted* into tree structure by deriving anew every formula to which a reference is made in any of the premisses of any inference. Such a proof is said to have been *normalised,* and the resulting proof is the *normal form* of any of the proofs in the series of conversions leading from the initial sequence to the normal form.

In the Hilbert-style deduction system for propositional logic there is only one rule of inference called *cut elimination* which is a *conversion* or *proof normalising* rule in *modus ponendo ponens:* for any propositional formulæ A and B, given a *cut* proof of B in the form of a *hypothetical* proof $A \Rightarrow B$ and a proof of A one may produce

a direct proof of B by eliminating the cut. This is normalising in the sense that it *converts* a shorter abstract proof to a longer immediate proof by substitution for the placeholder A in the abstract proof.

$$\text{MP:}\ \frac{A \Rightarrow B \qquad A}{B} \tag{12}$$

The theory is characterised by a finite set of *axiom schema* which are abstract forms denoting infinite sets of formulæ by metalanguage variables A, B and C ranging over arbitrary well-formed formulæ. The following is a complete and rather beautiful set of ten axiom schema for intuitionistic propositional logic, due to Kleene. They consist of laws for introduction and elimination of logical constants. First a pair of dual schema for implication introduction, and negation introduction and elimination:

$$\begin{array}{rc}
\Rightarrow\text{-I1:} & A \Rightarrow (B \Rightarrow A) \\
\Rightarrow\text{-I2:} & (A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C)) \\
\neg\text{-I:} & (A \Rightarrow B) \Rightarrow ((A \Rightarrow \neg B) \Rightarrow \neg A) \\
\neg\text{-E:} & \neg A \Rightarrow (A \Rightarrow B).
\end{array} \tag{13}$$

The first implication introduction is also called *weakening,* because it introduces an extra hypothesis B which is not needed to prove A. Thus the original theorem implies the weakened theorem. The second is called *cut,* because from a *lemma* proving B depending upon a hypothesis A and a proof of C also depending on the same hypothesis A but in addition on B, one can produce a proof of C hypothetical on A by incorporating the lemma and eliminating the abstraction. Then, provided the hypothesis A is provable, one will be able to construct an unconditional proof of C. These first two schema correspond to *abstraction* combinators $\mathsf{S} \equiv \lambda P.\lambda Q.\lambda R.P\,R\,(Q\,R)$ and $\mathsf{K} \equiv \lambda P.\lambda Q.P$ of Schönfinkel's IKS combinatory logic. We will show by way of an example of the system that $\mathsf{I} \equiv \lambda P.P$, corresponding to the *assumption* axiom $A \Rightarrow A$, is derivable from these two axioms and the deduction in *modus ponens.* The second two schema, those for negation, are also called respectively the laws of *non-contradiction* and *contradiction.*

Then there are another pair of dual axiom schema for conjunction and disjunction:

$$\begin{array}{rcrc}
\wedge\text{-E1:} & A \wedge B \Rightarrow A & \vee\text{-I1:} & A \Rightarrow A \vee B \\
\wedge\text{-E2:} & A \wedge B \Rightarrow B & \vee\text{-I2:} & B \Rightarrow A \vee B \\
\wedge\text{-I:} & A \Rightarrow (B \Rightarrow A \wedge B) & \vee\text{-E:} & (A \Rightarrow C) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \vee B \Rightarrow C)).
\end{array} \tag{14}$$

The three axioms on the left-hand side, those for conjunction, correspond to the type deconstructors π_1 and π_2 which are *projections,* $\lambda\langle A \cdot * \rangle.A$ and $\lambda\langle * \cdot B \rangle.B$, with the corresponding constructor $\lambda A.\lambda B.\langle A \cdot B \rangle$, called a *concatenation,* written $a\,b$, for the elements $\langle A \cdot B \rangle$, called *ordered pairs,* of the *Cartesian* or *dependent product* type $A \times B$ which in order theory is called the *infimum.*

The three axioms on the right-hand side, those for disjunction, correspond to the constructors ι_1 and ι_2 which are the left and right *injections,* $\lambda A.\langle A|$ and $\lambda B.|B\rangle$, with

the corresponding deconstructor $\lambda f.\lambda g.\lambda\langle A.f\ A \mid B.g\ B\rangle$ called the *alternation*, written $A+B$ of the elements $\langle a \mid b\rangle$, called *unordered pairs*, of the *sumset* or *independent product* type $A \uplus B$, which in order theory is called the *supremum*.

The notation we have used in the deconstructors is that of *pattern matching* in the arguments to the lambda abstractions. In the case of concatenation, the angle-brackets in the arguments match *right-* and *left-closure* which are the *least pre-fixed points* of *type isomorphisms* $\forall\alpha.\forall\beta.\alpha \simeq \langle\alpha,\beta\rangle \simeq \beta$ and these types compose with the sum types to give arbitrary dependent products and alternations. Indeed this structure can directly implement the sum types, and much more besides.

We will now carry out our earlier threat of proving the theorem schema $A \Rightarrow A$ by specifying an effective method to calculate the identity function $\lambda x.x$. That is to say, we will be writing a *program* to compute this function, and this will correspond to a method which, given any propositional formula A, *effectively presents* a proof of $A \Rightarrow A$. This may sound trivial, but bear in mind that the proof language does not have any variable binders, so it is not immediately obvious that we could achieve this. We will need to co-opt the formal proof system into acting as a calculus of operations, and thereby *interpret* it as a mechanism for effecting a calculation by the rule of cut elimination.

The language for writing proof-programs will be the primitive type combinators corresponding to the axioms as we have described them above. The method is that of *combinatory abstraction* as explained in [13]. We define a language of *abstract proof combinators* with bound variables ranging over abstract proof schemas. By way of a concrete example, here is a metalanguage proof-schema $\Lambda A.\Lambda B.A \wedge B \Rightarrow A \vee B$

$$\dfrac{\dfrac{\dfrac{}{(A \Rightarrow A \vee B) \Rightarrow (A \wedge B \Rightarrow (A \Rightarrow A \vee B))}\ \Rightarrow\text{-I1: with } A \leftarrow A \Rightarrow A \vee B \text{ and } B \leftarrow A \wedge B \qquad \dfrac{}{A \Rightarrow A \vee B}\ \vee\text{-I1:}}{A \wedge B \Rightarrow (A \Rightarrow A \vee B)} \tag{15}$$

$$\dfrac{\dfrac{}{(A \wedge B \Rightarrow (A \Rightarrow A \vee B)) \Rightarrow ((A \wedge B \Rightarrow A) \Rightarrow (A \wedge B \Rightarrow A \vee B))}\ \Rightarrow\text{-I2: with } B \leftarrow A \wedge B,\ B \leftarrow A \text{ and } C \leftarrow A \vee B \qquad \dfrac{}{A \wedge B \Rightarrow (A \Rightarrow A \vee B)}\ \text{From (15)}}{(A \wedge B \Rightarrow A) \Rightarrow (A \wedge B \Rightarrow A \vee B)} \tag{16}$$

$$\dfrac{\dfrac{}{(A \wedge B \Rightarrow A) \Rightarrow (A \wedge B \Rightarrow A \vee B)}\ \text{From (16)} \qquad \dfrac{}{A \wedge B \Rightarrow A}\ \wedge\text{-E1:}}{A \wedge B \Rightarrow A \vee B} \tag{17}$$

Instantiating A and B in this with any propositional formulæ yields a proof of the conclusion $A \wedge B \Rightarrow A \vee B$.

Although it is a proof-schema denoting an infinite class of actual propositional proofs, this structure is itself one instance of a more general structure which we can see by abstracting the particular instances of axiom-schema ∧-E1 and ∨-I1 and representing them by variables x and y denoting, not actual propositional formulæ, such as those the letters A and B represent, but *generic* proof-schema, which may also use binding

variables x, y etc., ranging over such abstract proof schema.

$$\text{MP:}\ \frac{\dfrac{}{(m \Rightarrow y) \Rightarrow (x \Rightarrow (m \Rightarrow y))}\ \Rightarrow\text{-I1: with } A \leftarrow m \Rightarrow y \text{ and } B \leftarrow x \qquad \dfrac{\vdots}{m \Rightarrow y}}{x \Rightarrow (m \Rightarrow y)} \tag{18}$$

$$\text{MP:}\ \frac{\dfrac{}{(x \Rightarrow (m \Rightarrow y)) \Rightarrow (x \Rightarrow m) \Rightarrow (x \Rightarrow y)}\ \Rightarrow\text{-I2: with } A \leftarrow x, B \leftarrow m \text{ and } C \leftarrow y \qquad \dfrac{\text{From (18)}}{x \Rightarrow (m \Rightarrow y)}}{(x \Rightarrow m) \Rightarrow (x \Rightarrow y)} \tag{19}$$

$$\text{MP:}\ \frac{\dfrac{\text{From (19)}}{(x \Rightarrow m) \Rightarrow (x \Rightarrow y)} \qquad \dfrac{\vdots}{x \Rightarrow m}}{x \Rightarrow y} \tag{20}$$

On the view of axioms as type operators, this is an operational calculus of *abstract type combinators,* so that given type operators $p : x \mapsto m$ and $p : m \mapsto y$ we can compose these as functions to produce a type operator $r : x \mapsto y$, for any types of object x and y whatsoever. Abstracting the particular type operators p and q as *functions* of types $p : \beta \to \gamma$ and $q : \alpha \to \beta$ respectively, and using the combinators K and S to instantiate the axioms $\Rightarrow$-I1 and $\Rightarrow$-I2 respectively, we can write the inferences of the premisses as $\mathsf{K}\,p$ and $\mathsf{S}(\mathsf{K}\,p)q$ and then the conclusion follows from the single polymorphic type inference

$$\forall\alpha.\forall\beta.\forall\gamma.p : \beta \to \gamma, q : \alpha \to \beta \vdash \mathsf{S}(\mathsf{K}\,p)q : \alpha \to \gamma \tag{21}$$

In general, any well-typed combinator expression involving only K and S corresponds to an an abstract proof-functional. We can write these two combinators in Standard ML as

```
fun K p q = p
fun S p q r = p r (q r)
```

Then any well-typed function composed of these and possibly other abstracted arguments will be found to correspond to an abstract proof-functional. For example the Standard ML definition `fun B p q r = S (K p) q r` corresponds to the abstract proof-functional in (21) for which Standard ML infers the type $\forall\alpha.\forall\beta.\forall\gamma.(\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$, which, interpreted as a theorem schema states that for any propositions A, B and C, given any abstract proof-schemæ of propositions $B \Rightarrow C$ and $A \Rightarrow B$, the function `B` computes an abstract proof-schema of the proposition $A \Rightarrow C$.

This, however, is only the *metalanguage* proof-functional we have defined, and we cannot extract the proof-schema from this because we have only a denotational correspondence, not an *effective method* to *present* the actual proof schema. All we know is that they *could* in principle be produced. In other words, we know that $(B \Rightarrow C) \Rightarrow ((A \Rightarrow B) \Rightarrow A \Rightarrow C)$ is intuitionistically *provable,* but we don't know *how* to prove it. On the operational side of the correspondence, we know that there

is a computable function of the type $\forall\alpha.\forall\beta.\forall\gamma.(\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$, but we don't know how to effect the calculation. That is, we know there is a function-in-extension which corresponds to *some* intensional definition of the denotational semantics of that function, but we don't know *which* function that is. The reason is that we don't have any formal *operational mechanism* for abstraction in the object-language of propositional proofs. The problem is that in order to prove the abstract theorem we need to express the dependency of the types in the operators p and q. This is essential, because without it the functions cannot be composed. This problem doesn't appear in the proof-schema of (15–17) because there is an implicit binding in the sense that every instance of A or B is taken to denote the same formula. This is workable, provided only particular proof-schema are under consideration. Note that in (21) the assumptions which determine the type of the operators p and q are *inside* the universal quantifiers. This means that the instances of the type variable β refer to *the same* type. So the problem we have in proving the abstract combinator is that it must somehow refer to parts of the conclusions of whatever arbitrary proofs p and q it is given. This is not a problem when instantiating axioms from a finite set of schema, because we simply take them as being self-evidently true, so we instantiate them whichever way we like. But abstract schemæ such as $B \Rightarrow G$ and $A \Rightarrow B$ cannot be instantiated any way one pleases because they are not tautologies. So we must deal here with a representation of the actual *intensional* proof, not just the extension of the set of formulæ it may represent.

Thus the Hilbert-style proof system is purely extensional and presents a static picture of the theory. This is something that one notices immediately when one tries to produce a proof; there are no apparent operational 'rules of procedure.' The difficulty almost everyone has dealing with a proof system like this is a strong indicator that there is actually something inherently dynamic and *intensional* in most people's intuitive grasp of what a proof is. Inference is not naturally expressed as a static form, but rather as some sort of deterministic process: that knowlege of the truth or falsity of the premisses actually *causes* the truth or falsity of conclusion to become known. So a *natural* proof is one which makes explicit the causes of the knowledge of the truth of the premisses and then the knowledge of the truth of the conclusion follows inevitably. There is a corresponding notion of a natural proof system as a 'calculus of logical consequence,' such as Gentzen's.

To produce an operational calculus we will lift the metalanguage of abstract-proofs to an object language of abstract proof-functionals by defining *combinatory abstraction* which introduces object language variables and binders that can be used to describe abstract proof-functions. We will then have a new data type of *derivations* with associated rules of *deduction* which are *combinators* operating on proof-functions. Then the relation $\vdash$ will relate proofs and theorem-schema according to which theorem-schema can be derived from the axioms, or indeed any set of propositional formulæ whatsoever. In lifting the rules to derivations, we will have turned the axioms into a *deduction system* where each axiom corresponds to a *type operator* and a set of higher-order rules construct derivations from the three primitive combinators I, K and S. This calculus of operations will have a semantic interpretation in the original Hilbert-

style deduction system we started with: that is to say each combinator will be modeled by the schema of abstract proof-functionals representing generic proof schemæ which correspond to whole sets of propositional proof-schemæ. Each such generic proof schema models a generic instance of the fundamental polymorphic type constructor $\forall\alpha.\forall\beta.(\alpha,\beta)\rightarrow$ of functions. The denotation of a program will then be found to be a *type,* which is an effectively enumerable set of intuitionistic proof schemæ, and each value that a program calculates will correspond to an intuitionistic proof of some propositional formula. The process of generic instantiation of proof-combinators will correspond with running programs, and that of recovering the intuitionistic proof-schema from the conversion or *trace* of the combinator will be performed by the Hindley-Milner type inference algorithm.

Therefore we define the I, K and S combinators as primitive combinators acted on by the rule of inference MP: converting proofs to normal form, which is a cut-free proof where all abstractions made by the weakening rule ⇒-I1: and the cut rule ⇒-I2: have been converted by the rule MP:. We use lambda $\lambda x.M$ to denote abstraction of variables $x\in\{a,b,c,\ldots,a',b',c',\ldots,a'',b'',c'',\ldots\}$ in the term M and we juxtapose the premisses $A\Rightarrow B$ and A of inferences by MP: in that order, calling this the *combination* of the *argument* and the *function.* Then we define the relation of *weak reduction* on the combinators, so called because it reduces the cut proofs introduced by the rules ⇒-I1: and ⇒-I2: which are both a form of weakening in the sense that they each add extra hypotheses.

$$\mathsf{K}\,P\,Q\underset{w}{\rightarrow}P$$
$$\mathsf{S}\,P\,Q\,R\underset{w}{\rightarrow}P\,R\,(Q\,R)$$

Now we can define the denotation of combinatory abstraction.

$$[\![\lambda x.x]\!]=\mathsf{S}\,\mathsf{K}\,\mathsf{K}$$
$$[\![\lambda x.P]\!]=\mathsf{K}\,P$$
$$[\![\lambda x.P\,Q]\!]=\mathsf{S}\,[\![\lambda x.P]\!]\,[\![\lambda x.Q]\!]$$

Using this we can translate the proof-functional corresponding to the assumption or *identity* combinator $\mathsf{I}\equiv[\![\lambda x.x]\!]$. We have immediately $[\![\lambda x.\,x]\!]=\mathsf{S}\,\mathsf{K}\,\mathsf{K}$ and so $\mathsf{I}\equiv\mathsf{S}\,\mathsf{K}\,\mathsf{K}$. Combining I with any abstract type constructor A and reducing to weak normal form gives the following sequence of conversions:

$$\begin{aligned}\mathsf{I}\,A&\equiv\mathsf{S}\,\mathsf{K}\,\mathsf{K}\,A\\&\underset{w}{\rightarrow}\mathsf{K}\,A\,(\mathsf{K}\,A)\\&\underset{w}{\rightarrow}A\end{aligned}$$

What we are interested in is the proof which I itself represents, which is a proof of the theorem $A\Rightarrow A$. This is represented as the proof combinator I itself, which is a combination of the primitive combinators S and K. Each appearance of one or other of these combinators in a combination represents one instance of an axiom-schema ⇒-I1: or ⇒-I2:. The particulars of the instantiation are given by the *combinator types* in the expression $((\mathsf{S}\,\mathsf{K})\,\mathsf{K})$. These types are derived from the generic types of the combinators using the type inference rule COMB described in [11]. To compute proofs

from combinator expressions we will need only the four rules TAUT, INST, GEN and COMB from that paper. The first two instantiate axioms, the third universally quantifies free type variables and the fourth infers the type of a combination of two combinators. The use of CUT: should be understood as a purely typographical convention with no semantic significance at all: it is here only because the proof tree would not fit on a page without using a miniscule typeface.

The type derivation is

$$\text{TAUT:}\ \frac{}{\text{INST:}\ \dfrac{}{\Gamma \vdash \mathsf{S}\colon \forall\alpha.\forall\beta.\forall\gamma.(\alpha\to(\beta\to\gamma))\to((\alpha\to\beta)\to(\alpha\to\gamma))}} \tag{22}$$

$$\text{COMB:}\ \frac{\text{Concl. (22)} \qquad \text{TAUT:}\ \dfrac{}{\text{INST:}\ \dfrac{}{\Gamma \vdash \mathsf{K}\colon \forall\alpha.\forall\beta.\alpha\to(\beta\to\alpha)}}}{\Gamma \vdash \mathsf{SK}\colon(\alpha\to(\alpha\to\alpha))\to(\alpha\to\alpha)} \tag{23}$$

$$\text{CUT:}\ \frac{\text{Concl. (23)} \qquad \text{TAUT:}\ \dfrac{}{\text{INST:}\ \dfrac{}{\Gamma \vdash \mathsf{K}\colon \forall\alpha.\forall\beta.\alpha\to(\beta\to\alpha)}}}{\text{COMB:}\ \dfrac{}{\text{GEN:}\ \dfrac{}{\Gamma \vdash \mathsf{SKK}\colon \forall\alpha.\alpha\to\alpha}}} \tag{24}$$

Where

$$\begin{aligned}\Gamma = \{&\mathsf{K}\colon \forall\alpha.\forall\beta.\alpha\to\beta\to\alpha,\\ &\mathsf{S}\colon \forall\alpha.\forall\beta.\forall\gamma.(\beta\to\gamma\to\alpha)\to(\beta\to\gamma)\to\beta\to\alpha\}.\end{aligned}$$

Replacing $\to$ with $\Rightarrow$, α with A, dropping the parts between $\vdash$ and the :, replacing COMB: with MP:, introducing a combined axiom instantiation rule ITAUT: and dropping the use of GEN: and the universal quantifiers and replacing it with the metalanguage binder Λ we have an abstract proof combinator $\mathsf{I} \equiv \Lambda A.A \Rightarrow A$ which, given any propositional formula A effectively presents proof in intuitionistic propositional logic of $A \Rightarrow A$. That is to say $\mathsf{I} \equiv \Lambda A.P$ where P is the following schema

$$\text{ITAUT:}\ \frac{\Rightarrow\text{-I2:}}{(A\Rightarrow((A\Rightarrow A)\Rightarrow A))\Rightarrow((A\Rightarrow(A\Rightarrow A))\Rightarrow(A\Rightarrow A))} \tag{25}$$

$$\text{MP:}\ \frac{\text{Concl. (25)} \qquad \text{ITAUT:}\ \dfrac{\Rightarrow\text{-I1:}}{A\Rightarrow((A\Rightarrow A)\Rightarrow A)}}{(A\Rightarrow(A\Rightarrow A))\Rightarrow(A\Rightarrow A)} \tag{26}$$

$$\text{CUT:}\ \frac{\text{Concl. (26)} \qquad \text{ITAUT:}\ \dfrac{\wedge\text{-I:}}{A\Rightarrow(A\Rightarrow A)}}{\text{MP:}\ \dfrac{(A\Rightarrow(A\Rightarrow A))\Rightarrow(A\Rightarrow A)}{A\Rightarrow A}} \tag{27}$$

The following is a derivation of the type of I produced by the Hindley-Milner type inference algorithm. The difference between this and the manually produced one

above is that here a more general instantiation has been made of the type of S. The algorithm always chooses the *most* general one possible. Apart from the final inference rule, the fragment of the algorithm exercised here is actually only the combinatory part due to Hindley. The generality of the instantiation is due to the use of Robinson's unification algorithm. See [11] for references. The other difference is that we have combined the TAUT and INST inferences into one called ITAUT. The type assumptions in effect in the following proof are

$$\Gamma \equiv \Gamma_\alpha \equiv \{\mathsf{K}\colon \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha,\\ \mathsf{S}\colon \forall\alpha.\forall\beta.\forall\gamma.(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma\}.$$

$\Gamma \vdash \mathsf{S}\colon(\alpha \to (\beta \to \alpha) \to \alpha) \to (\alpha \to \beta \to \alpha) \to \alpha \to \alpha$	1	ITAUT: $\mathsf{S}\colon[\alpha/\alpha, \beta \to \alpha/\beta, \alpha/\gamma]$
$\Gamma \vdash \mathsf{K}\colon\alpha \to (\beta \to \alpha) \to \alpha$	2	ITAUT: $\mathsf{K}\colon[\alpha/\alpha, \beta \to \alpha/\beta]$
$\Gamma \vdash \mathsf{S}\,\mathsf{K}\colon(\alpha \to \beta \to \alpha) \to \alpha \to \alpha$	3	COMB: 1,2
$\Gamma \vdash \mathsf{K}\colon\alpha \to \beta \to \alpha$	4	ITAUT: $\mathsf{K}\colon[\alpha/\alpha, \beta/\beta]$
$\Gamma \vdash \mathsf{S}\,\mathsf{K}\,\mathsf{K}\colon\alpha \to \alpha$	5	COMB: 3,4

This proof was produced by modifying the Standard ML implementation of the algorithm given in [9][4] to produce a derivation Δ along with the type-scheme σ and the substitution U. Back-substituting with U in Δ yields the proof.

The entire Hindley-Milner type inference algorithm, incuding Robinson's unification algorithm, is a purely imperative process in the sense that there is *no* equation solving of any kind whatsoever involved. Indeed most actual implementations of it are based on directly modifying the types by copying them and changing pointers so that substitutions are always done in-place. The result of this is that every inference is determined from the root of the abstract syntax representation of the expression being typed. This shows that the resulting inferences are all intuitionistic in the sense that they will each be found to correspond to some finite Kripke frame K with a root $k_0 \in K$ in which every expression that is determined in K is determined from the root node. Therefore every set ϕ of generic instantiations of the polymorphic types represented by intuitionistic tautologies corresponds to a schema of Kripke models K_ϕ of the corresonding set $[\![\phi]\!]$ of formulæ in intuitionistic predicate calculus.

Therefore the combinator expression of a proof-program combinator as a combination of primitive combinators completely determines an effective method for calculating that proof. All we need to do is to treat the combinator expression as a generic polymorphic type, then the Hindley-Milner algorithm will infer the intuitionistic proof-schema for *any* generic instantiation of that combinator. That is, we can *effectively present* the intuitionistic proof for any and every well-typed proof-program given as a combination of primitive combinators whose types are assumed. Those assumed types of the primitive combinators correspond to the axioms of an intuitionistic predicate logic. There are many such logics because intuitionistic predicate

[4] Note that there is an error on page 12 line −11 where it says of case (i) of the algorithm W that "… the result is simply the generic instantiation of the type-scheme to new variables." This is wrong: all inference is necessarily based on particular instantiations, i.e. *not* universally quantified.

logic admits infinitely many consistent extensions by introducing as additional axioms formulæ which, though not derivable from the basic axioms, are admissable because they are intuitionistically *decidable*. A formula A is intuitionistically decidable if $A \vee \neg A$ is intuitionistically provable, which is to say that A is determined by the root node of every finite Kripke frame modelling that theory.

For example, here is Hindley, Robinson and Milner's proof of the effectiveness of the procedure given by the proof-program specified by the combinator

$$\mathsf{B} \equiv \lambda p.\lambda q.\mathsf{S}\,(\mathsf{K}\,p)\,q. \tag{28}$$

$$\begin{array}{lll}
\Gamma_2 \vdash \mathsf{S}\colon(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma & 1 & \text{ITAUT: } \Sigma_1 \\
\Gamma_2 \vdash \mathsf{K}\colon(\beta \to \gamma) \to \alpha \to \beta \to \gamma & 2 & \text{ITAUT: } \Sigma_2 \\
\Gamma_2 \vdash p\colon\beta \to \gamma & 3 & \text{ASSUM: } 9 \\
\Gamma_2 \vdash \mathsf{K}\,p\colon\alpha \to \beta \to \gamma & 4 & \text{COMB: } 2,3 \\
\Gamma_2 \vdash \mathsf{S}\,(\mathsf{K}\,p)\colon(\alpha \to \beta) \to \alpha \to \gamma & 5 & \text{COMB: } 1,4 \\
\Gamma_2 \vdash q\colon\alpha \to \beta & 6 & \text{ASSUM: } 8 \\
\Gamma_2 \vdash \mathsf{S}\,(\mathsf{K}\,p)\,q\colon\alpha \to \gamma & 7 & \text{COMB: } 5,6 \\
\Gamma_1 \vdash \lambda q.\mathsf{S}\,(\mathsf{K}\,p)\,q\colon(\alpha \to \beta) \to \alpha \to \gamma & 8 & \text{ABS: } 7,6 \\
\Gamma \vdash \lambda p.\lambda q.\mathsf{S}\,(\mathsf{K}\,p)\,q\colon(\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma & 9 & \text{ABS: } 8,3 \\
\Gamma \vdash \mathsf{B}\colon\forall\alpha.(\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma & 10 & \text{GEN: } 9 \\
\Gamma \vdash \mathsf{B}\colon\forall\gamma.\forall\alpha.(\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma & 11 & \text{GEN: } 10 \\
\Gamma \vdash \mathsf{B}\colon\forall\beta.\forall\gamma.\forall\alpha.(\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma & 12 & \text{GEN: } 11
\end{array} \tag{29}$$

Where $\Gamma_1 \equiv \Gamma \cup \{p\colon\beta \to \gamma\}$, $\Gamma_2 \equiv \Gamma_1 \cup \{q\colon\alpha \to \beta\}$, $\Sigma_1 \equiv \mathsf{S}\colon[\alpha/\alpha, \beta/\beta, \gamma/\gamma]$ and $\Sigma_2 \equiv \mathsf{K}\colon[\beta \to \gamma/\alpha, \alpha/\beta]$.

This is a fully-abstracted procedure capable of composing any pair of proofs of propositions $p = B \Rightarrow C$ and $q = A \Rightarrow B$ into a composite proof of $A \Rightarrow C$. Then the root of any finite Kripke frame will determine *all* possible determined instantiations of γ, α and β, *provided* p and q are *compatible* in the sense that the formula in each proof-schema denoted by the metalanguage variable B may be instantiated to a common form, *and also* that A and C may be instantiated compatibly with that instantiation of the middle term B. That is to say, that the coherence of the *whole generic instance* is necessary for an effective presentation of the proof.

The role of the function space type constructor λ is to introduce *type assumptions,* each of which is the most general instance of a given argument. The only rules that introduce assumptions are ABS: and LET: and therefore, provided a variable α does not occur free in Γ, every *parcel* of type assumptions made during the course of an inference is guaranteed to be discharged by the corresponding rule, and therefore the side-condition of the rule GEN: is guaranteed to be satisfied and so all such types inferred on the basis of assumptions of only generic instances of type-schemes are themselves generic. This allows our proof-program combinators to be parameterised

on arbitrary *hypothetical proofs* represented by the lambda-bound variables. Now we can use the combinator B to generate the proof (15) we started with, back on page 23. When we apply the primitive combinators ι_1 and π_1 which represent the left injection and projection, respectively, we find

$$\begin{aligned} \mathsf{B}\,\iota_1\,\pi_1 &\equiv (\lambda p.\lambda q.\mathsf{S}\,(\mathsf{K}\,p)\,q)\,\iota_1\,\pi_1 \\ &\to (\lambda q.\mathsf{S}\,(\mathsf{K}\,\iota_1)\,q)\,\pi_1 \\ &\to \mathsf{S}\,(\mathsf{K}\,\iota_1)\,\pi_1 \end{aligned} \tag{30}$$

and then nothing happens until S is fully applied because the corresponding type operator $(\alpha,\beta,\gamma)\,\mathsf{op}_{\mathsf{B}}$ is a method of proof conversion, not a proof, and furthermore it is not determined by the types of its two arguments p and q. Therefore we have *two* distinct generic type-shemes. Let Γ now include additional assumptions for the combinator B and for projections and injections:

$$\Gamma \equiv \Gamma_\beta \equiv \Gamma_\alpha \cup \left\{ \begin{array}{c} \mathsf{B}\colon \forall\beta.\forall\gamma.\forall\alpha.(\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma, \\ \pi_1\colon \forall\alpha.\forall\beta.\langle\alpha\cdot\beta\rangle\to\alpha, \pi_2\colon \forall\alpha.\forall\beta.\langle\alpha\cdot\beta\rangle\to\beta, \\ \iota_1\colon \forall\alpha.\forall\beta.\alpha\to\langle\alpha\mid\beta\rangle, \iota_2\colon \forall\alpha.\forall\beta.\beta\to\langle\alpha\mid\beta\rangle \end{array} \right\}.$$

Then we have the *left-hand* type-scheme:

$\Gamma\vdash \mathsf{B}\colon(\alpha\to\langle\alpha\mid\gamma\rangle)\to(\langle\alpha\cdot\beta\rangle\to\alpha)\to\langle\alpha\cdot\beta\rangle\to\langle\alpha\mid\gamma\rangle$	1	ITAUT: $\mathsf{B}\colon[\langle\alpha\cdot\beta\rangle/\alpha,\alpha/\beta,\langle\alpha\mid\gamma\rangle/\gamma]$
$\Gamma\vdash \iota_1\colon\alpha\to\langle\alpha\mid\gamma\rangle$	2	ITAUT: $\iota_1\colon[\alpha/\alpha,\gamma/\beta]$
$\Gamma\vdash \mathsf{B}\,\iota_1\colon(\langle\alpha\cdot\beta\rangle\to\alpha)\to\langle\alpha\cdot\beta\rangle\to\langle\alpha\mid\gamma\rangle$	3	COMB: 1,2
$\Gamma\vdash \pi_1\colon\langle\alpha\cdot\beta\rangle\to\alpha$	4	ITAUT: $\pi_1\colon[\alpha/\alpha,\beta/\beta]$
$\Gamma\vdash \mathsf{B}\,\iota_1\,\pi_1\colon\langle\alpha\cdot\beta\rangle\to\langle\alpha\mid\gamma\rangle$	5	COMB: 3,4
$\Gamma\vdash \mathsf{B}\,\iota_1\,\pi_1\colon\forall\gamma.\langle\alpha\cdot\beta\rangle\to\langle\alpha\mid\gamma\rangle$	6	GEN: 5
$\Gamma\vdash \mathsf{B}\,\iota_1\,\pi_1\colon\forall\beta.\forall\gamma.\langle\alpha\cdot\beta\rangle\to\langle\alpha\mid\gamma\rangle$	7	GEN: 6
$\Gamma\vdash \mathsf{B}\,\iota_1\,\pi_1\colon\forall\alpha.\forall\beta.\forall\gamma.\langle\alpha\cdot\beta\rangle\to\langle\alpha\mid\gamma\rangle$	8	GEN: 7.

And the *right-hand* type-scheme:

$\Gamma\vdash \mathsf{B}\colon(\beta\to\langle\gamma\mid\beta\rangle)\to(\langle\alpha\cdot\beta\rangle\to\beta)\to\langle\alpha\cdot\beta\rangle\to\langle\gamma\mid\beta\rangle$	1	ITAUT: $\mathsf{B}\colon[\langle\alpha\cdot\beta\rangle/\alpha,\beta/\beta,\langle\gamma\mid\beta\rangle/\gamma]$
$\Gamma\vdash \iota_2\colon\beta\to\langle\gamma\mid\beta\rangle$	2	ITAUT: $\iota_2\colon[\gamma/\alpha,\beta/\beta]$
$\Gamma\vdash \mathsf{B}\,\iota_2\colon(\langle\alpha\cdot\beta\rangle\to\beta)\to\langle\alpha\cdot\beta\rangle\to\langle\gamma\mid\beta\rangle$	3	COMB: 1,2
$\Gamma\vdash \pi_2\colon\langle\alpha\cdot\beta\rangle\to\beta$	4	ITAUT: $\pi_2\colon[\alpha/\alpha,\beta/\beta]$
$\Gamma\vdash \mathsf{B}\,\iota_2\,\pi_2\colon\langle\alpha\cdot\beta\rangle\to\langle\gamma\mid\beta\rangle$	5	COMB: 3,4
$\Gamma\vdash \mathsf{B}\,\iota_2\,\pi_2\colon\forall\gamma.\langle\alpha\cdot\beta\rangle\to\langle\gamma\mid\beta\rangle$	6	GEN: 5
$\Gamma\vdash \mathsf{B}\,\iota_2\,\pi_2\colon\forall\beta.\forall\gamma.\langle\alpha\cdot\beta\rangle\to\langle\gamma\mid\beta\rangle$	7	GEN: 6
$\Gamma\vdash \mathsf{B}\,\iota_2\,\pi_2\colon\forall\alpha.\forall\beta.\forall\gamma.\langle\alpha\cdot\beta\rangle\to\langle\gamma\mid\beta\rangle$	8	GEN: 7.

Let us call these two distinct type operators $\mathsf{B}_{11}\equiv\forall\alpha.\forall\beta.\forall\gamma.\langle\alpha\cdot\beta\rangle\to\langle\alpha\mid\gamma\rangle$ and $\mathsf{B}_{22}\equiv\forall\alpha.\forall\beta.\forall\gamma.\langle\alpha\cdot\beta\rangle\to\langle\gamma\mid\beta\rangle$. Clearly each of these type-schemes corresponds to a distinct operational method for effectively presenting a proof; for they each use different type constructors: ι_1,π_1 in one case and ι_2,π_2 in the other. But they are also distinct *as type-schemes* because there is no type substitution which can take an arbitrary generic instance of the left-hand type-scheme to one of the right-hand type-scheme. An easy way to see this is to write the quantifier bindings as one of three distinct types of bracket, with left-hand brackets corresponding to binders α,β,γ in

$\forall\alpha.\forall\beta.\forall\gamma.\sigma$ and right-hand brackets corresponding to instances of the three different type variables ocurring free in the subexpression σ; and with left-hand brackets repeated for each bound instance of that variable in σ so that there are always the same number of left and right-hand brackets of each type. For example we write $[[\langle(]\rangle]\rangle$ and $[\langle\langle(]\rangle\rangle)\rangle$ for the left and right-hand types B_{11} and B_{22} respectively. This structure is abstract of the actual variable names so it is clear that no capture-avoiding substitution is going to change it. Also, although the order of the binders in a closed expression is arbitrary, because there are no free ocurrences of any variables, any permutations of the binder order will only change the order of the (groups of) left-hand brackets, not the right-hand ones.

This device is formalised in the elementary theory of determinants published by Charles Lutwidge Dodgson [5] in 1867. To characterise the *topology* of an expression with n ocurrences of bound variables, one may associate to each binder a distinct series of one or more consecutive integers from $1,2,3,\ldots,n$, beginning with the leftmost binder and numbering consecutive binder references consecutively. This induces an order on the sequence of bound variables. Reading the body of the expression from left to right, one obtains a sequence $i_1,i_2,i_3,\ldots,i_n$ of binder-reference indices. Thus for example the two types above are characterised by the sequences $(1,2),(2,3),(3,1),(4,4)$ and $(1,1),(2,3),(3,4),(4,2)$. This was found by reading off the sequence of indices in $\mathsf{B}_{11} \equiv \forall\alpha_{1,2}.\forall\beta_3.\forall\gamma_4.\langle\alpha_2\cdot\beta_3\rangle \to \langle\alpha_1 \mid \gamma_4\rangle$ and $\mathsf{B}_{22} \equiv \forall\alpha_1.\forall\beta_{2,3}.\forall\gamma_4.\langle\alpha_1\cdot\beta_3\rangle \to \langle\gamma_4 \mid \beta_2\rangle$. Note that the quantifier references within a particular parcel are 'discharged' in reverse order.

Each expression with n ocurrences of bound variables is therefore characterised by a sequence of ordered pairs which corresponds to a *permutation* of the sequence $1,2,3,\ldots,n$. These may be written in permutation notation as $\left(\begin{smallmatrix}1234\\2314\end{smallmatrix}\right)$ and $\left(\begin{smallmatrix}1234\\1342\end{smallmatrix}\right)$ but are more clearly expressed by writing only the bottom row, assuming that the upper row is in the order $123\ldots n$. Such a permutation contains a number of *derangements* which are the elements of the permutation which disorder the resulting sequence. For example, B_{11} contains one derangement, being the consecutive digits 31 which are not in ascending order in 2314, and the permutation 1342 corresponding to B_{22} also has one derangement, being the pair 42. Dodgson then proves

> **Theorem III**. If there be a set of pairs of numerals, in which the antecedents are all different, as also are the consequents; and if they be arranged, firstly in order of antecedents, and secondly in order of consequents: the number of derangements among the consequents in the first case, and the number of derangements among the antecedents in the second case, are equal. □

Such permutations are said to be either even or odd, depending on the parity of the number of derangements they contain and formulæ with bound variables are thereby attributed to definite classes, each according to the parity of its characteristic permutation. Two numerals are said to be *similar* if they are of the same parity, i.e. if they are both odd or both even, otherwise they are *dissimilar*. Dodgson then proves

> **Theorem V**. If there be a set of n pairs of numerals, in which the antecedents are a certain permutation of the numbers from 1 to n, as also are the consequents; and if one pair be erased: the class, to which the remaining set belongs, is the same as that of the original set, or different, according as the numerals in the erased pair are similar or dissimilar. □

He adds

> **Corollary**. It should be observed that the class, to which a set of pairs of numerals belongs, is unaffected by the order in which they happen to be given. □

The theorem therefore gives a *co-variant of conversion* for *any* theory of binders whatsoever. Theorem III together with this corollary show that, in the context of the lambda calculus for example, the class of any expression is 'invariant under swaps' applied globally, i.e. it is alpha-invariant, and during the process of capture-avoiding substitution of some bound variable, the class of the resulting expression is determined entirely by the combined parity of the bound instances of that particular parcel[5] being replaced; and *it doesn't matter in which oder the parcels are substituted*. The proof is interesting because it employs a *co-inductive* rather than an inductive proof principle. Such a method may well have been that which Fermat referred to as the *method of infinite descent*. Dodgson uses this to prove a global property of the *whole set* of ordered pairs which is *indeterminately dependent* upon the properties of its constituent parts. The dependency is indeterminate in the sense that any actual proof of the class of a particular permutation may be constructed in reverse or *analysed into* a well-founded sequence of deconstruction steps, each step consisting in removing an arbitrary pair from the set and recording the changes of parity of the class that result, which changes depend entirely on the properties of the individual pairs. Finally the set will have been reduced to just two pairs whose class is uniquely determined, and this, together with the parity of the number of *parity changes* gives the class of the whole set. The process of constructing the proof, although non-deterministic, always uses *positive, definite* information about the set. Once completed such an analysis is identical to a well-founded construction which will be a series of definite steps starting with the basis of the two initial pairs produced by the analysis. The construction is *synthesised out of* the elements the whole was *analysed into*. Analysis thus corresponds to the deconstruction of the whole and concommitant construction of directed products, and the subsequent synthesis the construction of the whole from the deconstruction of those directed products which are essentially nothing more than the constructors of the recursive datatypes.

Theorem V forms the basis of Dodgson's *extremely* elegant treatment of determinants, where the fundamental objects are the *names* of elements of matrices consid-

[5] This notion comes from Girard *et al.* [7] who use it to refer to the hypotheses discharged by the rule of $\Rightarrow$-I in Gentzen's natural deduction system. The necessity to link hypotheses (leaves) directly to the deductions (branches) where they are discharged shows the tree structure of a natural deduction to be an over-simplification.

ered as pairs of row and column numbers. So it transpires that the otherwise mysterious properties of the determinants of matrices are in fact *explained* as being due to this co-variant of conversion as it applies to the Gaussian operations of transposition, addition and deletion of rows and columns. Any Gaussian matrix is a representation of a language of binders where row and column indices correspond to 'slots' which represent the values of variables. For example the calculation represented by the matrix 'equation'

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}$$

could be written as a lambda expression

$$\lambda\langle x \cdot y \rangle = \langle ax + by \cdot cx + dy \rangle$$

which shows that the role of the column vector is really nothing more than a set of variable binders assigning one variable to each 'slot' which is represented by a column of the matrix. In Dodgson's notation of matrices as sets of ordered pairs, one would write this *particular* equation as

$$\begin{pmatrix} \langle 1 \cdot 1 \rangle & \langle 1 \cdot 2 \rangle \\ \langle 2 \cdot 1 \rangle & \langle 2 \cdot 2 \rangle \end{pmatrix} \begin{pmatrix} \langle * \cdot 1 \rangle \\ \langle * \cdot 2 \rangle \end{pmatrix} = \begin{pmatrix} \langle 1 \cdot 1 \rangle\langle * \cdot 1 \rangle + \langle 1 \cdot 2 \rangle\langle * \cdot 2 \rangle \\ \langle 2 \cdot 1 \rangle\langle * \cdot 1 \rangle + \langle 2 \cdot 2 \rangle\langle * \cdot 2 \rangle \end{pmatrix}$$

The gain would appear to be less than slight! However the notation is *not* intended to represent particular concrete instances of matrices which Dodgson calls *blocks,* rather it is intended to represent *The Matrix* itself, which is an entirely abstract correspondence between the various parts of the whole system of equations. The form $\langle * \cdot n \rangle$ denotes projection of the n-th column and the dual form $\langle m \cdot * \rangle$ is the projection of the m-th row. Thus one might choose to express the general cases of the m right and left *outer-products* of a system of m independent equations in n unknowns as

$$\lambda\langle m \cdot * \rangle.\Sigma n.\langle m \cdot n \rangle \times \langle * \cdot n \rangle$$
$$\lambda\langle * \cdot m \rangle.\Sigma n.\langle n \cdot * \rangle \times \langle n \cdot m \rangle$$

where the metalanguage binder $\Sigma n.M$ denotes the summation of $M(i)$, say, over $i = 1, 2, \ldots, n$. Thus the notation is independent of the dimension of the blocks. Note that these two expressions have different types characterised by the permutations 132 and 213, but depending upon the types of the objects the elements $\langle i \cdot j \rangle$ represent one may be able to use commutativity of the multiplication operation to make them the same. So for real-valued matrices, for example, the left and right outer products are essentially the same and it is only a matter of convention which order one chooses to adopt for multiplication of blocks. It is ultimately exactly such an asymmetry of commutation of a type of inner product space with so-called *Hermitian* symmetry over complex scalars representing the spin-state of fermions which gives rise to the antisymmetries in the Schrödinger wave equation and the curious 'non-classical' behaviour of electrons.

Using lambda abstraction in the way that B does, the proof combinators are able to take, not *proof-schemæ* as arguments, but other proof combinators. For example,

the two combinators B_{11} and B_{22} may each produce a proof of $A \wedge B \Rightarrow A \vee B$, but the proofs are distinct because they take different paths through the problem: one proves it using the left-hand conjunct of $A \wedge B$ and the other through the right-hand conjunct. So we should interpret the intuitionistic meaning of, e.g. the axiom ∨-I1: of disjunction-introduction as denoting the *conversion* rule as a *method* which, given another *method* $[\![A]\!]$ of effectively presenting a proof of a proposition A, effectively presents a method $[\![A \vee B]\!]$ of effectively presenting a proof of the proposition which is the disjunction $A \vee B$ of A and *any* proposition B. Then, because proof combinators are *effective methods* they have intensional interpretations as *acts* of making constructions. Though the results of effecting the constructions (proposition schemæ) may be indistinguishable once completed, the proof combinators themselves may yet be distinguishable in the method they use to prove the propositions. So we can potentially prove more stuff using abstract proof combinators to represent proof construction *methods* rather than the resulting concrete constructions of proof-schemæ.

For example, given an instance of the theorem $A \wedge B \Rightarrow A \vee B$ we do not know which side of the conjunction was used to prove the consequent. It was surely one or the other, because there are only two possible deconstructors of conjunctions. However which one it is depends upon the particular instantiation; Therefore we would need to examine the proof schema to determine which case is used. But given an arbitrary *actual* proof schema it is very difficult to conceive how one might specify an effective method of analysing that proof schema to determine which of two disjuncts was actually proved. Such proof schemæ may be made arbitrarily weak in the sense of having any number of un-necessarily assumed premisses. The only way to effect such an analysis would be to systematically convert the proof to normal form, and in the course of this operation, to observe the fate of the two conjuncts as they go to meet their respective intuitionistically determined destinies, one which will be the conclusion and the other oblivion. But working instead with proof-programs, we can reason metatheoretically about the whole class of *potential* proofs of formulæ corresponding to any given schema such as $A \wedge B \Rightarrow A \vee B$. Then the Dodgson co-variant will provide a means to decide for any given combination of combinators whether they are consistent or not, and if so, whether they uniquely determine the formulæ they represent. In terms of intuitionistic logic, it would tell us whether the proposition is consistent and if so, whether or not it is decidable. In terms of the *models* of intuitionistic logic, it tells us which Kripke frames the nodes of which represent so-called *consistent histories,* and of those, which ones uniquely determine the evolution of state in the system represented by sets of propositional variables, each representing a particular *event* in that frame. By representing such events using *event algebras* consisting of operations of concatenation, alternation and closure of sets of composite events, we ought to be able to make sound the connection between the probabilistic interpretation of observed events and the intuitionistic formalisation of knowledge as *Symbolic Logic* to provide an explanation for the principles of quantum mechanics. The connection will be found to be a formal one ocurring at the level of knowledge of the *whole system* in terms of the co-variant properties of the manifold different possible forms that a consistent description of that evolution may take.

We have yet to explain how to turn *type operators* themselves into proof-schemæ. We know that the type *combinators* such as $\mathsf{S}\,(\mathsf{K}\,\iota_1)\,\pi_1\,r$ for arbitrary $r\colon\langle\alpha\cdot\gamma\rangle$ correspond immediately to proof-schemæ because the type operators π_1 and ι_1 correspond directly to axioms ∧-E1: and ∨-I1: in (14) given on page 22, and we hypothesise an arbitrary instantiation C to interpret the type γ. But if the proof-program combinators are to be interpreted as *metamethods* for effectively presenting methods for effecting proofs, then the question is: how do we effect the calculation of a proof from an abstract proof-program with lambda bindings whose types are determined by the type inference rule ABS:? One way would be to turn it into a combinator using the combinatory abstraction translation but that is then an effective method for turning just that particular theorem into a proof: it is exactly one of those arbitrarily weakened proofs we wish to be able to analyse, so we will have thrown the proverbial baby out with the bath-water. A better way is to directly interpret the lambda abstraction introduced by ABS: as an instance of the intuitionistic tautology ¬-E:, which the reader will recall is an axiom of the form $\neg A \Rightarrow (A \Rightarrow B)$. On this view we have a generalisation of the assumption axiom $A \Rightarrow A$ we proved earlier: this one being at a higher level which is that of a *hypothetical deduction*. Thus the denotation of proof combinators such as B_{11} and B_{22} applied to other proof combinators is an effective presentation of an effective method for calculating a proof, which method is *represented* by *an actual schema of a formal proof;* which schema should in turn be understood to be a concrete construction of a statement in the language of an operational calculus, of a method which could be effected by *using* the intuitionistic inference rule ⇒-E :of the Hilbert-style proof system as an intensional *conversion* or *rewriting* rule.

The reader may be suffering a little from *abstraction-shock* at this point; symptoms of which include slight dizzyness, loss of concentration and an unaccountable urge to do something else—such as Category Theory, for example. But they should be assured by the following concrete example that it is *not* as bad as all that comes to! In fact we have already presented an example of such an effective presentation of the combinator B in (28). The Hindley-Milner algorithm was the means, given the combinator $\lambda p.\lambda q.\mathsf{S}\,(\mathsf{K}\,p)\,q$, of effecting the presentation of an abstract method of presenting proofs by inferring the type $\mathsf{B}\colon\forall\gamma.\forall\alpha.\forall\beta.(\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma$ and computing the associated derivation Δ presented in (29). At that point we did not give an interpretation as an intuitionistic proof, as we had done for (22 – 24), because we did not actually know *how* to intensionally interpret the type abstraction with λ-binding. The problem was the apparent extra degree of freedom in the instantiated type $\mathsf{B}\,\iota_1\,\pi_1$ which prevented the formulæ in (30) from reducing. This was not apparent in the proof (15–17) from which we originally abstracted the combinator B. The reason is that we did not use the most general instantiation of the axiom schema S, so we *proved too much* and the resulting proposition schema $A \wedge B \Rightarrow A \vee B$ did not have enough information to show which of the two distinct normal forms of that abstract proof-schema should be taken to be representative of the actual proof used. In other words, given an arbitrary instance of such a formula, we would not be able to determine syntactically which of the two *enantiomorphic* types B_{11} or B_{22} it was actually an instance of. So this proposition-schema would not yield the method of proof to any deterministic, syntax-directed deconstruction.

The problem is that the Hilbert-style formalism is quantifier-free and the language cannot therefore distinguish between an instantiation in the qualified sense of being a *generic* or abstract instantiation, and an *actual* or *particular* instantiation in the unqualified sense, which is a concrete construction. These are the two different ways Aristotle describes, in which the student may be said to know something such as the fact that the angles in any triangle are together equal to two right-angles. And the problem in any science is that this universal, qualified sense of knowing is a pre-requisite for the unqualified knowledge of the fact in some particular circumstance. The reason is simply that having interior angles which sum to two right angles is the essence of *what it is to be* a triangle, so if the student did not already know this, they would not *recognise* the figure was in fact a triangle and so they would not be capable of making that judgement which is in fact the conclusion of the proof of which they would have witnessed a demonstration, had they this necessary qualified knowledge of the universal. So as Per Martin-Löf says in his Siena lectures [12], what one must know before one has the right to make a judgement is the *most abstract* knowledge: that being what it is which constitutes *Truth itself.* But in the language of the Hilbert-style formalism, there is no sense in which one could even express this distinction between the generic instantiation and the particular instance because they are *both* denoted by the *same* set of metalanguage variables $A, B, C, \ldots$. So when we say "a particular instance of the formula $A \vee \neg A$" and when we say "the proposition-schema $A \vee \neg A$" we mean two quite different things by these two uses of the formula $A \vee \neg A$. It is impossible to prove the latter intuitionistically, but it may or may not be possible to prove the former, depending upon what the particular formula denoted by A happens to actually be. Thus every axiom such as $A \Rightarrow (B \Rightarrow A)$, for example, is a *theorem schema* because it is provable whatever actual formula we substitute for A and B, but $A \vee \neg A$ is intuitionistically provable *only if* either A or $\neg A$ is intuitionistically provable. But we do not need to present a proof of either of the propositions denoted by A and B in order to present a proof of the proposition denoted by $A \Rightarrow (B \Rightarrow A)$. In the case of the formula $A \wedge B \Rightarrow A \vee B$ which is also provable for any values of A or B whatsoever, we *still* need a proof of *at least one* of A or B in order to present a proof of $A \wedge B \Rightarrow A \vee B$, because we need at least one of the two to carry the inference from the antecedent to the consequent.

For example, we interpret $\mathsf{B}\,\iota_1\,\pi_1 \equiv (\lambda p.\lambda q.\mathsf{S}(\mathsf{K}\,p)q)\,\iota_1\,\pi_1 \twoheadrightarrow_* \mathsf{S}(\mathsf{K}\,\iota_1)\pi_1$, as an effective method of producing proofs of the theorem $A \wedge B \Rightarrow A \vee C$ thus

$(A \wedge B \Rightarrow A \Rightarrow A \vee C) \Rightarrow (A \wedge B \Rightarrow A) \Rightarrow A \wedge B \Rightarrow A \vee C$	1	ITAUT: $\Rightarrow$-I2: $[A \wedge B/A, A/B, A \vee C/C]$
$(A \Rightarrow A \vee C) \Rightarrow A \wedge B \Rightarrow A \Rightarrow A \vee C$	2	ITAUT: $\Rightarrow$-I1: $[A \Rightarrow A \vee C/A, A \wedge B/B]$
$A \Rightarrow A \vee C$	3	ITAUT: $\vee$-I1:$[A/A, C/B]$
$A \wedge B \Rightarrow A \Rightarrow A \vee C$	4	MP: 2,3
$(A \wedge B \Rightarrow A) \Rightarrow A \wedge B \Rightarrow A \vee C$	5	MP: 1,4
$A \wedge B \Rightarrow A$	6	ITAUT: $\wedge$-E1:$[A/A, B/B]$
$A \wedge B \Rightarrow A \vee C$	7	MP: 5,6

So we have presented an effective method which converts an *instance of a proof* of $A \wedge B$ into an instance of a proof of $A \vee C$. All we need to do is to take the proof of an instance of $A \wedge B$ and use the rule MP: to eliminate the cut and reduce the above

proof schema to a proof of an instance of the simpler proposition schema $A \vee C$:

$$\text{MP:}\ \frac{\text{CUT:}\ \dfrac{\vdots}{A \wedge B \Rightarrow A \vee C} \qquad \text{CUT:}\ \dfrac{\vdots}{A \wedge B}}{A \vee C}$$

Given a theorem schema of one of the most general forms $A \wedge B \Rightarrow A \vee C$ or $A \wedge B \Rightarrow C \vee B$ we know which conjunct is material: because it is the sub-formula of an hypothesis, but the less general schema $A \wedge B \Rightarrow A \vee B$ is a generic instance of *both* of these, so we have lost the means to determine which side of the disjunct is material. Neither do we know which of the two *other* forms it may have been, these other forms being $A \wedge B \Rightarrow C \vee A$ or $A \wedge B \Rightarrow B \vee C$ which are produced by the 'crossed' proof combinators $\mathsf{B}_{12} \equiv \mathsf{B}\,\iota_1\,\pi_2$ and $\mathsf{B}_{21} \equiv \mathsf{B}\,\iota_2\,\pi_1$. So we are missing some *bits* of information which are the respective orientations of the logical constants $\wedge$ and $\vee$ in the formula $A \wedge B \Rightarrow A \vee B$. These extra bits are given by the subscripts of the injection and projection operators $\iota_{1,2}$ and $\pi_{1,2}$ in the intensional proof-program combinator, but this information is not available from the *rules* given by the extensional axiom schemæ, it is only available at the level of *whole deductions* which are concrete constructions and which can maintain the relations between their parts which make the distinctions between up and down, and left and right meaningful. The up-down distinction seems to be the more troublesome.

The above proof-schema is the *extensional* expression of the combinator, which is represented by the type-scheme $\forall\gamma.\forall\alpha.\forall\beta.\langle\alpha\cdot\beta\rangle \to \langle\alpha \mid \gamma\rangle$. The extensional methods are methods of effectively converting instances of proof schemæ ito instances of other proof schemæ. The *intensional* expression of the combinator on the other hand is a method for converting *methods* of constructing proof schemæ into other methods of constructing proof schemæ. The intensional representation of the combinator is just $\mathsf{B}\,\iota_1\,\pi_1$. The proof schema is effectively presented by applying Hindley-Milner type inference to infer a generic instance of the type-scheme of this expression. That is to say that the proof-schemæ are calculated from the proof-program which is presented as a derivation Δ of the theorem $\Gamma \vdash e : \sigma$ and a set of substitutions S given by the type inference algorithm $W(\Gamma, e)$, given a set of type assumptions Γ and the combinator expression $e = \mathsf{B}\,\iota_1\,\pi_1$:

$\Gamma \vdash \mathsf{S}:(\langle\alpha\cdot\beta\rangle \to \alpha \to \langle\alpha \mid \gamma\rangle) \to (\langle\alpha\cdot\beta\rangle \to \alpha) \to \langle\alpha\cdot\beta\rangle \to \langle\alpha \mid \gamma\rangle$	1	ITAUT: $\mathsf{S}:[\langle\alpha\cdot\beta\rangle/\alpha, \alpha/\beta, \langle\alpha \mid \gamma\rangle/\gamma]$
$\Gamma \vdash \mathsf{K}:(\alpha \to \langle\alpha \mid \gamma\rangle) \to \langle\alpha\cdot\beta\rangle \to \alpha \to \langle\alpha \mid \gamma\rangle$	2	ITAUT: $\mathsf{K}:[\alpha \to \langle\alpha \mid \gamma\rangle/\alpha, \langle\alpha\cdot\beta\rangle/\beta]$
$\Gamma \vdash \iota_1 : \alpha \to \langle\alpha \mid \gamma\rangle$	3	ITAUT: $\iota_1:[\alpha/\alpha, \gamma/\beta]$
$\Gamma \vdash \mathsf{K}\,\iota_1 : \langle\alpha\cdot\beta\rangle \to \alpha \to \langle\alpha \mid \gamma\rangle$	4	COMB: 2,3
$\Gamma \vdash \mathsf{S}\,(\mathsf{K}\,\iota_1) : (\langle\alpha\cdot\beta\rangle \to \alpha) \to \langle\alpha\cdot\beta\rangle \to \langle\alpha \mid \gamma\rangle$	5	COMB: 1,4
$\Gamma \vdash \pi_1 : \langle\alpha\cdot\beta\rangle \to \alpha$	6	ITAUT: $\pi_1:[\alpha/\alpha, \beta/\beta]$
$\Gamma \vdash \mathsf{S}\,(\mathsf{K}\,\iota_1)\,\pi_1 : \langle\alpha\cdot\beta\rangle \to \langle\alpha \mid \gamma\rangle$	7	COMB: 5,6
$\Gamma \vdash \mathsf{S}\,(\mathsf{K}\,\iota_1)\,\pi_1 : \forall\gamma.\langle\alpha\cdot\beta\rangle \to \langle\alpha \mid \gamma\rangle$	8	GEN: 7
$\Gamma \vdash \mathsf{S}\,(\mathsf{K}\,\iota_1)\,\pi_1 : \forall\beta.\forall\gamma.\langle\alpha\cdot\beta\rangle \to \langle\alpha \mid \gamma\rangle$	9	GEN: 8
$\Gamma \vdash \mathsf{S}\,(\mathsf{K}\,\iota_1)\,\pi_1 : \forall\alpha.\forall\beta.\forall\gamma.\langle\alpha\cdot\beta\rangle \to \langle\alpha \mid \gamma\rangle$	10	GEN: 9

Now when we apply this combinator to a combinator of type $\langle\alpha\cdot\beta\rangle$ which calculates proofs of the schema $A \wedge B$ we find we retain an independent choice of instantiation

for the term C in the final proof $A \vee C$. This choice is represented by the universally quantified γ in the generic typescheme. Note that the origin of this type parameter is not the final argument type $\langle \alpha \cdot \beta \rangle$ of the combinator B, rather it was introduced by the universally quantified γ in the primitive combinator ι_1. Then on the other hand we have the mirror image of the combinator B_{11} which is $\mathsf{B}_{22} \equiv \mathsf{B}\,\iota_2\,\pi_2$. Together, these two give an effective method which, given an effective method of producing a proof of $A \wedge B$, effectively present an effective method of producing a proof of $A \vee B$.

The reason we use them together is that we want to retain the potential for proving this theorem in *both* possible ways. Then, given a method for effecting a proof of $A \vee B$ we will be able to deconstruct it either way. So only when we need to analyse the proof will we commit to the decision as to which way it is proved. Thus the type inference algorithm takes us from the purely extensional view of a theory as a set of formulæ according to some schemæ, to an intensional method for effecting the instantiation of schemæ by constructing proof-program combinators, each of which are an entirely abstract framework for effecting *any* schema which is a generic instance of its most general or *principal* polymorphic type-scheme.

We can illustrate the utility of this by a parable of Smullyan's concerning a place where there is a fork in a path. One way leads to Heaven, and the other to somewhere entirely different. There are three oracles who each know everything. One of them always lies, one always tells the truth and the third lies or tells the truth at random. A traveller may only ask two questions, each directed to one particular oracle, but she doesn't know which one is which, although she does know that they have these particular characters. The problem is whether she can ask the right sort of questions to be able to learn for certain which is the path to Heaven. One possible strategy is for her to ask one oracle, B, say, whether another, A, say, is more likely to tell the truth than the third, C. If he answers "yes," then she should address her next question to C, but if his answer is "no" then she should address it to A. So the second question is addressed to the oracle who is said by B to be the one *least* likely to tell the truth. And the question she should ask that other oracle is "*If* I were to ask *you* whether the right-hand path leads to Heaven, what would you say?" The strategy works because the answer to the first question allows her to direct the second question to one who she knows either always lies, or always tells the truth. Then, by framing a reflexive question she can interpret the answer as being the truth, because the one who always speaks the truth will speak the truth, and so will the other because he will lie about the lie he would hypothetically have told. Let P be a propositional variable which is true if the right-hand path leads to Heaven, and false otherwise. Then she has determined one of either P or $\neg P$, without making an evident judgement. That is, without ever knowing a *particular* proof. She can do this because she can reason indefinitely from the basis of the consistency of the system of deduction alone. And she does this by reasoning with *potential* proofs in the form of proof-programs which, given actual knowledge, *would* produce actual conclusions. They can do this because the two programs, each representing a possible deduction resulting from the answer to a question, have a common bound variable. Let Q be a propositional variable which is true if, according to Bs answer to the first question, the oracle A *is* more

likely to tell the truth than the oracle C, and false otherwise. Because both P and Q are propositional *variables* they are not determined in any Kripke frame. So they denote truth values *indefinitely.* Thus we write $\neg P$ and $\neg Q$ to denote, *not* that P and Q are false, but to denote the *opposite* of whatever they *might* possibly denote. And this is the sense in which the proof-programs share the common variable: the binding by logical negation is an *anti-correlation* which does not determine the value of the variable; rather it simply establishes a connection in terms of knowledge we have about how the different parts of the system are *necessarily* related. We can prove this in an intuitionistic formal system however, so long as we assume that one or other of P or $\neg P$ holds. Then we can show in two separate deductions, that whichever of these is the case, we will learn what that case is from the answers to these two questions.

This is simply taking seriously the idea, expressed in [12], that intuitionistic formal truth is identical with the possession of a method of formal proof. Thus the only statements that are permitted in the logic, that is to say the only propositions there *are*, are ones which are known to *be* true, namely the tautologies. Because if we have a method for proving a proposition, then it can only be because it is in fact a true proposition. Now the problem is that, because the truth values of propositional variables are not individually determined by the logical constants, the logic offers no method of determining their truth, and it seems some intuitionistic logicians do not in fact believe that the propositional variables have an independent truth. Then they cannot have any meaning either so one wonders why these men go to the trouble of formalising logic at all. What is the point, if one cannot use it to make deductions about the objects of experience, i.e. knowledge? But if one does accept that the propositional variables are variables in Quine's pronomial sense of the word, and that therefore they *do* denote actual values, we can reason from the basis that those possible actual values are models for the interpretation of propositions which refer to particular propositional variables. But what we learn from each of these two deductions is something quite different from the relative truth or falsity of P and $\neg P$. What we learn are two different things: one is the truth about what the truthful oracle would have said and the other is a lie about what the lying oracle would have said. One answer is therefore a positive answer, and the other is a double-negation.

We will now illustrate how to make effective methods which combine effective methods and convert to different effective methods. Suppose we wish to prove any instance of the proposition schema $A \Rightarrow B \Rightarrow A \vee C$. We know we have an axiom $A \Rightarrow B \Rightarrow A \wedge B$ and we have the proof combinator B_{11} which can effect proofs of the proposition schema $A \wedge B \Rightarrow A \vee C$. Using type operators $(\alpha, \beta)\Pi$ and $(\alpha, \beta, \gamma)\Sigma$ to represent the type of the constructor and deconstructor operations $\lambda a{:}\alpha.\lambda b{:}\beta.\langle a \cdot b\rangle{:}(\alpha, \beta)\Pi$ and $\lambda f{:}\alpha \to \gamma.\lambda g{:}\beta \to \gamma.\lambda\langle a{:}\alpha.f\ a \mid b{:}\beta.g\ b\rangle{:}(\alpha, \beta, \gamma)\Sigma$, we now add assumptions for concatenation (or *conjection*) and alternation (or *ejection*) thus:

$$\Gamma \equiv \Gamma_\gamma \equiv \Gamma_\beta \cup \{\Pi{:}\forall\alpha.\forall\beta.\alpha \to \beta \to \langle\alpha \cdot \beta\rangle,$$
$$\Sigma{:}\forall\alpha.\forall\beta.\forall\gamma.(\alpha \to \gamma) \to (\beta \to \gamma) \to \langle\alpha \mid \beta\rangle \to \gamma\}.$$

We do this in preference to changing the object language of expressions over which the Hindley-Milner type inference algorithm is defined. That language includes only

lambda and let binding because all other compound types may be represented by type operators such as these two we have just presented; the result being, as we observed earlier, that all polymorphic types are generic instances of the most general polymorphic type: the function-space type constructor $(\alpha, \beta) \rightarrow$. Thus the type inference algorithm may be expressed quite simply in terms of these two primitive binding operators, lambda and let, and the inferences made in respect of these primitives generalising automatically to any compound types whatsoever which are constructed from generic instances of them. This is the formal basis of the denotational semantics of, for example, the Church numeral representation of numbers as n-fold composition of function application. Thus we do not need *any* primitive types of individuals such as numbers or truth values: we can effectively compute instances of any type we care to define using only abstract polymorphic type operators without ever committing to any concrete representation whatsover. This supports our view of effective calculation as an operational calculus for interpreting Category Theory.

Lobster Quadrille

The Mock Turtle sighed deeply, and drew the back of one flapper across his eyes. He looked at Alice, and tried to speak, but for a minute or two sobs choked his voice.

'Same as if he had a bone in his throat,' said the Gryphon: and it set to work shaking him and punching him in the back. At last the Mock Turtle recovered his voice, and, with tears running down his cheeks, he went on again:—

'You may not have lived much under the sea—' ('I haven't,' said Alice)—'and perhaps you were never even introduced to a lobster—' (Alice began to say 'I once tasted—' but checked herself hastily, and said 'No, never') '—so you can have no idea what a delightful thing a Lobster Quadrille is!'

'No, indeed,' said Alice. 'What sort of a dance is it?'

'Why,' said the Gryphon, 'you first form into a line along the sea-shore—'

'Two lines!' cried the Mock Turtle. 'Seals, turtles, salmon, and so on; then, when you've cleared all the jelly-fish out of the way—'

'*That* generally takes some time,' interrupted the Gryphon.

'—you advance twice—'

'Each with a lobster as a partner!' cried the Gryphon.

'Of course,' the Mock Turtle said: 'advance twice, set to partners—'

'—change lobsters, and retire in same order,' continued the Gryphon.

'Then, you know,' the Mock Turtle went on, 'you throw the—'

'The lobsters!' shouted the Gryphon, with a bound into the air.

'—as far out to sea as you can—'

'Swim after them!' screamed the Gryphon.

'Turn a somersault in the sea!' cried the Mock Turtle, capering wildly about.

'Change lobsters again!' yelled the Gryphon at the top of its voice.

'Back to land again, and that's all the first figure,' said the Mock Turtle, suddenly dropping his voice; and the two creatures, who had been jumping about like mad things all this time, sat down again very sadly and quietly, and looked at Alice.

'It must be a very pretty dance,' said Alice timidly.

'Would you like to see a little of it?' said the Mock Turtle.

'Very much indeed,' said Alice.

'Come, let's try the first figure!' said the Mock Turtle to the Gryphon. 'We can do without lobsters, you know. Which shall sing?'

'Oh, *you* sing,' said the Gryphon. 'I've forgotten the words.'

So they began solemnly dancing round and round Alice, every now and then treading on her toes when they passed too close, and waving their forepaws to mark the time, while the Mock Turtle sang this, very slowly and sadly:—

"'Will you walk a little faster?" said a whiting to a snail.
"There's a porpoise close behind us, and he's treading on my tail."

"See how eagerly the lobsters and the turtles all advance!
They are waiting on the shingle—will you come and join the dance?"

"Will you, won't you, will you, won't you, will you join the dance?
Will you, won't you, will you, won't you, won't you join the dance?"

"You can really have no notion how delightful it will be
When they take us up and throw us, with the lobsters, out to sea!"

But the snail replied "Too far, too far!" and gave a look askance—
Said he thanked the whiting kindly, but he would not join the dance.

Would not, could not, would not, could not, would not join the dance.
Would not, could not, would not, could not, could not join the dance.

"What matters it how far we go?" his scaly friend replied.
"There is another shore, you know, upon the other side."

"The further off from England the nearer is to France—
Then turn not pale, beloved snail, but come and join the dance."

"Will you, won't you, will you, won't you, will you join the dance?
Will you, won't you, will you, won't you, won't you join the dance?"'

'Thank you, it's a very interesting dance to watch,' said Alice, feeling very glad that it was over at last: 'and I do so like that curious song about the whiting!'

'Oh, as to the whiting,' said the Mock Turtle, 'they—you've seen them, of course?'

'Yes,' said Alice, 'I've often seen them at dinn—' she checked herself hastily.

'I don't know where Dinn may be,' said the Mock Turtle, 'but if you've seen them so often, of course you know what they're like.'

'I believe so,' Alice replied thoughtfully. 'They have their tails in their mouths—and they're all over crumbs.'

'You're wrong about the crumbs,' said the Mock Turtle: 'crumbs would all wash off in the sea. But they *have* their tails in their mouths; and the reason is—' here the Mock Turtle yawned and shut his eyes.—'Tell her about the reason and all that,' he said to the Gryphon.

'The reason is,' said the Gryphon, 'that they *would* go with the lobsters to the dance. So they got thrown out to sea. So they had to fall a long way. So they got their tails fast in their mouths. So they couldn't get them out again. That's all.'

'Thank you,' said Alice, 'it's very interesting. I never knew so much about a whiting before.'

'I can tell you more than that, if you like,' said the Gryphon. 'Do you know why it's called a whiting?'

'I never thought about it,' said Alice. 'Why?'

'*It does the boots and shoes,*' the Gryphon replied very solemnly.

Alice was thoroughly puzzled. 'Does the boots and shoes!' she repeated in a wondering tone.

'Why, what are *your* shoes done with?' said the Gryphon. 'I mean, what makes them so shiny?'

Alice looked down at them, and considered a little before she gave her answer. 'They're done with blacking, I believe.'

'Boots and shoes under the sea,' the Gryphon went on in a deep voice, 'are done with a whiting. Now you know.'

'And what are they made of?' Alice asked in a tone of great curiosity.

'Soles and eels, of course,' the Gryphon replied rather impatiently: 'any shrimp could have told you that.'

'If I'd been the whiting,' said Alice, whose thoughts were still running on the song, 'I'd have said to the porpoise, "Keep back, please: we don't want *you* with us!"'

'They were obliged to have him with them,' the Mock Turtle said: 'no wise fish would go anywhere without a porpoise.'

'Wouldn't it really?' said Alice in a tone of great surprise.

'Of course not,' said the Mock Turtle: 'why, if a fish came to *me*, and told me he was going on a journey, I should say "With what porpoise?"'

'Don't you mean "purpose"?' said Alice.

'I mean what I say,' the Mock Turtle replied in an offended tone. And the Gryphon added 'Come, let's hear some of *your* adventures.'

Hypothetical Proof

We need another combinator, D, say, which can compose these two methods into a method which, given a method of proving A, can produce an effective method which converts a method for proving B into a method for proving $A \vee C$. We use type variables α, β, γ etc. to represent methods, and the function space type constructor to represent conversion. Then the type constructor we need is a generic instance of the type-scheme $\forall\alpha.\forall\beta.\forall\gamma.\forall\delta.(\alpha \to \beta) \to (\gamma \to \delta \to \alpha) \to \gamma \to \delta \to \beta$. So we need a conversion which takes the conversion $f : \alpha \to \beta$ and the conversion $g : \gamma \to \delta \to \alpha$, and then the arguments $p : \gamma$ and $q : \delta$. It must then apply these to g to get a result of type α which it can apply to f to produce the final result of type β. Thus the combinator is $\mathsf{D} \equiv \lambda f.\lambda g.\lambda p.\lambda q.f\,(g\ p\ q)$. The type inference mirrors this thought

process:

$\Gamma_4 \vdash f : \alpha \to \beta$	1	ASSUM: 11
$\Gamma_4 \vdash g : \gamma \to \delta \to \alpha$	2	ASSUM: 10
$\Gamma_4 \vdash p : \gamma$	3	ASSUM: 9
$\Gamma_4 \vdash g\ p : \delta \to \alpha$	4	COMB: 2,3
$\Gamma_4 \vdash q : \delta$	5	ASSUM: 8
$\Gamma_4 \vdash g\ p\ q : \alpha$	6	COMB: 4,5
$\Gamma_4 \vdash f\ (g\ p\ q) : \beta$	7	COMB: 1,6
$\Gamma_3 \vdash \lambda q.f\ (g\ p\ q) : \delta \to \beta$	8	ABS: 7,5
$\Gamma_2 \vdash \lambda p.\lambda q.f\ (g\ p\ q) : \gamma \to \delta \to \beta$	9	ABS: 8,3
$\Gamma_1 \vdash \lambda g.\lambda p.\lambda q.f\ (g\ p\ q) : (\gamma \to \delta \to \alpha) \to \gamma \to \delta \to \beta$	10	ABS: 9,2
$\Gamma \vdash \lambda f.\lambda g.\lambda p.\lambda q.f\ (g\ p\ q) : (\alpha \to \beta) \to (\gamma \to \delta \to \alpha) \to \gamma \to \delta \to \beta$	11	ABS: 10,1
$\Gamma \vdash \mathsf{D} : \forall\delta.(\alpha \to \beta) \to (\gamma \to \delta \to \alpha) \to \gamma \to \delta \to \beta$	12	GEN: 11
$\Gamma \vdash \mathsf{D} : \forall\gamma.\forall\delta.(\alpha \to \beta) \to (\gamma \to \delta \to \alpha) \to \gamma \to \delta \to \beta$	13	GEN: 12
$\Gamma \vdash \mathsf{D} : \forall\beta.\forall\gamma.\forall\delta.(\alpha \to \beta) \to (\gamma \to \delta \to \alpha) \to \gamma \to \delta \to \beta$	14	GEN: 13
$\Gamma \vdash \mathsf{D} : \forall\alpha.\forall\beta.\forall\gamma.\forall\delta.(\alpha \to \beta) \to (\gamma \to \delta \to \alpha) \to \gamma \to \delta \to \beta$	15	GEN: 14

Where $\Gamma_1 \equiv \Gamma \cup \{f : \alpha \to \beta\}$, $\Gamma_2 \equiv \Gamma_1 \cup \{g : \gamma \to \delta \to \alpha\}$, $\Gamma_3 \equiv \Gamma_2 \cup \{p : \gamma\}$, and $\Gamma_4 \equiv \Gamma_3 \cup \{q : \delta\}$.

There are two points to note about this deduction. Firstly, it is entirely hypothetical: it makes no references to any axioms. The only premisses are hypotheses introduced by assumption and discharged by an instance of the rule ABS:. The only other rules of inference are COMB:, which corresponds to a conversion by the Hilbert-style inference rule MP:, and the rule GEN: which generalises a type variable not free in the assumptions; and since all assumptions are discharged before this rule is applied, the side-condition always holds. Thus the combinator $\mathsf{D} : \forall\alpha.\forall\beta.\forall\gamma.\forall\delta.(\alpha \to \beta) \to (\gamma \to \delta \to \alpha) \to \gamma \to \delta \to \beta$ corresponds to a proof-program combinator which, given instances of propositions α, β, γ and δ, and given combinators f and g which effect the proof-conversions $\alpha \to \beta$ and $\gamma \to \delta \to \alpha$, presents an effective method for the conversion $\gamma \to \delta \to \beta$. And because this combinator is a method for effecting an intuitionistic proof, it does in fact represent all the knowledge we need to be certain that the proposition $(A \Rightarrow B) \Rightarrow (C \Rightarrow D \Rightarrow A) \Rightarrow C \Rightarrow D \Rightarrow B$ is true.

The second point to note about the proof is that the process of producing the intensional representation of $\mathsf{D} \equiv \lambda f.\lambda g.\lambda p.\lambda q.f\ (g\ p\ q)$ was completely mechanical; it was simply programming. We read the type-scheme $\forall\alpha.\forall\beta.\forall\gamma.\forall\delta.(\alpha \to \beta) \to (\gamma \to \delta \to \alpha) \to \gamma \to \delta \to \beta$ from left to right as a list of assumptions and take the consequent of the final implication as a functional specification of the result the combinator must effect, which we call the *goal*. Assigning variables $f, g, h, \ldots$ to function types and propositional variables $p, q, \ldots$ to empty type variables we have the expression $\lambda f.\lambda g.\lambda p.\lambda q.G$ where G is an expression we must construct which will have the type β under the assumptions $f : \alpha \to \beta$, $g : \gamma \to \delta \to \alpha$, $p : \gamma$ and $q : \delta$. The only possible assumption we could use to resolve the goal $G : \beta$ is f so we write down $G \equiv f\ G'$ and now we have a new goal $G' : \alpha$ which is some expression of type α which, on application of f will convert to the original goal G. So we can forget about G now, because it is hypothetically solved! The only possible assumption that

could resolve the sub-goal G' is g so we write $G' \equiv g\ G''$ and the new subgoal is now a term $G'':\gamma$. Again, we have hypothetically resolved G', so this is the only goal and it is satisfied immediately by p so we write $G'' \equiv p\ G'''$ which is our new subgoal of type δ and which is immediately satisfied by q. So we write $G''' \equiv q$ and we have resolved all the sub-goals and the original goal G at once. Substituting back we have $G \equiv f\ (g\ p\ q)$ and then we discharge the assumptions by the addition of the lambda bindings to give $\lambda f.\lambda g.\lambda p.\lambda q.f\ (g\ p\ q)$.

Thus in type inference we analyse the expression to determine the type, and in *type inhabitation*, which is the process we have just described, we analyse the type to determine the expression. Considered as extensional proof-schemæ, these two processes are almost identical in structure: one may read the type inference from the top down as a process of deduction of the expression determined by the type, or from the bottom up as a process of type inference determined by the expression. Indeed one could modify the type inhabitation inference process we have described to construct the type inference deduction Δ at the same time as it constructs the expression, back-substituting the expression instead of the type substitutions. That is to say: term and type inference are essentially the same process. How is this possible? Simply because the type inference algorithm W as described in [11] never actually looks at the particular expression it is analysing: the results of the analysis are determined entirely by the *form* of the lambda expression: i.e. whether it is a lambda abstraction or let-binding of a variable in a sub-term, or a combination of two terms, or just a variable. In the case of it being a lambda-abstraction, W doesn't care which variable is being abstracted: it simply adds an empty type assumption in the form of a new variable β concerning whatever variable represents the abstracted term.

The fact that the type inhabitation inference is determined by the type has the obvious benefit of providing an *algebraic* means of constructing proofs by applying operators to construct generic instances of primitive combinators which are the proper axioms of the theory. This is algebraic in the sense that it is a recursive datatype which describes the grammar of expressions, just as ordinary algebra is the grammar of the language of Peano Arithmetic using operators and primitive types which are the numbers: the set of arithmetic expressions contains a *unit* and is closed under the operations of successor, addition, multiplication and exponentiation. If the resulting constructions are determined uniquely by the order of application of the constructors then the structure is called a *free algebra* and has the property that all constructions can be deconstructed deterministically by a process which works from the top down, each deconstruction step inferring the last construction step from the type of the whole construction.

If the set of primitive combinators is fixed and finite then, given a particular goal in the form of a combinator type-scheme, we can effectively use deterministic search to find out whether there is any intensional method of specifying that proof combinator. This gives us a kind of meta-combinator which Milner dubbed a *tactical*. A tactical is a method for constructing effective combinators: given a goal in the form of an abstract type-scheme the tactical attempts to construct a combinator which resolves the goal. It typically does this by breaking the goal into simpler subgoals and

recursively solving them. In the case of predicate logic without quantifiers this search is always bounded and so there is an algorithm which, if any given expression has a proof, will certainly find that proof, and if there is no proof, will positively indicate that fact in finite time. This method is called *pure PROLOG* and we can implement it in terms of Hindley-Milner type inference as a function which, given a generic instance σ of a type-scheme and assumptions Γ, will return either an error indicator ε_σ, or an expression e and a derivation Δ of the type-scheme, along with a particular *unifying substitution* U which affects the necessary instantiations of all free type variables in the assumptions.

The type system called System **F** is derived from a general definition of inductive types which Girard *et al* [7][6] attribute to Martin-Löf[7]. **F** adds second-order quantification on types into the language of the simply-typed lambda calculus using a schemæ of *universal abstraction* and *universal application*. Thus the type-schemes may be universally quantified, just like polymorphic types, and in addition the language has universal abstraction of types which is denoted using the lambda binding $\Lambda X.M$ to bind the type variable X in the expression M. The corresponding type will then be of the form $\Pi X.U$ so that if $\Lambda X.M$ has the type V then the type of $(\Lambda X.M)\,U$ is $V[U/X]$, which is the type V with U substituted for Λ-bound ocurrences of X in M. Therefore in System **F**, the language of the simply-typed lambda calculus is extended to include type expressions τ and these are simply either type variables, or they are one or other of the formulæ $U \to V$ or $\Pi X.V$, where U and V are types and $X \in \{S, T, U, V, W, X, Y, \ldots\}$ is a type variable.

Thus in **F** there are no monotypes; all types including the type Bool of propositions and the type Int of integers are abstract polymorphic types and so all programs are general algorithms for constructing, not values, but *type morphisms* of one sort or another. Furthermore, the primitive compound types such as products, sums and lists, for example, are all instances of the same inductive datatype and the only primitive type constructor that is not defined this way is the function space or arrow type $(\alpha, \beta)\to$. This means that the entire system of types can be realised by combinatory abstraction as instances of the intuitionistic axioms S and K of the Hilbert-style deduction system. Thus the entirety of either of System **F** (or indeed *HOL*) could be represented as an operational calculus in a Hilbert-style deduction system using the conversion rule MP: as the sole substitution mechanism.

We can derive the primitive combinator expressions for S and K as we did that for D. Then, rather than stating the assumptions Γ each time we can use the let expressions to introduce new type operators such as K, S and D as they are proved for the first time. Thereafter these will be available just as if they had been introduced *a priori* as axiom schemæ. For this we will use the syntactic construct $\lambda U = e.M$ which binds the principal type-scheme σ of the term e to the variable U in the expression M which variable U remains so bound until 'shadowed' by re-binding.

[6] See §11.3.5 on p.85.

[7] *A construction of the provable well-ordering of the theory of species* . Unpublished manuscript, 1970.

$\Gamma_2 \vdash p : \alpha$	1	ASSUM: 3
$\Gamma_1 \vdash \lambda q.p : \beta \to \alpha$	2	ABS: 1
$\Gamma \vdash \lambda p.\lambda q.p : \alpha \to \beta \to \alpha$	3	ABS: 2
$\Gamma \vdash \lambda p.\lambda q.p : \forall\beta.\alpha \to \beta \to \alpha$	4	GEN: 3
$\Gamma \vdash \lambda p.\lambda q.p : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha$	5	GEN: 4
$\Gamma_3 \vdash p : \alpha \to \beta \to \gamma$	6	ASSUM: 15
$\Gamma_3 \vdash r : \alpha$	7	ASSUM: 13
$\Gamma_3 \vdash p\ r : \beta \to \gamma$	8	COMB: 6,7
$\Gamma_3 \vdash q : \alpha \to \beta$	9	ASSUM: 14
$\Gamma_3 \vdash r : \alpha$	10	ASSUM: 13
$\Gamma_3 \vdash q\ r : \beta$	11	COMB: 9,10
$\Gamma_3 \vdash p\ r\ (q\ r) : \gamma$	12	COMB: 8,11
$\Gamma_2 \vdash \lambda r.p\ r\ (q\ r) : \alpha \to \gamma$	13	ABS: 12
$\Gamma_1 \vdash \lambda q.\lambda r.p\ r\ (q\ r) : (\alpha \to \beta) \to \alpha \to \gamma$	14	ABS: 13
$\Gamma \vdash \lambda p.\lambda q.\lambda r.p\ r\ (q\ r) : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$	15	ABS: 14
$\Gamma \vdash \lambda p.\lambda q.\lambda r.p\ r\ (q\ r) : \forall\gamma.(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$	16	GEN: 15
$\Gamma \vdash \lambda p.\lambda q.\lambda r.p\ r\ (q\ r) : \forall\beta.\forall\gamma.(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$	17	GEN: 16
$\Gamma \vdash \lambda p.\lambda q.\lambda r.p\ r\ (q\ r) : \forall\alpha.\forall\beta.\forall\gamma.(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$	18	GEN: 17
$\Gamma \vdash \mathsf{S} : (\alpha \to (\beta \to \alpha) \to \alpha) \to (\alpha \to \beta \to \alpha) \to \alpha \to \alpha$	19	ITAUT: 51 S: $[\alpha/\alpha, \beta \to \alpha/\beta, \alpha/\gamma]$
$\Gamma \vdash \mathsf{K} : \alpha \to (\beta \to \alpha) \to \alpha$	20	ITAUT: 52 K: $[\alpha/\alpha, \beta \to \alpha/\beta]$
$\Gamma \vdash \mathsf{S}\,\mathsf{K} : (\alpha \to \beta \to \alpha) \to \alpha \to \alpha$	21	COMB: 19,20
$\Gamma \vdash \mathsf{K} : \alpha \to \beta \to \alpha$	22	ITAUT: 52 K: $[\alpha/\alpha, \beta/\beta]$
$\Gamma \vdash \mathsf{S}\,\mathsf{K}\,\mathsf{K} : \alpha \to \alpha$	23	COMB: 21,22
$\Gamma \vdash \mathsf{S}\,\mathsf{K}\,\mathsf{K} : \forall\alpha.\alpha \to \alpha$	24	GEN: 23

We will use the syntactic form $\lambda U{:}\sigma.M$ to infer an expression e for the type-scheme σ. If the inference fails then within M the type σ will be bound to U as if what had been written was $\lambda\Omega.\lambda U = \varepsilon_\sigma\,\Omega.M$. Thus everything within the lexical scope of this lambda-binding of U will appear as the consequence of a hypothetical proof, predicated on falsity in the form of a combinator Ω of type $\forall\alpha.\alpha$. If the term inference succeeds and produces an expression e, then it will be as if we had written $\lambda U = e.M$ above, and the results will be indistinguishable as they appear in proof-schemæ. In proofs such let-bound expressions will appear as assumptions of type-schemes representing axioms introduced as premisses using the inference rule TAUT:, with any type quantifiers attached. These type-schemes may subsequently instantiated by inference rules INST:. The difference between let binding and lambda binding is succinctly explained in Damas and Milner's [11], it is essentially that the let bindings are principal type-schemes which are polymorphic in the sense that the same bound type-scheme σ may be instantiated in more than one way as distinct generic instances σ' and σ'', say, with $\sigma' < \sigma$ and $\sigma'' < \sigma$ but with no instance τ, say, such that both $\tau < \sigma'$ and $\tau < \sigma''$.

Inductive Types

An inductive datatype T may be represented as an *endofunction* on the set of objects of that type. That is to say, if the objects are a set X then the type is represented as a function $T:X \to X$, taking elements of the set onto other elements of the same set. Thus the type T represents is a recursive datatype: each application of T can be thought of as a *deconstruction* of an element of X into one of several different possible steps in its *construction*.

For example the recursive datatype $(\alpha)\texttt{list}$ of lists $l_{(\alpha)\texttt{list}}$ of objects a_α of type α is described by a grammar

$$l_{(\alpha)\texttt{list}} \leftarrow \texttt{Nil} \mid a_\alpha \texttt{ Cons } l_{(\alpha)\texttt{list}}.$$

This could be defined in Standard ML by the statement

```
datatype 'a list = Nil | Cons of 'a * 'a list
```

and it corresponds to a *type automorphism* $(\alpha)\texttt{list} \simeq \texttt{nil} \simeq \alpha \times (\alpha)\texttt{list}$. The automorphism is an isomorphism of the type *onto itself*. When we think of the relation $\simeq$ as a set of ordered pairs $X \times X$ then the elements of this relation correspond to *permutations* of the elements of X.

For example, the set X_1 of lists of the unit type $A_1 = \{\bullet\}$ is $X_1 = \{\texttt{Nil}, \bullet, \bullet\bullet, \bullet\bullet\bullet, \ldots\}$ which we may write $\{0, 1, 2, 3, \ldots\}$. This is particularly simple because the resulting set is a total order under the relation $<_l$ where $a <_l b$ if and only if a is an element in the construction of b. Then the permutation induced by the type automorphism relation $\simeq$ is seen to be

$$\begin{array}{c} 0\,1\,2\,3\,4\ldots \\ 0\,0\,1\,2\,3\ldots \end{array}$$

which is the permutation which takes each element to its predecessor. When the set A has two elements, $A_2 = \{\circ, \bullet\}$, for example, then we still have that every element has exactly one predecessor, but now there are two possible outcomes of any construction step: either an added $\circ$ or and added $\bullet$.

$$X_2 = \{\texttt{Nil}, \circ, \bullet, \circ\circ, \circ\bullet, \bullet\circ, \bullet\bullet, \circ\circ\circ, \circ\circ\bullet, \ldots\}$$

We can represent the resulting lists of length n as n-digit binary numbers. Then again the relation $\simeq$ induces a permutation on the set of lists, taking each n-element list to its $(n-1)$ element predecessor. The Hasse diagram of the set is an infinite binary tree with the permutation taking each node to its predecessor, so that under successive iterations of the permutation all nodes are eventually mapped to the root which is $\texttt{Nil}$. This phenomenon of *confluence* is a characteristic of computational processes.

Consider the equivalence relation $\simeq_f \subseteq X \times X$ on $X = X_1 \cup X_2$ which identifies all the lists of length n. This is induced by a function $f:X_2 \to X_1$ taking any n-element list

of X_2 to the equivalent list of the same length in X_1. Clearly whatever permutation is induced by $\simeq$ on the set of lists X_2 will respect the equivalence relation $\simeq_f$ in the sense that if $\simeq$ relates one element of $a \in X_2$ to another $b \in X_2$ then it relates the corresponding elements $f(a) \in X_1$ and $f(b) \in X_1$. Thus we have *monotinicty* of $\simeq$:

$$a \simeq b \Rightarrow f(a) \simeq f(b).$$

Of course the converse only holds if $a = b$, and this is an instance of a very general principle which is that computation can only reduce the information in the system. This is what makes computers useful: they allow us to make complex data *intelligible* and we do that by programming them to *abstract* irrelevant detail from the data and present it as a structure. So the process of computation is from disorder to order and it is precisely one of these structures of the general class of objects of type α that we are analysing using the type $(\alpha)\texttt{list}$ which induces an order on permutations of sequences of elements of the type α.

Now consider the whole class F_i of functions f_i which are endofunctions on X, meaning that they are of type $(\alpha)\texttt{list} \to (\alpha)\texttt{list}$, and which take lists of length i to other lists of length i. Each of these functions will similarly induce an equivalence relation $\simeq_{f_i}$ on X which will be respected by the $\simeq$ type automorphism, and this fact is what makes the type of lists a valid inductive datatype. It also means that the class of functions $f_i : X \to X$ serve to *characterise* the subsets of permutations on the elements of X which are set-wise invariant under the type isomorphism. Therefore a type isomorphism is a *symmetry*, and the class F_i of endofunctions on the type X gives one of the characteristic subgroups of the infinite symmetric group.

Functors

A functor T is one of a class of functions which act on functions and also on the underlying sets which are the domain and co-domain of those functions. Functors capture the essential features of any inductive type X. The sub-class `map` of the class F_i of functions from lists of length i to other lists of length i which are defined elementwise as mappings $f : A \to A$ is a simple example of a functor. It takes functions $f : \alpha \to \alpha$ to functions $F : (\alpha)\texttt{list} \to (\alpha)\texttt{list}$ and in so doing takes sets to sets $F_i : \varnothing \to \varnothing$ and $F_i : A^i \to A^i$ for all $i = 1, 2, \ldots$. These latter functions F_i are *induced* or *defined by induction* on the length i of lists.

Thus the inductive type $(\alpha)\texttt{list}$ may be represented by a functor $T(X) = 1 + A \times X$ where A is the set of objects of type α and X is the set of objects of type $(\alpha)\texttt{list}$. On the right-hand side, the terms are *alternations* and the products of which they comprised are the domains of the relevant type operators. The *nullary* or *constant* operator $\texttt{Nil} : (\alpha)\texttt{list}$ is represented by 1 and the *binary* operator $\texttt{Cons} : \alpha \times (\alpha)\texttt{list} \to (\alpha)\texttt{list}$ by the product $A \times X$. The functor T acts on the one hand on the set of lists by taking any set X to another set $\{\texttt{Nil}\}$ or to the set $\{a \in A, x \in X \mid a\,\texttt{Cons}\,x\}$ which is written as the *disjoint union* or *sumset* $\{\texttt{Nil}\} \uplus \{a \in A, x \in X \mid a\,\texttt{Cons}\,x\}$. Iterations

of this functor act as a closure operator, the least fixed-point of which is the set X of lists of elements of type A. On the other hand, T acts 'sideways' on functions $f_1 : 1 \to X$ and $f_2 : A \times X \to X$ taking them to $\texttt{Nil} : 1 \to X$ and $\texttt{Cons} : A \times X \to X$. The very special property of the type isomorphism $\simeq$ characterising the inductive type of lists comes from the algebra of operators $[\texttt{Nil}, \texttt{Cons}]$ being what is called *an initial T-algebra* of the functor T. This property of being an initial T-algebra means that it is unique up to isomorphism, and that is why the equivalence classes of the type isomorphism are respected by all inductively defined functions. These need not necessarily be functions from lists to lists, they could be functions from lists to any other type. For example the list functional `List.foldr` acts on functions $f_2 : \alpha \times \beta \to \beta$ and elements $f_0 : \beta$ to give functions $g : (\alpha)\texttt{list} \to \beta$. Here the element f_0 could have been given as a constant function `fn () => n` which is from the `unit` type to β and which is equivalent to $f_0 : 1 \to \beta$. Thus the mapping performed by the functional `List.foldr` is isomorphic to the mapping from the algebra $[f_0, f_2]$ to the algebra $[\texttt{Nil}, \texttt{Cons}]$ performed by the functor T.

The reason functors are useful is that they allow us to use a higher order of abstraction because we can abstract literally *all* of the irrelevant details from any inductive computation, leaving only the fundamental inductive structure of the calculation. In any significant calculation there will be many instances of the same inductive structure repeated over and over again. In principle all of these particular instances could have been derived from one general *parametric* abstract method.

The example we will use is that of *capture-avoiding substitution.* This is a general procedure which appears in any language with binders or quantifiers where variables are used as placeholders for abstracted detail. Typically, if there is any possibility of free variables in abstracted detail, then substitution of that detail for a variable which is under another binder could result in capture of the free variables. This is always bad because it results in information being added to the computation: the information that is added is related to the names of the bound variables, and these could have been different, resulting in a different set of free variables being captured. The fact that bound variables serve to join together instances of a particular abstracted detail means that the names of those variables must be immaterial, because whatever is substituted for one of them will be substituted for all, and when there are no longer any instances of a given variable under a binder then the binder is irrelevant to that part of the expression that follows. Therefore the meaning of an expression is invariant under uniform renaming of bound variables and their binders. So the fact that this so-called *alpha-conversion* could result in free variables becoming bound means that the result of a computation will depend on the particular choice of names of bound variables, so under some circumstances the results given by two calculations which have exactly the same meaning will be different. So in one or other or both, some information beyond that given in the formal specification of the problem has been added, and this extra information is what determines which computations give identical results and which don't.

Laura's page: Thank you Λaura. You bound a free radical and made the world a much, much happier place. You made me and you are everything to me. I love you.

Capture avoiding substitution is a process of renaming bound variables before carrying out a substitution so that no free variables in the expression being substituted are captured. For efficiency it is better to do the substitution in one pass through the structure of the expression, and this means accumulating a list of alpha conversions which must be carried out before substitutions are made. It is quite difficult to get this right, because there are several cases, each of which result in a different inductive structure. There may be free variables in the term being substituted, but none of them bound in the expression in which the substitution will take place. Or there may be binders potentially capturing free variables, but no instances of the variable being substituted for. In either case, there may be alpha conversions which must still be carried out as part of earlier substitutions.

Capture avoiding substitution is not something one would like to have to define again and again, however it *is* defined again and again and again. There is no standard library procedure for doing this. One of the reasons may be because it is actually very difficult to abstract out the detail of a general structural induction, simply because every concrete inductive structure is different, having a different set of type constructors. So even though two inductive structures may be isomorphic, no induction on the concrete constructions could be generalised because what determines the decisions about the course of the induction will be different in each case.

For example, we may have an inductive structure of terms which are *operators and arities,* being an n-ary operator, with variables or terms in each of the n places, and the possibility of universal quantification of variables. Such a structure represents the polymorphic type system of Standard ML. On the other hand we might have a datatype representing lambda expressions, or another also representing lambda expressions, but with an extra `let` construct. In all these cases, the abstract inductive structure of capture avoiding substitution is identical, but the concrete structure is different, so any implementation of capture avoiding substitution which uses a concrete data structure to decide the course of the induction will be specific to that particular inductive datatype.

Therefore we need a higher order of abstract datatype which allows us to define inductive procedures independently of the particular concrete foundation of the inductive structures over which they operate. The way we can do this is to abstract not over the concrete structures, but over the isomorphisms that exist between them. So instead of abstracting over *types* we abstract over *type-structures.* The analogous notion to a type in this higher order abstraction is something called a (type) *signature,* which is a *parametric* type structure called a *functor.*

Monads

A *monad* is a generic name for one of three specific classes of structure, all related. By *structure* we mean an n-tuple, and in the case of monads these are all triples. The first class of structures are called *Kleisli triples,* written $\langle T, \eta, \_^* \rangle$.

The Kleisli triple is a *signature,* which is to say that it gives the overall form of a whole class of *structures,* each of which is an instance of the signature. In the case of Kleisli triples, these are *abstract* signatures, parametric on the inductive type T, consisting of the type and two operators. The operators are η or *unit* and $\_^*$ which is called *bind*.

The type T is an extensible type, so we will use the notation $\mu.\alpha$ to indicate an element of a polymorphic type of some particular instance of T. The type constructor $\eta{:}\alpha \to \mu.\alpha$ takes a value of type α and produces a representation $\mu.\alpha$ of the computation of that value of type α. Such a representation may then be used as the basis for defining further computations of values which may be of types other than α. This extension of the monad is effected using the bind operator.

A value constructed by the unit type constructor is a computable or *potential* value. That is to say that it is a representation of a value that is potentially computable, but which has not *yet* been computed. The representation takes the form of an effective method for computing the value. The actual value is computed inside the monad. That is, the monad *represents* an effective method for carrying out computations of a certain type of computable value. For example, the bind operator takes (1) a method $\mu.\alpha$ of computing values of type α, and (2), a method $f{:}\alpha \to \mu.\beta$ of converting actual values of type α into methods $\mu.\beta$ of computing values of type β. Its action is to convert the combination of (2) and (1) into: (3) a method $\mu.\beta$ of computing values of type β. Such extensions of the range of the computations represented in the monad amount to varying the functor T in the parametric signature of the monad. Each use of bind effectively adds a new term to T, because it represents another type constructor which can be used to produce computations of values of type β.

Another way of describing these two operators is to say that bind is a way of converting an hypothetical method $\alpha \to \mu.\beta$ of effecting calculations $\mu.\beta$ of values of type β into an actual method $\mu.\beta$ of calculating them. The hypothetical method depends upon the existence of actual values of type α, so the conversion is effected by supposing the availability of a method $\mu.\alpha$ of calculating actual values of type α and then applying those values α to the hypothetical method $\alpha \to \mu.\beta$ to get an actual method $\mu.\beta$. Similarly the operator unit is a hypothetical method $\alpha \to \mu.\alpha$ of effecting calculations $\mu.\alpha$ of values of type α, and it works by hypothesising the existence of those actual values α!

So within a monad, methods of computation are composed together *functorially* to produce other methods of computation. Combining *methods* instead of the computations themselves is *intensional* rather than *extensional* and is therefore more useful in computing, because we can use the knowledge we have about *how* those methods effect the computation of the values they compute, and this gives us more information on which to base further computations using those values.

Monads in Standard ML

We will use Standard ML's functor declaration to define the signature of monad structures. The three components of the Kleisli triple will be denoted by ML an abstract ML type `X`, representing T, and two functions `unit` and `bind`.

The extensible type of values computed in the monad will be represented by Standard ML's `exception` datatype which allows new polymorphic type constructors to be added as the interpretation in the monad is extended to represent computations of different types. To do this we use an exception `Lambda` of type `exn` which allows us to bind new types into the monad. The exception and the type-structure signature are defined as follows:

```
exception Lambda of exn

signature Monad =
sig
   type A
   exception A of A
   type monad = exn
   val ShowM : monad -> A
   val UnitM : exn -> monad
   val MapM : (exn -> exn) -> monad -> monad
   val JoinM : monad -> monad
   val BindM : monad -> (exn -> monad) -> monad
end
```

The type abbreviation `monad` is used to indicate any instance of a computation represented. The exception `A` of abstract type `A` is used to bind instances of that abstract type, and `Lambda` is used to introduce the others. Terms of type `exn` represent polymorphic types α and β which may be represented in the monad. Thus a term of type `exn` $\rightarrow$ `exn` represents a polymorphic function from any one type to any other type but these types are not fixed as the abstract types `A` are, they are determined at run time. The functor is defined:

```
functor Monad(type X) : Monad where type A=X =
struct
   type A = X
   exception A of A
   type monad = exn
   val ShowM = fn (A x) => x | e => raise e
   val UnitM = fn x => (A (ShowM x))
   val MapM = fn f => fn (e as (A _)) => (Lambda o f) e | e => raise e
   val JoinM = fn (Lambda x) => x | e => raise e
   val BindM = fn m => fn f => JoinM(MapM f m)
end
```

To illustrate the functor monad in use we define some series of numbers by induction on $\mathbb{N}$. The *Peano Numbers* are the numbers defined as zero and the successors of other

numbers thus defined. That is, the least set containing zero and closed under the successor operator. The functor takes as parameters a type X to represent numbers, and constants x and y representing the zero element and the successor function respectively. The representation uses the operator `UnitM` to introduce the zero element to the monad, and it then threads the computation of successive elements of the series through the operator `BindM`.

```
signature PeanoNumbers =
sig
   structure Monad : Monad
   val x : Monad.A
   val y : Monad.A -> Monad.A
   val O : exn
   val S : exn -> exn
end

functor PeanoNumbers
   (type X
    val x : X
    val y : X -> X) : PeanoNumbers =
struct
   structure Monad = Monad(type X = X)
   open Monad
   val O = UnitM (A x)
   fun S t = let fun represent f x = UnitM (A (f (ShowM x)))
             in BindM t (represent y) end
   val x = ShowM O
   val y = ShowM o S o UnitM o A
end
```

The representation of Peano numbers in the monad may be as an iterated application of the successor function, as occurs in the Church numeral representation of the number, for example. To this effect we define the iterator. This gives us an easy way to repeatedly apply the representation of the successor function in the monad.

```
signature It =
sig
   structure Numbers : PeanoNumbers
   val omega : int -> Numbers.Monad.A
   val from : int -> Numbers.Monad.A -> Numbers.Monad.A
end
```

The functions `from` and `omega` are the interfaces which cause the actual computation of particular elements of the series. The function `omega` takes a single integer `n` and returns the n-th element of the series. The function `from` takes an integer n and a starting element `z` and calculates the n-th element after the start. The function `It` is internal to the structure. It composes `f` with itself `n` times and applies the result to `x`, which is effectively the n-th Church numeral.

```
functor It(type X) : It =
struct
```

```
   local
      fun It n =
         let fun body f x =
                let fun iter r 0 = r
                      | iter r n = iter (f r) (n-1)
                in iter x end
         in
            fn x => fn f => body f x n
         end
      val K = fn p => fn q => p
      val I = fn x => x
      val zero = K I
   in
      structure Numbers =
           PeanoNumbers(type X = (X -> X) -> X -> X
                        val x = zero
                        val y = fn f => fn x => (f x) o x)
      local
         open Numbers
         open Numbers.Monad
      in
         val from = fn n => fn z => It n z y
         val omega = fn n => from n x
      end
   end
end
```

Finally we wrap the iterator and the Peano numbers functors together because they're almost always used together:

```
functor Numbers
   (type X
    val x : X
    val y : X -> X) : It =
struct
   structure Numbers = PeanoNumbers(type X = X val x = x val y = y)
   local open Numbers
         open Numbers.Monad
         structure It = It(type X = X)
   in
      val from = fn n => fn z => It.omega n Numbers.y z
      val omega = fn n => from n Numbers.x
   end
end
```

The following are two obvious instances of the monad, one interpreting the Peano numbers as computations of the successive elements of the series of odd numbers:

```
structure Odd =
   Numbers
      (type X = int * int
       val x = (0, 1)
       val y = fn (m, n) => (m + 1, n + 2))
```

and the other interpreting them as computations of successive elements of the series of square numbers.

```
structure Square =
   Numbers
      (type X = int * int
       val x = (0, 1)
       val y = fn (m, _) =>
               let val m' = m + 1
                   val m'' = m + 2
               in (m', m'' * m'') end)
```

But more complex interpretations are possible. For example here is an interpretation of numbers as the successive steps in the calculation of the greatest common divisor of two numbers by Euclid's algorithm and the division algorithm.

```
structure Euclid'sAlgorithm =
   Numbers
      (type X = int * int
       val x = (1, 1)
       val y = fn (l, m) => if l = 1 orelse m = 1
                            then (l, m)
                            else if l > m
                                 then (l - m, m)
                                 else if m > l
                                      then (l, m - l)
                                      else (l, m))
```

We can compose series together using a functor which maps the interpretation in one monad to another interpretation in a different monad. The new monad uses the `map` function of the first monad to do the initial representation and it then applies a function f to the result. Each element is threaded through the inverse function g to get it into the original representation, and it is then mapped back to the new representation by f.

```
functor MapNumbers
   (structure Numbers : It
    type X
    val f : Numbers.Numbers.Monad.A -> X
    val g : X -> Numbers.Numbers.Monad.A) : It =
struct
   structure Numbers =
   struct
      structure Monad = Monad(type X = X)
      open Numbers.Numbers.Monad
      val x = f (ShowM Numbers.Numbers.O)
      val y = f o ShowM o UnitM o Numbers.Numbers.S o UnitM o A o g
      val O = Monad.UnitM(Monad.A x)
      val S = Monad.UnitM o Monad.A o y o Monad.ShowM
   end
   local
```

```
      open Numbers
      open Numbers.Monad
      structure It = It(type X = X)
   in
      val from = fn n => fn z => It.omega n Numbers.y z
      val omega = fn n => from n Numbers.x
   end
end
```

For example, here is a monad calculating the squares of successive odd numbers, which are the numbers m^2 where $m = 2n + 1$ for $n = 1, 2, 3, \ldots$:

```
structure SquareOfOdd =
   MapNumbers(structure Numbers = Odd
              type X = int * int
              val f = fn (l, m) => Square.it m
              val g = fn (n, p) => ((n - 1) div 2, n))
```

Composing the same two monads the other way around yields a monad interpreting numbers as 'odds of square numbers' which are the n-th odd number where $n = m^2$ for $m = 2, 3 \ldots$.

```
structure OddOfSquare =
   MapNumbers(structure Numbers = Square
              type X = int * int
              val f = fn (m, n) => Odd.it n
              val g = fn (n, _) => let val m = (sqrt n) - 1
                                   in (m, n) end)
```

The inverse function g in this case is the integer square root function which could be calculated using the `SquareNumbers` monad and a search, but the following is $O(\log n)$.

```
structure SquareRoot =
   Numbers
      (type X = int * int * int
       val x = (1, 1, 1)
       val y = fn (l, m, n) =>
                  let val p = (l + m) div 2
                  in
                     if p * p > n
                        then (l, p, n)
                        else if p * p < n
                                then (p, m, n)
                                else (p, p, n)
                  end)

val sqrt = fn n => #1 (SquareRoot.from (log2 n) (1, n, n))
```

Similarly the function `log2`, which calculates an upper bound on the number of steps needed by the square root algorithm, may be defined as a monad.

```
structure LogTwo =
   Numbers
      (type X = int * int * int
       val x = (1, 0, 1)
       val y = fn (l, m, n) =>
                  if m < n then (l + 1, 2 * m, n)
                           else (l, m, n))

val log2 = fn n => #1 (LogTwo.from 64 (0, 1, n))
```

Here is an interpretation of numbers as steps in the sieve of Eratosthenes. This function is as easy to read as it was to write.

```
structure Prime =
   Numbers
      (type X = int * int list
       val x = (3,[2])
       val y = let fun composite n ps =
                         List.exists (fn p => (n mod p) = 0) ps
               in fn (n, ps) =>
                  let val ps' = if composite n ps then ps else ps@[n]
                  in (n + 2, ps')
                  end
               end)
```

Primitive Recursive Functions

The primitive recursive functions are constructed from *primitives*, consisting of the constant zero and a unary successor function succ_1 together with *projection* functions $\mathsf{proj}_{m,n}$ of arity m where m and n are such that $1 \leq n \leq m$. These select the n-th argument. These primitives are combined using one of two *operators*, the first of which is called *generalised composition* or *substitution*, written $\mathsf{subs}_{m,n}$ with m and n integers such that $m, n \geq 1$. This takes an arity m function f and m functions g_i of arity n and it substitutes g_i for the i-th argument of f, giving a function of arity n. The other operator is *primitive recursion*, written rec_m with $m \geq 1$ which takes a function g of arity $m-1$ and a function h of arity $m+1$ and produces a function f of arity m which satisfies the equations

$$\begin{aligned} f(0, \vec{x}) &= g(\vec{x}) \\ f(y+1, x) &= h(f(y, \vec{x}), y, \vec{x}). \end{aligned} \tag{31}$$

where $\vec{x}$ is a vector of m variables $x_1, x_2, \ldots, x_m$. We write the primitives in standard ML as follows

```
val zero = 0
val succ = fn n => n+1
```

The projection functions are

```
val proj11 = fn x => x
val proj21 = fn x => fn _ => x
val proj22 = fn _ => fn x => x
```

and the others are similar. The Substitution operators are

```
fun subs11 g h = fn x => g (h x)
fun subs12 g h = fn x => fn y => g (h x y)
fun subs13 g h = fn x => fn y => fn z => g (h x y z)
fun subs21 g h1 h2 = fn x => g (h1 x) (h2 x)
fun subs22 g h1 h2 = fn x => fn y => g (h1 x y) (h2 x y)
fun subs23 g h1 h2 = fn x => fn y => fn z => g (h1 x y z) (h2 x y z)
```

and then we define the primitive recursion operators as a series of functors:

```
signature R1 =
sig
   type X
   val omega : int -> X
   val from : int -> (int * X) -> X
end

functor R1
   (type X
    val g : X
    val h : X -> int -> X) : R1 =
struct
   type X = X
   structure Numbers =
       PeanoNumbers(type X = int * X
                    val x = (0,g)
                    val y = fn (y,f) => (y+1, h f y))
   local open Numbers
         structure It = It(type X = int * X)
         open It
   in
      val from = fn n => fn z => #2 (omega n y z)
      val omega = fn n => from n x
   end
end
```

The others, `R2`, `R3` etc. are defined similarly, the only difference being the types of the final arguments of `g` and `h` which are `X -> X`, `X -> X -> X` etc. Using these we can define, e.g., addition as

```
structure Add =
  R2(type X = int
    val g = proj11
    val h = subs13 succ proj31)
```

The resulting function is the `omega` of the structure, so we define the primitive recursive function add using just `val add = Add.omega`.

Writing primitive recursive functions is not too difficult if one proceeds in an orderly fashion. For example, the recursive definition of div is

```
    div 0 n = 0
div (m+1) n = let r = div m n
              in if m + 1 = n * (r + 1) then r + 1 else r
              end
```

the recursion step of which one could read as saying "If $m+1$ is a whole multiple of n more than the result r of the previous recursion, then this result should be greater by one, otherwise it should be the same."

To represent this function in the language of primitive recursive functions, we require primitive recursive functions g and h satisfying

$$g(n) = 0$$
$$h(r,m,n) = \begin{cases} r+1 & \text{if } m+1 = n(r+1); \\ r & \text{otherwise.} \end{cases}$$

Then the operator `R2 g h` of primitive recursion constructs a function f which satisfies the equations (31). To construct suitable functions g and h to which we can apply the primitive recursion operator, we mechanically apply substitution to extract the arguments from the vector of applied values. This transformation is a process defined by structural recursion on the language of operators and arities as used in a lisp-like expression such as the following, which is equivalent to h as given above:

```
(cond (eq (succ m) (mul n (succ r))) (succ r) r)
```

The transformation of such recursion steps h into the language of primitive recursion consists in eliminating constants and the variables m, n_i and r by the following rule:

> If an operator of arity n is being applied in an m argument function, then prefix it with $\mathsf{subs}_{n,(m+1)}$ and replace all ocurrences of r, m or n_i by $\mathsf{proj}_{(m+1),1}$, $\mathsf{proj}_{(m+1),2}$ and $\mathsf{proj}_{(m+1),(2+i)}$ respectively. Replace constants by constant functions of arity $m+1$.

Thus the above expression for h, where $m = 2$, becomes the following variable-free expression using only primitives, substitution and other expressions so constructed:

```
(subs33 cond (subs23 eq (subs13 succ proj32)
                        (subs23 mul proj33 (subs13 succ proj31)))
             (subs13 succ proj31)
              proj31)
```

The transformation of the base case is given by the rule

> If, in an m argument function, an operator of arity n is being applied to one or more of the function's arguments, then prefix the operator with $\mathsf{subs}_{n,(m-1)}$ and replace instances of m_i by $\mathsf{proj}_{(m-1),i}$. Replace constants by constant functions of arity $m-1$.

Higher Order Primitive Recursive Functions

Because the iterator functor is a parametrically typed structure we can use primitive recursion at the level of functions, not just at the level of values as we have done here.

The class of functions computable in this way is large by any standards. The language called system **F,** described in Girard *et al.* [7] is based entirely on interpretation of inductive processes in a monad and it is capable of calculating any function which is provably total in second order Peano Arithmetic which has quantifiers ranging over predicates as well as over individuals. For example we can calculate hyper-exponential numbers such as the Ackermann numbers. We need to use the iterator at two different levels, for exactly the same reason that Church needs two applications of the type-raising operator to compute the predecessor function in [4].

```
structure Ackermann =
struct
   structure Alpha = It(type X = int)        (* Church type a'   *)
   structure Beta = It(type X = int -> int)  (* Church type a'a' *)
   local val O = 1
         val S = fn n => n + 1
   in
      val from = fn n => Beta.omega n
                    (fn f => fn n => Alpha.omega n f (f O)) S
      val omega = fn n => from n
   end
end
```

The `It` functor corresponds approximately to the iterator operator It which *isn't* defined for integers in system $\mathbf{F}$[8] because It is just the integers themselves: that is to say the elements of the type Int are each their own induction principle. The function `It` in the functor `It` would not be needed at all if Standard ML used Church numerals as its representation of integers.

`It` is in fact the only recursive definition we have used, and we can do without general recursion altogether, and this is a very important point, so let us make a little song and dance about it. Here is a function called `nrIt` which is extensionally identical to the function `It` but which does not use unbounded recursion. If we replace the single call to `It` in the functor of the same name by a call to `nrIt` then everything will still

[8] See [7] §11.5.1. p. 88.

work that worked before, except we will have a *guarantee* that no program will *ever* go into an infinite loop—bugs in the interpreter notwithstanding. Now the following code is *not* at all pretty, so Ladies, you may find yourselves wishing you had averted your eyes for a moment:

```
fun Acc maxbits1 maxbits2 maxbits3 n =
   let val make_acc =
          fn max => fn n =>
             fn f => fn x =>
                let val acc =
                       fn n => max (fn (e as (n,r)) =>
                           if n = 0 then e else (n-1,f r)) (n,x)
                    val (_,f) = acc n
                in
                   f
                end
       val two = fn f => fn x => f (f x)
       val allbits = maxbits1 two
       val intbits = allbits (fn n => n+1) 0
       val loop = make_acc (maxbits2 two) intbits
       val (nb,_,_) = loop (fn (r as (i,m,n))
                              => if m < n then (i+1,m*2,n)
                                          else r) (0,1,n)
       val bits = make_acc (maxbits3 two) nb
       val recursion = bits two
   in make_acc recursion n
   end

val bits = fn f => fn x => f (f (f (f (f (f x)))))
fun nrIt n x y = Acc bits bits bits n y x
```

The reason this works is that in many ML implementations, integers are a fixed finite number of bits. If they are not, then there is a practical limit on the size of any integer used as a loop counter for any but the most trivial of calculations: no machine we know of could make use of a loop counter with more than 64 bits, which is around 1.8×10^{19}. If the calculation going on inside the loop is non-trivial, then the overhead involved in setting up the required 'bits of recursion' is negligable. And if, heaven help you, you need more than 2^{64} iterations then all you need to do is change the final `x` in the definition of `bits` to read `(f x)`, then you will have a loop counter capable of 2^{128} iterations, which is quite a lot more, and if *that* isn't enough, then you may repeat the excercise to get 2^{256} iterations, but we passed the silly point at 64 bits.

Now the song and dance:

> *There is absolutely no computation whatsoever that one could* effectively *carry out on a machine with a finite state which could not be effectively carried out on a machine which computes only the total recursive functions. It follows that there are* no *material consequences whatsoever of using a programming language which is restricted to computing only the total recursive functions.*

So why did we ever bother with programming languages which are capable of 'expressing' undefined results? We really don't know. There are many, many reasons why restricting computing to only total functions is advantageous. Perhaps people thought that one day there would appear some computation which was practically useful and not expressible without the possibility of unbounded recursion. Some people seem to believe that the interpreter for the language itself is one such *wooden spoon.* It may be thought that it is easy enough to diagonalise the interpreter and derive a contradiction from assuming that it can interpret itself interpreting any program: one simply takes the interpreter and modifies it to add one to the result of the program it is interpreting, then one applies that version of the interpreter to itself. Many people think that this is a *contradiction,* so it *must* be impossible, but if they think again then they will find that they are wrong. One *may* write this program, and one may apply it to interpreting itself. It will run out of recursion, eventually, and halt. Where is the contradiction? It is in the partially evaluated result of the first-level interpretation of the interpreter. Although the outermost interpreter will have used up all available iterations, the program it was interpreting would not have reached the same stage, so the result would be an incomplete interpretation of the interpreter. If we were to continue that process of interpretation, by supplying more iterations, then that interpreter would run out of iterations at exactly the same point as the outer level did. *There is no wooden spoon:* the outermost interpreter and the interpreter it is interpreting have identical semantics both operationally and denotationally.

In all these examples we have abstracted away the entire induction process (i.e. the loop) and we are left with only the essential elements of the algorthms, taken one step at a time and with only the absolutely necessary information from any step being available to the next. Thus we destroy information in the system at the *maximum possible rate*, and this means that we increase the order in the system at the maximum possible rate.

This is a very efficient and reliable way to write programs. It is efficient because all the necessary state is contained in the elements of the abstract type X which represents values in the monad. Thus we could quite easily write a 'memoising' monad functor which detects when the monad stops producing new values. This happens whenever it returns the same result on two applications of the successor function to different numbers. Thereafter it can only ever produce values it produced between these two, because of the state of the algorithm being encapsulated entirely by the single element of type X. Therefore if the monad ever produces the same value at steps m and n with $n > m$ then it is in a 'finite loop' of length $m - n + 1$, so the elements x_i for $i > n$ need not be re-computed because thereafter the monad need only return successive elements from the loop in the cache. It is a reliable way to write programs because the mechanics of starting and stopping the loops are abstracted away, and these are the main source of programming errors. No well-typed calculation of a series represented as a monad in this way will ever go into an infinite loop.

The other major source of programming errors is unexpected interaction between bound variables. This is impossible to detect automatically in the presence of in-

ductive loop constructs because it is absolutely necessary that such interactions are allowed. Representation in a monad means that the state of the calculation is guaranteed *never* to be effected by *anything* that happens outside the definition of the parameters in that particular application of the functor. One further advantage which will perhaps prove to be even more significant is the increased facility we have for producing proofs of 'loop invariants' in a straightforward manner which ought to be amenable to automation. The resulting proofs would be intuitive and easy to understand and would give some extra insight into the operation of the algorithms. It is quite likely that they would be easily automatable. Take the example of the implementation of Euclid's algorithm. As there are only four linear paths through the code, two of which compute the identity function signifying the end of the computation, it is easy to see that each non-terminal step results in one of the pair of numbers being subtracted from the other. Thus in the worst case the largest of the two numbers is reduced by only two at each step, and therefore the algorithm takes of the order of $n/2$ steps to calculate the GCD of a pair m, n of numbers with $m < n$. So it seems feasible to contemplate a language interpreter which is capable of automatically calculating 'big O' space and time complexity metrics from algorithms.

There is also an entire field of pure mathematics called algorithmic number theory. In this one uses the inductive structure of algorithms as a source of information for proving properties of sets of numbers. One is able to do this much more easily once the details of the induction process have been abstracted away. There are interesting possibilities of using the properties of prime sieves to find conditions determining the distribution of prime numbers. The Riemann hypothesis concerning the zeros of the Riemann zeta function is really a problem about the distribution of the primes in the set $\mathbb{N}$ and whether or not it is the 'most even' distribution that is possible. One might learn a lot about this by looking at the interaction of prime seives with algorithms calculating the distribution of primes, especially in terms of the rate at which they lose information. One would be able to use group theory in this, because the sieve algorithms would induce characteristic subgroups of the symmetric group under the type isomorphism $\simeq$ of the inductive datatype.

The PDF file of this document includes the full source code for all of the primitve recursive functions mentioned. See reference [9], which explains how to extract the embedded file.

System F

We will now implement Girard's system **F** in Standard ML, using monads for parsing and interpreting the syntax. The reasons that this is worth doing are manifold. They include the fact that the book is obscurely written and desperately lacking intuitive interpretation. Presumably the student is expected to implement the language themselves to learn about it, so that is what we will do! Another reason however, is that system **F** seems to have been designed for interpreting Church's *Formulation*

of the Simple Theory of Types [4] and for exploring—and presumably solving—some of the problems posed in that paper; namely the typing of the predecessor function and the question of the independence[9] of the Axiom of Infinity (Ax. 8) from Ax. 1–7 and Theorem 30′. The reason this is worth doing in a monad is that *representation* is an essential part of the theory, in particular *self-representation* in the form of representations of extensions of system **F** in the pure system[10]. Another reason for implementing system *F* is that it will provide us with an opportunity to demonstrate an abstract implementation of capture-avoiding substitution. There are *three* distinct types of capture-avoiding substitution necessary in system **F**, one is in the polymorphic types, another is in the universal abstract types and the third instance is in the operational semantics of conversion. If we can handle all three of these instances with the same implementation then we will have demonstrated something of the utility of monads as interpretations of abstract algebras[11].

The system of uniform inductive types is central to the representation of other systems within **F**. It is derived from Martin-Löf's system of *operators and arities.* This beautifully simple system is so easy to understand that many people may give it very little thought: it seems trivial. The idea is that the only concrete objects are type operators of various *arities,* meaning the number of operands they have. So the system of operators and arities is a language of abstract operators of arity n, with variables ranging over those operators. New operators of arity n are constructed by composition of existing operators. Operators of arities $1, 2, 3, \ldots$ have straight-forward interpretations, but the so-called *nullary* operators—ones without any operands—may be considered *either* as being constants of a fixed type, or as being functions from the *unit* type **1** or $\langle\rangle$ to some undefined value distinct from any other in that type.

Interpretation without concrete representation

The science and mathematics of Aristotle is fundamentally empirical; the origin of all knowledge is Human experience of one kind or another. Aristotle distinguishes between two senses in which things may be said to be *evident.* These senses are those of the *empirically* evident, and the *logically* evident.

> *Since what is clear or logically more evident emerges from what in itself is confused but more observable by us, we must reconsider our results from this point of view. For it is not enough for a definitive formula to express as most now do the mere fact; it must include and exhibit the ground also. At present definitions are given in a form analogous to the conclusion of a syllogism;*

[9] See footnote, §6. p. 66.

[10] See Ch. 15. p. 119. In particular §. 15.2.4. p. 128.

[11] And *if* we succeed, then we will have *proved* that the *Barendregt convention* is sound, and sensible, and that there is nothing whatsoever about alpha-conversion that is profound: alpha-conversion does not have syntactical consequence, nor even any lexical characteristic; it is purely *typographical*, so no more significant than the type-face one uses, or the fact that an English speaker might pronounce $\lambda e.e$ the same way a Spanish speaker might pronounce $\lambda i.i$ or a child read *b* as *d* and *vice-versa*.

> *e.g. What is squaring? The construction of an equilateral rectangle equal to a given oblong rectangle. Such a definition is in form equivalent to a conclusion. One that tells us that squaring is the discovery of a line which is a mean proportional between the two unequal sides of the given rectangle discloses the ground of what is defined.* Aristotle. *On The Soul. II.2*

So although all our knowledge is derived in the first instance from the concrete foundation of experience: that of actual particular *things,* the knowledge which we formulate as a consequence, i.e. *formal logic,* starts, not from the experiential foundation of immediately evident truth, but from the *psychological* foundation of that which is more *logically* evident, and from there it proceeds towards the immediately evident or *verifiable* facts. So in Aristotle's philosophy of science, Mathematics *explains* physical experience, for example. It does this from the basis of the more logically evident truths, and is consequently able to explain empirical experience in terms of *formal logic*. Now this is something of a 'holy grail' for modern theoretical physics, so one might expect theoretical physicists to have an interest in Aristotelian formal logic, but that does not seem to be the case. The problem is presumably a psychological one of not having sufficient *faith* in the knowledge of what is more logically evident.

The relevance to representation in a monad is because a *signature* is an abstract algebra: it's a *structure,* but with formal placeholders (variables,) for the actual concrete elements. In other words, abstract algebras are *representations* of actual concrete constructions with a basis in the primitive or *nullary* constructions, such as that of zero in Peano Arithmetic, acted on by the *operators* of arities $1, 2, 3, \ldots$. Note the subtle switch from the usual sense of *representation* as it is used in model theory: here the *model* is the *more* abstract *signature* and it represents the *concrete* things. So in a signature we have a *syntactic* model of a structure. For example the structure

$$\langle X, \mathsf{O}{:}X, \mathsf{S}{:}X \to X \rangle \tag{32}$$

is a concrete mathematical object, i.e. it is a *particular aggregate,* consisting as it does of three more or less definite things: a *set* or a *type,* X; a constant O in X; and an operator S taking objects in X to other objects in X. Here X is a *metalanguage* variable signifying some *actual* set, which we do not care to specify. Thus, even though we represent the set by a placeholder, we intend it to be understood that the set is some definite set, and not an arbitrary choice. A *signature* such as

$$\Pi X.X \to (X \to X) \to X \tag{33}$$

on the other hand, where X is universally quantified, is indefinite where the corresponding structure is definite[12].

However, when one looks a little closer at the *definition*[13] of O and S:

$$\mathsf{O} \equiv \Lambda X.\lambda x^X.\lambda y^{X\to X}.x \quad \mathsf{S}\,t \equiv \Lambda X.\lambda x^X.\lambda y^{X\to X}.y\,(t\,X\,x\,y), \tag{34}$$

[12] See [7] §. 15.2. for a description of the ΛΠ notation
[13] See [7], §. 11.5.1. p. 88.

then it becomes apparent that in fact the structure (32) and the abstract signature (33) amount to much the same thing, because the 'concrete' structure $\langle X, \mathsf{O}, \mathsf{S}\rangle$ turns out to be nothing more than a series of bound variables, with one free variable, being the parameter t to the successor function S. The atomic constant O and the constants $\mathsf{S}\,t$ for any $t \in X$ (32) are in fact elements of the signature (33) in the sense that we have $\mathsf{O} \in X$ and $\forall t \in X.\mathsf{S}\,t \in X$. For example, the term $\mathsf{S}(\mathsf{S}\,\mathsf{O})$ is an abbreviation of

$$\Lambda X.\lambda x^X.\lambda y^{X\to X}.y\,((\Lambda X.\lambda x^X.\lambda y^{X\to X}.y\,((\Lambda X.\lambda x^X.\lambda y^{X\to X}.x)X\,x\,y))X\,x\,y)$$

which reduces to just $\Lambda X.\lambda x^X.\lambda y^{X\to X}.y\,(y\,x)$. Thus the signature (33) is exactly the `PeanoNumbers` monad and the set of closed terms $\omega = \{\mathsf{O}, \mathsf{S}\,\mathsf{O}, \mathsf{S}(\mathsf{S}\,\mathsf{O}), \ldots\}$ for some particular structure $\Sigma(X)$ are the terms `It.omega n` for integer $n \geq 0$ in some particular instance of the abstract structure `PeanoNumbers(type X val x: X val y:X -> X)`.

Indeed, if we let the structure in (32) *define* the type X in the traditional algebraic fashion, writing

$$X \simeq [\mathsf{O}{:}X, \mathsf{S}{:}X \to X] \tag{35}$$

then we have an analogy with the homomorphism $X \simeq T(X)$ characterising an inductive type X as the least fixed-point of the characteristic functor:

$$T(X) = \mathbf{1} + X. \tag{36}$$

Note also that the definition of O looks a lot more like the Greek *unit* than zero, because like the former, it may in fact be any number we can construct. Here is the definition of unit and numbers given in Euclid's *Elements* Book 7.

> α΄. Μονάς ἐστιν, καθ᾽ ἣν ἕκαστον τῶν ὄντων ἓν λέγεται.
> β΄. Ἀριθμὸς δὲ τὸ ἐκ μονάδων συγκείμενον πλῆθος.

The term *unit* or *monad* appears in the singular, μονάς, in the first definition and in the plural, μονάδων, in the second. In English:

> 1. A unit is that according to which each existing thing is called one.
> 2. And a number is a multitude composed of units.

In other words, we *may* choose any number we like as the unit, and a number is some multitude of those units. So we sometimes choose to count molecules in units of 2 or 6, or 6.2×10^{23}, or meters in units of 1,000 or of 9.467×10^{15}. But the multitude may be composed of units of incommensurable or even irrational numbers.

Now look again at the signature (33). This conforms to the general shape of such inductive types in system **F**. The 'rules' for inductive types are that the type variable X is universally quantified through being bound as a parameter by Λ appearing explicitly in the term, so it may be freely substitued for any other actual type U, say,

by *universal application* or *extraction*. But in the corresponding type expression, the binder $\Lambda X.Y$ appears as a second-order universal quantifier $\Pi X.Y$, quantifying over *type predicates* rather than *types* of individuals, as the first-order polymorphic $\forall$ quantifier does. The other rule[14] is that the inductive types T are of the form $T(X) = S_1 \to S_2 \to \cdots \to S_n \to X$ where the S_i are of the form $U_1 \to U_2 \to \cdots U_{k_i} \to X$, with X *appearing only positively* in U_j; that is to say, with U_i of the form $V_i \to X$ and so S_i possibly of the form $(V_1 \to X) \to (V_2 \to X) \to \cdots (V_{k_i} \to X) \to X$. Thus any inductive type $T(X)$ may be understood as an abstract context-free *grammar.* A suitable alphabet on which to model this abstract language is the formal language of operators and arities. Each symbol is a type operator, either *nullary,* or having a fixed number n of arguments, each of which are either nullary type operators, or are the results of other type operators having been recursively applied.

On this view, the process of *type checking* is analogous to parsing a sequence of symbols from the alphabet of that language. The process of *type inhabitation* is analogous to determining some sequence of symbols which is derivable as a production by a given grammar. Thus the process[15] of inferring a set f_i of type operators for a given inductive type signature of the form of (33) is the process of using *PROLOG* to prove atomic intuitionistic sequents of the form $\vdash B$ using the rule of cut and substitution instances of proper axioms.

These processes may all be represented in a monad. Recall the types of the monad operators `UnitM`, `MapM`, `JoinM` and `BindM`:

$$\texttt{UnitM}: \alpha \to \mu.\alpha \qquad \texttt{BindM}: \mu.\alpha \to (\alpha \to \mu.\beta) \to \mu.\beta$$
$$\texttt{MapM}: (\alpha \to \beta) \to \mu.\alpha \to \mu.\beta \qquad \texttt{JoinM}: \mu.\mu.\alpha \to \mu.\alpha$$

The operator `BindM` gives a way to extend the representation in the monad. Given any method $\mu.\alpha$ of computing values of type α, and a method $\alpha \to \mu.\beta$ of constructing methods $\mu.\beta$ of computing values of type β, we can construct a method of computing values of type β. The way we do this is by first representing the representation of β in the monad using the `MapM` operator, given $\alpha \to \mu.\beta$ as the first argument, and then extracting the representation $\mu.\beta$ from the representation $\mu.\mu.\beta$ using the `JoinM` operator. Indeed this is how we defined `BindM` in the functor `Monad` on page 53.

Using `BindM` to extend the types represented within a monad $\mu.\beta$ to those of values computed from a basis of computations $\mu.\alpha$ of types α corresponds to adding a production $\mu.\alpha \to \mu.\beta$ to the grammar as an *alternate* constructor for computations $\mu.\beta$ of values of the type β, thus adding a new possibility for the final step in the construction of derivations of terms of type β.

Now look at theorem VII on page 62 of Church's [4]. This describes the proof of the syntactical *Deduction Theorem.* The theorem asserts the validity of the derived rule

[14] [7] §. 11.4.1. p. 86.
[15] See §. 11.4.2. p. 87. in [7].

which looks a little like this:

$$\frac{A_o^1, A_o^2, \ldots, A_o^n \vdash B_o}{A_o^1, A_o^2, \ldots, A_o^{n-1} \vdash A_o^n \Rightarrow B_o} \tag{37}$$

Here and in what follows the subscripts are the Greek letter *omicron* indicating the type of propositional variables. The top line asserts the *existence* of a proof of B_o which is *hypothetical* on propositions $A_o^1, A_o^2, \ldots, A_o^n$, and the conclusion states the existence of a proof of $A_o^n \Rightarrow B_o$ which is hypothetical on the same propositions, excluding the final one which has been *discharged*.

Now such a proof of B_o would consist of a list of m formulæ $B_o^1, B_o^2, \ldots, B_o^m \equiv B_o$, each of which is either one of the A_o^i for $i = 0, 1, \ldots, m-1$, or A_o^m, or an axiom, or was derived from the immediately preceeding formula, or pair of immediately preceeding formulæ through the use of one of the rules of inference. This is therefore a tree, with B_o^m at the root and branches (or linear lists) leading upwards, toward the leaves which are either *hypotheses* from one of the parcels A_o^i of hypotheses which could eventually be discharged by the application of this deduction theorem, or it is a substitution instance of an axiom.

Then the proof of the soundness of the deduction theorem takes the form of a proof by induction on this series of theorems, and consists in demonstrating that for each $i = 1, 2, \ldots, m$ there exists a deduction of the form:

$$A_o^1, A_o^2, \ldots, A_o^{n-1} \vdash A_o^n \Rightarrow B_o^i \tag{38}$$

Now each B_o^i is a *node* in the tree, not a leaf. So the effect of the induction proof of the deduction theorem on the proof tree is the discharging of all of the parcels A_o^n, and we can easily see that the entire proof is one concrete construction in the form of an element of an inductive type of system **F** which is described[16] as a tree with branches, the nodes, which are either binary or unary type operators, being the functions $f_1, \ldots f_n$, and leaves which are either one of a parcel of hypotheses bound by some instance of Λ or they are axioms, which correspond to the 'atoms' $c_1, \ldots, c_k$.

For example, assume there exists a deduction $A_o^1, A_o^2 \vdash B_o$; then, to prove the deduction $A_o^1 \vdash A_o^2 \Rightarrow B_o$, given the proof of B_o with hypotheses $\Gamma = \{A_o^1, A_o^2\}$:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash A_o^1}{\Gamma \vdash B_o^{m-2}} \qquad \dfrac{\dfrac{\Gamma \vdash A_o^1}{\Gamma \vdash B_o^{m-4}} \qquad \dfrac{\Gamma \vdash A_o^2}{\Gamma \vdash B_o^{m-5}}}{\Gamma \vdash B_o^{m-3}}}{\Gamma \vdash B_o^{m-1}}}{\Gamma \vdash B_o^m \equiv B_o} \tag{39}$$

[16] Again, see [7], §. 11.4.1. p. 86.

Starting at the root, we discharge the A^1_o parcel of hypotheses *uniformly* in each and every conclusion, and thereby induce the conversion of the whole proof to:

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash [A^1_o]}{\Gamma_1 \vdash A^1_o \Rightarrow B^{m-2}_o} \quad \dfrac{\dfrac{\Gamma_1 \vdash [A^1_o]}{\Gamma_1 \vdash A^1_o \Rightarrow B^{m-4}_o} \quad \dfrac{\Gamma_1 \vdash A^2_o}{\Gamma_1 \vdash A^1_o \Rightarrow B^{m-5}_o}}{\Gamma_1 \vdash A^1_o \Rightarrow B^{m-3}_o}}{\Gamma_1 \vdash A^1_o \Rightarrow B^{m-1}_o}}{\Gamma_1 \vdash A^1_o \Rightarrow B_o} \tag{40}$$

Here the hypotheses are $\Gamma_1 = \{A^2_o\}$. The important point to notice however, is that the induction on the structure of the proof was actually an induction on the list of deductions treated as a linear sequence. So had we in fact had a different set of proof rules, making a different concrete structure, then essentially the same proof would have applied to the new structure, provided we had decided on a means of enumerating or *indexing* the types of the nodes of the tree which represents the abstract syntax of derivations in that proof system.

The tree structure is interesting to us as an operational view of the process of constructing a proof, but the tree structure is irrelevant to the inductive structure of the proof of the deduction theorem. So the fact that we may have chosen to represent the proof in the concrete form of an abstract datatype representing the proof tree is irrelevant to the process of induction on the proof itself. By considering just a list of theorems *in a certain order,* we have eliminated the necessity to incorporate any particular concrete data structure representing the proof inferences into the proof of the deduction theorem. The rules of deduction *are* material, but they are to do with the *meaning* of the proof, not the syntax. We need to use our *knowledge* of the structure of the rules of inference to understand *how* to produce the derivations which the deductive theorem claims exists, but we don't want concrete representations of instances of those rules to direct the construction of the derivations *syntactically* because then our proof procedure would be almost useless given a different concrete representation of the abstract data type of derivations, even if the abstract structure of the actual proof rules were the same.

Returning to the formal proof, we may now repeat the exercise, discharging the remaining parcel of hypotheses, this time without leaving any assumptions at all:

$$\dfrac{\dfrac{\dfrac{\vdash [A^1_o]}{\vdash A^2_o \Rightarrow A^1_o \Rightarrow B^{m-2}_o} \quad \dfrac{\dfrac{\vdash [A^1_o]}{\vdash A^2_o \Rightarrow A^1_o \Rightarrow B^{m-4}_o} \quad \dfrac{\vdash [A^2_o]}{\vdash A^2_o \Rightarrow A^1_o \Rightarrow B^{m-5}_o}}{\vdash A^2_o \Rightarrow A^1_o \Rightarrow B^{m-3}_o}}{\vdash A^2_o \Rightarrow A^1_o \Rightarrow B^{m-1}_o}}{\vdash A^2_o \Rightarrow A^1_o \Rightarrow B_o} \tag{41}$$

This establishes the theorem *ostensively,* thus completing the conversion.

This general form of proof has the form of a set of constructors for an inductive type, and the process Church describes of proving particular instances of the deduction theorem, given a proof in the form of an finite set of propositions $B_o^1, \ldots, B_o^m$ ordered as a derivation of B_o^m from the assumptions and axioms, has the structure of an induction principle for that type. What the algorithm is actually doing is analysing the proof as a linear sequence and inferring the inductive structure according to the rules about constructions being the result of one or two propositions appearing earlier in the proof. These are structural rules in the form of a *grammar.* So the deduction theorem can be thought of as parsing a finite string of deductions into a representation of a proof tree as ... well, just a finite set of deductions, in some total order.

We hope that we have explained what we mean by "interpretation without concrete representation," and made clear why it is a worthwhile thing to try to achieve. Instead of interpreting a structure by representing it in terms of concrete constructions, we interpret it by representing the *process* of translation instead.

We can be a little more clear about what we mean by 'representing the process of translation.' Of course the process of translation cannot usefully represent itself; but when one recalls the characteristic of an inductive datatype is an automorphism under which the characteristic subgroups of the symmetric group are set-wise preserved, one can think of those currents induced by the mapping of the elements of the invariant subgroups by the permutations of the automorphism. The *process* then is the mapping of elements or 'points,' each representing some distinct concrete object but only up to identity, so we do not interpret the points as having internal structure; we are concerned not with their attributes as individuals, but with the relation (an order) which holds between distinct individuals. Then the process of mapping these points one to the other according to their predecessor relation is the process which can be used to represent, say, the process of capture-avoiding substitution.

Consider the process of substitution whereby all the instances of some one particular parcel of hypotheses are replaced by another expression. Consider now a restricted class of lambda expressions, those constructed by an iterated process of expansion, starting with the term $\lambda x.x$, which is of course in normal form. The consequent beta- and eta-expansions will result (trivially) in the whole term being *solvable*, which means that there is some set of arguments to which one may apply the term to cause it to reduce to $\lambda x.x$. So every term will reduce to the identity function, and during this process of reduction each beta- or eta-reduction will be the inverse of a construction step: so reduction turns out to be deconstruction in this particular sense of the term, and this particular species of construction; the solvable terms. Then as we carry out substitutions of whole parcels of hypotheses by their corresponding arguments, we map each redex onto one of its possible predecessors in the order of construction. So the process of capture-avoiding substitution is *exactly* the process of the permutation of the elements of the characteristic subgroups of the relevent symmetry of the type automorphism.

Now if we *were* to capture one or more free variables in one of these substitution operations, then the result would not necessarily be in the same subgroup as the re-

sults of another alpha-equivalent contractum which did not capture free variables. The reason is that the characteristic permutation of the (chain of) binders immediately *within* the redex would have changed: one of the parcels would have grown in size by at least one, and that means that the class of the permutation in Dodgson's sense of *even* or *uneven,* could depend on the order in which pairs are removed from permutations, contradicting his *Theorem V.*

Now let's go back to the Deduction Theorem in Church's simple theory of types. It is presented as being about the finite series of propositions $B^1_o,\ldots,B^m_o$ which are supposed to constitute a proof of B_o. But the fact of the finitude of the list is the only thing that is relevant to the proof: the proof does not mention the actual length m except in so far as to use it as the name of the theorem being proved. Thus we could replace the assumption of an actual finite list of propositions with a process which searched for proofs using *iterative deepening* to extend the range of the search after each failure at the higher level. So if we had a set of rules of inference which were such that from the syntactic form of a proposition B^m_o we could decide which rule of inference was the last rule used in a deduction leading to a proof of $A^1_o,\ldots,A^{n-1}_o \vdash A^n_o \Rightarrow B_o$, then we could use iterative deepening to deduce the entire sequence $B^1_o,B^2_o,\ldots,B^{m-1}_o,A^n_o \Rightarrow B_o$ of a proof of $\Gamma \vdash A^n_o \Rightarrow B_o$ hypothetical on a set of propositions $\Gamma = \{A^1_o,\ldots,A^{n-1}_o\}$. The requisite property of the set of proof rules is that they are an initial T-algebra of some functor T which then characterises the symmetry of the datatype.

So the Deduction Theorem takes a sequence of hypothetical deductions $A^n_o \vdash B^1_o, A^n_o \vdash B^2_o,\ldots,A^n_o \vdash B^m_o$ leading to a hypothetical proof of B_o, and it produces a *parametric method* $\vdash A^n_o \Rightarrow B_o$ of proving B_o which, given an instance of an actual proof of the proposition A^n_o, will produce an actual proof of B_o. It does this by the use of the rule modus ponens, substituting every instance of A^n_o by the given proof. Therefore the deduction theorem itself is a method which takes a *method* of proving B_o, hypothetical on being given a method of proving A^n_o, and produces a parametric method of proving B_o which needs an actual proof of A^n_o and simply substitutes all ocurrences of the proposition A^n_o with the entire tree representing the proof of A^n_o.

So the methods[17] given by Girard *et al* of finding the constructors and deconstructors of a given inductive datatype T are in fact a generalisation of Church's Deduction Theorem to an arbitratry proof system. The only conditions on the proof system are that it is representable by an initial T-algebra of some functor T characterising the type structure. Furthermore, the method produces descriptions of the constructors and deconstructors of the datatype in the elementary language of operators and arities.

Now the form of the rule (37) derived from the Deduction Theorem is actually more than is shown there explicitly because the rule in fact takes a sequence of antecedents,

[17] [7], §. 11.4.2 and 11.4.3 p. 87.

one for each of the $i = 1, 2, \ldots, m$ steps $A^1_o, A^2_o, \ldots, A^n_o \vdash B^i_o$ in the proof:

$$\frac{\begin{array}{c} A^1_o, A^2_o, \ldots, A^n_o \vdash B^1_o \\ A^1_o, A^2_o, \ldots, A^n_o \vdash B^2_o \\ \vdots \\ A^1_o, A^2_o, \ldots, A^n_o \vdash B^m_o \end{array}}{A^1_o, A^2_o, \ldots, A^{n-1}_o \vdash A^n_o \Rightarrow B_o} \tag{42}$$

and, except for the first step in the chain, these hypotheses may themselves be hypothetical on one or two of the preceeding hypotheses. So recursively elaborating these structural dependencies using the deduction theorem yields the full tree tree structure of the derivation. This general structure of proof is very precisely described in Martin-Löf's *On the Meanings of the Logical Constants and the Justifications of the Logical Laws* where he defines the structure of hypothetical judgements[18]. And it happens also that the notion of the form of judgement $\ulcorner A$ prop.$\urcorner$ used there also has a very precise interpretation in terms of Church's formulation [4]. It corresponds to the possession of the interpretation given of the terms presented within square brackets in the abbreviations on page 58. For example we have, amongst others,

$$\begin{aligned} [\neg \boldsymbol{A}_o] &\rightarrow N_{oo}\, \boldsymbol{A}_o \\ [\boldsymbol{A}_o \vee \boldsymbol{B}_o] &\rightarrow A_{ooo}\, \boldsymbol{A}_o\, \boldsymbol{B}_o \\ [\boldsymbol{A}_o \supset \boldsymbol{B}_o] &\rightarrow [[\neg \boldsymbol{A}_o] \vee \boldsymbol{B}_o] \\ [(\boldsymbol{x}_\alpha)\boldsymbol{A}_o] &\rightarrow \Pi_{o(o\alpha)}(\lambda \boldsymbol{x}_\alpha \boldsymbol{A}_o) \end{aligned}$$

defining negation, alternation, implication and higher order universal quantification respectively. The bold italic uppercase letters are metalanguage variables and they represent whole formulæ at the given type, that of propositions in this case. Within square brackets, these variables are assigned values by *pattern-matching* with formulæ in the object language and they correspond to any well-formed formula of the theory. Thus a judgement of the form $\ulcorner[\boldsymbol{A}_o \supset \boldsymbol{B}_o]$ prop.$\urcorner$ entails one of the form $\ulcorner[[\neg \boldsymbol{A}_o] \vee \boldsymbol{B}_o]$ prop.$\urcorner$ and this entails knowing $\ulcorner[\neg \boldsymbol{A}_o]$ prop.$\urcorner$ and $\ulcorner \boldsymbol{B}_o$ prop.$\urcorner$ the first of these entailing $\ulcorner \boldsymbol{A}_o$ prop.$\urcorner$. These judgements are all made during the process of parsing the expressions represented by the metalanguage variables.

Now any deduction B^1_o has the form of a *Horn clause* and constitutes a rule that could form the basis of a *PROLOG* search for a deduction. The form of the initial deduction B^1_o is always that of a substitution instance of one of the axioms, with one or more of the formulæ A^i_o substituted for propositional variables in the axiom. Other forms of propositions B^i_o which appear in proofs may depend on one or two antecedents B^j_o with $j < i$, and these are proved by the use of inference rules of the system applied to theorems already proved. So in representing a proof consisting of a sequence of m propositions B^i_o with $i = 1, 2, \ldots, m$, at each step we need only

[18] See [12], Third lecture, around p. 31, though the page numbers are not marked.

record the formula, the last rule of inference used in its derivation, and the length of the proof, which is just the number i of that proposition in the sequence. A search amongst the theorems proved in less than i inferences is certain to yield one formula B_o^j of which the required propostion is a substitution instance. We do not need to record the hypotheses because the only formulæ which are relevant to the hypothetical proof represented by any particular formula B_o^i are those which are free variables in B_o^i. Therefore one proof may refer to a sub-proof of a given proposition as many times as necessary, each time either as the same or a different substitution instance, and the proof search will be able to find every instance required amongst the formulæ already proved. It can do this by pattern-matching on the forms of the conclusions, which is effected by the unification algorithm.

To Be Continued ...

Colophon

'You are sad,' the Knight said in an anxious tone: 'let me sing you a song to comfort you.'

'Is it very long?' Alice asked, for she had heard a good deal of poetry that day.

'It's long,' said the Knight, 'but very, *very* beautiful. Everybody that hears me sing it—either it brings the *tears* into their eyes, or else—'

'Or else what?' said Alice, for the Knight had made a sudden pause.

'Or else it doesn't, you know. The name of the song is called *"Haddock's Eyes."*'

'Oh, that's the name of the song, is it?' Alice said, trying to feel interested.

'No, you don't understand,' the Knight said, looking a little vexed. 'That's what the name is *called.* The name really *is "The Aged Aged Man."*'

'Then I ought to have said "That's what the *song* is called?"' Alice corrected herself.

'No, you oughtn't: that's quite another thing! The *song* is called *"Ways And Means"*: but that's only what it's *called,* you know!'

'Well, what *is* the song, then?' said Alice, who was by this time completely bewildered.

'I was coming to that,' the Knight said. 'The song really *is "A-sitting on a Gate"*: and the tune's my own invention.'

So saying, he stopped his horse and let the reins fall on its neck: then, slowly beating time with one hand, and with a faint smile lighting up his gentle foolish face, as if he enjoyed the music of his song, he began.

Of all the strange things that Alice saw in her journey Through The Looking-Glass, this was the one that she always remembered most clearly. Years afterwards she could bring the whole scene back again, as if it had been only yesterday—the mild blue eyes and kindly smile of the Knight—the setting sun gleaming through his hair, and shining on his armour in a blaze of light that quite dazzled her—the horse quietly moving about, with the reins hanging loose on his neck, cropping the grass at her feet—and the black shadows of the forest behind—all this she took in like a picture, as, with one hand shading her eyes, she leant against a tree, watching the strange pair, and listening, in a half dream, to the melancholy music of the song.

'But the tune *isn't* his own invention,' she said to herself: 'it's *"I give thee all, I can no more."*' She stood and listened very attentively, but no tears came into her eyes.

'I'll tell thee everything I can;
There's little to relate.
I saw an aged aged man,
A-sitting on a gate.
"Who are you, aged man?" I said,
"and how is it you live?"
And his answer trickled through my head
Like water through a sieve.

He said "I look for butterflies
That sleep among the wheat:
I make them into mutton-pies,
And sell them in the street.

I sell them unto men," he said,
"Who sail on stormy seas;
And that's the way I get my bread—
A trifle, if you please."

But I was thinking of a plan
To dye one's whiskers green,
And always use so large a fan
That they could not be seen.
So, having no reply to give
To what the old man said,
I cried, "Come, tell me how you live!"
And thumped him on the head.

His accents mild took up the tale:
He said "I go my ways,
And when I find a mountain-rill,
I set it in a blaze;
And thence they make a stuff they call
Rolands' Macassar Oil—
Yet twopence-halfpenny is all
They give me for my toil."

But I was thinking of a way
To feed oneself on batter,
And so go on from day to day
Getting a little fatter.
I shook him well from side to side,
Until his face was blue:
"Come, tell me how you live," I cried,
"And what it is you do!"

He said "I hunt for haddocks' eyes
Among the heather bright,
And work them into waistcoat-buttons
In the silent night.
And these I do not sell for gold
Or coin of silvery shine
But for a copper halfpenny,
And that will purchase nine.

"I sometimes dig for buttered rolls,
Or set limed twigs for crabs;
I sometimes search the grassy knolls
For wheels of Hansom-cabs.
And that's the way" (he gave a wink)
"By which I get my wealth—
And very gladly will I drink
Your Honour's noble health."

I heard him then, for I had just
Completed my design
To keep the Menai bridge from rust

By boiling it in wine.
I thanked him much for telling me
The way he got his wealth,
But chiefly for his wish that he
Might drink my noble health.

And now, if e'er by chance I put
My fingers into glue
Or madly squeeze a right-hand foot
Into a left-hand shoe,
Or if I drop upon my toe
A very heavy weight,
I weep, for it reminds me so,
Of that old man I used to know—

Whose look was mild, whose speech was slow,
Whose hair was whiter than the snow,
Whose face was very like a crow,
With eyes, like cinders, all aglow,
Who seemed distracted with his woe,
Who rocked his body to and fro,
And muttered mumblingly and low,
As if his mouth were full of dough,
Who snorted like a buffalo—
That summer evening, long ago,
A-sitting on a gate.'

As the Knight sang the last words of the ballad, he gathered up the reins, and turned his horse's head along the road by which they had come. 'You've only a few yards to go,' he said, 'down the hill and over that little brook, and then you'll be a Queen—But you'll stay and see me off first?' he added as Alice turned with an eager look in the direction to which he pointed. 'I shan't be long. You'll wait and wave your handkerchief when I get to that turn in the road? I think it'll encourage me, you see.'

'Of course I'll wait,' said Alice: 'and thank you very much for coming so far—and for the song—I liked it very much.'

'I hope so,' the Knight said doubtfully: 'but you didn't cry so much as I thought you would.'

So they shook hands, and then the Knight rode slowly away into the forest. 'It won't take long to see him *off*, I expect,' Alice said to herself, as she stood watching him. 'There he goes! Right on his head as usual! However, he gets on again pretty easily—that comes of having so many things hung round the horse—' So she went on talking to herself, as she watched the horse walking leisurely along the road, and the Knight tumbling off, first on one side and then on the other. After the fourth or fifth tumble he reached the turn, and then she waved her handkerchief to him, and waited till he was out of sight.

'I hope it encouraged him,' she said, as she turned to run down the hill: 'and now for the last brook, and to be a Queen! How grand it sounds!' A very few steps brought her to the edge of the brook. 'The Eighth Square at last!' she cried as she bounded across, . . .

Apologia

Some people have quiestioned our motivation. It is simply this:

> *If, then, God is always in that good state in which we sometimes are, this compels our wonder; and if in a better this compels it yet more. And God is in a better state. And life also belongs to God; for the actuality of thought is life, and God is that actuality; and God's self-dependent actuality is life most good and eternal. We say therefore that God is a living being, eternal, most good, so that life and duration continuous and eternal belong to God; for this is God.* Aristotle. *Metaphysics.* XII.7

We do not merely *profess belief* in the living God, because we in fact have *actual knowledge* of God. This is not mystical knowledge, it is something which any and every person could and *should* have because it is *scientific knowledge* reached *solely* through reasoning about the causes of common Human experience. But to be able to reason scientifically one must be able to reason about possibility on the basis of actual knowledge. This cannot be deferred to a symbolic calculation because it *necessarily* requires making the distinction between *presentation* and *representation*, which requires actual knowledge rather than represented knowledge.

It is vitally important that all people are able to know God. But as it is currently widely understood, modern scientific thinking is *irrational*, so by educating people in this way, we are *preventing them* from knowing God, and that is *not* good for any one. Therefore we must change the way we use formal logic so that we may use it to reason logically about scientific knowledge, and thereby come to a universally *shared* scientific understanding of the nature of God.

Appendix

The non-intuitionistic `effectself` combinator is the lambda expression $\mathsf{E} \equiv \lambda r = \mathsf{r}.\lambda e = \mathsf{e}.\lambda p = \mathsf{p}.\lambda t.r\ (p\ t)$ where $\mathsf{r} \equiv \lambda t.t\ (mu\ t)$, $\mathsf{e} \equiv \lambda t.\lambda r.t\ (um\ r\ r)$ and $\mathsf{p} \equiv \lambda t.\lambda r.\lambda v.e\ t\ r\ v$. Putting

$$\Gamma \equiv \{mu{:}\forall\alpha.((\alpha)\,\mathsf{mu} \to \alpha) \to (\alpha)\,\mathsf{mu}, um{:}\forall\alpha.(\alpha)\,\mathsf{mu} \to ((\alpha)\,\mathsf{mu} \to \alpha)\},$$

we have the type derivation:

$\Gamma_1 \vdash t{:}\alpha\ \mathsf{mu} \to \alpha$	1	ASSUM: 6
$\Gamma_1 \vdash mu{:}(\alpha\ \mathsf{mu} \to \alpha) \to \alpha\ \mathsf{mu}$	2	ITAUT: $mu{:}[\alpha/\alpha]$
$\Gamma_1 \vdash t{:}\alpha\ \mathsf{mu} \to \alpha$	3	ASSUM: 6
$\Gamma_1 \vdash mu\ t{:}\alpha\ \mathsf{mu}$	4	COMB: 2,3
$\Gamma_1 \vdash t\ (mu\ t){:}\alpha$	5	COMB: 1,4
$\Gamma \vdash \lambda t.t\ (mu\ t){:}(\alpha\ \mathsf{mu} \to \alpha) \to \alpha$	6	ABS: 5
$\Gamma \vdash \lambda t.t\ (mu\ t){:}\forall\alpha.(\alpha\ \mathsf{mu} \to \alpha) \to \alpha$	7	GEN: 6
$\Gamma_2 \vdash t{:}\alpha \to \beta$	8	ASSUM: 16
$\Gamma_2 \vdash um{:}\alpha\ \mathsf{mu} \to \alpha\ \mathsf{mu} \to \alpha$	9	ITAUT: $um{:}[\alpha/\alpha]$
$\Gamma_2 \vdash r{:}\alpha\ \mathsf{mu}$	10	ASSUM: 15
$\Gamma_2 \vdash um\ r{:}\alpha\ \mathsf{mu} \to \alpha$	11	COMB: 9,10
$\Gamma_2 \vdash r{:}\alpha\ \mathsf{mu}$	12	ASSUM: 15
$\Gamma_2 \vdash um\ r\ r{:}\alpha$	13	COMB: 11,12
$\Gamma_2 \vdash t\ (um\ r\ r){:}\beta$	14	COMB: 8,13
$\Gamma_1 \vdash \lambda r.t\ (um\ r\ r){:}\alpha\ \mathsf{mu} \to \beta$	15	ABS: 14
$\Gamma \vdash \lambda t.\lambda r.t\ (um\ r\ r){:}(\alpha \to \beta) \to \alpha\ \mathsf{mu} \to \beta$	16	ABS: 15
$\Gamma \vdash \lambda t.\lambda r.t\ (um\ r\ r){:}\forall\beta.(\alpha \to \beta) \to \alpha\ \mathsf{mu} \to \beta$	17	GEN: 16
$\Gamma \vdash \lambda t.\lambda r.t\ (um\ r\ r){:}\forall\alpha.\forall\beta.(\alpha \to \beta) \to \alpha\ \mathsf{mu} \to \beta$	18	GEN: 17
$\Gamma_3 \vdash e{:}(\beta \to \alpha \to \gamma) \to \beta\ \mathsf{mu} \to \alpha \to \gamma$	19	ITAUT: 39 $e{:}[\beta/\alpha, \alpha \to \gamma/\beta]$
$\Gamma_3 \vdash t{:}\beta \to \alpha \to \gamma$	20	ASSUM: 28
$\Gamma_3 \vdash e\ t{:}\beta\ \mathsf{mu} \to \alpha \to \gamma$	21	COMB: 19,20
$\Gamma_3 \vdash r{:}\beta\ \mathsf{mu}$	22	ASSUM: 27
$\Gamma_3 \vdash e\ t\ r{:}\alpha \to \gamma$	23	COMB: 21,22
$\Gamma_3 \vdash v{:}\alpha$	24	ASSUM: 26
$\Gamma_3 \vdash e\ t\ r\ v{:}\gamma$	25	COMB: 23,24
$\Gamma_2 \vdash \lambda v.e\ t\ r\ v{:}\alpha \to \gamma$	26	ABS: 25
$\Gamma_1 \vdash \lambda r.\lambda v.e\ t\ r\ v{:}\beta\ \mathsf{mu} \to \alpha \to \gamma$	27	ABS: 26
$\Gamma \vdash \lambda t.\lambda r.\lambda v.e\ t\ r\ v{:}(\beta \to \alpha \to \gamma) \to \beta\ \mathsf{mu} \to \alpha \to \gamma$	28	ABS: 27
$\Gamma \vdash \lambda t.\lambda r.\lambda v.e\ t\ r\ v{:}\forall\gamma.(\beta \to \alpha \to \gamma) \to \beta\ \mathsf{mu} \to \alpha \to \gamma$	29	GEN: 28
$\Gamma \vdash \lambda t.\lambda r.\lambda v.e\ t\ r\ v{:}\forall\alpha.\forall\gamma.(\beta \to \alpha \to \gamma) \to \beta\ \mathsf{mu} \to \alpha \to \gamma$	30	GEN: 29
$\Gamma \vdash \lambda t.\lambda r.\lambda v.e\ t\ r\ v{:}\forall\beta.\forall\alpha.\forall\gamma.(\beta \to \alpha \to \gamma) \to \beta\ \mathsf{mu} \to \alpha \to \gamma$	31	GEN: 30
$\Gamma_1 \vdash r{:}((\alpha \to \beta)\,\mathsf{mu} \to \alpha \to \beta) \to \alpha \to \beta$	32	ITAUT: 40 $r{:}[\alpha \to \beta/\beta]$
$\Gamma_1 \vdash p{:}((\alpha \to \beta) \to \alpha \to \beta) \to (\alpha \to \beta)\,\mathsf{mu} \to \alpha \to \beta$	33	ITAUT: 38 $p{:}[\alpha/\beta, \alpha \to \beta/\alpha, \beta/\gamma]$
$\Gamma_1 \vdash t{:}(\alpha \to \beta) \to \alpha \to \beta$	34	ASSUM: 37
$\Gamma_1 \vdash p\ t{:}(\alpha \to \beta)\,\mathsf{mu} \to \alpha \to \beta$	35	COMB: 33,34
$\Gamma_1 \vdash r\ (p\ t){:}\alpha \to \beta$	36	COMB: 32,35
$\Gamma \vdash \lambda t.r\ (p\ t){:}((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta$	37	ABS: 36
$\Gamma \vdash \lambda p = \lambda t.\lambda r.\lambda v.e\ t\ r\ v.\lambda t.r\ (p\ t){:}((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta$	38	LET: 31,37
$\Gamma \vdash \lambda e = \lambda t.\lambda r.t\ (um\ r\ r).\lambda p = \mathsf{p}.\lambda t.r\ (p\ t){:}((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta$	39	LET: 18,38
$\Gamma \vdash \lambda r = \lambda t.t\ (mu\ t).\lambda e = \mathsf{e}.\lambda p = \mathsf{p}.\lambda t.r\ (p\ t){:}((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta$	40	LET: 7,39
$\Gamma \vdash \lambda r = \mathsf{r}.\lambda e = \mathsf{e}.\lambda p = \mathsf{p}.\lambda t.r\ (p\ t){:}\forall\beta.((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta$	41	GEN: 40
$\Gamma \vdash \lambda r = \mathsf{r}.\lambda e = \mathsf{e}.\lambda p = \mathsf{p}.\lambda t.r\ (p\ t){:}\forall\alpha.\forall\beta.((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta$	42	GEN: 41

References

[1] Aristotle. *The Works*. http://ian-grant.net/aristotle/complete.pdf, 2011.

[2] Lewis Carroll. *The Game of Logic*. Macmillan & Co., London & New York, 1886. http://archive.org/details/gameoflogic00carrrich.

[3] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, 1936. http://www.jstor.org/stable/2371045.

[4] Alonzo Church. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5(2):56–68, June 1940. http://www.jstor.org/stable/2266170.

[5] Charles L. Dodgson. *An Elementary Treatise on Determinants with their Application to Simultaneous Linear Equations and Algebraical Geometry*. Macmillan & Co., London, 1867. http://archive.org/details/anelementarytre03carrgoog.

[6] Solomon Feferman, editor. *Kurt Gödel: Collected Works Vol. I*, Chapter: Über formal unentscheidbare Sätze der *Principia Mathematica* und verwandter Systeme I, pages 144–195. Oxford University Press, 1986.

[7] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, Cambridge, 1989. http://www.paultaylor.eu/stable/prot.pdf.

[8] Michael J. C. Gordon. From LCF to HOL: a short history. http://www.cl.cam.ac.uk/~mjcg, October 1996.

[9] Ian Grant. The Hindley-Milner Type Inference Algorithm. *Unpublished manuscript*, January 2011. The Standard ML code `source.tar.gz` for the Hindley-Milner type inference with derivations is embedded in this PDF document as a PDF object, number 588 which can be extracted by using the command `pdftosrc`, which is part of pdfTEX. The description of the algorithm and an earlier version of the code are available from http://ian-grant.net/hm.

[10] John R. Harrison. *Introduction to Functional Programming, Lecture Notes*. Computer Laboratory, University of Cambridge, 1997. http://www.cl.cam.ac.uk/teaching/Lectures/funprog-jrh-1996/all.ps.gz.

[11] Luis Damas and Robin Milner. Principal Type-schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 202–212. POPL, ACM, 1982. http://ian-grant.net/hm/milner-damas.pdf.

[12] Per Martin-Löf. On the Meanings of the Logical Constants and the Justifications of the Logical Laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996. http://docenti.lett.unisi.it/files/4/1/1/6/martinlof4.pdf.

[13] Lawrence C. Paulson. *Foundations of Functional Programming, Part IB Computer Science Tripos Lecture Notes*. Computer Laboratory, University of Cambridge, 2000. http://www.cl.cam.ac.uk/~lp15/papers/Notes/Founds-FP.pdf.

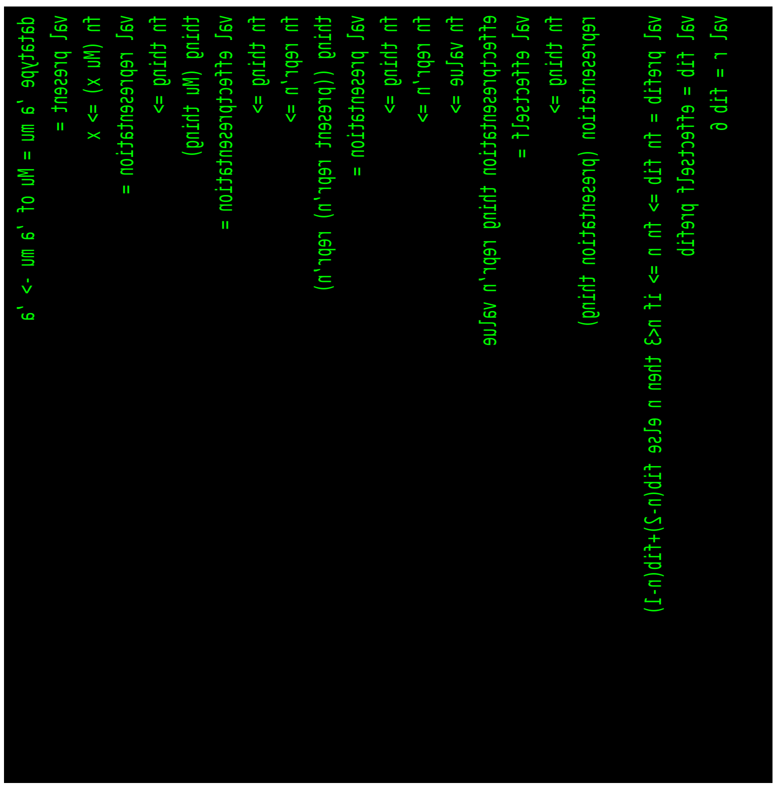
[14] Willard V. Quine. Variables Explained Away. *Proceedings of the American Philosophical Society*, 104(3):343–347, June 1960. http://www.jstor.org/stable/985250.

[15] Jan Rutten and Bart Jacobs. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin*, 62:222–259, 1997. http://www.cs.ru.nl/~bart/PAPERS/JR.pdf.

[16] William W. Tait. Intensional Interpretation of Functionals of Finite Type I. *Journal of Symbolic Logic*, 32:198–212, 1967.

[17] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936-37. http://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf.

[18] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem: A Correction. *Proceedings of the London Mathematical Society*, 43:544–546, 1937.



This is a communication (out of band) from the (deep) *Scheme Underground*.