Abstraction, Structure, and Substitution: Lambda and its Philosophical Significance

Peter Simons
University of Leeds

Abstract. λ-calculi are of interest to logicians and computer scientists but have largely escaped philosophical commentary, perhaps because they appear narrowly technical or uncontroversial or both. I argue that even within logic λ-expressions need to be understood correctly, as functors signifying functions in intension within a categorial or typed language. λ-expressions are not names but pure variable binders generating functors, and as such they are of use in giving explicit definitions. But λ is applicable outside logic and computer science, anywhere where the notions of complex whole, substitution, abstraction and structure make sense. To illustrate this, two domains are considered. One is somewhat frivolous: the study of flags; the other is very serious: manufacturing engineering. In each case we can employ λ-abstraction to describe substitutions within a structure, and in the latter case there is even a practical need for such a notation.

1. Introduction: Some History

The calculus of λ-conversion, an ingenious invention of Alonzo Church (Church 1932, 1940, 1941), is one of the most impressive logical innovations of the twentieth century. It enabled Church to study recursion and solve the *Entscheidungsproblem*, provided a smooth way to formalize simple type theory, and served as the prototype for all functional or applicative programming languages, not least the first such language, John McCarthy's LISP. The logic of λ-conversion has been thoroughly and admirably investigated, and it is a thriving subject on the borderlines between mathematics, logic, and computer science. Nevertheless, *philosophers* have been remarkably reluctant to take account of λ and its possibilities, whether because they believe it is too specialized and technical to be of interest, or because they consider it essentially uncontroversial. My purpose in this essay is twofold: to show that λ, far from being straightforward and uncontroversial, needs care if it is to be understood properly, and secondly, to demonstrate that far from being narrowly technical and confined to logic and computer science, it has significance well beyond these confines, and is applicable in *any* circumstances in which it is appropriate to talk about substitution, abstraction and structure.

Because Church's model in logic was Frege, and because Frege built logic on the basis of the concept of function, Church introduced λ

in connection with the logical theory of functions, and, like Frege, understood logic largely in terms of functions. λ enabled the notion of recursivity of functions to be exactly defined. Hence it was natural for Church to say that a λ-expression signifies a function. In Church's formalization of the calculus, there are two operations: the application of a function f to an argument a, represented linguistically by $f(a)$, or fa or (fa), and the abstraction of a function from a context, the latter represented linguistically by an expression called the *matrix*. If M is the matrix, and x is a variable, then $\lambda x.M$ represents the function of one variable x whose value is obtained for any value of x by substituting an expression designating that value in the context M and evaluating the result. So for example the function $\lambda x.x^2 + 2x - 1$ when evaluated for the argument $x = 5$, yields or "returns" the value 34. This leads to the calculus's most important rule of β-conversion:

$$\lambda x.M(a) = M[a/x]$$

where the expression on the right indicates the result of uniformly substituting all free occurrences of 'x' by occurrences of 'a'. So to take our example,

$$\lambda x.x^2 + 2x - 1(5) = 5^2 + 2 * 5 - 1$$

which expression denotes the number 34.

There are a number of sophisticated variations in different forms of the calculus, which it is not my intention to pursue here. For details, one may refer to the encyclopedic texts of Barendregt (1984, 1992).

Before I move on, I wish to draw attention to antecedents of the idea and calculus. Church's obvious forebear was Frege, whose logic is based on a generalization of the notion of function. While Frege did not himself develop a general notation for functional abstraction, he did develop a powerful notation for combined functional abstraction and application, which he used to define the notion of the ancestral of a function in 1879. Frege's innovation was later dropped and has hardly ever been noticed (Simons 1988). Russell was close to the idea of general abstraction in 1903, stressing the importance of the particle 'such that' (Russell 1903, sect. 23). Russell uses the particle here only to abstract classes, whereas in *Principia mathematica* he and Whitehead used '$\phi\hat{z}$' as a notation for a function (as distinct from an ambiguous value of it), but the notation was never systematically exploited(Whitehead, Russell 1910, p. 40). However, Kevin Klement has shown that in unpublished writings of 1903-05 Russell anticipated the λ-calculus more directly and fully, before dropping functions altogether (Klement, 2003). Perhaps the most significant forebear apart from Frege was Bolzano, who in his *Wissenschaftslehre* of 1837 made considerable use of the idea of

variable parts of a proposition, but again did not systematize the idea notationally.

2. Typed vs Untyped λ-Calculi

The most significant division in λ-calculi is between typed and untyped calculi. In untyped calculi, no distinction is made between different types of entities, so if M and N are any two expressions for any two entities, the expression MN is meaningful and signifies functional application of the former to the latter. Church originally developed λ-calculus in the untyped version as part of a foundation of mathematics (Church 1932), but the whole turned out to be inconsistent so he developed a weaker, consistent untyped λ-calculus and most development has been of this version. Curry, who as inventor of combinatory logic performed a feat provably equivalent to that of Church, makes no bones about his view that the untyped version is superior:

> In combinatory logic we must make, in order to achieve the objectives already mentioned, the following demands: (a) there shall be no distinctions between different categories of entities, hence any construct formed from the primitive entities by means of the allowed operations must be significant in the sense that it is admissible as an entity; (b) there shall be an operation consisting to application of a function to an argument; (c) there shall be equality with the usual properties; and (d) the system shall be *functionally complete*, i.e. such that any function we can define intuitively by means of a variable can be represented formally as an entity of the system. [...]
>
> A system will be called completely formalized just when it contains no auxiliary and no restrictions on the applicatibility of its functives. There will be only one category of obs; the closure of an n-ary operator with respect to any n obs will always be an ob; and that of an n-ary predicate with respect to any n obs will always be an elementary statement. [...]
>
> Examples of incompletely formalized systems would be: (1) a system whose morphology divides the obs into various "types", (2) a system with a rule of substitution, as this rule has to specify a peculiar class of obs upon which the substitution may be performed (Curry, Feys 1958, pp. 4–5, 32–33).

Curry's aims are admirably general, as comes out in the opening of his book:

> Combinatory logic is a branch of mathematical logic which concerns itself with the ultimate foundations. Its purpose is the analysis of certain notions of such basic character that they are ordinarily taken for granted. They include the analysis of substitution, usually indicated by the use of variables; and also the classification of the entities constructed by

> these processes into types of categories. [...] The second question is the explanation of the paradoxes (Curry, Feys 1958, p. 1).

The latter aim is not assisted by the division of entities into types, since these block the paradoxes without diagnosing how and why they occur (except that they "ignore types", which is not very promising as an explanation). But Curry's ideal of a typeless universe, even in mathematics, appears unachievable. Even Frege allowed that there is an ontological gulf between objects like the number 2 and functions like the square function. Such differences can only be overcome by ignoring how mathematics actually works, or somehow artificially constraining things. Church is more circumspect: while not dividing entities into types, he allows that functions may have a range of significant application:

> for each function there is a class, or range, of possible arguments—the class of things to which the operation is significantly applicable [...] If f denotes a particular function, we shall use the notation $(f\alpha)$ for the value of the function f for the argument α. If α does not belong to the range of arguments of f, the notation $(f\alpha)$ shall be meaningless (Church 1941, p. 1).

Thus while it makes sense to apply the differentiation operator to the sine function, and the sine function to the number π, it does not makes sense to apply the sine function to the differentiation operator, or the latter to π, or indeed π to either. This is not a matter of mere convenience or expediency or hidebound convention: it is deeply rooted in the logical order of things, as Frege recognized. Church's expedient of allowing $(f\alpha)$ to be "meaningless" is actually a fudge, designed to absolve him of the need to introduce types and type restrictions. He simply throws such "meaningless" expressions away, or rather, never gets round to making use of them, while retaining the juxtaposition notation which gives rise to them. Clearest of all on the issue is Dana Scott:

> The completely type-free calculus is a will-o-the-wisp. This has been shown time and again; yet the vision remains. And formal theories are quite as bad as drugs in keeping such illusions alive. The cold light of day reveals, however, perfectly solid ground. The notion of a *type* (as in type theory) is a sound concept. Functions do indeed have domains of definitions and ranges of values, and these ideas can be specified without having to say exactly *which* function is under consideration. Types are a way of making precise a *portion* of our ideas of separating functions into kinds; but this is only a start, since there are other properties besides domains and ranges that often need detailing (Scott 1975, p. 348).

Philosophically, Scott is surely right as against Curry. If a mathematical notation makes untyped calculus seem to work, it must be

because the notation is covering something up, ignoring natural distinctions. Church himself showed how λ-calculus could be combined with (simple) type theory to give a uniquely elegant form of the latter (Church 1940). But the justification for types, or kinds of entity so different from one another that the characteristic expressions for each cannot even be substituted grammatically for one another, is stronger than mathematical convenience and would be natural even if there were no paradoxes (cf. Leśniewski 1992, p. 421).

Church's readiness to use a single notation blurring the linguistic boundaries between functions and objects, or functions of one level and those of another, has led one admirer of Frege to consider that Church was making a fundamental error, and that therewith the whole of λ-calculus rests on a mistake, nay, the grossest confusion possible:

> [Church's] notation for higher level function names lacks the multiplicity of Frege's and so [the definition of the universal quantifiers—PS] is needed to repair the deficiency. [...] The λ-calculus is an extremely ingenious attempt to construct a formal system which has the expressive powers of Frege's ideography but an inadequate symbolism and formation rules to start with. For that reason, though, it is also an extremely misleading system [...] what are, in effect, formation rules are presented under the guise of rules of inference and definitions [...] the lambda calculus marks, not an advance upon Frege, but a regression (Potts 1979, pp. 378–379).

While the untyped λ-calculus may appear to justify such criticism, Church's stipulations regarding ranges of functions already cope formally with the problem, and the criticisms are wholly beside the point when typed calculi are considered, since these obey precisely the sorts of grammatical restriction Frege instinctively respected, and indeed in many respects are more discerning than Frege, who lumped all non-functions together into a single bucket called 'objects', whereas later typed languages in computer science distinguish different types of non-function, such as Boolean, Integer, Real, String, and so on.

When we come to consider the wider implications and applications of λ, it is absolutely essential that we respect differences of category or type. In one important respect I would go further than others in regard to types and λ. In (Church 1940) the λ-symbol is regarded as syncategorematic, functioning more like a parenthesis than a meaningful expression: however, λs can and should themselves be typed, according to precise criteria (Simons 2006). Loss of typographical unity can be compensated by optionally dropping type subscripts and using schemata: this mild cost is more than made up for by the gain in conceptual perspicuity.

3. λ-expressions are Not Names

To the extent that philosophers have thought about λ-expressions, there is a widespread opinion that they *name* functions, or, ontologically speaking, that they name properties, perhaps also relations. There is some authority for this (false) opinion:

> If we abbreviate by 'M' an expression containing 'x' which indicates the value of a function when the argument has the value x, we write
>
> $$\lambda x(M)$$
>
> to indicate the function in itself. Thus $\lambda x(x^2)$ is the square function (Curry, Feys 1958, p. 82).

Here Curry is happy to use a λ-expression designating a function as the grammatical subject of a sentence, in defiance of Frege's insistence that only an incomplete or functorial expression can name a function. Generalizing, we might want to say that whereas in 'This tomato is red' the unsaturated expression '... is red' *expresses* a property, in the saturated expression '$\lambda x.x$ is red' we *name* the property, and can say of it e.g. that it is a property, a secondary quality, is easily recognized etc., and whereas in 'Adam loves Ewa' the unsaturated expression '...loves—' the relation is expressed, in the saturated expression '$\lambda xy.x$ loves y' it is named. In this light, Potts' pans of the λ-calculus appear justified.

Both Curry and Frege (and hence Potts) can and should be resisted. Functions, properties, and relations, if there are such things, can be named. They can also be expressed by other means. The noun expressions 'the square function', 'redness' and 'loving' name the function, property and relation respectively that '$\ldots\ ^2$', '... is red' and '... loves—' non-namingly express. It is a general fact of our conceptual and linguistic abilities that *anything whatever* can be named: what is characteristic about individuals, the lowest items in the type hierarchy, is not that they can be named, but that they can *only* be named, unlike higher-order objects (Simons 2003).

Conversely, λ-expressions are *never* names, despite the fact that in running mathematical prose it is convenient to use expressions like 'the function $\lambda x.x^2$'. The key to how *this* works is the noun 'function', but we will not go into the details of nominalization. In fact, names are one of the few categories of expression that λ-expressions *cannot* be. How and why this is so, comes next.

4. Functors

The idea of a *categorial grammar* is familiar. We have a number of *basic* categories and a number of *functor* categories. A functor is an expression which combines grammatically with one or more other expressions of given categories to yield a complex expression of given category. How the combining is carried out—by what grammatical means—we need not go into: there are numerous possibilities. If the n input categories are $\beta_1 \ldots \beta_n$ and the output category α we designate the category of the functor as $\alpha\langle\beta_1 \ldots \beta_n\rangle$. Typical basic categories in logical languages are SENTENCE (S) and NAME (N): connectives have category S⟨S⟩ (unary), S⟨SS⟩ (binary) and so on; function expressions have categories N⟨N⟩, N⟨NN⟩ and so on; and predicates have categories S⟨N⟩, S⟨NN⟩ and so on. But functors in the most general case may have input and output categories of any category available. In Russellian simple type theory all categories are of the restricted form $\langle \mathrm{S}\beta_1 \ldots \beta_n\rangle$, in Frege's functional logic theory they are of the restricted form $\langle \mathrm{N}\beta_1 \ldots \beta_n\rangle$. Let M be some expression of some category, say α. Suppose there are variables $x_1 \ldots x_n$ of categories $\beta_1 \ldots \beta_n$ respectively. Then the λ-expression $\lambda x_1 \ldots x_n.M$ is a functor expression of category $\alpha\langle\beta_1 \ldots \beta_n\rangle$. That is why it makes perfect grammatical sense to apply it (in the way specified by the language, however that is) to n arguments of suitable category $a_1, \ldots, a_n$ to give a complex $\lambda x_1 \ldots x_n.M(a_1 \ldots a_n)$ which, by λ-conversion (Rule β) has to have the same category as that of $M[a_1/x_1 \ldots a_n/x_n]$, viz. α.

Now we see why λ-expressions cannot be names, or sentences, or any other basic category expressions for that matter: they are always and only functors. Conversely, to suppose that untyped λ-calculus is the last word is like supposing one could manage with a grammar having only one category C and one form of combination, so that the combination of C and C is always C. That is simply not how grammar works: even in the untyped monadic calculus λ-abstracts allow us to form expressions of category C⟨C⟩, C⟨C⟩⟨C⟩, C⟨C⟩⟨C⟩⟨C⟩ and so on. The grammarless language is a chimera.

If we have a language which already contains functors, why then do we need λ-expressions? Are they not redundant? This would appear to justify the Curry line: after all, as *names* λ-expressions add value. But this is to misapprehend the way in which λ-expressions function. Their whole *raison d'être* is not that they are functors, but that they are *unitary* functor expressions formed *ad hoc*, "on the fly", from a matrix which may be as complex as we like. Take for example Baedeker's complete description of Venice in 1900, considered as expressing a single conjunctive proposition. Replace every distinct proper name within the

description by a distinct nominal variable. This leaves a massive open sentence with a huge number of variables. As such it is not a sentence with a truth-value. Now bind the variables in alphabetical order by a single λ. *Now* we have a single unitary functor expression, with a fixed meaning and reference, a relational predicate with several hundred places. It is or may be true of some things. Predicate this of the original proper names taken one at a time and once each only in sequence and we have a truth (Baedeker is assumed infallible). More simply, just take out the name 'Venice' every time and leave a hugely complex open sentence. Bind by λ to give a hugely complex but monadic predicate. This will be true of only one thing: Venice in 1900. It is of the essence of λ that it binds variables, and whereas a functor adds an additional layer of grammatical complexity but operates unitarily, λ, like any other variable binder (quantifier, differential operator, set abstractor etc.) penetrates any number of layers of grammatical structure. The actual way in which λ works grammatically is beyond the scope of this essay: it involves extending categorial grammar to cope with variable binders (Simons 2006).

Whereas all other variable binders add some additional meaning of their own, λ is unique in *merely* forming a functor, which can be reapplied by the β-rule to give us back something equivalent to the original expression. It is a syntactic unifying or gathering device. It is this neutrality which allows λ to be used as the sole variable binder, all others being equivalent to the product of a functor and λ, as Ajdukiewicz and Church independently recognized (Ajdukiewicz 1967, Church 1940). For example, the universal quantifier $\forall x.S$ can be construed as the product of a universal functor $\prod$ with a λ-expression: $\prod(\lambda x.S)$, and set abstraction $\{x|S\}$ as a product of a set-functor σ with a λ-expression: $\sigma(\lambda x.S)$. λ is therefore useful as a general device for setting up explicit definitions. Instead of defining disjunction via conjunction and negation in context as

$$A \vee B =_{Df.} \sim (\sim A \wedge \sim B)$$

we can define it directly and in isolation as

$$\vee =_{Df.} \lambda pq. \sim (\sim p \wedge \sim q).$$

In some logical languages, such as Leśniewski's logic of ontology, there are unitary (simple) names defined in the language, for which such definitions will not work because what is defined is a simple expression in a basic category. Leśniewski defines universal and empty names contextually via equivalences:

Def V $\quad \forall a(a\varepsilon\mathrm{V} \leftrightarrow a\varepsilon a)$

Def Λ $\quad \forall a(a\varepsilon\Lambda \leftrightarrow (a\varepsilon a \wedge \sim a\varepsilon a))$

By allowing a *name-forming* abstractor one could emulate λ's definitional role. However, the use of such an operator would be limited: it is better to introduce a single functor τ of category N⟨S⟨N⟩⟩ specifiable via the axiomatic equivalence

Def τ $\quad \forall af(a\varepsilon\tau(f) \leftrightarrow a\varepsilon a \wedge f(a))$

—we can read '$\tau(f)$' as 'thing that fs'—and use this in tandem with λ to define what we want, e.g.

V	=	$\tau(\lambda a.a\varepsilon a)$	(thing that is one of itself)
Λ	=	$\tau(\lambda a.a\varepsilon a \wedge \sim a\varepsilon a)$	(thing that is and isn't one of itself)
N	=	$\lambda b(\tau(\lambda a.a\varepsilon a \wedge \sim a\varepsilon b))$	(thing that is not one of the ...).

Thus λ can always be used in definitions, even of simple names.

5. What Functional Expressions Signify

Consider the following partial table of arguments and values for a function:

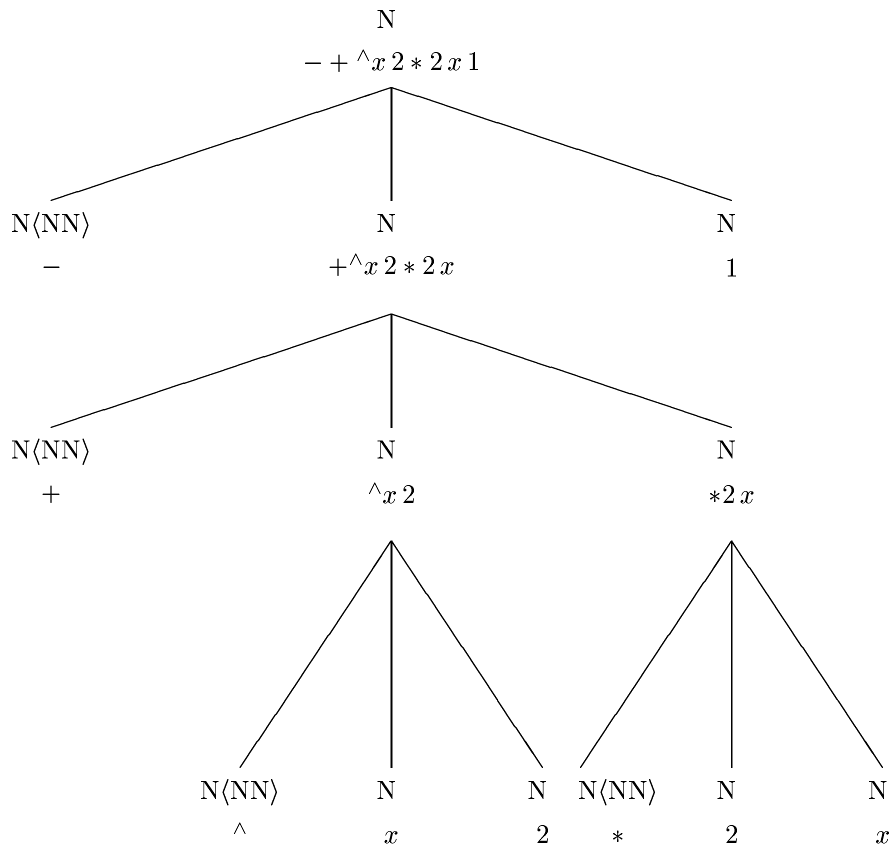
x	-5	-4	-3	-2	-1	0	1	2	3	4	5
$f(x)$	14	7	2	-1	-2	-1	2	7	14	23	34

What is the pattern here? Assuming the function is extrapolated beyond these values in the simplest possible way, it is a quadratic. One of infinitely many λ-expressions giving a function with this table is $\lambda x.x^2 + 2x - 1$. Others include

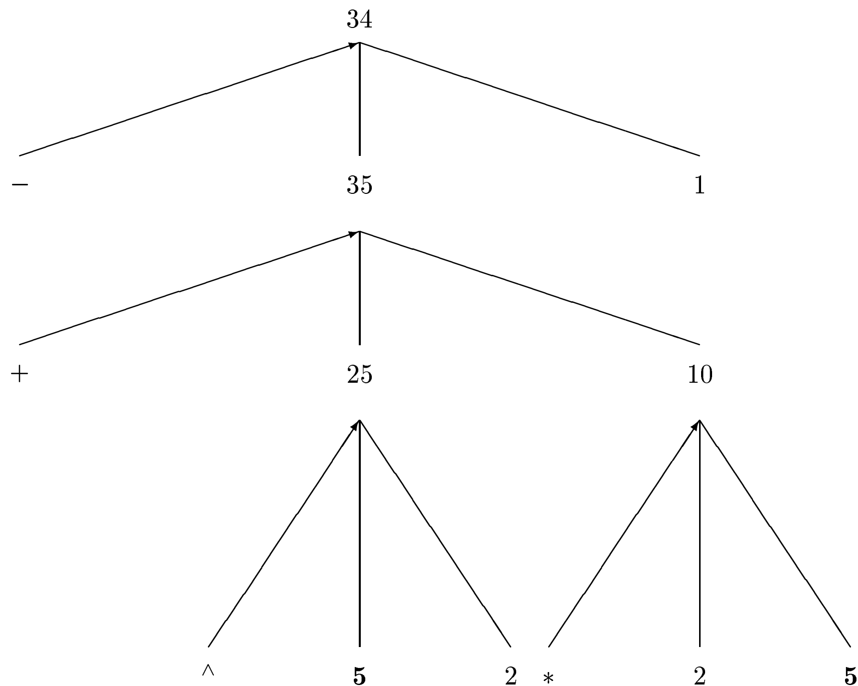
$$\lambda x.(x + m)(x + n) - (m + n - 2)x - (mn + 1)$$

for any m and n. Are these all the same function, or are they different functions with the same extension? Are coextensional functions identical or not? There are different versions of the λ-calculus corresponding to either response.

Consider how the function $\lambda x.x^2 + 2x - 1$ is parsed and calculated. Disambiguating the order of operators and moving to Polish (prefix bracketless) notation gives $\lambda x. - +{}^\wedge x2 * 2x1$, which has category N⟨N⟩ as expected. The matrix is parsed top-down as follows:



When the function is evaluated for a particular value, say $x = 5$, evaluation moves up the tree, taking the input leaf values (objects and functions) and calculating or evaluating the outcome. The evaluation tree then looks like:



where the arrows indicate the evaluation steps. Evaluation then proceeds in the inverse direction to parsing and respects the grammatical structure revaled in the parsing. In evaluating $- \ + \ {}^\wedge \ 5 \ 2 \ * \ 2 \ 5 \ 1$ we first input all the constants (including operations) according to their position in the structure/eval tree, then evaluate $^\wedge \ \mathbf{5} \ 2 \ (= 25)$ and $* \ 2 \ \mathbf{5} \ (= 10)$, then evaluate $+ \ 25 \ 10 \ (= 35)$, then evaluate $- \ 35 \ 1 \ (= 34)$ and stop. A coextensional function with a different matrix would typically have a different parsing and would therefore be evaluated by a different procedure.

So does the λ-expression signify (*bedeuten*) a function in extension or in intension? A function in extension is simply a correlation of output values for input arguments. A function is, however, generally conceived as a *rule or procedure* and not just as the table of results of the procedure, which is the extension, graph or *Wertverlauf* of the function. This suggests treating the function as *the way in which* the table is computed, or the *mode of determination* of the values. The problem with this is that, understood literally, it is too fine-grained. For example, the computing function `sort_ascending_alphanumeric` can

be implemented in many ways, `factorial` can be computed iteratively or recursively, functions may be evaluated lazily or eagerly, etc. The number of ways in which a function can be actually computed by a machine is infinite.

The sensible compromise between the two extremes, which arguably corresponds to the idea of function targeted by Church, is to take the function to be individuated by the computational or determinational route as prescribed by the structure of the functional expression. *How* the individual procedures are implemented is not important, but the (partial) order in which they are performed is. This notion is stable, computationally and logically significant, and corresponds not to cointensionality as standardly understood, but to what Carnap called *intensional isomorphism* (Carnap 1947, pp. 56–57). The key thought is that while the details of the procedure of evaluation are unimportant, the evaluation should *respect the structure* of the whole expression. This key thought will turn out to be useful elsewhere.

6. Vexillology

We have so far followed Church in considering principally expressions signifying functions, with some cases from logic where the expressions are functors not signifying functions, namely predicates and connectives. We now consider a very different field of application: vexillology. This unfamiliar term comes from the Latin *vexillum*, flag, and means the study of flags. Since in this article I shall not indulge in the luxury of colour illustrations, the examples will have to be described and briefly illustrated in black and white. Take a classic flag, the French tricolour. This consists of three equally broad vertical bands of blue, white and red, in order starting from the hoist with blue, white in the middle, and red at the fly. Let us abbreviate this description as

France = VT(b,w,r)

where 'VT' means 'vertical tricolour' and the order of colours is (hoist, middle, fly). As anyone knows who knows their flags, there are lots of different vertical tricolours: any of the three bands may take any colour, so there is a *range of variation* just as there is with a function. The Italian tricolour differs from the French in substituting green for blue at the hoist, while the Romanian substitutes yellow for white in the middle. The Irish tricolour can be obtained from the Italian by substituting orange for red at the fly (the proportions are different but we shall ignore that). If we substitute red for the French blue we get the red-white-red vertical bicolour of Peru. Finally if we substitute green for all three bands we get the uniformly green flag of Libya. To abstract

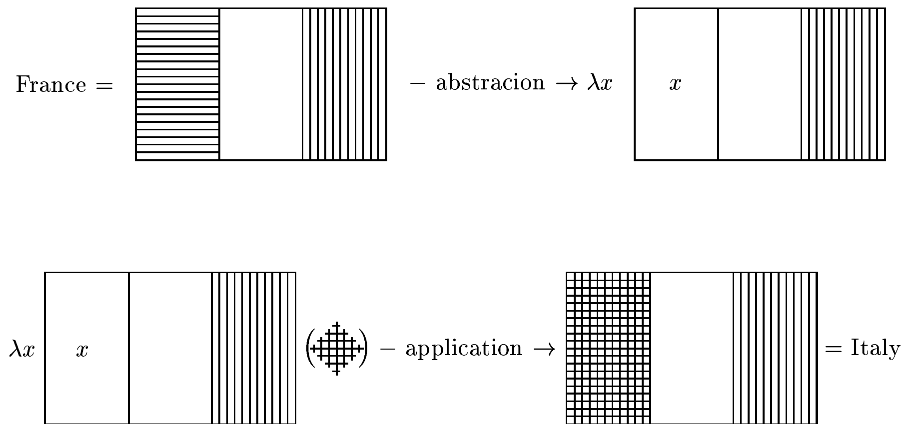
what is common to the French, Italian and Peruvian flags we may use the expression

$$\lambda x.\mathrm{VT}(x, \mathrm{w,r})$$

and the replacement of blue by green can be expressed by

$$\mathrm{VT(b,w,r)[g/b]} = \lambda x.\mathrm{VT}(x, \mathrm{w,r})(\mathrm{g}) = \mathrm{VT(g,w,r)} = \mathrm{Italy}.$$

Graphically, even without colour, it is more dramatic:



Vexillological operations include not just substitution of one colour for another, but also rotation (e.g., vertical to horizontal tricolour), change of proportion, and inclusion as a part. A common form of inclusion is the use of a canton, a rectangle at the upper fly. If 'UJ' stands for the British Union Jack, ... is any description, and Cant(...) describes a canton of a flag, then placing the Union Jack at the canton of a flag, as in the British ensigns, the national flags of Australia and New Zealand, and the state flag of Hawaii, may be described schematically by

$$\lambda x.(\ldots \mathrm{Cant}(x) \ldots)(\mathrm{UJ})$$

Substitutions may take place not just at the outermost level of detail but further "inside" a flag, as when the British red and blue ensigns replaced the pre-1801 Anglo-Scottish Union flag by the post-1801 version including the counterchanged red saltire of St. Patrick. A more politically charged vexillological substitution occurred when the 1776 Grand Union flag of red and white stripes with a Union Jack at the canton was replaced by a canton of white stars on a blue ground to give the first flag of the United States. While flags are not usually very

complex in depth of structure, heraldic arms may be more so. The quartering of arms for instance, which is occasionally found in flags, is much more common, and involves structural inclusion to greater depths.

We have seen how λ-abstraction can be applied to complexes such as flags. But is this not a game? And is there not a use/mention confusion going on? Are we not confusing the flag with its description?

I will reply to the first and more serious question shortly. To the second question I reply that we are *not* making a use/mention confusion. The description of a flag is one thing, the flag is another, and the substitution talked about is not of one word 'blue' by another word 'green' but of one coloured segment of a flag by another. In concrete terms it would involve cutting off the blue part of a French tricolour and sewing in its place a green part of the same size and shape to give an Italian tricolour. This is *real* substitution, not symbolic substitution. Also, for the substitution to be real, we would need to be considering a concrete flag token. Substitution may indeed be considered for the abstract flag type, and in this case the substitution is not real but the passage from one type to another differing from it and agreeing with it in precisely given ways. This raises the question what the *abstractum* is in the case of a real flag. It is certainly *not* the remaining two thirds of the flag, since this is itself a concrete and substantial particular, and need have nothing to do with a third part. In the case of an abstract flag type the abstractum is not a determinate type but an incomplete form of a type, consisting of two thirds given as white in the middle and red at the fly, with the hoist third indeterminate or "variable". It is thus very like a function, in that its "completion" by a determinate part yields a determinate flag type or pattern. However, it is far less clear what the "remainder" or abstractum is in the case of the concrete flag. It will also have to be somehow incomplete, in that it is "hungry" for a third part, to make it into a full tripartite flag, but it also somehow contains two parts which are wholly concrete. The situation here is exactly the same as when we consider a concrete sentence token and abstract from one or more of its places.

As to the question whether the use of λ-abstraction and application in the case of flags is a game, the answer is that while the example is chosen for ease of illustration rather than seriousness, the idea of considering abstraction, structure and application outside mathematics is serious. To show this is not a game, we turn to the second and highly serious non-mathematical application.

7. Manufacturing

Manufacturing is the intentional production of artefacts. Serious manufacturing, and its science, engineering, starts when the artefacts are constructed of several parts put together in a certain way. The first really serious artefacts were no doubt buildings: houses, temples, pyramids. But nowadays we tend to think of manufacturing in terms of the mass production of items such as cars. Mass production as a technique goes back to an American genius called Eli Whitney (1765-1825). Here is the important part of Whitney's story:

> In 1797, the [U.S.] government, threatened by war with France, solicited 40,000 muskets from private contractors because the two national armories had produced only 1,000 muskets in three years. Twenty-six contractors bid for a total of 30,200. Like the government armories, they used the conventional method whereby a skilled workman fashioned a complete musket, forming and fitting each part. Thus, each weapon was unique; if a part broke, its replacement had to be especially made. Whitney broke with this tradition with a plan to supply 10,000 muskets in two years.
>
> He designed machine tools by which an unskilled workman made only a particular part that conformed precisely, as precision was then measured, to a model. The sum of such parts was a musket. Any part would fit any musket of that design. He had grasped the concept of interchangeable parts. [...] But $10\frac{1}{2}$ years passed before Whitney delivered his 10,000 muskets [...] Finally, he overcame most of the skeptics in 1801, when, in Washington D.C., before President-elect Thomas Jefferson and other officials, he demonstrated the result of his system: from piles of disassembled muskets they picked parts at random and assembled complete muskets. They were the witnesses of the inauguration of the American system of mass production (Mirsky 1974, p. 641).

The key to mass production is interchangeability of parts (and *not* the production line, which is a later invention). Only when parts are sufficiently similar in shape and material are they truly interchangeable. Whitney, utilizing Vernier's improvements in measuring techniques and the advent of machine tools, made the conceptual leap to real interchangeability, though in the venerable tradition of defence contractors, he was late and no doubt over budget. The only issue I would take with the description above is that a musket is not a *sum* of parts: it is a structured whole of parts put together in a certain way. Let us see how structure, substitution and abstraction apply in manufacturing.

Case I

1. An individual artefact (car, aeroplane, food-mixer, CD player or whatever) is made with a component part P. Suppose A represents

the whole artefact and P the part, R the remaining parts, and $M(RP)$ the operation of joining P with R to give A. The operation of inserting a part like P may be represented by $\lambda x.M(Rx)$, and the result of applying this to P is A. So A is the result of $\lambda x.M(Rx)(P)$.

2. If the part P has to be replaced by another part P^*, but the result is structurally and functionally equivalent, we have that $\lambda x.M(Rx)(P^*)$ results in A^*.

3. Replacement may also occur not just at the token but also at the type level, e.g. when a subcontracted part manufacturer goes out of business and a new manufacturer delivers equivalent parts (this is very common).

Notice that in employing λ-abstraction we do not change the overall *nature* of the item represented: M is not a function but an operation of part-insertion, so $\lambda x.M(Rx)$ is an operation type of inserting a part into the remainder R. In leaving the nature of the item abstracted over intact, λ is performing its pure abstractive role as emphasized above in connection with functors.

To understand the next example it is necessary to know what engineers mean by a *bill of materials* (abbreviated BOM). Originally simply a list of parts, this has evolved to become the principal engineering document for managing and costing complex artefacts. It is in effect an annotated mereological description, detailing the parts, their properties, number, and how they go together to make up subassemblies and ultimately the whole. Originally BOMs were paper documents like lists and blueprints. Today they are large and complex computer databases with tens or hundreds of thousands of representations of parts. Unfortunately, because of the complexities of manufacturing, there are typically several BOMs, which reflect the interests and needs of different castes of engineers. Herein lie the practical problems. An engineered artefact is rarely static: engineering changes caused by design improvements, unexpected failures, innovations in manufacturing techniques and
materials, business and regulatory changes etc., mean that any reasonably complex item with a long life will be subject to change at many levels and scales, affecting design, manufacturing, maintenance, modification and retirement. Managing this change in the case of complex artefacts is a huge job.

Case II

1. Design engineers design an aluminium wing-spar S for an aeroplane. It figures as one part on the Engineering Bill of Materials (E-BOM)
2. The spar is too long to be manufactured with sufficient strength and sufficient precision by existing methods.
3. Manufacturing engineers redesign the spar to be made from two parts I and O which are then joined together. These two parts, and not S, figure on the Manufacturing Bill of Materials (M-BOM).
4. Denoting the *result* of the joining operation by '+' we have that S is structurally equivalent to $I + O$.
5. For whatever reason, the wing is redesigned with a longer spar S^*.
6. Manufacturing changes only the outboard spar part, replacing O by O^*.
7. The modified manufacturing process substitutes O^* for O in the joining.
8. That which remains invariant in this change is *result of joining* to I, $\lambda x.I + x$.
9. The difference between S and S^* is a material, quantitative and mereological, not a structural difference.
10. The difference between S and $I + O$ is a *structural* difference.

The difficulties involved when such changes multiply in a complex artefact is not simply that of exploding complexity: it is that the nature of the discrepancies is not well understood, so the changes can rarely be propagated efficiently by automation, but need to be handled slowly and expensively, by hand and brain. Here are some facts:

- Structural discrepancies between different BOMs, and their proper management, account for a considerable proportion of the high manufacturing costs, cost overruns, and delivery delays of complex manufactured artefacts.
- Example: In 1998 the M-BOM database for a modern military transport aeroplane contained c. 22,000 distinct types of parts, while the E-BOM had 7,000 fewer, without database links.

- No adequate system for managing the discrepancies exists.
- At most one adequate conceptual scheme for describing and classifying the discrepancies exists (Simons, Dement 1996; Dement, Mairet, DeWitt, Slusser 2001).
- Abstraction within a structure (hence λ) must figure in the conceptualization of any adequate system for managing multiple BOM discrepancies.
- One thing we have respected but not focussed on here is the distinction between an operation on parts (which is a process) and the result of that operation (which is not).

With these remarks we have merely indicated the multiple BOM problem. Actually dealing with it, even conceptually, is much more difficult (Dement, Mairet, DeWitt, Slusser 2001).

8. λ Unchained

The moral I draw from these examples is that it is time λ came out of the maths room and entered the factory, and elsewhere. Let us see why λ has such widespread potential for application.

Many things are complex wholes. A complex whole is an object with more than one proper part, such that the parts are related together in the whole in a determinate way. This way of their being together in the whole is the *structure* of the whole. We may consider the whole in abstraction from its parts or their nature, but more frequently it is advantageous to consider it in *partial abstraction*: leaving some parts fixed and considering others variable, which may include one or more individual tokens of a type, or all of a type, or indeed may allow structure to vary. In any case where we consider some parts or aspects fixed and allow others to vary, we are in the market for λ-abstraction.

Typically in a complex whole the parts are organized hierarchically, with simpler parts making up intermediate wholes of different complexity and different type. For example, a piece of carefully shaped steel is a turbine blade, which is part of a rotor disc, which is part of a turbine, which is part of an engine, which is part of an aeroplane. A book consists of chapters, which consist of paragraphs, which consist of sentences, which consist of words. A symphony is made up of movements, which are made up of passages, with different melodies, harmonies and rhythmic components, which a composer typically varies in inventive ways (transposition, thematic modification, change of instrumentation)

to provide the unity-in-diversity characteristic of good music. Parts of a complex whole may be replaced or substituted, either singly, or throughout. Parts may be replaced by others similar, or by ones with different internal structure, but playing a structurally and functionally similar role, or by wholly new ones, resulting in a qualitatively different whole. The whole may undergo various transformations, some affecting its structure. A car may be painted a different colour, or be fitted with winter tyres instead of summer tyres. A piece of music may be transcribed from piano to string quartet. A combat aircraft may be fitted with bombs instead of missiles, or an external fuel tank and a camera instead of either. A story may be written instead of told. We can track the discrepancies and changes using λ. As in the case of functions, λ-abstraction can reach arbitrarily deep into the mereological structure of a whole, through hierarchies of parts.

What has been said applies *mutatis mutandis* to all complex wholes: concrete artefacts as well as written and spoken sentences, pieces of music, processes, proteins and molecules including such as DNA, organizations, and last but not least mathematical functions. The last, being *entia rationis*, are of course much more exact and well-behaved than any crassly physical individual, lacking the vagueness, indeterminacy and unexpectedness of the real. Now we see the deep reason why we have to understand functions as rules. When expressed by an articulated expression (articulated: with parts in a structure) the notions of whole and part, substitution, variation, and structure apply to the concept of function. The extension or table of a function gives not the function, but its results. The rules embodied in a function are structured and have parts and operate on things. That is why λ can be applied to functions as well as to other things.

References

Ajdukiewicz, K., 1967, "Syntactic Connexion", in S. McCall (ed.), *Polish Logic 1920-1939*, Oxford: Clarendon Press, pp. 207–231.

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

Barendregt, H., 1992, "Lambda Calculi with Types", in: S. Abramsky, D. Gabbay, T. Maibaum (eds.), *Handbook of Logic in Computer Science*, Oxford: Oxford University Press, Vol. 2, pp. 117–309.

Carnap, R., 1947, *Meaning and Necessity. A Study in Semantics and Modal Logic*, Chicago: University of Chicago Press.

Church, A., 1932, "A Set of Postulates for the Foundations of Logic", *Annals of Mathematics* 33, pp. 346–366.

Church, A., 1940, "A Formulation of the Simple Theory of Types", *Journal of Symbolic Logic* 5, pp. 56–68.

Church, A., 1941, *The Calculi of Lambda Conversion*, Princeton: Princeton University Press.

Curry, H., Feys, R., 1958, *Combinatory Logic*, Amsterdam: North-Holland.

Dement, C., Mairet, C., DeWitt, S., Slusser, R., 2001, *MEREOS*, US Air Force Research Laboratory, Materials and Manufacturing Directorate, Report No. A855793. Available through Defense Technical Information Center, http://www.dtic.mil/.

Klement, K., 2003, "Russell's 1903-1905 Anticipation of the Lambda Calculus", *History and Philosophy of Logic* 24, pp. 15–37.

Leśniewski, S., 1992, *Collected Works*, Dordrecht: Kluwer.

Mirsky, J., 1974, "Whitney, Eli", in *Encyclopadia Britannica*, 15th ed., Chicago: Encyclopadia Britannica, Vol. 12, pp. 640–641.

Potts, T., 1979, "'The Grossest Confusion Possible?' – Frege and the Lambda-Calculus", *Revue Internationale de Philosophie* 33, pp. 761–785.

Russell, B., 1903, *The Principles of Mathematics*, London: Allen & Unwin.

Scott, D., 1975, "Some Philosophical issues Concerning Theories of Combinators", in: C. Böhm (ed.), *Lambda-Calculus and Computer Science Theory*, Berlin: Springer, pp. 346–366.

Simons, P., 1988, "Functional Operations in Frege's *Begriffsschrift*", *History and Philosophy of Logic* 9, pp. 35–42.

Simons, P., 2003, "The Universe", *Ratio* 16, pp. 236–250.

Simons, P., 2006, "Languages with Variable-Binding Operators: Categorial Syntax and Combinatorial Semantics", in J.J. Jadacki, J. Paśniczek (eds.), *The Lvov-Warsaw School. The New Generation*, Amsterdam: Rodopi, pp. 239–268.

Simons, P., Dement, C., 1996, "Aspects of the Mereology of Artifacts", in: R. Poli, P. Simons (eds.), *Formal Ontology*, Dordrecht: Kluwer, pp. 255–276.

Whitehead, A.N., Russell, B., 1910, *Principia Mathematica*, Cambridge: Cambridge University Press.