M. H. Newman's Typability Algorithm for Lambda-Calculus

J. Roger Hindley [*]

July 9, 2006

Abstract

This article is essentially an extended review with historical comments. It looks at an algorithm published in 1943 by M. H. A. Newman, which decides whether a lambda-calculus term is typable without actually computing its principal type. Newman's algorithm seems to have been completely neglected by the type-theorists who invented their own rather different typability algorithms over 15 years later.

1 Introduction

The English topologist M. H. Newman maintained through most of his career a vigorous side-interest in logic, and even wrote three papers in this field in 1941–2. One of these, [23], was about the Church-Rosser confluence theorem and is well known in the λ-calculus and term-rewriting world. The second, [25], was written jointly with Turing and concerned Church's type theory, on which Turing had already done some work.

The third, [24], is the topic of the present paper. It contained an interesting abstract algorithm for deciding typability in type-theories in general, including Church's typed λ-calculus and Quine's logical system called "New Foundation". This algorithm partly anticipated the typability algorithms that Curry and others made from the 1950s onward, yet it seems to have been unknown to those workers. (I first saw it in 2005.) It was slightly weaker than the later algorithms, however, and its strategy was very different from them.[1]

The present article is an extended review of Newman's [24], greatly aided by hindsight. It will show how Newman's typability test works in the two systems mentioned above, with examples. But it will not state that test in full generality; it is not meant to be a substitute for reading [24].[2]

[*]Mathematics Department, Swansea University, Swansea SA2 8PP, U.K. E-mail address: j.r.hindley@swan.ac.uk

[1]A short history of the other typability algorithms is in §4 below.

[2]The present article is expanded from a talk given in the workshop "Lambda Calculus, Type Theory and Natural Language", 12 September 2005 in King's College, London.

For helpful comments and corrections during this article's preparation I thank especially Mariangiola Dezani-Ciancaglini, Thomas Forster, Randall Holmes, Benjamin Pierce, Pawel Urzyczyn and the anonymous referees.

1.1 (Historical background) The starting point of Newman's [24] was a 1937 paper by W. V. Quine, [26]. In it, Quine had proposed a new foundation for mathematics, made by restricting the comprehension axiom-scheme in set theory to formulas he called *stratified* [26, pp.78–79]. Roughly speaking, a formula F is stratified iff there exists an assignment of types to its variables such that in every sub-formula $(x \in y)$, the type of y is the type of x plus one. (For Quine, types were integers.) Quine's system is nowadays called *NF* for "New Foundation"; a modern account is in [12].

In his [24], Newman described a general typability algorithm. As particular cases, he proved that it could (i) decide whether a formula of NF is stratified, and (ii) decide whether a λ-term is typable in simple type theory. He intended that it might apply to other systems too, as yet unbuilt.

His paper was reviewed by Church in the Journal of Symbolic Logic 9 (1944), pp.50–52. However, despite this, it seems to have been ignored by most of the later type-theory community. (Though it is listed in A. Rezus' bibliography [27].) Nor, as far as I know, did it influence those who later worked on NF. Why this neglect?

First, to state his algorithm in its most general form, Newman had to develop a notation for the structure of formulas in an abstract setting which covered both (i) and (ii). This might have seemed over-elaborate in the 1940s. (Though to a modern type-theorist with a computing background it is nothing unusual.)

Second, many logicians were busy with war work when the paper and review appeared.

Third, studies of Quine's NF only gained momentum about 30 years later, because that system turned out much harder to investigate than hoped.[3]

Fourth, in type theory no need was seen for a typability algorithm until the 1950s and '60s; Newman was ahead of his time here too. (He wrote his algorithm as if it applied to Church's type-system in [4], but in that system the concept of "typable" was actually trivial since terms without types were excluded from Church's language. Newman was in fact taking a Curry-style approach to type theory (see below). But Curry's interest then was in more elaborate type-systems than the simple one he formulated later. So, for different reasons, neither Church nor Curry would have found [24] directly interesting. Church's review consisted mainly of a summary of the paper with emphasis on Newman's general notation for describing formula-structure.)

1.2 (Typability) Church's type-theory in [4] was based on a language of typed λ-terms, each of which contained a built-in expression called its *type*.

In contrast, Curry's type-theories from [5] onwards were more flexible: his terms were *untyped*, and types were assigned to them by formal rules. Some

[3]See [12] for work on NF up to 1995. A typability algorithm for NF was made by R. Holmes in 1991/92 and implemented as part of a theorem-proving program; see [1].

terms received an infinite set of types, and some received none.

The concept of typability is relevant to type-systems made in Curry's style: an untyped λ-term M is called *typable* in such a system iff the system's rules assign a type to it. In the special case of simple type theory, a type τ is called a *principal type* (PT) of M iff the types assigned to M are exactly all the substitution-instances of τ.

A *typability test* is an algorithm which decides whether a term M is typable. It is called a *PT algorithm* iff it furthermore assigns to M a PT (if M has one).

In the 1950s and 1960s, some studies were made of the simplest interesting Curry-style λ-based type system (see $\mathrm{TA}_{\lambda\to}$ below), and several typability tests were invented for it. Their strategy was simply to try to assign to the input-term M the most general type consistent with the system's rules, and consequently they were also PT algorithms.

Indeed, it is hard to imagine a strategy that would test for typability without trying to assign a type. Nevertheless, that is exactly what Newman's test did. (It was therefore a partial anticipation of the later PT algorithms, but not a complete anticipation.) His strategy depended on noting relative positions of occurrences of atoms in the given term.[4]

1.3 (The system $\mathbf{TA}_{\lambda\to}$) To serve as a reference-point in what follows, I shall define here a Curry-style system $\mathrm{TA}_{\lambda\to}$ of simple type theory.[5] Newman referred to Church [4], but as mentioned above, his examples on λ-calculus show that he actually had something like $\mathrm{TA}_{\lambda\to}$ in mind. (His actual algorithm was stated in a precise general context independently of these examples.)

The *terms* of $\mathrm{TA}_{\lambda\to}$ are pure λ-terms (i.e. λ-terms whose only atoms are variables). Its *types* are simple types, i.e. types built from type-variables (and perhaps some atomic type-constants) by infixing "$\to$".

The *formulas* of $\mathrm{TA}_{\lambda\to}$ are expressions of form $M : \tau$, read as "assign type τ to term M".

$\mathrm{TA}_{\lambda\to}$ is a Natural Deduction system. A *deduction* $\mathcal{D}$ is a tree of formulas with assumptions at the tops of its branches and the conclusion at the bottom. Some assumptions may be *discharged* (shown by enclosing the assumption in brackets).

Assumptions have form $x : \tau$, where x is a (term-)variable. *Deductions* are built by two *rules*, written informally as follows.

($\to$e): From $P : (\sigma \to \tau)$ and $Q : \sigma$, deduce $(PQ) : \tau$.

($\to$ i): If a deduction $\mathcal{D}$ has conclusion $M : \tau$, and its only undischarged assumptions which contain x have form $x : \sigma$ (all with the same σ), then

[4] A similar strategy was re-discovered independently in the 1980s. It was used in 1988 by J. Tyszkiewicz in a proof that the problem of deciding typability is PTIME-complete, [30], and was there attributed to J. Tiuryn. Cf. also a strategy used in 1984 by C. Dwork, P. Kanellakis and J. Mitchell in the context of unification, [11].

[5] In what follows, the λ-calculus notation of [15, Ch.1] will be used; it is fairly standard. The symbol "$\equiv$" will denote identity of expressions. System $\mathrm{TA}_{\lambda\to}$ is also fairly standard; it is a slight simplification of [15, §15B System TA_{λ}], [2, §3, System $\lambda\to$-Curry] and [14, §2A8 System TA_{λ}].

deduce $(\lambda x.M):(\sigma \to \tau)$ and discharge all the undischarged assumptions $x:\sigma$ in $\mathcal{D}$.

1.4 (The system $\mathbf{TA}^{res}_{\lambda\to}$) Newman's test does not actually apply directly to $\mathrm{TA}_{\lambda\to}$, but to a variant, which I call here $\mathrm{TA}^{res}_{\lambda\to}$.

This system is obtained by restricting the rules of $\mathrm{TA}_{\lambda\to}$ to ensure that if a term M receives a type and x is any variable, then all occurrences of x in M (free, bound or binding) have the same type.

For example, the term $(\lambda x.x)(\lambda x.x)$ is not typable in $\mathrm{TA}^{res}_{\lambda\to}$. (It is typable in $\mathrm{TA}_{\lambda\to}$ by a deduction in which the second x from the left receives a type $\sigma \to \sigma$ and the fourth x receives σ, but this is not allowed in $\mathrm{TA}^{res}_{\lambda\to}$.)

The detailed formulation of $\mathrm{TA}^{res}_{\lambda\to}$ is left to the reader.

1.5 (Comparing $\mathbf{TA}^{res}_{\lambda\to}$ with $\mathbf{TA}_{\lambda\to}$) Typability in $\mathrm{TA}_{\lambda\to}$ can be proved to be invariant under changes of bound variables. But for $\mathrm{TA}^{res}_{\lambda\to}$ this fails; for example $(\lambda x.x)(\lambda x.x)$ is untypable but $(\lambda x.x)(\lambda y.y)$ is typable.

However, a test for typability in $\mathrm{TA}^{res}_{\lambda\to}$ provides also a test for typability in $\mathrm{TA}_{\lambda\to}$. Because, if M is any λ-term, its bound variables can be changed to get a term M° in which no λx occurs twice with the same x; and then

$$\begin{aligned} M^\circ \text{ is typable in } \mathrm{TA}^{res}_{\lambda\to} &\iff M^\circ \text{ is typable in } \mathrm{TA}_{\lambda\to} \\ &\iff M \text{ is typable in } \mathrm{TA}_{\lambda\to}. \end{aligned}$$

2 Newman's test, NF case

Newman's general typability test is stated in [24, pp.69–73], and is justified by proofs in later pages of that paper. The following shows how it applies to NF.

2.1 (Typability test for NF) Let F be a formula of NF, built from prime formulas of form $(x \in y)$, where x and y are variables.

Step 1. Reduce F by a series of replacements, as follows.

(a). If F contains two sub-formulas with the same right-hand side,

$$x_1 \in y, \qquad x_2 \in y,$$

then replace x_1 by x_2 throughout F.

(b). If F contains two sub-formulas with the same left-hand side,

$$x \in y_1, \qquad x \in y_2,$$

then replace y_1 by y_2 throughout F.

Motivation: Each of these replacements preserves stratifiability/unstratifiability. To see this, let F' be the result of making a replacement (a). If F is stratifiable by a type-assignment which gives y a type $n+1$, then x_1

and x_2 must both receive type n. Hence we can stratify F' by giving type n to x_2. Conversely, if F' is stratifiable by giving x_2 a type k, then F is stratifiable by giving k to both x_1 and x_2. The case of (b) is similar.

(By the way, these replacements are purely formal, and are not claimed to preserve the semantic meaning of F.)

Each replacement reduces the number of distinct terms occurring in F, so the series of replacements must terminate. Let F^* be the resulting formula. (Newman proves that F^* is independent of the order in which the replacements are made [24, p.70].) Then F^* is stratifiable iff F is so.

Step 2. Test F^* for stratifiability by searching for "cycles" with form

$$x_1 \in x_2, \quad x_2 \in x_3, \quad x_3 \in x_4, \quad \ldots \quad x_r \in x_1 \qquad (r \geq 1).$$

Iff no such cycles occur, Newman concludes that F^* (and hence F) is stratifiable.

2.2 (Note) If F contains quantifiers, and x_1 in 2.1(a) is a bound variable, it must be replaced in the quantifier too. Similarly for 2.1(b).

2.3 (Note) Types are not mentioned at all in the above test, only in its motivation. Indeed, only the relative positions of x_1, x_2, y, etc. in F play a role. This "positionality property" is the key to the test. In fact, Newman states abstract axioms in [24] which say, roughly speaking, that typability is determined by relative positions, and gives a proof that his test works under these axioms.

3 Newman's test, $\boldsymbol{\lambda}$ case

The strategy of Newman's test for λ-calculus is a development of that for NF. It will be shown here without details. As mentioned before, the test itself is stated in full in [24, pp.69–73], and there is a proof in [24] that it works.

3.1 (Typability in $\mathbf{TA}^{res}_{\lambda\to}$) Before describing the test, we try to express typability in $\mathrm{TA}^{res}_{\lambda\to}$ in a way that emphasises relative positions, not types. Let M be any λ-term. Define two relations γ_1 and γ_2 between subterms of M (i.e. terms which occur in M), thus.

(a). If M contains an application $Z \equiv (XY)$, define

$$X \ \gamma_1 \ Z, \qquad X \ \gamma_2 \ Y.$$

Motivation: If M was typable, any type-assignment for M would have to include

$$Z^\tau \ \equiv \ \big(X^{\sigma\to\tau} Y^\sigma\big)^\tau$$

(for some types σ, $\sigma \to \tau$, τ shown here as superscripts). By "$X\gamma_1 Z$" we mean informally that the type of X in such an assignment must have the type of Z as its conclusion. By "$X\gamma_2 Y$" we mean that the type of X must have the type of Y as its hypothesis.

(b). If M contains an abstraction $Z \equiv (\lambda x.U)$, define

$$Z\ \gamma_1\ U, \qquad Z\ \gamma_2\ x.$$

Motivation: If M was typable, any type-assignment for M would have to include

$$Z^{\sigma\to\tau} \ \equiv\ \left(\lambda x^{\sigma}.U^{\tau}\right)^{\sigma\to\tau}$$

(for some σ, τ). We mean by "$Z\gamma_1 U$" and "$Z\gamma_2 x$" that the type of Z in such an assignment must have as its conclusion the type of U, and as hypothesis the type of x.

3.2 (Note) Although the motivations of γ_1 and γ_2 are that "$P\gamma_1 Q$", "$P\gamma_2 R$" should always mean that the type of P has form (*type of* $R \to$ *type of* Q), their actual definitions do not mention types.

3.3 (Example, from [24] p.73) The term $u(ux)$ is typable in $\mathrm{TA}^{res}_{\lambda\to}$.

Method: Let $M\ \equiv\ u(ux)$. List the steps in the construction of M, thus:

$$M\ \equiv\ uP, \qquad P\ \equiv\ ux. \tag{1}$$

Then write all the γ-relationships for these steps:

$$u\ \gamma_1\ M, \quad u\ \gamma_2\ P, \quad u\ \gamma_1\ P, \quad u\ \gamma_2\ x. \tag{2}$$

Next, look for pairs of γ_1-statements in (2) with the same left side: we find only $\langle\, u\gamma_1 M,\ u\gamma_1 P\rangle$. Replace "$P$" by "$M$" throughout (1) and (2). This gives

$$M\ \equiv\ uM, \qquad M\ \equiv\ ux, \tag{3}$$

$$u\ \gamma_1\ M, \quad u\ \gamma_2\ M, \quad u\ \gamma_1\ M, \quad u\ \gamma_2\ x. \tag{4}$$

(It is not claimed that (3) has any semantic meaning!) Delete repetitions from (4) and look for γ_2-pairs with the same left side: we find only $\langle\, u\gamma_2 M,\ u\gamma_2 x\,\rangle$. Replace "$M$" by "$x$" throughout (3) and (4). This gives

$$x\ \equiv\ ux, \qquad x\ \equiv\ ux, \tag{5}$$

$$u\ \gamma_1\ x, \quad u\ \gamma_2\ x, \quad u\ \gamma_2\ x. \tag{6}$$

Delete repetitions and look again for γ-pairs with the same left side. There are none. Nor are there any with the same right side (see §3.6 later). Thus the replacement procedure halts, giving

$$x\ \equiv\ ux, \tag{7}$$

$$u\ \gamma_1\ x, \quad u\ \gamma_2\ x. \tag{8}$$

(Newman proves that this end-result is independent of the order in which the replacement-steps are made, except for alphabetic changes, such as "M" for "x"; in fact his uniqueness proof in [24, p.70] is a general one, covering $\mathrm{TA}_{\lambda\to}$ as well as NF.)

Finally, look for cycles in (8). There are none; we conclude that $u(ux)$ is typable.

3.4 (Searching for cycles) In λ-calculus a *cycle* is any sequence

$$X_1\ \gamma_{i_1}\ X_2,\quad X_2\ \gamma_{i_2}\ X_3,\quad X_3\ \gamma_{i_3}\ X_4,\quad \ldots\quad X_r\ \gamma_{i_r}\ X_1 \qquad (r \geq 1),$$

where $i_j \in \{1,2\}$ forall $j \leq r$. In a cycle, the type of X_1 properly contains that of X_2 (by the definitions of γ_1 and γ_2), which properly contains that of ... X_1. This is impossible.

3.5 (Replacing, when a pair has same LHS) If $u\ \gamma_1\ M$ and $u\ \gamma_1\ P$, as in (2), then the type received by u in any assignment to M has form $\sigma \to \tau$, and both M and P receive type τ. Hence M and P must receive the same type, and one of them can replace the other in the test.

3.6 (Replacing, when a pair has same RHS) In a list of γ-relationships such as (2), suppose there are two pairs with the same right-hand sides, with form

$$\begin{array}{ll} X\ \gamma_1\ Y, & X'\ \gamma_1\ Y, \\ X\ \gamma_2\ Z, & X'\ \gamma_2\ Z, \end{array} \tag{9}$$

for some terms X, X', Y, Z. Using §3.2, these say that if the type of X exists, it must have form

$$(\textit{type of } Z) \ \to\ (\textit{type of } Y),$$

and that the type of X' is exactly the same. Newman's test as stated in [24] allows X to be replaced by X' (or vice-versa) if a quadruple like (9) occurs.

Note that both lines of (9) are needed before we can say that the types of X and X' must be identical; if only one pair is present with the same right side, the test does not allow a replacement.

3.7 (Example) The term $\lambda x.ux$ is typable in $\mathrm{TA}^{res}_{\lambda\to}$.

Method: Let $M \ \equiv\ \lambda x.ux$. List the steps in the construction of M, thus:

$$M \ \equiv\ \lambda x.P, \qquad P \ \equiv\ ux. \tag{10}$$

Then write all the γ-relationships for these steps:

$$\begin{array}{ll} M\ \gamma_1\ P, & u\ \gamma_1\ P, \\ M\ \gamma_2\ x, & u\ \gamma_2\ x. \end{array} \tag{11}$$

These relationships form a quadruple like (9), so "M" can be replaced by "u" in (10) and (11). This gives

$$u \ \equiv\ \lambda x.P, \qquad P \ \equiv\ ux, \tag{12}$$

$$u\ \gamma_1\ P,\quad u\ \gamma_1\ P,\quad u\ \gamma_2\ x,\quad u\ \gamma_2\ x,$$

or, after deleting duplicates,

$$u\ \gamma_1\ P,\quad u\ \gamma_2\ x. \tag{13}$$

There is no cycle in (13), so $\lambda x.ux$ is typable.

3.8 (Difference between $\mathbf{TA}_{\lambda\to}$ and $\mathbf{TA}^{res}_{\lambda\to}$) The next example will show this difference clearly. In §§1.4 and 1.5 it was noted that the term $(\lambda x.x)(\lambda x.x)$ is typable in the former system but not the latter. When Newman's test is applied to this term, the result will agree with $\mathrm{TA}^{res}_{\lambda\to}$, not $\mathrm{TA}_{\lambda\to}$.

3.9 (Example) The term $(\lambda x.x)(\lambda x.x)$ is not typable in $\mathrm{TA}^{res}_{\lambda\to}$.

Method: Let $M \equiv (\lambda x.x)(\lambda x.x)$. List the steps in the construction of M, thus:

$$M \equiv PQ, \qquad P \equiv \lambda x.x, \qquad Q \equiv \lambda x.x. \tag{14}$$

Then write all the γ-relationships for these steps:

$$P\ \gamma_1\ M, \quad P\ \gamma_2\ Q, \quad P\ \gamma_1\ x, \quad P\ \gamma_2\ x, \quad Q\ \gamma_1\ x, \quad Q\ \gamma_2\ x. \tag{15}$$

Next, look in (15) for pairs of γ_1-statements with the same left side, or pairs of γ_2-statements similarly: we find only

$$\langle\, P\ \gamma_1\ M, \quad P\ \gamma_1\ x\,\rangle, \qquad \langle\, P\ \gamma_2\ Q, \quad P\ \gamma_2\ x\,\rangle.$$

Replace "M" by "x" in (14) and (15), and remove duplicates. This gives

$$x \equiv PQ, \qquad P \equiv \lambda x.x, \qquad Q \equiv \lambda x.x. \tag{16}$$

$$P\ \gamma_1\ x, \quad P\ \gamma_2\ Q, \qquad P\ \gamma_2\ x, \quad Q\ \gamma_1\ x, \quad Q\ \gamma_2\ x. \tag{17}$$

Then replace "Q" by "x" and remove duplicates. This gives

$$x \equiv Px, \qquad P \equiv \lambda x.x, \qquad x \equiv \lambda x.x. \tag{18}$$

$$P\ \gamma_1\ x, \quad P\ \gamma_2\ x, \quad x\ \gamma_1\ x, \quad x\ \gamma_2\ x. \tag{19}$$

There is a cycle in (19), namely $x\gamma_1 x$ (or $x\gamma_2 x$). Hence $(\lambda x.x)(\lambda x.x)$ is not typable in $\mathrm{TA}^{res}_{\lambda\to}$.

4 PT algorithms: history

As mentioned earlier, all the typability algorithms invented in the 1950s and '60s for $\mathrm{TA}_{\lambda\to}$ and its analogue in combinatory logic were also PT algorithms.[6]

A PT algorithm for the combinatory version of $\mathrm{TA}_{\lambda\to}$ was described informally by Curry and Feys in 1958 in [8, §§9B2–4], where several examples were worked out in detail. Roughly speaking, for a given term M the method was to list all M's subterm-occurrences with their types given as type-variables, to write out equations connecting these variables, and then solve these equations as generally as possible. Curry wrote this method out as a formal algorithm, with a proof of its correctness, in private notes [6] in 1966 and a paper [7] in 1969.

[6]This section is based on [14, §3A7] and [3, §8.5], whose information-sources include J. Kalman's [16] and D. Meredith's [20].

A different PT algorithm, which used J. A. Robinson's unification algorithm [28] to save itself some work, was made and proved correct by myself for combinatory logic in 1967 and published in [13, §3]. (Curry and I communicated during the writing of [7] and [13] but deliberately kept our approaches different.)

A Curry-style equation-solving PT algorithm for $\mathrm{TA}_{\lambda\to}$ was made, with a correctness proof, by James Hiram Morris in his doctoral thesis [22, Ch. 4, §E], independently of [7] and [13] and probably around 1967.

In the 1970s and '80s the programming language ML was developed by the group led by Robin Milner in Edinburgh, and it included a type-assignment system for λ-calculus augmented by the operator *let.* Milner described a PT algorithm for ML in 1978 [21, §4.1, Algorithm W]; it was unification-based like that in [13], though it was invented independently. It was rewritten and extended in [10], and its correctness was proved by Luis Damas in [9] in 1984.

After 1980, typability and PT algorithms were made for various more complex systems, some depending on pre-existing unification algorithms and some being self-sufficient: see J. Tiuryn's survey [29]. (For more complex systems the definition of PT is usually more complex than for simple types.)

In the opposite direction, if one is interested in just the core part of a PT algorithm rather than the whole algorithm, one finds that the history of this core extends back some way, but outside the field of type theory. Under the propositions-as-types correspondence, simple types whose atoms are variables become propositional formulas, and a combinator M whose PT is τ becomes a proof of τ by the propositional rule called *condensed detachment.*

Roughly speaking, the condensed detachment rule corresponds to the construction of the PT of a term PQ from those of P and Q.[7] It was invented by Carew Meredith in Dublin and was first used in print (but not formally defined) in [17, §9] in 1957. Meredith never published a formal statement of it, but his cousin and collaborator David Meredith was taught the rule by him in 1954, wrote it up as a program, and ran it on the computer UNIVAC1 in the U.S.A. in 1957. (See [16, §4].) Thus, by 1957 the key step in a PT algorithm had not only been stated formally, but had been implemented on a machine.

Looking further back: before 1939 an argument very like an analysis of a particular case of condensed detachment had been used by Jan Łukasiewicz, [18, Engl. edn., p.276]. He attributed his method to Tarski, and in a 1930 work [19, Engl. edn., p.44] a method was mentioned which might perhaps be the one to which he referred. But the actual method was not described or even sketched, so the latter reference must be treated as doubtful. Further comments are in [16, §4].

References

[1] J. Alves-Foss and M. R. Holmes. The Watson theorem prover. *Journal of Automated Reasoning*, 26:357–408, 2001.

[7]For a formal statement of this rule, see [16, §2] or [14, §7D]. The rule has similarities with Robinson's resolution rule, and pre-dates it.

[2] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2, Background: Computational Structures*, pages 117–309. Clarendon Press, Oxford, England, 1992.

[3] F. Cardone and J. R. Hindley. History of Lambda-calculus and Combinatory Logic. In D. Gabbay and J. Woods, editors, *Handbook of the History of Logic*. Elsevier, Amsterdam, to appear 2007.

[4] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[5] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the U.S.A.*, 20:584–590, 1934.

[6] H. B. Curry. Technique for evaluating principal functional character. Note dated March 17th. 1966, 5 pages, filed as T660317A in Curry archive, Pennsylvania State Univ. , U.S.A., 1966.

[7] H. B. Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23:83–92, 1969.

[8] H. B. Curry and R. Feys. *Combinatory Logic, Volume I.* North-Holland Co., Amsterdam, 1958. (3rd edn. 1974).

[9] L. Damas. *Type Assignment in Programming Languages.* PhD thesis, Computer Science Dept., Univ. Edinburgh, U.K., 1984.

[10] L. Damas and R. Milner. Principal type-schemes for functional programming languages. In *Ninth Annual A.C.M. Symposium on the Principles of Programming Languages (POPL)*, pages 207–212. Association for Computing Machinery, New York, 1982.

[11] C. Dwork, P. C. Kanellakis, and J. C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1:35–50, 1984.

[12] T. E. Forster. *Set Theory with a Universal Set.* Clarendon Press, Oxford, 1995. 1st edn. was 1992.

[13] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[14] J. R. Hindley. *Basic Simple Type Theory.* Cambridge Univ. Press, England, 1997.

[15] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-calculus.* Cambridge Univ. Press, England, 1986.

[16] J. Kalman. Condensed detachment as a rule of inference. *Studia Logica*, 42:443–451, 1983.

[17] E. J. Lemmon, C. A. Meredith, D. Meredith, A. N. Prior, and I. Thomas. Calculi of pure strict implication. Informally distributed, 1957. Publ. 1969 in *Philosophical logic*, ed. by Davis and Hockney and Wilson, D. Reidel Co., Dordrecht, Netherlands, pp. 215–250.

[18] J. Łukasiewicz. Der äquivalenzenkalkul. *Collectanea Logica*, 1:145–169, 1939. Journal vol. never appeared. Engl. transl: *The equivalential calculus*, in *Jan Łukasiewicz Selected Works*, ed. by L. Borkowski, North-Holland Co., Amsterdam, 1970, pp. 250–277.

[19] J. Łukasiewicz and A. Tarski. Untersuchungen über den Aussagenkalkül. *Comptes Rendus des Séances de la Societé des Sciences et des Lettres de Varsovie*, 23:30–50, 1930. Engl. transl: *Investigations into the sentential calculus*, in *Logic, Semantics, Metamathematics*, by A. Tarski, Clarendon Press, Oxford, 1956, pp. 38–59.

[20] D. Meredith. In memoriam Carew Arthur Meredith. *Notre Dame Journal of Formal Logic*, 18:513–516, 1977.

[21] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[22] J. H. Morris. *Lambda-calculus Models of Programming Languages*. PhD thesis, Massachusetts Inst. Technology, Cambridge, Mass., U.S.A., 1968.

[23] M. H. A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

[24] M. H. A. Newman. Stratified systems of logic. *Proceedings of the Cambridge Philosophical Society*, 39:69–83, 1943. Reviewed by A. Church in J. Symbolic Logic 9 (1944), pp. 50–52.

[25] M. H. A. Newman and A. M. Turing. A formal theorem in Church's theory of types. *Journal of Symbolic Logic*, 7:28–33, 1942.

[26] W. V. Quine. New foundations for mathematical logic. *American Mathematical Monthly*, 44:70–80, 1937.

[27] A. Rezus. *A Bibliography of Lambda-Calculi, Combinatory Logics and Related Topics*. Mathematisch Centrum, 413 Kruislaan, Amsterdam, 1982. ISBN 90–6196234–X.

[28] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.

[29] J. Tiuryn. Type inference problems: a survey. In B. Rovan, editor, *Mathematical Foundations of Computer Science 1990*, volume 452 of *Lecture Notes in Computer Science*, pages 105–120. Springer-Verlag, Berlin, 1990.

[30] J. Tyszkiewicz. Complexity of type inference in finitely typed lambda calculus. Master's thesis, Univ. Warsaw, Poland, 1988. (Unpublished).