# Notes on the history of type inference

Fritz Henglein
DIKU, University of Copenhagen

2010-01-29

Newman's typability algorithm Newman [1943] for Quine's Type Theory (Quine [1937]) and an implicit version of Church's Theory of Simple Types (Church [1940])—a Curry-style formulation without explicit types (Curry [1934])—was recently rediscovered by Hindley [2008].

The algorithm is a fascinating precursor to constraint-based type inference and program analysis techniques, which have been developed in the late 80s and onwards for both theoretical and practical purposes. Using terminology from constraint-based type inference Newman's algorithm can be described as follows when applied to simple typing:[1]

1. Ensure that all bound and free variables in the subject term $M$ are named apart. Let each subterm $X$ of $M$ be associated with a unique type variable $\alpha_X$. In Newman's description $\alpha_X$ is identified by the subterm $X$ itself.

2. Generate constraints for each subterm $Z$ of $M$:

   (a) For $Z \equiv XY$ Newman generates the constraints $X\ \gamma_1\ Z$ and $X\ \gamma_2\ Y$, which corresponds to the (type-)equational constraint $\alpha_X = \alpha_Y \to \alpha_Z$.

   (b) For $Z \equiv \lambda x.U$ generate $Z\ \gamma_1\ U$ and $Z\ \gamma_2\ x$, corresponding to $\alpha_Z = \alpha_x \to \alpha_U$.

   It can be observed that whenever there exists $X\ \gamma_1\ Y$ in Newman's constraints then they also contain $X\ \gamma_2\ Z$ for some $Z$, and this is preserved throughout the subsequent constraint simplification process. A —decidedly revisionist—interpretation of this observation is that the type constraint notation subsequently adopted in constraint-based type inference syntactically incorporates this duality by combining them into a single construct: Define $\alpha_X = \alpha_Y \to \alpha_Z$ if $X\ \gamma_1\ Z$ and $X\ \gamma_2\ Y$ in Newman's formulation.

3. Simplify the set of constraints as follows:

[1] This description reflects my own understanding based on Hindley's presentation (Hindley [2008]).

(a) (Unification closure) If the constraints contain distinct $X\ \gamma_1\ V$ and $X\ \gamma_1\ Z$, substitute $Z$ for $V$ in the constraints. If they contain distinct $X\ \gamma_2\ U$ and $X\ \gamma_2\ Y$, substitute $Y$ for $U$. Because of the above invariant, these two rules can be combined into a single type-equational rule: If there are distinct constraints $\alpha_X = \alpha_Y \to \alpha_Z$ and $\alpha_X = \alpha_U \to \alpha_V$ substitute $\alpha_Y$ for $\alpha_U$ and $\alpha_Z$ for $\alpha_V$.

(b) (Congruence closure) If the constraints contain

$$X\ \gamma_1\ Y, X'\ \gamma_1\ Y, X\ \gamma_2\ Z, X'\ \gamma_2\ Z$$

for distinct $X, X'$ then substitute $X$ for $X'$ in the constraints. The corresponding type-equational rule is: If $\alpha_X = \alpha_Y \to \alpha_Z$ and $\alpha_{X'} = \alpha_Y \to \alpha_Z$ occur in the constraints, substitute $\alpha_{X'}$ by $\alpha_X$.

(c) (Cycle test) If Newman's constraints, interpreted as a directed graph, contain a cycle, terminate with failure. (With only one type constructor, $\to$, which Newman's algorithm is implicitly formulated for, this is the only possible cause of failure. With additional type constructors a constructor clash failure rule needs to be added.) If no closure rule applies and the constraints contain no cycle, terminate with success.

The discovery of Newman's algorithm in the literaure is due to Hindley (Hindley [2008]). Hindley's presentation is short, transparent and insightful: It quickly lets the reader understand that Newman's is a constraint-based technique where type-equational constraints of the form $\alpha = \beta \to \gamma$ are coded by $\alpha\ \gamma_1\ \gamma$ and $\alpha\ \gamma_2\ \beta$, but otherwise processed as in unification closure, with additional congruence closure steps.

There is a substantial amount of work on constraint-based type inference and program analysis, which has been published starting in the late 80s and early 90s. After Wand's exposition of simple type inference (Wand [1987]), Henglein showed how to reduce let-polymorphism and polymorphic recursion to semi-unification, which are systems of equational and instantiation constraints (Leiß [1987], Henglein [1988], Kapur et al. [1988], Henglein [1989], Kfoury et al. [1990a;b], Kapur et al. [1991], Henglein [1993]). Existence of principal types and their computation by most general semi-unifiers with detailed proofs can be found in Henglein [1989]. A simple special case of this is simple type inference: Constraint simplification degenerates to unification closure (as above) and most general semi-unifiers to most general unifiers, which yield principal types for the original type inference problem (for simple typing the even stronger property of principal type *derivations*). An interesting consequence of that work is that the congruence closure steps in Newman's algorithm are unnecessary: The rules can be staged by first applying unification closure only, and checking for cycles at the very end. At any given point congruence closure steps may be interspersed. They do not affect the final result, but may be useful for compressing the constraint graph. Constraint-rewriting formulations of unification and their algebraic have been known since the 80s (Eder [1985], Martelli and Montanari [1982]).

Rank-1 bounded System F is known to be a cosmetic extension of let-polymorphism since typability in either case can be characterized by Tofte's "algorithmic" type inference rules where generalization only takes place at let-occurrences, and instantation only at variable occurrences (Tofte [1990]). (More interesting is rank-2-bounded System F (Kfoury and Tiuryn [1992], Henglein and Mairson [1994], Kfoury and Wells [1994]), even though it also can be reduced to let-polymorphism.) Adding explicitly typed constants is trivial if only type schemes (let-polymorphic types) or System F types of maximum rank 1 are used. A much harder problem is the case of typability with free variables of unknown type (Schubert [1998]).

# References


Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5 (2):56–68, 1940.

H.B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20:584–590, 1934.

E. Eder. Properties of substitutions and unifications. *J. Symbolic Computation*, 1:31–46, 1985.

F. Henglein. Type inference and semi-unification. In *Proc. ACM Conf. on LISP and Functional Programming*. ACM, ACM Press, July 1988.

Fritz Henglein. *Polymorphic Type Inference and Semi-Unification*. PhD thesis, Rutgers University, May 1989. Available as NYU Technical Report 443, May 1989, from New York University, Courant Institute of Mathematical Sciences, Department of Computer Science, 251 Mercer St., New York, N.Y. 10012, USA.

Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):253–289, April 1993. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/169701.169692.

Fritz Henglein and Harry Mairson. The complexity of type inference for higher-order typed lambda calculi. *Journal of Functional Programming (JFP)*, 4(4): 435–477, October 1994.

J. Roger Hindley. M. H. Newman's typability algorithm for lambda-calculus. *J. Log. Comput.*, 18(2):229–238, 2008.

D. Kapur, D. Musser, P. Narendran, and J. Stillman. Semi-unification. In *Proc. Foundations of Software Technology and Teoretical Computer Science*, pages 435–454. Springer, December 1988. Lecture Notes in Computer Science, Vol. 338.

Deepak Kapur, David Musser, Paliath Narendran, and Jonathan Stillman. Semi-unification. *Theoretical Computer Science*, 81(2):169 – 187, 1991. ISSN 0304-3975. doi: DOI: 10.1016/0304-3975(91)90189-9. Based on paper presented at Conf. on Foundations of Software Technology and Teoretical Computer Science (FST-TCS), December '88, Springer Lecture Notes in Computer Science, Vol. 338.

A. Kfoury and J. Tiuryn. Type reconstruction in finite rank fragments of the second-order $\lambda$-calculus. *Information and Computation*, 98(2):228–257, June 1992.

A. Kfoury, J. Tiuryn, and P. Urzyczyn. ML typability is DEXPTIME-complete. In *Proc. 15th Coll. on Trees in Algebra and Programming (CAAP), Copenhagen, Denmark*, pages 206–220. Springer, May 1990a. Lecture Notes in Computer Science, Vol. 431.

A. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. In *Proc. 22nd Annual ACM Symp. on Theory of Computation (STOC), Baltimore, Maryland*, pages 468–476, May 1990b.

A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order lambda-calculus. In *LISP and Functional Programming*, pages 196–207, 1994. doi: http://doi.acm.org/10.1145/182409.182456.

H. Leiß. On type inference for object-oriented programming languages. In *Proc. 1st Workshop on Computer Science Logic*, volume 329 of *Lecture Notes Computer Science*, pages 151–172. Springer-Verlag, October 1987.

A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, April 1982.

M. H. A. Newman. Stratified systems of logic. *Mathematical Proceedings of the Cambridge Philosophical Society*, 39(02):69–83, 1943. doi: 10.1017/S0305004100017722.

W. V. Quine. New foundations for mathematical logic. *The American Mathematical Monthly*, 44(2):70–80, February 1937. ISSN 00029890. doi: http://www.jstor.org/stable/2300564.

Aleksy Schubert. Second-order unification and type inference for Church-style polymorphism. In *POPL*, pages 279–288, 1998. doi: http://doi.acm.org/10.1145/268946.268969.

M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, November 1990.

M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, X:115–122, 1987.