Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a preprint version which may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/92446

Please be advised that this information was generated on 2021-04-17 and may be subject to change.

The correctness of Newman's typability algorithm and some of its extensions

Herman Geuvers[a,b], Robbert Krebbers[a]

[a] *Institute for Computing and Information Science*
Faculty of Science, Radboud University Nijmegen, The Netherlands
[b] *Faculty of Mathematics and Computer Science*
Eindhoven University of Technology, The Netherlands

Abstract

We study Newman's typability algorithm [14] for simple type theory. The algorithm originates from 1943, but was left unnoticed until [14] was recently rediscovered by Hindley [10]. The remarkable thing is that it decides typability without computing a type. We give a modern presentation of the algorithm (also a graphical one), prove its correctness and show that it implicitly does compute the principal type. We also show how the typing algorithm can be extended to other type constructors. Finally we show that Newman's algorithm actually includes a unification algorithm.

Keywords: Simply typed lambda calculus, Unification, Typing algorithms

1. Introduction

A type checking algorithm for simple type theory solves the problem whether an untyped λ-term M can be given a type according to the rules of simple type theory. As usual, we distinguish between the *type checking problem*: $M : \sigma$ (does the term M have type σ?) and the *type synthesis problem*: M : ? (compute a type for the term M). It is a classic result that for simple type theory, these problems are both decidable [9, 12]. The solution amounts to computing the *principal type* of M and then to check whether the given type σ is an instance of it. All known type checking algorithms for λ-calculus compute a principal type, using a unification algorithm. It is hard to imagine that one could do something fundamentally different. However, there is a (seemingly) different way to decide typing, due to Newman, already in 1943 [14], and recently reviewed by Hindley [10]. The strategy of Newman's algorithm is very different from present day algorithms, because it only decides whether a term is typable and does not compute its principal type. This looks strange, because the most common way to decide whether a λ-term is typable is trying to assign a type to it.

Email addresses: herman@cs.ru.nl (Herman Geuvers), mail@robbertkrebbers.nl (Robbert Krebbers)

In this paper, we describe and analyze Newman's algorithm (Section 3) and we show that it (implicitly) *does* compute a principal type. The way it computes it is actually quite close to other constraint based typing algorithms for simple type theory, for example Wand's algorithm [21]. We prove that Newman's algorithm is correct and discuss the correspondence with Wand's algorithm (Section 4).

Before doing that, we describe Newman's algorithm directly as a manipulation of the term-graph that represents the λ-term (Section 2). Following the method described by Newman, we manipulate the term-graph using a sort of *term-graph reduction* and if no further reductions are possible, we check whether a certain relation on the nodes is cyclic. The original term is typable if and only if the relation is acyclic. The correspondence between the graphical description and Newman's is left implicit, but will be clear to the reader. It is more interesting to extend the graphical description to include other type constructions, like *product types* and *weakly polymorphic types* (Section 5).

An important part of a typing algorithm is to solve unification problems. In the presentation by Wand [21], this is nicely singled out: one first creates a set of equations, which are then solved by a (standard) unification algorithm (for example Robinson's unification algorithm [19]). Looking back at Newman's algorithm, it can be observed that Newman does something similar: he creates a set of equations, which is then modified. To emphasize this, we also give a description of unification "à la Newman" (Section 6).

1.1. Historical background

Newman's starting point was Quine's proposal [18] for a new logical system called "New Foundations" in 1937. Newman developed an algorithm to decide typability for that system, but it was also able to decide typability for other type systems, for example the Principia Mathematica and Church's simply typed λ-calculus [5]. The latter connection is not detailed in Newman's paper, maybe because Church's paper had only appeared recently and there was more interest in Quine's New Foundations at the time. To be precise, Newman actually uses a system that is now better known as λ-calculus à la Curry [3], where there is no type information in the λ-term; λ-calculus à la Church has type information in the term and then the issue of typability is simple.

In 1944 Church reviewed Newman's paper [6], but his review was more like a summary of Newman's paper concluding with:

> The reader's first impression of Newman's paper may be that the machinery introduced is heavy in comparison with the results obtained. The value of the paper is in fact difficult to estimate at present, as this will depend on the extent to which results obtained in the future by Newman's methods justify the weight of machinery.

We could say that Newman was ahead of his time, since there was no actual need for a typing algorithm until the 1950s and 1960s. But at that point type theorists seem to have ignored or forgotten about Newman's work and invented their own algorithm [10].

1.2. Simple type theory and Newman's type system

A typed λ-calculus can be presented à la Church or à la Curry [3, 7]. In Church-style, a bound variable is typed in the λ-abstraction, for example $\lambda x : \alpha.\lambda y : \alpha \to \beta.y\,x : \alpha \to (\alpha \to \beta) \to \beta$. Furthermore, a free variable is typed in the context, for example $x : \alpha \vdash \lambda y : \alpha \to \beta.y\,x : (\alpha \to \beta) \to \beta$. Alternatives to this syntax are found in the literature, for example where free variables carry their type as an annotation instead of in the context, for example $\lambda y : \alpha \to \beta.y\,x^{\alpha} : (\alpha \to \beta) \to \beta$. Or, if we also let bound variables carry their type as an annotation: $\lambda y^{\alpha\to\beta}.y^{\alpha\to\beta}\,x^{\alpha} : (\alpha \to \beta) \to \beta$. The crucial point is that type information for the variables is fixed and can be computed via a simple lookup. Therefore, type checking and type-synthesis are trivial: types can be simply read off from the term.

In this paper a system à la Curry is studied because most functional programming languages deal with this and that is also what Newman's algorithm is about. Now the variables do not carry any type information and the question is whether, given an untyped λ-term M, we can decide whether M is typable (and then give a type for M) or M is not typable. In this section we summarize the main definitions and introduce notations for the rest of the paper. For an extensive discussion concerning these notions see, for example [3, 7].

Definition 1.1. *The* terms in $\lambda\!\!\to$ à la Curry *(typically N, M, P, ...) are inductively defined as follows.*

$$\Lambda ::= \mathsf{Var} \mid (\Lambda\Lambda) \mid (\lambda\mathsf{Var}.\Lambda)$$

Here Var *ranges over variables (typically x, y, z, ...). The* types of $\lambda\!\!\to$ *(typically σ, τ, φ, ...) are inductively defined as follows.*

$$\mathsf{Type} ::= \mathsf{TVar} \mid (\mathsf{Type} \to \mathsf{Type})$$

Here TVar *ranges over type variables (typically α, β, γ, ...).*

A *context* (typically Γ or Δ) is a finite set of variable declarations $x : \sigma$ ($x \in \mathsf{Var}$, $\sigma \in \mathsf{Type}$), where all variables are distinct. We write $\Delta(x)$ to denote the type that Δ assigns to x. The notions of *free variables* $\mathrm{FV}(M)$ of a term M, and *bound variables* $\mathrm{BV}(M)$ of a term M, are defined as usual, where λ binds variables. Similarly, we use the notion of *free type variables* $\mathrm{FTV}(\sigma)$ of a type σ, to denote the set of type variables occurring in σ.

We now give the deduction rules for assigning types to terms in simple type theory. This is the "modern" way of inductively describing the set of well-typed terms of a type. Newman does not explicitly define the well-typed terms.

Definition 1.2. *A* $\lambda\!\!\to$-typing judgment $\Gamma \vdash M : \rho$ *denotes that a λ-term M has type ρ in context Γ. The derivation rules for deriving such a judgment are as follows.*

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \qquad \frac{\Gamma, x : \sigma \vdash P : \tau}{\Gamma \vdash \lambda x.P : \sigma \to \tau}$$

(a) Variable (b) Application (c) Abstraction

We write $\nvdash M$ if no context Γ and type ρ exist such that $\Gamma \vdash M : \rho$.

Definition 1.3. *The* Type Synthesis (TSP) *or* Type Assignment Problem (TAP) *is given a λ-term M and a context Γ to find a type ρ such that $\Gamma \vdash M : \rho$.*

Newman puts a restriction on the λ-terms: he only considers terms M for which $\mathrm{FV}(M) \cap \mathrm{BV}(M) = \emptyset$ and for every $x \in \mathrm{BV}(M)$ there is exactly one λx-abstraction in M. So the bound and free variables are distinct and no variable has a duplicate binder. Nowadays, this "restriction" goes under the name of *Barendregt convention* [4] which states that, whenever we consider a number of λ-terms, we may always assume that all free variables are different from the bound ones and that all bound variables are distinct. We can safely assume this, because we can always rename bound variables to get a term that is α-equivalent to it. In the time Newman wrote his article, these issues had not yet been clarified completely, so Newman simply only considered terms that satisfy the Barendregt convention. It should be noted, however that the way Newman presented his algorithm, the actual names of variables *do really* matter.

1.3. Principal types and most general unifiers

Well-known typing algorithms, like Hindley-Milner's [9, 12] or Wand's [21], decide whether a term is typable by computing its principal type. Also, these algorithms rely on the computation of a *most general unifier*. While we postpone the discussion of Wand's algorithm until Section 4.2, we will define the notion of a most general unifier and a principal type now.

Definition 1.4. *Given a set of type equations $E = \{\rho_1 \doteq \sigma_1, \ldots, \rho_n \doteq \sigma_n\}$ and a substitution δ then δ* is a unifier of *E, notation $\delta \models E$, if $\delta(\rho) = \delta(\sigma)$ for each $\rho \doteq \sigma \in E$. We write $\not\models E$ (E is not unifiable) if no substitution δ exists such that $\delta \models E$.*

A substitution δ is the most general unifier *of E, notation $\delta \models_{\mathrm{mgu}} E$, if $\delta \models E$ and for all τ, if $\tau \models E$ then there exists a substitution ν such that $\tau = \nu \circ \delta$. (Any other unifier of E is an* instance *of δ.)*

Definition 1.5. *Given a λ-term M, a context Γ and a type τ, then $\langle \Gamma, \tau \rangle$ is a* principal pair *of M if:*

1. $\Gamma \vdash M : \tau$
2. $\Delta \vdash M : \rho \implies (\exists \sigma : \mathsf{TVar} \to \mathsf{Type} \,.\, \Delta \supseteq \sigma(\Gamma) \;\wedge\; \rho = \sigma(\tau))$

Definition 1.6. *Given a closed λ-term M and a type τ, then τ is a* principal type *of M iff $\langle \emptyset, \tau \rangle$ is a principal pair of M.*

The goal of a typing algorithm is to compute a principal type for a closed term and a principal pair for an open term. For $\lambda\!\to$, principal types (respectively principal pairs) exist and can be computed. By definition a principal pair (respectively principal type) is unique up to isomorphism. Here $\langle \Gamma, \tau \rangle$ and $\langle \Delta, \rho \rangle$ are *isomorphic*, notation $\langle \Gamma, \tau \rangle \cong \langle \Delta, \rho \rangle$, if there exist substitutions σ_1 and σ_2 such that $\Delta = \sigma_1(\Gamma)$, $\rho = \sigma_1(\tau)$, $\Gamma = \sigma_2(\Delta)$ and $\tau = \sigma_2(\rho)$.

1.4. Some basic notions from rewriting

In this paper we will be using the notions of *confluence* and *strong normalization*, not so much for the simple type theory, but for auxiliary relations that will be defined in what follows. We use the standard terminology of [20, 4, 1], which we recall here.

Definition 1.7. *Let* $\rightarrowtriangle$ *be a binary relation on a set* A *and let* $\twoheadrightarrow$ *denote the transitive reflexive closure of* $\rightarrowtriangle$.

- *The relation* $\rightarrowtriangle$ satisfies the diamond property *if for every* $a, b, c \in A$*, if* $a \rightarrowtriangle b$ *and* $a \rightarrowtriangle c$*, then there is a* $d \in A$ *such that* $b \rightarrowtriangle d$ *and* $c \rightarrowtriangle d$.

- *The relation* $\rightarrowtriangle$ *is* confluent *if* $\twoheadrightarrow$ *satisfies the diamond property. It is* locally confluent *if for every* $a, b, c \in A$*, if* $a \rightarrowtriangle b$ *and* $a \rightarrowtriangle c$*, then there is a* $d \in A$ *such that* $b \twoheadrightarrow d$ *and* $c \twoheadrightarrow d$.

- *An element* $a \in A$ *is* in $\rightarrowtriangle$-normal form *if there is no* $b \in A$ *for which* $a \rightarrowtriangle b$.

- *The relation* $\rightarrowtriangle$ *is* strongly normalizing *or* terminating *if for every element* $a \in A$ *there is no infinite* $\rightarrowtriangle$*-sequence starting from* a.

Lemma 1.8. (Newman's Lemma for abstract rewriting systems [13]). *Let* $\rightarrowtriangle$ *be a binary relation on a set* A*. If* $\rightarrowtriangle$ *is strongly normalizing and locally confluent, then* $\rightarrowtriangle$ *is confluent.*

2. Newman graphs

We define Newman's algorithm for $\lambda\!\to$. It is derived from [14] and [10], although the presentation is fundamentally different. First of all, we use a term-graph notation for λ-terms as indicated in the following definition.

Definition 2.1. *Given an untyped* λ*-term* M*, we define the* term-graph of M, Tgraph(M) *in the usual way. We replace an application by an @-node and a* λ*-abstraction* $\lambda x.M$ *by a* λ*-node where we also have a pointer to* x *as a "shared leaf", as indicated in Figure 1 in the left and middle drawing*[1].

[1] This is also known as a DAG (Directed Acyclic Graph) in which all occurrences of variables are shared [2].

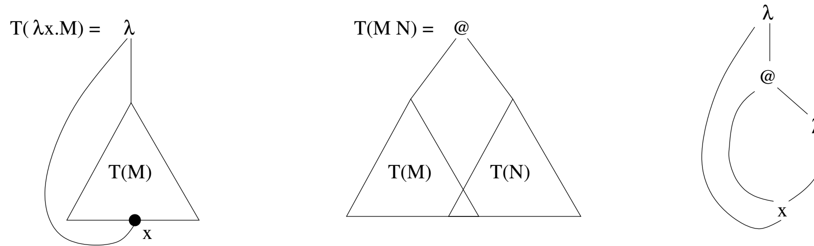
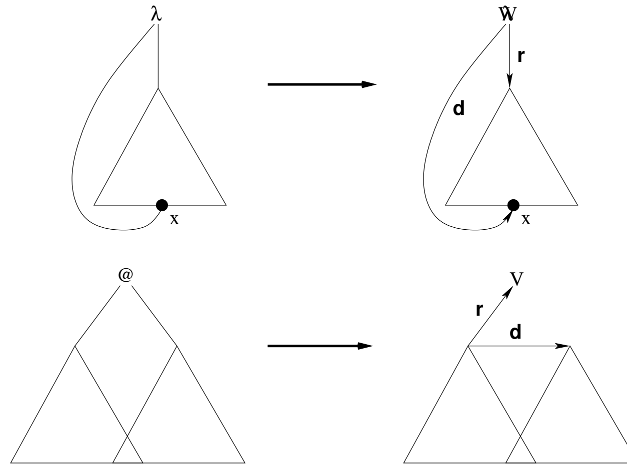


Figure 1: Translating a term to a term-graph.

As an example of a translation of a term to a term-graph, we have depicted the term-graph of $\lambda x.x(\lambda y.x)$ on the right in Figure 1.

Now we translate the term-graph of the term M, Tgraph(M), into the *Newman graph of* M, Ngraph(M). Of course, we could have defined Ngraph(M) directly from M, but the present definition more clearly emphasizes the relation between the two.

Definition 2.2. *Given an untyped λ-term M with* Tgraph(M)*, the* Newman graph *of M,* Ngraph(M) *is defined as depicted in the figures below. In the translation we ignore the @ and λx information in the nodes. Instead we give all nodes a unique label (typically U, V, X, Y, Z,...), except for the variable nodes for which we use the variable name itself as a label. Furthermore, we replace edges by labeled arrows and remove others.*



The intuitive meaning of the arrows should be clear:

$X \xrightarrow{\mathsf{d}} Y$	iff	the domain of the type of X is the type of Y
$X \xrightarrow{\mathsf{r}} Y$	iff	the range of the type of X is the type of Y

(In simple type theory, when $\sigma = \rho \to \tau$, then ρ is called the domain of σ and τ the range of σ.)

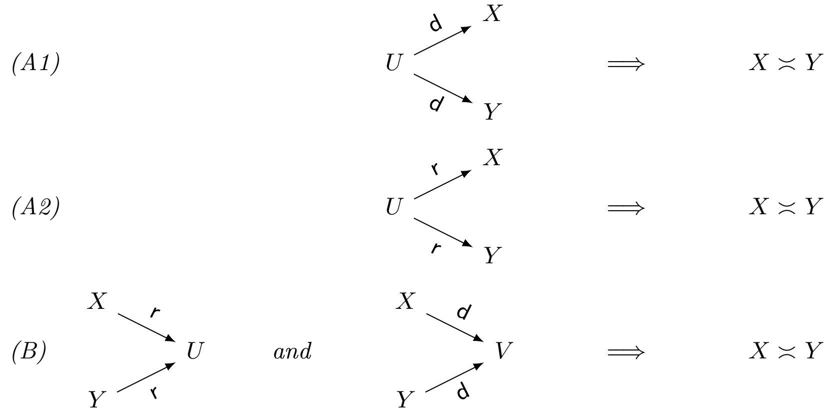
We will now *reduce* the Newman graph. A reduction step is performed by *joining two congruent nodes*, so we first define when two nodes X and Y are congruent.

Definition 2.3. *Given a Newman graph G, the* congruence relation $X \asymp_G Y$ *between nodes X and Y is inductively defined as follows.*

(A1) If $U \xrightarrow{\mathsf{d}} X$ and $U \xrightarrow{\mathsf{d}} Y$, then $X \asymp_G Y$
(A2) If $U \xrightarrow{\mathsf{r}} X$ and $U \xrightarrow{\mathsf{r}} Y$, then $X \asymp_G Y$
(B) If $X \xrightarrow{\mathsf{d}} U$, $X \xrightarrow{\mathsf{r}} V$ and $Y \xrightarrow{\mathsf{d}} U$, $Y \xrightarrow{\mathsf{r}} V$, then $X \asymp_G Y$

The intuitive meaning of $X \asymp_G Y$ is "the terms represented by X and Y have the same type". (To be precise: the terms represented by X and Y have the same type *if the term represented by G is typable*. In case the term represented by G is not typable, the terms represented by X and Y may not be typable either. Note that in any case $X \asymp_G Y$ is well defined.) Combined with the intuitive understanding of $\xrightarrow{\mathsf{d}}$ and $\xrightarrow{\mathsf{r}}$, this clarifies Definition 2.3. For example clause (A1) of Definition 2.3 then reads: if the domain of the type of U is both the type of X and the type of Y, then X and Y have the same type.

When writing $X \asymp_G Y$ we will often suppress G, and write $X \asymp Y$. The definition can be depicted graphically as follows.



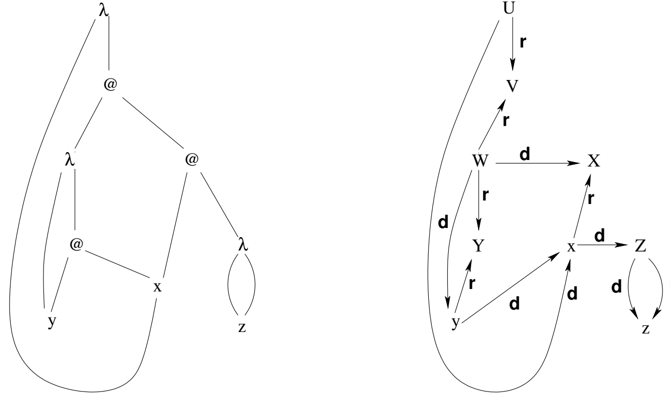
Remark 2.4. *Although Newman also considers clause (B) in his definition corresponding to $X \asymp_G Y$, it should be noted that for type checking, only clauses (A1) and (A2) are needed. So, Theorem 2.8 also holds if we restrict to the reduction steps that arise from joining nodes that are equal according to clauses (A1) and (A2). This will be proven in Section 3, Corollary 3.22.*

Definition 2.5. *A reduction step on Newman graphs, $G_1 \twoheadrightarrow G_2$, is defined by joining two nodes X and Y from G_1 for which we have $X \asymp Y$. So, if $X \asymp Y$,*

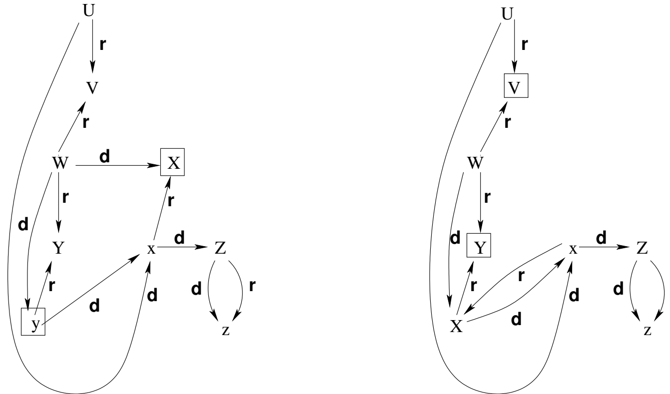
we join the nodes and all incoming and outgoing nodes are redirected to the new joint node, that we name either X or Y.

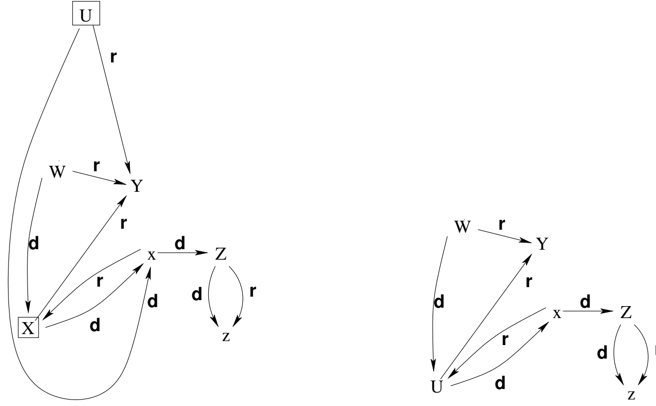
We treat an example to show the definitions at work.

Example 2.6. *Consider the λ-term $M \equiv \lambda x.\,(\lambda y.\,y\,x)\,(x\,(\lambda z.\,z))$. We depict its term-graph* $\mathrm{Tgraph}(M)$ *and its Newman graph* $\mathrm{Ngraph}(M)$*.*



We now join congruent nodes, using the congruence relation $\asymp$ as defined in Definition 2.3. In the first step, we observe the two congruent nodes X and y, which we indicate by a box. We join them and then we observe the congruent nodes V and Y, that we join. The rest of the example should be self-explanatory: in every step we join the congruent nodes and we indicate the two congruent nodes that we join in the next step.





In the final graph of the example, no reduction is possible anymore. The crucial point is now that it contains a *cycle* and therefore – according to Newman – the original term is not typable.

Definition 2.7. *Given a Newman graph G, the graph G is in* normal form *if no distinct nodes X and Y of G are congruent. Moreover the graph G is* stratified *in case the transitive closure of $\xrightarrow{\mathsf{d}} \cup \xrightarrow{\mathsf{r}}$ contains no cycles.*

The reduction $\twoheadrightarrow$ is strongly normalizing, simply because the number of nodes decreases, so we will always find a normal form. That the normal form of a Newman graph is unique, up to renaming of the labels, will be proven in Section 3. In Section 4 we will also prove that the normal form of $\mathrm{Ngraph}(M)$ is stratified if and only if M is typable in $\lambda{\to}$. This is proven by showing how the reduction of Newman graphs implicitly keeps track of the type information of the term and thus computes the principal type of a term.

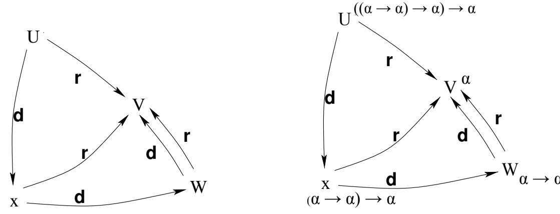
Theorem 2.8. *The reduction $\twoheadrightarrow$ is strongly normalizing and confluent*[2]*. Moreover, the normal form of the Newman graph of M is stratified if and only if M is typable in $\lambda{\to}$.*

Proof. The first is by Lemma 3.10 and 3.15, the second by Theorem 4.17. $\square$

Newman's algorithm can be extended to other type constructions in simple type theory, like product types and weakly polymorphic types. This will be shown in Section 5. For now, we indicate how we can use Newman's algorithm to "read off" the principal type of M from the normal form of the Newman graph of M. This will be detailed in the Section 4, but an example should be very explanatory.

Example 2.9. *Consider the λ-term $P \equiv \lambda x.x(\lambda y.x(\lambda z.y))$. If we reduce the Newman graph of P to normal form we obtain the graph on the left, where U corresponds to the root node of the term P.*

[2]See Definition 1.7 for a precise definition of these notions.



The graph contains no cycles. We compute the type "at node U" by labeling all nodes without outgoing edges by a type variable. In the graph above this is just node V that we label by α. Then we construct the types at all nodes in the obvious way: node x has type $(\alpha \to \alpha) \to \alpha$ because $\xrightarrow{\mathsf{d}}$ points to a node with label $\alpha \to \alpha$ and $\xrightarrow{\mathsf{r}}$ points to a node labeled with α. So, the principal type of the term P is $((\alpha \to \alpha) \to \alpha) \to \alpha$.

One may suspect that a normal form always has a simple cycle, involving only one or two nodes. However, this is not the case: if one considers the Newman graph of the term $N \equiv \lambda x.x(\lambda y.y(\lambda z.x))$, one can observe that only one reduction step is possible and then a normal form is obtained with a cycle of length 4 (hence N is not typable).

3. Newman's algorithm for λ→

In this section we present Newman's algorithm in a way close to Newman's original presentation [14]. Newman has described his algorithm in a very abstract and general way in order to apply it to various type systems. However, we are merely interested in λ→, therefore we will specialize our definitions for the case of λ→. Our description of Newman's algorithm has a lot in common with Hindley's [10], however our description is more extensive and we will prove some basic properties.

Newman's algorithm is basically a rewrite system. It starts with a set of equations presenting the graph of a λ-term. From this set of equations a relation $\asymp$ is generated which is used to rewrite the equations. This process is iterated until no further rewriting steps are possible. We prove that Newman's algorithm is strongly normalizing and confluent for the case of λ→.

Definition 3.1. *A* scheme of a λ-term *(typically S, S',...) is a finite set of equations of the following shape.*

$$\mathsf{Name} \doteq \mathsf{Name}\ \mathsf{Name} \qquad \mathsf{Name} \doteq \lambda\mathsf{Name}.\mathsf{Name}$$

Here Name *ranges over* term names *(typically $U, V, X, Y, Z, \ldots$) and* variables *(typically x, y, z, ...). Moreover, let* TermNames(S) *denote the set of term names occurring in S and* Names(S) *the set of names occurring in S.*

Definition 3.2. *Given a λ-term M, the* scheme $\mathcal{S}(M)$ *is computed simultaneously with the* entrance $\mathcal{E}(M)$ of the scheme *using the following algorithm.*

M	$\mathcal{S}(M)$	$\mathcal{E}(M)$
x	$\emptyset$	x
MN	$\{Z \hat{=} \mathcal{E}(M)\,\mathcal{E}(N)\} \cup \mathcal{S}(M) \cup \mathcal{S}(N)$	Z
$\lambda x.P$	$\{Z \hat{=} \lambda x.\mathcal{E}(P)\} \cup \mathcal{S}(P)$	Z

In the computation of $\mathcal{S}(M)$, the new term names Z should be chosen in such a way that they are fresh. That is, we have to make sure that:

- *in the application case,* $Z \notin \mathrm{TermNames}(\mathcal{S}(M)) \cup \mathrm{TermNames}(\mathcal{S}(N))$,
- *in the application case,* $\mathrm{TermNames}(\mathcal{S}(M)) \cap \mathrm{TermNames}(\mathcal{S}(N)) = \emptyset$,
- *in the λ case,* $Z \notin \mathrm{TermNames}(\mathcal{S}(P))$.

Example 3.3. *The scheme $S = \mathcal{S}(M)$ of the λ-term $M \equiv \lambda fx.f(fx)$ is:*

$$U \hat{=} \lambda f.V \qquad V \hat{=} \lambda x.W \qquad W \hat{=} fZ \qquad Z \hat{=} fx$$

Definition 3.4. *Given a scheme S, the relations $\overset{\mathsf{d}}{\succ}_S, \overset{\mathsf{r}}{\succ}_S \subseteq \mathsf{Name} \times \mathsf{Name}$ are inductively defined as follows.*

1. *If $Z \hat{=} MN$ then $M \overset{\mathsf{d}}{\succ}_S N$ and $M \overset{\mathsf{r}}{\succ}_S Z$*
2. *If $Z \hat{=} \lambda x.P$ then $Z \overset{\mathsf{d}}{\succ}_S x$ and $Z \overset{\mathsf{r}}{\succ}_S P$*

When writing $X \overset{\mathsf{d}}{\succ}_S Y$ or $X \overset{\mathsf{r}}{\succ}_S Y$ we will often suppress S, and write $X \overset{\mathsf{d}}{\succ} Y$ or $X \overset{\mathsf{r}}{\succ} Y$, respectively.

Note that the equations correspond to the edges of the term graph and the relations $\overset{\mathsf{d}}{\succ}$ and $\overset{\mathsf{r}}{\succ}$ correspond to the edges $\xrightarrow{\mathsf{d}}$ and $\xrightarrow{\mathsf{r}}$ of the Newman graph described in the Section 2.

Example 3.5. *The relations $\overset{\mathsf{d}}{\succ}$ and $\overset{\mathsf{r}}{\succ}$ for the λ-term $\lambda fx.f(fx)$ are:*

$$U \overset{\mathsf{d}}{\succ} f \quad U \overset{\mathsf{r}}{\succ} V \qquad V \overset{\mathsf{d}}{\succ} x \quad V \overset{\mathsf{r}}{\succ} W \qquad f \overset{\mathsf{d}}{\succ} Z \quad f \overset{\mathsf{r}}{\succ} W \qquad f \overset{\mathsf{d}}{\succ} x \quad f \overset{\mathsf{r}}{\succ} Z$$

Definition 3.6. *Given a scheme S, a binary relation $\succ_S \subseteq \mathsf{Name} \times \mathsf{Name}$ is defined as $\succ_S := \overset{\mathsf{r}}{\succ}_S \cup \overset{\mathsf{d}}{\succ}_S$.*

Definition 3.7. *Given a scheme S, a binary relation $\asymp_S \subseteq \mathsf{Name} \times \mathsf{Name}$ is inductively defined as follows.*

(A1) If $U \overset{\mathsf{r}}{\succ} X$ and $U \overset{\mathsf{r}}{\succ} Y$, then $X \asymp_S Y$
(A2) If $U \overset{\mathsf{d}}{\succ} X$ and $U \overset{\mathsf{d}}{\succ} Y$, then $X \asymp_S Y$
(B) If $X \overset{\mathsf{r}}{\succ} U$, $Y \overset{\mathsf{r}}{\succ} U$, $X \overset{\mathsf{d}}{\succ} V$ and $Y \overset{\mathsf{d}}{\succ} V$, then $X \asymp_S Y$

As usual, when writing $X \succ_S Y$ or $X \asymp_S Y$ we will often suppress S, and write $X \succ Y$ or $X \asymp Y$, respectively.

Newman also included the following clauses in his definition of $X \asymp Y$.

(C1) If $X \doteq MN$ and $Y \doteq MN$, then $X \asymp Y$
(C2) If $X \doteq \lambda x.P$ and $Y \doteq \lambda x.P$, then $X \asymp Y$

However, as the following lemma indicates, these clauses can be omitted for the specific case of $\lambda{\rightarrow}$.

Lemma 3.8. *If one of the following properties hold we have $X \asymp Y$.*

1. $X \doteq MN$ *and* $Y \doteq MN$
2. $X \doteq \lambda x.P$ *and* $Y \doteq \lambda x.P$

Proof. Suppose that $X \doteq MN$ and $Y \doteq MN$, then by clause (A):

$$M \overset{\mathsf{d}}{\succ} N \qquad \boxed{M \overset{\mathsf{r}}{\succ} X} \qquad M \overset{\mathsf{d}}{\succ} N \qquad \boxed{M \overset{\mathsf{r}}{\succ} Y} \qquad \Longrightarrow \qquad X \asymp Y$$

Suppose that $X \doteq \lambda x.P$ and $Y \doteq \lambda x.P$, then by clause (B):

$$\boxed{X \overset{\mathsf{d}}{\succ} x \qquad X \overset{\mathsf{r}}{\succ} P \qquad Y \overset{\mathsf{d}}{\succ} x \qquad Y \overset{\mathsf{r}}{\succ} P} \qquad \Longrightarrow \qquad X \asymp Y \qquad \square$$

Newman informally describes the meaning of $X \asymp Y$ as "X has the same type as Y", which means that X and Y receive the same type according to the typing rules of $\lambda{\rightarrow}$. He describes the meaning of $X \succ Y$ as "X has a higher type than Y", that means that the type that X receives contains the type that Y receives as a subterm.

Definition 3.9. *An* η-reduction *step*[3] *on schemes,* $S_1 \overset{X:=Y}{\rightarrow} S_2$*, is defined by replacing* X *by* Y *in all equations of* S_1 *provided that* $X \neq Y$ *and* $X \asymp_S Y$*. Multiple steps are denoted by* $S_1 \overset{\nu}{\twoheadrightarrow} S_2$ *where* ν *is a substitution. A scheme* S *is in* η-normal form *if no* η*-reduction steps are possible.*

The following Lemma is also proven by Newman [14].

Lemma 3.10. *The notion of* η*-reduction is strongly normalizing. That is, any* η*-reduction path leads to a normal form.*

Proof. The number of different term names reduces at each η-reduction step. Because a scheme consists of finitely many equations it is immediate that η-reduction leads to a normal form in finitely many steps. $\square$

Definition 3.11. *A* cycle *in a scheme* S *is a sequence* $X_1, \ldots X_n \in \mathsf{Name}$ *for which:*

$$X_1 \succ X_2 \wedge \ldots \wedge X_{n-1} \succ X_n \wedge X_n \succ X_1$$

A scheme S *is* stratified *if it contains no cycles.*

[3] This notion is something completely different from the well-known η-reduction in the λ-calculus: $\lambda x.Mx \rightarrow_\eta M$ provided that $x \notin \mathrm{FV}(M)$.

The situation with respect to the scheme and the relations defined from it is as follows, where an arrow indicates the generation of one entity from another.

$$\lambda\text{-term } M \longrightarrow \text{Scheme } S \longrightarrow \text{Relations } \stackrel{\mathsf{d}}{\succ}_S, \stackrel{\mathsf{r}}{\succ}_S \longrightarrow \text{Relation } \asymp_S$$

After an η-reduction step, we obtain a new scheme S', for which we redefine the relations $\stackrel{\mathsf{d}}{\succ}_{S'}$ and $\stackrel{\mathsf{r}}{\succ}_{S'}$ and thereby the relation $\asymp_{S'}$. After having repeated this process finitely many times we obtain a normal form S_f by Lemma 3.10. According to Newman M is typable iff S_f is stratified.

In Section 2 we have defined reduction on Newman graphs rather than term graphs. Likewise, it is also possible to perform η-reduction on the relations $\stackrel{\mathsf{d}}{\succ}_S$ and $\stackrel{\mathsf{r}}{\succ}_S$. This way we do not have to keep track of the equations of the scheme. However, this results in loss of information because it is not possible to restore an arbitrary scheme from the relations $\stackrel{\mathsf{d}}{\succ}_S$ and $\stackrel{\mathsf{r}}{\succ}_S$. For stratification, as defined in Definition 3.11, this is however no problem.

Example 3.12. *Consider the scheme $S = \mathcal{S}(M)$ of the λ-term $M \equiv f(fx)$.*

$$W \doteq fZ \qquad Z \doteq fx$$

$$\boxed{f \stackrel{\mathsf{d}}{\succ} Z} \qquad f \stackrel{\mathsf{r}}{\succ} W \qquad \boxed{f \stackrel{\mathsf{d}}{\succ} x} \qquad f \stackrel{\mathsf{r}}{\succ} Z$$

After one step of η-reduction, $S \stackrel{Z:=x}{\rightarrow} S'$, the scheme S' is obtained.

$$W \doteq fx \qquad x \doteq fx$$

$$f \stackrel{\mathsf{d}}{\succ} x \qquad \boxed{f \stackrel{\mathsf{r}}{\succ} W} \qquad \boxed{f \stackrel{\mathsf{r}}{\succ} x}$$

Finally a normal form S_f is obtained by $S' \stackrel{W:=x}{\rightarrow} S_f$.

$$x \doteq fx$$

$$f \stackrel{\mathsf{d}}{\succ} x \qquad f \stackrel{\mathsf{r}}{\succ} x$$

There are no cycles, thus we conclude that S_f is stratified.

The following lemma corresponds to what Newman calls "$\asymp$ and $\succ$ are preserved under any homomorphic change of letters".

Lemma 3.13. *If $S \stackrel{\nu}{\twoheadrightarrow} S'$, then for all $X, Y \in \mathsf{Name}$ we have:*

$$X \succ_S Y \implies \nu X \succ_{S'} \nu Y$$
$$X \asymp_S Y \implies \nu X \asymp_{S'} \nu Y$$

Proof. This lemma follows from the observation that $\succ$ and $\asymp$ depend merely on the relative positions of the term names in a scheme S. □

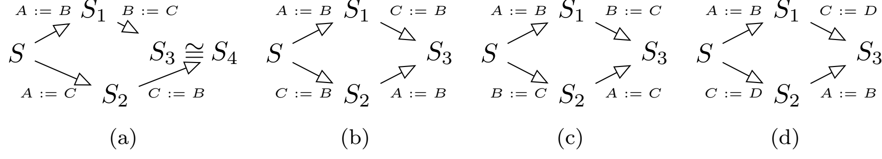


Figure 2: Cases considered in the proof of Lemma 3.15.

Example 3.14. *Consider the scheme $S = \mathcal{S}(M)$ of the λ-term $M \equiv \lambda x.xx$.*

$$W \doteq \lambda x.V \qquad V \doteq xx$$

$$W \overset{\mathsf{d}}{\succeq} x \qquad W \overset{\mathsf{r}}{\succeq} V \qquad \boxed{x \overset{\mathsf{d}}{\succeq} x} \qquad x \overset{\mathsf{r}}{\succeq} V$$

This scheme contains a cycle. By Lemma 3.13 we conclude that if we reduce S to a normal form S_f this scheme will contain a cycle as well, thus S_f is not stratified either.

Note that the previous example does not claim that S_f is unique, but only that in whatever normal form S_f we end up with, S_f is not stratified. We now prove that S_f is unique up to isomorphism, where S_1 and S_2 are *isomorphic*, notation $S_1 \cong S_2$, if there exist substitutions σ_1 and σ_2 such that $S_2 = \sigma_1(S_1)$ and $S_1 = \sigma_1(S_2)$. The following Lemma is also proven by Newman [14].

Lemma 3.15. *The notion of η-reduction is* locally confluent *up to renaming of term names. That is given a scheme S and reductions $S \overset{\nu}{\rightarrow} S_1$ and $S \overset{\delta}{\rightarrow} S_2$, there are schemes S_3 and S_4 and reductions $S_1 \overset{\nu'}{\twoheadrightarrow} S_3$ and $S_2 \overset{\delta'}{\twoheadrightarrow} S_4$ such that $S_3 \cong S_4$.*

Proof. Consider the following cases.

1. If ν and δ are equal then $S_1 = S_2$ and no further reduction is needed.
2. If ν and δ are each other's inverse ($\nu = [A := B]$ and $\delta = [B := A]$) then $S_1 \cong S_2$ and no further reduction is needed.
3. If $\nu = [A := B]$ and $\delta = [A := C]$ then there are reductions $S_1 \rightarrow S_3$ and $S_2 \rightarrow S_4$ such that $S_3 \cong S_4$ as shown in Figure 2a. These reductions are valid according to Lemma 3.13.
4. For the other cases there are reductions $S_1 \rightarrow S_3$ and $S_2 \rightarrow S_3$ as shown in Figure 2b-2d. These reductions are valid according to Lemma 3.13.

These are all the possible cases because there are two cases with two different term names involved, three cases with three different term names involved and only one case with four different term names involved. □

The following Theorem was also proven by Newman [14].

Theorem 3.16. *Each scheme S has a normal form (henceforth S_f) which is unique up to isomorphism.*

Proof. The notion of η-reduction is strongly normalizing by Lemma 3.10 and locally confluent by Lemma 3.15, thus using Newman's Lemma for abstract rewriting systems (Lemma 1.8) η-reduction is confluent. Hence the normal form S_f of S is unique up to isomorphism. □

Corollary 3.17. *The following properties hold for each scheme S.*

1. *Whether S_f is stratified is independent of the order of η-reduction.*
2. *The number of steps in an η-reduction path from S to S_f is fixed and moreover less or equal to the number of names in S.*

Another interesting observation is that the (B) clause in Definition 3.7 can be omitted. That is, given a scheme S, in order to determine whether S_f is stratified we do not need to perform any reduction steps by clause (B). We introduce the following definition before we prove this result.

Definition 3.18. *Let $S \rightarrowtriangle_A S'$ denote an η-reduction step by clause (A1) or (A2) and let $S \rightarrowtriangle_B S'$ denote an η-reduction step by clause (B).*

Lemma 3.19. *If $S_1 \rightarrowtriangle_B S_2 \rightarrowtriangle_A S_3$ then $S_1 \rightarrowtriangle_A S_4$ for some S_4.*

Proof. Let $X \succ^{\mathsf{d}}_{S_1} U$, $Y \succ^{\mathsf{d}}_{S_1} U$, $X \succ^{\mathsf{r}}_{S_1} V$, $Y \succ^{\mathsf{r}}_{S_1} V$ and let $S_1 \overset{X:=Y}{\rightarrowtriangle_B} S_2$ be the resulting $\rightarrowtriangle_B$-step. Now we have to consider the following cases.

1. There exists a $Z \neq U$ such that $X \succ^{\mathsf{d}}_{S_1} Z$ or $Y \succ^{\mathsf{d}}_{S_1} Z$. Then we can perform an $\rightarrowtriangle_A$-step in S_1, namely $[U := Z]$.
2. There exists a $Z \neq V$ such that $X \succ^{\mathsf{r}}_{S_1} Z$ or $Y \succ^{\mathsf{r}}_{S_1} Z$. Then we can perform an $\rightarrowtriangle_A$-step in S_1, namely $[V := Z]$.
3. There does not exist a Z as described in the preceding cases. Then the step $S_2 \rightarrowtriangle_A S_3$ does not depend on the substitution of Y for X, so we can do the same $\rightarrowtriangle_A$-step in S_1. □

Corollary 3.20. *We have the following properties.*

1. *A $\rightarrowtriangle_B$-step cannot create an $\rightarrowtriangle_A$-step. That is, if S is in $\rightarrowtriangle_A$-normal form and $S \rightarrowtriangle_B S'$, then S' is in $\rightarrowtriangle_A$-normal form.*
2. *The reduction $\rightarrowtriangle_B$ can be postponed. That is, given a scheme S, we have $S \twoheadrightarrow_A S' \twoheadrightarrow_B S_f$ for some S'.*

Proof. The first follows immediately from Lemma 3.19. For the second: every scheme S reduces to a (unique) scheme in normal form S_f (Theorem 3.16) with a reduction sequence of fixed length (Corollary 3.17). By Lemma 3.19, we can move all $\rightarrowtriangle_A$-steps to the beginning of the reduction sequence. □

Lemma 3.21. *Given a scheme S that is in $\rightarrowtriangle_A$-normal form and let $S \rightarrowtriangle_B S'$, then S contains a cycle if S' contains a cycle.*

Proof. Let $X \succ^{\mathsf{d}}_{S} U$, $Y \succ^{\mathsf{d}}_{S} U$, $X \succ^{\mathsf{r}}_{S} V$, $Y \succ^{\mathsf{r}}_{S} V$ and let $S \overset{X:=Y}{\rightarrowtriangle_B} S'$ be the resulting $\rightarrowtriangle_B$-step. Let C be the cycle in S'. We now exhibit a cycle in S by distinguishing the following cases.

1. The cycle C includes Y. Since S is in $\twoheadrightarrow_A$-normal form we know by Corollary 3.20 that S' is in $\twoheadrightarrow_A$-normal form. Hence there is no $Z \neq U$ such that $Y \overset{\mathsf{d}}{\succ}_{S'} Z$ or $Z \neq V$ such that $Y \overset{\mathsf{r}}{\succ}_{S'} Z$, therefore C includes $W \succ_{S'} Y \overset{\mathsf{d}}{\succ}_{S'} U$ or $W \succ_{S'} Y \overset{\mathsf{r}}{\succ}_{S'} V$ for some name W. So in S we either have $W \succ_S Y$ or $W \succ_S X$. In the first case we are finished: C is also a cycle in S. In the second case we have $X \overset{\mathsf{d}}{\succ}_S U$ or $X \overset{\mathsf{r}}{\succ}_S V$, so we can rearrange C into a cycle in S by passing through X instead of Y.
2. The cycle C does not include Y. Then C already exists in S. □

Corollary 3.22. *Given a scheme S and let $S \twoheadrightarrow\!\!\!\!\twoheadrightarrow_A S'$ such that S' is in $\twoheadrightarrow_A$-normal form, S_f is stratified iff S' is stratified.*

Proof. The normal form of scheme S' is the same as the one of S, which is S_f. Due to Corollary 3.20, we have $S' \twoheadrightarrow\!\!\!\!\twoheadrightarrow_B S_f$ and each scheme in this reduction sequence is in $\twoheadrightarrow_A$-normal form. By Lemma 3.21, we find that for each $S_1 \twoheadrightarrow_A S_2$ in this reduction sequence, S_1 contains a cycle iff S_2 contains a cycle. Thus, S_f is stratified iff S' is stratified. □

4. Computing a type using Newman's algorithm

In this section we prove that Newman's algorithm is correct and we extend it to result in a principal type. Furthermore, we compare it to Wand's algorithm and his correctness proof [21].

4.1. Correctness of Newman's algorithm

To prove soundness and completeness with respect to typing we define a set of type equations for each scheme. Firstly, we prove that the most general unifier of these equations gives rise to a principal type. Secondly, we prove that performing η-reduction while keeping track of the performed substitutions corresponds to computing a most general unifier.

Definition 4.1. *Given a scheme S, a set of type equations $\mathsf{E}(S)$ is inductively defined as follows.*

1. *If $Z \simeq MN$ then $\overline{M} \simeq \overline{N} \to \overline{Z} \in \mathsf{E}(S)$*
2. *If $Z \simeq \lambda x.P$ then $\overline{Z} \simeq \overline{x} \to \overline{P} \in \mathsf{E}(S)$*

Here $\overline{X}$ is a fresh type variable for each term name X.

Fact 4.2. *Given a scheme S and $X, Y, Z \in \mathsf{Name}$, we have:*

$$X \overset{\mathsf{d}}{\succ} Y \wedge X \overset{\mathsf{r}}{\succ} Z \iff \overline{X} \simeq \overline{Y} \to \overline{Z} \in \mathsf{E}(S)$$

Definition 4.3. *Given a λ-term M, define $\Gamma_M := \{x : \overline{x} \mid x \in \mathrm{FV}(M)\}$.*

Lemma 4.4. *Given a λ-term M and a substitution σ such that $\sigma \models \mathsf{E}(\mathcal{S}(M))$, we have $\sigma(\Gamma_M) \vdash M : \sigma(\overline{\mathcal{E}(M)})$.*

Proof. This property is proven by induction on the structure of M.

(var) Now $\mathcal{E}(x) = x$. The result is immediate because $x : \sigma(\overline{x}) \vdash x : \sigma(\overline{x})$ for each substitution σ.

(app) Now $\mathcal{S}(MN) = \{\mathcal{E}(MN) \doteq \mathcal{E}(M)\ \mathcal{E}(N)\} \cup \mathcal{S}(M) \cup \mathcal{S}(N)$. So we have $\sigma \models \mathsf{E}(\mathcal{S}(M))$ and $\sigma \models \mathsf{E}(\mathcal{S}(N))$ and therefore by the induction hypothesis $\sigma(\Gamma_M) \vdash M : \sigma(\overline{\mathcal{E}(M)})$ and $\sigma(\Gamma_N) \vdash N : \sigma(\overline{\mathcal{E}(N)})$. Moreover we have $\sigma(\overline{\mathcal{E}(M)}) = \sigma(\overline{\mathcal{E}(N)}) \to \sigma(\overline{\mathcal{E}(MN)})$ hence by the application rule and weakening $\sigma(\Gamma_{MN}) \vdash MN : \sigma(\overline{\mathcal{E}(MN)})$.

(λ) Similar to the preceding case. □

Lemma 4.5. *Given a derivable typing judgment $\Delta \vdash M : \rho$, there exists a substitution σ such that $\sigma \models \mathsf{E}(\mathcal{S}(M))$, $\Delta \supseteq \sigma(\Gamma_M)$ and $\rho = \sigma(\overline{\mathcal{E}(M)})$.*

Proof. This property is proven by induction on the type derivation $\Delta \vdash M : \rho$.

(var) Let $\Delta \vdash x : \rho$ such that $x : \rho \in \Delta$. Now $\mathcal{E}(x) = x$ and $\mathcal{S}(x) = \emptyset$. Define $\sigma = [\overline{x} := \rho]$. Then we have $\sigma \models \mathsf{E}(\mathcal{S}(x))$. Also $\Delta \supseteq \sigma(\Gamma_x)$ and $\rho = \sigma(\overline{\mathcal{E}(M)})$, so we are done.

(app) Let $\Delta \vdash MN : \rho$, then we have $\Delta \vdash M : \delta \to \rho$ and $\Delta \vdash N : \delta$. Now $\mathcal{E}(MN) = Z$ and $\mathcal{S}(MN) = \{Z \doteq \mathcal{E}(M)\ \mathcal{E}(N)\} \cup \mathcal{S}(M) \cup \mathcal{S}(N)$. By the induction hypothesis we obtain substitutions σ_N and σ_M. We define σ as follows.

$$\begin{aligned}
\sigma(\overline{Y}) &= \sigma_M(\overline{Y}) && \text{if } Y \in \text{Names}(\mathcal{S}(M))\\
\sigma(\overline{Y}) &= \sigma_N(\overline{Y}) && \text{if } Y \in \text{Names}(\mathcal{S}(N))\\
\sigma(\overline{Z}) &= \rho\\
\sigma(\alpha) &= \alpha && \text{otherwise}
\end{aligned}$$

This substitution is well defined because

(a) The term MN satisfies the Barendregt convention and therefore all bound variables in M and N are fresh.

(b) The term names assigned to the subterms of M and N are disjoint (by Definition 3.2).

(c) By the induction hypothesis we have $\sigma_M(\overline{x}) = \Delta(x)$ for all variables $x \in \text{FV}(M)$ and $\sigma_N(\overline{x}) = \Delta(x)$ for all variables $x \in \text{FV}(N)$.

By the induction hypothesis we have $\sigma \models \mathsf{E}(\mathcal{S}(M))$, $\sigma \models \mathsf{E}(\mathcal{S}(N))$ and $\sigma(\overline{\mathcal{E}(M)}) = \delta \to \rho = \sigma(\overline{\mathcal{E}(N)}) \to \sigma(\overline{Z})$, so $\sigma \models \mathsf{E}(\mathcal{S}(MN))$. Moreover, we have $\Delta \supseteq \sigma(\Gamma_{MN})$ and $\rho = \sigma(\overline{\mathcal{E}(MN)})$, so we are done.

(λ) Let $\Delta \vdash \lambda x.P : \delta \to \rho$, then we have $\Delta, x : \delta \vdash P : \rho$. Now $\mathcal{E}(\lambda x.P) = Z$ and $\mathcal{S}(\lambda x.P) = \{Z \doteq \lambda x.\mathcal{E}(P)\} \cup \mathcal{S}(P)$. By the induction hypothesis we obtain a substitution σ_P. We define σ as follows.

$$\begin{aligned}
\sigma(\overline{Y}) &= \sigma_P(\overline{Y}) && \text{if } Y \in \text{Names}(\mathcal{S}(P))\\
\sigma(\overline{x}) &= \delta\\
\sigma(\overline{Z}) &= \delta \to \rho\\
\sigma(\alpha) &= \alpha && \text{otherwise}
\end{aligned}$$

By the induction hypothesis we obtain that $\sigma \models \mathsf{E}(\mathcal{S}(P))$ and $\sigma(\overline{Z}) = \delta \to \rho = \sigma(\overline{x}) \to \sigma(\overline{\mathcal{E}(P)})$, hence $\sigma \models \mathsf{E}(\mathcal{S}(\lambda x.P))$. Moreover, we have $\Delta \supseteq \sigma(\Gamma_{\lambda x.P})$ and $\delta \to \rho = \sigma(\overline{\mathcal{E}(\lambda x.P)})$, so we are done. □

Corollary 4.6. *The equations à la Newman are correct. That is, given a λ-term M and its scheme $S = \mathcal{S}(M)$, we have*

1. *If $\sigma \models_{\mathrm{mgu}} \mathsf{E}(S)$ then $\langle \sigma(\overline{\mathcal{E}(M)}), \sigma(\Gamma_M)\rangle$ is a principal pair of M.*
2. *If $\not\models \mathsf{E}(S)$ then $\not\vdash M$.*

Proof. Immediate from Lemma 4.4 and 4.5. □

Now that we have shown that the first part of Newman's algorithm actually consists of the generation of type equations, we will prove that performing η-reduction corresponds to computing a most general unifier. First we define a substitution $\widehat{S}$ for each scheme S that is stratified and in normal form. Also, we prove that this is in fact the most general unifier of $\mathsf{E}(S)$.

Definition 4.7. *Given a scheme S that is stratified and in $\twoheadrightarrow_A$-normal form, the substitution $\widehat{S} : \mathsf{TVar} \to \mathsf{Type}$ is defined as follows.*

$$\widehat{S}(\alpha) = \begin{cases} \widehat{S}(\beta) \to \widehat{S}(\gamma) & \text{if } \alpha \doteq \beta \to \gamma \in \mathsf{E}(S) \\ \alpha & \text{otherwise} \end{cases}$$

Lemma 4.8. *Given a scheme S that is stratified and in $\twoheadrightarrow_A$-normal form, the substitution $\widehat{S}$ is well defined.*

Proof. In order to show that $\widehat{S}$ is well defined we have to prove that the definition of $\widehat{S}$ is unambiguous and total.

For the first condition we have to show that for each $\overline{Z}$ there is at most one equation $\overline{Z} \doteq \overline{X} \to \overline{Y} \in \mathsf{E}(S)$. Let us suppose the contrary, $\overline{Z} \doteq \overline{X} \to \overline{Y} \in \mathsf{E}(S)$ and $\overline{Z} \doteq \overline{A} \to \overline{B} \in \mathsf{E}(S)$, now by Fact 4.2 we have $X \asymp A$ and $Y \asymp B$. So a contradiction is obtained because S is in $\twoheadrightarrow_A$-normal form.

Moreover by Fact 4.2 and stratification we know that no cycles in the type equations exist, hence $\widehat{S}$ is total. □

Lemma 4.9. *Given a scheme S that is stratified and in $\twoheadrightarrow_A$-normal form, we have $\widehat{S} \models_{\mathrm{mgu}} \mathsf{E}(S)$.*

Proof. We have $\widehat{S}(\overline{X}) = \widehat{S}(\overline{A}) \to \widehat{S}(\overline{B})$ for each $\overline{X} \doteq \overline{A} \to \overline{B} \in \mathsf{E}(S)$ by definition, hence $\widehat{S} \models \mathsf{E}(S)$. So it remains to prove that $\widehat{S}$ is a most general unifier of $\mathsf{E}(S)$. Therefore suppose that we have a substitution δ such that $\delta \models \mathsf{E}(S)$. Now we should define a substitution σ such that $\sigma \circ \widehat{S} = \delta$. We define $\sigma := \delta$. We prove that $\delta(\widehat{S}(\overline{X})) = \delta(\overline{X})$ for all $X \in \mathsf{Name}$ by well-founded induction over $\succ_S$ (this is allowed because S is stratified). So suppose that we have $\delta(\widehat{S}(\overline{Y})) = \delta(\overline{Y})$ for all $Y \in \mathsf{Name}$ such that $X \succ Y$. Now we prove that $\delta(\widehat{S}(\overline{X})) = \delta(\overline{X})$ by distinguishing the following cases.

1. $\overline{X} \doteq \overline{A} \to \overline{B} \in \mathsf{E}(S)$. Then we have $\delta(\widehat{S}(\overline{A})) = \delta(\overline{A})$ and $\delta(\widehat{S}(\overline{B})) = \delta(\overline{B})$ by the hypothesis. Hence we have $\delta(\widehat{S}(\overline{X})) = \delta(\overline{X})$ as shown below.

$$\delta(\widehat{S}(\overline{X})) = \delta(\widehat{S}(\overline{A})) \to \delta(\widehat{S}(\overline{B})) = \delta(\overline{A}) \to \delta(\overline{B}) = \delta(\overline{X})$$

2. $\overline{X} \doteq \overline{A} \to \overline{B} \notin \mathsf{E}(S)$. Then we have $\delta(\widehat{S}(\overline{X})) = \delta(\overline{X})$. □

Lemma 4.10. *Given a scheme S that is not stratified, we have $\not\models \mathsf{E}(S)$.*

Proof. If S is not stratified, $\mathsf{E}(S)$ contains a cycle and therefore $\not\models \mathsf{E}(S)$. □

So far we have proven two parts of the correctness of Newman's algorithm: the equations $\mathsf{E}(S)$ are correct and $\widehat{S_f}$ is the most general unifier of $\mathsf{E}(S_f)$. So it remains to show how we can use Newman's algorithm to compute a most general unifier of $\mathsf{E}(S)$. As introduced before, we will do this by keeping track of the substitutions while performing η-reduction. Because these substitutions replace names for names instead of types for type variables we introduce the following definition.

Definition 4.11. *Given a substitution ν : $\mathsf{Name} \to \mathsf{Name}$, the substitution $\overline{\nu}$: $\mathsf{TVar} \to \mathsf{TVar}$ is defined as $\overline{\nu}(\overline{X}) = \overline{\nu(X)}$.*

In the remainder of this section we will prove that if $S \overset{\nu}{\twoheadrightarrow} S_f$ then $\widehat{S_f} \circ \overline{\nu}$ is the most general unifier of $\mathsf{E}(S)$ iff S_f is stratified. This finally leads to a correctness proof of Newman's algorithm.

Lemma 4.12. *If $S \overset{\nu}{\twoheadrightarrow} S'$ and $\delta \models \mathsf{E}(S')$ then $\delta \circ \overline{\nu} \models \mathsf{E}(S)$.*

Proof. Immediate because $\delta \models \overline{\nu}(\mathsf{E}(S))$ implies $\delta \circ \overline{\nu} \models \mathsf{E}(S)$. □

Lemma 4.13. *If $S \overset{X:=Y}{\to} S'$ and $\delta \models \mathsf{E}(S)$ then $\delta(\overline{X}) = \delta(\overline{Y})$.*

Proof. We prove this result by distinguishing the following cases.

(A1) $\overline{Z} \doteq \overline{X} \to \overline{V}$, $\overline{Z} \doteq \overline{Y} \to \overline{W} \in \mathsf{E}(S)$. Now we have $\delta(\overline{Z}) = \delta(\overline{X}) \to \delta(\overline{V})$ and $\delta(\overline{Z}) = \delta(\overline{Y}) \to \delta(\overline{W})$. So $\delta(\overline{X}) \to \delta(\overline{V}) = \delta(\overline{Y}) \to \delta(\overline{W})$ and therefore $\delta(\overline{X}) = \delta(\overline{Y})$.

(A2) Similar to the preceding case.

(B) $\overline{X} \doteq \overline{V} \to \overline{W}$, $\overline{Y} \doteq \overline{V} \to \overline{W} \in \mathsf{E}(S)$. Now we have $\delta(\overline{X}) = \delta(\overline{V}) \to \delta(\overline{W})$ and $\delta(\overline{Y}) = \delta(\overline{V}) \to \delta(\overline{W})$ and therefore $\delta(\overline{X}) = \delta(\overline{Y})$. □

Corollary 4.14. *If $S \overset{\nu}{\twoheadrightarrow} S'$ and $\delta \models \mathsf{E}(S)$ then $\delta \circ \overline{\nu} = \delta$.*

Proof. Immediate from Lemma 4.13. □

Lemma 4.15. *If $S \overset{\nu}{\twoheadrightarrow} S'$ and $\delta \models \mathsf{E}(S)$ then $\delta \models \mathsf{E}(S')$.*

Proof. We have to prove that we have $\delta(\overline{\nu X}) \doteq \delta(\overline{\nu A}) \to \delta(\overline{\nu B})$ for each equation $\overline{X} \doteq \overline{A} \to \overline{B} \in \mathsf{E}(S)$. By assumption we have $\delta(\overline{X}) \doteq \delta(\overline{A}) \to \delta(\overline{B})$ so by Corollary 4.14 we are done. □

Theorem 4.16. *Given a scheme S and let $S \overset{\nu}{\twoheadrightarrow} S_f$, we have:*

1. *If S_f is stratified then $\widehat{S_f} \circ \overline{\nu} \models_{\text{mgu}} \mathsf{E}(S)$.*
2. *If S_f is not stratified then $\not\models \mathsf{E}(S)$.*

Proof. By Lemma 4.9 we have $\widehat{S_f} \models \mathsf{E}(S_f)$ and by Lemma 4.12 we obtain that $\widehat{S_f} \circ \overline{\nu} \models \mathsf{E}(S)$. So for the first property it remains to prove that $\widehat{S_f} \circ \overline{\nu}$ is a most general unifier of $\mathsf{E}(S)$. Therefore let us suppose that we have a substitution δ such that $\delta \models \mathsf{E}(S)$. By Lemma 4.15 we have $\delta \models \mathsf{E}(S_f)$ and by Lemma 4.9 we obtain a substitution σ such that $\delta = \sigma \circ \widehat{S_f}$. Now it remains to prove that $\delta = \sigma \circ \widehat{S_f} \circ \overline{\nu}$, but by Corollary 4.14 we have $\delta = \delta \circ \overline{\nu}$, so we are done.

To prove the second property let us suppose that we have a substitution δ such that $\delta \models \mathsf{E}(S)$, then by Lemma 4.15 we have $\delta \models \mathsf{E}(S_f)$. But now we obtain a contradiction with Lemma 4.10 because S_f is not stratified. □

Theorem 4.17. Newman's algorithm is sound and complete with respect to typing. *That is, given a λ-term M, its scheme $S = \mathcal{S}(M)$ and let $S \overset{\nu}{\twoheadrightarrow} S_f$, we have:*

1. *If S_f is stratified then $\langle \sigma(\overline{\mathcal{E}(M)}), \sigma(\Gamma_M) \rangle$, where $\sigma = \widehat{S_f} \circ \overline{\nu}$, is a principal pair of M.*
2. *If S_f is not stratified then $\not\vdash M$.*

Proof. Immediate from Corollary 4.6 and Theorem 4.16. □

4.2. Comparison with the algorithm of Wand

The generation of type equations as defined in Definition 4.1 and the statements of Lemma 4.4 and 4.5 look quite similar to Wand's algorithm and his correctness proof [21], respectively. However we show that there are some notable differences between Wand's method and ours (borrowed from Newman).

Wand describes his algorithm with a *skeleton* and an *action table*. Given a closed term M one starts with an empty set of equations E and a set of goals G consisting of one initial goal $(\emptyset, M, \alpha)$ where α is a fresh type variable. While the set of goals G is non-empty one picks a goal g from G, deletes it from G, and uses the following action table to generate new goals and equations.

g	SG(g)	EQ(g)
(Γ, x, τ)	$\emptyset$	$\tau \doteq \Gamma(x)$
$(\Gamma, \lambda x.M, \tau)$	$(\Gamma; x : \alpha_1, M, \alpha_2)$	$\tau \doteq \alpha_1 \to \alpha_2$
$(\Gamma, M\,P, \tau)$	$(\Gamma, M, \alpha \to \tau), (\Gamma, P, \alpha)$	$\emptyset$

The newly generated goals SG(g) are added to the set of goals G and the newly generated equations EQ(g) are added to the set of equations E. This process is repeated until the set of goals G is empty. The correctness of Wand's algorithm states that $\delta \models_{\text{mgu}} E$ iff $\delta(\alpha)$ is a principal type of M.

So we observe at least the following important differences between Wand's method and ours.

1. Wand's algorithm results in a set of type equations of the shape $\sigma \doteq \tau$ where $\sigma, \tau \in \mathsf{Type}$. In our method the generated equations are of the shape $\alpha \doteq \beta \to \gamma$ where $\alpha, \beta, \gamma \in \mathsf{TVar}$.
2. Wand's algorithm works for terms that do not satisfy the Barendregt convention. Therefore Wand's algorithm keeps track of the free variables explicitly by carrying a context around.
3. Wand's algorithm takes a type variable α as its input. For the generated equations E we have $\delta \models_{\mathrm{mgu}} E$ iff $\delta(\alpha)$ is a principal type of M. Our algorithm results in a type variable $\overline{\mathcal{E}(M)}$ and a set of equations $\mathcal{E}(M)$ such that $\delta \models_{\mathrm{mgu}} \mathsf{E}(\mathcal{S}(M))$ iff $\delta(\overline{\mathcal{E}(M)})$ is a principal type of M. So, Wand's algorithm generates equations top-down while our algorithm generates equations bottom-up.

One could try to modify Wand's action table in such a way that it results in type equations of the required shape[4] in the following way.

g	$SG(g)$	$EQ(g)$
(Γ, x, α)	$\emptyset$	$\alpha \doteq \Gamma(x)$
$(\Gamma, \lambda x.M, \alpha)$	$(\Gamma; x : \alpha_1, M, \alpha_2)$	$\alpha \doteq \alpha_1 \to \alpha_2$
$(\Gamma, M\,P, \alpha)$	$(\Gamma, M, \alpha_1), (\Gamma, P, \alpha_2)$	$\alpha_1 \doteq \alpha_2 \to \alpha$

However, in the case of a variable an equation of the shape $\alpha \doteq \beta$ is generated. Unfortunately, due to Wand's top-down approach we cannot take these equations into account in the application and λ cases.

5. Extension with other type constructions

The typing algorithm à la Newman can be extended to include other type constructors. First we show how to include *product types*. This is a proper extension of Newman's method, because the algorithm should now also fail due to *conflicting type constructors*, a case Newman does not deal with. This requires a refinement of the notion of being stratified. Other "simple extensions", like *sum types* can be added in a similar way.

Secondly, we show how to add *weak polymorphism*, which involves universal quantification over type variables (but only on the top level). This is more interesting, because it is basically only relevant if we consider terms inside a context, in which the variables have weakly polymorphic types.

5.1. Product types

We first look into product types, so we define the rules of the system.

[4] For clarity of presentation we generate type equations instead of term equations that have to be translated into type equations.

Definition 5.1. *The system* $\lambda\!\rightarrow$ *is extended with* product types *as follows. Firstly, we extend the terms to include pairing and the projections.*

$$\Lambda ::= \mathsf{Var} \mid \langle \Lambda, \Lambda \rangle \mid \pi_1 \Lambda \mid \pi_2 \Lambda \mid (\Lambda\Lambda) \mid (\lambda \mathsf{Var}.\Lambda)$$

Secondly, we extend the simple types with product types.

$$\mathsf{Type} ::= \mathsf{TVar} \mid (\mathsf{Type} \rightarrow \mathsf{Type}) \mid (\mathsf{Type} \times \mathsf{Type})$$

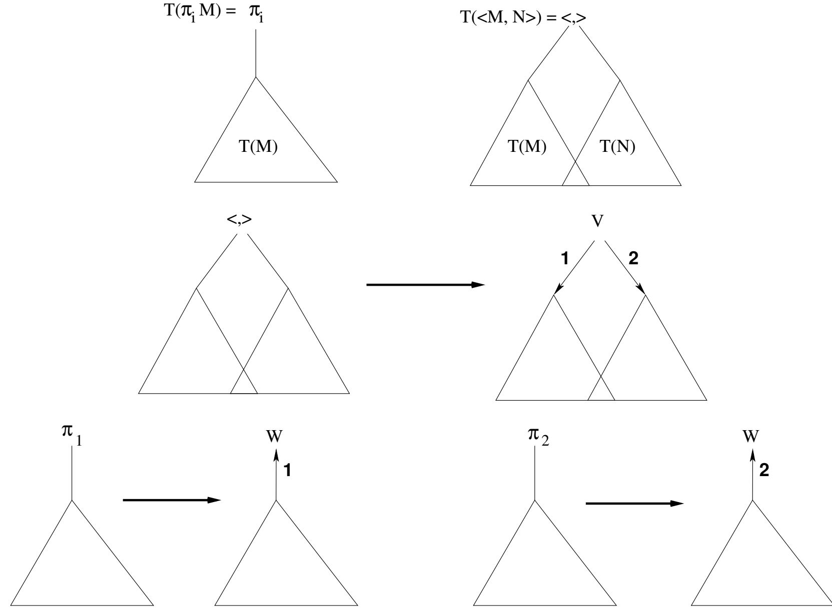
Thirdly, we extend Definition 1.2 to include the following rules.

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash \langle M, N \rangle : \sigma \times \tau} \qquad \frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \pi_1 M : \sigma} \qquad \frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \pi_2 M : \tau}$$

(a) Pairing (b) First Projection (c) Second Projection

Now we extend the tree of a term and the graph of a term to include pairing and projection constructions.

Definition 5.2. *We extend the translation of terms to trees and the translation of trees to Newman graphs as depicted in the figure. We write* Tgraph(M) *for the tree we obtain and* Ngraph(M) *for the Newman graph we obtain. In the construction of the Newman graph, we again choose a fresh label for each node.*



The intuition will be clear again:

$X \xrightarrow{1} Y$ iff X is a product type whose first component type is Y

$X \xrightarrow{2} Y$ iff X is a product type whose second component type is Y.

We now extend the notion of *congruence* between nodes to include the new arrows in the graph.

Definition 5.3. *The notion of* congruence $\asymp_G$ *between nodes in a Newman graph of Definition 2.3 is extended by adding the following inductive clauses.*

1. *If* $U \xrightarrow{1} X$ *and* $U \xrightarrow{1} Y$, *then* $X \asymp_G Y$
2. *If* $U \xrightarrow{2} X$ *and* $U \xrightarrow{2} Y$, *then* $X \asymp_G Y$
3. *If* $X \xrightarrow{1} U$, $X \xrightarrow{2} V$ *and* $Y \xrightarrow{1} U$, $Y \xrightarrow{2} V$, *then* $X \asymp_G Y$

The notion of reduction of Definition 2.5 is immediately extended: we can join nodes X and Y in case $X \asymp Y$, where the congruence relation $X \asymp Y$ is now the one of Definition 5.3. A graph is in *normal form* in case no further reductions are possible. The only notion that really changes is that of a graph being *stratified*.

Definition 5.4. *Given a Newman graph G in the system extended with products, then G is* stratified *in case the following conditions hold.*

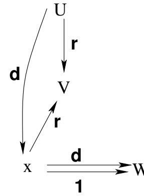
1. *The relation* $\xrightarrow{1} \cup \xrightarrow{2} \cup \xrightarrow{\mathsf{d}} \cup \xrightarrow{\mathsf{r}}$ *contains no cycles.*
2. *The relation* $\xrightarrow{1} \cup \xrightarrow{2} \cup \xrightarrow{\mathsf{d}} \cup \xrightarrow{\mathsf{r}}$ *contains no* conflict.

Here we define a conflict *as a node X that has an outgoing arrow* $\xrightarrow{l_1}$ *and an outgoing arrow* $\xrightarrow{l_2}$ *such that* $l_1 \in \{1, 2\}$ *and* $l_2 \in \{\mathsf{d}, \mathsf{r}\}$.

Note that the new case in the definition of stratified corresponds to the case of conflicting function symbols in the unification algorithm. In the type system with only $\to$, we have only one function symbol, so unification can only fail due to the occurs check (the cyclicity in the definition of stratified). In the presence of products, one can also fail due to conflicting type constructors.

Lemma 5.5. *The reduction* $\twoheadrightarrow$ *is strongly normalizing and confluent. Moreover, the normal form of the Newman graph of M is stratified if and only if M is typable in* $\lambda\!\!\to$ *extended with products.*

Example 5.6. *Consider the* λ*-term* $M \equiv \lambda x.x(\pi_1 x)$. *The normal form of its Newman graph is depicted below and one can observe that it is not stratified, due to a conflict in the node* x.



Just as we have done for arrow types in example 2.9, one can now add type information to the nodes in a stratified Newman graph, to compute the type of the original term. We will not detail that here.

5.2. Weak polymorphism

In this section we treat the λ-calculus with weak polymorphism (henceforth $\lambda 2w$) and give a typing algorithm "à la Newman". The *weak polymorphism* means that the types of our system are *type schemes*, of the form $\forall\vec{\alpha}.\sigma$, where σ is a simple type.

Definition 5.7. *The* weakly polymorphic types *are of the following form*

$$\mathsf{Type}_\omega ::= \forall \mathsf{TVar}.\mathsf{Type}_\omega \mid \mathsf{Type}$$

The free type variables*,* $\mathrm{FTV}(\sigma)$*, of a weakly polymorphic type are inductively defined as follows.*

$$\begin{aligned}\mathrm{FTV}(\alpha) &= \{\alpha\}\\ \mathrm{FTV}(\sigma \to \tau) &= \mathrm{FTV}(\sigma) \cup \mathrm{FTV}(\tau)\\ \mathrm{FTV}(\forall\alpha.\sigma) &= \mathrm{FTV}(\sigma) \backslash \{\alpha\}\end{aligned}$$

A context Γ is now a sequence of declarations of the form $x : \sigma$, where σ is a weakly polymorphic type.

Definition 5.8. *A* $\lambda 2w$-typing judgment $\Gamma \vdash M : \rho$ *denotes that a* λ*-term* M *has type* ρ *in context* Γ*. The derivation rules for deriving such a judgment are as follows.*

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$
(c) Variable

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$
(d) Application

$$\frac{\Gamma, x : \sigma \vdash P : \tau}{\Gamma \vdash \lambda x.P : \sigma \to \tau}\ \tau \in \mathsf{Type}$$
(e) Abstraction

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall\alpha.\sigma}\ \alpha \notin FTV(\Gamma)$$
(f) Generalization

$$\frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M : \sigma[\alpha := \tau]}\ \tau \in \mathsf{Type}$$
(g) Instantiation

Example 5.9. *Consider a derivation of* $x : \forall\beta.\beta \to \beta \vdash xx : \forall\alpha.\alpha \to \alpha$*. We abbreviate* $\Gamma = x : \forall\beta.\beta \to \beta$*.*

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash x : \forall\beta.\beta \to \beta}{\Gamma \vdash x : (\alpha \to \alpha) \to \alpha \to \alpha} \qquad \dfrac{\Gamma \vdash x : \forall\beta.\beta \to \beta}{\Gamma \vdash x : \alpha \to \alpha}}{\Gamma \vdash xx : \alpha \to \alpha}}{\Gamma \vdash xx : \forall\alpha.\alpha \to \alpha}$$

Note that we cannot abstract over x, because it has a weakly polymorphic type. So the term $\lambda x.xx$ is not typable in $\lambda 2w$.

To extend the notion of Newman graph to include terms of $\lambda 2w$, we only have to look at the polymorphic variables in the context, as the term structure does not change. So, we define the translation of a declaration $x : \forall\vec{\alpha}.\sigma$ into a Newman graph. First we define the translation of a simple type into a Newman graph. (We could do this via first defining the tree of the type, but we skip that step now.)

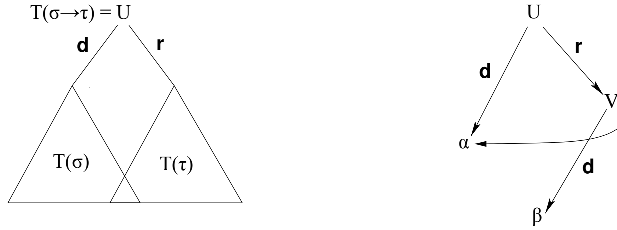


Figure 3: Translating a type to a Newman graph.

Definition 5.10. *Given a type $\sigma \in \mathsf{Type}$, we define the* Newman graph *of this type,* $\text{Ngraph}(\sigma)$ *by induction as described in Figure 3, where we choose a fresh label (U in the picture) and we share all identical type variables in one leaf.*

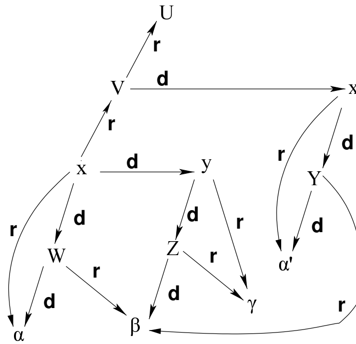
As an example we indicate the translation of $\alpha \to \beta \to \alpha$ in Figure 3. Here we see the sharing of node α.

Definition 5.11. *Given a context $\Gamma = x_1 : \forall\vec{\alpha_1}.\sigma_1,\ \ldots,\ x_n : \forall\vec{\alpha_n}.\sigma_n$ and an untyped λ-term M with $\text{FV}(M) \subset \{x_1, \ldots, x_n\}$, we define the* Newman graph *of M as in Definition 2.2, with the proviso that:*

1. *Every* occurrence *of a free variable x_i is labeled uniquely and replaced by the Newman graph of its type,* $\text{Ngraph}(\sigma_i[\vec{\alpha_i} := \vec{\beta}])$ *where $x_i : \forall\vec{\alpha_i}.\sigma_i \in \Gamma$ and $\vec{\beta}$ consists of fresh type variables.*
2. *All occurrences of* free type variables *in the types of $x_1, \ldots, x_n$ are shared.*

To see what the definition means exactly, we treat an example.

Example 5.12. *Let us consider the λ-term $M \equiv x\,y\,x$ in the context $\Gamma = x : \forall\alpha.(\alpha \to \beta) \to \alpha,\ y : \forall\gamma.(\beta \to \gamma) \to \gamma$. Then the Newman graph of M in Γ is as follows.*



We see that the graph of the type of x occurs twice, because x occurs twice in the term M. Furthermore, the free type variable β is shared by all the three type trees, of the first occurrence of x, of y, and of the second occurrence of x.

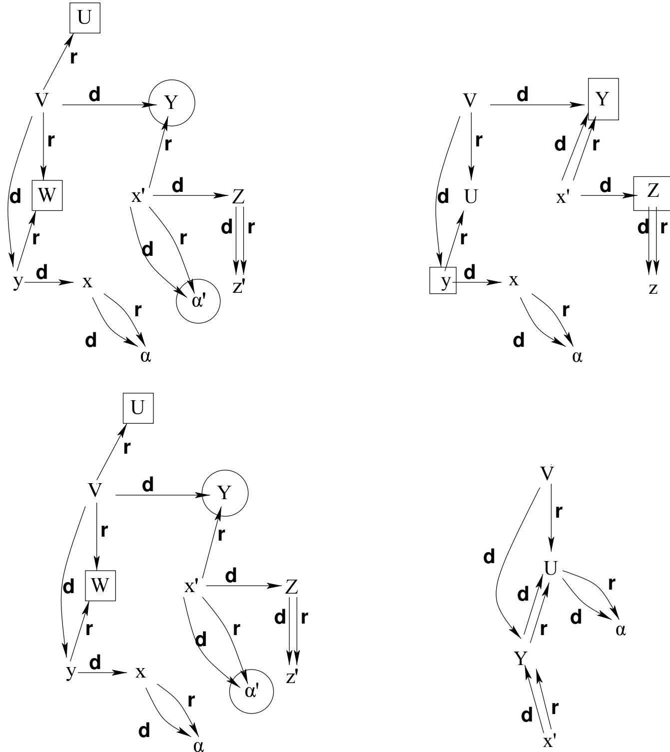
The congruence relation (Definition 2.3) does not change, and neither does the reduction on Newman graphs (Definition 2.5). The normal form of the Newman graph of a $\lambda 2w$-term is again unique up to renaming of labels and we have the following Lemma.

Lemma 5.13. *The reduction $\rightarrowtriangle$ is strongly normalizing and it is confluent. Moreover, the normal form of the Newman graph of M is stratified if and only if M is typable in $\lambda 2w$.*

We can also "read off" the type from the Newman graph in normal form. As we are in $\lambda 2w$, we want to generalize as many type variables as possible. This can be done by keeping track of the variables that are "fixed in the context", like β in the example context of Example 5.12.

To show this, we look at the term $Q \equiv (\lambda y.\, y\, x)\, (x\, (\lambda z.\, z))$ in the context $\Gamma = x : \forall\alpha.\alpha \to \alpha$. Remark that this term is closely related to the term $M \equiv \lambda x.\, (\lambda y.\, y\, x)\, (x\, (\lambda z.\, z))$ that we have considered in Example 2.6 and which is not typable in $\lambda\!\to$.

Example 5.14. *Let us consider the λ-term $Q \equiv (\lambda y.\, y\, x)\, (x\, (\lambda z.\, z))$ in the context $\Gamma = x : \forall\alpha.\alpha \to \alpha$. We draw its Newman graph and reduce it to normal form. In boxes and in circles we indicate the congruent nodes that we join in a couple of larger steps.*



The graph in normal form contains no cycle, so we conclude that the term is typable. We can again read off the type by annotating the nodes without outgoing edges with type variables and constructing the type for the node U. We conclude that the principal type of the term in $\lambda 2w$ is: $\forall\alpha.\alpha \to \alpha$.

6. Unification à la Newman

In the previous sections we have seen that Newman's algorithm consists of the following steps.

1. Create a set of equations from a term.
2. Using these term equations, define *relations* corresponding to type constraints. From these relations a *congruence relation* is created.
3. Rewrite the equations by replacing a name by another congruent one; restart from (2) until no more rewrites are possible.

Newman rewrites the equations, but we have seen that we can also rewrite the relations, and basically forget about the equations after step (1). In our graphical representation we have reformulated Newman's algorithm as follows.

1. Create a set of equations from a term and from that a set of *relations* corresponding to type constraints.
2. Define from these relations a *congruence relation* on names.
3. Rewrite the *relations* by replacing a name by another congruent one; restart from (2) until no more rewrites are possible.

Our relations for deciding typing are $\xrightarrow{\mathsf{d}}$ and $\xrightarrow{\mathsf{r}}$ for $\lambda{\to}$ (Section 2) and $\xrightarrow{\mathsf{d}}$, $\xrightarrow{\mathsf{r}}$, $\xrightarrow{1}$, and $\xrightarrow{2}$ for $\lambda{\to}$ extended with products (Section 5.1). These relations correspond to type constraints of the shape $\alpha \doteq \beta \to \gamma$ and $\alpha \doteq \beta \times \gamma$.

As shown in Section 4, the phases of Newman's algorithm are comparable to the typing algorithm of Wand, where a set of equations is created, which is then "solved" by a unification algorithm. We have seen that the equations of Wand are constructed top-down, while Newman creates the relations bottom-up. However, on another track, Newman's algorithm is very close to what happens in Wand's algorithm, because the steps (2) and (3) in the description above actually do unification! In this section we will make this precise and show that Newman's method gives rise to a unification algorithm and actually a quite efficient one. It is in set-up very close to what is called *unification of term DAGs (Directed Acyclic Graphs)* in [16, 2]. Before going into the definitions we give an example that should clarify the connection with unification.

Example 6.1. *Suppose we have a first order language with two function symbols: f of arity 2 and h of arity 1. Now we want to find a unifier for the one-member set of equations $E = \{f(x, h(v)) \doteq f(h(y), x)\}$.*

We introduce names for all subterms and replace E by the equivalent set of equations $E' = \{A \doteq f(x, B), B \doteq h(v), A \doteq f(C, x), C \doteq h(y)\}$. We now rewrite *$E'$ by replacing a symbol X by Y in case the set of equations contains two equations of the shape $U \doteq f(\ldots, X, \ldots)$ and $U \doteq f(\ldots, Y, \ldots)$, where X*

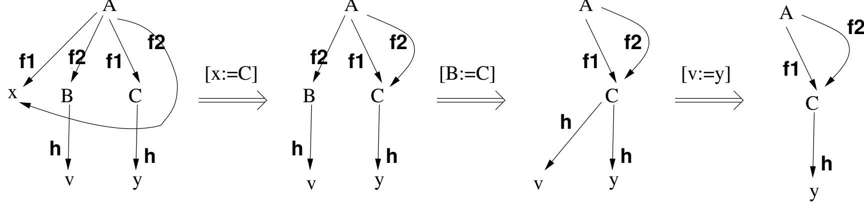


Figure 4: Unification algorithm á la Newman graphically.

and Y are in the same position in the argument list of f. The rewriting goes as follows.

$$\left\{ \begin{array}{c} A \simeq f(x,B) \\ B \simeq h(v) \\ A \simeq f(C,x) \\ C \simeq h(y) \end{array} \right\} \Rightarrow_{[x:=C]} \left\{ \begin{array}{c} A \simeq f(C,B) \\ B \simeq h(v) \\ A \simeq f(C,C) \\ C \simeq h(y) \end{array} \right\} \Rightarrow_{[B:=C]}$$

$$\left\{ \begin{array}{c} A \simeq f(C,C) \\ C \simeq h(v) \\ C \simeq h(y) \end{array} \right\} \Rightarrow_{[v:=y]} \left\{ \begin{array}{c} A \simeq f(C,C) \\ C \simeq h(y) \end{array} \right\}$$

If we collect the substitutions we have performed along the way, we obtain the most general unifier $[x := h(y), v := y]$ of E.

In the example we see that we never substitute a compound term, but only names for names. This prevents a size blow up that one could encounter in a simple-minded ordinary unification algorithm. We also give a graphical representation of the unification, based on the same example.

Example 6.2. *Let us consider the set of equations $E = \{f(x, h(v)) \simeq f(h(y), x)\}$ again. We now replace each term by a* Newman graph*, which is a DAG obtained by creating for every subterm $f(t, q)$ a new node U with $\xrightarrow{f_1}$ pointing to the node of subterm t and $\xrightarrow{f_2}$ pointing to the node of subterm q. We share variable nodes. Furthermore, we represent an equation $r \simeq s$ by joining the two top nodes of the terms r and s. For E, this produces the Newman graph depicted in Figure 4.*

We now define precisely how unification "à la Newman" works. Let a signature be given with function symbols with fixed arity (typically f, g, ...) and first order terms over this signature (typically t, q, ...). We first define for every term t a set of equations of the form $U \simeq t$, where U is a "term name" that names the subterm q of t.

Definition 6.3. *Given a first order term t, the set of* Newman equations Neqns(t) *is defined as follows.*

1. *Create a fresh* term name *(typically U, V, X, Y, Z) to* name *each non-variable subterm of t.*

2. *Create a set of equations* Neqns(t) *from* t *by putting* $U \doteq f(L_1, \ldots, L_n)$ *where* U *is the name of the subterm* $f(t_1, \ldots, t_n)$ *and* L_i *is the name of the subterm* t_i *(and* L_i *is just* x *if* t_i *is the variable* x*).*

Equivalently, we could have defined the *Newman graph* of a term, by depicting a term as a DAG, where all the nodes are labeled by the term names introduced in Definition 6.3. We now pursue the equational approach, but we could equivalently have defined everything graphically. In any case is it good to keep also the graphical presentation of Example 6.2 in mind. We now define the Newman equations for a set of unification problems, which are equations of the form $t \doteq q$. The idea is to take, for the unification problem $t \doteq q$, the Newman equations for t and the Newman equations for q (Definition 6.3) and to add an equation $U \doteq V$ that equates the "roots" of t and q.

Definition 6.4. *Given a set of equations* E*, the set of* Newman equations Neqns(E) *is inductively defined as follows.*

E	Neqns(E)
$\emptyset$	$\emptyset$
$\{f(\vec{t}) \doteq g(\vec{q})\} \cup E'$	$\text{Neqns}(f(\vec{t})) \cup \text{Neqns}(g(\vec{q})) \cup \{U \doteq V\} \cup \text{Neqns}(E')$

Here U *is the name of* $f(\vec{t})$ *and* V *is the name of* $g(\vec{q})$*.*

So, a set of equations is transformed into a set of equations generated from the equations for terms, extended with equations that equate all names of the top terms.

Definition 6.5. *Given a set of equations* E*, we rewrite the set of Newman equations* $E' = \text{Neqns}(E)$ *as follows.*

(A) If E' *contains the equations* $N \doteq f(L_1, \ldots, L_n)$ *and* $N \doteq f(K_1, \ldots, K_n)$*, then perform the substitution* $[L_i := K_i]$ *for some* i *such that* $L_i \neq K_i$ *on all equations in* E'*.*

(B) If E' *contains the equations* $N \doteq f(L_1, \ldots, L_n)$ *and* $M \doteq f(L_1, \ldots, L_n)$*, then perform the substitution* $[N := M]$ *on all equations in* E'*.*

In the definition of rule (A), it is also possible to consecutively perform substitutions $\sigma_1, \ldots, \sigma_n$, where $\sigma_1 = [L_1 := K_1]$, $\sigma_2 = [L_2 := \sigma_1(K_2)]$, $\ldots$, $\sigma_n = [L_n := \sigma_{n_1}(\ldots(\sigma_1(K_n)\ldots)]$ on all equations in E'. While this might be more efficient it does not correspond to Newman's original method.

Definition 6.6. *Given a set of equations* E*, the set of equations* E *has a* conflict *if it contains equations* $U \doteq f(\vec{L}), U \doteq g(\vec{K})$ *with* $f \neq g$*. The set of equations* E *has a* circularity *if it contains a set of equations of the form* $\{N_1 \doteq f_1(\ldots, L_1, \ldots), L_1 \doteq f_1(\ldots, L_2, \ldots), \ldots, L_n \doteq f_n(\ldots, N_1, \ldots)\}$*. Moreover, the set of equations* E *is* stratified *if it contains no conflict and no circularity.*

We have seen that rule (B) can be omitted for $\lambda\!\!\rightarrow$ (Corollary 3.22). Here, the situation is the same, rule (B) is not needed to compute the most general unifier of a set equations: a set of equations E in (A)-normal form is stratified iff the (B)-normal form of E is stratified. However, there may be cases where one can only perform a (B) step, so the rule is not superfluous if one wants to reduce the number of equations as much as possible.

Definition 6.7. *Given a set E of equations on first order terms,* unification à la Newman *is defined as the process that does the following.*

1. *Generate the set of Newman equations* Neqns(E).
2. *Rewrite the set* Neqns(E) *according to the rules of Definition 6.5.*
3. *If no rule applies (nor (A), nor (B)), we terminate, say with a set of equations E', and then*
   - *we stop with failure in case E' contains a circularity or a conflict,*
   - *otherwise we create the answer substitution by composing the substitutions we have performed along the way with E'.*

Proposition 6.8. *Unification à la Newman decides whether a set of equations E is unifiable, and if so, it computes the most general unifier of the set E.*

The proof is basically the same as for Theorem 4.16. However, there is some additional work, because we can now also fail because of a conflict. In simple type theory, there is only one function symbol (the type constructor $\rightarrow$), so the only way one can fail is due to a circularity.

In this section we have shown that Newman's typing algorithm implicitly includes all the basic ideas for a unification algorithm. We have used the DAG representation of terms to present the idea behind the algorithm and to precise the connection with our graphical presentation of Newman's typing algorithm in Section 2. The space complexity of this unification algorithm à la Newman is very good because a blow-up in size is prevented since we never substitute a compound term. However, the algorithm is non-deterministic since we do not explicitly specify how to choose equations and how to check for circularity or conflicts. For an example of an efficient and completely worked out algorithm for unification of DAGs we refer to [2]. Their algorithm contains a clever representation of equivalence classes of congruent nodes and stores some additional information in the nodes in order to obtain an almost linear time complexity. Moreover, for linear time algorithms we refer to [16, 11].

7. Conclusions

We have studied Newman's typability algorithm and shown that it is correct for $\lambda\!\!\rightarrow$ and that it implicitly computes the principal type of the term. It turns out that Newman's method lends itself very well to a graphical representation and that it is quite flexible in that it can be extended to other type constructions and stronger type disciplines like weakly polymorphic λ-calculus. In its

modular set-up – even though that is not made so explicit by Newman itself – the algorithm is closely related to Wand's, even though the actual generation of the set of type equations is performed in a different way.

Looking back, Newman was ahead of his time: not only did he define a typing algorithm, but his algorithm also includes the basics of an efficient unification algorithm. In his case, this was just used to solve type equations, but it generalizes directly to other equations, as we have shown.

An interesting question is whether Newman's algorithm can be generalized to cover a polymorphic let-construction. Then one has terms $\texttt{let}\ x = N\ \texttt{in}\ P$, where N can be of a weakly polymorphic type. For typing, the situation is now fundamentally asymmetric: the type computed for N is "passed on to" the computation of the type for P. In Newman's algorithm, we are not obliged to first normalize the graph of N and then continue (via a kind of "copying step" of the graph of N) with the graph of P, so it is not so clear how to adapt Newman's method to include let-polymorphism. On the other hand, it is also not straightforward to extend constraint based typing algorithms (for example Wand's algorithm) to include let-polymorphism. Recent research [8, 15, 17] has shown that constraint based typing algorithms can be extended to let-polymorphism by not only considering *equality constraints* but also *instantiation constraints*. It would be interesting to see whether Newman's method can be extended to such constraints and therefore could be applied to let-polymorphism.

Acknowledgments. We are grateful to the anonymous referees who provided several helpful suggestions. Also, we thank James McKinna for comments on early drafts of this paper.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[2] F. Baader and W. Snyder. Unification Theory. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 447–533. Elsevier Science Publishers, 2001.

[3] H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[4] H. P. Barendregt. *The lambda calculus: its syntax and semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, 1984.

[5] A. Church. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[6] A. Church. Review of [14]. *The Journal of Symbolic Logic*, 9:50–52, 1944.

[7] H. Geuvers. Introduction to type theory. In *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, Revised, Selected Papers*, volume 5520 of *LNCS*. Springer, 2009.

[8] F. Henglein. *Polymorphic Type Inference and Semi-Unification*. PhD thesis, Rutgers University, 1989.

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

[10] J. R. Hindley. M. H. Newman's Typability Algorithm for Lambda-calculus. *Journal of Logic and Computation*, 18(2):229–238, 2008.

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

[12] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[13] M. H. A. Newman. On Theories with a Combinatorial Definition of "Equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

[14] M. H. A. Newman. Stratified Systems of Logic. *Proceedings of the Cambridge Philosophical Society*, 39(2):69–83, 1943.

[15] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[16] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.

[17] F. Pottier and D. Rémy. The Essence of ML Type Inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[18] W. V. Quine. New Foundations for Mathematical Logic. *American Mathematical Monthly*, 44:70–80, 1937.

[19] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, January 1965.

[20] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

[21] M. Wand. A Simple Algorithm and Proof for Type Inference. *Fundamenta Infomaticae*, 10:115–122, 1987.