Polymorphic type checking for the type theory of the *Principia Mathematica* of Russell and Whitehead

M. Randall Holmes

November, 2011

This is a brief report on results reported at length in our paper [2], made for the purpose of a presentation at the workshop to be held in November 2011 in Cambridge on the *Principia Mathematica* of Russell and Whitehead ([?], hereinafter referred to briefly as *PM*).

That paper grew out of a reading of the paper [3] of Kamareddine, Nederpelt, and Laan. We refereed this paper and found it useful for checking their examples to write our own independent computer type-checker for the type system of *PM* ([1]), which led us to think carefully about formalization of the language and the type system of *PM*

A modern mathematical logician reading *PM* finds that it is not completely formalized in a modern sense. The type theory in particular is inarguably not formalized, as no notation for types is given at all! In *PM* itself, the only type annotations which appear are occasional numerical indices indicating order; the type notation we use here extends one introduced later by Ramsey. The authors of *PM* regard the absence of explicit indications of type as a virtue of their system: they call it "systematic ambiguity"; modern computer scientists refer to this as "polymorphism".

The language of *PM* is also not completely formalized, and it is typographically inconvenient for computer software to which ASCII input is to be given. The notation of *PM* for abstractions (propositional functions) does not use head binders; the order of the arguments of a complex expression is determined by the alphabetical order of the bound variables. For example $\hat{a} < \hat{b}$ is the "less than" relation while $\hat{b} < \hat{a}$ is the "greater than" relation (this is indicated by the alphabetical order of the variables). In *PM*, the fact that a variable is bound in a propositional function is indicated by circumflexing it. Variables bound by quantifiers are not circumflexed. A feature of the notation of [3], carried over into ours, is that no circumflexes are used: notations for propositions and the corresponding propositional functions are identical. This turns out to work reasonably well, as propositions usually do not occur as components of other propositions in the contexts where propositional functions do. A further apparent notational limitation which is also true to the usage of *PM* is that a composite term never appears in applied position: terms $x_1(a_1, R_1(x_2))$ in which a propositional function represented by a variable is applied to a list of arguments do appear, but any attempt to replace x_1 here with a specific propositional function (for example $x_4(x_3)$) would force the actual substitution of the arguments of x_1 into the propositional function replacing it: the result in this case would be $R_1(a_1)$, which the reader may work out. (R_1 is a unary relation constant in this example.)

The logical world of *PM* is inhabited by *individuals* and *propositional functions* (pfs). We provide constants a_i to represent individuals. We provide general variables x_i to represent objects whose type may be deduced from context.

Notations for propositions and for pfs are the same, due to our not using circumflexes, with the qualification that a propositional notation in which no variable is free will not be taken to represent a pf here (our theory and our software do support the option of allowing 0-ary pfs, but here we exclude them). A *term* is defined as a notation which is either an individual constant, a variable, or a pf. We briefly and informally describe our notation, omitting many details. An atomic proposition is of the form $R_n(A_1, \ldots, A_n)$, in which a constant n-ary predicate is applied to a list of n terms, none of which may be pfs. If P and Q are propositional notations, then $P \vee Q$ and $\neg P$ are propositional notations (other propositional connectives can be added). If x_i occurs free in P, $(\forall x_i.P)$ is a propositional notation (universal quantifier; the existential quantifier $(\exists x_i.P)$ can be added: `[xi]P` is the shape this takes on the computer). In a completely formal definition, the notion of a free occurrence of a variable in a propositional notation would be defined recursively simultaneously with the notion of "propositional notation". If $A_1, \ldots, A_n$ are terms, $x_i(A_1, \ldots, A_n)$ and $x_i!(A_1, ..., A_n)$ are propositional notations. Both of these indicate application of a pf to a list of terms. It is worth noting this clause of the definition of free occurrence of a variable: the free variables in $x_i(A_1, \ldots, A_n)$ or $x_i!(A_1, ..., A_n)$ are just x_i and those A_i's which are variables: variables occurring in any of the A_i's which are composite and thus pfs are bound. The distinction between these two notations for application is that in the second one the "order" of the applied variable is presumed to be minimal, if we are working in the ramified theory of types: this adapts a notation found in `PM`.

We now describe the definition of the notion of substitution: let $A[B_1, \ldots B_k | y_1 \ldots y_k]$ denote the result of substituting B_i for y_i for each i simultaneously, where each y_i is actually some variable x_j and distinct y_i's are distinct x_j's. If A is an elementary proposition $R_n(C_1, \ldots, C_n)$, replace each y_i which occurs as one of the C_i's with the corresponding B_i, with the caveat that if a y_i occurs among the C_i's for which A_i is a pf, the substitution is undefined. Substitution into negations and disjunctions is unproblematic. Substitution into $(\forall x_i.P)$ is almost unproblematic: substitution of any term for x_i is ignored, but all other substitutions are carried out on P. In more familiar systems, there is a problem if some B_k contains the variable x_i, but this is not the case here: in any context in which the variable y_k can appear, replacing y_k with B_k will give a pf term in which the occurrences of x_i in B_k are bound in B_k or in even smaller scopes. The variable cannot be captured by the quantifier.

Substitution of a term B_k for the variable $y_k = x_i$ in the context $x_i(C_1, \ldots, C_n)$ is the interesting clause. This can only be done if A_k is a variable (in which case we just replace it) or if it is a propositional notation with exactly n free variables $z_1, \ldots, z_n$ (given in increasing order of their actual index), in which case the result is $A_k[C_1, \ldots C_n|z_1, \ldots z_n]$. Otherwise the substitution is undefined.

So far we have defined both notation of our theory and substitution without reference to type. At this point we have a serious problem. The definition of substitution fails! For the notation $\neg x_1(x_1)$ is a propositional notation so far as we can tell up to this point, and the attempt to replace the variable x_1 in $\neg x_1(x_1)$ with $\neg x_1(x_1)$ itself (to compute $\neg x_1(x_1)[\neg x_1(x_1), x_1]$) leads to an infinite regress. In effect, we have reproduced Russell's paradox (or perhaps more accurately, Curry's paradox) in this formalism.

Introducing types fixes this. We will start anachronistically with the simple theory of types of Ramsey (without orders). We provide the type 0 of individuals and the type () of propositions. We provide a variable type $[x_i]$ to be assigned to the variable x_i whenever we cannot determine its type (this is a provision for polymorphism). If we have a list of types $\tau_1, \ldots, \tau_n$ we provide the notation $(\tau_1, \ldots, \tau_n)$ for the n-ary relation type for which the ith argument has type i.

We then assign types to notations recursively. The general form of a type judgment is that a term t is assigned type τ as a subterm of a larger term T (which we call the context). A constant a_i has type 0 as a subterm of any T. A variable x_i is assigned type $[x_i]$ as a subterm of any T (and may be assigned other types, see below). An elementary propositional notation $R_1(A_1, \ldots, A_n]$ has a type $(0, \ldots, 0)$ where the number of zeroes is the number of distinct variables occurring among the `Ai`'s, or `()` if no variables occur; in addition, any variables occuring in the argument list are assigned type 0 in any context including $R_1(A_1, \ldots, A_n]$. In a notation $x_1(A_1, \ldots, A_n)$ or $x_1!(A_1, \ldots, A_n)$, suppose the terms A_i can be assigned types τ_i in the current context. We then assign the type $(\tau_1, \ldots, \tau_n)$ to x_i in this context. Suppose that $\sigma_1, \ldots, \sigma_m$ is the list of types assigned to the distinct variables among the A_i's and x_i, listed in index order: then we can assign the type $(\sigma_1, \ldots, \sigma_m)$ to $x_1(A_1, \ldots, A_n)$ or $x_1!(A_1, \ldots, A_n)$. In any other pf term, let $\sigma_1, \ldots, \sigma_m$ be a list of types assigned to the variables occurring free in the notation listed in index order: assign the type $(\sigma_1, \ldots, \sigma_m)$ to this notation. If no variables are free at all, assign the type (). Note that we only assign types to pf subterms of a context if they are either the whole context or one of the arguments of a pf application term. Finally, we only regard a context term as well-typed if all types assigned to each of its subterms are compatible, including additional type assignments which are forced in a way we now describe. The type 0 is compatible only with 0 and types $[x_i]$. The type $(\tau_1, \ldots, \tau_n)$ is only compatible with types $[x_i]$ where $[x_i]$ does not occur in $(\tau_1, \ldots, \tau_n)$ in the obvious sense and with types $(\sigma_1, \ldots, \sigma_n)$ where each τ_i is compatible with σ_i. The type $[x_i]$ is compatible with any type in which $[x_i]$ itself does not occur as a proper component. Further, whenever type $[x_i]$ is assigned to a subterm A, type t is assigned to the same subterm A, and a type u is assigned to any other term B, the result $u[t/[x_i]]$ of replacing all occurrences of x_i with t in u is also assigned to B. A term is typable if it is assigned a type by this process and any two types assigned to the same subterm by this process are compatible.

The idea here is that types are computed by unification in a way familiar from the polymorphic type systems of computer languages like ML. It is straightforward to establish that a term which can be assigned a type can be assigned a single most general type from which all specific types that can be assigned to it can easily be determined. Further, it is straightforward to establish that the definition of substitution works correctly if one stipulates that to be well-formed a term must be typable and that types of B_k's considered as their own context must be compatible in a suitable sense with types of y_k's as subterms of the context A in a substitution $A[B_1, \ldots B_k | y_1 \ldots y_k]$.

There are some interesting technical points. Although variables appearing in propositional functions appearing as arguments of propositional function application terms are bound, the type algorithm can lead to difficulties (bound variable collisions) as the algorithm unifies variable types. This can be avoided by renaming bound variables before the type algorithm is applied.

The implementation of the simple theory of types in software was quite straightforward: it is both suggested by and served to inform the development of the formalization outlined above. The ramified theory of types (the full type system of `PM` with orders) presents more difficulties.

The motivation behind the ramified theory is as follows. The type of a propositional function in the simple theory of types is determined by the types of its arguments, and all types of its arguments must be simpler than its type: understanding the meaning of a propositional function involves understanding the entire range of the types of its arguments, so it cannot without circularity be a component of one of those types. The authors of *PM* hold that understanding the meaning of a pf involves prior understanding of the type over which any quantified variable appearing in the pf ranges. More concretely, Russell suggests in *PM* that a quantified sentence is to be understood as expressing an infinitary conjunction or disjunction in which sentences referring to every object of the type quantified over must occur. If quantified sentences are to be interpreted in this way, then the appearance of a quantified variable in a pf with the same type as the pf or a more complex type would lead to formal circularity on expansion to infinitary form.

The restriction is enforced in *RTT* by adding to each type a new feature, a non-negative integer called its "order". The order of type 0 (the type of individuals) is 0. The type () of propositions in simple type theory is partitioned into types $()^n$ for each natural number n, where the order n will be the least natural number greater than the order of the type of any variable which occurs in the proposition (including quantified variables). A pf notation P containing n free variables x_{i_k} (listed in increasing order) with types t_k will be assigned type $(t_1, \ldots, t_n)^m$, where m is the smallest natural number greater than the order of any of the types t_k and the order of the type of any variable quantified in P. A similar rule applies to the typing of head variables x_i in expressions $x_i(A_1, \ldots, A_n)$ or $x_i!(A_1, \ldots, A_n)$: the type of x_i will be $(t_1, \ldots, t_n)^r$ where each t_k is the type of A_k, and the order r is larger than the orders of the t_k's; in the term $x_i!(A_1, \ldots, A_n)$, the order r must be the smallest order larger than all orders of t_k's.

The technical difficulty which arises, which we will not go into in the brief space of this report, but which we will try to give a flavor of in our presentation by considering examples, is that unification of polymorphic types with orders has a kind of complexity not usually found in sensible type systems. Each polymorphic type variable has an unknown order, and unification will generate a set of arithmetic inequalities relating the orders of the polymorphic variables. The final version of the software implements an algorithm which will precisely describe the conditions on orders of polymorphic types which are needed for a term to be typable, but these can be rather complicated. We have a partial algorithm which will very often generate a ramified type which will work without inequalities on the orders of polymorphic type as side conditions (it worked for all examples in [3]). This partial algorithm is the one we plan to use in our proposed proof checker for the language of *PM*, but the complete type checking algorithm was quite interesting and challenging to develop.

References

[1] Holmes, M. Randall, software files (in standard ML) `rtt.sml` (source for the type checker) and `rttdemo.sml` (demonstration file), accessible at `http://math.boisestate.edu/~holmes/holmes/rttcover.html`.

[2] Holmes, M. Randall, "Polymorphic type–checking for the ramified theory of types of Principia Mathematica", in Fairouz Kamareddine, ed., Thirty–five Years of Automating Mathematics, Kluwer, 2003, pp. 173-215.

[3] Kamareddine, F., Nederpelt, T., and Laan, R., "Types in mathematics and logic before 1940", *Bulletin of Symbolic Logic*, vol. 8, no. 2, June 2002.

[4] Whitehead, Alfred N. and Russell, Bertrand, *Principia Mathematica (to *56)*, Cambridge University Press, 1967.