Polymorphic type-checking for the ramified theory of types of *Principia Mathematica*

M. Randall Holmes [1]

Department of Mathematics, Boise State University, 1910 University Drive, Boise, Idaho 83725-1555, USA

Abstract

A formal presentation of the ramified theory of types of the *Principia Mathematica* of Russell and Whitehead is given. The treatment is inspired by but differs sharply from that in a recent paper of Kamareddine, Nederpelt and Laan. A complete algorithm for determining typability and most general polymorphic types of propositional functions of the ramified theory of types is presented, unusual in requiring reasoning about numerical inequalities in the course of deduction of type judgments (to support unification of orders). Software implementing these algorithms has been developed by the author, and examples of the use of the software are presented. This is an abridged version of a longer paper which may appear later elsewhere.

This paper was inspired by reading [3], where Kamareddine, Nederpelt and Laan present a formalization of the ramified theory of types (usually to be abbreviated *RTT*) of [5], the *Principia Mathematica* of Russell and Whitehead (hereinafter *PM*). It is surprising that the theory of types of *PM* (the oldest one) is nowhere given a rigorous formal description; in fact, *PM* has no notation for types! There are various formal systems of ramified type theory in the literature (the author has even presented one in [1]), but the one in [3] is the only one known to us that is close to *PM* in details of its notation.

While reading [3], we developed a type checker ([2]) for its version of *RTT*. We used the same notation for propositional functions that is used in [3] (except that we were able to omit type labels on quantified variables, which makes our notation closer to that of *PM*), but we took a quite different approach to reasoning about types. From the checker it is possible to "reverse engineer" a formal treatment of the type system of *RTT* different from that of [3], which we give here.

This is an abridged version of a longer paper which we hope to publish elsewhere. Here we omit a section which discusses differences between the

[1] Email: holmes@math.boisestate.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

notation of [3] and the original notation of *PM*.

The logical world of *PM* is inhabited by *individuals* and *propositional functions*. We will usually abbreviate "propositional function" as "pf" (following [3]). In this section we introduce notation for these.

An individual is denoted by one of the symbols $a_1, a_2, a_3, \ldots$ (in the computer implementation, `a1, a2, a3...`). We call these symbols "individual constants".

Before we present the notation for propositions, we need to introduce variables and primitive relation symbols. A variable is one of the symbols $x_1, x_2, x_3, \ldots$ (`x1, x2, x3...` in the computer implementation). A primitive relation symbol is a string of upper-case letters with a numerical subscript indicating its arity (in the paper, R_1 and S_2 are primitive relation symbols: these would be `R1` and `S2` in the computer implementation).

We note that we will freely use the word "term" in the sequel for any piece of notation, whether propositional notation, the name of an individual, or a variable.

Now we present the definition of notation for propositions. The notion of free occurrence of a variable in a proposition is defined at the same time. It is worth noting here that notation for a proposition is usually but not always also notation for a propositional function (pf).

atomic proposition: A symbol $R_n(v_1, \ldots, v_n)$ consisting of a primitive relation symbol with arity n followed by a list of n arguments v_i, each of which is either a variable x_{j_i} or an individual constant a_{j_i}, is an atomic proposition. ($R_0()$ is also an atomic proposition in the system of [3], and for us as well for now. The software that motivates this paper supports the ability to turn on or off a requirement that primitive relation symbols and propositional functions have positive arity). The free occurrences of variables in an atomic proposition are exactly the typographical occurrences of variables in it.

negation: If P is a proposition, then $\neg P$ (`~P` in the computer implementation) is a proposition, the negation of the proposition P. The free occurrences of variables in $\neg P$ are precisely the free occurrences of variables in P.

binary propositional connectives: If P and Q are propositions, then $(P \vee Q)$ is a proposition. Other connectives can be defined. In the computer implementation, propositional connectives are strings of lower case letters: `(P v Q), (P implies Q), (P and Q), (P iff Q)`. The free occurrences of variables in $(P \vee Q)$ are the free occurrences of variables in P and Q; defined binary connectives would have the same rule.

quantifiers: If P is a proposition in which the variable x_i occurs free (this condition is what requires us to define "free variable" at the same time as "propositional notation"), $(\forall x_i.P)$ is a proposition (this is written `[xi]P` in the computer implementation). The existential quantifier $(\exists x_i.P)$ (written

`[Exi]P` in the computer implementation) can be introduced by definition. The free occurrences of variables in $(\forall x_i.P)$ are the free occurrences of variables other than x_i in P (and similarly for any other quantifier).

In [3], the structure of the typing algorithm required the attachment of explicit type labels to variables bound by quantifiers. In our system, this is not necessary. This is closer to the situation in *PM*, where no type indices appear (There is no notation for types in *PM*, so there can't be type indices; there are occasional appearances of numerical superscripts representing "orders").

propositional function application ("matrix" and general): If x_i is a variable and $A_1, \ldots, A_n$ is an argument list in which each A_i is of one of the forms a_{j_i} (an individual constant), x_{j_i} (a variable) or P_i (notation for a proposition, representing a pf), then $x_i(A_1, \ldots, A_n)$ and $x_i!(A_1, \ldots, A_n)$ are propositions. In the latter notation, the exclamation point indicates that the "order" of the type of the variable x_i is as low as possible: this will be clarified when types and orders are discussed. The notation $x_i!(A_1, \ldots, A_n)$ does not appear in [3]; its use in this paper is a generalization of the notation for "matrices" (predicative functions) in *PM*. $x_i()$ is also a proposition in the system of [3] (the variable x_i represents a proposition in this case); $x_i()$ and $x_i!()$ are propositions for us as well for now: if we require that primitive relation symbols and pfs have positive arity, then we exclude such propositions). The free occurrences of variables in $x_i(A_1, \ldots, A_n)$ or $x_i!(A_1, \ldots, A_n)$ are the head occurrences of x_i and those A_i's which are variables: note that the free occurrences of variables in those A_i's which are pf notations are *not* free occurrences of variables in $x_i(A_1, \ldots, A_n)$ or $x_i!(A_1, \ldots, A_n)$.

completeness of definition: All propositional notations are constructed in this way.

As usual, an occurrence of a variable in a proposition which is not free is said to be bound. Note that a variable x_i is not a propositional notation.

The notation for a propositional function is the same as the notation for a proposition: a pf is construed as a function of the variables which appear in it (or rather of the variables which appear free in it). When 0-ary predicates are forbidden (this is arguably the case in *PM* (see remark on p. 38) and is supported as an option by our checker), a propositional notation must contain a free variable to represent a pf; otherwise a propositional notation without free variables will represent a 0-ary propositional function. The full system of the checker also allows propositional variables (which are used in *PM*) but does not allow occurrences of propositional variables in pfs.

Since we do not have head binders in the notation for pfs to determine the order of multiple arguments, we allow the order of the indices of the variables (which we may refer to occasionally as "alphabetical order") to determine the order in which arguments are to be supplied to the function. This follows *PM*.

We refer to the atomic propositions and the propositional function applica-

tion terms as "logically atomic", and to other terms as "logically composite".

We now give the recursive definition of simultaneous substitution of a list of individuals, variables and/or propositional functions A_k for variables x_{i_k} in a proposition P, for which we use the notation $P[A_k/x_{i_k}]$. The clauses of the definition follow the syntax. It is required that the subscripts i_k be distinct for different values of k.

atomic propositions: Let $R_n(v_1, \ldots, v_n)$ be an atomic proposition. For each v_i and index k, define v_i' as A_k if v_i is x_{i_k}; define v_i' as v_i if v_i is not any x_{i_k}. If any v_i' is a pf notation, $R_n(v_1, \ldots, v_n)[A_k/x_{i_k}]$ is undefined; otherwise $R_n(v_1, \ldots, v_n)[A_k/x_{i_k}]$ is defined as $R_n(v_1', \ldots, v_n')$.

negation: $(\neg P)[A_k/x_{i_k}] = \neg(P[A_k/x_{i_k}])$

binary propositional connectives: $(P \vee Q)[A_k/x_{i_k}] = (P[A_k/x_{i_k}] \vee Q[A_k/x_{i_k}])$. The rule is the same for any binary propositional connective.

quantification: Let $(\forall x_j.P)$ be a quantified sentence (the rule is the same for any quantifier). Define A_k' as x_j in case $i_k = j$ and as A_k otherwise. Then $(\forall x_j.P)[A_k/x_{i_k}]$ is defined as $(\forall x_j.P[A_k'/x_{i_k}])$.

pf variable application: Let $x_j(V_1, \ldots, V_n)$ or $x_j!(V_1, \ldots, V_n)$ be a proposition built by application. Define B' for any notation B as A_k if B is x_{i_k} and as B otherwise. We define $x_j(V_1, \ldots, V_n)[A_k/x_{i_k}]$ as $x_j'(V_1', \ldots, V_n')$ and $x_j!(V_1, \ldots, V_n)[A_k/x_{i_k}]$ as $x_j'!(V_1', \ldots, V_n')$ except in the case where x_j' is a pf notation Q: in this case something rather more complicated happens. It will be undefined unless there are precisely n variables which occur free in Q. If there are n variables which occur free in Q, define t_k so that x_{t_k} is the kth free variable in Q in alphabetical order. Then define $x_j(V_1, \ldots, V_n)[A_k/x_{i_k}]$ or $x_j!(V_1, \ldots, V_n)[A_k/x_{i_k}]$ as $Q[V_k'/x_{t_k}]$.

There is a serious difficulty with this "definition". Consider the pf $\neg x_1(x_1)$ (this certainly is a pf by our definition above). Now substitute $\neg x_1(x_1)$ for the variable x_1 in the proposition $\neg x_1(x_1)$ itself. We will obtain the negation of the result of replacing x_1 with $\neg x_1(x_1)$ in $x_1(x_1)$. Giving $\neg x_1(x_1)$ the name R for the moment, we see that the result of the latter substitution will be $R[R/x_1]$; but this is exactly the substitution we started out trying to make, so we have landed in an infinite regress. This illustrates the fact that the circularity of the proposed "definition" of substitution is essential – in the last clause, there is no guarantee that the instance of substitution $Q[V_k'/x_{t_k}]$ to be carried out is "simpler" in any way than the original substitution $x_j'(V_1, \ldots, V_n)[A_k/x_{i_k}]$ being defined, and our example shows that it need not be.

It is hoped that the reader will notice that this is essentially Russell's paradox of naive set theory. Our solution will be the official solution of *PM*: we will impose a type system, under which the term $\neg x_1(x_1)$ will fail to denote a pf, and the problem will disappear. For the moment, we withdraw the definition of substitution, and will return to it after we have presented the type system.

The self-contained approach to the definition of substitution taken here may be contrasted with the rather elaborate invocation of λ-calculus in [3]. Though our definition appears to have failed at this point, the type system will allow us to give the definition above as a legitimate inductive definition. The reason we can do this and the authors of [3] cannot is that their definition of the typing algorithm depends on the notion of substitution, and ours does not. (Our algorithm does depend on the notion of substitution into notations for *types*, as we will see below, but the definition of substitution into types does not present logical difficulties presented by the definition of substitution into propositions or pfs).

We follow [3] in presenting the simple theory of types without orders first, though historically it was presented by Ramsey as a simplification of the ramified theory of types of *PM*.

The base type of the system of *PM* is the type 0 inhabited by individuals. (Nothing prevents the adoption of additional base types, or indeed the avoidance of commitment to any base type at all).

All other types are inhabited by propositional functions. In the simple theory of types, the type of a pf is determined precisely by the list of types of its arguments.

We introduce notation for simple types:

Individuals: 0 is a type notation.

Propositions: () is a type notation (for the type of propositions).

Propositional Functions: If $t_1, \ldots, t_n$ are type notations, $(t_1, \ldots, t_n)$ is a type notation. (If pfs were required to have positive arity, we would require $t_i \neq ()$ here).

Variable Types: For each variable x_i, we provide a type notation $[x_i]$. (This notation is an innovation for this paper: it represents an unknown (polymorphic) type to be assigned to x_i; these types may also be called "polymorphic types").

Completeness of Definition: All simple type notations are derived in this way.

No Nontrivial Identifications: Types not containing variable types are equal precisely if they are typographically identical.

As is noted in [3], there is no notation for types in *PM*: this notation is apparently due to Ramsey (except for our innovation of variable types, whose purpose will become clear below).

Our aim in this essay is to avoid the necessity of assigning types overtly to variables, which is truer to the approach taken in *PM* itself. It is useful to consider what a system with explicit type assignment would look like, though.

The type assignment is represented as a partial function from terms to types: $\tau(x_i)$ is the type to be assigned to x_i, and more generally $\tau(t)$ is the type to be assigned to the individual constant, variable, or propositional function

t. Types in the range of τ are constant types (they contain no type variables $[x_i]$). We require that bound variables be typed as well as free variables, and identity of variables implies identity of type regardless of free or bound status. We stipulate that every variable is in the range of τ and that the inverse image of each type under τ contains infinitely many variables: this has the same effect as providing infinitely many variables labelled with each type. The following rules simultaneously tell us which terms are typable (have values under τ) and how to compute the value of τ if there is one. Functions τ satisfying these rules are called "type functions on P", where P is a fixed proposition or propositional function.

individuals: If x_i appears as an argument in an atomic subproposition of P, $\tau(x_i) = 0$. $\tau(a_i) = 0$ for any individual constant a_i.

propositional functions: If Q is a propositional function appearing as a subterm of P, every subterm of Q has a value under τ, and the n free variables of Q, indexed in increasing order, are x_{i_k}, $\tau(Q) = (\tau(x_{i_1}), \ldots, \tau(x_{i_n}))$. If Q contains no free variables, then $\tau(P) = ()$.

variable application: If $x_j(A_1, \ldots, A_n)$ or $x_j!(A_1, \ldots, A_n)$ is a subterm of P, then $\tau(x_j) = (\tau(A_1), \ldots, \tau(A_n))$.

These rules are to be understood as additional restrictions on well-formedness of terms: a term P is to be considered well-formed iff there is a type function τ on P. Notice that the value of τ at every term (or its lack of value) is completely determined by the values of τ at variables. The process described terminates by induction on the structure of propositional notations: to compute the type (or assess the typability) of any notation other than a variable or individual constant, we appeal only to the types of proper subterms of that notation, and we are given types of variables and individual constants at the outset.

We now proceed to develop a system for expressing and reasoning about type assignments to subterms of propositional functions, adopting rules on the basis of their validity for an intended interpretation in terms of type functions.

There are four kinds of type judgments. In the following, P stands for a propositional function or proposition, t, u stand for types (variable types $[x_i]$ are permitted to appear as types and as components of complex types) and x_i stands for a variable. The meanings of these judgments will be modified by a redefinition of the notion of "type function on P" which will be given below.

ill-typedness: "P is ill-typed" is defined as "there is no type function τ on P".

propositional function type assignment: "P has type t" means "for all type functions τ on P, $\tau(P) = t$", where any type $[x_i]$ appearing in t is interpreted as $\tau(x_i)$.

variable type assignment: "x_i has type t in P" means "for all type functions τ on P, $\tau(x_i) = t$", where any type $[x_j]$ appearing in t is interpreted

as $\tau(x_j)$.

type equality: "$t = u$ in P" is defined as "for all type functions τ on P, $t = u$", where any type $[x_j]$ appearing in t or u is interpreted as $\tau(x_j)$.

We now develop rules for deduction about type judgments, showing that the rules are valid in the intended interpretation.

We begin with the observation that the conditions defining a type function on P depend only on the appearances of variables in logically atomic subterms of P: these conditions assign types to arguments appearing in atomic propositions, to propositional functions, which can only appear as arguments of propositional function application terms, and to the head variables of propositional function application terms. It follows immediately from this that τ is a type function on P under precisely the same conditions under which it is a type function on $\neg P$ or on $(\forall x_i.P)$ (if the latter is well-formed), since these terms contain precisely the same logically atomic subterms. Further, it follows that any type function on $(P \vee Q)$ is also a type function on P and on Q, since it will satisfy the conditions on logically atomic subterms of P and Q, since the set of logically atomic subterms of $(P \vee Q)$ is the union of the set of logically atomic subterms of P and the set of logically atomic subterms of Q.

These facts can be expressed as rules for reasoning about type judgments:

negations: $\neg P$ is ill-typed iff P is ill-typed. x_i has type t in $\neg P$ iff x_i has type t in P.

quantification: $(\forall x_i.P)$ (if well-formed) is ill-typed iff P is ill-typed. x_j has type t in $(\forall x_i.P)$ iff x_j has type t in P.

binary propositional connectives: If P or Q is ill-typed, $(P \vee Q)$ is ill-typed. If x_i has type t in P or x_i has type t in Q, then x_i has type t in $(P \vee Q)$.

There are three kinds of occurrences of variables in logically atomic subterms: a variable can appear as an argument of an atomic proposition, as the head variable of a pf application term, or as an argument of a pf application term. The following rules express the type judgments we can make about occurrences of variables in each context:

individual variables: If $x_i = A_k$ in $R_n(A_1, \ldots, A_n)$, then x_i has type 0 in $R_n(A_1, \ldots, A_n)$.

applied variables: If A_i has type t_i for each i, then x_j has type $(t_1, \ldots, t_n)$ in $x_j(A_1, \ldots, A_k)$ or $x_j!(A_1, \ldots, A_k)$.

argument variables: x_i has type $[x_i]$ in P for any propositional function P (this kind of occurrence gives us no type information).

In this way a possibly variable type may be assigned to each occurrence of a variable on the basis of its logically atomic context. This is called the "local" type of the occurrence. However, more than one typographically different type

may be assigned to the same variable. For example, x_1 is assigned type 0 and type $[x_1]$ in $R_1(x_1) \vee x_2(x_1)$. Different types assigned to the same variable will of course be equal. We can express this in terms of type judgments.

multiple types: If x_i has type t in P and x_i has type u in P then $t = u$ in P.

variable type equations: If $[x_i] = t$ in P then x_i has type t in P.

Definition: We assign an integer *arity* to each type which is not a type variable. 0 has arity -1. () has arity 0. $(t_1, \ldots, t_n)$ has arity n. Note that a type may have variable type components, but it will still have arity if it is not itself a type variable. Note also that types which are equal will have equal arity if their arity is defined.

type distinction: If t and u each have arity and have distinct arities and $t = u$ in P, then P is ill-typed.

absurdity: If P is ill-typed, then P has type t, $t = u$ in P and x_i has type t in P for any t, u, and x_i (this is obviously true under the intended interpretation – we need it for a completeness result).

componentwise equality: If $(t_1, \ldots, t_n) = (u_1, \ldots, u_n)$ in P, then $t_i = u_i$ in P for each i.

type substitution: If x_i has type t in P and x_j has type u in P, then x_j has the type $u[t/[x_i]]$ obtained by substituting t for all occurrences of $[x_i]$ in u.

A consideration related to type substitution is that no type can be ill-founded: the type of a variable x_i cannot have $[x_i]$ as a proper component.

ill-foundedness: If x_i has type t in P and $t[t/[x_i]] \neq t$, then P is ill-typed.

Finally, we need the rule for typing propositional functions.

propositional function type: If the variables free in P, listed in order of increasing index, are $(x_{i_1}, \ldots, x_{i_n})$ and x_{i_k} has type t_k for each k, then P has type $(t_1, \ldots, t_n)$.

An additional rule is stated which we do not use in the computer implementation for simple type theory (though we do use it in ramified type theory), but which is needed for a completeness result for type functions as we have defined them.

types from arguments: If x_i has type t in A_k, then x_i has type t in $x_j(A_1, \ldots, A_n)$ and $x_j!(A_1, \ldots, A_n)$.

It should be clear that each of these rules is sound for the intended interpretation. We will prove that this set of rules is complete for the intended interpretation as well.

Theorem: For each propositional function P, there is a type t such that "P has type t" is deducible from the rules above and the types possible as values $\tau(P)$ for a type function τ on P are precisely the types obtainable by substituting arbitrary types for each type variable appearing in t.

Proof of Theorem: We describe the computation of the type t. The idea is to construct a set of judgments "x_i has type t_i" deducible using the type judgment rules which satisfies all the rules for a type function except that t_i's may type variables: arbitrary instantiation of the type variables (and extension of the function to variables not appearing in P) then yields a true type function.

Begin the construction of the set of judgments by computing the "local" type of each occurrence of each variable x_i. We prove the theorem by structural induction: we assume that each pf argument of pf application terms can be assigned a type satisfying the conditions of the theorem (so that we can assign types to the head variables of these terms).

This fails to induce a type function on P (mod instantiation of type variables with concrete types) only if more than one type is assigned to the same variable. We describe a procedure for resolving such situations.

If any variable is assigned types of different arities, or if any variable x_i is assigned a type which contains $[x_i]$ as a proper component, the process terminates with the judgment that P is ill-typed.

If x_i is assigned any type t which is not a variable type (t may be a composite type with variable components) replace all occurrences of $[x_i]$ in types assigned to other variables with the type t. If x_i is assigned type $[x_j]$ ($j \neq i$), replace all occurrences of the type $x_{\mathtt{min}\{i,j\}}$ in types assigned to all variables with the type $x_{\mathtt{max}\{i,j\}}$. This is justified by the type substitution rule. In the process described below, carry out these substitutions whenever a new type assignment is made. Notice that such a substitution will occur at most once for any given variable x_i, since it eliminates the target type everywhere. Of course, if $[x_i]$ is introduced as a proper component of the type of x_i, terminate with a judgment of ill-typedness.

If x_i is assigned types $[x_j]$ and t in P, add the judgment "x_j has type t in P" and eliminate the type assignment "x_i has type $[x_j]$ in P" (note that all occurrences of $[x_j]$ will then be eliminated if t is not a type variable). In one special case we proceed differently: if x_i is assigned types $[x_j]$ and $[x_k]$, we assign x_i, x_j, and x_k the type $x_{\mathtt{max}\{i,j,k\}}$.

If x_i is assigned types $(t_1, \ldots, t_n)$ and $(u_1, \ldots, u_n)$ in P, the judgments $t_i = u_i$ follow for each relevant i. From these equality judgments continue to deduce further equality judgments in the same way. This process will terminate with either a judgment that P is ill-typed or a finite nonempty set of nontrivial judgments of the form $[x_k] = v_k$, each of which has "x_k has type v_k" as a consequence, which we add to our list of type assignments. Assign to x_i the type which results if all these types x_k are replaced with the corresponding v_k's in either of the two types being reconciled (the same type results in either case). Note that no new assignment to x_i can result, because $[x_i]$ cannot be a component of the type assigned to x_i unless P is ill-typed.

This process must terminate, because each step of the process described

eliminates at least one variable type $[x_i]$ from consideration or terminates with a judgment of ill-typedness.

When the process terminates, we will either have concluded that P is ill-typed (and this judgment will be honest because the rules are sound for the intended interpretation) or we will have obtained a set of type assignments to the variables appearing in P satisfying the conditions for a type function: any instantiation of type variables appearing in these types with constant types will give a type function on P.

It is important to note that this is a type algorithm based on the quite standard approach of type unification implemented, for example, in the type checking of the computer language *ML* (a standard reference is [4]).

We can now salvage the definition of substitution given above.

Convention: We stipulate henceforth that propositional notations are well-formed iff they are well-formed under the original definition and the judgment "P is ill-typed" cannot be deduced using the algorithm given above.

Theorem: $P[A_k/x_{i_k}]$, defined as above, will be well-defined as long as there is a fixed set of substitutions σ of types for polymorphic type variables such that the type of each A_k is the result of applying σ to the type of x_{i_k} in P.

Proof of Theorem: We only need to consider the case in which a propositional function Q is substituted for the variable x_j in a term $x_j(A_1, \ldots, A_n)$ or $x_j!(A_1, \ldots, A_n)$.

We reproduce the problematic clause from the definition of substitution.

"Let $x_j(V_1, \ldots, V_n)$ or $x_j!(V_1, \ldots, V_n)$ be a proposition built by application. We carry out the substitution of a finite list of terms A_k for corresponding variables x_{i_k}. Define B' for any notation B as A_k if B is typographically x_{i_k} and as B otherwise. We define $x_j(V_1, \ldots, V_n)[A_k/x_{i_k}]$ as $x'_j(V'_1, \ldots, V'_n)$ and $x_j!(V_1, \ldots, V_n)[A_k/x_{i_k}]$ as $x'_j!(V'_1, \ldots, V'_n)$ except in the case where x'_j is a pf notation Q: in this case something rather more complicated happens. It will be undefined unless there are precisely n variables which occur free in Q. If there are n variables which occur free in Q, define t_k so that x_{t_k} is the kth free variable in Q in alphabetical order. Then define $x_j(V_1, \ldots, V_n)[A_k/x_{i_k}]$ or $x_j!(V_1, \ldots, V_n)[A_k/x_{i_k}]$ as $Q[V'_k/x_{t_k}]$."

The type of the pf Q being substituted for x_j in P is the image under the fixed substitution σ of the type of x_j in P, and so is the image under σ of a proper component of the type of P. Thus, by a structural induction on types, the substitution $Q[V'_k/x_{t_k}])$ into Q used to define the substitution into P succeeds, because the image under σ of the type of Q is simpler than the image under σ of the type of P. Note that because P is well-typed, that substitution $Q[V'_k/x_{t_k}])$ will meet the typing conditions we require for substitutions: the fact that Q has the same type that x_j has in P, each V'_k has the same type as V_k in P, and $x_j(V_1, \ldots, V_n)$ is a subterm of P is sufficient to see this.

So the problem of substitution is solved by the adoption of simple type

theory.

The motivation behind the ramified theory is as follows. The type of a pf in STT is determined by the types of its arguments, and all types of its arguments must be proper components of its type and thus simpler than its type. It can said further (though such qualms are no longer fashionable) that understanding the meaning of a pf involves understanding the entire type over which any quantified variable appearing in the function ranges, so the type of a pf must be more complex than that of any variable over which quantification occurs in the pf. 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 propositional function of the same type as the propositional function or of 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 (zero). 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 propositional function P containing n free variables x_{i_k} (listed in increasing order) with types t_k will have 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.

Polymorphic type-checking for this system is made difficult by the fact that a polymorphic type $[x_i]$ has unknown order (denoted by $|x_i|$) and a term $x_i(A_1, \ldots, A_n)$ has only a lower bound on its order, and so it is necessary to do a certain amount of arithmetical reasoning on unknown orders. A typical order is the maximum of a natural number n and several expressions of the form $|x_i| + m$. Unification of orders is a not entirely trivial problem.

This is all made concrete as follows. We begin with the definition of formal polymorphic orders.

Natural numbers are polymorphic orders. $|x_i|$ is a polymorphic order for each x_i. Formal maxima of polymorphic orders are polymorphic orders and so is the formal sum of a polymorphic order and a natural number.

Elementary properties of maximum and addition allow us to reduce any polymorphic order to a canonical form, which will be the maximum of a single natural number (if the natural number is 0 it is omitted) and a list of expressions $|x_i| + m$ (if m is 0 it is omitted) presented in ascending order of the

parameter i. Adding a natural number to such a standard form and taking the maximum of two such standard forms are computable operations.

If m and n are polymorphic types, we say $m > n$ when $\max(m, n+1) = m$. This is not a total order, of course.

The result $u[m/|x_i|]$ of substituting a polymorphic order m for the polymorphic order $|x_i|$ in a polymorphic order u is the result of replacing the occurrence of $|x_i|$ in u (if there is one: otherwise the result of the substitution is u) with m, then simplifying to canonical form.

Substitution into orders is needed to handle changes in order which take place when a more detailed type is substituted for a polymorphic type variable.

Now we are in a position to define ramified types (and their orders, simultaneously).

individuals: 0 is a ramified type of order 0.

propositions: If n is a polymorphic order, $()^n$ is a ramified type of order n.

propositional functions: If $t_1, \ldots, t_n$ are ramified types and m is a polymorphic order greater than the order of any of the types t_k, then $(t_1, \ldots, t_n)^m$ is a ramified type of order m.

polymorphic types: For each variable x_i, there is a ramified type $[x_i]$ of order $|x_i|$.

We present the rules for a term-typing function τ as above. Notice that here the orders will be fixed non-negative integers.

individuals: If x_i appears as an argument in an atomic proposition, $\tau(x_i) = 0$. $\tau(a_i) = 0$ if a_i appears.

propositional functions: If P is a propositional function and the n free variables of P, indexed in increasing order, are x_{i_k}, $\tau(P) = (\tau(x_{i_1}), \ldots, \tau(x_{i_n}))^m$, where m is one greater than the maximum of the orders of the types of the variables appearing in P (free or bound). If P contains no free variables, then $\tau(P) = ()^m$, where m is one greater than the maximum of the orders of the types of the variables appearing in P.

variable application: If $x_j!(A_1, \ldots, A_n)$ is a term, then $\tau(x_j) = (\tau(A_1), \ldots, \tau(A_n))^m$, where m is one plus the maximum of the orders of the types of the A_i's. If $x_j(A_1, \ldots, A_n)$ is a term, then $\tau(x_j) = (\tau(A_1), \ldots, \tau(A_n))^m$, for some m greater than the order of the type of any A_i.

Notice that in the ramified theory there is an additional case where the type of a variable cannot be rigidly deduced from its context: as before, the type of a variable argument to a pf variable is polymorphic (though it may be determined from other features of the context) and in addition the order of the type of x_j in a term $x_j(A_1, \ldots, A_n)$ only has a lower bound, not a fixed value (though further information in the context might fix the order or further restrict it). This will be reflected in additional appearances of polymorphic variables in our algorithm.

We will regard a pf as well-formed when there is a type function τ which assigns a type to that function. Some pfs will have many possible types, as above, which will be indicated by the appearance of type variables $[x_i]$ and order variables $|x_i|$ in the type resulting from the algorithm.

We now develop rules for deduction about type judgments, showing that the rules are valid in the intended interpretation. Our development will be parallel to the development for simple type theory above. We present only those clauses which differ from the clauses in the development for STT.

applied variables: If A_i has type t_i for each i, and the order of t_k is o_k for each k, then x_j has type $(t_1, \ldots, t_n)^r$ in $x_j!(A_1, \ldots, A_k)$, where $r = 1 + \max(o_1, ..., o_k)$, and x_j has type $(t_1, \ldots, t_n)^s$ in $x_j(A_1, \ldots, A_k)$, where $s = \max(|x_j|, o_1 + 1, \ldots, o_n + 1)$.

Definition: We assign an integer *arity* to each type which is not a type variable. 0 has arity -1. () has arity 0. $(t_1, \ldots, t_n)^m$ has arity n. Note that a type may have variable type components, but it will still have arity if it is not itself a type variable. Note also that types which are equal will have equal arity if their arity is defined. (this clause appears simply because order appears in composite types – note that the order has no effect on the arity).

componentwise equality: If $(t_1, \ldots, t_n)^{m_1} = (u_1, \ldots, u_n)^{m_2}$ in P, then $t_i = u_i$ in P for each i. (this clause appears, again, simply because order is mentioned; it is also the case that $m_1 = m_2$ will hold, but we have no judgment of this form available to us).

type substitution: If x_i has type t in P and x_j has type u in P, then x_j has the type $u[t/[x_i]]$ obtained by substituting t for all occurrences of $[x_i]$ in u. If x_i has type t in P and $u = v$ in P, then $u[t/[x_i]] = v[t/x_i]$ in P. (this clause appears because we need substitution into equality judgments; such a rule would be valid in STT but is not needed there).

In the rules above and below, it is important to note that substitution of a type t for a type variable $[x_i]$ also has the effect of substituting the order of t for all occurrences of the order variable $|x_i|$.

ill-foundedness: If x_i has type t in P and $t[t/[x_i]] \neq t$, then P is ill-typed. (Note that the computation of $t[t/[x_i]]$ includes the reduction of its order to standard form).

There is a form of circularity which does not lead to ill-typedness: a variable x_i may have a type whose order t is a maximum of types including $|x_i|$; the calculation of $t[t/[x_i]]$ includes the simplification of the order of t, which will reduce $t[t/[x_i]]$ to t.

propositional function type: If the variables free in P, listed in order of increasing index, are $(x_{i_1}, \ldots, x_{i_n})$, and the variables quantified in P are $(x_{i_{n+1}}, \ldots, x_{i_m})$, x_{i_k} has type t_k for each k and type t_k has order o_k for each k, then P has type $(t_1, \ldots, t_n)^r$, where $r = 1 + \max(o_1, \ldots, o_m)$.

It should be clear from our discussion that each of these rules is sound for the intended interpretation. However, this set of rules is not complete.

We now introduce the notion of "bounding variable" of an order.

Definition: If an order n is presented in the standard form $\max(n_0, n_1 + |x_{i_1}|, \ldots, n_k + |x_{i_k}|)$, and some n_j with $(j \neq 0)$ is equal to 0, then x_{i_j} is said to be a "bounding variable" of n.

It is important to observe that the only orders deduced by any of our rules which can have bounding variables are the polymorphic orders $|x_i|$ themselves and the orders assigned to x_j in terms $x_j(A_1, \ldots, A_n)$, which have bounding variable $|x_j|$. Any other polymorphic order that we assign is the successor $1+n$ of some order n, and it is clear that no successor order can have a bounding variable.

Further, the following rule clearly holds for types assigned by our algorithm:

bounding variables: If x_i has type t in P and the order of t has bounding variable x_j, then x_j has type t in P.

The reason for this is that any rule which assigns a type with bounding variable x_j in the first instance actually assigns this type to the variable x_j. Further, this implies that we can assume that any type with a bounding variable has only one, since the types of the bounding variables can be shown to be equal by this rule, and type substitution can then be used to eliminate one of them.

We present an incomplete but often successful algorithm for computation of the type of a proposition or propositional function P in RTT. This algorithm follows the STT algorithm very closely.

Provisional algorithm: We describe the computation of the type t. The idea, as before, is to construct a set of judgments "x_i has type t_i" deducible using the type judgment rules which satisfies all the rules for a type function except for possibly containing type variables: arbitrary instantiation of the type variables then yields a true type function.

Begin the construction of the set of judgments by computing the "local" type of each occurrence of each variable x_i. The algorithm is recursive in the same way as the STT algorithm: we assume that each pf argument of pf application terms has been successfully assigned a type.

If any variable is assigned types of different arities, or if any variable x_i is assigned a type which contains $[x_i]$ as a proper component, the process terminates with the judgment that P is ill-typed (note that if x_i is assigned a type with bounding variable $|x_i|$, this does not lead to forbidden circularity).

If x_i is assigned any type t which is not a variable type (including composite types with variable components) replace all occurrences of $[x_i]$ in types assigned to other variables with the type t. Note that this does not necessarily eliminate all occurrences of x_i: if the type of x_i has bounding

variable x_i, occurrences of $|x_i|$ will remain.

The assignment of type $[x_j]$ to a variable x_i is handled as in the STT algorithm.

Such substitutions will usually occur at most once for any given variable x_i, since the target type is usually eliminated everywhere. Of course, if $[x_i]$ is introduced as a proper component of the type of x_i, terminate with a judgment of ill-typedness. The exception in which the variable x_i is assigned a type with bounding variable x_i remains to be considered. Notice that as soon as a variable is assigned a type which does not have a bounding variable, any type which that variable may have been assigned which had a bounding variable will be converted to a form which does not have a bounding variable by the global substitution process.

If x_i is assigned types $[x_j]$ and t in P, add the judgment "x_j has type t in P" and eliminate the type assignment "x_i has type $[x_j]$ in P", except in two special situations which follow: if x_i is assigned types $[x_j]$ and $[x_k]$, we assign x_i, x_j, and x_k the type $x_{\mathtt{max}\{i,j,k\}}$, as in the STT algorithm. If the type t has bounding variable x_j, it must be the case that the judgment "x_j has type t in P" has already been made. In this case we define t' as $t[[x_{\max\{i,j\}}]/x_j]$ and assign this type to both x_i and x_j, replacing all occurrences of $[x_i]$ and $[x_j]$ in all type judgments with $[x_{\max\{i,j\}}]$. Observe that in each case at least one polymorphic type has been eliminated from all type judgments.

If x_i is assigned types $(t_1, \ldots, t_n)^{m_1}$ and $(u_1, \ldots, u_n)^{m_2}$ in P, the judgments $t_i = u_i$ follow for each relevant i. From these equality judgments continue to deduce further equality judgments in the same way, ending up with a finite set of nontrivial judgments "x_k has type v_k" which can be used to unify the two composite types just as in the STT algorithm.

If x_i is assigned types $(t_1, \ldots, t_n)^{m_1}$ and $(u_1, \ldots, u_n)^{m_2}$ in P, or if x_i is assigned types $()^{m_1}$ and $()^{m_2}$, the orders m_1 and m_2 should be the same. If m_1 has bounding variable x_j and m_2 has no bounding variable, we make the additional judgment "x_j has type $(u_1, \ldots, u_n)^{m_2}$ in P" and replace all occurrences of $|x_j|$ with m_2 (other occurrences of $[x_j]$ should already have been eliminated). We proceed symmetrically if m_2 has a bounding variable and m_1 has no bounding variable. If m_1 and m_2 have bounding variables x_j and x_k respectively, we make the additional judgments "x_j has type $(u_1, \ldots, u_n)^{m_2}$ in P" and "x_k has type $(t_1, \ldots, t_n)^{m_1}$ in P", then replace all occurrences of $|x_j|$ and $|x_k|$ (there should be no frank occurences of $[x_j]$ or $[x_k]$) in type judgments with $|x_{\max\{j,k\}}|$. Both of these maneuvers are justified by the bounding variable rule.

This process must terminate. Each step of the process described eliminates at least one variable type $[x_i]$ from consideration (along with any occurrences of $|x_i|$) or terminates with a judgment of ill-typedness.

When the process terminates, we will either have concluded that P is ill-typed (and this judgment will be honest because the rules are sound for the intended interpretation) or we will have type assignments to the

variables appearing in P almost satisfying the conditions for a type function: "almost" because the same variable may be assigned distinct ramified types corresponding to the same simple type but having typographically different orders. If each variable has been assigned a unique type by the end of the process, then the algorithm succeeds in defining a type function τ up to assignments of concrete type values to type variables, as above.

This algorithm is still based on the quite standard approach of type unification implemented, for example, in the type checking of the computer language *ML* (see [4]).

The algorithm above is sound but incomplete. If it yields a type, it will always be a correct type, but there are propositions and pfs which cannot be typed by this algorithm but which are typable in RTT. In practice, the algorithm is quite good; it is not easy to write a typable term of RTT which it will not type (though we shall present an example).

A complete algorithm requires true order unification. This will depart from the usual methods of type checking, because it will require reasoning about numerical inequalities.

It might seem that we would need new judgments "$m = n$ in P", where m, n are orders, but in fact the type judgment "$()^m = ()^n$ in P" is equivalent. We do allow ourselves the abbreviation "$m = n$ in P" for "$()^m = ()^n$ in P" where it is clear that orders are being discussed (we call type equality judgments of this form "order equality judgments"), but not in the statement of the following (obviously sound) additional rules:

componentwise equality of composite types (order): If $(t_1, \ldots, t_n)^{m_1} = (u_1, \ldots, u_n)^{m_2}$ in P, then $()^{m_1} = ()^{m_2}$ in P.

order substitution: If x_i has type t in P and m is the order of t, and $()^p = ()^q$ in P, then $()^{p[m/|x_i|]} = ()^{q[m/x_i]}$ in P.

We outline our basic approach to reasoning about order unification. An order equality judgment in standard form will take the form $\max\{n_0, n_1 + |x_{i_1}|, \ldots, n_k + |x_{i_k}|\} = \max\{m_0, m_1 + |x_{j_1}|, \ldots, m_l + |x_{j_l}|\}$. This is equivalent to a disjunction of conditions, each of which asserts the equality of one of the terms of the first maximum with one of the terms of the second maximum along with the inequalities asserting that the two chosen terms are greater than or equal to the other terms of the respective maxima from which they are taken. If one or both of the orders has a bounding variable, the bounding variable is the only possible maximum chosen (which simplifies the calculation in these cases by reducing the number of cases).

All of the resulting statements can be expressed using assertions of the form $|x_i| \geq n$, $|x_i| \leq n$, or $|x_i| - |x_j| \leq n$, where n is an integer. Any equation or inequality between terms of the forms n_0 or $n_k + |x_{i_k}|$ can be converted to a conjunction of inequalities of the forms above by substracting an appropriate quantity from each side of the equality or inequality and converting an equation to the conjunction of two inequalities in the obvious way. Any conjunct

of the form $|x_i| \leq r$ where $r < 0$ (which will also be obtained (e.g.) from an equation $|x_i| + m = |x_i| + n$ where $m \neq n$) can be used to conclude that an entire conjunction is false.

We now describe the computation of complete conditions for well-typedness of a term from a number of order equality judgments. Convert each order equality judgment to a disjunction of conjunctions of inequalities of the forms described above. A conjunction of disjunctions of conjunctions is converted to a disjunction of conjunctions in the obvious way.

Now each conjunction of inequalities is processed separately. Present all inequalities in a uniform way by rewriting $|x_i| \leq n$, $|x_i| \geq n$ as $|x_i| - 0 \leq n$, $0 - |x_i| \leq -n$, respectively. Every inequality is then written in the form $A - B \leq n$. For each x_i which appears, include $0 - |x_i| \leq 0$, $0 - 0 \leq 0$ and $|x_i| - |x_i| \leq 0$ in the conjunction. Wherever $A - B \leq n_1$ and $A - B \leq n_2$ both appear, retain just $A - B \leq \min\{n_1, n_2\}$. Wherever $A - B \leq m$ and $B - C \leq n$ both appear, add $A - C \leq m + n$ to the conjunction. Apply these operations repeatedly if necessary. If any conjunct of the form $|x_i| - 0 \leq r$ with $r < 0$ or $|x_i| - |x_i| \leq r$ with $r < 0$ appears, conclude that the conjunct is false. We claim that this procedure will produce a canonical complete conjunction equivalent to the conjunction we started with.

Lemma: Any conjunction of a set of inequalities of the form $A - B \leq n$, where A and B are either 0 or variables with natural number values, is converted to a canonical equivalent form by the procedure described above.

Proof of Lemma: The proof of the Lemma is omitted from this abridged version of the paper.

Conjunctions can then be simplified by eliminating redundant conjuncts (a conjunct is redundant if eliminating the conjunct then computing the canonical form gives the same result as computing the canonical form of the original conjunction): in practice this gives quite manageable displayed forms for conditions.

Once each disjunct is computed, identical disjuncts or conjunctions weaker than other disjuncts can be recognized and eliminated (by comparing canonical forms) and a simplified form of the disjunction of conditions under which the term is well-typed can be computed (or ill-typedness can be reported if all conjuncts reduce to falsehood).

This can be applied to produce a complete algorithm: use the provisional algorithm described above to generate a list of type assignments whose failures of uniqueness are induced only by failures to unify order, then apply the procedure described above to reduce the order equality judgments that are required to arithmetic assertions about polymorphic orders. Note that under the resulting conditions it is possible to select any of the types given for each variable or propositional function as correct, since all types given for any one object will be equal under the conditions derived from the unification of the orders.

The simplification of the arithmetic conditions on polymorphic orders made possible by the use of canonical forms for conjunctions combined with the elimination of redundant conjuncts and disjuncts is essential for manageable-sized output (earlier versions showed this) and gives good results.

The reasoning above was informal arithmetical reasoning. It is theoretically interesting to observe that it can be handled by an extension of our system of type judgments. This is not how the software does it, and we do not discuss the details in this abridged version of the paper.

Here we omit a section in which comparisons between the system of this paper and the system of [3] is found, except for the comment in the following paragraph. The other points listed in the section found here in the unabridged paper are made (perhaps briefly) elsewhere in the paper.

The range of terms recognized as well-typed by our system is far larger than that recognized by the system of [3], and apparently larger than that recognized by *PM*! The system of [3] only supports types all of whose component types are "predicative". Probably the modifications of the system required to lift this restriction would not be extensive. On reading [3] originally, we thought this was a weakness of their development, but in fact it seems to reflect the intentions of the authors of *PM*: see p. 165. However, we think that more complex impredicative pfs would be needed for work in *PM* without the axiom of reducibility (and if one assumes this axiom one might as well work in *STT*).

We are working in *RTT* in all examples, but the software does not display order superscripts on types when the order is the smallest possible. Some features of the output of our software are suppressed.

```
Term input:
     S2(a1,a2)
final type list:
unconditional type:
     ()
```

Just as in example 49, clause 1, of [3], the propositional notation $S(a_1, a_2)$ is recognized as a proposition because it contains no free variables.

```
Term input:
     (R1(x1) v S1(x1))
final type list:
     x1:  0
unconditional type:
     (0)
```

This is parallel to the second example in clause 2 in example 49 of [3].

```
Term input:
     (R1(x1) v S1(x2))
final type list:
```

```
     x1:  0
     x2:  0
unconditional type:
     (0,0)
```

This term $R_1(x_1) \vee R_2(x_2)$ would be treated quite differently from the term above in the system of [3], whereas the treatment of both propositional functions in the system of this paper is very similar. In both terms, our checker first generates the list of free variables, then each free variables is typed using local rules, and the types of the free variables are listed to form the type of the pf.

The system of [3] uses a different (and more usual) kind of context than our system. The form of a type judgment of the system of [3] is $\Gamma \models f : t$, where f is a term, t is the type assigned to that term, and Γ, the "context", is a list of assignments of types to variables. In our system, a type judgment about an entire term (propositional notation) has no context, while type judgments about variables have as context the term in which they appear.

In the system of [3], the term $R_1(x_1) \vee R_2(x_1)$ is typed by first considering the typing of $R_1(a_1) \vee R_2(a_1)$, which is immediately seen to have type (), and in which the term a_1 has type 0, then using the rule for typing substitutions to insert new component with type 0 into the type () of $R_1(a_1) \vee R_2(a_1)$ to obtain the type (0). The term $R_1(x_1) \vee R_2(x_2)$ is typed by observing that the two disjuncts have the property that all variables of the first are alphabetically prior to the variables of the second, typing the first and the second as (0) in the same way we typed the previous term, then concluding that the type of the whole is the "product" $(0, 0)$ of two copies of (0) (speaking somewhat loosely). This might give some idea of the very different flavor of the two approaches.

```
Term input:
     [x1](x1!() v ~x1!())
final type list:
     x1:  ()
unconditional type:
     ()^1
```

This is example 51 from [3]. Order is important in this example. Note that the variable x_1 represents a proposition (a 0-ary propositional function); the order of its type is 0. The entire term is also a proposition (it contains no free variables, because x_1 is bound by the quantifier) but its order is at least 1, because it must be greater than the order of the quantified variable. It is precisely 1 because we used "predicative" pf application. The order 0 of the type of x_1 is not displayed because it is as small as possible.

We can see an explicit polymorphic type by implementing the term in Remark 58 of [3], stipulating that the application is predicative.

```
Term input:
     x2!(x1)
```

```
final type list:
     x1:  [x1]
     x2:  ([x1])
unconditional type:
     ([x1],([x1]))
```

In this term, x_1 is of a completely unknown type $[x_1]$, while x_2 is seen to be of type $([x_1])$ (it is a predicate of objects of type $[x_1]$), so the whole term is of type $([x_1],([x_1]))$: the order of the components is determined by the fact that x_1 is alphabetically prior to x_2.

In [3], two different derivations are given, showing how two different types can be assigned to this pf, whereas here we get a single computation yielding *all* types. If we get more information from the context, the type will become more specific:

```
Term input:
     (x2!(x1) v S1(x1))
final type list:
     x1:  0
     x2:  (0)
unconditional type:
     (0,(0))
```

Here we know from local information elsewhere in the term that the type of x_1 is 0, so we get a more specific type for the whole pf.

We now give a large example. There are two different conditions under which the given pf is well-typed.

```
Term input:
     (x1!(x2,x2) v x1!([x3][x5]x3!(x5,x8),[x6][x9]x6!(x4,x9)))
unconditional type:
     ?!?
conditional type:
     ((([x8])^max(|x5|+2,
     |x8|+2,2),([x8])^max(|x8|+2,|x9|+2,2)),
     ([x8])^max(|x5|+2,|x8|+2,2))
  WITH
     |x5| <= |x9| and
     |x8| <= |x9| and
     |x9| <= |x5|
  OR
     |x5| <= |x8| and
     |x9| <= |x8|
```

In more standard notation, the propositional function is

$$x_1!(x_2,x_2) \vee x_1!((\forall x_3.(\forall x_5.x_3(x_5,x_8))),(\forall x_6.(\forall x_9.(x_6!(x_4,x_9)))))$$

The entire term is a propositional function of the arguments x_1 and x_2; it is necessary to figure out what the types of x_1 and x_2 are. Because of the presence of the subterm $x_1!(x_2, x_2)$, we know that the two arguments of any occurrence of x_1 must be of the same type. So the propositional functions $(\forall x_3.(\forall x_5.x_3(x_5, x_8)))$ and $(\forall x_6.(\forall x_9.(x_6!(x_4, x_9)))$ are of the same type. Each of these is a function of one variable, x_8 in one case and x_4 in the other, so x_4 and x_8 are of the same type. This base type is polymorphic: we know nothing about it.

Now we need to analyze orders. The order of the type of $(\forall x_3.(\forall x_5.x_3(x_5, x_8)))$ is two greater than the maximum of the orders of $[x_5]$ and $[x_8]$. The increment of two is because x_3 has type one greater than this maximum, and the order is raised one more because of the quantifier over the type of x_3. Similarly, the order of the type of $(\forall x_6.(\forall x_9.(x_6!(x_4, x_9)))$ is two greater than the maximum of the order of $[x_4] = [x_8]$ and the order of $[x_9]$. These two orders have to be the same. There are two ways for this to happen: either the order of $[x_5]$ is greater than the order of $[x_8]$, in which case the order of $[x_9]$ also has to be greater than the order of $[x_8]$ and actually must be the same as the order of $[x_5]$, or the order of $[x_8]$ is greater than or equal to the orders of $[x_5]$ and $[x_9]$ (which in this case need not be the same). And these two cases are what the output above describes.

The type of x_1 will be $([x_2], [x_2])$; the type of x_2 will be (x_8). So the underlying simple type of this expression is $((([x_8]), ([x_8])), ([x_8]))$, and this is what we see above, adorned with appropriate orders.

We omit a section on applications to proof-checking for *PM* which will appear in the unabridged paper.

References

[1] Holmes, M. Randall, "Subsystems of Quine's "New Foundations" with Predicativity Restrictions", *Notre Dame Journal of Formal Logic*, vol. 40, no. 2 (spring 1999), pp. 183-196.

[2] 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`.

[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] Milner, R., "A theory of type polymorphism in programming", *J. Comp. Sys. Sci.*, 17 (1978), pp. 348-375.

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