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

M. Randall Holmes

September 2, 2011

Abstract

A formal presentation of the ramified theory of types of the *Principia Mathematica* of Russell and Whitehead is given (along with the simplified theory of types of Ramsey). The treatment is inspired by but differs sharply from that in a recent paper of Kamareddine, Nederpelt and Laan. Algorithms for determining whether propositional functions are well-typed are described, including a complete algorithm for the ramified theory of types, which is unusual in requiring reasoning about numerical inequalities in the course of deduction of type judgments. Software implementing these algorithms has been developed by the author, and examples of the use of the software are presented. The approach is compared with that of Kamareddine, Nederpelt and Laan, and some brief observations are made about use of the type checker in a proof checker for the ramified theory of types under development.

1 Introduction

This paper was inspired by careful reading of the paper [4], where Kamareddine, Nederpelt and Laan present a formalization of the ramified theory of types (hereinafter RTT) of [7], the *Principia Mathematica* of Russell and Whitehead (hereinafter PM). It is surprising to discover on close reading of PM that its theory of types (the oldest one) is nowhere given a complete formal description which is up to modern standards of rigor. There are various formal systems of ramified type theory in the literature (the author has even presented one, based on earlier work of Marcel Crabbé, in [2]), but the one in [4] is clearly motivated by a desire to closely implement the notation of PM, although the approach to formalization of reasoning about types they take is much more modern.

During our reading of [4] we developed a type checker ([3]) for the formalized version of RTT presented in that paper. The approach we took to the type system in the course of the development of this checker was quite different from the approach taken in [4], and allows type-checking for a wider range of terms of the language of RTT than does the system of [4]. From the implementation of type checking we developed at that time, it is possible to "reverse engineer" a formal treatment of the type system of RTT, which we give here.

We appeciate useful conversations with Professor Kamareddine (and access to the LaTeX source of [4] to facilitate typesetting of the examples taken or adapted from that paper), and also the helpful remarks of anonymous referees.

2 Informal Presentation of the System of *Principia Mathematica*

We give an informal presentation of the notions of proposition and propositional function as actually given in *PM*, in order to motivate the formalization of [4]. We feel that such a presentation is necessary because superficial examination reveals that the system of [4] is not identical to the system presented in *PM*. This section is intended to provide support for the claim that the system of [4] (with certain modifications which we will indicate) is in fact an accurate formalization of the intentions of *PM*.

At the outset, *PM* takes some selection of the propositional connectives as primitive. We follow the original text and take negation and disjunction as primitive; the last edition of *PM* suggests the use of the Sheffer stroke. It should be noted that *PM* uses propositional variables, a feature not found in [4], and we include propositional variables in our formal language developed below. Propositional variables are not important for the investigation of type theory of propositional functions (in fact, no propositional variable is allowed to appear in a propositional function in our implementation) but they turn out to be indispensible in practical formalization of reasoning about propositions.

The "atomic propositions" of *PM* are of the form $R_n(a_{i_1}, \ldots, a_{i_n})$, in which R_n is an n-ary predicate of individuals and the a_{i_j}'s are names of individuals. The type of individuals is the sole base type of the system of *PM*. The system of [4] allows the case $n = 0$, which would give us constant propositions $R_0()$; *PM* does not allow this. Our software allows one to choose to allow or exclude 0-ary predicates.

The "elementary propositions" of *PM* are formed by combining atomic propositions with logical connectives.

Variables (taking individual values at this point) are now introduced. Variables (when representing individuals) can appear in the same contexts as individual constants. An elementary proposition containing variables is an ambiguous proposition (its meaning is not determined until values are assigned to the variables).

The next step is to introduce *propositional functions*. A propositional function is obtained by replacing each variable x in an ambiguous elementary proposition with $\hat{x}$. The resulting expression denotes a function of as many variables as appear in it. The order in which arguments are supplied to the function is determined by the alphabetical order of the variables appearing in it (in our notation, this is determined by the order of the numerical indices of the variables). For example, in an arithmetic context $\hat{x} < \hat{y}$ and $\hat{b} > \hat{a}$ would be the same propositional function (or at least would have the same extension).

PM defines quantifiers in terms of propositional functions. The sentence $(x)(\phi x)$ $((\forall x.\phi(x))$ in our notation) is obtained by applying an operation of "generalization" to the propositional function $\phi\hat{x}$. The official line in *PM* is that propositions in which quantified sentences appear as arguments of propositional connectives do not really occur: a system of contextual definitions "defines away" sentences which apparently have this feature as sentences in prenex normal form. It would be extraordinarily inconvenient to actually take this view in a computer implementation, and fortunately *PM* presents an alternative formulation of logical rules for quantified sentences which allows the propositional functions to take quantified sentences as arguments in the usual way. The one unfamiliar feature is that since a propositional function must actually contain its variable argument, the scope of a quantifier must include a free occurrence of the quantified variable for the sentence to be well-formed, and our software does enforce this. Our formalization does not otherwise acknowledge the dependence of quantifiers on propositional functions.

Since we take this view, we associate propositional functions $\phi\hat{x}$ with quantified sentences ϕx of arbitrary complexity with free occurrences of the variable x.

We now discuss higher-order variables and propositional functions. The notation of *PM* for arbitrary ambiguous propositions, considered as propositional functions, is $\phi\hat{x}$, $\phi(\hat{x}, \hat{y})$, etc. Parentheses are not used to enclose argument lists of length one, and argument lists of length 0 (yielding variable propositions $\phi()$) do not occur, though they do occur in the system of [4]; permission to use such expressions can be turned on or off in our software. Note that variables ϕ have been introduced representing propositional functions. An eccentricity of the *PM* notation is that when $\phi\hat{x}$ occurs as an argument to a propositional function, it is written $\phi\hat{x}$, not ϕ. Quantifiers over functions are written $(\phi), (\exists\phi)$, though there is an assertion in *PM* that this is an abbreviation for $(\phi\hat{x}), (\exists\phi\hat{x})$. This penchant for complex "variables" for propositional functions seems to be motivated by a desire to clearly indicate the status (for *PM*) of propositional functions as "incomplete symbols".

It seems to us that the implementation of this in more complicated cases in *PM* is incorrect. For example, *PM* tells us (p. 52) that $F(\phi\hat{x})$ is an ambiguous expression for a function with a single argument which is itself a propositional function of a single individual variable. We are then told that a variable representing a function of this kind would be written $F(\hat{\phi}\hat{x})$ (with the circumflex over the ϕ). But this seems wrong. The symbol $\hat{\phi}\hat{x}$ should be a constant, the name for the propositional function A such that $A(\phi\hat{x}, a) = \phi a$ (this function is often mentioned as an example in *PM*, but notation for it is never given). So $F(\hat{\phi}\hat{x})$ should represent the application of an ambiguous third-order function to this constant second-order function. A bound variable standing for an arbitrary first order function should properly be written $\widehat{\phi\hat{x}}$ (with the circumflex over the entire complex variable), and a variable second-order function should be written $F(\widehat{\phi\hat{x}})$. It is not our purpose here to reform the notation of *PM*, as we actually prefer the notation of [4], but this problem ought to be noted.

Constant propositional functions do not appear in applied position either in *PM* or in [4]. The reason for this is that a constant propositional function is an expression with holes in it, and to apply the function is to substitute the arguments for the holes in the original expression. Our computer implementation does support syntax for constant function application without substitution, but we will not use it here.

Because of the very limited use of notation for propositional functions in *PM*, we do not see examples of constant propositional functions appearing as arguments to propositional functions in *PM*, but it seems reasonable that if one were to take the function $F(\hat{x} = \hat{y}, a, b)$, and instantiate F with $\hat{\phi}(\hat{z}, \hat{w})$, that one would obtain $a = b$. At any rate, this extension of notation (allowing constant propositional functions to appear as arguments) is found in [4].

Simple variables do not always represent individuals. *PM* takes advantage of "systematic ambiguity" (what we would call "polymorphism"); the type of variables whose type cannot be determined by examination of an expression may be arbitrarily complex. But any variable which appears in applied position somewhere in a proposition or propositional function will appear with formal arguments whenever it appears as an argument to a variable function itself.

We now discuss the types and orders of *PM*. *PM* does not anywhere give a formalized discussion of its type system; in fact, there is no notation for types in *PM*! But the informal discussion is clear enough that the intentions of the authors can be determined.

Type is determined as follows. The simplest type is that of individuals. The type of a propositional function (abstracting out the order of the type, which we will address in the next paragraph) is determined by the types of its arguments.

Every type has an order. The order of the type of individuals is 0. The order of a propositional function is one plus the maximum of the orders of the types of its arguments and the orders of the types of quantified variables. It is the effect of quantification on order that makes order a nontrivial concept. The motivation of this concept is that a quantified sentence is viewed as being in effect an infinite disjunction or conjunction over the type of the quantified variable: thus it is important to prevent the possibility of a propositional function containing a quantifier over its own type (or a more complex type), as this would lead to a formal circularity.

Ramsey simplified the type system of *PM* to eliminate the orders: this "simple theory of types" (contrasted with the "ramified theory of types" of *PM*) is discussed in [4] and in this paper as well.

Thus for any list of types of arguments to be supplied to a function, an infinite sequence of function types of progressively higher order is obtained. *PM* gives a special status to "predicative" functions, whose order is the least possible given the orders of the types of the arguments of the function, and whose arguments are all in their turn of predicative types. A special notation $\phi!x$ is used for the application of functions of predicative types. This notation is not used in [4], but we introduce it here, with a generalization. For us, $\phi!(x_1, \ldots, x_n)$ refers to a function of the arguments x_i whose order is the least possible given the orders of the types of the x_i's, but we do not require that the types of the x_i's be predicative themselves for this notation to be used.

We can now briefly describe the notation of [4] (our extension of this notation is formally described in the next section). In the notation of [4], all variables are simply letters (possibly with numerical suffixes), and there are no circumflexed variables. All occurrences of variables within propositional functions are to be understood as circumflexed (bound as arguments of the propositional function). The only ambiguity this introduces is that a top-level expression for a proposition looks the same as the expression for the corresponding propositional function. This ambiguity exists only at the top level, because propositions do not occur as arguments to propositional functions. It appears that a formalized version of the language of *PM* along the lines suggested above (with the correction to scopes of circumflexes) would be readily intertranslatable with the language based on that of [4] which we describe formally in the next section, mod occasional renamings of bound variables due to the fact that a bound individual variable and a bound function variable in different contexts might take the same shape in this language and would have to renamed before translation into the original *PM* notation.

3 Propositions as Mere Syntax

The logical world of *PM* is inhabited by *individuals* and *propositional functions*. We usually abbreviate the phrase "propositional function" as "pf", following [4]. In this section, we formally describe the notation for propositions and pfs.

Notation for individuals is simplicity itself: an individual is denoted by one of the symbols $a_1, a_2, a_3, \ldots$ (in the computer implementation, `a1, a2, a3...`).

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). (We call these "general" variables on the few occasions when we need to distinguish them from "propositional variables" introduced below.) 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 general variable.

Now we present the definition of notation for propositions. The notion of free occurrence of a (general) variable in a proposition is defined at the same time.

In the system of [4], any notation for a proposition is also notation for a propositional function. It is necessary here to exclude propositional notations which contain propositional variables (which do not occur in [4]). In *PM* (e.g., on p. 38) it states clearly that a proposition must contain a free variable to be read as a propositional function, which motivates the implementation in our software of an option to exclude 0-ary relation symbols and pfs. If 0-ary pfs are excluded, a propositional notation will be a pf notation iff it contains no propositional variables and at least one free general variable. If 0-ary pfs are permitted, the criterion is simply that the notation contain no propositional variable.

propositional variable: A variable taken from $p_1, p_2, p_3 \ldots$ (`p1, p2, p3...` in the computer implementation) is a proposition. This is a propositional variable. (There are no propositional variables in the system of [4], but there are in *PM*). No (general) variables occur, free or otherwise, in a propositional variable.

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 [4], and for us if we admit 0-ary pfs). 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. Disjunction is the only primitive binary propositional connective in *PM*, but we will allow use of other connectives: $(P \rightarrow Q)$, $(P \wedge Q)$, $(P \equiv Q)$ with the usual meanings. 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; the rule is the same if a different binary propositional connective is used.

quantifiers: If P is a proposition in which the variable x_i occurs free (this stipulation is what requires us to define freedom of variables at the same time as syntax of propositions), $(\forall x_i.P)$ 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 computer allows any string of upper-case letters to be used as a quantifier, and other quantifiers could be introduced. The free occurrences of variables in $(\forall x_i.P)$ are the free occurrences of variables other than x_i in P; the rule would be the same for any other quantifier.

In [4], 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 (though numerical indices representing orders do appear occasionally).

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, suitable to represent 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 the paper [4]; its use in this paper is a generalization of the use of a similar notation for "matrices" (predicative functions) in *PM*. $x_i()$ is also a proposition in the system of [4] (the variable x_i represents a proposition in this case); $x_i()$ and $x_i!()$ are propositions for us as well if we admit 0-ary pfs. 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 carefully that the free occurrences of variables in those A_i's which are propositional 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.

There are no binders in notation for a propositional function, which will give our treatment a somewhat unfamiliar flavor. Since we do not have head binders 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.

We refer to the atomic propositions and the pf application terms as "logically atomic" (propositional variables are also logically atomic, but they do not occur in pf notations), and to other terms as "logically composite".

4 The Definition of Substitution and Its Failure

We now give the recursive definition of simultaneous substitution of a list of individuals, variables and/or pfs 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.

propositional variable: $p_j[A_k/x_{i_k}] = p_j$.

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 typographically the same as x_{i_k}; define v'_i as v_i if it is not typographically the same as any x_{i_k}. If any v'_i is a propositional function, $R_n(v_1, \ldots, v_n)[A_k/x_{i_k}]$ 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 $(\forall x_j.P[A'_k/x_{i_k}])$.

propositional function variable application: Let $x_j(V_1, \ldots, V_n)$ or $x_j!(V_1, \ldots, V_n)$ be a proposition built by pf application. 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 $Q[V'_k/x_{t_k}]$.

There is a serious difficulty with this "definition". Consider the pf $\neg x_1(x_1)$. 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 an infinite regress. This shows that the proposed "definition" of substitution is essentially circular – 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; we 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 [4]. 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 [4] cannot is that their definition of the typing algorithm depends on the notion of substitution, and ours does not. (The definition of our type algorithm does rely on the notion of substitution into notations for types, but the definition of substitution into type notations does not present such logical complications).

5 The Simple Theory of Types

We follow [4] 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 0-ary pfs are excluded, no complex type will have () as a component; this will be enforced by requiring $t_i \neq ()$ 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 [4], 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 does for us imply 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 have 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 assigned to any notation other than a variable or individual constant (or assess its typability), we appeal only to the types assigned to proper subterms of that notation, and we are given types of variables and individual constants at the outset.

A weakening of this algorithm is possible if we take into account the possibility of renaming bound variables. This is implemented in our software for the simple theory of types, both for quantified variables and for most variables appearing in pf arguments, but not in the ramified type theory implementation. Renaming of bound variables can be forced by a command in the software prior to application of the type algorithm, however.

We now proceed to develop a system for expressing and reasoning about type assignments to subterms of pfs, 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 or pf notation, 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 general variable.

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 in terms of 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 (note that this is equivalent to "if there is a type function on $(P \vee Q)$ there is a type function on P and a type function on Q"). If x_i has type t in P or x_i has type t in Q, then x_i has type t in $(P \vee Q)$. (Note that if $\tau(x_i) = t$ must be true for any type function τ on some subterm of P, it must be true for any type function τ on P.)

There are three kinds of occurrences of variables in logically atomic subterms; the ways in which these occurrences are typed are summarized by the following rules:

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 expresses the fact that the appearance of a variable as an argument of a pf application term does not constrain its type at all).

In this way a possibly variable type may be assigned to each occurrence of a variable. 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 correct 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 from our discussion 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 an arbitrary type 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 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. We prove the theorem by structural induction: we assume that each pf argument of a pf application subterm of P can be assigned a type satisfying the conditions of the theorem (this is needed to compute the "local" types of head variables of pf application terms).

The only way in which this can fail to induce a type function on P (mod instantiation of type variables with concrete types) is if more than one type is assigned to the same variable. We show how to resolve such situations.

If any variable is assigned types of different arities, the process terminates with the judgment that P is ill-typed. 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 (including composite types 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. 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 [5]).

The algorithm actually implemented in our software for simple type theory differs from the theoretical algorithm in not using the rule deducing types of variables from types of their occurrences in propositional function arguments. This means that the type of a variable x_i in a propositional function argument will only interact with the types of variables in the larger term if the type $[x_i]$ appears as a component type of the type of the argument. This is legitimate, because we could arrange for all variables of the propositional function argument (being bound) to be renamed to avoid collisions with types of variables appearing elsewhere. However, variables whose polymorphic type appears in the type assigned to the argument are not considered as being renamed.

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, in the version which implicitly allows renaming of bound variables appearing in pf arguments but not in their polymorphic types.

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 pf application. 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 constant propositional function 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.

6 The Ramified Theory

The motivation behind the ramified theory is as follows. The type of a propositional function in *STT* is determined by the types of its arguments, and all types of its arguments must be simpler than its type: understanding the meaning of the pf involves understanding the entire range of the types of its arguments, so it cannot without circularity be an item in one of those types. But it can further be said 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 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 (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 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.

We begin the formal treatment with the definition of formal polymorphic orders.

natural number: A natural number n is a polymorphic order.

polymorphic variable: For each variable x_i, the symbol $|x_i|$ is a polymorphic order.

addition: The formal sum of a polymorphic order and a natural number is a polymorphic order.

maximum: The formal maximum of two polymorphic orders is a polymorphic order.

simplification: Addition is understood to be commutative and associative. Each sum appearing in a polymorphic order is of the form $|x_i| + m$: two polymorphic variables are never added, so there is no need for more complex sums.

Maximum is understood to be commutative and associative. The identity $\max(a, b) + c = \max(a + c, b + c)$ can be used to convert any polymorphic order to a maximum of sums. No more than one natural number not added to a polymorphic order needs to appear in such a maximum of sums (because $\max(m, n)$ can be simplified to either m or n). No more than one sum involving the same $|x_i|$ needs to appear, since $\max(|x_i|+m, |x_i|+n) = |x_i| + \max(m, n)$. So there is a unique canonical form for polymorphic orders, 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 readily computable operations.

order of polymorphic orders: 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.

substitution into orders: 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.

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 $|x_i|$.

There are two possible ways of understanding the relationships between the orders. Explicit assertions in *PM* support the idea that any two types must be disjoint, and so two types $(t_1, \ldots, t_n)^r$ and $(t_1, \ldots, t_n)^s$ with $r \neq s$ must be disjoint. This is the view we take here. There is a possible alternative approach, taken up by other workers (see [6]), that $(t_1, \ldots, t_n)^r \subseteq (t_1, \ldots, t_n)^s$ holds when $r < s$. We do not take this view, but we found consideration of this alternative view very useful in constructing early versions of the type inference algorithm for *RTT*.

We present the rules for a term-typing function τ as above. Notice that here the orders will be fixed non-negative integers: polymorphic orders appear in our algorithm because the structure of terms gives insufficient information to fix orders precisely in some cases.

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

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, outside proper propositional function arguments). 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 quantified over 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 order m strictly larger than the order of each $\tau(A_k)$.

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 variable propositional function is polymorphic, 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.

As above, we will regard a pf as well-typed when there is a type function τ which assigns a type to that pf. 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. As above, a more liberal type algorithm could be obtained by requiring that bound variables be renamed to be distinct from one another and from free variables when this preserves meaning, but this is not implemented in our software. There is a tool which will rename all bound variables in such a way that they are typographically distinct whenever possible; this can be applied before typing to get the most general typing conditions for a pf.

We now describe the rules of type inference for RTT. We include only those clauses which differ from the corresponding clauses in the STT algorithm.

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 $s = \max(|x_j|, o_1 + 1, \ldots, o_n + 1)$. (In RTT, we distinguish the two kinds of pf application term).

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. (We reproduce this definition because of the mention of order, though order does not affect arity).

componentwise equality (identification of components): 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.

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. (Recall that the computation of $t[t/[x_i]]$ includes the reduction of its order to standard form; this resolves the apparent circularity of the case in our algorithm where we assign a variable x_i a type t whose order is a maximum of orders including $|x_i|$; in $t[t/[x_i]]$, the order of t is apparently modified by the replacement of $|x_i|$ with the entire order of t, but on simplification the order of t is restored to its original form, so in fact $t[t/[x_i]] = t$ in this case and no judgment of ill-typedness results)

As above, we need the rule for typing propositional functions. This rule needs to take into account the effect of quantified variables on order.

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 $r = 1 + \max(o_1, \ldots, o_m)$.

We need the following rule and we do not subsequently relax it as in simple type theory.

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 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 bounding variable.

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 in the STT algorithm, 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 types may have variable components: 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.

As in the STT algorithm, what remains is to unify distinct types assigned to the same variables (or show that they cannot be unified).

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: the only occurrence of $[x_i]$ in the type assigned to x_i is the occurrence of $|x_i|$ in its order. Substitution of the type t of x_i for $[x_i]$ in t has the effect of replacing $|x_i|$ with the order of t in the order of t, and after simplification the order is left the same. Order variables can lead to fatal circularity, though: if x_i is assigned a type t with an order which is a maximum of orders one of which is $|x_i| + r$, with $r \neq 0$, then $t[t/[x_i]] \neq t$ and we can conclude that P is ill-typed.

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. If x_i is assigned type $[x_j]$ $(j \neq i)$, proceed as in the STT algorithm.

Notice that 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 any 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.

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. Note that all occurrences of $[x_j]$ will then be eliminated if t is not a type variable and does not have order with bounding variable x_j. In these special cases where $[x_j]$ would not be eliminated 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 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 $[x_{\max\{i,j\}}]$.

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. 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. Assign to x_i the types which result if all these types x_k are replaced with the corresponding v_k's in each of the two types being reconciled (the resulting types will not necessarily be the same, because the orders may be different). 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.

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. In this algorithm, we only use this information if one or both of the orders m_1 or m_2 has a bounding variable. 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 (any occurrences of $[x_j]$ as a type 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 all occurrences of its order $|x_i|$) 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 obtained a set of type assignments to the variables appearing in P almost satisfying the conditions for a type function: the difficulty is that 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 [5]).

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 can be read as well-typed terms of 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 some examples).

A complete algorithm requires unification of orders. 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 a new kind of type judgment to express equations between polymorphic orders, but in fact "order equality judgments" of the form "$m = n$ in P", where m and n are polymorphic orders, are equivalent to type equality judgments "$()^m = ()^n$ in P". We will allow ourselves to abbreviate type equality judgments as order equality judgments when this can cause no confusion.

Obviously sound additional rules are

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 holds, then $()^{p[m/|x_i|]} = ()^{q[m/x_i]}$ in P holds.

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 assertion 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: We will refer to items such as A and B above as "literals" for the moment. In our application, literals are 0 and polymorphic orders $|x_i|$ of variable types.

We claim first that inconsistency of the conjunction of a set of inequalities is always detected by this procedure. Suppose we have a partial assignment of values to literals (with 0 assigned the value 0) and we wish to consider possible values of a literal A to which a value has not been assigned. The conditions of forms $A - B \leq n$, $C - A \leq m$ for B and C to which values have been assigned determine intervals in which the value A can lie. Now intervals have the logically interesting property that any set of intervals which intersect pairwise actually have nonempty intersection. If it is not possible to assign a value to A consistent with given inequalities involving A and assignments of value, then there must be a pair of intervals $A - B \leq n$, $C - A \leq m$ for B and C to which values have been assigned which do not intersect (as intervals of the same kind obviously always intersect). The values assigned to B and C then cannot satisfy $C - B \leq m + n$, which is one of the equations added to the set by our procedure, as well as being a logical consequence of the original conjunction, so the values assigned to B and C were already inconsistent with the conjunction of inequalities. This means that if a conjunction of literals is actually satisfiable, then we can proceed by completing the conjunction as above, and using the completed conjunction and the values assigned previously to other literals to determine the possible values for each literal; this will work regardless of the order in which the literals are considered.

We claim further that two equivalent conjunctions will be expanded to the same form by this procedure. This is easy: suppose one conjunction, when expanded, contains $B - 0 \leq n_0$ and the other contains $B - 0 \leq n_1$ ($n_0 \neq n_1$). It follows that the range of values which can be assigned to B at the very first step of the process of assignments of values to literals is different, so the original conjunctions cannot have been equivalent. Now suppose that one conjunction, when expanded, contains $B - A \leq n_0$ and the other contains $B - A \leq n_1$ ($n_0 \neq n_1$). Now assign a value to A (compatible with its bound relative to 0). The range of values possible to assign to B (the bound on whose value relative to 0 being the same in both expanded conjunctions) will be different for the two expanded forms, which shows that the two expanded conjunctions cannot be equivalent, so the original conjunctions were not equivalent.

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).

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 if the conditions are consistent, since all types given for any one object will be equal under the conditions derived from the unification of the orders.

A notable point about the algorithm is that 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 gives quite manageable output (earlier versions which computed and displayed things more lazily gave unmanageably large displays which were not useful in practice).

The reasoning above was informal arithmetical reasoning. It is useful to observe that it can be coded into the language of order equality type judgments. We do not do this in the software: the type inference algorithm just implements the provisional algorithm described above while the inequalities are handled by a dedicated representation of quite conventional reasoning about arithmetic inequalities. So we feel no need to do more than sketch the way in which this reasoning could be incorporated directly into the system of reasoning about types. We use the language of order equality judgments, but recall that these abbreviate special type equality judgments.

order inequality: A judgment "$m \leq n$ in P" is equivalent to "$n = \max\{m, n\}$ in P", and so requires no expansion of our language of type judgments.

type subtraction: The judgments we have found it convenient to write as "$A - B \leq n$ in P" can be expressed formally as "$A \leq B + n$ in P".

relations to zero: The judgments $0 - m \leq 0$ and $m - m \leq 0$ assumed for all orders in the algorithm above expand to judgments automatically made by the algorithm for simplifying polymorphic orders.

$$0 - m \leq 0 \equiv 0 \leq 0 + m \equiv 0 \leq m \equiv m = \max\{0, m\}$$

$$m - m \leq 0 \equiv m \leq 0 + m \equiv m = \max\{m, m\}$$

expansion of equations between maxima: "$\max\{m, n\} = p$" implies "($n \leq m$ and $n = p$) or ($m \leq n$ and $m = p$)". Of course, this needs to be applied on both sides of the equals sign. It also requires us to expand our language to allow the handling of cases: the distributivity of conjunction over disjunction will also be needed if this is to be completely formalized. Note that the special treatment of orders with bounding variables can be justified using the type judgment rule for bounding variables given above combined with order unification.

"triangle inequality" steps: The deduction from $A - B \leq m$ and $B - C \leq n$ to $A - C \leq m + n$ is justified as follows: we actually read $A - B \leq m$ as $A \leq B+m$: from $A \leq B+m$ and $B \leq C+n$ deduce $A+B \leq B+C+m+n$, and from this deduce $A \leq C+m+n$ using the rules "deduce $m+p \leq n+q$ from $m \leq n$ and $p \leq q$" and "deduce $m \leq n$ from $m + p \leq n + p$". These rules doubtless can be "simplified" to corresponding rules about equations, but the basic shape of the additional inference rules needed to justify triangle inequality steps is clear.

absurdity: Judgments of the form $m \leq -r$ where $r > 0$ or $m - m \leq -r$ where $r > 0$ signal absurdity: this is implemented by rules asserting that from $0 = m+r$ or $m = m+r$ (where $r > 0$) in P we deduce that P is ill-typed.

7 Relations to Other Work

In this section we discuss the relationship of the development in this paper to the development in [4]. We are not familiar with the details of any other attempt to faithfully implement the theory of types of *PM* in modern terms: we are familiar with some other treatments of the ramified theory of types, but they seem to be more remote from the actual usage of *PM*.

The system of [4] uses a different (and more usual) kind of context than our system. The form of a type judgment of the system of [4] is $\Gamma \models f : t$, where f is a term, t is the type assigned to that term, and Γ is a finite function from variables to types representing types assigned to variables in the context. 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. To make comparison easier, we reproduce in its entirety (though certainly without full explanation) the recursive definition of type judgments from [4]. We will refer back to this in the following section of examples.

Definition 40 (Ramified theory of types (RTT)), from [4]: The *judgements* $\Gamma \vdash f : t^a$ are inductively defined as follows:

1. **(start)** For all a we have: $\vdash a : \mathtt{0}^0$.

   For all atomic pfs f we have: $\vdash f : ()^0$;

2. **(connectives)** Assume $\Gamma \vdash f{:}(t_1^{a_1}, \ldots, t_n^{a_n})^a$, $\Delta \vdash g{:}(u_1^{b_1}, \ldots, u_m^{b_m})^b$, and $x < y$ for all $x \in \mathrm{dom}(\Gamma)$ and $y \in \mathrm{dom}(\Delta)$. Then

$$\Gamma \cup \Delta \vdash f \vee g : \left(t_1^{a_1}, \ldots, t_n^{a_n}, u_1^{b_1}, \ldots, u_m^{b_m}\right)^{\max(a,b)};$$

   and

$$\Gamma \vdash \neg f : (t_1^{a_1}, \ldots, t_n^{a_n})^a;$$

3. **(abstraction from parameters)** If $\Gamma \vdash f : (t_1^{a_1}, \ldots, t_m^{a_m})^a$, $t_{m+1}^{a_{m+1}}$ is a *predicative* type, $g \in \mathcal{A} \cup \mathcal{P}$ is a parameter of f, $\Gamma \vdash g : t_{m+1}^{a_{m+1}}$, and $x < y$ for all $x \in \mathrm{dom}(\Gamma)$, then

$$\Gamma' \vdash h : (t_1^{a_1}, \ldots, t_{m+1}^{a_{m+1}})^{\max(a, a_{m+1}+1)}.$$

Here, h is a pf obtained by replacing all parameters g' of f which are α_Γ-equal to g by y. Moreover, Γ' is the subset of the context $\Gamma \cup \{y : t_{m+1}^{a_{m+1}}\}$ such that $\mathrm{dom}(\Gamma')$ contains all and only those variables occurring in h;

4. **(abstraction from pfs)** If $(t_1^{a_1}, \ldots, t_m^{a_m})^a$ is a *predicative* type, $\Gamma \vdash f : (t_1^{a_1}, \ldots, t_m^{a_m})^a$, $x < z$ for all $x \in \mathrm{dom}(\Gamma)$, and $y_1 < \cdots < y_n$ are the free variables of f, then

$$\Gamma' \vdash z(y_1, \ldots, y_n) : (t_1^{a_1}, \ldots, t_m^{a_m}, (t_1^{a_1}, \ldots, t_m^{a_m})^a)^{a+1},$$

where Γ' is the subset of $\Gamma \cup \{z{:}(t_1^{a_1}, \ldots, t_m^{a_m})^a\}$ such that $\mathrm{dom}(\Gamma') = \{y_1, \ldots, y_n, z\}$;

5. **(weakening)** If Γ, Δ are contexts, $\Gamma \subseteq \Delta$, and $\Gamma \vdash f : t^a$, then also $\Delta \vdash f : t^a$;

6. **(substitution)** If y is the ith free variable in f (according to the order on variables), and $\Gamma \cup \{y : t_i^{a_i}\} \vdash f : (t_1^{a_1}, \ldots, t_n^{a_n})^a$, and $\Gamma \vdash k : t_i^{a_i}$ then

$$\Gamma' \vdash f[y{:=}k] : (t_1^{a_1}, \ldots, t_{i-1}^{a_{i-1}}, t_{i+1}^{a_{i+1}}, \ldots, t_n^{a_n})^b.$$

Here, $b = 1 + \max(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, c)$,
and $c = \max\{j \mid \forall x{:}t^j \text{ occurs in } f[y{:=}k]\}$

(if $n = 1$ and $\{j \mid \forall x{:}t^j \text{ occurs in } f[y{:=}k]\} = \varnothing$ then take $b = 0$) and once more, Γ' is the subset of $\Gamma \cup \{y : t_i^{a_i}\}$ such that $\mathrm{dom}(\Gamma')$ contains all and only those variables occurring in $f[y{:=}k]$;

7. **(permutation)** If y is the ith free variable in f (according to the order on variables), and $\Gamma \cup \{y{:}t_i^{a_i}\} \vdash f : (t_1^{a_1}, \ldots, t_n^{a_n})^a$, and $x < y'$ for all $x \in \mathrm{dom}(\Gamma)$, then

$$\Gamma' \vdash f[y{:=}y'] : (t_1^{a_1}, \ldots, t_{i-1}^{a_{i-1}}, t_{i+1}^{a_{i+1}}, \ldots, t_n^{a_n}, t_i^{a_i})^a.$$

Γ' is the subset of $\Gamma \cup \{y{:}t_i^{a_i}, y'{:}t_i^{a_i}\}$ such that $\mathrm{dom}\Gamma'$ contains all and only those variables occurring in $f[y{:=}y']$;

8. **(quantification)** If y is the ith free variable in f (according to the order on variables), and $\Gamma \cup \{y{:}t_i^{a_i}\} \vdash f : (t_1^{a_1}, \ldots, t_n^{a_n})^a$, then

$$\Gamma \vdash \forall y{:}t_i^{a_i}[f] : (t_1^{a_1}, \ldots, t_{i-1}^{a_{i-1}}, t_{i+1}^{a_{i+1}}, \ldots, t_n^{a_n})^a.$$

There is a major notational difference between the propositional function notation of [4] and our own (which can be seen in the definition of type judgments just above). The authors of [4] attach type labels to quantified variables. This is certainly not in the spirit of *PM*, where there is no notation for types at all. It would be possible to modify their system to make this unnecessary, but it would then be necessary to include type hypotheses for quantified variables in the environment.

The authors of [4] are forced by the structure of their system into adopting a much more complicated definition of substitution (by "substitution", we mean "substitution into propositional (or pf) notations" throughout this paragraph; substitution into type notations is used in the definition of our system of type judgments, but involves no logical difficulties). The difficulty is that some of the rules of their system of type judgments are defined in terms of the notion of substitution (as can be seen above), so substitution has to be defined prior to the adoption of the type system. As a result, a complicated detour through lambda-calculus is required to define the notion of substitution successfully, whereas in our development we are able to correct the natural definition of substitution by appealing to the (simple) theory of types, because we make no use of substitution in our definition of type judgments. Once we have defined types, we are able to use the natural definition of substitution, with the additional stipulation that all terms involved have to be well-typed and substitutions for variables have to reflect the inferred types of the variables.

Polymorphism is represented differently in the two systems. In the system of [4], there are no polymorphic type judgements, but a term may be assigned different types in different contexts. In our system, a single (but possibly polymorphic) type is always assigned to a term, whose structure is general enough to indicate all possible types for the term. The side conditions on polymorphic orders generated by the complete algorithm for *RTT* complicate this picture somewhat.

The range of terms recognized as well-typed by our system is far larger than that recognized by the system of [4], and apparently larger than that recognized by *PM*!. The system of [4] 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 [4] 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, where they assert that all non-predicative propositional functions are to be formed from predicative ones by generalization, and that no bound variables of non-predicative type are needed. However, there is a problem with this (also apparently recognized by the authors of *PM* in an immediately following remark on p. 165): without variables of possibly non-predicative type, one cannot express the axiom of reducibility in a typable form. *PM* makes a special provision for this by introducing application of function variables without assigned order on p. 165; we suppose that terms with such variables in them would not define propositional functions for *PM* if it was desired not to have types with impredicative components. The system of *PM* can conveniently restrict impredicativity to the top level of types as they do (while apparently forbidding quantification over impredicative types) because the axiom of reducibility allows one to associate with each element of an impredicative type with predicative components a coextensional element of the predicative type with the same components, and one can quantify over this type; in the absence of the axiom of reducibility, one would need to be able to quantify over impredicative types directly in order to be able to say anything about them, and this would mean that one could define propositional functions with more complex types.

The system of [4] is more modern in appearance than ours; we do recognize this as an advantage of that system. Our program of using propositional notations themselves as environments has at least one strange effect to go along with its advantages. In the simple theory of types, it is reasonable to avoid assigning types to bound variables (that is, to define the type algorithm in such a way as to effectively rename bound variables as they are encountered, so that a bound variable may have the same shape as a free variable or differently bound variable of a different type elsewhere without causing a type conflict). However, without a conventional environment the only way to associate a polymorphic type with a variable seems to be to name the polymorphic type after the variable to which it is assigned. This makes it impractical to attempt to rename variables bound in arguments of propositional functions, which has odd effects on typing in the simple theory of types which will be seen in the examples. In the ramified theory, it seems to be best to type all variables which appear, free or bound (even in [4], the authors remark that it is necessary to assign types to some bound variables).

We believe that our system is better in certain ways than the system of [4]. The fact that our notation for propositional functions does not require type indices is truer to the original system of PM. The fact that the definition of our type inference system does not depend on the notion of substitution allows the definition of substitution to be simpler and more natural in our formalization. We believe that our system lends itself better to mechanical implementation, but this is perhaps unfair since the system described here was reverse-engineered from a mechanical implementation (though it should be noted that the formal system was reverse-engineered from an early version of the program which didn't work very well, and improvements in the formalization then drove improvements in the program). It would be interesting to see whether and how well the system of [4] lends itself to automation. The system of [4] handles bound variables in a way a little more in accord with modern tastes than ours does. The system of [4] is more faithful to PM in limiting types to those with predicative components, but we feel that any serious attempt to work in RTT without reducibility would require the lifting of this restriction.

The simple theory of types is of course very similar to quite standard type systems except for its lack of head binders in function notation, and the type inference algorithm for this system is recognizably of a standard kind, except for the adaptations to the head-binder-free notation for functions. The ramified theory of types is very eccentric as a type system, and the complete algorithm we exhibit for it is unusual in its need to reason about arithmetic in order to manage order unification. From the standpoint of modern theories of types, the orders of RTT are peculiar union types, in which quite heterogeneous kinds of thing are conglomerated together.

8 Examples

True to the historical origins of this paper, we will begin by presenting some examples from [4]. Some features of the output of our software are suppressed.

We are running the RTT checker, but in many cases this will not be obvious, as our system does not display order superscripts on types unless the order is more than one greater than the maximum order of the component types.

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

Just as in example 49, clause 1, of [4], the propositional notation $S(a_1, a_2)$ (the computer requires a suffix on the predicate indicating its arity) 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; our usage of suffixes on predicates to indicate arity forbade reproducing the form $R_1(x_1) \vee R_2(x_1)$ of the original.

```
Term input:
     (R1(x1) v S1(x2))
final type list:
     x1:  0
     x2:  0
unconditional type:
     (0,0)
```

In this example, we have two distinct free variables, so we get a pf with two arguments (both of type 0).

The treatment of the terms of the last two examples in the system of this paper is much the same. In both terms, the list of free variables is generated, then each free variable is typed using local rules, and the types of the free variables are listed to form the type of the propositional function. But the treatment of the last two examples is quite different in the system of [4], for reasons best discovered by examining derivations in the style of that paper which we now exhibit (there are inessential differences between the notations used in the two systems for the pfs involved). The rule numbers refer back to definition 40 of [4], which is reproduced in the previous section of this paper. The derivations under the heading 1 are taken from [4], example 49, while we set up the derivation under heading 2 ourselves for comparison.

1. $$\dfrac{\vdash \mathtt{R_1(a_1)} : () \qquad\qquad \vdash \mathtt{R_2(a_1)} : ()}{\vdash \mathtt{R_1(a_1)} \lor \mathtt{R_2(a_1)} : ()}\ \text{rule 2}$$

   but *not*:

   $$\dfrac{\mathtt{x_1} : 0 \vdash \mathtt{R_1(x_1)} : (0) \qquad\qquad \mathtt{x_1} : 0 \vdash \mathtt{R_2(x_1)} : (0)}{\mathtt{x_1} : 0 \vdash \mathtt{R_1(x_1)} \lor \mathtt{R_2(x_1)} : (0,0)}\ \text{rule 2}$$

   ($\mathtt{x_1} \not< \mathtt{x_1}$ because $<$ is strict). To obtain $\mathtt{R_1(x_1)} \lor \mathtt{R_2(x_1)}$ we must make a different start:

   $$\dfrac{\dfrac{\vdash \mathtt{R_1(a_1)} : () \qquad\qquad \vdash \mathtt{R_2(a_1)} : ()}{\vdash \mathtt{R_1(a_1)} \lor \mathtt{R_2(a_1)} : ()}\ \text{rule 2} \qquad\qquad \vdash \mathtt{a_1} : 0}{\mathtt{x_1} : 0 \vdash \mathtt{R_1(x_1)} \lor \mathtt{R_2(x_1)} : (0)}\ \text{rule 3;}$$

   $$\dfrac{\dfrac{\vdash \mathtt{R_1(a_1)} : () \qquad\qquad \vdash \mathtt{R_2(a_1)} : ()}{\vdash \mathtt{R_1(a_1)} \lor \mathtt{R_2(a_1)} : ()}\ \text{rule 2} \qquad\qquad \vdash \mathtt{a_1} : 0}{\mathtt{x_1} : 0 \vdash \mathtt{R_1(x_1)} \lor \mathtt{R_2(x_1)} : (0)}\ \text{rule 3;}$$

2. $$\dfrac{\dfrac{\vdash \mathtt{R_1(a_1)} : () \qquad \vdash \mathtt{a_1} : 0}{\mathtt{x_1} : 0 \vdash \mathtt{R_1(x_1)} : (0)}\ \text{r. 3} \qquad\qquad \dfrac{\vdash \mathtt{R_2(a_1)} : () \qquad \vdash \mathtt{a_1} : 0}{\mathtt{x_2} : 0 \vdash \mathtt{R_2(x_2)} : (0)}\ \text{r. 3}}{\mathtt{x_1} : 0, \mathtt{x_2} : 0 \vdash \mathtt{R_1(x_1)} \lor \mathtt{R_2(x_2)} : (0,0)}\ \text{r. 2}$$

   The application of rule 2 here is correct because $\mathtt{x_1} < \mathtt{x_2}$.

In the system of [4], the term $R_1(x_1) \lor R_2(x_1)$ is typed by first considering the typing of $R_1(a_1) \lor R_2(a_1)$, 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 a new component with type 0 into the type () of $R_1(a_1) \lor R_2(a_1)$ (the new component correlates with the new variable which replaces a_1) to obtain the type (0). The term $R_1(x_1) \lor R_2(x_2)$ 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 example should make clear the quite different flavors of the two approaches.

```
Term input:
     (x2(a1) v S1(a1))
final type list:
     x2:  (0)^max(|x2|,1)
unconditional type:
     ((0)^max(|x2|,1))
```

This is the first example given in example 49 in [4]. Our system tells us that the function x_2 (called z in the original) can have a type of any order with sole component 0: the order $|x_2|$ of this type will be at least 1, which is expressed by writing it as the maximum of 1 and $|x_2|$ (this is an order with a bounding variable).

```
Term input:
     [x1](x1() v ~x1())
final type list:
     x1:  ()^max(|x1|,0)
unconditional type:
     ()^max(|x1|+1,1)
```

This is example 51 from [4]. 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. As in the previous example, there is no upper bound on the possible order of the type here. This can be changed, though, using the "predicativity" qualifier of propositional function application:

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

Now we know that the order of x_1 is 0 (since it is the smallest possible order it is not displayed) and the order of the type of the whole term is seen to be exactly 1.

We have yet to see an explicit polymorphic type. This can be remedied by considering the term in Remark 58 of [4].

```
Term input:
     x2(x1)
final type list:
     x1:  [x1]
     x2:  ([x1])^max(|x1|+1,|x2|,1)
unconditional type:
     ([x1],([x1])^max(|x1|+1,|x2|,1))
```

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 $([x_1], ([x_1]))$, in which the order of the components is determined by the fact that x_1 is alphabetically prior to x_2. The order index on the type $([x_1])$ of x_2 appears because we have no order restriction on x_2. We get a prettier display if we change to predicative application:

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

In [4], this is also an example of polymorphism (the pf is written z(x) instead of our $x_2(x_1)$): two different derivations are given, each yielding a different type,

$$\dfrac{\dfrac{\vdash \mathtt{R}(\mathtt{a}_1):() \qquad \vdash \mathtt{a}_1:0}{\mathtt{x}:0 \vdash \mathtt{R}(\mathtt{x}):(0)}\text{ rule 3}}{\mathtt{x}:0,\mathtt{z}:(0) \vdash \mathtt{z}(\mathtt{x}):(0,(0))}\text{ rule 4}$$

versus

$$\dfrac{\dfrac{\vdash \mathtt{R}(\mathtt{a}_1):()}{\mathtt{x}:() \vdash \mathtt{x}():(())}\text{ rule 4}}{\mathtt{x}:(),\mathtt{z}:(()) \vdash \mathtt{z}(\mathtt{x}):((),(()))}\text{ rule 4,}$$

whereas in our system we get a single computation showing us what *all* types look like.

If we supply more information in the context (the context can only be manipulated in our system by embedding the term to be typed in a larger term), the polymorphic 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 additional local information in the term that the type of x_1 is 0, so we get a more specific type for the whole propositional function.

Here we give more complete output for a larger example term. The example propositional function is adapted from the definition of a real number as a Dedekind cut in example 71 in [4]. Predicative propositional function application has been used throughout to simplify the display.

```
Term input:
     ((([Ex2]x1!(x2) and [Ex2]~x1!(x2))
     and [x2][x3](x1!(x3) implies (L2(x3,x2) implies x1!(x2))))
     and [x2](x1!(x2) implies [Ex3](x1!(x3) and L2(x2,x3))))
basic list:
     x1:  ([x2])
     x1:  [x1]
     x1:  ([x3])
     x2:  [x2]
     x2:  0
     x3:  0
     x3:  [x3]
unification list:
     x~2:  [x~1]
     x~2:  ([x2])
     x~2:  ([x3])
     x~1:  ([x3])
     x~1:  ([x2])
     x~1:  [x~2]
     x1:  ([x3])
     x1:  ([x2])
     x1:  [x1]
     x2:  0
     x2:  [x2]
     x2:  [x3]
     x3:  [x3]
     x3:  [x2]
     x3:  0
final type list:
     x~2:  (0)
     x~1:  (0)
     x1:  (0)
     x2:  0
     x3:  0
unconditional type:
     ((0))
```

The additional displays shown here (suppressed in previous examples) give some hint at the internal processes of the type algorithm. The "basic list" contains the local information about types of variables. The "unification list" contains information derived by unifying types pairwise. The final list is obtained by the process of eliminating superfluous type variables by global substitutions. The additional variables x_{-1} and x_{-2} are used as "placeholders" internally by the algorithm (in its internal representation of type equality judgments). The type obtained is the same as the type $((0^0)^1)^2)$ claimed for this propositional function in [4]: recall that minimal order indices are not displayed.

We give an example of the curious type phenomena which can result from identifications of variables with bound variables in propositional function arguments which happen to be used in the names of polymorphic types.

```
- test "x1(x3(x2))";

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

((([x2],([x2]))))
```

The format is different because we are here using the STT type algorithm. The final line is the type of the term. $x_1(x_3(x_2))$ contains one free variable x_1, which is a function taking one argument of the type of $x_3(x_2)$; $x_3(x_2)$ is itself a function of two arguments, x_2, whose type is $[x_2]$ (ambiguous) and x_3, whose type is $([x_2])$, since it takes one argument of type $[x_2]$. The type of $x_3(x_2)$ is thus $([x_2], ([x_2]))$ (recall that arguments are supplied to a propositional function in alphabetical order of the free variables representing them), the type of x_1 is $(([x_2], ([x_2])))$and the type of $x_1(x_3(x_2))$ is $((([x_2], ([x_2]))))$.

The term $x_1(x_2(x_1))$ apparently has exactly the same meaning, since $x_2(x_1)$ is the same object as $x_3(x_2)$, but the result of typing this term is quite different.

```
- test "x1(x2(x1))";

basic list:
     x1:  (([x1],([x1])))
unification list:
     x1:  (([x1],([x1])))
final type list:
     x1:   !?!

!?!
```

This fails to type. The difficulty is that the types of the two occurrences of x_1 are forced to be the same, and this results in circularity.

In other cases this is harmless in our implementation of STT:

```
- test "x1(x1(x2))";

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

(((([x2]),[x2])))
```

There is no problem here because, although the types of the two occurrences of x_1 are incompatible, all information about the type of x_1 is discarded when the typing of the argument $x_1(x_2)$ is finished, since it is not used in the polymorphic type of this term. But the RTT algorithm will not accept this:

```
Term input:
     x1!(x1!(x2))
basic list:
     x1:  ((([x2]),[x2]))
     x1:  [x1]
     x1:  ([x2])
     x2:  [x2]
unification list:
     x~2:  [x~1]
     x~2:  ([x2])
     x~2:  ((([x2]),[x2]))
     x~1:  ((([x2]),[x2]))
     x~1:  ([x2])
     x~1:  [x~2]
     x1:  ((([x2]),[x2]))
     x1:  ([x2])
     x1:  [x1]
     x2:  (([x2]),[x2])
     x2:  [x2]
final type list:
     x~2:   ?!?
     x~1:   ?!?
     x1:   ?!?
     x2:   ?!?
unconditional type:
     ?!?
```

Here type information from the propositional function argument is preserved, and it is noticed that x_1 needs to be assigned type (x_2) and type $((([x_2]), [x_2]))$, which are incompatible.

We now give some examples of the application of the complete type algorithm for RTT.

```
Term input:
     (x1(x2,x2) v x1([x3]x3(x4),[x5][x7]x7(x5,x6)))
unconditional type:
     ?!?
conditional type:
     ((([x6])^max(|x3|+1,|x6|+2,2),
     ([x6])^max(|x5|+2,
     |x6|+2,|x7|+1,2))^max(|x1|,|x3|+2,|x5|+3,|x6|+3,|x7|+2,3),
     ([x6])^max(|x3|+1,|x6|+2,2))
   WITH
     |x3| <= |x7| and
     |x5|+1 <= |x7| and
     |x6|+1 <= |x7| and
```

```
      |x7|+2 <= |x1| and
      |x7| <= |x3|
```

The complete checker tells us that this propositional function does not type under the provisional algorithm (under the heading "unconditional type"), then gives a type and a set of conditions on polymorphic orders under which this propositional function is well-typed in RTT.

Here is another example in which there are two different conditions under which the given propositional function 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|
```

We will attempt to talk our way through the typing of the second example. 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.

9 Applications to Proof Checking

We briefly discuss the application of the typing software in the development of a proof checker for the system of *PM*, as expressed in our version of the notation of [4].

Details of the proof checker itself are not especially relevant at this point (we are attempting to follow the rules of inference in *PM* closely). But there are a couple of observations worth making.

One never has any occasion to see a type index in the course of using the proof checker. This is appropriate, since *PM* does not even have notation for types, so we never see such notation in *PM*'s theorems or proofs.

The type checker is used ubiquitously as part of the process of checking well-formedness of propositions and propositional functions. This is natural.

There is one place in the logic where the type checker plays an important and perhaps not entirely obvious role. This is in the implementation of the rule of *modus ponens*. When one deduces a proposition Q from premises P and $P \rightarrow Q$, there is a subtle fallacy which can occur, and which use of the type checker enables one to avoid.

All propositions of *PM* (and so all theorems of the nascent proof checker) are to be understood in the most general possible way: they are to be true for all possible values of their free variables under all possible assignments of type. The difficulty is that the form of the proposition $P \rightarrow Q$ may give more type information than Q (and also more than P, but this is harmless). So if the modus ponens rule were implemented in a naive way, it might be possible to deduce a proposition Q which is true for all type assignments to Q which render $P \rightarrow Q$ well-typed, but not for some other type assignments for Q. So the proof checker needs to check that the type checking of $P \rightarrow Q$ gives the same type information about Q that the type-checking of Q alone gives.

We make the following conjectures, which we plan to discuss in a later paper where we will have more to say about the proof checker.

If types constructed from the type of propositions are admitted, the use of the naive form of modus ponens will lead to paradox. The reason for this is that under reasonable assumptions the type of propositions, for example, has only two elements, so one could prove an assertion like $(\exists ab.\forall x.x = a \vee x = b)$ using hypotheses from which one could infer that x was a proposition, but produce the conclusion in systematically ambiguous form. This conclusion leads to contradiction because one can prove that some other types (also constructible from the type of propositions) have more than two elements: for example, $((), ())$ has four elements.

On the other hand, if types constructed from the type of propositions are not permitted $(()$ can occur only as the type of a proposition, not as the type of a propositional function) then we believe that use of the naive rule of modus ponens does not lead to contradiction, though it leads to unexpected results, such as the ability to prove the "axiom of infinity" in pure logic. The reason for this has to do with the relationship between the ramified theory of types and the set theory *NFP* defined by Marcel Crabbé in [1], which is the predicative version of Quine's "New Foundations". I have shown elsewhere (in [2]) that *NFP* is mutually interpretable with the ramified theory of types with the axiom of infinity. Though there are some details to check, we believe that it is possible to construct a model of the ramified theory of types, using its relationship with *NFP*, in such a way that all the types are isomorphic in a suitable sense, so that if Q is a theorem for any assignment of types to its variables, it is a theorem for all assignments of types to its variables, which is a sufficient condition for the naive rule of modus ponens to be valid. If the type of propositions is permitted as a component, then it is possible to construct types of distinct finite cardinalities, which cannot be isomorphic with one another, so it is necessary to forbid the use of the type of propositions as a component type if one wishes to exploit this (presumed) result.

References

[1] Crabbé, M. "On the consistency of an impredicative subsystem of Quine's NF". *Journal of Symbolic Logic* 47 (1982), pp. 131-136.

[2] 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.

[3] 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`.

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

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

[6] Peressini, Anthony F., "Cumulative versus noncumulative ramified types", *Notre Dame Journal of Formal Logic*, vol. 38, no. 3, summer 1997.

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