The implementation of an unusual higher-order logic*

M. Randall Holmes

Boise State University

Abstract. The aim of this paper is to discuss issues which arose in the implementation of a theorem prover based on an unusual higher-order logic, namely the set theory *NFU* proposed by Jensen in 1969 as a variant of the notorious theory "New Foundations" of Quine. Both of these theories provide a universal set, something not found in familiar set theories. This is implemented in the form of an equivalent λ-calculus developed by the author. The theorem prover in which it is implemented is Watson, developed by the author since 1990 (an earlier version of the prover was called Mark2). The implementation of quantification using the higher-order logic of Watson is described in the paper, in order to give some impression of issues that arose in the practical implementation of this logic.

1 Introduction

References for the set theories *NFU* and *NF* are the papers [11] of Jensen and [13] of Quine. More recent and longer treatments are given in Forster's [3] and our [10]. For the Watson theorem prover see [8]. Papers on the logic of Watson are [7] (which introduced the "stratified λ-calculus" implemented in Watson, though this was implicit in our earlier [5]) and [9]. [8] and [9] cover all aspects of the logic of the prover, and if anything shortchange issues connected with its higher-order logic (and its relation to *NFU*); [7] emphasized this but was written before much practical experience had been gained with implementation and use of the stratified λ-calculus.

Most refinements of the implementation of the higher-order logic have been made in response to problems encountered in the course of implementing first-order logic. The logic of quantification is not handled by logical primitives in the logic of Watson, but implemented in terms of the higher-order logic. Rigidities in the naive initial implementation of the stratified λ-calculus led to difficulties in the handling of quantification, which have at this point been repaired; the repair process led to a deeper understanding of the practical aspects of implementing this logic, which is something of which we hope to give some impression in this paper.

* The author gratefully acknowledges the support of US Army Research Office grant DAAG55-98-1-0263

The higher-order logic of Watson is unusual in being untyped – at least in a sense. The sense in which it is untyped deserves some investigation: there is a polymorphic type theory lurking invisibly at its foundations, and some of the recent developments have given it explicit machinery for handling absolute types. Part of our program is the investigation of the implementation of typed theories in the untyped logic of Watson: the way in which types are implemented involved the application in a practical context of concepts hitherto of purely theoretical interest in *NF*-like theories (in particular, the notion of a *strongly cantorian set*).

2 What is the Logic?

In this section we briefly describe the set theory *NFU* and the stratified λ-calculus.

NFU is a first-order theory with equality, membership, and the projection relations π_1 and π_2 of a primitive ordered pair as primitive predicates. We enrich our language with definite descriptions $(\iota x.\phi)$, intended to mean "The unique x such that ϕ" if there is such an object, and otherwise the empty set. The empty set $\emptyset$ is given as a primitive constant.

The set theoretical axioms of this version of *NFU* follow.

empty set: $x \notin \emptyset$
weak extensionality: $x \in A \to (A = B \equiv (\forall y.y \in A \equiv y \in B))$
ordered pair: $(\forall z.(\exists xy.(\forall w.z\pi_1 w \to w = x \wedge z\pi_2 w \to w = y)))$
$\wedge\ (\forall xy.(\exists z.(\forall w.(w\pi_1 x \wedge w\pi_1 y) \equiv w = z)))$

The first axiom tells us that the empty set has no elements. The second one says that nonempty sets with the same elements are equal: there may be many objects with no elements. The third axiom tells us that the projection relations really are projection relations. They allow us to introduce the ordered pair $\langle x, y\rangle$ as $(\iota z.z\pi_1 x \wedge z\pi_2 y)$, and ensure that this pair has the correct properties. The pair is not assumed to be surjective, though this assumption could be made and has some advantages. The presence of the empty set as a primitive allows us to make the following

Definition: We define $\mathtt{set}(x)$ (read "x is a set") as $x = \emptyset \vee (\exists y.y \in x)$.

It follows from the axioms of empty set and weak extensionality that sets with the same elements are equal. We call non-sets *urelements*.

We now present the comprehension scheme of *NFU*. This requires a definition as a preamble.

Definition: Let ϕ be a formula in the language of *NFU*. We say that a partial function σ from the terms of the language of *NFU* to the natural numbers (or, equivalently, to the integers) is a *stratification* of ϕ if it satisfies the conditions that for each atomic subformula of ϕ of the forms $x = y$, $x\pi_1 y$ or $x\pi_2 y$ with x and y both in the domain of σ, we have $\sigma(x) = \sigma(y)$ and for

each atomic formula $x \in y$ with x and y both in the domain of σ, we have $\sigma(x) + 1 = \sigma(y)$, while for each complex term $(\iota x.\phi)$ which appears in ϕ we have $\sigma((\iota x.\phi)) = \sigma(x)$: further, we require that every variable bound in ϕ and every definite description in ϕ belong to the domain of σ. We say that the formula ϕ is *stratified* if there is a stratification of ϕ.

We will regard a formula as stratified if it is possible to get a stratification by renaming bound variables. It is convenient to be able to talk about stratified terms as well as stratified formulas: a formula $(\iota x.\phi)$ is said to be stratified under the same conditions that $(\forall x.\phi)$ is said to be stratified.

We can now present the

Axiom scheme of stratified comprehension: For each formula ϕ in which the variable A is not free, $(\exists A.(\forall x.x \in A \equiv \phi))$ is an axiom if it is stratified. (Note that it is not sufficient under our definition for ϕ to be stratified: it is necessary for the variable x, which is not necessarily bound in ϕ, to be in the domain of the stratification).

It follows that for each formula ϕ such that $(\exists A.(\forall x.x \in A \equiv \phi))$ is stratified there is a unique *set* $(\iota A.\mathtt{set}(A) \wedge (\forall x.x \in A \equiv \phi))$, which we can abbreviate $\{x \mid \phi\}$.

NFU as presented here is an untyped theory, in the sense that all objects are of the same sort. But the criterion of stratification for comprehension axioms is a typability criterion. A formula is stratified just in case it can be typed in a version of Russell's simple theory of types (as simplified by Ramsey), with constants (and parameters) interpreted polymorphically. We reflect this in our terminology by referring to the value of a stratification of ϕ at a term which appears in ϕ as the *relative type* of that term in ϕ.

We have extended the language of *NFU* with the definite description operator: it is straightforward to show that the criterion of stratification for formulas with definite descriptions given here is precisely equivalent to the criterion for stratification for equivalent formulas obtained by the usual contextual elimination of definite descriptions (due to Russell). The advantage of this extension is that it makes it clear how stratification works for term constructions. *NFU* could be further extended by allowing the Hilbert operator $(\epsilon x.\phi)$ with the same stratification conditions as $(\iota x.\phi)$; this would have the effect of adjoining the axiom of choice to the theory.

We have essentially strengthened *NFU* by including type level projection operators in our language: this has essentially the same effect (mod technicalities) as adjoining an axiom of infinity to *NFU* as formulated in [11] (the consistency of *NFU* + Infinity is shown there as well). The type level projection operators have the further technical advantage that they simplify the properties of the ordered pair and of functions and relations. What makes the pair $\langle x, y\rangle$ whose definition is given above "type-level" is that it is assigned the same type as its projections x and y for purposes of stratification. We could define $\{x\}$ as $\{y \mid y = x\}$ and $\{x, y\}$ as $\{z \mid z = x \vee z = y\}$, and then define the usual Kuratowski ordered

pair $\{\{x\},\{x,y\}\}$. This term is well-typed in the sense that it can appear in stratified formulas, but the value of a stratification at a term $\{\{x\},\{x,y\}\}$ will be two higher than its value at x and y, so it is not type-level. This would prove inconvenient in the handling of functions and relations.

An advantage of defining *NFU* in a way which includes term constructions (and an ordered pair) is that it requires less preliminary development to present the natural interpretation of the stratified λ-calculus in *NFU* than it does in the usual formulation. We present some definitions.

Definition: We define $\mathtt{function}(f)$ (read "x is a function") as $\mathtt{set}(f) \wedge (\forall zw.(z \in f \wedge w \in f) \rightarrow ((\exists xy.z = \langle x,y\rangle) \wedge (\forall x.(z\pi_1 x \wedge w\pi_1 x) \rightarrow z = w)))$. This says that all elements of f are ordered pairs and no two distinct elements of f have the same first projection. (Note that the empty set is a function but other objects with no elements (urelements) are not).

Definition: We define $f(x)$ as $(\iota y.\, \langle x,y\rangle \in f)$. We define $\mathtt{dom}(f)$ as $\{x \mid (\exists y.\, \langle x,y\rangle \in f)\}$.

Note that $f(x)$ has the same relative type as x in any stratification, while f has relative type one higher than that of x or $f(x)$. If we had used the Kuratowski pair, f would have had to have type *three* higher than that of x or $f(x)$.

We can prove an extensionality theorem for functions, which we state without proof:

Extensionality theorem for functions: $(\mathtt{function}(f) \wedge \mathtt{function}(g) \wedge \mathtt{dom}(f) = \mathtt{dom}(\mathtt{g}) \wedge (\forall x.f(x) = g(x))) \rightarrow f = g$.

We will usually concern ourselves with functions f such that $\mathtt{dom}(f) = \{x \mid x = x\}$ (functions of universal domain), so the requirement that functions have the same domain will automatically be satisfied.

Further, we can present an abstraction meta-theorem (analogous to the stratified comprehension axiom).

Definition: We define $\{\langle x,y\rangle \mid \phi\}$ as $\{z \mid z = \langle x,y\rangle \wedge \phi\}$ (where z is a new variable).

Definition: Let T be a term. We define $(\lambda x.T)$ as $\{\langle x,y\rangle \mid y = T\}$.

Stratified Abstraction Meta-Theorem: For any term T, $(\forall x.(\lambda x.T)(x) = T)$ is a theorem if it is stratified.

Note that the term $(\lambda x.T)$ represents a function of universal domain if it is not the empty set due to a failure of stratification. The term $(\lambda x.T)$ has relative type one higher than the variable x, if it has a stratification.

Additional term constructions are needed for the interpretation of the logic of Watson in *NFU*. We need terms to code truth values: $V = \{x \mid x = x\}$ (the universal set) is used to represent $\mathtt{true}$; $\emptyset$ is used to represent $\mathtt{false}$. To code a formula ϕ, use the term $(\iota x.\phi \rightarrow x = V)$ (if ϕ is true, this is V; if ϕ is false, this is $\emptyset$ by the default interpretation of definite descriptions). The definition of terms by cases is important under Watson: we define $\mathtt{if}\, A\, \mathtt{then}\, B\, \mathtt{else}\, C$ as

$(\iota x.A = V \to x = B \wedge T \neq V \to x = C)$: if A is the universal set (which codes `true`) then this term is equal to B, and otherwise it is equal to C.

The stratified λ-calculus is an equational theory. Its basic term constructions are variables, the constants `true`, `false`, the function application $T(U)$, the ordered pair $\langle T, U\rangle$ and the projection functions π_1 and π_2, definition by cases `if` T `then` U `elseV`, equation terms $T = U$, and the abstraction term $(\lambda x.T)$.

A complete formal description of this calculus is given in [9]. Here we restrict our attention to issues related specifically to the higher-order logic, especially the application and implementation of the notion of stratification.

The formulation of the abstraction axiom of stratified λ-calculus is quite analogous to the formulation of the stratified comprehension scheme. The clauses of the definition of relative type given below are what would be expected from the definition of these constructions given above in *NFU*.

Definition: A *stratification* of a term T is a partial function σ from terms to integers with all subterms of T which contain variables in its domain, having the following properties:

pair, equality, cases: If σ maps $U = V$ or $\langle U, V\rangle$ to n, it also maps U and V to n. If σ maps `if` U `then` V `else` W to n, then it maps U, V, and W to n.

function application: If σ maps $U(V)$ to n, it maps U to $n+1$ and V to n.

abstraction: If σ maps $(\lambda x.U)$ to n, it maps x and U to $n-1$.

If there is a stratification of T, we say that T is *stratified*. We will regard T as stratified if a renaming of bound variables will enable one to construct a stratification. The value of a stratification of T which sends T to 0 at a subterm U of T is called the *relative type* of U in T, and is uniquely determined under this definition.

The abstraction scheme of the stratified λ-calculus is as follows:

Abstraction Scheme: For any stratified term $(\lambda x.T)$, $(\lambda x.T)(U) = T[U/x]$, where $T[U/x]$ denotes the result of substituting U for x in T (with due respect for the usual technicalities involving bound variables).

The stratified λ-calculus has no explicit extensionality axiom. But note that if we can prove $(\forall x.f(x) = g(x))$, we will be able to prove $(\lambda x.f(x)) = (\lambda x.g(x))$ using properties of equality. A weak extensionality is implicit in the ability to substitute equals for equals inside abstraction terms, which lets one prove that co-extensional abstraction terms are equal. Strong extensionality asserts $f = (\lambda x.f(x))$ for all f; this says "every object is a function". The stratified λ-calculus with strong extensionality is known to be equivalent to *NF* (see [5]), which is not known to be consistent.

It turns out to be convenient to allow unstratified λ-terms to be well-formed in Watson (earlier versions of Watson, and the formalization of stratified λ-calculus in [7], treated stratification as a condition for well-formedness of terms). The details are in [9]; there is further brief discussion of the use of unstratified abstraction terms below.

3 Implementing the Logic

We give an informal summary of Watson term constructions.

free variables: A question mark followed by a string of alphanumerics which is not a numeral, as `?x`

bound variables: A question mark followed by a numeral, as `?2`

constants: Strings of alphanumerics, as `a`, `12`. Numerals and a few special objects are predeclared. Other constants are declared in Watson theories.

infix terms: A term, followed by an operator (a string of special characters) followed by a term. Parentheses may be used as needed; Watson supports user defined operator precedence and left or right grouping. The predeclared infix `@` represents function application; the predeclared infix `,` represents ordered pair. Operators can be declared or defined in user-defined theories.

prefix terms: An operator, followed by a term. Users can declare or define prefix operators.

abstraction terms: A term enclosed in brackets. De Bruijn levels are used to determine variable binding: the bound variable `?1` is bound in the outermost set of brackets in a context, the bound variable `?2` is bound in the second from the outside, and so forth.

We describe the implementation of stratification in the prover. Each infix operator has a *left type* and a *right type*; each prefix operator has a *right type*. The left and right types are the types which would be assigned to the left and right arguments of a term built with the operator if the whole term were assigned type 0. For example, `+` (in a reasonable implementation of arithmetic) has left type and right type 0, because the arguments of the addition operator have the same relative type as the whole term; `@`, the function application operator, has left type 1 and right type 0, because the left argument of an application is one type higher than the whole application term, while the right type has the same type as the whole term. Negative type displacements are possible. For example, the unary operator `‘iota` (an initial backquote converts an alphanumeric string into an operator in Watson syntax) defined by the equation `‘iota ?x = [?x=?1]` (this codes the construction of singleton sets, if we code sets as characteristic functions), has right type -1: the type of the argument `?x` is one lower than that of its “singleton set” `‘iota ?x`.

We define an ML datatype representing a subset of the Watson language: it would clearly not be hard to extend the data type and the definition of the stratification function to the full language.

free variables: If `s` is a string, `FreeVar s` is a term.

bound variables: If `n` is a numeral, `BoundVar n` is a term.

infix term: If `T` and `U` are terms and `s` is a string, `Infix(T,s,U)` is a term (`s` is used here as the name of an operator).

abstraction term: If `T` is a term, `Function T` is a term.

We assume the existence of a type of finite partial functions from natural numbers to integers augmented with an error value `Error` (distinct from the empty function) and a function `merge` which takes two finite functions as arguments and returns the union of the two functions if the union is a function and `Error` if the union of the two functions is not a function or if one of the arguments is `Error`.

We are provided with functions `left` and `right` which will give us the left and right types of an operator when applied to its name.

The stratification function `strat` takes three arguments, a natural number `level` indicating how many abstraction brackets enclose the current context (this manages the reference of bound variables), an integer `type` indicating the relative type of the (occurrence of a) subterm being considered, and an element of the term data type. We define the function `strat` by recursion on the term type. `strat` will return a finite function (or `Error`).

free variables: `strat level type (FreeVar s)` will be the empty function.

bound variables: `strat level type (BoundVar n)` will be the finite function {`(n,type)`} unless `n > level`, signalling a variable binding error, in which case `Error` will be returned.

infix terms: `strat level type (Infix(T,s,U))` will be `merge`(`strat level (type+left(s)) T`,`strat level (type+right(s)) U`).
`merge` is used to check consistency of relative typing.

abstraction terms: `strat level type (Function T)` will be the restriction to values less than or equal to `level` of `merge`({`(level+1,type-1)`},`strat (level+1) (type-1) (Function T)`) (where any restriction of `Error` is taken to be `Error`). The fact that the value at `level+1` is discarded allows us to stratify terms that would otherwise need to be stratified by renaming bound variables. The use of `merge` ensures that all occurrences of the currently bound variable are assigned the correct type (if indeed the currently bound variable is assigned any type at all).

For any term `T` not a subterm of any abstraction term, the value `strat 0 0 T` will be the empty function if `T` is stratified and `Error` if it is not.

There are two versions of the stratification function, differing in their treatment of free variables. The usual version (given here) treats free variables as polymorphic (i.e., occurrences of the same free variable may occur in a stratified term with different relative types). A version used by the definition utility of Watson also types free variables: functions and operators can be given parameterized definitions in Watson, and in such definitions it must be possible to demand consistent relative types for free variables as well. Constants (not provided here) are always treated polymorphically.

We begin our discussion of the implementation of Watson with this stratification function in mind. We will discover fairly soon that it is too rigid and needs to be refined; but we will discuss the refinement after discovering why it is needed.

4 A Case Study: Implementing Quantifiers Using Stratified Abstraction

The basic logic of Watson is equational. Watson theorems are equations between terms, implicitly universally quantified over their free variables, and the basic logical move in Watson is the application of such an equational theorem as a rewrite rule.

Watson has no built-in logic of propositions (quantified or otherwise). Propositional logic can be implemented using the built-in logic of expressions defined by cases or as a user-declared equational theory (which has advantages if a non-classical logic (as a temporal logic or a logic of belief) is to be implemented). We will need to give a little thought to the properties of the propositional operators in the course of the development, but the details of their implementation are not important here.

Quantifiers are defined using the higher-order logic. If a term T codes a proposition ϕ (has value `true` or `false` depending on whether ϕ is true or false) then the term $(\lambda x.T) = (\lambda x.\texttt{true})$ codes $(\forall x.\phi)$. If stratified abstraction is to be used to code quantification, then one immediately has the restriction that only stratified propositions ϕ can be coded. Theory suggests that this is not too onerous a restriction: in practice, most reasoning in *NFU* corresponds to reasoning in type theory, and it is known that any stratified theorem of an extension of *NFU* with stratified axioms has a proof using only stratified formulas.

The quantifiers $\forall$ and $\exists$ are best thought of simply as functions to be applied to abstractions: whne thinking of stratification, the issues for the term $(\forall x.\phi)$ are the same as those for the more explicit term $(\lambda x.T) = (\lambda x.\texttt{true})$ (in Watson notation, `[T] = [true]`) that is not equally well covered by thinking of it as $\forall(\lambda x.T)$ (`forall @ [T]`).

In the development of quantification under Watson we will see two main themes which interact with each other: one of these is the development of a limited form of higher-order matching, to support efficiency in the application of theorems involging abstractions (usually theorems about quantification in this case-study, of course); the other is the discovery that the stratification algorithm given in the previous section, though adequate in theory, is too rigid for practical use and needs further refinement.

The abstraction functions of Watson were originally implemented using three built-in pseudo-theorems called `EVAL`, `BIND`, and `UNEVAL`. The `EVAL` pseudo-theorem is applicable to terms of the form `[T]@U` and returns `T{U/?1}`, the result of replacing `?1` (or, in a context of nested brackets, the appropriate bound variable of higher index) with `U` in `T`. `EVAL` only applies if `[T]` is a stratified abstraction term. We use braces instead of brackets in notation for substitution because of the special role of brackets in Watson notation (the braces are not Watson notation themselves); we ignore the details of the renumbering of bound variables which will occur when substitutions are carried out. The `BIND` pseudo-theorem takes a parameter: applying `BIND @ U` to a term `T` converts it to the form `[T{?1/U}]@U` (if the abstraction term `[T{?1/U}]` is stratified). The `UNEVAL` pseudo-theorem (which was introduced later than the others, as it took

us longer to realize that it was needed) also takes a parameter: applying `UNEVAL @ [T]` to `[T{U/?1}]` yields `[T] @ U` if `[T]` is stratified.

Now consider a typical quantification rule, presented equationally: The proposition $(\forall x.\phi \wedge \psi)$ is logically equivalent to $(\forall x.\phi) \wedge (\forall x.\psi)$.

One might think that this would be expressed by the equational theorem `forall @ [?P & ?Q] = forall @ [?P] & forall @ [?Q]`. This is not the case. The problem is that the free variables `?P` and `?Q` cannot match expressions that contain `?1`, because `?1` is not meaningful in the top-level context here: as a result, the given equation only applies when the formulas ϕ and ψ being coded contain no instance of x, which is certainly not very satisfactory! An equational theorem which *does* express our intention is the following: `forall @ [?P@?1 & ?Q@?1] = forall @ [?P@?1] & forall @ [?Q@?1]`, in which the possible dependence of ϕ and ψ on the bound variable is signified by representing them by expressions `?P@?1` and `?Q@?1` rather than the free variables `?P` and `?Q`.

Applying this theorem was rather laborious using early versions of the prover. To prove the rather silly theorem `forall @ [?1 = ?1 & ?1 = ?1] = forall @ [?1 = ?1] & forall @ [?1 = ?1]` using the theorem above, it was necessary first to use `BIND @ ?1` to rewrite `forall @ [?1 = ?1 & ?1 = ?1]` to `forall @ [[?2 = ?2]@?1 & [?2 = ?2]@?1]`, then rewrite with the theorem to get `forall @ [[?2 = ?2]@?1] & forall @ [[?2 = ?2]@?1]`, then rewrite with `EVAL` to get `forall @ [?1 = ?1] & forall @ [?1 = ?1]`. The theorem could not be applied directly because the terms `?P@?1` and `?Q@?1` did not match the term `?1=?1`. The solution is to allow terms `?P@?x` to match terms `T` of different surface forms, by stipulating that `?P` would match `[T{?1/?x}]` in this case, if the abstraction term is stratified. This also requires a refinement of the definition of substitution: the result of substituting `[T]` for the free variable `?P` in the context `?P @ ?x` needs to be `T{?x/?1}` rather than `[T] @ ?x`, if one is to avoid tedious applications of `EVAL`. With these changes, the rule `forall @ [?P@?1 & ?Q@?1] = forall @ [?P@?1] & forall @ [?Q@?1]` rewrites the term `forall @ [?1 = ?1 & ?1 = ?1]` to `forall @ [?1 = ?1] & forall @ [?1 = ?1]` in one step (and similarly for more realistic examples).

This is a limited form of higher-order matching. The limitation is that only local information is used to determine matching to function application terms. The effects of the limitation can be seen with the rewrite `(?P @ ?x) & forall @ [?P @ ?1] = forall @ [?P @ ?1]`. The converse of this presents no problems: if we use `forall @ [?P @ ?1] = (?P @ ?x) & forall @ [?P @ ?1]` to rewrite `forall @ [?1 = ?1]`, we get `?x_1 = ?x_1 & forall @ [?P @ ?1]` (Watson generates fresh variables when a rewrite introduces a new variable). This converse rewrite can be given a parameter: if we named it `REWRITE@?x : forall @ [?P @ ?1] = (?P @ ?x) & forall @ [?P @ ?1]` and rewrite the example term above with `REWRITE @ 3`, it will rewrite to `3 = 3 & forall @ [?P @ ?1]`. An attempt to rewrite `3 = 3 & forall @ [?P @ ?1]` with the theorem `(?P @ ?x) & forall @ [?P @ ?1] = forall @ [?P @ ?1]` will fail: the problem is that the matching function, which uses only local information to work on terms, will match `?P` to `[3=3]` when it matches `?P @ ?x` to `3 = 3` (because it has no

guidance on what to match `?x` with, and so abstracts relative to a fresh variable `?x_1`) while it correctly matches `?P` with `?1 = ?1` on the two occasions when it matches `?P @ ?1` with `?1 = ?1`, since it is given a concrete argument `?1` to match. These matches are inconsistent.

This can be avoided using the device of parameterized rewrite rules (in effect solving the higher-order matching problem for this example). To understand this approach, it is necessary to be aware that Watson supports a kind of programming with rewrite rules ([8] gives a full account). If `Rewrite` is the name of a theorem (qua rewrite rule) and `T` is a term, `Rewrite => T` is a term, denoting the same object as the term `T`. The effect of the prefixed `Rewrite` is to signal the intention to rewrite with this rule, which will be carried out by the tactic interpreter if it is invoked on such a term, or if it generates such a term in the course of another rewrite.

To solve the problem of rewriting `(?P @ ?x) & forall @ [?P @ ?1]` to `forall @ [?P @ ?1]`, given a theorem `REWRITE1: (?P @ ?x) & forall @ [?P @ ?1] = forall @ [?P @ ?1]`, first prove a theorem `REWRITE2 @ ?P: (?P @ ?x) & forall @ [?P @ ?1] = forall @ [?P @ ?1]`. Note that this "theorem" is supplied with a parameter. Then prove a theorem `REWRITE3: ?U & forall @ [?P @ ?1] = (REWRITE2 @ ?P) => ?U & forall @ [?P @ ?1]`. (This theorem is true because annotations with rewrite rules do not affect the reference of terms.)

When `REWRITE3` is applied to the term `3=3 & forall @ [?1=?1]`, the free variable `?U` matches `3=3` and the free variable `?P` matches `[?1=?1]` using higher-order matching. The term is then rewritten to `(REWRITE2 @ [?1=?1]) => 3=3 & forall @ [?1=?1]`. Now a technical detail of the implementation of parameterized rewrite rules comes into play: `REWRITE2 @ ?P` (called the "format" of the rule `REWRITE2`) is matched to the actual parameterized rewrite rule `REWRITE2 @ [?1=?1]` in the term, and the resulting substitution (of `[?1=?1]` for `?P`) is carried out on the body of the equational theorem `REWRITE2` before it is matched to its target: rewriting `(?P @ ?x) & forall @ [?P @ ?1] = forall @ [?P @ ?1]` with this substitution gives `?x_1=?x_1 & forall @ [?1=?1] = forall @ [?1=?1]`, and using this equation to rewrite `3=3 & forall @ [?1=?1]` of course gives `forall @ [?1=?1]`. The powerful move here is the ability to rewrite the body of a rewrite rule using the match of "formats" before the rewrite is used.

We are aware of very interesting work on more powerful forms of higher-order matching, as for example in [12] and [14]. We have chosen not to attempt to implement anything like this. The limited higher-order matching of Watson is efficient though stupid, and its capabilities can be enhanced as needed using "rewrite rule programming" as in the example above. It is known, for example, that full second-order matching is an NP-hard problem (see [2]).

Watson also stipulates that a term `[T] @ U` matches any term which is matched by `T{U/?1}`. This was originally seen to be needed when considering instantiation of universal hypotheses: the hypothesis `forall @ [?1=?1]`, for example, can be considered as justifying the rewrite rule `[?1=?1] = [true]`, which one would like to use to rewrite terms `V = V` to `true` regardless of the

value of `V`. This was originally done with applications of the pseudo-theorem `UNEVAL@[?1=?1]` followed by direct application of the rewrite rule and applications of `EVAL`; this refinement of matching allows a simpler implementation.

More difficulties with the implementation became evident as soon as multiple quantifiers were considered. For example, the term representing the perfectly sensible formula $(\forall x.(\exists y.x = y))$ is `forall @ [forsome @ [?1 = ?2]]`. This translation is easy: unexpectedly, this term is unstratified! The formula $(\forall x.(\exists y.x = y))$ certainly *is* stratified, but the connectives and quantifiers are not term constructors in *NFU* as they are in Watson. In *NFU*, types need only be assigned using considerations local to atomic formulas; in Watson the typing is driven by the structure of the whole term as follows:

$$\texttt{(forall}^{1}\texttt{ @ [(forsome}^{0}\texttt{ @ [(?1}^{-2}\texttt{ = ?2}^{-2}\texttt{)}^{-2}\texttt{]}^{-1}\texttt{)}^{-1}\texttt{]}^{0}\texttt{)}^{0}$$

The failure of stratification can now be seen: the variable `?1` is assigned type -2, but is also required to have type one lower than the type 0 of the abstraction term in which it is bound.

In our interpretation of *NFU* in a synthetic combinatory logic in [5] we solved this problem by noting that the relative type of a term representing a truth value can be freely raised or lowered by any constant amount without affecting its value. If T denotes a truth value, then the term $(\lambda x.T) = (\lambda x.\texttt{true})$ is an equivalent term in which the relative type of T is lowered by one, while $(\texttt{if}\,T\,\texttt{then}\,(\lambda x.\texttt{true})\texttt{else}(\lambda x.\texttt{false}))(u)$ is an equivalent term in which the type of T is raised by one. Unfortunately, it would not be a practical solution to require the user of Watson to explicitly raise the relative type of the subterm `forsome @ [?1 = ?2]` in the above example, for example by replacing it with `((forsome@[?1=?2])||[true],[false])@0` (the construction with `__||__,__` builds expressions defined by cases).

Instead, we refined the stratification algorithm of Watson to recognize terms which are "stratifiable" by such methods without having to explicitly carry out the manipulations. Moreover, the problem can be solved in more generality: that terms with propositions as values could have their types manipulated in this way is a special case of a phenomenon which is of theoretical interest in *NFU* and related theories, and which we had already predicted would have interest in applications of stratified λ-calculus (in [6]).

We present some set-theoretical background. The Cantor theorem $|A| < |\mathcal{P}(A)|$ (the cardinality of a set is strictly less than that of its power set) is not a theorem of *NFU*. This is not surprising: $|A| < |\mathcal{P}(A)|$ is not well-typed in Russell's type theory. Define $\mathcal{P}_1(A)$ as the set of all one-element subsets of A, and one can recast the theorem as $|\mathcal{P}_1(A)| < |\mathcal{P}(A)|$. This is well-typed, and is a theorem of type theory and of *NFU*, the correct analogue of the standard Cantor theorem. Of course the standard theorem is obviously false in a theory with a universal set V: $|V| < |\mathcal{P}(V)|$ is absurd. The correct assertion is $|\mathcal{P}_1(V)| < |\mathcal{P}(V)|$: the collection $\mathcal{P}_1(V)$ of all one-element sets is smaller than the collection $\mathcal{P}(V)$ of all sets (which is obviously no larger than the universal set V, and is smaller in all known models of *NFU*). Thus the "obvious bijec-

tion" $\{\langle x, \{x\}\rangle \mid x = x\}$ between $\mathcal{P}_1(V)$ and V (the singleton map) cannot be a function (note that its definition is not stratified).

A set A satisfying the unstratified equation $|A| = |\mathcal{P}_1(A)|$ (A is the same size as the set of its one-element subsets), is said to be *cantorian*: cantorian sets satisfy the standard form of Cantor's theorem. We say that A is *strongly cantorian*, which we abbreviate hereinafter *s.c.*, just in case $\{\langle x, \{x\}\rangle \mid x \in A\}$ is a set. Strongly cantorian sets have very convenient properties in relation to stratification. Let A be a strongly cantorian set, and let k be the function $\{\langle x, \{x\}\rangle \mid x \in A\}$ and k^{-1} be its inverse function. Suppose that a is a variable restricted to A. Then occurrences of a can have their relative type freely raised and lowered: replacing a with the equivalent term $(\iota x.x \in k(a))$ ("the element of $\{a\}$") raises its type by one; replacing a with the equivalent term $k^{-1}(\{a\})$ lowers the type of a by one. The types of occurrences of a variable restricted to a strongly cantorian set can be independently adjusted to restore stratification; the relative types of such variables can be ignored (it is not the case that the internal typing of any *term* belonging to an s.c. set can be ignored; the form of such a term may force relationships between the types of unrestricted variables).

The class of strongly cantorian sets is closed under all natural type constructions, including the formation of subtypes, cartesian products, and arrow types. *NFU* does not prove that any sets other than concrete finite sets are s.c., but the assumption that the set of natural numbers is s.c. is consistent with *NFU* (see [11]) and implies that all data types relevant to computer science will be s.c. sets. We suggested in [6] that the notion "strongly cantorian set" might be analogous to "data type" in an implementation of the theory of computer programming using stratified λ-calculus; practical experience with Watson bears this out.

Our course of action in response to the problem first encountered with nested quantification was to liberalize the stratification algorithm by allowing the prover to recognize and exploit the fact that certain terms represent elements of s.c. sets. We had already considered the need to deal with terms representing objects of particular sorts and with operations on objects of particular sorts in a system like Watson whose logic is unsorted. The solution adopted was to use *retractions* (functions f such that $f(f(x)) = f(x)$) as "type labels". For example, theories of propositional logic included a unary operator `|-`, a retraction onto truth values, mapping `true` to itself and everything else to `false`. The theorem `(?x = ?y) = (|- ?x = ?y)` expressed the fact that equations are propositions; the fact that conjunction was a binary operation on booleans with a boolean value was expressed by the theorem `?x & ?y = |-((|- ?x) & (|- ?y))`. One application of "rewrite rule programming" in Watson, even before the developments now outlined, was the automatic introduction or elimination of such "type labels" as needed.

The new doctrine added at this point was that "data types" should be strongly cantorian. A primitive operator `:` was chosen to represent the application of a retraction with s.c. domain to its argument, with the predeclared axiom `?t : (?t : ?x) = ?t : ?x` providing that "type labels" are retractions, and with a special role in the stratification algorithm, exploiting the strongly

cantorian character of its range. The example term `forall @ [forsome @ [?1 = ?2]]` became stratifiable with the insertion of a type label thus: `forall @ [bool: forsome @ [?1 = ?2]]`.

We did not want to always have to insert explicit type labels to restore stratification (though even this was a great improvement over the original situation). So Watson was given the ability to recognize operators or functions with strongly cantorian "input" or "output". For example the term `?x = ?y` has "strongly cantorian output": the value of this term is s.c., so the relative type of a subterm of this shape can be raised or lowered freely. The term `?x & ?y` has the stronger property of "strongly cantorian input": both of its arguments are "restricted" to s.c. sets (non-boolean values of the arguments are retracted into the boolean type before the conjunction is carried out), and so the relative types of subterms `T` and `U` in a subterm `T & U` can be raised or lowered independently of one another. The prover requires the user to exhibit a theorem of an appropriate form in order to register an operator as having one of these properties. Once these features are installed, the term `forall @ [forsome @ [?1 = ?2]]` is recognized as stratified without any need for explicit type annotation after the user registers `forsome` as having "strongly cantorian output" using the theorem `forsome @ ?P = bool: forsome @ ?P` as witness.

The extended stratification algorithm uses a richer data type to represent relative types (which are simply integers in the original algorithm). Types are either exact integer values n or "floating" types $A + n$ where A is a variable representing an unknown displacement of the type. Stratifications are either finite functions from natural numbers (qua indices of bound variables) to relative types or the error value, as before.

The `merge` algorithm can sometimes return a non-error value when the union of its finite function arguments is not a function: it will attempt to convert the union to a function by making appropriate assignments to the variables representing unknown displacements: for example, if the same variable is assigned types $A + 2$ and 3 by a stratification, the variable A will be assigned the value 1 ($m + 1$ will replace $A + m$ in the range of the union of the functions); if the conflict is between values $A + 3$ and $B + 5$, either $A + (m - 2)$ will replace all values $B + m$ or $B + (m + 2)$ will replace all values $A + m$.

We describe the stratifications of terms `T : U` with explicit type labels (the handling of appropriate subterms of infix terms built with operators registered as having strongly cantorian input or output is similar: these can be understood as having implicit type labels): the stratification of `T : U` will be `Error` if the stratification of `T` is anything but the empty function (this is satisfied if `T` is constant, but also permits limited forms of dependence of `T` on variables), and will otherwise be obtained from the stratification of `U` in the following way: the basic idea is that all constant types n in the stratification of `U` are replaced by types $C + n$ where C is a fresh variable; but if there is just one variable with an exact type in `U` the typing of that variable is simply omitted (no information is conveyed by a relative type $C + n$ unless some other variable has a type $C + m$, allowing us to determine the exact displacement between two types).

The full stratification algorithm of *NFU* for terms translating formulas is recovered, once all logical operations have been declared as having strongly cantorian input and the equality and membership predicates have been declared as having strongly cantorian output. This approach solves problems not only in the implementation of quantification in higher order logic, but in higher-order logic proper: for example, in a theory of program semantics we were able to construct function types while essentially ignoring stratification restrictions, the syntactic price being the need to occasionally attach type labels to bound variables. The type of states $S = A \to V$ in our theory was inhabited by functions from a type of addresses A to a type of values V; the type of expressions in the theory was $(S \to V) = ((A \to V) \to V)$. Early attempts to develop this theory were impeded by stratification failures (note that V appears at two different relative types in this type definition), but making the base types A and V strongly cantorian and proving that important operations have strongly cantorian input and/or output allows reasoning about these types in a natural way.

The final problem we consider, which combines the issues of higher-order matching and stratification, is the treatment of higher-order matching where more than one variable is involved.

The canonical problem along these lines is the implementation of the logical equivalence $(\forall x.(\forall y.\phi)) \equiv (\forall y.(\forall x.\phi))$. One might suppose by analogy with the handling of single quantifiers above that this would be presented as `forall @ [forall @ [(?P @ ?1) @ ?2]] = forall @ [forall @ [(?P @ ?2) @ ?1]]`, with the intention that the term `(?P @ ?1) @ ?2` would match arbitrary terms `T` coding ϕ.

This approach doesn't work. One problem is that `forall @ [forall @ [(?P @ ?2) @ ?1]]` isn't stratified: even with the improvements to stratification described above, the addition of a type label (giving `forall @ [forall @ [bool: (?P @ ?2) @ ?1]]`; equally good is
`forall @ [forall @ [|- (?P @ ?2) @ ?1]]`, where `|-` is the proposition label described above, if it is registered as having strongly cantorian output) is needed for it to be possible to sensibly type `?2`.

A more profound problem is that `(?P @ ?1) @ ?2` isn't suitable to match general Watson expressions: its typing is rigid, dictating that `?1` has type one higher than `?2`. This typing problem is the reason why the technique of "currying" usually used in combinatory logic and λ-calculus for representing functions of several variables (using $f(x)(y)$ to represent the application of a function to two arguments x and y and regarding $(\lambda xy.T)$ as meaning $(\lambda x.(\lambda y.T))$) is not appropriate in Watson: in stratified λ-calculus we write the generic application of a function of two variables as $f(x, y)$ (the usual abuse of notation for $f(\langle x, y\rangle)$), and indeed write application terms in the traditional form $f(x)$ rather than in the form (fx) with left grouping which is so convenient for currying; moreover, the defining equation for our $(\lambda xy.T)$ is $(\lambda xy.T)(x, y) = T$.

The typical situation is that in which `?1` and `?2` have the same relative type. In this case the theorem of quantifier switching can be implemented as `forall @ [forall @ [?P @ (?1,?2)]] = forall @ [forall @ [?P @ (?2,?1)]]`.

If one matches `?P @ T,U` to a term `V`, the variable `?P` will match `[T{p1@?1;p2@?1/U;V}]`, if this is stratified: for example, `?P @ (?x,?y)` will match `?y + ?x`, with `?P` matching the function `[p2@?1 + p1@?1]`. The projection functions `p1` and `p2` are automatically applied in the course of substitution so that laborious eliminations of automatically generated projections are not needed. With these features, rewriting `forall@[forall@[?1+?2 = ?2+?1]]` with the quantifier switching theorem of this paragraph gives `forall@[forall@[?2+?1 = ?1+?2]]` in one step.

It is possible to prove a sequence of different quantifier switching theorems which will work for each possible value of the difference in types between `?1` and `?2` while preserving complete respect for stratification, though the forms of higher-order patterns used to match terms with various differences in type are rather esoteric. This is not really an ideal situation, and it is in fact completely solved in the current version of Watson. The general solution goes beyond the scope of this paper (it is more fully described in [9]): unstratified abstraction terms are admitted into the language, and a special function application operator `@!` is admitted with the completely general rule `[T] @! U = T{U/?1}` for all abstraction terms `[T]` which Watson accepts as well-formed (which includes all terms built without use of the special application operator `@!` and a limited class of terms built with applications of that operator which are "inessential" in a suitable sense). This amounts to augmenting our logic with a theory of "proper class functions", and is obviously logically dangerous; the extension is formally described and shown to be safe in [9]. The problem of theorems like quantifier switching can then be solved by using curried terms in `@!` as higher order patterns: `forall @ [forall @ [(?P @! ?1) @! ?2]] = forall @ [forall @ [(?P @! ?2) @! ?1]]` turns out to be an all-purpose quantifier switching theorem.

This extension of the logic has the further effect that the expressive power of the Watson logic becomes the same as that of *NFU*: the ability to represent and manipulate unstratified abstraction terms has the effect of allowing the expression of unstratified quantification and standard first-order reasoning about unstratified formulas. There are very interesting unstratified axioms which are useful extensions of *NFU* (see [10] or [3]); one of them (the assertion that the natural numbers are s.c.) is an official part of the logic). Some of these are expressible under Watson without unstratified abstraction (because the implicit quantification over free variables in Watson theorems can be unstratified), but the full range of unstratified axioms (and some things which are formally infinite axiom schemes) become expressible in the full Watson logic.

5 Relation to Earlier Work

This paper tries to give a more accessible account of the underlying logic (as a set theory) than was given in [7].

A defect of our survey [9] of the logic of Watson noted by a referee was that in its attempt to give an account of the *entire* logic of Watson (including

its equational/rewriting aspects, its logic of case expressions, and the implementation of first-order logic using unstratified abstraction) the account of the higher-order logic suffered; here we try to give more of the flavor of the work with the higher-order logic.

We rely on earlier mathematical work on the implementation of the logic of quantification in an equational higher-order logic to the extent that we never discuss the details of how this is done at all! The details may be seen, for example, in [1] or [4]. It should be noted that the implementation of quantification is here used as a case study of the use of the higher-order logic; it is now possible to implement first-order logic in the Watson logic without any attention to stratification at all, but the flexibility added to the type system in the course of this development remains critical for reasoning about functions (as in the example of a theory of program semantics briefly mentioned above).

References

1. Peter Andrews, *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof*, Academic Press, Orlando, 1986.
2. H. Comon and Y. Jurski, "Higher-order matching and tree automata" in *Proc. Annual Conf. of the European Assoc. for Computer Science Logic, Aarhus, Denmark, Aug. 1997*, LNCS vol 1414, Springer-Verlag 1998, pp. 157-176.
3. T. E. Forster, *Set theory with a universal set, an exploration of an untyped universe*, 2nd. ed., Oxford logic guides, no. 31, OUP, 1995.
4. David Gries and Fred B. Schneider, *A Logical Approach to Discrete Math*, Springer-Verlag, 1993.
5. M. Randall Holmes, "Systems of combinatory logic related to Quine's 'New Foundations' ", *Annals of Pure and Applied Logic*, 53 (1991), pp. 103-33.
6. M. Randall Holmes, "The set theoretical program of Quine succeeded (but nobody noticed)", *Modern Logic*, vol. 4 (1994), pp. 1-47.
7. M. Randall Holmes, "Untyped λ-calculus with relative typing", in *Typed Lambda-Calculi and Applications* (proceedings of TLCA '95), Springer, 1995, pp. 235-48.
8. M. Randall Holmes, "The Watson Theorem Prover", to appear in the *Journal of Automated Reasoning*. The on-line documentation of the prover accessible from `http://math.boisestate.edu/~holmes/proverpage.html` incorporates an early draft of this paper.
9. M. Randall Holmes, "A strong and mechanizable grand logic", in *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000"*, *Lecture Notes in Computer Science*, vol. 1869, Springer-Verlag, 2000.
10. M. Randall Holmes, *Elementary Set Theory with a Universal Set* (Cahiers du Centre de Logique, vol. 10), Academia-Bruylant, Louvain-la-Neuve, 1998.
11. Ronald Bjorn Jensen, "On the consistency of a slight (?) modification of Quine's 'New Foundations' ", *Synthese*, 19 (1969), pp. 250-63.
12. Dale Miller, "Unification under a mixed prefix", *J. Symbolic Computation*, vol. 11 (1992), pp. 1-38.
13. W. V. O. Quine, "New Foundations for Mathematical Logic", *American Mathematical Monthly*, 44 (1937), pp. 70-80.
14. Jan Springintveld, "Third-order matching in the polymorphic lambda-calculus", in G. Dowek, J. Heering, K. Meinke, and B. Möller, eds. *Higher-Order Algebra, Logic, and Term Rewriting*, LNCS vol. 1074, Springer-Verlag, 1995.