A Strong and Mechanizable Grand Logic*

M. Randall Holmes

Boise State University

Abstract. The purpose of this paper is to describe a "grand logic", that is, a system of higher order logic capable of use as a general purpose foundation for mathematics. This logic has developed as the logic of a theorem proving system which has had a number of names in its career (EFTTP, Mark2, and currently Watson), and the suitability of this logic for computer-assisted formal proof is an aspect which will be considered, though not thoroughly. A distinguishing feature of this system is its relationship to Quine's set theory *NF* and related untyped λ-calculi studied by the author.

1 Introduction

The theory we develop here will be referred to as `W`, after the current name "Watson" of the theorem prover in which it is implemented (for a more through discussion of this prover see [8]). The notation of the system will be presented just as it is presented to (and by) the theorem prover.

The roots of this logical system are in Quine's set theory "New Foundations" (*NF*) of [10], but it cannot be described simply as an implementation of *NF*. *NF* is not known to be consistent; the grand logic presented here is (partly) based on the variation *NFU* of *NF* presented by Jensen in [9], which is known to be consistent and suitable for applications (see [7] for a development). *NF* and *NFU* are set theories; `W` is an untyped λ-calculus. *NF* and *NFU* are usually presented using standard first-order logic; this system interprets the notions of propositional and predicate logic in terms of its own rather different primitives.

2 Syntax

The formal theory `W` presented here is an equational theory. All statements of `W` are equations between terms (intended for use as rewrite rules) and the focus of the theory is on the structure of terms rather than on the structure of propositions. Terms representing truth values stand in for propositions, and the usual notions of propositional and predicate logic are expressed as operations on terms representing truth values.

This section is devoted to the syntax of the language of `W`. First of all, if `A` and `B` are terms, `A = B` is an equation (as statement of `W`); but the `=` operator

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

also occurs as a term constructor with the natural meaning (`A = B` is a term which is equal to `true` if `A = B` holds and equal to `false` if `A = B` is false). The overloading of `=` should always be easily disambiguated in what follows.

Any string of positive length consisting of characters taken from the sets of letters, digits, and the special characters `?` and `_` is an atomic term.

Atomic terms are of four kinds:

numerals: Any atomic term consisting only of digits is a numeral. (This category may be regarded as subsumed under "constants" below: it is not of special logical interest).

bound variables: An atomic term consisting of `?` followed by a non-zero-initial numeral is a bound variable.

free variables: An atomic term beginning with `?` and containing another non-numeric character is a free variable.

constants: An atomic term not beginning with `?` and containing a non-numeric character is a constant.

Before constructions of composite terms are introduced, a preliminary discussion of kinds of operator is needed.

operators: A string of special characters (not listed, but excluding all characters found in atomic constants and excluding paired forms such as quotes, braces, brackets, and parentheses) is an operator. In particular, `@` and `@!` are operators representing two different kinds of function application, and `,` is the ordered pair constructor. Also, a string of alphanumerics preceded by a backquote `` ` `` is an operator.

It is important to note that an operator is not itself a term. We oversimplify by stipulating that each operator is either prefix or infix, but not both (there is some overloading in the prover). (Operators are declared infix or prefix in particular theories.)

We now present the constructions of composite terms.

prefix terms: A prefix term consists of a (prefix) operator followed by a term.

abstraction terms: An abstraction term (a function) consists of a term enclosed in brackets. (Abstraction terms implement λ-terms, and standard λ-notation will sometimes be used).

parenthesized term: A term enclosed in parentheses.

infix terms: An infix term consists of a atomic, abstraction, or parenthesized term, followed by an (infix) operator, followed by a term.

case expressions: A case expression consists of a parenthesized term, followed by `||`, followed by a parenthesized term, followed by `,`, followed by a term. The special operator `||` may only occur in terms of this form.

reduction of parentheses: Parentheses around an atomic term or abstraction term may always be removed. Parentheses around an infix term or case expression may be removed except when it is the left subterm of an infix term or one of the two leftmost subterms of a case expression. If a term is obtained from another term by reduction of parentheses (or by the addition of parentheses for clarity), it is regarded as being the same term.

completeness of description: The class of terms is the intersection of all sets containing all atomic terms and closed under the term constructions given above.

This description of the syntax is based on the default order of precedence of the Watson prover, in which all operators have the same precedence and group to the right.

3 Equational Logic

The bedrock of the logic of `W` is equational logic. All statements in the language of `W` are equations, understood to be implicitly universally quantified over the free variables occurring in them. All free variables are untyped, with an exception described below (in the discussion of class abstraction).

In this section we restrict ourselves to the sublanguage of the language of `W` which excludes bound variables. (Abstraction terms without bound variables may occur; these represent constant functions.)

We define substitution for the restricted language without bound variables: if `A`, `T` are terms and `?x` is any variable, we define `A{T/?x}` as the result of replacing all occurrences of the variable `?x` with `(T)`. (Of course, the parentheses may then often be reduced away).

The basic rules of the equational logic of `W` are as follows:

reflexivity: For any term `A`, `A = A` is a theorem.
symmetry: If `A = B` is a theorem, then `B = A` is a theorem.
transitivity: If `A = B` is a theorem and `B = C` is a theorem, then `A = C` is a theorem.
localization: If `A = B` is a theorem and `C` is a term then `C{A/?x} = C{B/?x}` is a theorem. (`?x` being any free variable).
specification: If `A = B` is a theorem and `C` is a term then `A{C/?x} = B{C/?x}` is a theorem. (`?x` being any free variable).

These rules will need to be refined when bound variables are introduced.

4 The Logic of Terms Defined by Cases

We now consider the first part of the grand logic `W`, corresponding to propositional logic and the logic of identity.

We introduce the predeclared constants `true` and `false`, representing the truth values.

In a case expression `T || U , V`, we refer to the subterm `T` as the *hypothesis* of the case expression and to the subterms `U` and `V` as its *branches*. The intended meaning of the term `(T = U) || V , W` is "if `T = U` then `V` else `W`"; when `T` is not an equation, `T || U, V` is intended to have the same meaning as `(true = T) || U , V`.

We introduce axioms governing the behavior of the special term construction of "case expressions".

The basic axioms are the following:

P1: `((?x = ?x) || ?y , ?z) = ?y`
P2: `((true = false) || ?y , ?z) = ?w`
HYP: `((?a = ?b) || (A{?a/?x}) , B) =`
`((?a = ?b) || (A{?b/?x}) , B)`
DIST: `(A{((?a = ?b) || ?c , ?d)/?x}) =`
`(?a = ?b) || A{?c/?x} , A{?d/?x}`

The axioms **P1** and **P2** implement special cases of our preformal understanding that a case expression will be equal to its first branch when the hypothesis is true and to its second branch when the hypothesis is false. In an expression `(A = B) || T , U`, it should be clear that we can freely replace `A` with `B` or vice versa in the context `T` without affecting the value of the term: this is captured by the axiom **HYP**. The name is taken from the idea that this implements "reasoning under hypotheses". The axiom **DIST** allows the "distribution" of the hypothesis of a subterm over a larger context.

It should be noted that **HYP** and **DIST** are axiom schemes rather than single axioms (thanks to a referee for pointing out that I needed to say this!) They could in principle be replaced in almost all applications by single axioms of the form

*__HYP:__ `((?a = ?b) || (?A @! ?a) , ?B) =`
`((?a = ?b) || (?A @! ?b) , ?B)`
*__DIST:__ `(?A @! ((?a = ?b) || ?c , ?d) =`
`(?a = ?b) || (?A @! ?c) , (?A @! ?d)`

where (as noted above) `@!` is a function application operator. In earlier versions of the prover, axioms of the latter forms were actually used; applying such axioms to get each instance of the full schemes involved λ-abstraction and β-reduction. In the current version of the prover, there is built-in support for the application of the schemes, not involving any use of the function machinery of the prover, so it seems more natural to present the schemes.

The axiom set actually built into Watson is slightly larger, but we want to emphasize the extreme simplicity of the logic of case expressions presented here. The additional content provided by Watson can be presented as the pair of axioms:

EQ: `(?a = ?b) = (?a = ?b) || true , false`
GH: `((true = ?x) || ?y , ?z) = ?x || ?y , ?z`

These can be regarded as providing implicit definitions of terms with the operator `=` and of case expressions with hypotheses which are not equations. Though these are useful constructions, it is worth noting that they are not an essential part of the underlying theory.

The following propositions are easy consequences of the six axioms given so far:

E1: `(?a = ?a) = true`
E2: `(true = false) = false`
B1: `(true || ?x , ?y) = ?x`
B2: `(false || ?x , ?y) = ?y`

Watson has **E1** and **B1-2** as built-in assumptions instead of **P1-2**; **E2** is provable from these and **EQ**, **GH**, as are **P1-2**.

The axiom `HYP` allows substitutions to be made in the left branch of the hypothesis under the locally valid assumption that the hypothesis is true; it may seem that we have neglected similar things one can do in the right branch using the assumption that the hypothesis is false, but this is not the case! We present three theorems, the last of which captures the use of negative hypotheses:

T0: `((?a = ?b) || ?x , ?x) = ?x`
T1: `((?a = ?b) || A{((?a = ?b) || ?x, ?y)/?v} , ?z) =`
`(?a = ?b) || A{?x/?v} , ?z`
T2: `((?a = ?b) || ?x, A{((?a = ?b) || ?y , ?z)/?v}) =`
`(?a = ?b) || ?x , A{?z/?v}`

While the axiom **HYP** allows us to rewrite only in the left branch of a case expression, the theorems **T1** and **T2** allow rewriting of into both the left and right branches of case expressions, though of a more restricted nature.

We omit the easy proofs of **T0** and **T1**. We do prove **T2**.

```
(?a = ?b) || ?x, A{((?a = ?b) || ?y , ?z)/?v} = (EQ)

((?a = ?b) || true , false) || ?x ,
   A{(((?a = ?b) || true , false) || ?y , ?z)/?v} = (substitution)

(?u || ?x , A{(?u || ?y , ?z)/?v})
   {((?a = ?b) || true , false)/?u} = (DIST)

(?a = ?b) ||
   ((?u || ?x , A{(?u || ?y , ?z)/?v}){true/?u}),
   (?u || ?x , A{(?u || ?y , ?z)/?v}){false/?u} = (substitution)

(?a = ?b) ||
   (true || ?x , A{(true || ?y , ?z)/?v}) ,
   (false || ?x , A{(false || ?y , ?z)/?v}) = (B1 and B2)

(?a = ?b) || ?x , A{(false || ?y , ?z)/?v} = (B2)

(?a = ?b) || ?x , A{?z/?v}
```

which completes the proof of **T2**.

Though we regard the formulation using **DIST** and **HYP** as more mathematically elegant, it should be noted that taking **HYP** and **T0-2** as primitive

assumptions is equivalent, and this is the axiomatization which is effectively hard-wired into the prover. We omit the short proof of **DIST** from **T0-2**.

Propositional connectives are readily defined using expressions defined by cases. We give only the definition of negation as an example.

Definition: $\sim$`T` is defined as `T || false, true`

It is worth remarking that in case `T` is neither `true` nor `false`, this definition treats it in the same way as `false` (and so $\sim$`T` is equal to `true` in this case). Similar considerations apply to the other propositional connectives.

We now prove a completeness theorem for the logic of case expressions in its intended interpretation.

A theory in the language of `W` is an set of equations between terms in the language of `W` (with some fixed set of constants and constant operators), closed under the application of the rules of equational logic (reflexivity, symmetry, transitivity, localization and specification).

Definition: If M is a set with more than one element, an environment for M relative to a theory is a map from the free variables of the language of `W` to elements of M.

An interpretation of the theory M is a map which takes any pair consisting of an environment for M and a term to an element of M.

An interpretation I is said to be sound for a theory if the following conditions hold:

1. If σ is an environment and `v` is a free variable, then $I(\sigma, v) = \sigma(v)$.
2. If t and u are terms such that $t = u$ is an equation in the theory, and σ is any environment, then $I(\sigma, t) = I(\sigma, u)$.
3. $I(\sigma, \texttt{true}) \neq I(\sigma, \texttt{false})$ for any σ.
4. If t is a term containing no free variables, then $I(\sigma, t) = I(\sigma', t)$ for any environments σ and σ'.

We now prove a

Completeness Theorem: Any theory which does not have `true = false` as an element has a sound interpretation in a set M which is at most countably infinite.

Proof: We construct an interpretation whose range is the set of equivalence classes of variable-free terms of the language of the theory under a suitable equivalence relation.

Terms `T` and `U` will be equivalent if they are provably equal in the theory. This is not sufficient to define the desired equivalence relation, because the theory may not be complete (it may not allow the decision of all equations). To define the complete theory, enumerate all equations between variable-free terms in its language. Consider the first equation `T = U` on this list with the property that neither `T = U` nor `((T = U)||true,false) = false` is a theorem. It is straightforward to establish that `T = U` is a theorem iff `((T = U) || true , false) = true` is a theorem, and so that `((T =`

`U)||true,false) = false` cannot both be theorems (because symmetry and transitivity of equality would force `true = false` to be a theorem, contrary to hypothesis).
We claim that adding `T = U` to the theory and closing under the application of the rules of equational logic will still produce a consistent theory (`true = false` will not belong to the extended theory). Suppose otherwise: then we would have a proof

```
true = V1 = ... = Vn = false
```

with each step justified by an element of our theory or the equation `T = U` (possibly combined with an application of the rule of localization).
We could then modify this proof to the following form:

```
(T=U)||true,false =
(T=U)||V1,false =
...
(T=U)||Vn,false =
(T=U)||false,false = (T0)
false
```

Each step of this proof would be valid in the original theory: steps using equations of the original theory obviously remain valid and the steps using `T = U` would be justified by applications of the axiom **HYP** of the logic of case expressions. So the original theory would prove `((T=U)||true,false) = false`, which we have seen is impossible.
We can then repeat this process to obtain a complete theory (one which decides every equation). The resulting complete theory allows us to define a sound interpretation I in the set of equivalence classes of variable-free terms of the language; $I(\sigma, t)$ will be the equivalence class of the term obtained from the term t by replacing each free variable v occurring in t with some element of the equivalence class of terms $\sigma(v)$. Since the language itself is no more than countably infinite, any partition of the set of terms of the language is likewise no more than countably infinite. The proof is complete.

It follows from the Completeness Theorem and the definability of notions of propositional logic and the logic of identity in the logic of case expressions given here that this logic codes all valid reasoning in propositional logic and the logic of identity (as claimed above). Complete implementations of propositional logic reasoning in several styles have been made in Watson, using the principles described here.

The definition of another version of this logic of case expressions and the proof of its completeness are found in our unpublished [4].

We are well aware that the use of an `if`...`then`...`else`... construction as a primitive in the definition of logical connectives is not novel. We have not seen it

widely advertised that the four basic axioms (which we repeat here in standard notation for emphasis) together with rules of equational logic are sufficient to provide a basis for propositional logic and the logic of identity in an untyped context.

P1: $(\texttt{if}\, x = x\, \texttt{then}\, y\, \texttt{else}\, z) = y$
P2: $(\texttt{if}\, \texttt{true} = \texttt{false}\, \texttt{then}\, y\, \texttt{else}\, z) = z$
HYP: $(\texttt{if}\, a = b\, \texttt{then}\, F(a)\, \texttt{else}\, c) = (\texttt{if}\, a = b\, \texttt{then}\, F(b)\, \texttt{else}\, c)$
DIST: $F(\texttt{if}\, a = b\, \texttt{then}\, c\, \texttt{else}\, d) = (\texttt{if}\, a = b\, \texttt{then}\, F(c)\, \texttt{else}\, F(d))$

5 Bound Variables and Substitution

We introduce the notation for variable binding used in the prover, which is a system of the sort introduced by de Bruijn (in [2]) with "nameless dummies", though it is not the usual scheme of "de Bruijn indices". We also introduce the formal definition of substitution for this system and extend the rules of equational logic to the language as extended with bound variables.

The construction of functions is the only variable binding construction in the logic of Watson. There are two different kinds of function application, set function application, represented by the infix operator `@`, and class function application, represented by the infix operator `@!`. The same variable binding construction builds both set and class functions; there is a syntactical constraint on permitted occurrences of `@!` in functions, but no corresponding restriction on permitted occurrences of `@`. The application of the β-reduction rule is more restricted for the `@` operator than for `@!`, as will be discussed in the next section.

We recall that an atomic term consisting of `?` followed by a non-zero-initial numeral is a bound variable.

An occurrence of a term in Watson is said to have "level n" if it occurs as a subterm of n abstraction terms (if it is enclosed in n pairs of brackets, on a typographical level). The bound variable `?n` cannot occur sensibly at a level lower than n (where the two occurrences of "n" in different type faces represent a positive integer and its numeral). The intended semantics is that an abstraction term `[T]` occurring at level $n-1$ represents a λ-term (a function) in which the bound variable is `?n`: so for example the term `[?1]` at level 0 stands for the function $(\lambda x.x)$: the term `[[?1]]` (at level 0) is $(\lambda x.(\lambda y.x))$, the map which sends x to the constant function of x (the K combinator) while `[[?2]]` is $(\lambda x.(\lambda y.y))$, the constant function whose value is the identity function. The term `[[?3]]` has no semantics at level 0 except as a subterm of a larger term; we cannot see the bracket that binds the bound variable `?3`.

We formally qualify the notion of term to facilitate discussion of subterms of nontrivial level: a "level n term" is a term in which each bound variable `?m` occurs enclosed in at least $m-n$ brackets. Note that any level n term is also a term of level m for each $m > n$, though the semantics of typographically identical terms may be different at different levels. Level 0 terms are the terms which have sensible semantics in a top-level term; level n terms are those terms

which can appear in a level 0 term inside n enclosing brackets. In a term being considered as a level n term, we will speak of subterms enclosed in m brackets as occurring at level $m + n$ (tacitly assuming that there are n more brackets somewhere in a larger context).

This scheme is closer to the usual variable binding scheme than the familiar scheme of "deBruijn indices" (we have seen the scheme we use referred to as "deBruijn levels"): a term in the usual λ-calculus can be converted to this form by renaming the outermost bound variables to `?1`, the next-to-outermost bound variables to `?2`, etc., then replacing all the binders ($\lambda x. \ldots$) with brackets. An advantage of this scheme over deBruijn indices is that instances of the same bound variable always look the same; a disadvantage is that terms with bound variables will have to have the bound variables renumbered when they are substituted into a context at a different level.

Practical experience with using this system suggests that as long as brackets are not too deeply nested the notation is intelligible. In the current Watson theory package, an operator `.` is provided with the defining axiom `(?x.?y) = ?y` (ignore the first argument); a tactic is provided which converts every bracket term `[T]` in the current context to the form `[?n.T]`, where `?n` is the appropriate bound variable. There is a converse tactic to get rid of such annotations. The development of a tactic of this kind under Watson is easy, and it restores the advantages of the usual variable binding notation with a binder at the head of the term (if one doesn't mind having the names of one's bound variables chosen for one). Note that the introduction and removal of such annotations is automated; if a user introduces incorrect annotations by hand, they are easily checked and corrected; it is exactly the fact that the semantics of the annotations are trivial which makes it possible for prover tactics (which are not allowed to change a term in a way which affects its reference) to correct the annotations where necessary.

On a formal level, the introduction of variable binding requires a change in the definition of substitution and an extension of the rules of equational logic.

If `A` and `B` are terms of the same level n and `?x` is a free variable, we define a term `A{B/?x}` of the same level n. Where m and n are numerals with $m > n$ and `B` is a level n term, we define `B{m/n}` as the term which results when each bound variable `?i` in `B` with index $i > n$ is replaced by `?j` with index $j = i + m - n$. (A variable which is bound by a bracket in `B` (a `?i` with $i > n$) will no longer be associated with the correct bracket if the term `B` is substituted into a context enclosed in m brackets instead of n brackets: it will need to have its numbering shifted by $m - n$.) The refinement in the definition of `A{B/?x}` is that each occurrence of `?x` needs to be replaced with `(B{m/n})` rather than simply `(B)`, where m is the level of that occurrence of `?x`.

The definition of `A{B/?x}` can be extended to the case where the level n of `B` is greater than the level l of A, just in case no occurrence of the variable `?x` in `A` is at a level less than n. The form of the definition is exactly the same in this case; this extension is needed for the formalization of the rule of localization.

We present extended axioms of equational logic for `W`, defining notions of theorem at each level n. Of course, our true theorems are the level 0 theorems.

reflexivity: For any level n term `A`, `A = A` is a level n theorem.
symmetry: If `A = B` is a level n theorem, then `B = A` is a level n theorem.
transitivity: If `A = B` is a level n theorem and `B = C` is a level n theorem, then `A = C` is a level n theorem.
localization: If `A = B` is a level n theorem and `C` is a level m term in which all occurrences of `?x` are at level n or higher (so $n \geq m$) then `C{A/?x} = C{B/?x}` is a level m theorem. (`?x` being any free variable). (notice that the definition of substitution handles any needed renumbering of bound variables in the equation `A = B` that is applied).
specification: If `A = B` is a level n theorem and `C` is a level n term then `A{C/?x} = B{C/?x}` is a level n theorem. (`?x` being any free variable).
level conversion: If `A = B` is a level n theorem and $m > n$, then `A{m/n} = B{m/n}` is a level m theorem.

The harmonization of this system with the logic of case expressions given above amounts to recognizing that the new definition of substitution needs to be used. There is no essential change in the proof of completeness; it goes the same way mod renumbering of bound variables in the equation `T = U` mentioned in that proof when used at different levels.

We now develop the defining axiom of the class map application operator `@!`. Where `?n` is a bound variable, `T` is a level n term and U is a level $n-1$ term, we define `T{U/?n}` as the level $n-1$ term which results if all occurrences of `?n` in `T` are replaced by `(U)` and all occurrences of bound variables `?i` with $i > n$ in `T` are replaced by `?j` with $j = i - 1$. We can then state the rule of β-reduction in the very natural form:

(class) β-reduction: `([T] @! U) = T{U/?n}` is a level $n-1$ theorem for each level n term `T` and level $n-1$ term `U`.

Another form in which this could be stated (without introducing new notation, but that is its only merit!) is "`([T{n/(n-1)}{?n/?x}] @! U) = T{U/?x}` is a level $n-1$ theorem for any level $n-1$ terms `T` and `U`".

So far we appear to have axiomatized untyped λ-calculus, which is incompatible with the presence of functions without fixed points, such as negation, which we can already define. Paradox is avoided by a restriction on the formation of abstraction terms containing the operator `@!`. We define the head of a term and its number of arguments as follows: if a term `T` is not of the form `U @! V`, then it is its own head and has 0 arguments; if a term `T` is of the form `U @! V`, then its head is the head of `U` and it has one more argument than `U`. We define an n-function as follows: a term not of the form `[T]` is a 0-function and a term of the form `[T]` is an $(n+1)$-function iff `T` is an n-function. The restriction on the formation of abstraction terms is that if a term with n arguments appears as a subterm of an abstraction term, its head must be either an n-function or a free variable. Notice that under this condition any heads of subterms of abstraction

terms which are n-functions for $n > 0$ can be eliminated by β-reductions. For example, this restriction forbids the formation of fixed points of arbitrary operators `F` by self-application of abstraction terms `[F @! (?1 @! ?1)]`, because `?1 @! ?1` is prevented from occurring as a subterm of an abstraction term.

6 The Theory of Class Abstraction

We have seen above that the operations of propositional logic can be interpreted in the logic of case expressions. We now use the class function machinery of Watson to interpret quantification.

The essential idea is that the universal quantifier can be interpreted as the function `forall` defined as `[?1 = [true]]` (i.e., $(\lambda x.(x = (\lambda y.\texttt{true})))$; there is nothing new about this idea!). If a formula ϕ is represented by a term `T`, then the formula $(\forall x.\phi)$ will be represented (mod technicalities about the variable binding) by `[T] = [true]` = `forall @! [T]`. If the formula ϕ is represented by a term `T`, the formula $(\exists x.\phi)$ will be represented by the term $\sim$`([T] = [false])`. So we define `forsome` as `[`$\sim$`(?1=[false])]`. (the definition of `forsome` is not quite as nice as that of `forall`, because it does not mean quite what one would like when `T` takes on values which are not truth-values). In any event, `forall @ [T]` and `forsome @ [T]` will code the intended quantified statements when `T` codes a formula (and so has boolean value).

We now demonstrate that first order logic with equality on infinite domains is captured exactly by the logic of case expressions augmented with our scheme of class functions. The precise sense in which this is true is as follows: we can take any countably infinite model of a first order theory and introduce a definition of the class function abstraction and application operations which will satisfy the formal rules of this system and under which the internal definitions of the quantifiers will succeed.

Take any first order theory T (with equality) with a finite or countably infinite set of primitive predicates, constants and function symbols and having a countably infinite model M. We indicate how to represent the machinery of class abstraction within M in such a way that the definitions of the quantifiers in terms of class abstraction succeed.

We partition the countably infinite set M into a sequence of countably infinite subsets M_i indexed by the natural numbers.

We need to translate the language of T into the language of `W`. Select two distinct elements of the model M as referents of the terms `true` and `false`. Propositions of the language of T will be interpreted as terms with values `true` or `false`. Introduce constants (in the sense of `W`) translating each constant of T. Introduce operators translating each predicate and function symbol of T (equality is translated by `=`). Unary predicates or function symbols will correspond to prefix operators, and binary predicates or function symbols will correspond to infix operators. If there are operators of ternary or higher arity, these can be accomodated by introducing the pair operator `(,)`; for example, an atomic sentence with a ternary predicate symbol like $Rxyz$ would have the translation

`x ‘R y , z`. The pair operator can represent any injection from $M \times M$ into M.

We define a class L_0 of terms in the language of `W` containing interpretations of all terms and quantifier-free propositions of the language of T. L_0 contains all free variables and translated constants of T, plus `true` and `false`. It is closed under the construction of terms with (translated) predicates and function symbols of T (plus `=` and `,`) and the construction of case expressions (used to handle propositional connectives). It is the smallest class of terms satisfying these closure conditions.

For each term `T` in L_0 which contains no free variable other than a fixed variable `?x`, construct the abstraction term `[T{1/0}{?1/?x}]` (after this point, we abbreviate this as `[T{?1/?x}]`). These abstraction terms are permitted in `W`, because no such term `T` will contain any occurrence of `@!`; this will remain true throughout the iterative construction we are about to carry out. Each such abstraction term corresponds in a natural way to a function from M to M; we assign an element of M_0 as the referent of each such abstraction term, assigning the same referent to terms which correspond to the same function. This will succeed because there are clearly no more than countably infinitely many such terms.

Our intention is now to augment our language by adding all the abstraction terms we have just constructed as new constants. We extend the language L_0 to include all abstractions over terms of L_0 (notice that this is not the same as taking the closure of L_0 under the abstraction term construction!); we call this extended language L_1 (it is clearly harmless to allow free variables in abstraction terms; the reference of an abstraction term with free variables in it will be determined once the reference of each free variable is determined). Notice that for each term `T` of L_0 which codes a proposition ϕ, we have `[T{?1/?x}] = [true]`, which codes $(\forall x.\phi)$, as a term of L_1; the language L_1 allows us to express some quantified sentences.

We then proceed in the same way through steps indexed by the natural numbers. When the language L_n has been constructed, we consider the set of all terms `T` of L_n which contain no free variable other than `?x`. We construct abstraction terms `[T{?1/?x}]` for each such term. Each such abstraction term corresponds to a function from M to M. Some of these terms will correspond to functions with the same extension as an abstraction term already defined; assign these the same referent as the term(s) with which they are coextensional. Assign to each term which has a “new” extension a referent in M_n (none of whose elements will have been used yet as referents for abstraction terms), assigning terms with the same extension the same referent. It will be possible to do this because the class of abstraction terms being considered is no more than countably infinite. We extend the language L_n with all abstraction terms `[T{?1/?x}]` for `T` a term of L_n, obtaining the language L_{n+1}; we are able to determine the reference of any term of L_{n+1} once the reference of any free variables in the term is given. Notice that if any proposition ϕ is coded by a term `T` of L_n, the proposition $(\forall x.\phi)$ will be coded by the term `[T{?1/?x}] = [true]` in L_{n+1}.

We consider the language L_ω, the union of all the languages L_n. We augment L_ω with the class application operator `@!`, using the β-reduction scheme to determine the meaning of terms `[T] @! U` and assigning a default value to each term `T @! U` where the referent of `T` is not the referent of any abstraction term. L_ω allows us to abstract freely over terms which do not contain `@!` (this should be clear from the construction); abstraction over terms with a subterm with n arguments and head an n-function makes sense because the occurrences of `@!` in such subterms can be eliminated by repeated β-reduction, obtaining an abstraction term provided by L_ω; abstraction over terms with a subterm with n arguments and head a free variable makes sense as long as we stipulate that such free variables are implicitly typed as n-functions (the prover enforces this). So the abstraction terms allowed by `W` are all interpretable, since the abstraction terms in L_ω are interpretable. Note further that L_ω, although we have only directly interpreted quantfier-free sentences of T, actually provides indirect interpretations for all quantified sentences of T.

This discussion establishes that the notions of class abstraction can be added harmlessly (as a conservative extension) to any first-order theory with an infinite model. It further needs to be shown that the deductive machinery of the theory of class abstraction is strong enough to recover the usual properties of quantifiers. This is best seen by pointing out that both of the usual rules for universal quantifiers can be emulated in the theory of class abstraction:

```
(forall @ [T]) = true        premise
([T] = [true]) = true        definition of forall
[T] = [true]                 simple case expression reasoning
([T] @ ?x) = [true] @ ?x     localization
T{?x/?1} = true              beta-reduction on both sides
```

demonstrates universal instantiation.

```
T = true                          level 0 theorem (premise)
T{1/0} = true                     level 1 theorem,
                                  by level conversion
T{1/0}{?1/?x} = true              level 1 theorem, by specification
[T{1/0}{?1/?x}] = [true]          level 0 theorem, by localization
forall @ [T{1/0}{?1/?x}]          definition of forall
```

demonstrates universal generalization.

Universal generalization and universal instantiation, combined with propositional logic which we know we can interpret using the logic of case expressions, is enough to verify the rules for the existential quantifier.

Generalized predicates can be represented by free variables in this system (free variables appearing as heads of application terms) but such free variables cannot be replaced with bound variables, which prevents the representation of quantification over predicates: there is no second-order logic here.

7 Higher Order Logic Via Stratified Abstraction

The "λ-calculus" described in the previous section is a late innovation in the logic of Watson. It is a quite weak system. Watson incorporates another much stronger λ-calculus, equivalent in strength and expressive power to a safe variant of Quine's set theory "New Foundations", on which it is based. It is also equivalent in strength to Church's simply typed λ-calculus with an axiom of infinity (with refinements introduced below, it is somewhat stronger). It differs from the Church system in being untyped. It can be noted here that the entire logic `W` is untyped (except for the implicit typing of free variables appearing as heads of curried class application terms). The stronger λ-calculus is called "stratified λ-calculus", and is discussed at length in [5], [6], and [7]. Here our treatment will be briefer.

The stratified λ-calculus is best understood initially via a related typed system, a fragment of the simple type theory of Church (see [1]). The fragment has types indexed by the natural numbers: type 0 is the type ι of individuals (of unspecified character), and type $n+1$, for each n is the type $(n \to n)$ of functions from type n to type n. In addition, type 0 has at least two distinct elements and each type satisfies the type identity $(n \times n) = n$: i.e., each type supports an ordered pair (Watson actually assumes $(n \times n) \subseteq n$; surjectivity of the pair is not assumed).

This type system shares a characteristic with Russell's type theory of sets which Church's simple type theory does not have: all the types look the same in a certain sense. If one raises each type index in an axiom of this typed λ-calculus, one obtains another axiom; it is easy to see from this fact that the same holds for theorems. This suggests that it is reasonable to suppose that the whole structure consisting of types $0, 1, 2 \ldots$ is isomorphic to the structure consisting of types $1, 2, 3 \ldots$ – or even that the type distinctions can be collapsed completely. This is the same as the motivation for the modification of Russell's theory of types for sets which gives Quine's set theory "New Foundations".

It turns out that the ability to safely collapse a type theory using polymorphism in this way is sensitive to details of its axiomatization. If one assumes full extensionality (that every object is a λ-term) it is an open question whether the collapse can be carried out (equivalent to the open question of the consistency of *NF*). If one does not assume extensionality, one obtains a theory which is known to be consistent and essentially equivalent to Jensen's variation *NFU* + Infinity of "New Foundations", which has the same consistency strength and expressive power as Russell's theory of types or Church's simple theory of types (with infinity). This collapsing process is discussed in detail in [6].

The theory obtained when the type structure is collapsed is one-sorted – objects of the theory are not typed – but a notion of "relative type" still plays an important role in the theory. The point is that when the type distinctions are collapsed one still has only those instances of the scheme of β-reduction `[T]@U = T{U/?1}` which make sense in terms of the type scheme. This is vitally important: one does not want to acquire instances of β-reduction like `[∼?1@?1]@?x = ∼?x@?x`, from which, if one defines `R` as `[∼?1@?1]`, one can deduce the dis-

astrous theorem `R@R = ∼R@R`. But the abstraction term `[∼?1@?1]` is in some sense illicit, because there is no way to type it sensibly in terms of the typed system described above.

We now describe the way that these ideas are implemented in the logic `W` of Watson. Each operator is supplied with "relative types" for its arguments (called the left type and right type): if an operator `%` (infix for the sake of the example) has left type i and right type j, this tells us that if a term `T % U` were of type n in the typed system, then `T` would be type $n+i$ and `U` would be type $n+j$. For example, the function application operator `@` has left type 1 and right type 0, because a type $n+1$ function is applied to a type $n+0$ argument to get a type n term. It should be noted that negative relative types are possible: for example, a singleton set operator would have type -1 for its sole argument.

There is an additional option: some operators (such as the class application operator `@!`) are "opaque"; abstraction into an opaque context is not allowed in stratified abstraction terms.

The machinery of relative types is used to identify function abstracts which are allowed in the function abstraction scheme for the application operator `@`. Such abstraction terms are said to be "stratified" by analogy with terminology used in "New Foundations" and related set theories for formulas permitted in set abstracts.

Formal definitions of relative type and stratification follow:

Definition: Occurrences of subterms of a term (with exceptions in opaque contexts) are said to have "relative type" in that term. Relative type is defined recursively:

1. The relative type of a term in itself is 0.
2. If the relative type of an occurrence of the term `A` in a term `T` is n, and the left (resp. right) type of the operator `%` is i, then the relative type of the analogous occurrence of `A` in the obvious occurrence of `T` in `T % U` (resp `U % T` or `% T` (in the case of a unary operator)) is $n+i$. If `%` is opaque, then the relative type of the analogous occurrence of `A` in the obvious occurrence of `T` in any of these terms is undefined.
3. If the relative type of an occurrence of the term `A` in a term `T` is n, then the relative type of the analogous occurrence of `A` in the occurrence of `T` in `[T]` is $n-1$.
4. The relative type of an occurrence of `A` in `T || U , V` is the same as the relative type of its occurrence in the appropriate one of `T`, `U`, `V`.

Definition: An abstraction term `[T]` is "stratified" if the relative type in `T` of each occurrence (there need not be any occurrences) of the variable `?n` bound by the brackets is defined and equal to 0, and if each abstraction term appearing as a proper subterm of `[T]` is stratified.

Axiom scheme (stratified β-reduction): `[T]@U = T{U/?n}` is a level n axiom, when `[T]` is a stratified abstraction term.

Note that a term like `[∼?1@?1]` is actually well-formed, and we do have `([∼?1@?1] @! U) = ∼U@U`, but we do not have the disastrous `([∼?1@?1] @ U)`

`= ~U@U`, because `[~?1@?1]` is not stratified. Notice also that because the class application operator `@!` is "opaque", one cannot define functions in the higher order logic which depend in any nontrivial way on facts about class application.

An important note here is that it might be thought to be dangerous that we have made set function application and class function application coincide for stratified abstraction terms. This turns out not to be a problem as long as one provides enough non-functions (terms `T` not equal to `[T @ ?1]`). The only real curiosity here is that one can prove that set function application is nonextensional by considering class abstractions like `[~?1 @ ?1]` which would be paradoxical if they were also set abstractions.

8 Experience with Watson

It is possible to develop the theory of quantification using the machinery of the stratified λ-calculus alone (and this is how it was done originally). If the class application operator were not used, unstratified abstraction terms would be treated as ill-formed (there is a current release of Watson which still takes this approach). The representation of $(\forall x.\phi)$ as `forall @ [T]` and the verification of instantiation and generalization would still work, with the restriction that we would only consider formulas represented by stratified terms. It is known that all stratified theorems of systems like "New Foundations" or *NFU* have proofs which involve only stratified sentences, and most sentences of mathematical interest are stratified; this restriction did not initially seem to be a problem in practical work with the prover, except for a technical problem detailed below.

There is a technical difficulty with the implementation of first-order logic using stratified λ-calculus which must be noted. A sentence like $(\forall x.(\exists y.x = y))$, which is regarded as "stratified" in the context of a set theory like "New Foundations", is represented in the language of Watson by a term `forall @ [forsome @ [?1 = ?2]]` which is not on the face of it stratified! If the whole term is assigned type 0, the subterm `forall` gets type 1 and the subterm `[forsome @ [?1 = ?2]]` gets type 0; thus the subterm `forsome @ [?1 = ?2]` gets type -1, from which we see that `forsome` gets type 0 and `[?1=?2]` gets type 0. We then see that `?1 = ?2`, and so both `?1` and `?2`, get type -2. The rules of stratification require that the type of `?1` (-2) be the same as the type of the body `forsome @ [?1 = ?2]` of the abstraction term in which it is bound, and this is not the case: the term `forsome @ [?1 = ?2]` has type -1.

This is a merely technical problem because one can show that the relative type of a term with a boolean value (like `forsome @ [?1 = ?2]`) can be freely raised or lowered by any desired amount to recover stratification. The equations `((P || [true] , [false])) @ 0 = P` and `([P] = [true]) = P` hold when `P` is replaced by either `true` or `false`; these equations can be used to freely raise or lower the type of a term whose value is known to be boolean. Of course, no one wants to carry out manipulations of this kind explicitly in a theorem proving system! The solution of this problem was to enable the prover to recognize for itself subterms belonging to classes on which type raising or lowering is possible

and exploit this information to recognize a more general class of terms as stratified. With this generalization of stratification, the technical problem outlined above became entirely invisible to the user.

Classes on which type-raising and lowering is possible (called "strongly cantorian sets", abbreviated s.c.) are of considerable interest in set theories like *NF*. If it is assumed that the set of natural numbers is s.c., it follows that most sets of interest in mathematics and certainly all sets of interest in computer science applications are s.c. The relaxation of stratification restrictions for natural number values and values belonging to common data types proves useful in practice; it has the side-effect of making the logic somewhat stronger than the simple theory of types with infinity. It is beyond the scope of this paper, but it is worth noting briefly that a theme of our research is the study of an analogy between the notion of "s.c. set" and the notion of "data type", and that practical experience with Watson seems to indicate that this can be a useful analogy.

The problems with the treatment using stratified λ-calculus which caused us to introduce the class application operator were subtler, having to do with uniform treatment of quantifiers over variables of different relative types in formal rules for first-order logic. For example, the addition of the class machinery makes it possible to handle the logical principle $(\forall xy.\phi) \leftrightarrow (\forall yx.\phi)$ uniformly; if quantification were implemented using set abstraction, it would be necessary to take the difference in relative type between x and y into account in each such equivalence. The introduction of class application and abstraction increases the ability of the prover to apply limited forms of higher-order matching as well.

The representation of mathematical constructions in this higher-order logic is very similar to the representation in the fragment of Church's type theory described above. Since the latter system is not usually used, some remarks are in order. The lack of types like $((\iota \to \iota) \to \iota)$ and $(\iota \to (\iota \to \iota))$ in this fragment of simple type theory does not create significant problems with expressive power: both of these types are readily represented in type $2 = ((\iota \to \iota) \to (\iota \to \iota))$ by exploiting the coding of any type or subcollection of a type in the linear type scheme using a collection of constant functions in the next higher type. The same device works in general to handle types outside the linear type scheme. Mathematics as implemented in Watson tends to look very much like mathematics implemented in a typed λ-calculus, except for this kind of occurrence of the constant function operator to adjust relative types. The declaration of "data types" as s.c. often allows such occurrences of the constant function operator to be omitted.

9 Conclusions and Relations to Other Work

The main purpose of this paper is to document the mathematical underpinnings of the Watson theorem prover. We have been more attentive to the features which are not documented elsewhere (the logic of case expressions and the new class abstraction machinery) than to the higher order logic embodied in the stratified λ-calculus. We feel that this system has certain features of independent interest,

however. The use of the logic of case expressions as a foundation for propositional logic seems interesting to us; certainly the axiomatization is economical. It is less novel to identify the abstraction implicit in quantification with the abstraction which constructs functions, and in fact the latest version with class application as well as set application retreats from such an identification. The development of Watson has been an outgrowth of our interest in the application of the untyped set theories in the style of Quine and the related λ-calculi, and we believe that untyped grand logics ought to be of interest in theorem proving in general.

We are aware that there is other work using deBruijn indices and related schemes (including the one given here) which would be technically similar to our formal development of substitution, especially by researchers in the area of "explicit substitution". We can only make up for our lack of references by pleading ignorance of this work; our development is independent, though certainly not original.

We do not believe that any theorem proving system is very close in its details to Watson, and in any event the details of the theorem prover are not relevant to this paper. The closest system in terms of its underlying mathematical framework is probably `HOL` ([3]), which implements Church's classical simple type theory of functions.

References

1. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940.
2. N. deBruijn, "Lambda-calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem", in Nederpelt, *et. al.*, eds., *Selected Papers on Automath*, North Holland, 1994.
3. M. Gordon. A proof generating system for higher-order logic. Technical Report 103, University of Cambridge Computer Laboratory, January 1987.
4. M. Randall Holmes, "A functional formulation of first-order logic 'with infinity' without bound variables", preprint, available at the Watson web page `http://math.boisestate.edu/~holmes/proverpage.html`
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, "Untyped λ-calculus with relative typing", in *Typed Lambda-Calculi and Applications* (proceedings of TLCA '95), Springer, 1995, pp. 235-48.
7. M. Randall Holmes, *Elementary Set Theory with a Universal Set*, Academia-Bruylant, Louvain-la-Neuve, 1998.
8. M. Randall Holmes, "The Watson Theorem Prover", preprint, available as part of the online documentation at the Watson web page `http://math.boisestate.edu/~holmes/proverpage.html`
9. Ronald Bjorn Jensen, "On the consistency of a slight (?) modification of Quine's 'New Foundations' ", *Synthese*, 19 (1969), pp. 250-63.
10. W. V. O. Quine, "New Foundations for Mathematical Logic", *American Mathematical Monthly*, 44 (1937), pp. 70-80.