The Watson Theorem Prover[†]

M. Randall Holmes
Dept of Mathematics and Computer Science, Boise State University,
holmes@math.boisestate.edu

Jim Alves-Foss
Center for Secure and Dependable Software, University of Idaho,
jimaf@csds.uidaho.edu

Abstract. Watson is a general purpose system for formal reasoning. It is an interactive equational higher-order theorem prover. The higher-order logic supported by the prover is distinctive in being type-free (it is a safe variant of Quine's *NF*). Watson allows the development of automated proof strategies which are represented and stored by the prover in the same way as theorems. The mathematical foundations of the prover and the way these are presented to a user are discussed. The paper also contains discussions of experiences with the prover and relations of the prover to other systems.

Keywords: theorem prover description, equational reasoning, type theory, New Foundations, higher-order logic

1. Introduction

Watson is a general-purpose system for formal reasoning. It has been developed with applications to software development or verification in mind, but we have also considered applications in education or in pure mathematics. Earlier incarnations of the prover have been called EFTTP (which was a very different system) and Mark2 (which was similar though not identical to Watson; there are occasional references in the paper to differences between Mark2 and Watson: the best reference for Mark2 is the final report (??) of Holmes's first ARO grant).

Watson is a system for computer-supported reasoning by human beings rather than for automatic proof of theorems, though it does support the development of automated proof strategies.

Its logic is equational and rewriting plays a considerable role in the prover, though it is not a typical rewriting system. The properties of expressions defined by cases play an important role in its logic.

Watson supports a higher-order logic, which is distinctive in being type-free: it is a λ-calculus equivalent to a variant of Quine's set theory "New Foundations". References for the theoretical aspects of

[†] We gratefully acknowledge the support of Army Research Office grants DAAH04-94-0247,DAAH04-96-1-0397, and DAAG55-98-1-0263

the higher-order logic include Quine's (??) and Jensen's (??) for the original systems of set theory and Holmes's (??), (??) and (??) for the development of the λ-calculus version.

This paper contains an overview of the mathematical foundations of Watson, followed by a discussion of the way these foundations are implemented in the prover, from the standpoint of a user. These two sections take up most of the paper. Two short sections, one on experiences with the prover and one on the relationships of the prover with other systems, complete the paper.

We are grateful for the support of the Army Research Office through three separate grants. We wish to thank the following students for their hours of work and dedication to this project: at the University of Idaho we thank Sol Espinosa, Fongshing Lam, Aaron Schneider, Kundala Shankar, and Minglong Wu; and at Boise State University we thank Michael Parvin for his excellent help.

Source code and documentation for the Watson system are available at `http://math.boisestate.edu/~holmes/proverpage.html`.

2. The Logic of Watson

The logic of Watson is dominated by three themes. It is *algebraic* (equational). An important role is played by *definition by cases*. Its higher-order logic uses *stratified abstraction*. In this section, we will use notation closer to conventional mathematical notation than to Watson's internal language, which will be introduced in the next section.

2.1. Algebraic logic

There is little that is surprising to be said about the algebraic aspect of Watson. All theorems in a Watson theory are equations, implicitly universally quantified over their free variables.

We exhibit a set of formal axioms for equational logic which are supported by Watson:

reflexivity: $A = A$ is an axiom for any term A.

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.

We introduce notation $A[T/x]$ for the result of substituting the term T for the free variable x in the term A. A formal analysis of substitution will be needed later when abstraction is considered.

substitution: if $A = B$ is a theorem, then $C[A/x] = C[B/x]$ is a theorem.

generality: if $A = B$ is a theorem, then $A[C/x] = B[C/x]$ is a theorem.

This logic of equations is familiar to all of us from high school algebra, though its formal study is not entirely trivial.

The path not taken here leads to the more usual approach to logic in which propositions rather than terms are taken as basic and propositional connectives and quantifiers are logical primitives. In Watson, the propositional connectives and quantifiers are not logical primitives: they can be defined in terms of the logical primitives of Watson (definition by cases and abstraction) or they can be introduced as user-defined primitives in a Watson theory.

2.2. Definition by cases

An expression defined by cases is of the form **if** P **then** T **else** U, which takes the value T if the proposition P is true and the value U if the proposition P is false. This is the notation we will use in this section: the Watson notation is
`P || T , U.`

Any term may appear in the role of P; we adopt Frege's convention that any object not equal to the truth value **true** is understood to play the role of the truth value **false** where a proposition is expected. We refer to P as the *hypothesis* of the expression, and to T and U as its *branches* (positive and negative, respectively).

We present the axioms of definition by cases supported by Watson. Notice that equality appears as a term-forming operation as well as in the role of a predicate here; it should be clear what is meant by each instance of the symbol once this is understood.

hypotheses are propositions: (**if** P **then** T **else** U) = (**if** $P =$ **true** **then** T **else** U)

equations are propositions: $(A = B) =$ **if** $A = B$ **then true else false**

truth value roles: $T =$ **if true then** T **else** U;

$U =$ **if false then** T **else** U.

special equations: $(A = A) =$ **true**;

(**true** = **false**) = **false**.

case distribution: $A[(\textbf{if } P \textbf{ then } T \textbf{ else } U)/x] = (\textbf{if } P \textbf{ then } A[T/x] \textbf{ else } A[U/x])$.

application of hypothesis: $(\textbf{if } A = B \textbf{ then } C[A/x] \textbf{ else } U) = (\textbf{if } A = B \textbf{ then } C[B/x] \textbf{ else } U)$

These axioms, together with the axioms of equational reasoning above, are sufficient to interpret the logic of propositional connectives and identity. This has been demonstrated using the prover; it is also explained in an unpublished paper (??) of Holmes available on the Web.

This logic of case expressions will probably not be familiar to the reader, but it should at least be clear that the axioms are true. To see that they are sufficient to support propositional logic and the logic of identity requires a little work.

2.3. Stratified abstraction

The components of the logic of Watson given so far support the logic of propositional connectives and equality, in conventional terms.

The new component of the logic introduced here is a restricted form of λ-abstraction, the introduction of functions by abstraction. The new component is adequate to support the logic of quantification, and also a powerful higher-order logic.

The particular higher-order logic used will be unfamiliar to most readers, though (we hope) not difficult to explain. It is a variation on the nonstandard set theory "New Foundations" (*NF*) introduced by W. V. O. Quine in (??) (1937), as revised by R. B. Jensen in (??) (1969). Set theories like *NF* avoid paradox by allowing the scheme of set comprehension to be applied only to "stratified" predicates. We will not discuss the set theories here, but will describe the analogous system of λ-calculus with "stratified" λ-abstraction, which is the higher-order logic of Watson (references for this system are (??) (based on Holmes's Ph.D. thesis), which describes a system of synthetic combinatory logic rather than a λ-calculus, (??), which describes the λ-calculus, and chapter 23 of Holmes's book (??), which introduces further material about the representation of data types, discussed below).

The notation we will use in this subsection for λ-abstraction is different from the notation used by Watson. Watson uses a scheme of name-free variable binding due to de Bruijn (see (??); not de Bruijn indices, but a related scheme which we have seen called "de Bruijn levels"); this will be explained below. In this subsection, the usual λ-notation will be used. The usual notation $f(x)$ for function application will be used instead of the notation (fx) more usual in λ-calculus.

We introduce the term constructions of a stratified λ-calculus. These are defined by mutual recursion with a notion of “relative type”. We begin with the description of the type constructions and follow it with a description of the notion of “relative type”.

variables: We have a countably infinite supply of variables.

primitive constants: **true** and **false** are primitive constants, as are the projection functions π_1 and π_2. There is a temptation to identify the projection operators with the truth values which we will resist here.

equations: If T and U are terms, $T = U$ is a term.

pairs: If T and U are terms, (T, U) is a term.

case expressions: If P, T, and U are terms, **if** P **then** T **else** U is a term.

function application: If T and U are terms, $T(U)$ is a term.

λ-term: If T is a term in which the variable x appears free with no type relative to T other than 0 (it need not appear at all) then $(\lambda x.T)$ is a term, in which the variable x is called the binding variable.

As usual, an occurrence of a variable x in a term T is termed *bound* if it appears as part of some occurrence of a subterm $(\lambda x.U)$ of T with the same binding variable; occurrences of variables which are not bound are said to be *free*.

We now define the notion of relative type. Each occurrence of a subterm of a term T has a type relative to T, which is an integer.

the whole term: The type of the term T relative to itself is 0.

equations: If the type of an occurrence of $A = B$ relative to T is n, the types relative to T of the obvious occurrences of A and B are also n.

pairs: If the type of an occurrence of (A, B) relative to T is n, the types relative to T of the obvious occurrences of A and B are also n.

case expressions: If the type of an occurrence of **if** P **then** Q **else** R relative to T is n, the types relative to T of the obvious occurrences of P, Q, and R are also n.

function application: If the type of an occurrence of $A(B)$ relative to T is n, the types relative to T of the obvious occurrences of A and B are $n+1$ and n, respectively.

λ-terms: If the type of an occurrence of $(\lambda x.U)$ relative to T is n, the type relative to T of the obvious occurrence of U is $n-1$ (by the condition on well-formedness of λ-terms, this will be the same as the type of the free occurrences of the variable x in U).

A well-formed λ-term is said to be *stratified*, for reasons which will be explained in the next subsection.

The axioms of this theory are the axioms of algebraic and case expression logic from above, plus the following axioms for the new notions of pair, application and λ-abstraction:

projection functions: $\pi_1(x,y) = x$; $\pi_2(x,y) = y$.

β-reduction: $(\lambda x.T)(A) = T[A/x]$

Unfortunately, things are not as simple as the brief paragraph before the new axioms tries to make them appear. Algebraic logic is extended to allow substitution of equals for equals inside λ-terms, without regard to the fact that the terms for which substitutions are made may fail to have reference outside the λ-terms, because of the presence of bound variables. This extension of the notion of substitution is equivalent to a weak form of extensionality for functions. Notice that we do *not* adopt the full axiom of extensionality, which we exhibit:

extensionality axiom not adopted: $(\lambda x.f(x)) = f$

If we were to adopt this assumption we would have a system equivalent to Quine's "New Foundations", which is not known to be consistent. As it is we have a system equivalent to an extension of the modification *NFU* + Infinity of *NF* due to Jensen in (??), which is known to be equivalent to Russell's theory of types (with infinity), which is generally believed to be consistent.

Further, the notion of substitution which appears both here and in the earlier axioms is considerably complicated by the need to avoid collisions of bound variables. We will not give a formal account of substitution here, but we will give a formal description below of the notion of substitution for Watson's own notation (using de Bruijn levels).

We briefly indicate how quantification is supported. We can define the universal quantifier thus:

definition of $\forall$: $(\forall x.T) = ((\lambda x.T) = (\lambda x.\mathbf{true}))$

The notation of Watson does not permit the introduction of new binders: the actual notation is more like $\forall(\lambda x.T)$. The existential quantifier $\exists$ may be defined using deMorgan's laws as usual. Reasoning with the existential quantifier may be facilitated by introducing a new primitive function **choose** and the dreaded

axiom of choice: $(\exists x.T) = T[\mathbf{choose}(\lambda x.T)/x]$

The HOL system uses this as the definition of the existential quantifier (and defines the universal quantifier in terms of the existential quantifier); this axiom can be used to facilitate existential reasoning in Watson, though the built-in logic without choice also supports adequate existential reasoning. It is interesting to note that the logic of Watson with full extensionality is inconsistent with the axiom of choice (because NF is inconsistent with choice).

2.4. The relationship between stratified λ-calculus and usual type theories

In the previous section, we introduced the term constructions and axioms of a stratified λ-calculus without motivation in terms of more familiar mathematical approaches. In this section, we will describe a motivation for this system and indicate its relationship with more familiar type systems.

Stratified λ-calculus is not a type system at all in the usual sense. Terms in stratified λ-calculus do not have types. The scheme of relative types serves only to restrict what λ-terms (functions) can be defined. For example, the term $x(x)$, which would not make sense in a typed system, is meaningful in stratified λ-calculus, but the term $(\lambda x.x(x))$ is not.

Although stratified λ-calculus is an untyped system itself, it has a close relationship with a typed system, consideration of which can help us to see what is going on. We briefly introduce a quite conventional typed λ-calculus, then indicate how it can be restricted to get a system related to our stratified λ-calculus.

We restrict ourselves to pairing, function application and abstraction as term constructions for the sake of simplification. We assume a base type ι of individuals. If α and β are types, $\alpha \times \beta$ and $\alpha \rightarrow \beta$ are types, called product types and arrow types respectively. The inhabitants of $\alpha \times \beta$ are to be understood to be pairs with first projection of type α and second projection of type β. The inhabitants of $\alpha \rightarrow \beta$ are understood to be functions from type α to type β. The types are exactly those which can be constructed from ι using the given type constructors.

This motivates the following restrictions on our term constructions: variables require type labels (we still have a countable supply of each type); the projection functions π_1 and π_2 are replaced by infinitely many different versions with type superscripts of the forms $(\alpha \times \beta) \rightarrow \alpha$ and $(\alpha \times \beta) \rightarrow \beta$, respectively; any pair term is still well-formed, but requires the appropriate product type superscript; a function application term $T(U)$ is only well-formed if T has an arrow type $\alpha \rightarrow \beta$ and U has type α (the application term is then of type β); a λ-term $(\lambda x.T)$ is always well-formed if T is well-formed, and has type $\alpha \rightarrow \beta$, where α is the type of x and β is the type of T.

In this system, a term like $x(x)$ cannot be well-formed because no type superscript appropriate to x can be constructed. A hint of the advantage which we see in the stratified λ-calculus can be seen in the proliferation of projection functions in different types. Similarly, there is a different identity function $(\lambda x^\alpha.x^\alpha)$ on each type α (inhabiting $\alpha \rightarrow \alpha$); this is a general phenomenon.

The restricted version of this type scheme which is related to stratified λ-calculus is obtained as follows. Its types are labelled by natural numbers. Type 0 is to be identified with ι. Type $n+1$ is to be identified with $n \rightarrow n$ (where n stands for the type already labelled by n). Further, we identify type n with the product type $n \times n$ (this assumption is harmless if types are understood to have infinitely many inhabitants).

The inhabitants of type 0 are individuals, and enjoy a surjective pairing function under which each individual is identified with some pair of individuals. The inhabitants of type $n+1$ are the functions from type n to type n, for each n; it is easy to define the surjective pair on type $n+1$ in a uniform manner in terms of the pair on type n.

There is a great deal of polymorphism in this restricted type system. In fact, every theorem which can be proved about types 0,1,2... has a precise analogue with each type label raised by one which can be proved about types 1,2,3..., and each definable object in the system using types 0,1,2... has an analogue defined in the same way using types 1,2,3.... This polymorphism motivates the idea of collapsing the type structure entirely: suppose that all the types are in fact the same domain, but keep the restrictions on the formation of abstractions inherited from the typed system, and we obtain a stratified λ-calculus.

The practical application of this to working with Watson is to keep in mind the relationship between relative types of objects in the definition of a function in stratified λ-calculus and the concrete types of individuals, functions from individuals to individuals (type 1), functions from type 1 functions to type 1 functions (type 2), etc., in this typed calculus.

It might seem we lose expressive power through not allowing types such as $(\iota \to \iota) \to \iota$ or $\iota \to (\iota \to \iota)$ (functions from type 1 to type 0 and vice versa), but these types are both readily coded into type 2. A function from type 1 to type 0 is represented by a function of type 2 taking a type 1 function to the (type 1) constant function of the type 0 value of the coded function. A function from type 0 to type 1 is coded by a function of type 2: values at constant functions of its intended type 0 arguments of the coded function are the intended type 1 values, while values at nonconstant functions are ignored (they may be taken to be a default value). A combination of these devices and similar considerations about product types allows the coding of any type in the simple type theory of Church. Experience suggests to us that there is not any serious loss of mathematical fluency.

The use of the term "stratified" for well-formed λ terms can be motivated by considering the fact that we have to organize the functions and arguments appearing in the specification of a new function by a λ term into "strata" corresponding to the integer types of our restricted type system. When considering a complex term, it can be helpful to draw a diagram with horizontal levels into which each object appearing in the λ-term is placed, and make sure that functions on one level are applied only to arguments on the next level down.

A final comment: the common practice of "currying" (replacing a function of type $(\alpha \times \beta) \to \gamma$ whose typical value might be written $f(x, y)$ with the related function of type $\alpha \to (\beta \to \gamma)$ whose typical (iterated) value might be written $f(x)(y)$) used in conventional systems of typed and untyped λ-calculus is not used in stratified λ-calculus because it is not sound in terms of relative type: in $f(x, y)$ the relative types of x and y are the same, while they are different in $f(x)(y)$. This is the reason why we have used the conventional notation $f(x)$ for function application rather than the notation fx more usual in combinatory logic and λ-calculus (the latter notation lends itself better to currying).

2.5. Strongly cantorian domains

The equations $(P = \mathbf{true}) = (\mathbf{if}\ P\ \mathbf{then}\ \pi_1\ \mathbf{else}\ \pi_2)(\mathbf{true},\mathbf{false})$ and $(P = \mathbf{true}) = ((\lambda x.P) = (\lambda x.\mathbf{true})$ (where P is a variable and so in particular does not contain a free occurrence of x) are readily proven true in our system of stratified λ-calculus. The intriguing feature of these equations is that the relative type of P on the two sides of each equation is different; in fact, any expression whose value is a truth-value can have its type freely raised or lowered by any amount by application of these equations, which suggests that a more liberal criterion of stratification is possible where certain subterms are known to represent truth values.

The fact about the set of truth values indicated in the paragraph above is stated in the terminology of *NF* and related set theories as "the set of truth values is *strongly Cantorian*". Formally speaking, the defining characteristic of a strongly Cantorian set A is that the restriction of the constant function constructor to the set A is realized by a function. We say "realized by a function" because all functions in our logic have universal domain and may be applied to objects other than elements of the set A. We can sharpen this by using retractions (functions τ such that $\tau(\tau(x)) = \tau(x)$ for all x) with range A to represent sets A. We then can define a (nonempty) strongly Cantorian domain as the range of a retraction τ such that there is a function κ such that $\kappa(\tau(x)) = (\lambda y.\tau(x))$ for all x.

The importance of strongly Cantorian sets in set theories like *NF*, and of strongly Cantorian domains in stratified λ-calculus, is that comprehension or abstraction over these sets is less affected by stratification restrictions. Moreover, they are closed under operations precisely analogous to the type constructors under which we would want data types to be closed in computer science: the realm of strongly Cantorian sets is closed under cartesian product, power set, and the set theoretical analogue of the arrow type constructor. It contains all concrete finite sets. The assertion that the set of natural numbers is strongly Cantorian is consistent with our stratified λ-calculus (though it strengthens it somewhat) and implies further that any set we are likely to want to use as a data type in computer science (or, indeed, in most of mathematics outside of technical set theory) is strongly Cantorian. This assertion completes the underlying logic of Watson.

Just as in the case of the truth values above, any term whose value is guaranteed to belong to a fixed strongly Cantorian set may have its relative type freely raised or lowered in determining stratification of a λ-term. It turns out to be a practical necessity for the handling of quantification that the stratification-checking features of Watson be able to recognize at least the type of truth-values as a strongly Cantorian domain, and we went ahead and incorporated a feature of the prover supporting the recognition of general strongly Cantorian domains, which will be discussed further below. As hinted in the preceding paragraph, computer science data types are expected to be represented in theories developed under Watson as strongly Cantorian domains; retractions with strongly Cantorian range are used as type labels.

There is a discussion of the theory behind the representation of data types using strong Cantorian domains in chapter 23 of our book (??).

3. The implementation of the logic in Watson

In this section, we discuss the ways in which the logical features described from a mathematical standpoint in the previous section are implemented in Watson.

3.1. The syntax of the term language of Watson

Watson is primarily concerned with the manipulation of terms (as opposed to, say, propositions). This subsection describes the syntax of the term language.

3.1.1. *Atomic terms*

Atomic terms of the language of Watson are of four kinds. All atomic terms are represented by strings of nonzero length of alphanumeric characters (Watson is case-sensitive) plus the special characters `?` and _.

numerals: A string consisting entirely of digits is a numeral.

bound variables: A string consisting of a single `?` followed by a string of digits of nonzero length is a bound variable.

free variables: A string beginning with `?` which is not a bound variable is a free variable (`?` by itself is a free variable).

constants: A string beginning with a character other than `?` and containing at least one non-digit is a constant.

3.1.2. *Operators*

An operator is represented by a string of special characters other than `?`, _, braces, brackets, parentheses or quotes, possibly followed by a suffix consisting of the back-quote ‘ followed by a string eligible to be an atomic term. It is permissible for an operator to have no suffix; a suffix by itself is also a permissible operator; the empty string is not an operator.

An operator beginning with a caret ^ , other than the one-character operator ^ itself, is an operator variable. The syntactical privileges of operator variables are the same as those of other operators.

3.1.3. *Compound Terms*

Compound terms are of four kinds, syntactically. The deep structure of certain terms is not the same as the surface syntax. The description given here is appropriate to the default syntax of Watson; we will indicate below how this is modified by user-defined precedence and grouping, which is available as an option in Watson, though it has not been used so far in any extensive theory development.

prefix terms: An operator followed by a term is a term. As will be described below, some surface prefix terms are actually infix terms in disguise.

parenthesized terms: A term enclosed in parentheses `()` is a term. Of course, a parenthesized term does not differ in any way in its internal representation from the same term without parentheses.

abstraction terms: A term enclosed in brackets `[]` is a term.

infix terms: An atomic, parenthesized or function term followed by an operator followed by a term is a term.

case expression restriction: The only terms of the form `term1 || term2` permitted are of the form `term1 || term2 , term3`. The apparent subterm `term2 , term3` of such a term does not correspond to anything in the internal representation of such a term (i.e., `__ || __ , __` is a 3-place mixfix operator).

3.1.4. *Aside on Precedence and Grouping*

The default precedence and grouping convention of Watson, reflected in the previous subsection, is that of the old computer language APL: all operators have the same precedence, and all group to the right. This convention is at odds with standard mathematical usage, but not at all hard to learn.

Watson supports user-defined precedence and grouping: precedences are natural numbers, with all operators assigned even precedence grouping to the right and all operators assigned odd precedence grouping to the left. The default precedence of operators not assigned a specific precedence also may be reset from 0 to any desired value (this may also affect the default grouping, of course).

Historically, we do not believe that any user other than the designer has actually made use of the user-defined precedence features of Mark2 or Watson in extended work. Users seem to adapt well to the default "APL" conventions. We believe that user-defined precedence is likely to be important in any future educational applications of Watson, where familiarity of notation will be at a premium; there also might be application areas where modifications in precedence will prove to be of practical importance.

3.2. Implementing algebraic logic in Watson

In this subsection, we discuss the way in which the principles of algebraic logic described above are implemented by Watson. The syntax of actual commands is introduced only in the examples.

3.2.1. *A brief introduction to declarations*

Constant atomic terms and non-variable operators must be declared before being used in a theory. We will see below that it is permitted and sometimes important to declare variable operators. It is neither necessary nor possible to declare free or bound variables.

Equational axioms can be introduced by a simple command exhibited below in the first example. There are more complex commands discussed below for introducing constants and operators by definition.

3.2.2. *The basic session model of Watson*

A typical Watson session begins with a user entering a term. The user then manipulates the term by applying theorems to subterms of this term as rewrite rules (all Watson theorems are equations), and closes by proving a theorem to the effect that the term originally entered is equal to the term finally arrived at.

There are two moves available to the user which qualify this picture: he may interchange the term originally entered and the term most recently arrived at, or he may carry out a global substitution of some term for a free variable, which will affect the term originally entered as well as the term most recently arrived at.

Though it appears to the user that he is “editing” a term, what he is actually editing is an equation. Internally, the prover stores two terms, the left side and right side of the equation being edited. What the user views is the right side of the equation (there is also a command available to view the left side). There is a command which interchanges the left and right sides of the equation, and there is a global assignment command which replaces a free variable by the same term everywhere in the equation; all other commands manipulate the right side only. At the end of the session, the user proves the current equation: he assigns a name to it and it is stored as a new rewrite rule available for the proof of further theorems.

We reproduce the rules of equational reasoning from above and point out how they are implemented in the basic session model:

reflexivity: $A = A$ is an axiom for any term A.

Each such axiom is implemented by simply entering the term A (which has the effect of making the current equation $A = A$).

symmetry: if $A = B$ is a theorem, then $B = A$ is a theorem.

If $A = B$ has been proved, the user can swap the left and right hand sides of the theorem to prove $B = A$.

transitivity: if $A = B$ is a theorem and $B = C$ is a theorem, then $A = C$ is a theorem.

A sequence of rewrites which prove $A = B$, followed by a sequence of rewrites which prove $B = C$, give a proof of $A = C$.

We recall the notation $A[T/x]$ for the result of substituting the term T for the free variable x in the term A. If T is a term containing bound variables (or "embedded local hypotheses", a topic to be discussed below), there are some technicalities to be considered below in how the substitution is effected.

substitution: if $A = B$ is a theorem, then $C[A/x] = C[B/x]$ is a theorem.

This is implemented by the ability to apply the theorem $A = B$ as as a rewrite rule, possibly repeatedly (how this is done is described in more detail in the next subsection).

generality: if $A = B$ is a theorem, then $A[C/x] = B[C/x]$ is a theorem.

This is implemented by the global assignment command.

3.2.3. *Navigation within terms*

As we noted above, we have not described in detail how applications of theorems as rewrite rules are carried out. Recall that all theorems proved by Watson are equations (implicitly universally quantified over their free variables) and so are suitable for use as rewrite rules.

As we stated above, the user views the right side of the equational theorem under construction. In fact, the user views not only the right side, but also a selected subterm of the right side. When the user applies a theorem as a rewrite rule, it is to this selected subterm that he applies it. The user does *not* rewrite proper subterms of the selected subterm to which the rewrite rule is applicable (as he would in most rewriting systems); the appropriate side of the rewrite rule is matched only to the selected subterm itself (and applied if applicable).

The user has further commands at his disposal which control which subterm of the right side of the equation is selected to view. These are referred to collectively as "term navigation commands". These include commands to move to the right or left subterm of the current selected subterm, to the parent subterm of the current selected subterm, or to the top term (the entire right side of the equation). There are some more sophisticated navigation commands as well.

Certain terms present technical challenges under this picture. An abstraction term has only one immediate subterm: moving to the left or to the right takes one to that same subterm. A case expression `?p || ?x, ?y` doesn't really have an immediate right subterm; the prover allows one to move to this virtual subterm on the way to modifying its real left and right subterms, but it does not allow the application of rewrite rules to the virtual subterm. Moving to the left in a prefix term will reveal the hidden left subterm if there is one and produce a subterm error otherwise.

The selected subterm is enclosed in braces { } in the display of the entire right subterm, unless it happens to be the virtual subterm of a case expression.

3.2.4. *An example*

We give an example of a simple Watson proof to flesh out the abstract discussion above. Watson is implemented in SML, and its commands are ML function evaluations. Commands end with semicolons; more than one command may be put on a line. A command with no apparent arguments needs the null argument () of ML. Most parameters passed to commands are strings, and so are enclosed in double quotes ". Comments may be enclosed in decorated parentheses `(* *)` (in what follows we take the liberty of commenting not only code but also the term displays, in a way which would not happen in real Watson output). The prover normally displays additional prompts, which are suppressed here for compactness of presentation. The prompt at which an ML command to the prover is entered is `-`.

```
(* some declarations needed for setup *)

- declareinfix "+";
- axiom "COMM" "?x+?y" "?y+?x";
- axiom "ASSOC" "(?x+?y)+?z" "?x+?y+?z";
```

```
(* We begin the proof of a simple theorem.  "s" is an
abbreviation for "start", the full name of the command
which starts a proof. *)

- s "?x+?y+?z";

{?x + ?y + ?z}             (* the resulting display *)
?x + ?y + ?z

(* We demonstrate navigation commands. *)

- left();
{?x} + ?y + ?z             (* the resulting display *)
?x

- up();                    (* returns to previous
                           position; display omitted *)
(* We start the proof. *)

- ri "COMM"; ex();         (* introduce a rewrite rule
                           (ri abbreviates ruleintro);
                           invoke tactic interpreter *)
{COMM => ?x + ?y + ?z}     (* display after ri *)
COMM => ?x + ?y + ?z

{(?y + ?z) + ?x}           (* display after execution *)
(?y + ?z) + ?x

- left(); ri "COMM"; ex(); (* move to left subterm,
                           introduce rewrite rule,
                           invoke tactic interpreter *)
{?y + ?z} + ?x             (* display after left *)
?y + ?z

{COMM => ?y + ?z} + ?x     (* display after
COMM => ?y + ?z            introducing rewrite *)

{?z + ?y} + ?x             (* display after executing
?z + ?y                    tactic interpreter *)

- up(); ri "ASSOC"; ex();  (* move up, introduce rewrite
                           (associativity)
                           then use tactic interpreter *)
```

```
{(?z + ?y) + ?x}          (* display after moving up *)
(?z + ?y) + ?x

{ASSOC => (?z + ?y) + ?x}  (* display after
ASSOC => (?z + ?y) + ?x    introducing rewrite rule *)

{?z + ?y + ?x}             (* display after executing
?z + ?y + ?x               tactic interpreter *)

- p "REVERSE";             (* prove new theorem REVERSE *)

REVERSE:                   (* resulting theorem display *)
?x + ?y + ?z  =
?z + ?y + ?x
ASSOC , COMM , 0           (* dependencies on axioms;
                           0 is a list terminator *)

- s "?a+?b+?c+?d";         (* test new theorem *)
{?a + ?b + ?c + ?d}        (* display *)
?a + ?b + ?c + ?d

- ri "REVERSE"; ex();      (* introduce rewrite rule
                           and use tactic interpreter *)
{REVERSE => ?a + ?b + ?c + ?d}
REVERSE => ?a + ?b + ?c + ?d

{(?c + ?d) + ?b + ?a}      (* display of final result *)
(?c + ?d) + ?b + ?a
```

A particular feature to which we wish to call the attention of the reader is the two-step handling of rewrite rules (introduction of a reference to the rewrite rule to be applied into the term, followed by invocation of the “tactic interpreter” `execute` (abbreviated `ex`)), which might seem odd. The reason for this should become evident in the next subsection.

3.2.5. *The tactic language introduced*

The model of proof introduced in this subsection and exemplified in the previous subsection is effective in principle but tedious in practice. Watson provides a technique for executing many proof steps automatically, which may properly be discussed at this point. A program for executing proof steps automatically is called a “tactic” by a perhaps false analogy with usage in HOL and other provers of the LCF family. In those systems, a tactic is an ML program, an object of quite a different sort than a theorem. In Watson, a tactic is represented by the prover to itself as an equational theorem, but nonetheless may exhibit complex execution behavior. We describe here the way in which this effect is achieved.

The key feature of the language of Watson which makes the tactic language possible is the ability to represent the intention to apply a rewrite rule inside a term. This is done using the special infixes `=>` and `<=`.

A term like `(COMM => ?x + ?y) + ?z` has precisely the same mathematical referent as the term `(?x + ?y) + ?z`. The presence of the embedded theorem `COMM` (a commutativity axiom) indicates the intention of applying the theorem `COMM` as a rewrite rule to the indicated subterm; running the tactic interpreter `execute` will cause the term to be converted to the form `(?y + ?x) + ?z` (which of course still has the same referent). The effect of the other special infix `<=` is to signal the intention of rewriting with the "converse" of the given theorem: if `ASSOC` is the theorem `((?x+?y)+?z)=?x+?y+?z`[1], the effect of executing the tactic interpreter on `ASSOC <= ?a+?b+?c` will be to convert the term to the form `(?a+?b)+?c`.

When there are several embedded theorems present, the tactic interpreter applies all of them, following a depth-first strategy (this needs to be qualified where expressions defined by cases are involved; see below). This applies as well to any embedded theorems which are introduced by rewriting with other theorems: it is possible to prove theorems which contain embedded theorems (though only on the right side). When an embedded theorem cannot be used to rewrite the subterm to which it is attached, it is simply dropped.

Further, it is possible to prove theorems which invoke recursive calls to themselves. It is necessary to use a special declaration command to do this, as we will see in the example below (since theorem names can be embedded in terms, they are required to be declared; the `prove` command normally serves to declare a theorem name, but this will not work when recursive dependencies are present; this is analogous to the treatment of forward declarations and function prototypes in programming languages).

[1] Recall that the default precedence of Watson is to the right, so the right side of this equation is equivalent to `?x + (?y + ?z)`.

3.2.6. *An simple example of tactic development*

```
- axiom "ZERO" "0+?x" "?x";      (* declarations made above
                                 still in force *)
                                 (* 0 is a numeral, so
                                 predeclared *)

- declarepretheorem "ZEROES";    (* declare intention
                                 of proving a theorem
                                 ZEROES *)

- s "0+?x";                      (* display omitted *)
- ri "ZERO"; ex();

{?x}                             (* the expected display *)
?x

- ri "ZEROES";

{ZEROES => ?x}                   (* we leave the intention
                                  hanging; after all,
 ZEROES => ?x                     what does ZEROES do? *)

- prove "ZEROES";

ZEROES:                          (* the ``theorem'' *)

0 + ?x  =
ZEROES => ?x                     (* note presence of
                                 recursion *)

ZERO , 0                         (* axioms used *)

(* now we test it *)

- s "0+0+0+?x";                  (* display omitted *)
- ri "ZEROES"; ex();

{?x}                             (* the final display *)
?x
```

```
(* demonstration of trace feature *)

- startover();                  (* this command resets
                                both sides of the equation
                                to the left side *)

{0 + 0 + 0 + ?x}
0 + 0 + 0 + ?x

- ri "ZEROES"; steps();         (* the steps command
                                traces tactic execution *)

ZEROES => 0 + 0 + 0 + ?x        (* display traced steps *)
ZEROES => 0 + 0 + ?x
ZEROES => 0 + ?x
ZEROES => ?x
?x                              (* note that the embedded
                                theorem is simply dropped
                                when it does not apply;
                                this makes termination
                                possible *)
```

This extremely simple example gives the basic idea of Watson's tactic language; ZEROES as an equational theorem is certainly true (this follows from the axiom ZERO and the fact that embedded theorems have no effect on term reference); as seen here, this "equational theorem" has execution behavior more general than can be achieved with any single application of a rewrite rule.

3.2.7. *More sophisticated tactics*

In this subsection and the following example, we discuss operations on tactics and tactics with parameters.

introduction of new variables: When a rewrite introduces new free variables, they are automatically supplied with a numerical subscript as "new" variables. Something more complex may happen if the new variable is introduced inside an abstraction (bracketed) term; see below. The introduction of new variables can always be avoided by the use of parameterized versions of theorems which supply values to the new variables; see examples below.

```
(* example of introduction of new variables *)

- declareunary "-";          (* declare - as a
                             prefix operator *)
- axiom "INV" "?x + -?x" "0";

- s "0";
- rri "INV";                 (* rri = revruleintro
                             applies converse of
                             theorem *)

{INV <= 0}                   (* display *)
INV <= 0

- ex();

{?x_82 + - ?x_82}            (* note appearance
?x_82 + - ?x_82              of new variables *)
```

control structures: The special infix operators =>>, <<=, *> and <* allow one to apply a rewrite rule (or its converse) depending on whether the application of a preceding rewrite rule succeeded or failed.

The complex rewrite rules thm1 =>> thm2 and thm1 <<= ?thm2 have the effect of applying thm1 then applying thm2 (resp. its converse) only if the application of thm1 fails. (the handling of these infixes by Watson is quite different from their former handling by Mark2: a chain of alternatives applied to a term which Watson represents as (thm1 =>> thm2 =>> ... =>> thmn) => term was represented by Mark2 as thmn =>> ... thm2 =>> thm1 => term). The commands altruleintro (ari) and altrevruleintro (arri) introduce these "alternative rule infixes" (their use is illustrated in the examples below).

The complex rewrite rules thm1 *> thm2 and thm1 <* thm2 have the effect of first applying thm1, then applying thm2 only if the application of thm1 succeeded.

For either of the above kinds of operator, if the theorem thm1 happens to be a built-in operator (e.g., the abstraction and reduction tactics BIND and EVAL introduced below), thm1 is said to "succeed" if it makes a change in the target term, and "fail" otherwise. For theorems of the usual kind, success or failure depends on whether the appropriate side of the theorem matches the target term.

In the following example, we illustrate the application of =>> to fine-tune the behavior of a tactic. We prove a theorem which applies the identity of addition on either side.

```
(* a preliminary result *)

- s "?x+0";                (* displays suppressed *)
- ri "COMM"; ri "ZERO";
- prove "COMMZERO";

COMMZERO:
?x + 0  =
ZERO => COMM => ?x + 0
0                          (* no axiom was used
                           in proving this,
                           though some
                           were mentioned *)

(* the tactic to apply identity on either side --
naive version *)

- s "?x+?y";
- ri "ZERO"; ri "COMMZERO";
- p "EITHERZERO";

EITHERZERO:
?x + ?y  =
COMMZERO => ZERO => ?x + ?y
0

(* the problem with this *)

- s "0+?x+0";
- ri "EITHERZERO"; ex();

{?x}                       (* the final display *)
?x

(* the difficulty is that successive applications
of theorems cannot be relied upon to serve as
alternatives; one may apply and then another after
it in the list *)
```

```
(* we modify the tactic *)

- s "?x+?y";
- ri "ZERO"; ari "COMMZERO"; (* ari =
                             altrevruleintro
                             introduces an
                             alternative
                             theorem to be
                             applied if COMM fails *)
- reprove "EITHERZERO";

EITHERZERO:
?x + ?y  =
(ZERO =>> COMMZERO) => ?x + ?y  (* note the
                                syntactical
0                               effect of ari *)

(* we repeat the test above *)

- s "0+?x+0";
- ri "EITHERZERO"; ex();

{?x + 0}                    (* the final display:
?x + 0                      only one application
                            of identity is made *)
```

tactics with parameters; operators as tactics Watson allows the user to develop tactics with parameters, which may be used to pass terms or other theorems or tactics as data to a tactic. The `prove` command will take parameter lists built with the predeclared function application infix "`@`" and pairing infix "`,`"; it will match the parameter list against actual embedded occurrences of the theorem, which needs to have parameters matching the original parameter list in order to be used successfully for rewriting. Watson also allows operators (prefix or infix) to be "proved" as (necessarily parameterized) tactics.

```
(* an example of parameterized and operator theorems:
simple structural tactics *)

- declareopaque "^+";           (* variable operator
                                declaration will be
                                explained below *)
```

```
- s "?x^+?y";
- right(); ri "?thm";
- p "RIGHT@?thm";

RIGHT @ ?thm:
?x ^+ ?y  =
?x ^+ ?thm => ?y
0

(* a test *)

- s "?x+?y+?z";
- ri "RIGHT@COMM"; steps();

(RIGHT @ COMM) => ?x + ?y + ?z (* display of
?x + COMM => ?y + ?z           rewriting steps *)
?x + ?z + ?y

(* operators as theorems *)

- s "?x";
- ri "?thm1";
- ri "?thm2";
- p "?thm1**?thm2";

?thm1 ** ?thm2:
?x  =
?thm2 => ?thm1 => ?x
0

(* a test *)

- s "?x+?y+?z";
- ri "COMM**ASSOC"; ex();

{?y + ?z + ?x}                 (* final display *)
?y + ?z + ?x
```

The prefix operator `!@` can be applied to a theorem whose application introduces new variables to produce a parameterized theorem to which values of these new variables can be supplied; the operator `!$` applied to a theorem whose converse introduces new variables produces a parameterized theorem with the same effect as the converse of the original theorem.

In the following example, we develop a tactic converse of a theorem which eliminates variables using parameters and using the built-in operator !$.

```
(* parameterized converse theorem *)

- s "0";
- initializecounter();      (* initializes suffixes
                            of new variables *)
- rri "INV"; ex();

{?x_1 + - ?x_1}             (* display *)
?x_1 + - ?x_1

- assign "?x_1" "?x";       (* rename new variable
                            using global assignment *)
- prove "INV_INVERSE@?x";

INV_INVERSE @ ?x:
0  =
?x + - ?x
INV , 0

- s "0";
- ri "INV_INVERSE@3"; ex();

{3 + - 3}                   (* final display *)
3 + - 3

(* the same effect using a built in operator *)

- s "0";
- ri "(!$ INV)@3"; ex();

{3 + - 3}                   (* final display *)
3 + - 3
```

We close the section with a not altogether trivial example.

```
- declarepretheorem "ASSOCS";
- s "(?x+?y)+?z";
- ri "ASSOC"; ex();
- ri "ASSOCS";
- p "ASSOCS";

(* ASSOCS applies associativity
as many times as possible at the top *)

ASSOCS:
(?x + ?y) + ?z  =
ASSOCS => ?x + ?y + ?z
ASSOC , 0

(* a test *)

- s "((?x+?y)+?z)+(?u+?v)+?w";
- ri "ASSOCS"; ex();

{?x + ?y + ?z + (?u + ?v) + ?w}  (* the final display *)
?x + ?y + ?z + (?u + ?v) + ?w

(* the full tactic *)

- declarepretheorem "ALLASSOCS";

- s "?x+?y";
- ri "ASSOCS"; ri "RIGHT@ALLASSOCS";
- p "ALLASSOCS";

ALLASSOCS:
?x + ?y  =
(RIGHT @ ALLASSOCS) => ASSOCS => ?x + ?y
0                              (* no axiom dependencies
                               because no axiom was
                               actually used to rewrite *)
```

```
(* test *)

s "(((?x+(?y+?z)+?w)+(?u+?v)+?w)+
                      ((?u+?v)+?w)+?e)+(?u+?v)+?e+?f";

{(((?x + (?y + ?z) + ?w) + (?u + ?v) + ?w) + ((?u
           + ?v) + ?w) + ?e) + (?u + ?v) + ?e + ?f}

(* duplication of displays suppressed *)

- ri "ALLASSOCS"; ex();

(* the final display -- intermediate and duplicate
displays suppressed *)

{?x + ?y + ?z + ?w + ?u + ?v + ?w + ?u + ?v + ?w + ?e
   + ?u + ?v + ?e + ?f}
```

3.3. Implementing case expression logic in Watson

Case expression logic is implemented in Watson by the addition of special features to the tactic interpreter.

The execution order of the tactic interpreter is normally depth-first; however, when called on a case expression `P || T , U`, the hypothesis `P` is rewritten first. If `P` rewrites to the form `true = X`, it is automatically further rewritten to `X`; if the final form of the hypothesis is `true` or `false`, the whole expression is rewritten to `T` or `U` respectively, and the dropped alternative is never rewritten at all (this is the one case in which the tactic interpreter is non-strict in its "order of evaluation").

The tactic interpreter recognizes certain built-in tactics built with numerals and the special operator `|-|`. These enable rewriting with the hypotheses of case expressions in appropriate contexts. In a tactic `m |-| n`, the numeral `m` will be 0, 1, or 2, indicating the type of rewriting being done, and the numeral `n` will indicate which hypothesis is being used. The hypothesis of the largest case expression which contains the subterm being rewritten as a subterm (not necessarily proper) of one of its branches is numbered 1; the hypothesis of the second largest such case expression is numbered 2, and so forth.

The special tactic `0 |-| n` does rewriting in the positive branch of the case expression whose hypothesis is numbered `n`. If the hypothesis is of the form `A = B`, the tactic `0 |-| n` will rewrite `A` to `B`; if it is introduced in the converse sense it rewrites `B` to `A`. The target of this rewrite needs to be identical to the appropriate side of the equation used to rewrite, not just a match as in the case of rewriting with theorems. If the hypothesis `X` is not an equation, it is treated just as if it were the equation `true = X`.

The special tactic `1|-|n` rewrites case expressions `P || T , U`, in the case where the hypothesis `P` is the same as the hypothesis numbered `n`, to `T` or `U` depending on whether the subterm being rewritten is in the positive or negative branch of the case expression with the `n`th hypothesis. The converse of this rewrite rule rewrites the subterm to a new case expression with the `n`th hypothesis, with the original form of the subterm as one branch and a new variable as the other branch. The special tactic `2 |-| n`, which is only used in the converse sense, rewrites the subterm in the same way as the converse of `1 |-| n`, except that it takes a parameter which is used in place of the new variable as the new branch. Note that the new branch introduced by these converse rewrite rules will have contradictory local hypotheses applicable to it.

The special axioms

```
CASEINTRO: ?x = ?p || ?x , ?x

EQUATION: (?a = ?b) = (?a = ?b) || true , false
```

which are provided in the logical preamble supplied by Watson to every theory, are used to introduce new case expressions.

It should be noted here that the fact that the meaning of built-in tactics referring to hypotheses of case expressions is context dependent necessitates a complication of the definition of substitution: when an expression containing such tactics is substituted into a context, it may be necessary for some of these tactics to be renumbered. The full definition of substitution will be given below.

3.3.1. *Examples of reasoning about case expressions*

```
- s "(?a=?b)||((?c=?b)||(?a+?b),?x),?y";
- right();left();right();left();left();    (* navigate
                                              to term ?a *)

(?a = ?b) || ((?c = ?b) || ({?a} + ?b) , ?x) , ?y
?a                                        (* the display *)

- lookhyps();     (* view locally relevant hypotheses *)
```

```
(* hypotheses displayed *)

1 (positive):
?a  =
?b

2 (positive):
?c  =
?b

(* end of hypothesis display *)

- ri "0|-|1"; ex();

(?a = ?b) || ((?c = ?b) || ({?b} + ?b) , ?x) , ?y
?b                                     (* the display *)

- rri "0|-|2"; ex();         (* apply second hypothesis
                             in converse sense *)

(?a = ?b) || ((?c = ?b) || ({?c} + ?b) , ?x) , ?y
?c                                     (* the display *)

- rri "1|-|1"; ex();   (* introduce new case expression
                               with first hypothesis *)

(?a = ?b) || ((?c = ?b) || ({(?a = ?b) || ?c , ?x_3}
      + ?b) , ?x) , ?y                 (* the display *)

(?a = ?b) || ?c , ?x_3

- ri "1|-|1"; ex();

(?a = ?b) || ((?c = ?b) || ({?c} + ?b) , ?x) , ?y
?c                                     (* the display *)

- up(); up(); right();

(?a = ?b) || ((?c = ?b) || (?c + ?b) , {?x}) , ?y
?x                                     (* the display *)

- lookhyps();
(* hypotheses displayed *)
```

```
1 (positive):                (* note change of sense of
                                   second hypothesis *)
?a  =
?b

2 (negative):
?c  =
?b

(* end of hypothesis display *)

- rri "(2|-|2)@0"; ex();          (* introduce new case
                       expression stipulating the value
                             to go in the new branch *)

(?a = ?b) || ((?c = ?b) || (?c + ?b) , {(?c = ?b)
      || 0 , ?x}) , ?y

(?c = ?b) || 0 , ?x

(* sample definitions of propositional connectives *)

- defineinfix "NOT" "~?x" "?x||false,true";

NOT:
~ ?x  =
?x || false , true
NOT , 0

- defineinfix "AND" "?x&?y"
                        "?x||(?y||true,false),false";

AND:
?x & ?y  =
?x || (?y || true , false) , false
AND , 0
```

3.4. Implementing stratified abstraction in Watson

The most complex built-in functions of Watson are those which support stratified abstraction.

3.4.1. *The handling of variable binding: de Bruijn levels and the formal definition of substitution*

As we noted above, the term construction using brackets represents λ-abstraction. There is nothing in Watson's notation corresponding to the λx component of the notation $(\lambda x.T)$; the variable bound in a bracket is determined by the syntax of its context using a scheme due to deBruijn (but differing from the most familiar name-free binding scheme due to deBruijn).

The variable bound in an outermost set of brackets is always `?1`; the variable bound in a set of brackets which is in the scope of one set of brackets is `?2`; in general, the variable bound in a set of brackets which is enclosed in `n-1` further brackets is `?n`. For example, the constant function $(\lambda x.x)$ is written `[?1]`, but its constant function $(\lambda y.(\lambda x.x))$ is written `[[?2]]` (because the bound variable in this term is bound by the inner of the two sets of brackets).

This scheme (which we have seen referred to as "de Bruijn levels") has two advantages. The first advantage is that it is not necessary to manage binding with arbitrary variables, which leads to a complex implementation and a very complex definition of substitution. The advantage that this scheme has over the more popular of de Bruijn's schemes (de Bruijn indices) is that the variable bound in a given bracket is represented in the same way wherever it appears. The disadvantage of de Bruijn levels (which de Bruijn indices do not have) is that the bound variables in a term may need to be renumbered when it is substituted into a different term. So far, users of Watson (and Mark2) have found the de Bruijn level scheme to be usable in practice; what pressure the designer has felt from users to convert to the usual variable-binding scheme has been relieved by the availability of a tactic provided in the libraries which readily converts bracket terms to a form more like the usual form when nesting of brackets is sufficient to cause confusion.

The use of de Bruijn levels creates a complication of the definition of substitution in a much more pervasive way than the similar problem that arises with the numbering of hypothesis tactics. On the other hand, the complication is simpler than the one that would be occasioned by use of the usual variable binding schemes. For users the system adopted is certainly better than de Bruijn indices, but admittedly less readable than the usual schemes of variable binding when there is enough nesting of abstraction terms (but, as noted above, there is a tactic which converts such terms to a more readable form).

When we consider an occurrence of a subterm in a larger term, we define its *level* as the number of bracket terms of which it is a proper subterm, and its *hlevel* (for "hypothesis level") as the number of branches of case expressions of which it is a (not necessarily proper) subterm.

The semantics of bound variables requires that each bound variable `?n` appear only in contexts contained in at least `n` bracket terms (the "bound variable" `?0` can appear in any context; it is used for special purposes by internal functions of the prover not discussed in this paper). This is enforced by the declaration checking functions of the prover. An analogous requirement could be imposed on hypothesis tactics, but is not in practice: no harm can be done by meaningless hypothesis tactics, and we are interested in allowing natural numbers to be passed as parameters to the `|-|` operator in tactics, which would be forbidden if declaration checking of hypothesis tactics were to be enforced.

We define what it means (and under what circumstances it is possible) to substitute a term `T` found at level l_1 and hlevel h_1 (in some larger term) into a context found at level l_2 and hlevel h_2 in a term `U`. One circumstance under which such a substitution is impossible is that in which a variable free in the term `T` becomes bound in its new context when `T` is substituted into `U`. A bound variable `?n` is free in `T` iff $n \leq l_1$; it becomes bound (or meaningless) in its new context in `U` if $n > l_2$; thus, substitution is impossible if any bound variable `?n` appears in `T` with $l_2 < n \leq l_1$. A similar restriction is imposed on indices of hypothesis tactics in relation to hlevel. Each variable bound in `T` (i.e., `?n` with $n > l_1$) is enclosed by n brackets in the term of which `T` is a subterm, while it is enclosed in $n - l_1 + l_2$ brackets in the term `U`; thus its index must be changed from n to $n - l_1 + l_2$ (while the indices of locally free bound variables remain the same). Precisely analogously, hypothesis tactic indices $> h_1$ are translated by $h_2 - h_1$.

The definition of substitution used by Watson is somewhat further complicated by the issue of "higher-order matching" discussed below.

3.4.2. *Abstraction and reduction by built-in tactics*

The semantic interpretation of bracket terms as functions is enforced by the three built-in tactics `BIND`, `EVAL` and `UNEVAL`.

`BIND` takes a parameter: when `BIND @ T` is applied to a term `U` at level l, the effect is to translate `U` to level $l + 1$, replace all occurrences of the result of translating `T` to level $l + 1$ in the translation of `U` by occurrences of `?[l+1]` (we hope that this is a pardonable abuse of notation), enclose the result in brackets and apply it (using `@` to represent function application) to `T`; this process only succeeds if the bracket term obtained is stratified (otherwise `U` is unchanged). In terms of λ-notation, the result can be described as $(\lambda x.U[x/T])(T)$.

`EVAL` takes no parameter; when `EVAL` is applied to a term of the form `[T] @ U` at level l, the effect is to translate `U` to level $l + 1$, replace ?[l+1] with the translated form of U in T, and translate this modified form of T from level ?[l+1] to level l; this modified form of T is the result. Mathematically, this is simply β-reduction, and it always succeeds.

UNEVAL is a little more esoteric, and it took some experience to realize that it was needed. When UNEVAL @ [T] is applied to U, the result is a term of the form [T] @ X precisely if there is such a term which would reduce to U on application of EVAL. The effect of UNEVAL is to rewrite an expression as a value of the function given as its parameter if this is possible.

The project of providing built-in support to synthetic abstraction and reduction algorithms found in the original prover EFTTP (as discussed in our reference (??)) and still supported by little-used features of Mark2 has been abandoned. Synthetic abstraction and reduction algorithms are straightforward to implement in the tactic language of Mark2 or Watson. We are still carrying out investigations in this area using the Watson tactic language as a tool. But we no longer see it as reasonable to support this in the prover's built-in logic.

3.4.3. *Examples of the implementation of abstraction in Watson*

```
- s "?x";

{?x}
?x

- ri "BIND@?x"; ex();

{[?1] @ ?x}          (* [?1] is the identity function *)
[?1] @ ?x

- ri "EVAL"; ex();     (* evaluate the function *)

{?x}
?x

- s "[?1]";

{[?1]}
[?1]
```

```
- ri "BIND@?y"; ex();

{[[?2]] @ ?y}        (* [[?2]] is the constant function
 [[?2]] @ ?y         whose value is [?1];
                     this is an example of
                     relabelling of bound variables *)

- ri "EVAL"; ex();

{[?1]}
[?1]

(* abstraction does not need to be
with respect to atomic terms *)

- s "(?x+?y)=?x+?y";

{(?x + ?y) = ?x + ?y}
(?x + ?y) = ?x + ?y

- ri "BIND@?x+?y"; ex();

{[?1 = ?1] @ ?x + ?y}
[?1 = ?1] @ ?x + ?y

s "3+3";

{3 + 3}
3 + 3

- ri "UNEVAL@[?1+?1]"; ex();

{[?1 + ?1] @ 3}
[?1 + ?1] @ 3

(* definition of the universal quantifier *)

(* a full account of definition commands is given
in a later section *)
```

```
- defineconstant "forall@?x" "?x=[true]";

forall:
forall @ ?x  =
?x = [true]
forall , 0

(* definition of the existential quantifier *)

- defineconstant "forsome@?x" "~forall@[~?x@?1]";

forsome:
forsome @ ?x  =
~ forall @ [~ ?x @ ?1]
forsome , 0
```

3.4.4. *The implementation of stratification in Watson: relative type and opacity*

In this subsection, we discuss the implementation of the notion of stratification under Watson. Each operator in Watson must be declared with a "left type" and a "right type" which determine the displacement of the relative types of the left and right subterms of a term built with that operator from the relative type of the term. A strict prefix operator only needs a right type.

An alternative is to declare an operator as "opaque": abstraction into a term built with an opaque operator is not permitted, so such a term may not contain any variable bound in an abstraction term which includes it. In other words, opaque operators may be used in the definitions of functions only to construct constants. Variable operators may be declared opaque if it is desired (as in structural tactics such as `RIGHT` above) that the variable operator match operators with arbitrary relative types. Variable operators can also be declared with specific relative types. Undeclared variable operators are treated as opaque. This is the main application of opaque operators in existing theories, though there are sensible applications of "constant" opaque operators in mathematics, and it turns out that defined operators used as type constructors (for types represented as retractions onto strongly Cantorian domains) need to be opaque.

The assignment of relative types to occurrences of subterms of a term then proceeds as follows: the relative types of the left and right subterms of a term built with an infix operator are obtained by adding the left and right types (respectively) to the type of the whole subterm; the right subterm of a term built with a strict prefix operator is handled in the same way. The immediate proper subterm of an abstraction term has relative type one lower than the abstraction term, and the immediate subterm must have the same relative type as all occurrences in the abstraction term of the bound variable bound in the abstraction term (this is the condition of stratification)[2].

3.4.5. *Examples of the implementation of stratification*

```
- s "?x@?x";

{?x @ ?x}
?x @ ?x

- ri "BIND@?x"; ex();

{?x @ ?x}                     (* the function [?1@?1]
?x @ ?x                       violates stratification
                              restrictions *)


- s "[?1@?1]";

Watson:  Term is not stratified

- s "[[?1]]";                 (* this is the ``function''
                              which sends objects to their
                              constant functions
                              (the K combinator); it cannot
                              be typed in this theory *)

Watson:  Term is not stratified

- s "(?f@?x)+?g@?x";

{(?f @ ?x) + ?g @ ?x}
(?f @ ?x) + ?g @ ?x
```

[2] The latest versions of Watson allow the formation of unstratified abstraction terms for special purposes: in these versions, the entry of an unstratified abstraction term will not cause a declaration error, but the examples in the next subsection should otherwise run similarly

```
- ri "BIND@?x"; ex();

{[(?f @ ?1) + ?g @ ?1] @ ?x}
[(?f @ ?1) + ?g @ ?1] @ ?x

- left();ri "BIND@?g";ex();

{[[(?f @ ?2) + ?1 @ ?2]] @ ?g} @ ?x
[[(?f @ ?2) + ?1 @ ?2]] @ ?g

- left(); ri "BIND@?f"; ex();

(* stratification restrictions prevent this
from working; the problem is that ?f and ?g
are at the same relative type *)

({[[(?f @ ?2) + ?1 @ ?2]]} @ ?g) @ ?x
[[(?f @ ?2) + ?1 @ ?2]]

(* we show how to achieve a function with ?f and ?g
as parameters *)

- startover();

{(?f @ ?x) + ?g @ ?x}
(?f @ ?x) + ?g @ ?x

- assign "?f,?g" "(P1@?F),P2@?F";

(* the assign command will carry out
assignments based on matches of complex terms *)

(* P1 and P2 are projection operators for the pair
represented by the comma operator; their defining
axioms are predeclared *)

{((P1 @ ?F) @ ?x) + (P2 @ ?F) @ ?x}
((P1 @ ?F) @ ?x) + (P2 @ ?F) @ ?x

- ri "BIND@?x"; ex();

{[((P1 @ ?F) @ ?1) + (P2 @ ?F) @ ?1] @ ?x}
[((P1 @ ?F) @ ?1) + (P2 @ ?F) @ ?1] @ ?x
```

```
- left(); ri "BIND@?F"; ex();

(* the abstract on the left is a pure abstract
which handles addition of functions *)

{[[((P1 @ ?1) @ ?2) + (P2 @ ?1) @ ?2]] @ ?F} @ ?x
[[((P1 @ ?1) @ ?2) + (P2 @ ?1) @ ?2]] @ ?F

(* we demonstrate application
and evaluation of this abstract *)

- assign "?F" "?f,?g";

{[[((P1 @ ?1) @ ?2) + (P2 @ ?1) @ ?2]] @ ?f , ?g}
@ ?x

[[((P1 @ ?1) @ ?2) + (P2 @ ?1) @ ?2]] @ ?f , ?g

- ri "EVAL"; ex();

{EVAL => [[((P1 @ ?1) @ ?2) + (P2 @ ?1) @ ?2]] @ ?f
   , ?g} @ ?x
EVAL => [[((P1 @ ?1) @ ?2) + (P2 @ ?1) @ ?2]] @ ?f
, ?g

- left();left();left();

[({P1 @ ?f , ?g} @ ?1) + (P2 @ ?f , ?g) @ ?1] @ ?x
P1 @ ?f , ?g

- ri "P1"; ex();

[({?f} @ ?1) + (P2 @ ?f , ?g) @ ?1] @ ?x
?f

- up();up();right();left();

[(?f @ ?1) + {P2 @ ?f , ?g} @ ?1] @ ?x
P2 @ ?f , ?g

- ri "P2"; ex();
```

```
[(?f @ ?1) + {?g} @ ?1] @ ?x
?g

- top(); ri "EVAL"; ex();

{(?f @ ?x) + ?g @ ?x}
(?f @ ?x) + ?g @ ?x

(* Here's something which ought to be allowed -- we'll
show how to fix this in the next section *)

- s "forall@[forsome@[?1=?2]]";

Watson:  Term is not stratified

(* the term above represents the sentence ``for all x,
for some y, x = y'', which certainly ought to be
meaningful (and, as we will see below, Watson will
treat this as stratified, given further information) *)
```

3.4.6. *Support for strongly cantorian domains*

Strongly cantorian domains are supported in two ways by Watson.

The explicit support for strongly Cantorian domains is bound up with the properties of the built-in operator `:`, which signals the application of a retraction with strongly Cantorian range. A predeclared axiom `TYPES: (?t:?t:?x) = ?t:?x` ensures that type labels have the effect of retractions. The stratification facility will raise and lower the relative type of the right subterm of a term of the form `?t:?x` as needed to stratify the context. It is not the case that all relative type information from such a subterm is suppressed: the difference between the relative types of two bound variables may be determined by information internal to such a term, and will cause a stratification error if it conflicts with other information derived in attempting to stratify the term.

Implicit support for strongly Cantorian domains is provided by the "scin/scout" features of the prover. An operator may be declared "scout" (for "strongly Cantorian output") if a theorem has been proved to the effect that the output of the operator will belong to a fixed strongly Cantorian domain. Similarly, an operator may be declared "scin" (for "strongly Cantorian input") if a theorem has been proved to the effect that all of its (one or two) inputs will belong to fixed strongly Cantorian domains. A term which is scout will have its type raised or lowered if necessary for purposes of stratification; a term which is scin will have the type(s) of its immediate subterms raised or lowered (independently of one another if there are two of them) if necessary for purposes of stratification. Functions may also be declared scin or scout.

These two features together greatly reduce the rigidity of the stratification criterion of Watson; in particular, the declaration of the propositional connectives and quantifiers as scin or scout as appropriate enables the fluent handling of first-order logic (avoiding stratification difficulties illustrated in an example above).

3.4.7. *Examples of the implementation of strongly cantorian domains*

```
- declareconstant "bool";

(* attaching a type label to the inner quantified
statement in the term with nested quantifiers that
caused problems above causes the prover to be able
to stratify it *)

- s "forall@[bool:forsome@[?1=?2]]";

(* the bool type label can be defined in such a way
as to make the following "axioms" easily proved
theorems *)

- axiom "FORALLBOOL" "forall@?x" "bool:forall@?x";

- axiom "FORSOMEBOOL" "forsome@?x" "bool:forsome@?x";

(* the theorems FORALLBOOL, FORSOMEBOOL witness the
fact that the functions forall, forsome have output
of boolean type, thus are "scout" *)

- makescout "forall" "FORALLBOOL";
- makescout "forsome" "FORSOMEBOOL";

- s "forall@[forsome@[?1=?2]]";

(* the information that forall and forsome are scout
is enough to allow this term to be stratified; it
removes any problem of stratification due to nested
quantifiers *)

{forall @ [forsome @ [?1 = ?2]]}  (* no error message! *)
forall @ [forsome @ [?1 = ?2]]
```

3.4.8. *Definitions in Watson*

Constants, operators and "type labels" (retractions with strongly cantorian range) can be defined in Watson. Examples of the definition commands have appeared in example sections above, where this seemed appropriate.

The definition of an atomic constant as a complex term is straightforward:

```
- defineconstant "four" "2+2";

four:
four  =
2 + 2
four , 0
```

The prover checks for defects such as circularity in the proposed definition and, if no error is found, proves a new theorem embodying the definition. When a constant is defined, the name of the new theorem is the same as the name of the defined constant (which cannot have been declared previously).

The issue of stratification arises in the definitions of functions and operations.

```
- defineconstant "Double@?x" "?x+?x";

Double:
Double @ ?x  =
?x + ?x
Double , 0

- defineconstant "(Comp@?f,?g)@?x" "?f@?g@?x";

Comp:
(Comp @ ?f , ?g) @ ?x  =
?f @ ?g @ ?x
Comp , 0

- defineconstant "Comp2@?f,?g,?x" "?f@?g@?x";

Watson:  Format, declaration or stratification failure of
proposed definition of Comp2@?f,?g,?x
```

While reading the two definitions of composition of functions, recall that all operations have the same precedence and group to the right.

In the first definition of composition of functions, note that defined functions can have lists of arguments and can be “curried” if the types of their arguments warrant this. In the second definition of composition, we see a failure of stratification.

Operators can be defined similarly.

```
(* defining operators with ‘‘flat’’ type *)

- defineinfix "OR" "?x|?y" "~(~?x)& ~?y";

OR:
?x | ?y  =
~ (~ ?x) & ~ ?y
OR , 0

(* defining typed operators *)

- showdec "@";                (* show the declaration of
                              the function application
                              operator *)

Watson:  Reserved operator @ left type:  1 right type: 0

- definetypedinfix "CONV_APP" 0 1 "?x <@ ?y" "?y @ ?x";

(* the integer parameters are the left and right types
of the defined operator <@ (the converse of application,
which has left type 1 and right type 0, must have
left type 0 and right type 1) *)

CONV_APP:
?x <@ ?y  =
?y @ ?x
CONV_APP , 0

(* definition of opaque operators *)

- defineopaque "WEIRD" "?x +‘? ?y" "(?x@?x)+?y@?y";

WEIRD:
?x +‘? ?y  =
(?x @ ?x) + ?y @ ?y
WEIRD , 0
```

The name of the theorem defining an operator must be supplied as a parameter to the `defineinfix` or `definetypedinfix` commands, as we see in the examples. The `defineinfix` command is used to define “flat” infixes (those with left and right type of 0); the `definetypedinfix` command is used to define infixes with nontrivial left and right type, and takes the types of its arguments as parameters. Completely unstratified operations such as `+‘?` in the example above can also be defined as “opaque” operators; the use of opaque operators in definitions and in contexts with bound variables is extremely restricted, but they do have some applications.

The most complex form of definition in Watson is the facility of defining “type labels” (retractions onto strongly cantorian ranges). The usual restrictions of non-circularity and stratification in definitions of functions and operators are first modified (the stratification requirements are sharper) then extended with the requirement that the operation being defined be shown to be a retraction. We illustrate the definition of a type by defining the type of booleans (using a new identifier Bool to avoid conflict with the type label bool declared above).

```
- start "?x||true,false";
- assign "?x" "?x||true,false";  (* sets up theorem
                            that this operation is a
                                       retraction *)
- ri "(!@CASEINTRO)@?x"; ex();
- top(); downtoleft "?x||?y,?z"; ri "1|-|1"; ex();
- top(); downtoright "?x||?y,?z"; ri "1|-|1"; ex();
- top(); ex();
- p "BOOLRETRACT";

BOOLRETRACT:
(?x || true , false) || true , false  =
?x || true , false
CASEINTRO , 0

- defineconstanttype "BOOLRETRACT"
                        "Bool:?x" "?x||true,false";

Bool:
Bool : ?x  =
?x || true , false
Bool , CASEINTRO , 0
```

The thing to notice is that the defineconstanttype command takes as an additional parameter a theorem witnessing the fact that the operation being defined is a retraction. The reason that Watson can tell that this retraction has strongly cantorian domain has to do with technical aspects of the way Watson determines whether case expressions are stratified (it implicitly views the hypothesis of a case expression as being restricted to a strongly cantorian domain).

It is possible to define functions and operations which generate type labels (type constructors). There are subtleties with the use of type constructors which make it necessary to declare constructors as "opaque" operators.

3.4.9. *Fine points of matching: limited higher-order matching and commutative matching*

This section is devoted to two refinements of matching, one integral to Watson and the other optional.

A theorem in the current Watson library is the unsurprising

```
FORALLDIST:

forall @ [(?P @ ?1) & ?Q @ ?1]  =
(forall @ [?P @ ?1]) & forall @ [?Q @ ?1]
```

A term to which this theorem ought to apply is `forall@[(?1=?1)&?1=?1]`; application of the theorem should yield `(forall@[?1=?1])&forall@[?1=?1]`. And indeed it does:

```
- s "forall@[(?1=?1)&?1=?1]";

{forall @ [(?1 = ?1) & ?1 = ?1]}
forall @ [(?1 = ?1) & ?1 = ?1]

- ri "FORALLDIST"; ex();

{(forall @ [?1 = ?1]) & forall @ [?1 = ?1]}
(forall @ [?1 = ?1]) & forall @ [?1 = ?1]
```

However, this would not have worked in early versions of Mark2 (the precursor of the present Watson prover), where this proof would have taken the following form:

```
- s "forall@[(?1=?1)&?1=?1]";

{forall @ [(?1 = ?1) & ?1 = ?1]}
forall @ [(?1 = ?1) & ?1 = ?1]

- right();right();

forall @ [{(?1 = ?1) & ?1 = ?1}]
(?1 = ?1) & ?1 = ?1

- right(); ri "BIND@?1"; up(); left(); ri "BIND@?1"; ex();

forall @ [{([?2 = ?2] @ ?1) & [?2 = ?2] @ ?1}]
([?2 = ?2] @ ?1) & [?2 = ?2] @ ?1

- top(); ri "FORALLDIST"; ex();

{(forall @ [[?2 = ?2] @ ?1]) & forall @ [[?2 = ?2]
         @ ?1]}

(forall @ [[?2 = ?2] @ ?1]) & forall @ [[?2 = ?2]
      @ ?1]

- left();right();right();ri "EVAL";
top();right();right();right();ri "EVAL"; top(); ex();

{(forall @ [?1 = ?1]) & forall @ [?1 = ?1]}
(forall @ [?1 = ?1]) & forall @ [?1 = ?1]
```

The difficulty with the old version of the prover (and with the intuition that leads one to believe that the application shown is a direct application of the theorem) is that `?P@?1` and `?Q@?1` are each expected to match `?1=?1`, which they do not in any obvious syntactical sense. This was a serious problem for fluent reasoning with quantifiers (and also caused unnecessary work in other applications of functions). (It is also important to note that `forall@[?P&?Q]` is not a candidate to match `forall@[(?1=?1)&?1=?1]` at all, because `[?P&?Q]` is a constant function; the variables `?P` and `?Q` can only match terms not depending on `?1` at all).

The solution was to adopt a limited form of higher-order matching: in the context where `?n` is the highest-numbered bound variable, the expression `?P@?n`, where `?P` is a free variable, will match any expression in `?n` which can be expressed as a function of `?n` (stratification restrictions are at work here): the free variable `?P` will be matched to a suitable abstraction term. A modification of substitution which causes reductions to occur in terms of the form `?P@?n` when an abstract is substituted for `?P` is also needed. With these modifications of matching and substitution, we get the natural behaviour illustrated in the first example above.

Recently, we have implemented further refinements of higher-order matching and correlated improvements in substitution, which allow natural treatments of situations involving multiple bound variables as arguments, both when the arguments are curried and when they occur in tuples.

A further point is that new variables introduced by the application of theorems or tactics inside bracket terms may have an unexpected form: under certain conditions, a "new variable" introduced at level n will take the form `?x_1 @ ?n` rather than the form `?x_1` which one expects. It is better to avoid the use of theorems which introduce new variables in such contexts (in fact, it is always better to avoid the introduction of new variables except in interactive "scratch work"); it is possible that a stratification error will result from such a theorem application (the type checking done uses an imperfect heuristic to guess whether the new variable has correct relative type); such errors are caught by the prover and cause the theorem application to fail, and can always be avoided by avoiding the introduction of new variables (by using parameterized versions of theorems and their converses, either user developed or constructed using the `!@` and `!$` built-in operators). It would be possible to redesign Watson to avoid stratification errors of this kind, but it would involve a considerable elaboration of the data structures representing context information in the prover for a very limited benefit.

An optional refinement of matching which is available is commutative matching, which can be turned on (or off) with the `cmatch` command. With commutative matching on, the prover will in effect attempt to apply commutative laws wherever possible in the attempt to apply a theorem. A simple example:

```
- s "?x+?y+?z";

{?x + ?y + ?z}
?x + ?y + ?z
```

```
- ri "ASSOC"; ex();      (* ASSOC does not apply *)

{?x + ?y + ?z}
?x + ?y + ?z

cmatch();  (* this turns on commutative matching *)

- ri "ASSOC"; ex();       (* an application of COMM
                          gives a term to
                          which ASSOC can be applied *)

{?y + ?z + ?x}
?y + ?z + ?x
```

An important fact about commutative matching is that it is not applied in certain contexts even when it is turned on: for example, the alternative and on-success rule infixes use strict matching to determine whether a theorem succeeds or fails. The additional freedom of commutative matching would disrupt the control structures of the internal programming language too much (for example, the generalized associativity tactic given above would fail, because it depends on `ASSOC` to fail on sums whose left term is not a sum).

We are not convinced that commutative matching is a useful refinement of the prover (largely because of its conflicts with the needs of the tactic language), but we continue to experiment with it.

3.4.10. *Special effects*

We discuss some miscellaneous built-in tactics.

`INPUT` and `OUTPUT` allow interaction between a tactic and the user. The `INPUT` command displays the local hypotheses and the selected subterm; the prover then waits for the user to enter a theorem or tactic, which will then be applied. If the user hits return, the prover will simply continue. The `OUTPUT` command simply displays its term parameter. The `INPUT` command has been used in the development of a goal-driven natural deduction prover as a family of Watson tactics.

The `FLIP` command takes a parameter, intended to be the name of a theorem (in fact, intended to be a "commutative law", though the prover does not enforce this). If `FLIP @ ?thm` is applied to an infix term the displayed forms of whose arguments are not in lexicographic order, the theorem `?thm` is applied; otherwise nothing is done. The intention is to support the ability to define canonical forms for expressions built with commutative operators, by sorting terms.

4. Experience with the prover

4.1. Experiences of users

The Watson prover has a shallow initial learning curve. Undergraduate students can learn to prove equational theorems in an algebraic style with the prover quite rapidly.

The more complex case expression and stratified abstraction facilities seem also to be accessible to the advanced undergraduate and graduate math and CS students who have been users of the system.

The use of the tactic language to write complex recursive tactics has been mostly confined to the designer and one student. All users have been able to make effective use of tactics provided in libraries.

Users do not seem to have trouble with the default order of operations of the prover, but we have provided full support for conventional order of operations for applications (such as those in education) where the use of conventional order of operations would be advisable.

Users do not seem to have trouble with the use of deBruijn levels in place of conventional variable binding; those who have done reasoning with quantifiers or higher-order functions seem to have adapted to this. Since de Bruijn levels are a notational variant on the usual treatment of bound variables (we believe it would be harder to adapt to deBruijn indices), we don't find this surprising. A tactic is available in the libraries which makes the notation for abstracts look more like standard notation: this tactic decorates each bracket term with a null occurrence of the variable bound in it at its head (e.g., `[?1 + ?1] becomes [?1.?1 + ?1]`, where the defining theorem of the connective `.` is `?x.?y = ?y`. Another tactic removes the annotations supplied by the first tactic.

The tactic language has been useful in several contexts. A tautology-checking tactic has been used extensively. Recently, we have implemented type inference algorithms as tactics in a theory of natural numbers, integers, rationals and reals in which manipulation of type labels (retractions with strongly cantorian range) was becoming burdensome. The use of tactics for application of theorems to subterms of the selected term, composition and conversion of tactics, and global rewriting (of more than one variety) is ubiquitous.

Watson theories can be saved and retrieved in two different ways. If one is only interested in the theorems in a theory, there are `storeall` and `load` commands which respectively store and retrieve the theorems of a given theory. If one is interested in the details of proofs (especially if one needs to change a theory by modifying the details of proofs) the correct approach is to develop a proof script in a text editor. Originally, proof scripts were run under ML using its `use` command, but this was too slow. The prover now has its own `script` command which reads scripts (which are also usually readable by the ML interpreter). Scripts can be nested in a way which supports the development of libraries of theories, though the modularity provided is not very sophisticated.

A refinement we have considered but not implemented (due to the ease of use of proof scripts) is the development and storage of "proof objects" which would represent proofs in a way which could be manipulated by prover functions rather than a text editor. There is a natural way to do this in Watson (proofs could be represented as tactics) but there is as yet little need for this.

Multiple theorems under construction in multiple theories can be stored on the desktop at the same time in a Watson session; this facilitates development work.

There is a facility, little-used so far, which can be used to prove theorems in one theory by analogy with theorems from another: if the user provides a match between primitives and axioms of a source theory and concepts and theorems of a target theory, theorems of the source theory can be exported to the target theory (with automatic translation of notions of the source theory to notions of the target theory as needed). This theorem export facility also allows the export of tactics; if a tactic is exported, all of the subtactics it needs will automatically be exported as well. We believe that this facility of the prover will become more important as mathematical libraries grow more extensive. We have used it in one place in the publicly available libraries to transfer a proof of complete induction and related principles between theories with conflicting notation.

4.2. Experience with mathematical content areas

The Watson prover has been used to develop theories of propositional and predicate logic in several styles. An equational style (as exemplified in (??)) is most natural, and theories implementing the propositional and predicate logic chapters of (??) in detail exist. We have also implemented a sequent calculus and a goal driven natural deduction prover. A tableau approach to first-order logic has natural relationships with Watson's built-in logic of case expressions, and a tableau approach to propositional logic has been used to develop a complete tautology checker as a tactic. A goal-directed natural deduction prover has been implemented as an interactive tactic using the `INPUT` command.

Watson's notation for propositional logic presents no difficulties. Experience leads us to believe that users can readily adapt to the somewhat nonstandard notation for quantifiers. There are few problems with the stratification functions of the prover as long as the boolean type has been declared as strongly cantorian and the propositional connectives and quantifiers made scin or scout as appropriate.

The prover has been used to develop theories of basic arithmetic and algebra needed for the projected computer science applications. Watson readily supports algebraic reasoning; its case expression machinery allows a natural way to handle exceptions to algebraic rules (such as those occasioned by the need to avoid division by zero) which are usually ignored by computer algebra systems, making the typical CAS logically unsound. We have also shown that Watson is a practical environment for carrying out proofs by induction in a development of Peano arithmetic. A theory of program semantics based on (??) is under construction. A student has recently completed a master's project involving the verification of published proofs of security properties of communication protocols using Watson.

For example, students working on this project have been able to develop the following theory files (this list is not exhaustive):

Sets: Sets are defined as abstractions from boolean terms (predicates), and set membership as a specialization of function application. A library of elementary theorems has been developed covering the topics of set membership, equivalence, complements, unions, intersections, differences, subsets, proper subsets, and disjoint sets. On a more purely mathematical note, the designer used the student work described above in a proof of the Schröder-Bernstein theorem.

NatOrder: A theory file related to the order properties of natural numbers (requiring proofs by induction).

Bit Values: A theory file that contains definitions and theorems to assist in the mapping between compositions of Boolean values (bit strings) and natural numbers to be used in hardware circuit specification and verification.

Gates: A theory of basic gates, including and, or, xor, inv, nand, nor and multiplexors. This also includes a theory covering boolean algebra for the basic logic gates. Specifically this file focuses on the composition and comparison of basic gates including associativity and symmetry laws as well as De Morgan laws for the gates.

Protocols: With help from the authors, a student has developed a theory of protocol specifications and verification based on the CAPSL specification language (??) and the SvO logic (??). An interesting aspect of this project was that it required the implementation of a modal logic (of belief) under Watson, which proved possible despite the designer's doubts.

Program Semantics: The designer and a student have been collaborating on a file which implements a theory of program semantics in the style promoted by Dijkstra and Gries, following (??). This file has involved extensive use of the higher-order features of the prover, since reasoning about complex function types is involved.

The designer has developed a theory of synthetic combinatory logic, with a variety of different abstraction and reduction tactics, which he hopes to use as a research tool in collaboration with a specialist in this area. The fact that the prover is good for this application area is not surprising, as the development of the control structures of the tactic language was largely driven by the now abandoned attempt to use a synthetic combinatory logic rather than a λ-calculus as the logic of the prover.

5. Related Work

There are dozens of theorem provers available and in use in the world[3]. The majority of these are based on classical predicate or propositional logic. However, there are only a few systems, including Watson, that have the power to perform reasoning in a higher-order logic. In addition some provers are interactive like Watson whereas others are much more automated. It is appropriate to limit detailed consideration to interactive higher-order logic systems in this section due to space limitations of this paper. A complete treatment would compare the propositional, predicate, tableau and other capabilities of Watson with respect to systems that support those styles of reasoning.

In our laboratory we have passing experience with several theorem proving systems, but have really focused our efforts on HOL (??; ??; ??) and Watson. In this section we first provide the mathematical comparison between Watson and other systems and then provide a more detailed comparison to the HOL system.

5.1. Mathematical Comparison

The designer gained his first appreciation of interactive theorem proving from reading about Nuprl ((??)) but there is no obvious genetic relationship between Watson and Nuprl.

[3] A comprehensive list of theorem provers can be found at http://www-formal.stanford.edu/clt/ARS/systems.html.

The prover work would have been impossible without the availability of the programming language Standard ML ((??)), which itself is a byproduct of the development of the Edinburgh LCF theorem prover ((??)).

Watson is contrasted with such provers as Automath ((??)) or Nuprl ((??)) by the use of classical rather than constructive logic. Overall, Watson is probably most similar to HOL ((??; ??; ??)) in its outlook on things (classical higher-order logic) but it is still very different from HOL.

Watson is mostly devoted to manipulation of terms/expressions rather than to propositions (there is some treatment of propositions rather than terms in the built-in mechanisms for handling case expressions), and in this way is like rewrite rule systems (such as RRL ((??))). But it is once again remote from actual rewrite rule based provers; for example, the Knuth-Bendix algorithm ((??)) has no application to Watson so far (though it might be interesting to develop an interface between Watson and an implementation of Knuth-Bendix which would serve as a tactic generator).

Watson may share some "look and feel" with computer algebra systems (CAS) like Maple ((??)) and Mathematica ((??)); it looks more like a system for calculation than do most theorem provers. An interesting project would be to implement a baby CAS as a package of tactics in Watson; some work like this has been done with earlier versions of the system. Watson is much smaller than a CAS (much more is left for users to do) and is also more demanding than a CAS (computer algebra systems are not in general logically sound).

Watson is not really at all comparable to systems designed for fully automated proof like Otter ((??)) and the Boyer-Moore prover ((??)). Otter was the vehicle for the only other work we know of in which automated reasoning software has been applied to systems related to Quine's New Foundations ((??)). In principle, tactics written in Watson could effectively become automatic theorem provers (the existing tautology checker might fall into this category) but real work along these lines would require that the tactic language of Watson be given a much more efficient implementation.

Watson is distinctive in its use of an untyped logic (though some type inference is supported), in its approach to rewriting and the details of its tactic language, and in the application of the logical properties of expressions defined by cases (via a system of conditional rewriting).

5.2. HOL

HOL is a general theorem proving system developed at the University of Cambridge (??; ??; ??) that is based on Church's theory of simple types, or higher-order logic (??). Although Church developed higher-order logic as a foundation for mathematics, it can be used for reasoning about computational systems of all kinds. Similar to predicate logic in allowing quantification over variables, higher-order logic also allows quantification over predicates and functions, thus permitting more general systems to be described.

Watson also has a relationship to Church's simple theory of types: its logic can be obtained by using polymorphism to collapse a subset of Church's type system.

Like Watson, HOL is not a fully automated theorem prover but is more than simply a proof checker, falling somewhere between these two extremes. HOL has several features that contribute to its use as a verification environment:

1. Several built–in theories, including booleans, individuals, numbers, products, sums, lists, and trees. These theories build on the five axioms that form the basis of higher-order logic to derive a large number of theorems that follow from them. In Watson, these theories could be built-in for efficiency purposes, however to date we have not chosen to do that. Watson does provide some basic primitive operations for booleans and natural numbers, with additional types being considered.

2. Rules of inference for higher-order logic. These rules contain not only the eight basic rules of inference from higher–order logic, but also a large body of *derived* inference rules that allow proofs to proceed using larger steps. The HOL system has rules that implement the standard introduction and elimination rules for Predicate Calculus as well as specialized rules for rewriting terms. This is similar to some of the built-in rules in Watson used for equational reasoning. The difference in the approach of the two systems permits a natural use of deduction in HOL with extra support required for rewriting. In Watson the converse is true with deduction requiring the additional machinery and rewriting being the natural extension.

3. A large collection of tactics. Examples of tactics include `REWRITE_TAC` which rewrites a goal according to some previously proven theorem or definition, `GEN_TAC` which removes unnecessary universally quantified variables from the front of terms, and `EQ_TAC` which says that to show two things are equivalent, we should show that they imply

each other. In Watson, tactics are not ML scripts or operations on the basic theory objects but are represented as equational theorems containing unexecuted applications of theorems (possibly recursive) as discussed earlier.

More than one implementation of propositional and predicate logic is provided in Watson: there is an implementation of Gries's equational system, a sort of "tableau" approach using the case-definition machinery (the style which students typically learn) and recently an implementation of sequent proofs and a goal-directed natural deduction prover implemented as an interactive tactic using the INPUT command. In addition there is a library that provides an automatic tautology checker.

The "native" style of Watson reasoning heavily emphasizes rewriting and relies on deductive power for reasoning by cases with considerable use of reasoning by contradiction. Other styles have been utilized and tactics can be written that support a more deductive approach.

None of these approaches are as sophisticated yet as the HOL proof package but there is no reason to believe that they cannot be upgraded to this level; this is a matter of system maturity. There has been some effort to implement "HOL tactics" in Watson.

4. A proof management system that keeps track of the state of an interactive proof session. The basic HOL system is similar to that of Watson. In addition Watson keeps track of theorem and axiom usage in a manner that permits automatic invalidation of theorems proven using axioms that have been invalidated.

5. A metalanguage, ML, for programming and extending the theorem prover. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be aggregated to form new theories for later use. The metalanguage makes the verification system extremely flexible. Watson is also based on ML, but prefers to utilize its internal language to specify tactics as well as theorems. Watson is also based on ML, but prefers to utilize its internal language to specify tactics as well as the proof scripts for theorems. This may make it easier for users to learn to program in Watson, though the style of programming is unusual. In our experience, development of complex tactics is difficult regardless of the language used. It is our belief that using the same language for proof scripts, specifications, theorems and tactics may simplify the tactic development effort. However, at this time we have no empirical evidence to support this claim.

In general, the core HOL and Watson systems are similar in capabilities and support environment but are developed from a different approach. In HOL, theorem specification and manipulation are based on propositions. A theorem is specified as a predicate that has been reduced to true using the proof structure of HOL. In Watson, all theorems are equations. Although it is trivial to transform predicates into equations and back, the underlying approach used in the system plays a direct role on proof developments. An equational prover, such as Watson, allows the user to specify theorems in a more natural form, $X = Y$. This leads to a potentially different proof structure. In the HOL system, the user who wishes to prove that $X = Y$ must specify this at the outset and then reduce this to true. In Watson, the user may follow this same approach, resulting the in theorem $(X = Y) = true$ or alternatively they may start with $X(Y)$ and reduce the term to $Y(X)$. The flexibility allows a more general approach to proof development, permitting the use of whichever technique is more appropriate to the specific theorem under development.

5.3. Other Comments

We describe a research direction for Watson which takes us into the area of automated proof. Watson supplies a command which will tell the user if it is possible to convert the currently displayed term to a desired form in a single step; we have extended this feature so that it will search for two-step proofs (in which two theorems are applied as rewrite rules). Beyond two steps, the explosion in size of the search space becomes a problem. After discussions with a colleague who works in parallel programming, we think that it may be possible to increase the efficiency of an algorithm which searches automatically for proofs involving a few rewriting steps by applying parallel programming techniques, and we hope to start doing research in this area in the not too distant future. (In addition, the search could be made more effective by applying heuristic or AI based search strategies for multi-step solutions. This is currently not a focus of the research group.) Our interest is not in completely automating proofs, but in making it so that the prover will be a more intelligent assistant to the user in interactive proofs.

We are philosophically interested in the distinction between a “big theory” and a “little theory” approach advocated by the developers of IMPS ((??)). Watson’s higher-order logic is strong enough to support a “big theory” approach (in which everything is ultimately founded on concepts defined in the built-in higher-order logic) but we have provided support for a “little theory” approach (in which users can axiomatize their own application areas in little theories, then fit these theories together). Our mechanism for theorem export is motivated by the desire to support something like the IMPS approach, but the IMPS group can do much more impressive things with their system: for example, it is possible in Watson to search for theorems which apply to the currently displayed term, but it is not possible to search for theorems in other theories on the desktop which could be exported to the current theory and applied: IMPS has a capability analogous to this which we would like to emulate. This is a research direction we need to work in: the modularity provided by Watson needs improvement.

References

S. Brackin, C. Meadows, and J. Millen, "CAPSL Interface for the NRL Protocol Analyzer" , *IEEE Symposium on Application-Specific Systems and Software Engineering Technology* (ASSET '99, Dallas, March 1999)

B. Boyer, M. Kaufmann and J.S. Moore The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, Vol. 29, No. 2, pp. 27–62, 1995.

A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *HDL Descriptions to Guaranteed Correct Circuit Designs*. Elsevier Scientific Publishers, 1987.

B. Char *et al*, "The Maple symbolic computation system", *SIGSAM Bulletin*, vol. 17 1983), no. 3/4, pp. 31-42.

A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940.

Edward Cohen, *Programming in the 90's: an introduction to the calculation of programs*, Springer-Verlag, 1990.

R.L. Constable et al. Implementing Mathematics with the Nuprl Proof Development System. Prentice Hall, 1986.

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.

W.M. Farmer, J. D. Guttman and F. J. Thayer. IMPS: An Interactive Mathematical Proof System (system abstract) *10th International Conference on Automated Deduction*, pp. 653–654, 1990.

Gordon, M. J. C., Milner, R., and Wadsworth, C. P. *Edinburgh LCF: A Mechanised Logic of Computation*, Volume 78 of Lecture Notes in Computer Science. Springer-Verlag, 1979.

M. Gordon. A proof generating system for higher-order logic. Technical Report 103, University of Cambridge Computer Laboratory, January 1987.

M. Gordon and T. F. Melham, *Introduction to HOL: a theorem proving environment for higher order logic*, Cambridge, 1993.

David Gries and Fred B. Schneider, *A Logical Approach to Discrete Math*, Springer-Verlag, 1993.

M. Randall Holmes, "Systems of combinatory logic related to Quine's 'New Foundations' ", *Annals of Pure and Applied Logic*, 53 (1991), pp. 103-33.

M. Randall Holmes, "A functional formulation of first-order logic 'with infinity' without bound variables", preprint, available from the author.

M. Randall Holmes, "Untyped λ-calculus with relative typing, in *Typed Lambda-Calculi and Applications* (proceedings of TLCA '95), Springer, 1995, pp. 235-48.

M. Randall Holmes, "The Mark2 Theorem Prover", final progress report of Army Research Office grant DAAH04-94-0247.

M. Randall Holmes, *Elementary Set Theory with a Universal Set*, Academia-Bruylant, Louvain-la-Neuve, 1998.

Jech, Thomas, "OTTER experiments in a system of combinatory logic", *Journal of Automated Reasoning*, 14, pp. 413-426.

Ronald Bjorn Jensen, "On the consistency of a slight (?) modification of Quine's 'New Foundations' ", *Synthese*, 19 (1969), pp. 250-63.

D. Kapur and H. Zhang,. An overview of RRL (Rewrite Rule Laboratory) *Proc. of Third International Conf. of Rewriting Techniques and Applications,* Chapel Hill, North Carolina, April 1989.

Knuth, D. and Bendix, P., "Simple word problems in universal algebras", in *Computational Problems in Abstract Algebra* (ed. Leech), Pergamon Press, 1970, pp. 263-97.

W. McCune. Otter 3.0 Reference Manual and Guide Technical Report ANl-94/6, Argonne National Laboratory (1994).

Milner, *et al.*, *The Definition of Standard ML (Revised)*, MIT Press, 1997.

Nederpelt, *et al.*, eds., *Selected Papers on Automath*, North Holland, 1994.

W. V. O. Quine, "New Foundations for Mathematical Logic", *American Mathematical Monthly*, 44 (1937), pp. 70-80.

Syverson, Paul F. and Paul C. van Oorschot. "On Unifying Some Cryptographic Protocol Logics", *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy Oakland CA May 1994*, IEEE CS Press (Los Alamitos, 1994)

Stephen Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, 1988