The Mark2 Theorem Prover[*]

M. Randall Holmes

June 8, 2020

Contents

[*]supported by ARO grant DAAH04-94-0247

1 Introduction

This paper will describe Mark2, a general-purpose system for computer aided reasoning developed by the author. In its current version, it is the final progress report for ARO grant DAAH04-94-0247. The support of the Army Research Office is appreciated.

The current version of the prover documentation is attached as an appendix. The prover source code and a collection of proof scripts can be obtained from our Web page, `http://math.idbsu.edu/~holmes` (follow the link to Mark2).

The prover implements a higher order logic somewhat stronger than the usual simple theory of types. This logic is implemented in three layers.

The first layer is devoted to equational or algebraic reasoning: all Mark2 theories are equational in the sense that their theorems are equations and are used as rewrite rules. The tactic language of Mark2 is based on a device for expressing recursively chained rewrite rules as theorems within a Mark2 theory; the basic ideas of the tactic language live in the first layer, though it acquires refinements as we consider the second and third layers.

The second layer is devoted to reasoning about expressions defined by cases. This layer of the logic implements propositional logic and the logic of identity.

The third layer is devoted to abstraction. Quantification and a strong higher order logic are implemented on this level. The higher order logic used is unusual in being (at least superficially) type-free; it is a version of Quine's set theory "New Foundations" (see [21], [18]).

The three layers of the logic provide an organizational principle for this paper; each of the layers is considered in turn.

There are many examples of prover sessions in the text; they are not as a rule realistic examples of applications, being generally very low-level in their approach. More efficient proofs become possible when more groundwork has been done than is possible in the scope of a surveyable example.

2 The Algebraic Layer and the Tactic Language

This section discusses the features of Mark2 which support equational reasoning and the implementation of tactics (programs which automatically carry out many proof steps) as "equational theorems" of a special form.

2.1 The Term Language of Mark2

In this section, we summarize those features of the term language of Mark2 which are relevant at the algebraic level.

Any string which contains no characters other than letters, digits or the special characters `?` and `_` may be an atomic term of the language of Mark2.

Atomic terms of the language of Mark2 are of four kinds:

numerals: Strings of digits are recognized by Mark2 as numerals. Unlike other atomic constants, numerals do not need to be declared. Mark2 provides built-in operations of infinite-precision unsigned integer arithmetic as part of the tactic language.

constants: An atomic term which contains a non-digit and does not begin with `?` is a *constant*. Constants used in a given Mark2 theory must be declared in that theory.

bound variables: An atomic term which consists of `?` followed by a non-zero-initial numeral is a *bound variable*. The function of bound variables is discussed in the section on the "abstraction layer".

free variables: An atomic term which begins with `?` and is not a bound variable is a *free variable*. Any free variable may be used in any theory without declaration.

A string of special characters not including `?`, `_` or paired forms like parentheses, braces, and brackets is called an *operator*.

An operator of more than one character beginning with `^` is an *operator variable*. Operator variables can be used without declaration, though there are situations in which it is desirable to declare operator variables as having special features.

An operator of more than one character beginning with `:` is a *type-raised operator*, the result of one or more applications of a certain operation to the operator (variable or constant) obtained by stripping off the initial occurrences of `:`. This operation is discussed in the section on the "abstraction layer".

All other operators (including the one character operators `:` and `^`) are *operator constants*. Operator constants must be declared in a theory in which they are used; there are a number of operators which are automatically declared by Mark2.

Complex terms of Mark2 are of three kinds:

infix terms: These consist of a term followed by an operator followed by a term. Parentheses may be required, depending on the precedence of operators involved. Parentheses may be used freely in terms to be parsed by the prover; the prover will display only those parentheses which are needed. Precedence of operators is set by the user; the default is for all operators to have the same precedence and group to the right (the convention of APL).

prefix terms: These consist of an operator followed by a term. Remarks about parentheses are the same as in the case of infix terms. A prefix term is always regarded by Mark2 as being an abbreviation for an infix term; an operator cannot be used as a prefix unless a declaration has been made of the "default left argument" to be understood as the left argument of the infix terms of which terms with the operator as prefix are abbreviations.

bracket terms: These consist of a term enclosed in brackets. If a term contains no bound variables, the result of enclosing it in brackets is the constant function with the referent of the enclosed term as its constant value. The general function of brackets is to define functions by abstraction: bracket terms are λ-terms. DeBruijn levels (see [5] for the origin of this concept) are used to determine variable binding: outermost bracket subterms of a term bind `?1`, all second-to-outermost bracket subterms bind `?2` and so forth. A bound variable may not appear in a context in which it is not bound. This will be discussed further in the section on the "abstraction layer".

Syntactical refinements which are expected to be added are postfix terms and more general forms for operators (allowing alphanumeric characters to be used in names of operators).

2.2 Substitution and Rewriting

For the moment, we restrict ourselves to the subset of the language without bound variables. Bracketed terms are present, with `[T]` denoting the constant function whose value everywhere is `T`. The virtue of this subset of the language is that the definition of substitution is very simple. If `T` is a term, we use the notation `T[A/?x]` to denote the result of replacing the free variable `?x` with the term `A` wherever the variable appears in `T`.

The set of *substitution instances* of a term `T` is the smallest set of terms which contains `T` itself and contains the term `U[A/?x]` for each term `A` and free variable `?x` whenever it contains `U`.

A term of the form `A = B` is called an *equation* (unsurprisingly). We say that a term `T` is rewritten to the term `U` by the equation `A = B` whenever the equation `T = U` is a substitution instance of the equation `A = B`. This has one peculiar consequence: if `B` happens to contain free variables which do not occur in `A`, then there are many possible results of rewriting `T` with `A = B`. In practice, variables in `B` which do not occur in `A` are rewritten to new variables (i.e., variables wich did not occur in `A`).

If `A = B` is an equation, the equation `B = A` is said to be the converse of `A = B`. Equations `A = B` and `B = C` are said to have composition `A = C` (composition can be given a somewhat more general definition, but this suffices for our purposes).

If `A = B` is an equation, we define **R**(`A = B`) as the equation `(?new ^+ A) = (?new ^+ B)`, where the variable `?new` and the infix variable `^+` do not appear in `A` or `B` (strictly speaking, this does not define a unique equation, but the notion of rewriting with all such equations is the same). Similarly, we define **L**(`A = B`) as `(A ^+ ?new) = (B ^+ ?new)` and **V**(`A = B`) as `[A] = [B]`. We refer to **R**, **L**, and **V** as "localizing operations" on equations. We formalize the notion of applying a finite sequence of localizing operations to an equation: the empty sequence applied to an equation gives the equation itself, the application of a one-term sequence (O_1) is the same as the application of O_1, while the application of the sequence $(O_1, \ldots, O_n)$ of localization operators O_i to an equation `A = B` is the result of applying $(O_1, \ldots, O_{n-1})$ to the equation O_n(`A = B`).

A Mark2 "abstract theory" is a set of equations containing all identity equations `T = T` (mod considerations of declared constants and operations) and closed under converse, composition, substitution instance, and localizing operations. It should be clear that the notion of a Mark2 abstract theory coincides with the notion of a set of equations closed under the usual methods of proof in equational theories (within the bounds of the details of Mark2 notation). A "concrete theory" in Mark2 is simply a set of equations (regarded as representing the minimal abstract theory of which it is a subset).

2.3 The Basic Session Model

The aim of a session under Mark2 is the proof of an equational theorem in a given theory. One is given a set of theorems already postulated or proved (a concrete theory); the aim is to demonstrate that a desired equation can be added to the concrete theory, i.e., that it can be obtained from the sentences in the concrete theory by a combination of the operations on equations listed in the previous section.

Initially the user enters a term `T`. The user then sees the term `T`, but the object actually being manipulated is the equation `T = T`. A sequence of transformations are applied to this equation, each of which is guaranteed to preserve the condition that the equation produced will belong to the minimal abstract theory containing the given concrete theory. The allowed transformations are the conversion of the equation (interchange of left and right hand sides), global assignment of a value to a free variable (repeated application of this justifies replacement of the equation by any substitution instance) and rewriting of the right hand side of the equation by an element of the theory or by the result of application of a sequence of localizing operations to an element of the theory.

At each step, the prover is manipulating an equation internally, but the user sees things rather differently. What the user is presented with is a term, the right hand side of the equation which the prover is considering, and a selected subterm of that term. The choice of selected subterm encodes the sequence of localizing operations which will be applied to any equation of the concrete theory invoked as a rewrite rule. In other words, the selected subterm is the specific subterm which an equation from the concrete theory may be used to rewrite. The user may manipulate the selection of this subterm by "moving" through the term, either to the left (an **L** or **V**, depending on the form of the previous selected subterm is added to the sequence of these operations to be applied), to the right (precisely analogous to movement to the left), up (the sequence of localizing operations to be applied has its last element deleted, so rewriting will be carried out on the "parent" term of the previously selected subterm) or to the top (the list of localizing operations is emptied and rewrites will be applied to the entire right-hand side of the equation). More sophisticated movement commands are available, but this is the basic set. The sequence of localizing operations is actually represented by a list of booleans (truth values), in which `true` represents **L**, `false` represents **R**, and either truth value may indifferently represent `V`.

The options available from the user's standpoint are to interchange the left and right hand sides of the equation (thus replacing the right-hand term with the left-hand term as the term being viewed), make a global assignment to a free variable (this may affect both sides of the equation being proved), navigate within the term he/she sees (change the selection of the subterm to which rewrites are to be applied), or apply equations in the concrete theory as rewrite rules to the selected subterm of the right hand side of the equation. When the equation has achieved the desired form, it may be added to the concrete theory; an identifier is declared to represent the newly proved theorem.

This is a model of equational proof which is adequate in theory but of course tedious in practice. It differs from the usual approach to proof with rewrite rules in requiring complete control over where rules are applied (the usual approach can be realized using the tactic language). The introduction of the metaphor of navigation through the parse tree of the term being manipulated goes some way toward making this sort of reasoning feasible.

2.4 The Tactic Language

One of the novel features of Mark2 is the ability to represent "tactics" (programs for the automatic execution of many proof steps) as equational theorems, stored in theories in the same way as conventional theorems. The term "tactic" is borrowed from the provers of the LCF family such as Nuprl ([3]) or HOL ([8]), but in those systems a tactic is an ML function, an object of quite a different sort than a theorem.

Mark2 provides predeclared operators which allow one to represent in a term, without changing its semantics, the intention of rewriting with an equation. A term of the form `thm => term` has the same referent as `term`, but has the additional connotation that it is intended to apply the equation in the current theory with the name `thm`. It is thus required that theorems in the current theory have names which are declared constants in the current theory. A term `thm <= term` has a similar additional connotation, except that the converse of the theorem referred to by `thm` is to be applied.

Special commands allow the introduction of such rewriting annotations. When the tactic interpreter is invoked, the intentions signalled by all such rewriting annotations are carried out. When the theorem `thm` cannot be used to rewrite the term `thm => term` or `thm <= term` in the indicated sense, the annotation is simply dropped. The tactic interpreter follows a depth-first strategy; the term `thm => term` or `thm <= term` is not considered for rewriting until `term` itself is reduced to a form free of annotations.

The power of this idea is seen when it is realized that one can prove equations as theorems which involve rewriting annotations. Rewriting with such theorems will introduce new rewriting annotations; the tactic interpreter carries these out in the same way that it carries out annotations present when it is invoked.

The tactic language of Mark2 is also discussed in our [14].

2.5 An Example Session

The prover is implemented in Standard ML, and its interface is currently the SML interface. We do not claim that this is a desirable state of affairs; a better interface is one of our goals.

In this subsection, we will present examples of declarations setting up a small theory, and some simple proof sessions, in order to give a concrete referent to the discussion up to this point.

In all sessions presented in this paper, it is useful to be aware that the SML prompt is a hyphen `-`; this allows one to distinguish commands given by the user to the prover from prover responses.

We first declare an operator.

```
- declareinfix "+";
```

At this point, the prover knows nothing about this operator: we continue by supplying some basic axioms.

```
- axiom "COMM" "?x+?y" "?y+?x";

COMM:
?x + ?y =
?y + ?x
["COMM"]

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

ASSOC:
(?x + ?y) + ?z =
?x + ?y + ?z
["ASSOC"]

- axiom "ZERO" "0+?x" "?x";

ZERO:
0 + ?x =
?x
["ZERO"]
```

The command and the prover's responses are given. In connection with the associativity axiom, it is worth recalling here that the default grouping of all operators is to the right.

We first prove a simple theorem.

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

{?x + ?y + ?z}

?x + ?y + ?z
```

We enter the term which will serve as the left side of the theorem to be proved.

```
- right();

?x + {?y + ?z}

?y + ?z
```

We "move" to the right subterm.

```
- ruleintro "COMM";

?x + {COMM => ?y + ?z}

COMM => ?y + ?z
```

We indicate the intention of rewriting this subterm with the commutative law.

```
- execute();

?x + {?z + ?y}

?z + ?y
```

We carry out this intention: `execute` is the command which invokes the tactic interpreter.

```
- up();

{?x + ?z + ?y}

?x + ?z + ?y

- ruleintro "COMM"; execute();

{COMM => ?x + ?z + ?y}

COMM => ?x + ?z + ?y
```

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

(?z + ?y) + ?x
```

We move back up to the top of the term and apply the commutative law.

```
- ruleintro "ASSOC"; execute();

{ASSOC => (?z + ?y) + ?x}

ASSOC => (?z + ?y) + ?x

{?z + ?y + ?x}

?z + ?y + ?x
```

We apply the associative law. This completes the proof of the intended theorem; we now record the theorem as proved.

```
- prove "ROTATE";

ROTATE:
?x + ?y + ?z =
?z + ?y + ?x
["ASSOC","COMM"]
```

The new theorem is assigned the name `ROTATE`. It can be invoked in the same way the axioms were invoked:

```
- start "?a+?b+?c+?d";

{?a + ?b + ?c + ?d}

?a + ?b + ?c + ?d

- ri "ROTATE"; execute();

{ROTATE => ?a + ?b + ?c + ?d}

ROTATE => ?a + ?b + ?c + ?d

{(?c + ?d) + ?b + ?a}

(?c + ?d) + ?b + ?a

- ri "ROTATE"; execute();
```

```
{ROTATE => (?c + ?d) + ?b + ?a}

ROTATE => (?c + ?d) + ?b + ?a

{?a + ?b + ?c + ?d}

?a + ?b + ?c + ?d
```

Now we show how simple tactics are developed. After this point in the paper, we will not show displays of the selected subterm when it is at the top level.

```
- s "?x+0";

{?x + 0}

- ri "COMM"; execute();

{COMM => ?x + 0}

{0 + ?x}

- ri "ZERO"; execute();

{ZERO => 0 + ?x}

{?x}

- prove "COMMZERO";

COMMZERO:
?x + 0 =
?x
["COMM","ZERO"]
```

The theorem `COMMZERO` is an ordinary equational theorem, of course.

```
- start "?x+?y";

{?x + ?y}

- ruleintro "ZERO"; ruleintro "COMMZERO";

{ZERO => ?x + ?y}

{COMMZERO => ZERO => ?x + ?y}
```

```
- prove "EITHERZERO";

EITHERZERO:
?x + ?y =
COMMZERO => ZERO => ?x + ?y
["COMM","ZERO"]
```

The new “theorem” `EITHERZERO` involves rewriting annotations; it is a simple tactic.

```
- start "?x+0";

{?x + 0}

- ruleintro "EITHERZERO"; execute();

{EITHERZERO => ?x + 0}

{?x}

- start "0+?x";

{0 + ?x}

- ruleintro "EITHERZERO"; execute();

{EITHERZERO => 0 + ?x}

{?x}
```

We see that `EITHERZERO` has an effect which cannot be achieved by rewriting with a single equation; it eliminates addition of zero on the right or the left.

```
- declarepretheorem "ZEROES";

- start "0+?x";

{0 + ?x}

- ri "ZERO"; execute();

{ZERO => 0 + ?x}

{?x}

- ri "ZEROES";
```

```
{ZEROES => ?x}

- prove "ZEROES";

ZEROES:
0 + ?x =
ZEROES => ?x
["ZERO"]
```

The special declaration with `declarepretheorem` is required because the `ruleintro` command checks for declarations of theorems introduced; by the nature of the situation, `ZEROES` must be introduced before it has actually been proved (and so declared). The effect of rewriting with `ZEROES` is more impressive than that of rewriting with `EITHERZERO`:

```
- start "0+0+0+0+0+0+?x";

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

- ri "ZEROES"; execute();

{ZEROES => 0 + 0 + 0 + 0 + 0 + 0 + ?x}

{?x}
```

The effect of `ZEROES` is to eliminate addition of zero on the left as many times as it can.

```
- startover();

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

- ri "ZEROES"; steps();

{ZEROES => 0 + 0 + 0 + 0 + 0 + 0 + ?x}

ZEROES => 0 + 0 + 0 + 0 + 0 + 0 + ?x
ZEROES => 0 + 0 + 0 + 0 + 0 + ?x
ZEROES => 0 + 0 + 0 + 0 + ?x
ZEROES => 0 + 0 + 0 + ?x
ZEROES => 0 + 0 + ?x
ZEROES => 0 + ?x
ZEROES => ?x
?x
```

We recapitulate the last proof. The `steps` command invokes the "tactic debugger", which traces the effect of `execute` step by step. Note that it is important that the rewriting annotation with `ZEROES` will be dropped when it no longer applies; this is why it is possible for the recursion to terminate. These examples, while very simple, should suggest why the tactic interpreter of Mark2 supports, in effect, a full-fledged programming language. Refinements described in the next section increase the ease of use of this programming language.

2.6 Refinements of the Tactic Language

In this section, some refinements of the tactic language of Mark2 are discussed, with motivating examples.

The first refinement is best illustrated by considering the tactic `EITHERZERO` developed in the previous subsection. It turns out that this tactic has unexpected behavior in certain circumstances:

```
- s "0+?x+0";

{0 + ?x + 0}

- ri "EITHERZERO"; execute();

{EITHERZERO => 0 + ?x + 0}

{?x}
```

The difficulty here is that we may have thought of the applications of `ZERO` and `COMMZERO` as alternatives, but there is a case where one can be applied and then the other.

The solution is the definition of another pair of infixes for rewriting annotation: the new infixes `=>>` and `<<=` signal the intention of applying an equation as a rewrite rule if a previous attempt to rewrite has failed. In an expression of the form `thm_n =>>...thm_2 =>> thm_1 => term`, the tactic interpreter will attempt to apply each of the theorems in turn, starting with `thm_1`; once a theorem applies, the subsequent theorems (working outward) will be ignored. The innermost annotation infix must be `=>` (or `<=`) because there is no previous rewrite in the sequence which might fail.

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

{?x + ?y}

- ruleintro "ZERO";
{ZERO => ?x + ?y}

- altruleintro "COMMZERO";

{COMMZERO =>> ZERO => ?x + ?y}

- prove "EITHERZERO2";
```

```
EITHERZERO2:
?x + ?y =
COMMZERO =>> ZERO => ?x + ?y
["COMM","ZERO"]

- s "0+?x+0";

{0 + ?x + 0}

- ri "EITHERZERO2";

{EITHERZERO2 => 0 + ?x + 0}

- execute();

{?x + 0}
```

Here we see more natural behavior for EITHERZERO2; it eliminates one addition of zero on either the right or the left. This is a trivial example, of course; it has turned out to be important in ensuring reliable behavior of more complex tactics to be able to be certain that only one of a list of alternatives would be applied.

The second major refinement of the tactic language allows parameters to be supplied to tactics. The infix @ is the predeclared infix of function application; it is also used to link parameters to tactics (parameterized tactics can be thought of as theorem-valued functions). Multiple parameters can be linked with the predeclared pair infix ,. Here is a simple example of a parameterized tactic:

```
- start "?x^+?y";

{?x ^+ ?y}

- right();

?x ^+ {?y}

?y

- ruleintro "?thm";

?x ^+ {?thm => ?y}

?thm => ?y

- prove "RIGHT@?thm";
```

```
RIGHT @ ?thm:
?x ^+ ?y =
?x ^+ ?thm => ?y
[]

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

{?x + ?y + ?z}

- ruleintro "RIGHT@COMM"; execute();

{(RIGHT @ COMM) => ?x + ?y + ?z}

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

The parameterized tactic `RIGHT` allows one to apply a theorem (supplied as the parameter) to the right subterm of the target term.

Parameterized tactics can have operators as names:

```
- start "?x";

{?x}

- ruleintro "?thm1"; ruleintro "?thm2";

{?thm1 => ?x}

{?thm2 => ?thm1 => ?x}

- prove "?thm1**?thm2";

?thm1 ** ?thm2:
?x =
?thm2 => ?thm1 => ?x
[]
```

The infix `**` has been "proved" as a theorem implementing theorem or tactic composition.

Another elementary use of parameterization is to control the effects of the converses of theorems which "destroy information":

```
- thmdisplay "REFLEX";

REFLEX:
?a = ?a =
true
["REFLEX"]
```

```
- start "true";

{true}

- revruleintro "REFLEX"; execute();

{REFLEX <= true}

{?a_10 = ?a_10}
```

Note that the converse of `REFLEX` introduces a computer-generated new variable. We would like to have control over what is introduced.

```
- start "true";

{true}

- initializecounter();

- revruleintro "REFLEX"; execute();

{REFLEX <= true}

{?a_1 = ?a_1}

- assign "?a_1" "?a";

{?a = ?a}

- prove "REV_REFLEX@?a";

REV_REFLEX @ ?a:
true =
?a = ?a
["REFLEX"]

- start "true";

{true}

- ruleintro "REV_REFLEX@3"; execute();

{(REV_REFLEX @ 3) => true}

{3 = 3}
```

The `initializecounter` command is a technicality; it forces the counter on computer-generated new variables back to 1 so that the proof will work correctly if run as a script (there would be problems with the `assign` command otherwise). One can see examples here of the use of converses of theorems as rewrite rules and of the use of global assignment. One can also see that a parameterized tactic can take objects of a theory as parameters as well as theorems or tactics.

2.7 An Extended Example: Combinatory Logic

As an extended example of the capabilities of Mark2 as a rewriting system, we present a development of untyped combinatory logic in the style of Curry (see [4]).

The theory *CL* is an equational theory. Objects of the theory are called "combinators". Atomic terms of the language of this theory are atomic constants I, S, K, B, C, and any others that may be desired, plus an infinite supply of variables. The only term construction is function application: if T and U are terms, (TU) is a term. Excess parentheses are deleted from the left (the opposite of the default Mark2 convention): $ABCD$ is read $(((AB)C)D)$.

The axioms of *CL* are as follows:

II: $IX = X$ (for any term X)

KK: $KXY = X$ (for any terms X and Y)

SS: $SXYZ = XZ(YZ)$ (for any terms X, Y, and Z)

Axioms could be provided for the combinators B and C (of composition and conversion), but it is more natural to introduce these combinators by definition (as we will see in the development below).

We begin with declarations and axioms:

```
- declareinfix ".";  (* this is the "function application" internal to CL *)
- setprecedence "." 1;  (* this gives the correct grouping *)
- declareconstant "I";  (* atomic combinators *)
- declareconstant "K";
- declareconstant "S";
- axiom "II" "I . ?x" "?x";
- axiom "KK" "K . ?x . ?y" "?x";
- axiom "SS" "S . ?x . ?y . ?z" "?x . ?z . (?y . ?z)";
(* the output from these axiom declarations is omitted *)
```

The `setprecedence` command is used to tell the prover that the `.` operator of "function application" (in the *CL* sense) groups to the left rather than to the right. Mark2 regards all operators with odd precedence as grouping to the right and all operators with even precedence as grouping to the right.

In this section, some abbreviations of Mark2 command names are used: `ri` for `ruleintro`, with initial `a` for `alt-` and `r` for `rev-`, `p` for `prove`, and `s` for `start`.

We develop some utilities:

```
- start "?x^+?y";

{?x ^+ ?y}

- left();

{?x} ^+ ?y

?x

- ri "?thm";

{?thm => ?x} ^+ ?y

?thm => ?x

- p "LEFT@?thm";

LEFT @ ?thm:
?x ^+ ?y =
(?thm => ?x) ^+ ?y
[]

(* the output from the following very similar proof is suppressed *)

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

RIGHT @ ?thm:
?x ^+ ?y =
?x ^+ ?thm => ?y
[]

(* output from this proof suppressed *)
- declarepretheorem "EVERYWHERE";
- start "?x.?y";
- right(); ri "EVERYWHERE@?thm"; ari "?thm";
- up();left(); ri "EVERYWHERE@?thm"; ari "?thm";
- top();
- ri "?thm";
- p "EVERYWHERE@?thm";

EVERYWHERE @ ?thm:
?x . ?y =
?thm => (?thm =>> (EVERYWHERE @ ?thm) => ?x)
. ?thm =>> (EVERYWHERE @ ?thm) => ?y
[]
```

`LEFT`, `RIGHT`, and `EVERYWHERE` are examples of "structural" tactics which enable the application of theorems at points other than where the tactic is actually applied. The `LEFT` and `RIGHT` tactics enable application of tactics to the immediate left and right subterms of the selecte subterm; the `EVERYWHERE` tactic allows a theorem to be applied to all subterms of the selected subterm in a "bottom up" fashion, if the term is built only with the `.` operator.

We develop an abstraction algorithm along classical lines (proof commands are given, but output is suppressed; the statement of each theorem proved is given, followed by its proof):

```
(* development of an abstraction algorithm *)

(*
ABSI @ ?x:
?x =
I . ?x
["II"]
*)

s "?x";
rri "II"; ex();
p "ABSI@?x";

(*
ABSK @ ?x:
?y =
K . ?y . ?x
["KK"]
*)

s "?y";
initializecounter();
rri "KK"; ex();
assign "?y_1" "?x";
p "ABSK@?x";

(*
ABSS @ ?x:
?T . ?x . (?U . ?x) =
S . ?T . ?U . ?x
["SS"]
*)

s "?T . ?x . (?U . ?x)";
rri "SS"; ex();
p "ABSS@?x";
```

These theorems are the basic clauses of the abstraction algorithm. The parameter that each of them takes is the term relative to which abstraction is to be carried out.

We now declare the name `ABS` of our abstraction tactic to prepare for its recursive definition.

```
declarepretheorem "ABS";
(* a subtactic for handling application expressions *)

(*
ABSAPP @ ?x:
?T . ?U =

(ABSS @ ?x) => ((ABS @ ?x) => ?T) . ((ABS @ ?x)
   => ?U)
["II","KK","SS"]
*)

s "?T.?U";
right(); ri "ABS@?x";
up(); left(); ri "ABS@?x";
top();
ri "ABSS@?x";
p "ABSAPP@?x";

(* the abstraction algorithm itself -- Curry's (fab) *)

(*
ABS @ ?x:
?t =

(ABSK @ ?x) =>> (ABSAPP @ ?x)
=>> (ABSI @ ?x) => ?t
["II","KK","SS"]
*)

s "?t";
ri "ABSAPP@?x";
ari "ABSI@?x";
ari "ABSK@?x";
p "ABS@?x";
```

```
(* We demonstrate why this is not a very good algorithm *)

- s "?x.(?y.?z)";
- ri "ABS@?z"; ex();
- left();
- ri "ABS@?y"; ex();
- left();
- ri "ABS@?x"; ex();
- left();
```

The display which follows is:

```
{S . (S . (K . S)
      . (S . (S . (K . S) . (S . (K . K) . (K . S)))
         . (S . (S . (K . S) . (S . (K . K) . (K
                     . K)))
            . (S . (K . K) . I))))
   . (S . (S . (K . S)
         . (S . (S . (K . S) . (S . (K . K) . (K
                     . S)))
            . (S . (S . (K . S) . (S . (K . K) . (K
                        . K)))
               . (K . I))))
      . (S . (K . K) . (K . I)))}
. ?x . ?y . ?z
```

This is not the optimal form of the composition combinator B!

```
(* the added steps needed to repair this *)

(* ABSFIX handles the idea of using K on complex terms (not just
atomic terms) which do not contain ?x; it might seem that we would
have a problem recognizing such terms, but it turns out to be easy *)

(*
ABSFIX:
S . (K . ?a) . (K . ?b) . ?x =
K . (?a . ?b) . ?x
["KK","SS"]
*)

s "S.(K.?a).(K.?b).?x";
ri "SS";
ri "EVERYWHERE@KK"; ex();
ri "ABSK@?x"; ex();
p "ABSFIX";
```

```
(* ABSFIX2 handles the idea of letting ?f rather than S(K?f)I be the
abstract from ?f.?x *)

(*
ABSFIX2:
S . (K . ?a) . I . ?x =
?a . ?x
["II","KK","SS"]
*)

s "S.(K.?a).I.?x";
ri "SS"; ri "EVERYWHERE@KK";  ri "EVERYWHERE@II"; ex();
p "ABSFIX2";

(* the following commands modify the tactic ABS to incorporate new steps *)

ae "ABS";
ri "ABSFIX";ri "ABSFIX2";
rp "ABS@?x";  (* rp is the short form of the "reprove" command *)

(*
ABS @ ?x:
?t =

ABSFIX2 => ABSFIX => (ABSK @ ?x) =>> (ABSAPP @ ?x)
=>> (ABSI @ ?x) => ?t
["II","KK","SS"]
*)

(* we repeat the development of the B combinator from above *)

- s "?x.(?y.?z)";
- ri "ABS@?z"; ex();
- left();
- ri "ABS@?y"; ex();
- left();
- ri "ABS@?x"; ex();
- left();

(* resulting display: *)

{S . (K . S) . K} . ?x . ?y . ?z

(* things come out much simpler! *)
```

We develop a reduction algorithm. The infix `*>` between theorems has the following effect: when `thm2 *> thm1` is applied, one applies `thm1`, then further applies `thm2` only in case `thm1` succeeds.

```
(* reduction algorithm *)

(*
RED:
?t =

(RED *> SS) =>> (RED *> KK) =>> (RED *> II)
=> (RIGHT @ RED) => (LEFT @ RED) => ?t
["II","KK","SS"]
*)

- declarepretheorem "RED";
- s "?t";
- ri "LEFT@RED"; ri "RIGHT@RED";
- ri "RED*>II";
- ari "RED*>KK";
- ari "RED*>SS";
- p "RED";
```

We give an example of the use of the definition facility of Mark2 (which will be further discussed below).

```
(* we define the B combinator and prove its characteristic theorem *)

defineconstant "B" "S . (K . S) . K";
```

The effect of this command is to create the following theorem:

```
B:
B =
S . (K . S) . K
["B"]

(* a theorem is developed for B analogous to the characteristic theorems
of the primitive combinators *)

(*
BB:
B . ?x . ?y . ?z =
?x . (?y . ?z)
["B","II","KK","SS"]
*)

- s "B . ?x . ?y . ?z";
- ri "EVERYWHERE@B";  (* definitional expansion followed by reduction *)
- ri "RED"; ex();
- p "BB";
```

We demonstrate the possibility of a tactic operating on lists of parameters of arbitrary length in Mark2:

```
(* we develop an even more general abstraction tool *)

- setprecedence "," 1;  (* give pairing the same left associativity as
internal application *)
- dpt "ABSLIST";
- s "?t";
- ri "ABS@?y";
- ri "LEFT@ABSLIST@?x";
- p "ABSLIST@?x,?y";

ABSLIST @ ?x , ?y:
?t =
(LEFT @ ABSLIST @ ?x) => (ABS @ ?y) => ?t
["II","KK","SS"]
```

The prover allows lists of arguments to be supplied to tactics linked by the predeclared operator `,` (ordered pair). This tactic abstracts relative to its last argument, then invokes itself with the remainder of its list of parameters as its argument. It will fail with an atomic parameter, so the first argument in the list needs to be a dummy (we always use `0`). The structure of the tactic makes it a good idea to change the grouping of the `,` operator from the default.

```
(* once again, we develop the B combinator *)

- s "?x.(?y.?z)";
- ri "ABSLIST@0,?x,?y,?z"; ex();

(* this works "all at once"; now we can develop other familiar
combinators *)

(* a last example: the (suppressed) output is suggested
by the form of the following definition *)

(* the C combinator -- conversion *)

- s "?x.?z.?y";
- ri "ABSLIST@0,?x,?y,?z"; ex();
- defineconstant "C" "S . (S . (K . S) . (S . (K . K) . S)) . (K . K)";

(* its characteristic theorem *)
```

```
(*
CC:
C . ?x . ?y . ?z =
?x . ?z . ?y
["C","II","KK","SS"]
*)

- s "C.?x.?y.?z";
- ri "EVERYWHERE@C"; ri "RED"; ex(); (* definitional expansion + reduction *)
- p "CC";
```

The choice of combinatory logic as the vehicle for an extended example of the rewriting capabilities of the prover is not accidental. As we will explain in more detail below (under abstraction) the development of the tactic language was originally driven by the requirements of reduction and synthetic abstraction algorithms. Independently of this consideration, the example affords an illustration of the ability to carry out nontrivial sorts of "computation" using tactics.

The interested reader may find a more extended development of combinatory logic, including an algorithm for strong reduction, in a longer file on the author's Web page from which the example as given here is adapted.

2.8 Arithmetic and "Functional Programming"; Technical Rewriting Issues

Certain terms are rewritten by the tactic interpreter without explicit rewriting annotations. Arithmetic expressions with the predeclared operators `+!`, `*!`, `-!`, `/!`, `%!`, `<!`, and `=!` are automatically evaluated by the tactic interpreter; these are operations and relations of infinite precision unsigned integer arithmetic (the relations give boolean output (`true` or `false`)). Expressions defined by cases with an explicit boolean hypothesis are automatically collapsed: `true || ?x , ?y` is automatically rewritten to `?x` and `false || ?x , ?y` is automatically rewritten to `?y` (the predeclared operators used here are discussed below under the conditional layer).

The user may create similar effects using "functional programming" functionalities of the prover. The user may "bind" a function to a function name or an operator with instructions that this term always be applied when the function or operator is present in a term of the correct form. The form of the theorem needs to be appropriate. For example, if the user has introduced a function `car` with theorem `CAR` asserting `(car @ ?x , ?y) = ?x` and further issues the command `proveprogram "car" "CAR"`, then the tactic interpreter will automatically apply the theorem `CAR` wherever it sees the function `car` applied to a term. A preclared prefix `#` is provided which has the effect of suppressing the execution of any "functional program" attached to the top-level function or operator of its argument. There is another predeclared prefix `##` which implements laziness: rewriting annotations inside such a term are ignored unless and until a rewrite from outside the scope of the prefix affects the term.

Mark2 is quite different in its aims from other rewriting systems, to the point where it is difficult to compare them. Other rewriting systems usually involve aggressively rewriting with a list of given rules, using the rules wherever it is possible to apply them. In Mark2, the user (or a program written by the user) are supposed to indicate how rules are to be applied in detail. In standard rewriting systems, reversible rules are a problem, since obvious looping possibilities arise. In Mark2 the use of converses of rewrite rules is not a difficulty. (It is interesting to note that if the conversion infixes `<=` and `<<=` are suppressed, the logic of Mark2 changes from equational logic to precisely the "rewriting logic" proposed by the developers of OBJ (see [19]).

Mark2 can be viewed as a rewrite system of the usual sort, in which the targets of rewrite rules are not the arguments of rewriting annotations but the entire terms of form `thm => term` and other related forms (as well as arithmetic terms and those on which "functional programming" acts). One can then ask questions of familiar kinds. For example, Mark2 is "almost" confluent, if "functional programming" is ignored. The one failure of confluence is in the behaviour of the alternative infixes `=>>` and `<<=`; in a term `?thm_n =>> ... =>> ?thm_2 => ?thm_1 => term`, rewriting any subterm on the right side of a `=>>` may cause the final output to change. We regard this failure as unimportant, for a reason which can be expressed precisely: if we redefined the alternative infixes so that `(?thm_2 =>> ?thm_1) => ?term` had the behaviour now assigned to `?thm_2 =>> ?thm_1 => ?term`, then changed the precedence of the alternative infixes from 0 to 1 (which would cause them to group to the left), terms with alternative rule infixes would look and behave exactly as they do now, except that the subterms which we must avoid executing to preserve confluence now would not exist at all. We do not see any reason why a user would invoke the tactic interpreter on a term to the right of an alternative infix, so we are not inclined to make the indicated change. Another condition which needs to be noted to justify our claim that the system is essentially confluent is that the prover does not allow rewrite rules with rewriting annotations in their left sides to be applied; this is enforced by not allowing any term to match a term with rewriting annotations in it. If we view Mark2 as a conventional rewriting system, the redexes are those terms with rewriting annotations whose arguments contain no rewriting annotations; this enforces the diamond property. "Functional programming" complicates the definition of a redex somewhat.

Mark2 tactics can, of course, implement rewriting strategies of the usual more aggressive sort, as well as refinements of such strategies.

An area in which we did some work with `EFTTP` (the precursor of Mark2) which has not yet been upgraded to Mark2 is the investigation of "parallel execution orders" for the prover. A "breadth-first" strategy of reduction might have some advantages over the current "depth-first" approach (though speed would not be one of them, at least on a conventional machine). Such a strategy would allow the prover to carry out reductions higher up in the parse tree when it recognized that they would not interfere with reductions of terms at lower levels; this would allow some non-strict execution order (allowing some tactics to terminate which otherwise would not). An element of nondeterminism would be introduced, because theorems in lists of alternatives might be applied in preference to theorems appearing earlier in the list because it would become clear sooner that they could be applied.

2.9 Theorem Export and Theorem Search

2.9.1 Theorem Export

The reader may have noticed that every theorem proved by Mark2 is annotated with a list of names of theorems. These theorems are the axioms and definitions on which Mark2 determines that the proof of the theorem depends.

Lists of dependencies are maintained for each theorem in order to support a facility of *theorem export*: Mark2 allows theorems proved in one theory to be exported to another under suitable conditions.

The prerequisites for export of a theory from theory A to theory B is the existence of a "view" of theory B from theory A (this term is borrowed from the developers of IMPS ([7]). The view is a list of translations of names of axioms and definitions, and possibly of other symbols. It does not need to be exhaustive: Mark2 will match theorems of A with their translations into B and either reject the view and abort the export (if the theorems do not match in form) or extend the translation implied by the view as necessary. For example, if the commutativity of + in A matches the commutativity of * in B, the translation table will be extended to force + to be translated by *; if the commutativity of + in A is reported to match a theorem in B which does not have the form of a commutative law, or if we are already committed to a different translation of +, the attempt at theorem export will fail. The export of a theorem via a view will succeed if a coherent translation of the axioms on which the theorem depends into terms of theory B can be obtained from the view. The theorem export system will export tactics just as it exports theorems; all tactics or theorems called by the tactic explicitly exported will also be exported (recursively). Name collisions are automatically avoided.

An effect of the requirements of the theorem export system is that, while the system does allow theorems to be modified in form or tactics to be debugged (by reproving a theorem/tactic of the same name) it never allows a theorem or tactic to be modified in a way which introduces additional dependencies, since this would corrupt the dependency information given about other tactics which invoke the given theorem or tactic, compromising the validity of theorem export.

The theorem export system is fairly cumbersome to invoke and has not been used extensively; it has however been valuable for exporting complex recursively defined tactics from one theory to another. We would like to improve the theorem export system to allow, for example, the user to be able to search for theorems in theories other than the one in which he/she is currently working which might be applicable to a current situation.

2.9.2 Theorem Search

The system allows the user to search for theorems in the current theory applicable to a given situation. One can ask for theorems which match a given equation. One can ask for a theorem which will convert the current term to a given form. One can ask for a list of theorems applicable to the current selected subterm. All of these features have the effect that it is not too unpleasant to work in a theory defined by another user (or by oneself a long time ago); one can discover the names of theorems one needs fairly painlessly.

Finally, most daringly, it is possible to invoke theorem searches automatically: a "theorem" of the form `?x=?y` can be executed in the tactic language, having the effect of the first theorem the prover finds in its theorem list which justifies the equation. When equations are used as theorems in a tactic, interesting effects can be achieved, since theorem searches can then be automatically set up by the tactic rather than the user, though tactics which automatically search for theorems tend to be slow. This is another feature which invites further development.

3 The Conditional Layer

This layer of the logic of Mark2 is devoted to the properties of expressions defined by cases. The canonical form for expressions of this kind is `(?a = ?b) || ?x , ?y`, which can be read "if $a = b$ then x else y". Terms of the form `?p || ?x , ?y`, where `?p` is not an equation, are understood to be equivalent to `(true = ?p) || ?x , ?y`; terms of the form `?p || ?q`, where `?q` is not a pair, are currently treated by the prover as ill-formed expressions.

In an expression of the form `?p || ?x , ?y`, we refer to `?x` and `?y` as the *cases* and to `?p` as the *hypothesis*.

The logical properties of the conditional construction which drive the handling of conditional expressions in Mark2 are the following (originally proposed as axioms for a combinatory logic implementing first-order logic with equality in the author's preprint [16]; see further discussion of this system in a later section):

projection (1): (if **true** then x else y) $= x$.

projection (2): (if **false** then x else y) $= y$.

distribution: C[if p then x else y] = if p then $C[x]$ else $C[y]$, where $C[\ldots]$ represents any context.

hypothesis: (if $a = b$ then $C[a]$ else c) = (if $a = b$ then $C[b]$ else c)

Auxiliary properties which handle the nature of equations and hypotheses of odd forms are:

equation: $(a = b)$ = if $a = b$ then **true** else **false**.

odd hypothesis: (if p then x else y) = (if **true** = p then x else y)

The role of arbitrary contexts in the distribution and hypothesis properties of the conditional was originally handled by using abstraction machinery to replace a context $C[x]$ with a function application $f(x)$. This is no longer done, for three reasons: the distribution and hypothesis properties can be adequately handled using a finite collection of instances; the abstraction system of Mark2 does not allow abstraction from every context, so an implementation using abstraction would not be fully effective or would require additional special cases; an approach based on abstraction turns out to be tactically flawed as a way to prove theorems.

The schemes of distribution and collection are replaceable by the following finite list of properties (here we use Mark2 notation, because the structure of Mark2 syntax determines the list):

positive left subterm: (`?p || (?a ^+ ?b) , ?c`) = (`?p || ((?p || ?a , ?x) ^+ ?b) , ?c`)

negative left subterm: (`?p || ?a, ?b ^+ ?c`) = `?p || ?a , (?p || ?x, ?b) ^+ ?c`)

positive right subterm: (`?p || (?a ^+ ?b) , ?c`) = (`?p || (?a ^+ ?p || ?b, ?x) , ?c`)

negative right subterm: (`?p || ?a, ?b ^+ ?c`) = `?p || ?a, ?b ^+ (?p || ?x, ?c)`)

positive value: (`?p || [?a] , ?b`) = (`?p || [?p || ?a , ?x] , ?b`)

negative value: (`?p || ?a, [?b]`) = (`?p || ?a , [?p || ?x , ?b]`)

limited hypothesis: (`(?a = ?b) || ?a , ?c`) = (`(?a = ?b) || ?b , ?c`)

case introduction: `?x` = `?p || ?x , ?x`

The use of this list of properties allows one to avoid using the context $C[\ldots]$ in the hypothesis property by making use of the first six properties to bring a copy (or copies) of the hypothesis down to the level(s) of the occurrence(s) of one side of the equation which are to be replaced with the other, then applying the limited hypothesis property in place of the full hypothesis property. All instances of the distribution property can be proved by first applying case introduction to introduce a hypothesis at the top level of the term, then using the first six properties to propagate hypotheses into the term which can be used to eliminate the identical hypothesis where it occurs in subterms.

It requires a little work to show that the four properties given originally do imply the first six properties on the second list. It is sufficient to exhibit the general method of proof by proving the positive and negative value properties using the first four properties.
`?p || [?a] , ?b` =

`(true = ?p) || [true || ?a , ?x] , ?b` (by odd hypothesis and first projection) =
`(true = ?p) || [?p || ?a , ?x] , ?b` (by replacing `true` with `?p` using the hypothesis property) =
`?p || [?p || ?a , ?x] , ?b` (by odd hypothesis) .
The same technique works for all the positive properties. The negative properties are a little trickier:
`?p || ?a, [?p || ?x, ?b]`
= (odd hypothesis) `(true = ?p) || ?a, [?p || ?x , ?b]`
= (equation) `((true = ?p) || true , false) || ?a, [?p || ?x, ?b]`
= (distribution) `(true = ?p) || (true || ?a , [true || ?x, ?b]) , (false || ?a , [false || ?x , ?b])`
= (projection) `(true = ?p) || ?a , [?b]`
= (odd hypothesis) `?p || ?a, [?b]`.
The method of proof is similar for the other negative properties.

The equation, odd hypothesis, and case introduction hypotheses are provided as predeclared axioms by the prover. The subterm, value, and limited hypothesis properties are supported by "hard-wired" functions of the prover in a way that we will now describe.

The application of the subterm and value properties requires a strategy in which the hypothesis is duplicated ever deeper into the term being manipulated. The prover allows the application of such a strategy in effect without the syntactical cost by maintaining a list of hypotheses which are locally applicable and the senses in which they apply (positive or negative) as part of its navigation functions. The limited hypothesis property is available in the form of a "tactic" `0 |-| n`, where `n` is a numeral, which will replace the current subterm, if it is an instance of the left side of the `n`-th hypothesis (which needs to be an equation), with the right side; if `0 |-| n` is applied in the converse sense (using `<=` instead of `=>`), the right side of the hypothesis will be replaced with the left side. The subterm and value properties are available as a tactic `1 |-| n` (with a variant `2 |-| n`), which will eliminate an occurrence of the `n`-th hypothesis as the hypothesis of the current subterm, with the case selected to replace the subterm being determined by the sense of the `n`-th hypothesis. The converse will introduce a new occurrence of a hypothesis identical to the `n`-th hypothesis, with cases the current subterm and a new variable (or parameter supplied to the tactic, in the case of the `2 |-| n` variant form), their order being determined by the sense of the `n`-th hypothesis.

These built-in functions are fully supported in the tactic language; they can be introduced and executed automatically by recursive tactics. In this context, a phenomenon must be noted which will be even more marked in the abstraction layer: the notion of substitution can no longer be defined in the most naive way. The refinement needed is simple in this case: a term with instances of rewriting annotations `i |-| n` will need to have all the indices `n` incremented by a suitable amount if it is substituted into a conditional expression, so as to preserve the condition that each occurrence of `i |-| n` is a reference to the hypothesis of the `n`-th conditional expression from the top of the term containing the given rewriting annotation.

An effect of the demands of the conditional layer is that the prover always rewrites the hypothesis of a case expression as far as possible before doing anything with the cases, so that it will know the correct meanings of `i |-| n`'s that it encounters in the cases. Since the tactic interpreter automatically collapses case expressions with hypothesis `true` or `false`, this means that non-strict execution can occur; a subexpression which might otherwise not terminate may be eliminated due to the reduction of a hypothesis to a boolean value. This proves useful in working with recursively defined notions.

In the discussion of the abstraction layer below, there will be found a discussion of the reasons why an attempt to implement the functions of the conditional layer in pure rewriting terms, while possible, proved very inconvenient. The sense in which we think that it fails to be a pure rewriting system is that it involves the use of built-in tactics whose meaning depends on their context (as noted above, the alternative rewriting annotation mechanism already breaks the prover as a "pure" rewriting system, but not as definitively).

3.1 Examples of Conditional Layer Functions

We first give an example of the modified notion of substitution required with the introduction of this layer:

```
- start "(?a=?b)||?x,?y";
- assign "?x" "(?c=?d)||((0|-|1)=>?c),?e";
```

Notice that the expression to be substituted for the variable `?x` contains a rewriting annotation referring to the hypothesis `?c=?d`. When we carry out this assignment, the rewriting annotation is renumbered to preserve its reference:

```
{(?a = ?b)
   || ((?c = ?d) || ((0 |-| 2) => ?c) , ?e) , ?y}
```

Normally, a rewriting annotation would not be introduced in this way (`ruleintro` or its relations would be used) but this is the easiest way to exhibit this substitution phenomenon. It usually occurs, invisibly to the user, in the course of the execution of tactics.

Since the rewriting annotation in this term refers to the equation `?c=?d` and the term to which it is applied is `?c`, its execution has an interesting effect:

```
- execute();

{(?a = ?b) || ((?c = ?d) || ?d , ?e) , ?y}
```

The first example proof is a proof that the product of two real numbers can be zero iff one of the factors in the product is zero. We begin with basic axioms relating 0, 1, multiplication, and multiplicative inverse:

```
- declareinfix "*";
- axiom "COMM2" "?x*?y" "?y*?x";
- axiom "ASSOC2" "(?x*?y)*?z" "?x*?y*?z";
- axiom "TIMESZERO" "0*?x" "0";
- declareunary "|/";
- axiom "ONE" "1*?x" "?x";
- axiom "INV" "?x* |/?x" "(0=?x)||0,1";
- axiom "ZERONOTONE" "0=1" "false";
```

The form of the axiom of multiplicative inverse deserves comment. We treat the multiplicative inverse of 0 as an otherwise unspecified real number; thus, `?x * |/?x` will have the value 0 when `?x` is 0 (the axiom `TIMESZERO` would also allow us to prove this); in other cases, we get the expected result of 1.

We display the axiom of case introduction and prove a related tactic, which allows us to supply the hypothesis as a parameter.

```
CASEINTRO:
?x =
?y || ?x , ?x
["CASEINTRO"]

(* parameterized "CASEINTRO" *)

(*
PCASEINTRO @ ?p:
?x =
?p || ?x , ?x
["CASEINTRO"]
*)

- initializecounter();
- s "?x";
- ri "CASEINTRO"; ex();
- assign "?y_1" "?p";
- prove "PCASEINTRO@?p";
```

Finally, before beginning the proof, we introduce the logical connective of disjunction by definition from the case expression primitives:

```
- defineinfix "OR" "?p|?q" "?p || true, ?q || true , false";
```

This causes the declaration of the new connective and the introduction of the following theorem:

```
OR:
?p | ?q =
?p || true , ?q || true , false
["OR"]
```

The proof follows. We will show all prover commands, but only selected prover responses. Duplicate displays of a term and an identical selected subterm will be suppressed. Comments in the proof text itself may be useful.

```
- start "0=?x*?y";
- ri "PCASEINTRO@0=?x"; ex();
```

This command introduces the hypothesis `0 = ?x`, giving the display

```
{(0 = ?x) || (0 = ?x * ?y) , 0 = ?x * ?y}
```

We now go to the case where `0 = ?x` is true:

```
- right();left();  (* case where 0 = ?x *)
- right();left();  (* now looking at ?x *)
```

At this point we see the following:

```
(0 = ?x) || (0 = {?x} * ?y) , 0 = ?x * ?y

?x
```

Application of the built-in tactic `0 |-| 1` in its converse form will convert the selected subterm `?x` to `0`.

```
- rri "0|-|1"; ex();  (* converse changes ?x to 0 *)
- up();
- ri "TIMESZERO"; ex();  (* we now see a multiplication by 0 *)
- up();
```

We now see this:

```
(0 = ?x) || {0 = 0} , 0 = ?x * ?y

0 = 0
```

We can apply the axiom `REFLEX` seen above to convert the selected subterm to `true`, completing the proof of this case.

```
- ri "REFLEX"; ex();  (* proof of this case is complete *)
- up();right();
- ri "PCASEINTRO@0=?y"; ex(); (* the status of ?y is relevant if ?x is not 0 *)
```

At this point, we introduce the hypothesis `0 = ?y` (but only under the hypothesis that `0 = ?x` is false). We see the following:

```
(0 = ?x) || true
, {(0 = ?y) || (0 = ?x * ?y) , 0 = ?x * ?y}

(0 = ?y) || (0 = ?x * ?y) , 0 = ?x * ?y
```

We now "move" into the two cases thus defined in turn. The case where `0 = ?y` is handled in a manner virtually identical to the way in which the previous case is handled (but note the use of `0 |-| 2` instead of `0 |-| 1`).

```
- right();left();  (* the case where 0 = ?y *)
- right();right();
- rri "0|-|2"; ex();  (* we need to refer to hypothesis 2 here *)
- up();
- ri "COMM2"; ri "TIMESZERO"; ex();
- up();
- ri "REFLEX"; ex();  (* this completes the proof of this case *)
- up();right();

(* this takes us to the last case, which we need to show equal to false,
not true; where ?x and ?y are both non-zero, ?x*?y will be non-zero *)
```

We have now arrived at the last case, which is the really interesting one. We expect to be able to evaluate `0 = ?x * ?y` as `false` in this case. We do this by using the theorem `EQUATION`, which implements the "equation" auxiliary property of case expressions above, to split `0 = ?x * ?y`, then move into the case where `0 = ?x * ?y` is true, and show that under the accumulated hypotheses we can rewrite `true` into `false`; this case is contradictory. This is a standard method of proof in Mark2.

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

We pause to give the display at this point, then resume the proof:

```
(0 = ?x) || true , (0 = ?y) || true
, {(0 = ?x * ?y) || true , false}

(0 = ?x * ?y) || true , false

(* proof resumes *)

- right();left();  (* we move to the subterm "true" *)
- initializecounter();
- rri "REFLEX"; ex();
- assign "?a_1" "?x*?y*(|/?x)* |/?y";
```

We have now rewritten the instance of `true` to which we moved to the following form:

```
(?x * ?y * (|/ ?x) * |/ ?y) = ?x * ?y * (|/ ?x)
* |/ ?y
```

We use a prover command to display the current hypotheses:

```
- lookhyps();
1 (negative):
0 = ?x
2 (negative):
0 = ?y
3 (positive):
0 = ?x * ?y
```

Our strategy is to show that we can use these hypotheses to rewrite one of the instances of `?x * ?y * (|/ ?x) * |/ ?y` to 0 and the other to 1, which will enable us to rewrite the equation to `false`!

```
- left();
- rri "ASSOC2"; ex();
```

The selected subterm is now `(?x * ?y) * (|/ ?x) * |/ ?y`, after regrouping, which will allow us to move left and use hypothesis 3 followed by the axiom `TIMESZERO` to convert the left side of the equation to 0.

```
- left();
- rri "0|-|3"; ex();
- up();
- ri "TIMESZERO"; ex();
- up();right();   (* we move to the right side of the equation *)
- rri "ASSOC2";rri "ASSOC2"; ex();
- left();left();
- ri "COMM2"; ex();
- up();
- ri "ASSOC2"; ex();
- right();
```

The right side of the equation now has the form `0 = (?y * {?x * |/ ?x}) * |/ ?y`, where the braces indicate the position of the selected subterm. The selected subterm `?x * |/ ?x` is a target for the axiom `INV` governing the multiplicative inverse.

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

We pause to display the entire situation.

```
(0 = ?x) || true , (0 = ?y) || true
, (0 = ?x * ?y)
|| (0 = (?y * {(0 = ?x) || 0 , 1}) * |/ ?y)
, false

(0 = ?x) || 0 , 1
```

The built-in tactic `1 |-| 1` can now be used to collapse the selected subterm to its negative case, since its hypothesis is identical to hypothesis 1.

```
- ri "1|-|1"; ex();
- up();
- ri "COMM2"; ri "ONE"; ex();
up();
```

The right side of the equation now has the form `?y * |/ ?y`, to which we can apply the same strategy of applying `INV` and collapsing the resulting case expression (but using `1 |-| 2`, since hypothesis 2 is now the relevant one).

```
- ri "INV"; ex();
- ri "1|-|2"; ex();
- up();
```

We have now converted the equation to `0 = 1`, which an axiom allows us to rewrite to `false`.

```
- ri "ZERONOTONE"; ex();
- up();up();
```

We again display the entire situation.

```
(0 = ?x) || true , (0 = ?y) || true
, {(0 = ?x * ?y) || false , false}

(0 = ?x * ?y) || false , false
```

Rewriting with the converse of `CASEINTRO` will convert the selected subterm to `false`.

```
- rri "CASEINTRO"; ex();
- top();
```

The resulting top-level expression is `(0 = ?x) || true , (0 = ?y) || true , false`, which admits rewriting using the converse of the definition of logical disjunction given above, which gives us the final form of the theorem.

```
- rri "OR"; ex();
- prove "FACTORZERO";

FACTORZERO:
0 = ?x * ?y =
(0 = ?x) | 0 = ?y
["ASSOC2","CASEINTRO","COMM2","EQUATION","INV","ONE","OR","REFLEX",
"TIMESZERO","ZERONOTONE"]
```

This proof is rather lengthy, but it should be noted that it is on a completely "nuts-and-bolts" level; a standard proof would presume some rules of inference for propositional logic which are being handled here by explicit manipulation of case expression structures. Some of its elements could be facilitated by tactics in a richer environment; for example, tactics could carry out the algebraic regroupings in the last case in single steps.

Our final example is a proof of the commutative property of disjunction (as defined above), suggesting how case expression machinery can be used to reason in propositional logic. A complete tautology checker is an involved but not difficult tactic to develop.

```
- s "?p|?q";
- ri "OR"; ex();
- ri "PCASEINTRO@?q"; ex();
```

We introduce the starting term `?p|?q` and expand the expression by introducing the hypothesis `?q`. We see

```
?q || (?p || true , ?q || true , false) , ?p
   || true , ?q || true , false
```

Our strategy will be to eliminate the two occurrences of `?q` as hypothesis of proper subterms of this case expression and apply obvious rewrites to the results.

```
- right();left();
- right();right();
- ri "1|-|1"; ex();
```

This is the situation after the collapse of the hypothesis `?q` in the positive case. The obvious rewriting opportunity is to apply the converse of case introduction to the conditional expression with `true` as both of its cases.

```
?q || (?p || true , {true}) , ?p || true , ?q
|| true , false

true
```

We carry this out and proceed to the other case.

```
- up();up();
- rri "CASEINTRO"; ex();
- up();right();
- right();right();
- ri "1|-|1"; ex();
```

This is the form of the expression after the collapse of the hypothesis `?q` in the negative case. This has the correct form to be the target of the converse of the definition of disjunction; we go to the top and cmplete the proof.

```
?q || true , ?p || true , {false}

false
(* the proof resumes *)
- top();
- rri "OR"; ex();
- p "ORCOMM";

ORCOMM:
?p | ?q =
?q | ?p
["CASEINTRO","OR"]
```

4 The Abstraction Layer

4.1 Definition

The simplest form of definition does not involve us in abstraction: this is a definition of a constant as seen above:

```
- defineconstant "C" "S . (S . (K . S) . (S . (K . K) . S)) . (K . K)";
```

in which a new atomic constant is introduced and given a meaning. The prover does need to check that the new definition does not involve any objects not yet declared; this enforces non-circularity, because it is also required that the object to be defined has not been declared already.

As we have seen, Mark2 does allow the definition of operators as in the example

```
- defineinfix "OR" "?p|?q" "?p || true, ?q || true , false";
```

These are actually not problematic either.

Our first encounter with the functions of the abstraction layer is when we attempt to define constants or operators with parameters, i.e., when we define functions or operators on functions, as in

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

or

```
- defineinfix "ADD_FUN" "(?f ++ ?g) @ ?x" "(?f @ ?x) + ?g @ ?x"
```

The additional parameter is needed in the second example as the name of the theorem which will be generated by the definition process.

Such definitions cannot be allowed without some restrictions. Clearly we can define the negation operator of propositional logic thus:

```
- defineinfix "NOT" "~?x" "(true = ?x) || false, true";
```

(here we follow the convention of Frege that any object other than the truth value "true" should be treated as being false).

We could then (if parameterized definitions were unrestricted) make the following curious definition:

```
- defineinfix "Curry_Paradox @ ?x" "~ ?x @ ?x";
```

We would then find that `Curry_Paradox @ Curry_Paradox` was equal to its own negation by definition, from which the built-in tactics and predeclared axioms of the conditional layer would suffice to derive `true = false`.

Mark2 avoids such paradoxical definitions by a restriction on abstraction in general which is perhaps easiest to understand in definitions (where variable binding is not involved). The restriction can be thought of as a system of types, though objects are not typed in the underlying logic of Mark2. These ideas are based ultimately on the definition of Quine's set theory "New Foundations" ([21]). We have discussed their application to a λ-calculus in our [15].

The type system to which we make reference is one in which the types are indexed by the natural numbers. The intended relationship between successible types is that type $n + 1$ is the type of functions with type n input and type n output. Every Mark2 operator is assigned type information in the form of a pair of integers called its "left type" and "right type". The meaning of these numbers is that if an operator `+` has left type L and right type R, and a term `T + U` is assigned type N, then `T` must be assigned type $N + L$ and `U` must be assigned type $N + R$. The intended relation of the type scheme to function application dictates that the left type of `@` be 1 and its right type 0. Similarly, the relation of the type scheme to function application indicates that if `[T]` is assigned type N, `T` should be assigned type $N - 1$. The ordered pair and equality operators are assigned left and right types of 0, as is the conditional-building infix.

A term is said to be *stratified* if and only if each free variable in the term is assigned the same type wherever it appears (when the entire equation is assigned any fixed type). A definition type-checks correctly in Mark2 if the equation term representing the theorem proposed as a definition is stratified. For example, the theorem `ADD_FUN` proposed as a definition above is legitimate: if one assigns type 0 to the whole equation `((?f ++ ?g) @ ?x) = (?f @ ?x) + ?g @ ?x` then one assigns type 0 to `?x` and type 1 to `?f` and `?g` wherever they appear. One assumes here that `+` has left and right type 0; one knows that `++` will have left and right type 0 because it is introduced by the `declareinfix` command. Special `declaretypeinfix` and `definetypedinfix` commands are needed to declare or define operators with nonzero left and/or right types. The definition of `Curry_Paradox` is not stratified: there is no way to assign a single type to `?x` in the expression `?x @ ?x`, which is enough to show that there is no way to do this in the full proposed defining theorem.

Although a notion of "relative type" of terms is employed in definitions, there are no type restrictions on the formation of Mark2 terms. `?x @ ?x` is a perfectly valid term in Mark2, though it cannot be defined as a function of `?x`.

We will see in later sections that some refinements of the notion of stratification as described here will be needed.

The freedom from paradox of permitting all function definitions which can be stratified follows from the consistency of *NFU*, the version of Quine's set theory "New Foundations" ([21]) which was shown by R. B. Jensen to be consistent relative to the usual set theory in [18]. The results of Jensen's paper are sufficient for the level of mathematical strength needed here; for the equivalence of formulations in terms of functions to formulations in terms of sets, see the author's [11], [15], [12] and [17].

4.2 Stratified λ-Calculus

It is often convenient to be able to introduce functions without having to define a new name for each function used. Mark2 supports a notation for functions using what amounts to λ-abstraction.

Up to this point, we have not allowed bound variables in our terms, and bracketed terms `[T]` have been understood to refer to constant functions with the value `T`. The referent of a bracketed term `[T]` is actually a λ-abstraction; the variable bound in a given bracket is `?1` if it is the outermost bracket in the term, `?2` if it is a proper subterm of exactly one bracket term, and, in general, `?n` if it is a proper subterm of exactly $n-1$ bracket terms. This scheme of "de Bruijn levels" (not de Bruijn indices!) is derived from de Bruijn's [5].

Furthermore, there is a stratification restriction on the formation of bracket terms: the relative type of the variable bound in a bracket term `[T]` must be the same as the type assigned to `T` everywhere that it occurs in `T` (both of these types are one less than the type assigned to `[T]` itself). The term `[~ ?1 @ ?1]` must be ill-formed for the same reason that the definition of `Curry_Paradox` above must fail.

It should be noted that the notion of stratification applied in checking the well-formedness of a bracket term is not the same as the notion of stratification applied in checking definitions; the former is a restriction on free variables, whiel the latter is a restriction on bound variables. Bracket terms appearing in definitions are required to be stratified in the sense applying to bracket terms; there is no restriction on free variables appearing with more than one type in stratified bracket terms.

The use of variable binding necessitates a further change in the definition of substitution; the use of the particular variable-binding scheme we have adopted minimizes the complexity of this change.

The required redefinition of substitution (and also of matching) is driven by the need to preserve the correlations of bound variables with their binding contexts.

Each subterm of a given term can be assigned a "level", which is the number of bracket terms of which the occurrence is a proper subterm (we are counting multiple occurrences of subterms as distinct subterms). This natural number corresponds in its role to the more complex "environment" which we would need to keep track of in an implementation of conventional variable binding. (A similar notion of level appears in the detailed implementation of substitutions involving the built-in tactics of the conditional layer).

Each subterm has a minimum level at which it can appear. For example, the term `?1` can appear only at levels ≥ 1, since the variable `?1` will otherwise have no binding context. Certain typographically identical terms have different meanings at different levels: for example, `[?1]` refers to the identity function at level 0, but to the constant function with value `?1` at each higher level. This can be inconvenient, and would be avoided if the scheme of deBruijn indices were used instead of deBruijn levels (or if conventional variable binding were used). The advantage of deBruijn levels over deBruijn indices is that a bound variable has the same name throughout its binding context. The advantage of either of the deBruijn schemes over conventional variable binding is the extreme simplicity of the "environment".

In a subterm appearing at level L, variables `?n` with $n > L$ are bound in the subterm ("locally bound") and variables `?n` with $n \leq L$ are "locally free" (they are bound in a larger context). A term appearing at a level $M > L$ will match this term iff it is the result of increasing the index of each locally bound variable by $M - L$ and leaving the indices of "locally free" variables alone. Observe that the resulting term will have no variables `?n` with $L < n \leq M$. This constraint applies when we discuss the reverse direction of matching: a term appearing at a level $M < L$ will match a given term at level L iff it contains no `?n` with $M < n \leq L$ and it is the result of decreasing the index of each locally bound variable by $L - M$ while leaving the indices of locally free variables alone (a locally free variable should not become locally bound in the course of a substitution). The definition of substitution follows the definition of matching: a term to be substituted into a context at level L from a context at level M is replaced in the level L context by a term that would match its occurrence at level M.

The navigation facility of Mark2 keeps track of the level of the current term, and uses this to control the matching and substitution involved in the application of rewrite rules. The implementation of the tactic language also needs to take the changing levels of terms at which rewriting annotations are generated and "executed" automatically in the course of the execution of a tactic. The `lookhyps` command of the conditional layer always displays hypotheses in the form appropriate to the current level (promoting bound variables in hypotheses so that they have the same meanings as bound variables in the currently selected subterm). Our experience is that users adapt successfully to the use of deBruijn levels in place of conventional bound variables.

Mark2 supports the semantics of bracket terms as functions with two powerful built-in tactics, `BIND` and `EVAL`. The tactic `BIND` takes a parameter: the effect of execution of `BIND @ T` is to rewrite the target term into a bracket term applied to `T`, if stratification restrictions permit. It is similar to `ABS` in the combinatory logic example above. `EVAL`, when applied to a term of the form `[T] @ U`, will carry out the indicated function application (this will always succeed). It is analogous to `RED` in the example above.

A further refinement of the definition of matching in connection with function notation has proved useful. This is a limited form of higher order matching. In a bracket in which the bound variable is `?n`, a term of the form `?f @ ?n`, where

?f is a free variable, is taken to match any term T which may correspond to it in position, with ?f being taken to represent [T]. Notice that no stratification problem can arise, since a term matching ?f @ ?n in this way would necessarily be a subterm of a similar bracket term; stratification or meaningless bound variable difficulties would not arise. A simple example should clarify what is meant:

```
-  defineinfix"TIMESFUN" "(?f**?g)@?x" "(?f@?x)*?g@?x";

TIMESFUN:
(?f ** ?g) @ ?x =
(?f @ ?x) * ?g @ ?x
["TIMESFUN"]
```

The notion of multiplication of functions is defined.

```
- start "[(?f@?1)*?g@?1]@?x";
- ri "EVAL"; ex();

{EVAL => [(?f @ ?1) * ?g @ ?1] @ ?x}

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

- rri "TIMESFUN"; ex();
{TIMESFUN <= (?f @ ?x) * ?g @ ?x}

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

- p "TIMESFUNABSTRACT";

TIMESFUNABSTRACT:
[(?f @ ?1) * ?g @ ?1] @ ?x =
(?f ** ?g) @ ?x
["TIMESFUN"]
```

This theorem expresses an obvious relationship between products of functions and abstractions from products.

```
- s "[(?1*?1)*(?1*?1*?1)]@?x";
{[(?1 * ?1) * ?1 * ?1 * ?1] @ ?x}

- ri "TIMESFUNABSTRACT"; ex();
{TIMESFUNABSTRACT => [(?1 * ?1) * ?1 * ?1 * ?1]
   @ ?x}

{([?1 * ?1] ** [?1 * ?1 * ?1]) @ ?x}
```

The higher-order matching function of the prover is seen to operate in the fact that the prover can recognize that the theorem TIMESFUNABSTRACT applies at all: Mark2 needs to recognize the terms `?1 * ?1` and `?1 * ?1 * ?1` as being of the forms `?f @ ?1` and `?g @ ?1`, which they are not, superficially; the functions matching `?f` and `?g` are constructed by abstraction (`[?1 * ?1]` and `[?1 * ?1 * ?1]` are the functions constructed). Just as there is a change in the definition of what it is to match a term of the form `?f @ ?n`, so there is a change in the notion of substitution into a term `?f @ ?n`; if `?f` is to be replaced with an abstraction, Mark2 will replace `?f @ ?n` with the body of that abstraction (the result of removing its brackets). A simple example follows:

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

{[?f @ ?1]}

- assign "?f" "[?1*?1]";

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

This feature adds no logical strength to the prover, but it makes it possible to avoid many routine applications of `EVAL` and `BIND` which would otherwise be required.

4.3 Examples of Abstraction Layer Functions

We present some examples of the functions of this layer. First of all, we try to enter an impossible term:

```
- start "[?1@?1]";

MARK2:  Meaningless bound variable or unstratified abstraction error

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

The prover warns that the term is impossible; it displays it but will not allow a theorem to be proved from it or even allow it to be backed up onto the desktop as a saved environment.

```
- start "?1";

MARK2:  Meaningless bound variable or unstratified abstraction error

{?1}
```

Similarly, the prover will not allow a term to be entered with a bound variable which is not bound by any bracket.

Abstraction terms are usually introduced using the `BIND` tactic:

```
- start "?x";

{?x}

- ri "BIND@?x"; ex();  (* introduce the identity function *)
{(BIND @ ?x) => ?x}

{[?1] @ ?x}

- ri "EVAL"; ex();  (* this tactic reverses the effect of BIND *)
{EVAL => [?1] @ ?x}

{?x}

- ri "BIND@?y"; ex();  (* introduce a constant function *)
{(BIND @ ?y) => ?x}

{[?x] @ ?y}
```

The term `[?1]` represents the identity map (when it occurs at level 0). The term `[?x]` represents the constant function with value `?x` everywhere.

```
- start "[?x]";

{[?x]}

- assign "?x" "[?1]";

{[[?2]]}
```

This is an example of bound variable renumbering. The assignment of the value `[?1]` to `?x` makes this object the constant function of the identity function. Clearly, the bound variable needs to be bound by the innermost set of brackets, so its index will be 2 instead of 1.

```
- start "[[?1]]";
MARK2:  Meaningless bound variable or unstratified abstraction error

{[[?1]]}
```

The prover rejects the term `[[?1]]`. This would stand for the function which takes each object `?1` to its constant function `[?1]` (the K combinator of `CL`), but this abstraction does not type correctly in our scheme: the bound variable `?1` is assigned relative type -2 in this term, and for it to be stratified, it would have to have type -1, one lower than the type of the whole term instead of two lower.

We now attempt to construct an abstraction representing the composition operation (a relative of the B combinator).

```
- start "?f@?g@?x";

{?f @ ?g @ ?x}

- ri "BIND@?x";

{(BIND @ ?x) => ?f @ ?g @ ?x}

- ex();

{[?f @ ?g @ ?1] @ ?x}

- left();

{[?f @ ?g @ ?1]} @ ?x

[?f @ ?g @ ?1]

- ri "BIND@?g";

{(BIND @ ?g) => [?f @ ?g @ ?1]} @ ?x

(BIND @ ?g) => [?f @ ?g @ ?1]

- ex();

{[[?f @ ?1 @ ?2]] @ ?g} @ ?x

[[?f @ ?1 @ ?2]] @ ?g

- left();

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

[[?f @ ?1 @ ?2]]
- ri "BIND@?f"; ex();

({(BIND @ ?f) => [[?f @ ?1 @ ?2]]} @ ?g) @ ?x

(BIND @ ?f) => [[?f @ ?1 @ ?2]]

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

[[?f @ ?1 @ ?2]]
```

At this point our attempt breaks down. The relative type of `?f` in the selected subterm is 0 (the same as the type of the selected subterm itself) instead of -1 as would be required for the abstraction to succeed. The B combinator itself is not stratified. The problem can be seen in the fact that f and g are at the same type in $f(g(x))$, and at two different types in $B(f)(g)(x)$.

We take a different tack to solve the problem successfully:

```
- start "?f@?g@?x";

{?f @ ?g @ ?x}

- assign "?f" "P1@?h";

{(P1 @ ?h) @ ?g @ ?x}

- assign "?g" "P2@?h";

{(P1 @ ?h) @ (P2 @ ?h) @ ?x}
```

P1 and P2 are the projection operators associated with the predeclared ordered pair operator (,); their definition is in the logical preamble, a collection of declarations which are automatically run when the prover is invoked.

```
- ri "BIND@?x";
{(BIND @ ?x) => (P1 @ ?h) @ (P2 @ ?h) @ ?x}

- ex();

{[(P1 @ ?h) @ (P2 @ ?h) @ ?1] @ ?x}

- left();

{[(P1 @ ?h) @ (P2 @ ?h) @ ?1]} @ ?x

[(P1 @ ?h) @ (P2 @ ?h) @ ?1]

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

{(BIND @ ?h) => [(P1 @ ?h) @ (P2 @ ?h) @ ?1]} @ ?x

(BIND @ ?h) => [(P1 @ ?h) @ (P2 @ ?h) @ ?1]

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

[[(P1 @ ?1) @ (P2 @ ?1) @ ?2]] @ ?h
```

We now define a function implementing composition based on the development above and prove that it does what we intend it to do. An EVERYWHERE tactic more general than the one explicitly developed above for *CL*, but with essentially the same function, is used to shorten this proof somewhat.

```
- defineconstant "Comp" "[[(P1 @ ?1) @ (P2 @ ?1) @ ?2]]";

(* defining theorem suppressed *)

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

{(Comp @ ?f , ?g) @ ?x}

(Comp @ ?f , ?g) @ ?x

- ri "EVERYWHERE@Comp"; ex();

{(EVERYWHERE @ Comp) => (Comp @ ?f , ?g) @ ?x}

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

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

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

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

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

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

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

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

- up();

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

- ri "EVAL"; ex();

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

{(P1 @ ?f , ?g) @ (P2 @ ?f , ?g) @ ?x}

- ri "EVERYWHERE@P1"; ex();
```

```
{(EVERYWHERE @ P1) => (P1 @ ?f , ?g)
   @ (P2 @ ?f , ?g) @ ?x}

{?f @ (P2 @ ?f , ?g) @ ?x}

- ri "EVERYWHERE@P2"; ex();

{(EVERYWHERE @ P2) => ?f @ (P2 @ ?f , ?g) @ ?x}

{?f @ ?g @ ?x}

- prove "Comp_Thm";

Comp_Thm:
(Comp @ ?f , ?g) @ ?x =
?f @ ?g @ ?x
["Comp","P1","P2"]
```

The construction of the `Comp` abstraction exemplifies the fact that "curried" arguments (as in $f(x)(y)$) are not interchangeable in function in the Mark2 logic with paired arguments (as in $f(x, y)$); the curried arguments x and y in the example are at different relative types, while the paired arguments are at the same relative type.

4.4 Quantification, Types and Retractions

The function of the abstraction layer is two-fold: it handles function definition, as we have already indicated, but it also handles quantification. The fluent handling of quantification necessitates a further refinement of the definition of stratification, which has the effect of providing Mark2 with logical machinery for handling data types.

The definition of the operations of first-order logic presents no difficulties. The operations of propositional logic are all definable using functions of the conditional layer alone (we have already seen definitions of disjunction and negation, from which definitions of all propositional connectives can be derived). A technical problem is that Mark2 is "applicative" in the sense that any operation can be applied to any object (this is a term introduced by Curry in [4]); thus, it is necessary to consider the effects of propositional connectives on non-truth-values, and it is sometimes necessary to use the operator `|- ?p`, defined as `?p || true , false`, to force an object to be a truth value. For example, the double negation law becomes `~ ~?p = |- ?p`.

Universal quantification is defined using abstraction as follows:

```
- defineconstant "forall@?P" "[?P@?1] = [true]";

forall:
```

```
forall @ ?P =
[?P @ ?1] = [true]
["forall"]
```

The universal quantifier is a function which, applied to a propositional function (or, in fact to any function at all) returns `true` if the target is a function whose value is `true` everywhere, and `false` if the target is a function whose value fails to be `true` somewhere (in the case of a propositional function, this would imply that its value was `false` there, but the definition needs to apply to all functions, for which this is not always the case). The existential quantifier is then easily defined.

The logical machinery already provided is adequate for reasoning in first-order logic, but the form in which the reasoning would have to be represented would be quite peculiar. The difficulty which arises is that many natural formulas of first-order logic are unstratified if represented in the most obvious way, and the circumlocutions required to avoid this problem would make the system very annoying to use.

An example makes the problem clear:

```
- start "forall@[forall@[?1=?2]]";

MARK2:  Meaningless bound variable or unstratified abstraction error
```

The term entered should be the natural way to represent $(\forall x.(\forall y.x = y))$. Equally clearly, this term is unstratified: the type assigned to `?1` would be -2, and it must be -1 for the outermost bracket to be stratified.

The reason that this does not imply a lack of strength in the logic of Mark2 is that it is possible to raise and lower types of terms known to be truth values freely. If p is a truth value, then the function application $(\textbf{if } p \textbf{ then } \pi_1 \textbf{ else } \pi_2)(\textbf{true},\textbf{false})$ is equal to p, but the occurrence of p in this expression is at relative type 1 instead of 0. Similarly, $(\lambda x.p) = (\lambda x.\textbf{true})$ is an expression equivalent to p (if p is a truth value) in which the relative type of p is -1. By iterating these constructions, the type of an expression known to be a truth value can be freely manipulated in order to get stratification conditions to hold. In the expression above, manipulating the type of the subexpression `forall@[?1=?2]` (raising it by one) would be sufficient to convert the whole expression to a stratified form.

However, a user of a theorem proving system should not have to carry out such operations explicitly. Experience with earlier versions of the prover convinced us that fluent reasoning with nested quantifiers would remain impossible unless the prover was given the ability to recognize expressions whose types could be raised and lowered freely, and so given the ability to recognize a wider range of terms as being stratified. In the case of truth values, the extended definition of stratification is already in place in the related set theories, in which one does not assign types to propositions at all, considering only relationships between terms within atomic propositions in determining relative types.

The solution we adopted is more general than one which simply allows truth values to be implicitly raised and lowered in type. A domain with the property that variables restricted to that domain can be freely raised or lowered in relative type is called a "strongly Cantorian set" in "New Foundations" and related systems. A set A (sets are naturally represented in Mark2 by characteristic functions) is strongly Cantorian iff the restriction of the K combinator (the operator which builds constant functions) to A is implemented by a function. In set theories, it is more natural to refer to the singleton set construction rather than the constant function construction. We discuss these ideas in detail in our [17].

Two functionalities were added to the prover, a basic functionality allowing the declaration of strongly Cantorian sets, and an auxiliary functionality making it much easier to use the basic functionality.

There is a predeclared operator `:` such that for any fixed `t`, `t : ?x` is taken to represent the result of applying a retraction to `?x` whose range is a strongly Cantorian set, the identity of which depends on `t`. The prover knows this because there is a predeclared axiom which asserts that `(t : t : ?x) = t : ?x` (applying `t` with the colon operator is a retraction) and because the definition of stratification is modified to allow the type of a term of the form `t : ?x` to be raised and lowered freely (with a uniform effect on the types of its subterms).

To solve the problem of quantification, it is sufficient to declare an atom `bool` with the defining axiom `(bool : ?p) = true = ?p`; this tells the prover that any term `bool:?p` can have its relative type freely raised and lowered. The term `forall@[bool:forall@[?1=?2]]` will be stratified.

This is the basic functionality. The term `t` in terms of the form `t : ?x` is referred to as a "type label", and that is a good indication of the way in which such terms are used. All domains commonly used as data types in computer science may be assumed to be strongly Cantorian sets, and the class (it is not a set) of strongly Cantorian sets is closed under the usual type constructors. Elsewhere (in [12]), we have argued that an identification of the notion "strongly Cantorian set" with the notion of "data type" is reasonable in the context of systems like "New Foundations".

The auxiliary functionality added to the prover which makes the basic functionality of labels for strongly Cantorian types more usable is the ability to allow the prover to recognize that certain terms belong strongly Cantorian domains even though they do not have explicit type labels. For any operator `+`, if there is a theorem of the form `?x + ?y` = `t : ?x + ?y`, we say that the operator is "scout" (this is short for "has strongly Cantorian output"), and a declaration can be made to the prover which will cause the prover to recognize any term of the form `?x + ?y` as capable of being freely raised or lowered in type. An operator `+` for which we have a theorem `?x + ?y = (s : ?x) + t : ?y` may be declared "scin" (short for "has strongly Cantorian input"; this stronger property allows the left and right subterms of a term with this operator to be raised or lowered in type independently of one another. Functions can also be declared "scin" or "scout" in the presence of suitable theorems: for example, the theorem `forall @ ?p = bool :  forall @ ?p` allows us to declare `forall` as "scout", and this allows the prover to recognize the problem term `forall@[forall@[?1=?2]]` as stratified without any type labels needing to be inserted. This declaration, along with declarations of the propositional connectives as "scin", completely removes all stratification problems with a natural notation for first-order logic.

This is a system of type inference with the curious feature that it is not necessary for the prover to know what type any term has; it is merely necessary for the stratification function of the prover to be able to determine that a term belongs to *some* (strongly Cantorian) type.

The enhanced definition of stratification is considerably more complex to implement, but the gain in fluency is crucial. Stratification as originally defined is a completely local property; as we will see in examples, the new definition of stratification, while it allows a broader class of terms to be recognized as stratified, makes the recognition of failures of stratification a little harder.

4.4.1 Examples of Extension of Stratification

We will work in this section with the type `bool` of booleans.

```
-declareconstant "bool";

-axiom "BOOL" "bool:?x" "true=?x";

BOOL:
bool : ?x =
true = ?x
["BOOL"]
```

These declarations introduce the boolean type.

```
start "forall@[forall@[?1=?2]]";

MARK2:  Meaningless bound variable or unstratified abstraction error
```

As above, the system will not allow us to enter this obviously legitimate quantification. We show how to remedy this situation. The crucial point is that the operator `=` is "scout" (its output is always of type `bool`).

```
- start "bool:?x=?y";

{bool : ?x = ?y}

- ri "BOOL"; ex();

{BOOL => bool : ?x = ?y}

{true = ?x = ?y}
```

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

{EQUATION => true = ?x = ?y}

{(true = ?x = ?y) || true , false}

- rri "ODDCHOICE"; ex();

{ODDCHOICE <= (true = ?x = ?y) || true , false}

{(?x = ?y) || true , false}

- rri "EQUATION";

{EQUATION <= (?x = ?y) || true , false}

{?x = ?y}

- prove "EQBOOL";

EQBOOL:
bool : ?x = ?y =
?x = ?y
["BOOL","EQUATION","ODDCHOICE"]

- makescout "=" "EQBOOL";
```

The theorem `EQBOOL` is accepted by Mark2 as witnessing the fact that `=` has boolean output. The term we had trouble with above still will not work, because we also need to know that the quantifier `forall` has boolean output:

```
- s "forall@?P";

{forall @ ?P}

- ri "forall"; ex();

{forall => forall @ ?P}

{[?P @ ?1] = [true]}

- rri "EQBOOL"; ex();

{EQBOOL <= [?P @ ?1] = [true]}

{bool : [?P @ ?1] = [true]}
```

```
- right();

bool : {[?P @ ?1] = [true]}

[?P @ ?1] = [true]

- rri "forall"; ex();

bool : {forall <= [?P @ ?1] = [true]}

forall <= [?P @ ?1] = [true]

bool : {forall @ ?P}

forall @ ?P

- prove "ALLBOOL";

ALLBOOL:
forall @ ?P =
bool : forall @ ?P
["BOOL","EQUATION","ODDCHOICE","forall"]

- makescout "forall" "ALLBOOL";
```

Now we try out the offending quantification:

```
- start "forall@[forall@[?1=?2]]";

{forall @ [forall @ [?1 = ?2]]}
```

Of course, this is a false statement; one must not expect to see a proof of it! We present a proof that it is false (long, and in a dense format):

```
- ri "forall"; ex();
{forall => forall @ [forall @ [?1 = ?2]]}
{[forall @ [?1 = ?2]] = [true]}
- ri "EQUATION"; ex();
{EQUATION => [forall @ [?1 = ?2]] = [true]}
{([forall @ [?1 = ?2]] = [true]) || true , false}
- right();left();
([forall @ [?1 = ?2]] = [true]) || {true , false}
true , false
([forall @ [?1 = ?2]] = [true]) || {true} , false
true
- ri "BIND@?x"; ex();
```

```
([forall @ [?1 = ?2]] = [true])
|| {(BIND @ ?x) => true} , false
(BIND @ ?x) => true
([forall @ [?1 = ?2]] = [true]) || {[true] @ ?x}
, false
[true] @ ?x
- left();
([forall @ [?1 = ?2]] = [true]) || ({[true]} @ ?x)
, false
[true]
- rri "0|-|1"; ex();
([forall @ [?1 = ?2]] = [true])
|| ({(0 |-| 1) <= [true]} @ ?x) , false
(0 |-| 1) <= [true]
([forall @ [?1 = ?2]] = [true])
|| ({[forall @ [?1 = ?2]]} @ ?x) , false
[forall @ [?1 = ?2]]
- up();
([forall @ [?1 = ?2]] = [true])
|| {[forall @ [?1 = ?2]] @ ?x} , false
[forall @ [?1 = ?2]] @ ?x
- ri "EVAL"; ex();
([forall @ [?1 = ?2]] = [true])
|| {EVAL => [forall @ [?1 = ?2]] @ ?x} , false
EVAL => [forall @ [?1 = ?2]] @ ?x
([forall @ [?1 = ?2]] = [true])
|| {forall @ [?x = ?1]} , false
forall @ [?x = ?1]
- assign "?x" "true";
([forall @ [?1 = ?2]] = [true])
|| {forall @ [true = ?1]} , false
forall @ [true = ?1]
- ri "forall"; ex();
([forall @ [?1 = ?2]] = [true])
|| {forall => forall @ [true = ?1]} , false
forall => forall @ [true = ?1]
([forall @ [?1 = ?2]] = [true])
|| {[true = ?1] = [true]} , false
[true = ?1] = [true]
- ri "EQUATION"; ex();
([forall @ [?1 = ?2]] = [true])
|| {EQUATION => [true = ?1] = [true]} , false
EQUATION => [true = ?1] = [true]
([forall @ [?1 = ?2]] = [true])
|| {([true = ?1] = [true]) || true , false}
, false
```

```
([true = ?1] = [true]) || true , false
- right();left();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true]) || {true , false})
, false
true , false
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true]) || {true} , false)
, false
true
- ri "BIND@false";
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true])
   || {(BIND @ false) => true} , false) , false
(BIND @ false) => true
- ex();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true]) || {[true] @ false}
   , false)
, false
[true] @ false
- left();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true]) || ({[true]} @ false)
   , false)
, false
[true]
- rri "0|-|2"; ex();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true])
   || ({(0 |-| 2) <= [true]} @ false) , false)
, false
(0 |-| 2) <= [true]
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true])
   || ({[true = ?1]} @ false) , false) , false
[true = ?1]
- up();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true])
   || {[true = ?1] @ false} , false) , false
[true = ?1] @ false
- ri "EVAL";
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true])
   || {EVAL => [true = ?1] @ false} , false)
```

```
, false
EVAL => [true = ?1] @ false
- ex();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true]) || {true = false}
   , false)
, false
true = false
- ri "NONTRIV"; ex();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true])
   || {NONTRIV => true = false} , false) , false
NONTRIV => true = false
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true]) || {false} , false)
, false
false
- up();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true]) || {false , false})
, false
false , false
- up();
([forall @ [?1 = ?2]] = [true])
|| {([true = ?1] = [true]) || false , false}
, false
([true = ?1] = [true]) || false , false
- rri "CASEINTRO"; ex();
([forall @ [?1 = ?2]] = [true])
|| {CASEINTRO <= ([true = ?1] = [true]) || false
   , false}
, false
CASEINTRO <= ([true = ?1] = [true]) || false
, false
([forall @ [?1 = ?2]] = [true]) || {false} , false
false
- up();up();
([forall @ [?1 = ?2]] = [true]) || {false , false}
false , false
{([forall @ [?1 = ?2]] = [true]) || false , false}
([forall @ [?1 = ?2]] = [true]) || false , false
- rri "CASEINTRO"; ex();
{CASEINTRO <= ([forall @ [?1 = ?2]] = [true])
   || false , false}
CASEINTRO <= ([forall @ [?1 = ?2]] = [true])
|| false , false
```

```
{false}
- prove "NOT_SO";
NOT_SO:
forall @ [forall @ [?1 = ?2]] =
false
["BOOL","CASEINTRO","EQUATION","NONTRIV","ODDCHOICE","forall"]
```

This is a *very* low level proof, in which all the nuts-and-bolts of all three levels are being handled "by hand". The use of the `BIND` and `EVAL` to handle instantiation of universal statements should be noted.

Finally we do a more complex stratification example. To facilitate matters, we introduce a basic property of logical conjunction (`&`) as an axiom.

```
- declareinfix "&";

- axiom "ANDSCIN" "?x&?y" "(bool:?x)&bool:?y";

ANDSCIN:
?x & ?y =
(bool : ?x) & bool : ?y
["ANDSCIN"]

- makescin "&" "ANDSCIN";

- definetypedinfix "IN" 0 1 "?x<<?y" "true=?y@?x";

IN:
?x << ?y =
true = ?y @ ?x
["IN"]

- s "?x<<?y";

{?x << ?y}

- ri "IN"; ex();

{IN => ?x << ?y}

{true = ?y @ ?x}

- rri "EQBOOL"; ex();

{EQBOOL <= true = ?y @ ?x}

{bool : true = ?y @ ?x}
```

```
- right();

bool : {true = ?y @ ?x}

true = ?y @ ?x

- rri "IN"; ex();

bool : {IN <= true = ?y @ ?x}

IN <= true = ?y @ ?x

bool : {?x << ?y}

?x << ?y

- p "INSCOUT";

INSCOUT:
?x << ?y =
bool : ?x << ?y
["BOOL","EQUATION","IN","ODDCHOICE"]

makescout "<<" "INSCOUT";
```

The relation << represents the membership relation (boolean-valued functions represent sets). We now develop some "set definitions" and sentences about set theory somewhat in the style of "New Foundations".

```
- s "[?1=?1]";  (* the universal set *)

{[?1 = ?1]}

- s "[?1<<?1]";  (* complement of the Russell class *)

MARK2:  Meaningless bound variable or unstratified abstraction error

- s "[(?x<<?1)&(?1<<?x)]";  (* unobviously stratified *)

{[(?x << ?1) & ?1 << ?x]}

- s "forall@[forall@[forall@[(?1<<?2)&(?2<<?3)]]]";

(* a stratifiable assertion *)

{forall
   @ [forall @ [forall @ [(?1 << ?2) & ?2 << ?3]]]}
```

```
- s "forall@ [forall @ [forall@[(?1 << ?2) & (?2<<?3) & (?3<<?1)]]]";

(* this is unstratified (though any two of the conjuncts are OK) *)

MARK2:  Meaningless bound variable or unstratified abstraction error

{forall
   @ [forall
      @ [forall
         @ [(?1 << ?2) & (?2 << ?3) & ?3 << ?1]]]}
```

4.5 Synthetic Abstraction

An aspect of the original program of the Mark2 research which has been abandoned is an attempt to make use of synthetic abstraction instead of variable binding constructions. (We hope that we can be forgiven the minor remaining eccentricity of deBruijn level notation in place of the usual bound variables). In this section we will discuss the historical reasons why we started with a synthetic approach, the non-negligible work we did in the direction of implementing a synthetic approach, and the considerations which finally convinced us to abandon this approach. The development of the Mark2 tactic language was strongly impelled by the needs of synthetic abstraction and reduction algorithms, and the fitness of the tactic language for this purpose should be suggested by the extended example of implementation of untyped combinatory logic above.

Work on the Mark2 project was originally inspired by our definition in our Ph. D. thesis ([10]; better, see [11]) of a system of untyped synthetic combinatory logic precisely equivalent in strength and expressive power to "New Foundations". At the same time, we exhibited a version of this logic with a weakening of extensionality which has the same strength and expressive power as Jensen's NFU + Infinity, a mathematically adequate subsystem of "New Foundations" which is known to be consistent. The details of this system are not important in this context, because it was never directly implemented in the prover.

The system implemented in the first precursor of Mark2, the `EFTTP` system described in our original grant proposal [13], is a weaker system, a two-sorted synthetic combinatory logic equivalent in consistency strength and expressive power to first-order logic on infinite domains (first order logic plus sentences asserting "there are at least n objects" for each concrete natural number n). The details of this system have a great deal to do with the development of Mark2.

The system is called EFT, for "external function theory". It is described in our preprint [16], but most salient portions of that preprint are incorporated into this paper below. The prover which implemented it was called `EFTTP`; there are profound differences between `EFTTP` and Mark2, but an underlying family resemblance remains.

There are two sorts of term in the language of EFT, object terms and Function terms. Our typographical convention is that object variables and constants begin with lower-case letters, while Function variable and constants begin with upper-case letters. The intended interpretation is that the objects are the elements of some infinite set or proper class and that the Functions are the functions (possibly proper class functions) from that set into itself (or possibly a restricted class of these functions closed under suitable operations).

t and f are distinct atomic object constants (used to represent the truth values). If x and y are object terms, (x, y) is an object term, the ordered pair with projections x and y. If F is a Function term and x is an object term, $F[x]$ is an object term, the value of F at x or the result of application of F to x. Cond and Id are atomic Function terms. Id is intended to be the identity Function; $\mathrm{Cond}[(x, y), (z, w)]$ is intended to be z if $x = y$, w otherwise. If x is an object term, $|x|$ is a Function term, the constant Function of x. If F and G are Function terms, (F, G) is a Function term, the product of F and G, and $F\langle G\rangle$ is a Function term, the composition of F and G. If F is a Function term, $F!$ is a Function, called the "Hilbert Function" of F. $F![y]$ is intended to be an object such that $F[F![y], y]$ is not t, if there is any such object; its value is a matter of indifference otherwise. We write $F[x, y]$, $F\langle G, H\rangle$, instead of $F[(x, y)]$, $F\langle (G, H)\rangle$, respectively. We define the n-tuple $(x_1, x_2, ..., x_n)$ inductively as $(x_1, (x_2, ..., x_n))$.

EFT is an algebraic theory. This means that all sentences of EFT are equations between object terms of EFT. The rules of inference of EFT are as follows:

A. Reflexivity, symmetry, transitivity of equality.

B. If $a = c$, $b = d$ are theorems, $(a, b) = (c, d)$ is a theorem.

C. If $a = b$ is a theorem, $F[a] = F[b]$ is a theorem.

D. Uniform substitution of an object term for an object variable or of a Function term for a Function variable in a theorem yields a theorem.

Note that the rules do not directly permit substitutions of equals for equals where object terms appear as subterms of Function terms. This was reflected in the prover `EFTTP` by the fact that the navigation commands of the prover did not allow one to move to a Function subterm or to its object subterms.

The axioms are as follows:

(CONST) $|x|[y] = x$

(ID) $\mathrm{Id}[x] = x$

(PROD) $(F, G)[x] = (F[x], G[x])$

(COMP) $F\langle G\rangle [x] = F[G[x]]$

(PROJ1) $\mathrm{Cond}[(x, x), (y, z)] = y$

(PROJ2) $\text{Cond}[(t,f),(y,z)] = z$

(DIST) $F[\text{Cond}[(x,y),(z,w)] = \text{Cond}[(x,y),(F[z],F[w])]$

(HYP) $\text{Cond}[(x,y),(F[x],z)] = \text{Cond}[(x,y),(F[y],z)]$

(HILBERT) $\text{Cond}[(F[F![y],y],t),(F[x,y],t)] = t$

Axioms (PROJ1), (PROJ2), (DIST), and (HYP) should be recognizable as the basic properties of case expressions used in the discussion of the conditional layer of the prover above. Axiom (HILBERT) is used to handle quantification (by introducing an external choice operator); the other axioms are used to develop a synthetic abstraction algorithm.

It should be clear that the axioms are true in the intended interpretation (given the Axiom of Choice to support axiom (HILBERT)), and they should also serve to clarify the exact interpretations of the term constructions. If a version of the "intended interpretation" with a restricted class of functions interpreting the Functions of the theory is to be constructed, the axioms indicate the set of operations under which the restricted class of functions needs to be closed.

We have the following Abstraction Theorem:

Theorem: If s is an object term and x is an object variable which does not appear as a subterm of any Function subterm of s, there is a function term $(\lambda x)(s)$ such that x does not appear as a subterm of $(\lambda x)(s)$ and "$(\lambda x)(s)[x] = s$" is a theorem.

Proof: By induction on the structure of terms. If s is x, $(\lambda x)(s)$ is Id; if s is an atom a distinct from x, $(\lambda x)(s)$ is $|a|$. If s is of the form (u,v), $(\lambda x)(s) = ((\lambda x)(u),(\lambda x)(v))$. If s is of the form $U[v]$, U does not involve x and $(\lambda x)(s) = U\,\langle(\lambda x)(v)\rangle$.

$(\lambda xy)(s)$ such that "$(\lambda xy)(s)(x,y) = s$" is a theorem can be defined as $(\lambda z)(s_0)$, where z is a variable not appearing in s and s_0 is the result of replacing x with $\text{Cond}[(t,t),z]$ and y with $\text{Cond}[(t,f),z]$ wherever they appear in s. $(\lambda z)(\text{Cond}[(t,t),z])$ and $(\lambda z)(\text{Cond}[(t,f),z])$ are the projection Functions π_1 and π_2.

An interesting point about abstracts constructed following the proof of the Theorem is that they are parallel in structure to the term from which they are abstracted. This helped to make *EFT* appear to an environment in which it might be practical to avoid the use of bound variables.

Theorem: *EFT* is equivalent in deductive strength and expressive power to first-order logic with equality and the axiom scheme consisting of the assertions "there are n objects" for each concrete n (these can of course be expressed in first-order logic).

Proof: We begin the proof that *EFT* is equivalent to first-order logic with equality + "there are n objects for each concrete n" with a model-theoretic argument. Consider a first-order theory with infinite models. We construct a model of *EFT* from an infinite model of the first-order theory by letting a countably infinite model of the first order theory be the domain of objects and the set of all functions from the domain of objects into itself be the set of Functions. Since the domain of objects is infinite, there is a map from its Cartesian product with itself into itself; use any such map to define the pair of the interpretation of *EFT* (note that we could choose a map from the Cartesian product *onto* the domain of objects, getting a surjective pair). Since the model is infinite, it contains two distinct objects which can be designated as t and f. This is a case of the "intended interpretation" of *EFT* (we need the Axiom of Choice to provide functions interpreting Function terms $F!$ of *EFT*).

Observe that each predicate ϕ of the first-order theory, considered relative to a variable x, corresponds in a natural way to a function from the domain of objects (which includes the domains of interpreted n-tuples of objects for each n) to $\{t, f\}$, taking x for which ϕ holds to t and x for which $\sim \phi$ holds to f, and so can be interpreted as an *EFT* Function (we actually allow all *EFT* Functions to interpret predicates, by having values other than t or f interpret "false"). If F represents ϕ relative to x, ϕ will be represented by $F[x]$ (x may represent an n-tuple of objects as indicated below). Equality can be interpreted as $(\lambda xy)(\text{Cond}[(x, y), (t, f)])$. If p and q are objects interpreting propositions P and Q, $\text{Cond}[(p, t), (f, t)]$ interprets $\sim P$ and $\text{Cond}[((t, t), (p, q)), (t, f)]$ interprets $P\&Q$ (we recall the convention that an object other than t or f repesents the truth value "false"). Propositional logic can be interpreted in *EFT*. Thus, it is possible to interpret every quantifier-free sentence of the first-order theory. Using the abstraction theorem, we can express a quantifier-free sentence in the form $F[(x_1, x_2, x_3, ...)]$, with F containing none of the variables x_i. We now show how to quantify on x_1 and get an expression of the same form: $F[x, y] = t$ for all x exactly if $F[F![y], y] = F\langle F!, \text{I}d\rangle [y] = t$, so universal quantification with respect to x_1 yields $F\langle F!, \text{I}d\rangle [x_2, ...]$, an expression of the same form. If Neg abbreviates $(\lambda p)(\text{Cond}[(p, t), (f, t)])$, then $\text{Neg}\langle F\rangle [\text{N}eg\langle F\rangle![y], y] = t$ exactly if $\text{Neg}\langle F\rangle [x, y] = t$ for all x, so $F[\text{Neg}\langle F\rangle![y], y] = F\langle \text{N}eg\langle F\rangle!, \text{I}d\rangle [y] = t$ exactly if $F[x, y] = t$ for some y, and so existential quantification with respect to x_1 yields $F\langle \text{N}eg\langle F\rangle!, \text{I}d\rangle [x_2, ...]$, again an expression of the same form. It is thus possible to interpret all quantified sentences in prenex normal form, as long as a final dummy argument is provided in the interpretation of the quantifier-free sentence, which is easy. Every sentence in the first-order theory can thus be interpreted in the model of *EFT*.

We now consider the model theory of *EFT*. An "*EFT* theory" is defined as a set of equations in a language extending the language of *EFT* which contains the axioms and is closed under application of the rules of *EFT*. We show how to use an *EFT* theory to build a model of the intended interpretation of *EFT*, in a way which makes it clear that the interpretations of axioms and rules of first-order logic with equality are valid in *EFT*, establishing the equivalence of the two theories.

Fix an *EFT* theory T. We suppose that T has some collection of object and Function constants extending that of *EFT*. We assume that the set of terms of T can be well-ordered. The objects of the model of the intended interpretation which we will build will be equivalence classes of constant terms of T under the equivalence relation of being equated by a theorem of T. The Function terms of T will be interpreted as functions from interpreted objects to interpreted objects; to make this possible, we will interpret Function terms of the form $F!$ as actually having the form "F"!; we will assume that the exclamation-point operator acts on names of functions in T, not on functions themselves. The reason for this is that we have no guarantee that $F!$ and $G!$ will have the same extension when F and G have the same extension; the axioms ensure that the other constructions on Functions respect extension. (An alternative approach would be to add another axiom asserting that $F!$ and $G!$ have the same extension when F and G have the same extension.)

We now observe that $x =$ (by (CONST)) $|x|[\text{Cond}[(a,b),(u,v)]] =$ (by (DIST)) $\text{Cond}[(a,b),(|x|[u],|x|[v])] =$ (by (CONST)) $\text{Cond}[(a,b),(x,x)]$. We also prove the following

Lemma: If $x = y$ belongs to the minimal theory containing T and "$a = b$", then "$\text{Cond}[(a,b),(x,z)] = \text{Cond}[(a,b),(y,z)]$" belongs to T.

Proof: "$x = y$" belongs to the indicated theory if and only if there is a proof of "$x = y$" from T and "$a = b$". This proof can be presented in the form $x = w_1 = ... = w_n = y$, where w_1 is exactly x and w_n is exactly y, and each equation "$w_i = w_{i+1}$" is the result of a substitution of equals for equals based on an equation in T or the equation "$a = b$". Consider the chain of equations $\text{Cond}[(a,b),(w_i,z)] = \text{Cond}[(a,b),(w_{i+1},z)]$. If $w_i = w_{i+1}$ is based on a substitution of equals for equals on the basis of an equation in T, then this equation belongs to T. If $w_i = w_{i+1}$ is based on a substitution of a for b or vice-versa, we can use abstraction to expand it to the form $w_i = F[a] = F[b] = w_{i+1}$ (recalling that substitutions of equals for equals are only allowed by the rules of *EFT* for object terms which are not subterms of Function terms, so the abstraction theorem can be applied), and the more complex equation can be derived from a substitution instance of (HYP), so is an element of T.

This Lemma shows that all reasoning in theories extending T is actually encoded in reasoning in T itself. The converse of the Lemma is obviously true. Observe that from $t = f$ we can deduce $u = \text{Cond}[(t,t),(u,v)] = \text{Cond}[(t,f),(u,v)] = v$; a theory is universal (contains all equations) iff it contains $t = f$. We now observe that $\text{Cond}[(a,b),(u,v)] =$ (by (PROJ1) and (PROJ2))

Cond$[(a,b),($Cond$[(t,f),(v,u)]$,Cond$[(f,f),(v,u)])] =$ (apply abstraction and (DIST))
Cond$[($Cond$[(a,b),(t,f)],f),(v,u)]$.
A similar argument shows that Cond$[(a,b),(u,v)] =$ Cond$[($Cond$[(a,b),(t,f)],t),(u,v)]$.
The equation Cond$[(a,b),(t,f)] = t$ would be expected to be equivalent to $a = b$; it is easy to deduce the former from the latter; we deduce the latter from the former as follows: $b =$ (by a lemma proven above) Cond$[(a,b),(b,b)] =$ (by (HYP)) Cond$[(a,b),(a,b)] =$ (prev. paragraph) Cond$[($Cond$[(a,b),(t,f)],t),(a,b)] =$ (by hyp.) Cond$[(t,t),(a,b)] = a$. We expect the equation Cond$[(a,b),(t,f)] = f$ to correspond to the negation of $a = b$, and we see that it has at least one appropriate property; note that we can deduce Cond$[(a,b),(u,v)] = v$ from Cond$[(a,b),(t,f)] = f$ and the result above. It is clear that if $a = b$ and Cond$[(a,b),(t,f)] = f$, we can deduce $t = f$ and all other equations (by a substitution of a for b and (PROJ1)). We also observe that if an equation $c = d$ follows from T and $a = b$ and also follows from T and Cond$[(a,b),(t,f)] = f$, it follows from T alone: $c =$ Cond$[(a,b),(c,c)] =$ Cond$[(a,b),(d,c)] =$ Cond$[($Cond$[(a,b),(t,f)],f),(c,d)] =$ Cond$[($Cond$[(a,b),(t,f)],f),(d,d)] =$ d is a correct deduction in T.

We call an *EFT* theory "contradictory" iff it is the universal theory, and "consistent" if it is not contradictory (note that a theory is consistent iff it does not prove $t = f$). We can now conclude that for each consistent theory T and equation $a = b$ not in T, there is a consistent extension of T which contains either $a = b$ or Cond$[(a,b),(t,f)] = f$. From this we can deduce that each consistent theory T has a consistent extension which contains either $a = b$ or Cond$[(a,b),(t,f)] = f$ for each a and b (this requires the assumption that the set of terms can be well-ordered). We call a theory with this property "complete". We also observe that an equation is a theorem of T if and only if it holds in every complete consistent extension of T.

We now build two-valued models of the intended interpretation of T. Let T$'$ be a complete consistent extension of T. Let the objects of the model be the equivalence classes of object terms of the language of T under the relation "equated by a theorem of T$'$". Let the functions of the model be the functions from objects of the model to objects of the model induced in the obvious way by Function terms of T, recalling that the operation ! is taken to act on names of functions, not the functions themselves. Note that equality is actual equality, t and f are distinct (and as many distinct objects as desired can be built using pairing, t and f), the ordered pair is well-defined, and Id and Cond have the intended meanings, as do the product and composition operations on functions and the Hilbert operator. All such models are actually cases of the intended interpretation of *EFT*, and any equation true in all such models is a theorem of the theory T (if it were not a theorem, it would be consistent to adjoin its "negation" to T, which could be extended to a consistent complete theory). In the intended interpretation of *EFT*, the translations of theorems of first-order logic with equality given above are valid, so the translations of theorems of first-order logic with equality are valid theorems of T, and the equivalence of *EFT* and first-order logic with an external infinity of objects is established.

This proof is included here to support our claims about the logical adequacy of the conditional layer of the prover. Our implementation of quantification is different from the one used in *EFT*, since it relies on the presence of variable binding machinery.

The prover `EFTTP` which was the first precursor of Mark2 had term structure (and internal term data types) based exactly on the term structure of *EFT*, with an extension. A third sort of FUNCTORS was introduced: the only FUNCTOR terms are atomic constants representing operations on Functions. If `FN` is a FUNCTOR and `F` is a Function term, `FN`{`F`} is a Function term. The presence of pairing on Functions allowed the definition of FUNCTORS of more than one argument. An example of a FUNCTOR which was found useful was a functor `ITER` of iteration such that `ITER`{`F,|n|`} represented the Function "apply `F` n times". A further extension of the term language of `EFTTP` was the introduction of rewriting annotations, which were supported by a separate term construction rather than by a case of the usual operator construction as in Mark2; a result of this was that the idea of introducing parameterized tactics could not even be considered, as the data structures and parser of `EFTTP` did not allow for nonatomic "theorem names" (except for a unary operation for building converses of theorems).

Synthetic abstraction was very appealing in `EFTTP`, since the structure of Functions abstracted from terms was precisely parallel to the structure of the object term from which they were abstracted.

The reason why `EFTTP` was abandoned was syntactical. The syntax of *EFT* was a straitjacket on the system. The immediate motivation of the development of Mark2 was the desire to be able to use infix notation in ordinary algebra and arithmetic. In `EFTTP`, the associative law of addition (for example) had to be represented by

```
+[+[x?,y?],z?] = +[x?,+[y?,z?]]
```

At the same time, we observed the potentially polymorphic relationship between the sorts of object, Function, and FUNCTOR, which can be thought of as types 0, 1, and 2 in the type system underlying the relative type system of Mark2. Our already considerable familiarity with systems like "New Foundations" predisposed us to collapse these types together and adopt an untyped higher-order logic regulated by stratification rather than any absolute type scheme. Our annoyance at special term constructions went so far that we abandoned all distinctions of kind of term except that between constant functions and other terms, treating special term constructions such as ordered pair, function application, and rewriting annotation as cases of the general construction of infix terms. This added syntactical flexibility has been enormously fruitful, as for example in the development of parameterized tactics as noted above.

In primitive Mark2, the bracket construction was still employed strictly to represent constant functions, and the intention was to develop synthetic abstraction algorithms to support the (much stronger) higher order logic which had replaced the first-order logic of `EFTTP`. The only context in which stratification was actually implemented was in the definition facility; but it should be noted that the ability to define stratified functions of arbitrary order all by itself, plus the presence of an ordered pair with left and right types 0, gave the system the logical strength of *NFU* + Infinity or the theory of types of Russell, which is far stronger than first-order logic alone. (The current system is even stronger; the assumption that the set of natural numbers is strongly Cantorian, which is natural to make as soon as strongly Cantorian types are supported, gives additional strength above that of the theory of types, though still considerably weaker than the full standard set theory *ZFC*.)

Synthetic abstraction of a limited kind was developed and used in Mark2. The aim was to preserve the parallelism of structure between abstracts and terms found in *EFT*, and this can be achieved in a certain limited domain.

In the very earliest versions, the abstraction tactic needed a special subtactic for each operator over which abstraction was to be supported. This was improved by the introduction of the operation on operators represented by initial colon, which has the following defining property: for each "flat" operator (i.e., operator with both relative types zero) `+`, we have `(?f :+ ?g) @ ?x` equal to `(?f @ ?x) + (?g @ ?x)`. The colon converts a flat operator into the corresponding operation on functions. A single "built-in tactic" `RAISE` supported all instances of the property of the colon operation. The algorithm which was developed for primitive Mark2 could abstract a term relative to a variable (or other term) `?x` if the operators appearing in the term were all either function application, with no occurrence of `?x` in functions applied, or flat. The abstraction could be obtained from the term by replacing the variable `?x` with the identity function `Id`, each other atomic term outside of functions with its constant function, leaving functions alone, and replacing function application with composition; the parallelism of structure was as good as in *EFT*. However, the full abstraction capabilities of Mark2 (found at that point only in the definition facility) were considerably greater.

One indication of the unsatisfactory character of this abstraction is that it cannot be iterated. In *EFT*, considerations of sort made iteration of abstraction inconceivable, but in Mark2 it could be contemplated – but not actually carried out in an attractive way. The inability to abstract into applied functions (which makes the abstraction "predicative") was not even then reflected by any weakness of the Mark2 logic; impredicative abstractions could be introduced using the definition facility. (The prover does have a state in which abstraction of all kinds is restricted to predicative cases alone; we do not think that this is a practical restriction, though it has some interest because it implements a particular fragment of "New Foundations" which has been the object of some study).

A full synthetic abstraction facility "in principle" was obtained by extending the colon operator to apply to non-flat operators. Some care had to be taken in its definition; the relative types of `:+` for non-flat operators `+` are the same as for `+` itself, and the defining property has to be adjusted for each case of relative types to preserve stratification. We give one example: `(?f :@ ?g) @ ?x = (?f @ [?x]) @ (?g @ ?x)` (note that this permits abstraction into function applications). However, this full synthetic abstraction lost a great deal of its "readability", and lost almost all of it when it was iterated (as now became possible). Moreover, writing the full abstraction algorithm was a considerable tour-de-force, because the prover has limited ability to abstract over operators, and very limited ability to detect the relative types of operators in an automatic manner. This is a disability to be expected; operators, especially non-flat ones, are not first-class citizens of the world of the logic of Mark2.

A further reason (and really the determining reason) that synthetic abstraction was abandoned is that the practical usefulness of rewriting inside "lambda terms" became clearer and clearer as we attempted to implement even the simplest reasoning about quantifiers using synthetic abstraction. Rewriting inside λ-terms implements a weak form of extensionality which is technically difficult to implement in combinatory logic, as is already well-known to researchers in this area.

The functions of the conditional layer were not present in primitive Mark2, either, because our thinking had not yet disentangled them from an essential involvement with abstraction. The way that a conditional which would now be handled with `0 |-| 1` or its kind was handled is that a term `(?a = ?b) || ...?a... , ?c` was first converted to the form `(?a = ?b) || (?F @ ?a) , ?c` using a synthetic abstraction tactic, then converted to `(?a = ?b) || (?F @ ?b) , ?c` by the application of an equational axiom expressing (HYP), then converted to `(?a = ?b) || ...?b... , ?c` by a reduction algorithm. One of the great successes of the tactic language was that this process could be invoked as a tactic to be applied to the whole conditional expression, and it would carry out the indicated substitution without the user ever seeing an abstraction term. We did not become seriously disaffected from synthetic abstraction terms at this stage, but later on when we actually had to read them (as in implementing induction or quantification).

However, this approach to conditional rewriting has serious disadvantages. In order to apply a rewrite justified by the hypothesis of a conditional expression, it is necessary to go up to the level of the whole conditional expression. Moreover, there are serious headaches when it is desired to rewrite just one occurrence of a term which occurs in more than one place in the same case of the relevant conditional expression; this can be circumvented by marking the target to be rewritten in some syntactically distinctive way, but it is rather unpleasant to have to do it. Nonetheless, we succeeded (among other things) in writing a complete tautology checker entirely in the algebraic layer (essentially the only layer in primitive Mark2), using synthetic abstraction and reduction algorithms and axiom (HYP) to simulate the functionalities now handled by the conditional and abstraction layers of the prover.

An effect of the functions of the conditional layer which was not anticipated is that they actually strengthen the prover's logic (though inessentially). The reason for this is that, if one is limited to using abstraction in the procedure above, a hypothesis can only be applied at its own relative type. Thus, all reasoning in primitive Mark2, in the absence of unstratified user-declared axioms which enable type shifting, could be viewed as reasoning in a type theory without identifications of objects at different type levels. But the conditional layer allows substitutions justified by hypotheses to be carried out at relative types other than the type of the hypothesis; thus, unintentionally, the conditional layer was the one which actually forced the built-in logic of Mark2 to be a system like "New Foundations" instead of a merely notationally polymorphic version of type theory (this had always been the intended interpretation, and user adoption of unstratified axioms could exclude a notationally polymorphic interpretation as well). This is an inessential modification, in the sense that the logical power of a notationally polymorphic version of type theory and a genuinely untyped system with stratified abstraction are known to be the same (as long as extensionality is not assumed in the latter). But it is still an interesting observation.

4.5.1 Examples of Synthetic Abstraction

We present material from the proof script developing the predicative abstraction and reduction algorithms originally used in Mark2. Its general similarity to algorithms used in the extended combinatory logic example above should be noted.

```
(* the following  declaration ensures that the infix variable ^& will
match only flat infixes *)

declaretypedinfix 0 0 "^&";

(*
RAISE0:
(?f @ ?x) ^& ?g @ ?x =
(?f :^& ?g) @ ?x
[]
*)

- s "(?f@?x)^&(?g@?x)";
- ri "RAISE"; ex();
- p "RAISE0";

- dpt "ABSTRACT";

(*
ABSTRACT1 @ ?x:
?x =
Id @ ?x
["ID"]
*)

- s "?x";
- rri "ID";ex();
- p "ABSTRACT1@?x";

(*
ABSTRACT2 @ ?x:
?f @ ?a =
COMP <= ?f @ (ABSTRACT @ ?x) => ?a
["COMP"]
*)

- s "?f@?a";
- rri "COMP";
- right(); right(); ri "ABSTRACT@?x";
- prove "ABSTRACT2@?x";

(*
ABSTRACT3 @ ?x:
?a ^& ?b =
RAISE0 => ((ABSTRACT @ ?x) => ?a)
^& (ABSTRACT @ ?x) => ?b
[]
*)

- s "?a^&?b";
- right();
- ri "ABSTRACT@?x";
- up();left();
- ri "ABSTRACT@?x";
- top();
- ri "RAISE0";
- prove "ABSTRACT3@?x";

(*
ABSTRACT4 @ ?x:
?a =
[?a] @ ?x
[]
*)

- s "?a";
- ri "BIND@?x"; ex();
- p "ABSTRACT4@?x";

(* ABSTRACT@term will (attempt to) express a target term as a function
of its parameter "term" *)

(*
ABSTRACT @ ?x:
?a =
(ABSTRACT4 @ ?x) =>> (ABSTRACT3 @ ?x)
=>> (ABSTRACT2 @ ?x) =>> (ABSTRACT1 @ ?x) => ?a
["COMP","ID"]
*)

- s "?a";
- ri "ABSTRACT1@?x";
- ari "ABSTRACT2@?x";
- ari "ABSTRACT3@?x";
- ari "ABSTRACT4@?x";
- p "ABSTRACT@?x";

(* REDUCE will reverse the effect of ABSTRACT; it will "evaluate"
functions built by ABSTRACT *)

(*
REDUCE:
?f @ ?x =
(ABSTRACT4 @ ?x) <<= ((RL @ REDUCE) *> RAISE0)
<<= ((RIGHT @ REDUCE) *> COMP) =>> ID => ?f @ ?x
["COMP","ID"]
*)

- dpt "REDUCE";
- s "?f@?x";
- ri "ID";
- ari "(RIGHT@REDUCE)*>COMP";
- arri "(RL@REDUCE)*>RAISE0";
- arri "ABSTRACT4@?x";
- prove "REDUCE";

(* old approach to hypotheses *)
(* equational forms of tactics given without proof;
the proofs of the tactics involve no actual rewriting *)

PIVOT:
(?a = ?b) || ?T , ?U =
(RIGHT @ LEFT @ EVAL) => HYP => (?a = ?b)
```

```
|| ((BIND @ ?a) => ?T) , ?U
["HYP"]

REVPIVOT:
(?a = ?b) || ?T , ?U =
(RIGHT @ LEFT @ EVAL) => HYP <= (?a = ?b)
|| ((BIND @ ?b) => ?T) , ?U
["HYP"]
```

We now present examples of the use of these tactics.

```
- declareinfix "+"; declareinfix "*"; declareconstant "sin";

- s "2*?x+?x*?x+?y+sin@3*?x";

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

{([2] :* Id :+ Id :* Id :+ [?y] :+ sin @@ [3]
      :* Id)
   @ ?x}
```

It takes a little practice to see it, but the parallelism of structure here is precise. The connective `@@` represents composition of functions. Note that each atom other than `?x` is replaced with its constant function, each occurrence of `?x` with `Id`, each operator (except function application) with its type-raised version, and application with composition.

Here is an example of the old style of use of hypotheses.

```
- s "(?y=?z)||(?x+?y+?z),?w";

{(?y = ?z) || (?x + ?y + ?z) , ?w}
- ri "PIVOT"; ex();

{(?y = ?z) || (?x + ?z + ?z) , ?w}
```

Notice that the substitution has been carried out without any visible fuss.

```
- s "(?y=?z)||(?x+?y+?z),?w";

- ri "PIVOT"; steps();

{PIVOT => (?y = ?z) || (?x + ?y + ?z) , ?w}

(RIGHT @ LEFT @ EVAL) => HYP => (?y = ?z)
|| ((BIND @ ?y) => ?x + ?y + ?z) , ?w

(RIGHT @ LEFT @ EVAL) => HYP => (?y = ?z)
|| ([?x + ?1 + ?z] @ ?y) , ?w
```

```
(RIGHT @ LEFT @ EVAL) => (?y = ?z)
|| ([?x + ?1 + ?z] @ ?z) , ?w

(?y = ?z) || (LEFT @ EVAL)
=> ([?x + ?1 + ?z] @ ?z) , ?w

(?y = ?z) || (EVAL => [?x + ?1 + ?z] @ ?z) , ?w

(?y = ?z) || (?x + ?z + ?z) , ?w
```

The version of the PIVOT tactic given here actually uses the “modern” built-in abstraction and reduction instead of the synthetic algorithms. We leave to the reader’s imagination what the expansion into steps would have looked like with the original abstraction and reduction.

Attempting to undo the last operation gives us a nice opportunity to illustrate the main problem with PIVOT.

```
- left();right();

{?y = ?z} || (?x + ?z + ?z) , ?w
?y = ?z

(?y = {?z}) || (?x + ?z + ?z) , ?w
?z

- rri "ID"; ex();

(?y = {ID <= ?z}) || (?x + ?z + ?z) , ?w
ID <= ?z

(?y = {Id @ ?z}) || (?x + ?z + ?z) , ?w
Id @ ?z

- top();right();left();

{(?y = Id @ ?z) || (?x + ?z + ?z) , ?w}
(?y = Id @ ?z) || {(?x + ?z + ?z) , ?w}

(?x + ?z + ?z) , ?w

(?y = Id @ ?z) || {?x + ?z + ?z} , ?w
?x + ?z + ?z

- right();left();
(?y = Id @ ?z) || (?x + {?z + ?z}) , ?w
?z + ?z
```

```
(?y = Id @ ?z) || (?x + {?z} + ?z) , ?w
?z

- rri "ID"; ex();

(?y = Id @ ?z) || (?x + {ID <= ?z} + ?z) , ?w
ID <= ?z

(?y = Id @ ?z) || (?x + {Id @ ?z} + ?z) , ?w
Id @ ?z

- top();

{(?y = Id @ ?z) || (?x + (Id @ ?z) + ?z) , ?w}

- ri "REVPIVOT"; ex();

{REVPIVOT => (?y = Id @ ?z)
   || (?x + (Id @ ?z) + ?z) , ?w}

{(?y = Id @ ?z) || (?x + ?y + ?z) , ?w}

- left();right();

{?y = Id @ ?z} || (?x + ?y + ?z) , ?w

?y = Id @ ?z

(?y = {Id @ ?z}) || (?x + ?y + ?z) , ?w

Id @ ?z

- ri "ID"; ex();

(?y = {ID => Id @ ?z}) || (?x + ?y + ?z) , ?w

ID => Id @ ?z

(?y = {?z}) || (?x + ?y + ?z) , ?w
```

In this case, where it is only desired to apply the converse of the hypothesis as a rewrite rule in one place, it is necessary to move back and forth between the top of the conditional expression and the place where the change is to be made, applying then removing syntactical markers to control the behavior of `REVPIVOT`. With the full functionality of the conditional layer, both of these would look precisely the same: one would apply first `0|-|1`, then its converse, in both cases at the level of the atomic term being modified. Moreover, the role of abstraction is completely eliminated.

5 Relations to Other Work

Mark2 is not closely similar to any other work in automated reasoning. We have discussed above its similarities to and differences from other work in the area of rewriting. The remoteness of Mark2 from the rewriting community is best expressed in the fact that the Knuth-Bendix algorithm and its refinements are essentially irrelevant to the Mark2 research program, though we think that it would be interesting to write an interface which would apply the Knuth-Bendix algorithm to a Mark2 theory as an automatic tactic generator. Thus, rewriting is used by Mark2 in a quite different way than it is used by a power like Otter ([20])

Genetically, Mark2 is related to the group of proof systems descended from Edinburgh LCF, which have been referred to as "logical frameworks", though in a very narrow sense. We were familiar with Nuprl (though we did not have user experience) when we started work; this inspired us to write the prover in ML. But Mark2 does not manipulate proofs in the sense that Nuprl ([3]) or similar systems do (proof objects could be developed as complex constructions in the tactic language in Mark2, but this would be a very different approach from that found in the "logical frameworks" family). Mark2 is oriented toward terms rather than proofs or even propositions. Mark2 differs further from Nuprl and some other "logical frameworks" in using classical non-constructive logic. The higher-order logic based on *NFU* used by Mark2 is considerably less remote from standard mathematical practice than the complex constructive higher-order logics used by Nuprl or Coq ([6]). (This assertion may not be self-evident; it is the thesis of my paper [12] and my pending book [17]).

The prover to which Mark2 is probably most similar in overall outlook is HOL ([8]), though this is not at all obvious. HOL, though it is related to the LCF provers, has abandoned their commitment to proof objects and reasons directly in a higher order logic based on Church's simple theory of types, which is of about the same level of strength and readily mutually interpretable with the stratified higher order logic of Mark2. Both systems use classical logic. We have borrowed from HOL the idea of using the Axiom of Choice (in the form of assuming a selection operator) to facilitate reasoning with existential quantifiers. However, the superficial appearances of the systems are totally different, and HOL is primarily a manipulator of propositions, where Mark2 is primarily a manipulator of terms.

There are some incidental relationships with other systems which should be noted. We have observed above that the "rewrite logic" proposed by the developers of OBJ (in [19]) could readily be implemented in Mark2; banning the converse rewriting annotation operators (`<=`, `<<=`, and `<*`) would restrict the equational logic of Mark2 to rewriting logic. The theorem export system of

Mark2 implements the same insights as the "little theories" approach to theory modularity of the the developers of IMPS ([7]), though the implementation in IMPS is far more elegant. Though we had developed our system of theorem export already when we encountered the IMPS work, the IMPS developers' writings made it much clearer to us what we had done, and also made it clear what further developments would be necessary. Our theorem export system remains underutilized because it is still too awkward for ready use.

The features of Mark2 which appear to be really novel are its device for representing tactics as equational theorems and (oddly) its facility for easy application of rewrite rules at single locations in terms by "navigation within terms". The use of *NFU* as the higher order logic of a theorem prover is certainly novel, and the logic of case expressions used appears to be new.

6 Implementation Issues

The title of this section involves an equivocation on the meaning of "implementation" which we now explain: there is a section here on issues related to the implementation of the prover as a computer program, and a section on the implementation of mathematical theories using the prover.

6.1 Implementation of the Prover

The prover is implemented in Standard ML. The facilities of this language are naturally adapted to writing this kind of software, and we do not find anything special to say about the implementation, except that we may (accidentally, surely) have achieved the first computer implementation of Quine's definition of stratification. We say "accidentally" because stratification is a special case of the general problem of type inference, which is of course well understood; a much more complex system of type inference is found in the language ML itself, for example. Stratification is of some historical interest because "New Foundations" was one of the first polymorphic system ever defined ([21], 1937), which would make the problem of determining whether a formula is stratified one of the oldest instances of the problem of type inference.

We are interested in issues of resource use by the prover, especially issues of space. Memory management for the current version of Mark2 is entirely handled by Standard ML; in an attempt to try out some ideas for management of data structures representing terms, we have explored the idea of implementing the prover in `C++`. There is an implementation of a bare subset of the prover in `C++`, written by a student, in which theorems can be proved by hand, but which does not support the tactic language. We followed up this work (which incorporated some but not all of the memory optimizations we had in mind) with an implementation of the prover up to term display and navigation which enforces maximal sharing of memory on terms and related data types. This has not yet been upgraded to support the ability to carry out proofs, though matching functions have been written. We find ourselves very impressed with the capabilities of ML after this work. There is yet another implementation of an old version of the prover (supporting the algebraic layer and tactics), which runs on a PC and is written in Caml Light, another dialect of ML.

A project at the University of Idaho (see proposal [1]) has done some preliminary work on a graphic user interface for the prover, which seems like an obvious improvement to pursue. The ability to click a subterm of a displayed term and go there would enormously streamline the navigational functions of the prover (from the user standpoint, at least).

6.2 Implementation of Mathematical Theories

A library of elementary proof scripts is found on our Web page (address given in the introduction), covering first-order logic (both propositional and the logic of quantifiers), some elementary algebra and some elementary set theory. Preliminary investigations of induction proofs in the natural numbers are found there.

In logic, a complete tautology checker has been written as a tactic. There is a set of tools which allows the emulation of tableau proofs, but the most promising approach to doing proofs in logic (on all levels) seems to be the implementation of the equational approach of Gries exemplified in [9] and applied to program verification by Cohen in [2]. There is a file with basic declarations for an implementation of Cohen's approach to program verification on the Web page.

Our strongest evidence for the serviceability of the prover for projected applications in software verification and development from specifications has been the success of students in learning to use it and using it to develop bodies of theory. A undergraduate computer science student at Boise State, Michael Parvin, has developed the results in propositional logic and quantification found in [9]. Parvin's propositional logic work in particular has been found serviceable in the development of further theories. A group of graduate students at the University of Idaho is continuing this work under another grant from the Army Research Office (see the proposal [1]).

We have also posted a file with technical developments in untyped combinatory logic; we are testing the applicability of Mark2 as a research tool for investigations of such concepts as strong reduction in this area of logic.

7 Progress Made Under This Grant

The benefits to this work of the financial assistance of the Army Research Office have been inestimable (and the continuing support of the ARO for theory development by students at the University of Idaho as proposed in [1] is appreciated). The grant proposal was written from the standpoint of the `EFTTP` system. By the time funding began, the transition to the primitive Mark2 system had already been made. We were still committed to a program of using synthetic abstraction in place of variable binding constructions.

The construction of mathematical theories formally verified by machine is an extremely time-consuming task. This is especially true when the platform itself is undergoing development; actual theory development needs to be attempted to see what changes in the platform are needed. In this project, serious problems with reasoning with quantifiers were revealed by our essays in theory development: eliminating these was a multi-step process, requiring first the introduction of explicit variable binding, then the limited higher-order matching, and finally the development of facilities for automatic handling of strongly Cantorian types. This evolution was completed only fairly recently, which has retarded theory development.

Visible improvements of the prover during the period of the grant include the introduction of parameterized tactics and tactic operators, the installation of the entire conditional and abstraction layers (with the underlying prerequisite sorting out of concepts crucial to prover development), and, most recently, the implementation of strongly Cantorian types, with the "side effect" (really its primary motivation) of the fluent implementation of quantification.

We believe that the logic of the prover has reached essentially its final form, though there may be some further minor refinements. The theorem export system, and theory modularity generally, still may see major modifications (or even a fundamental shift in approach). We have some interest in developing proof objects, which would imply refinements of the tactic language, though the success of work with proof scripts has made this seem less urgent.

The `C++` implementation of part of the prover is an experiment which we still may carry to its conclusion, but it is not clear whether these kinds of performance issues are crucial to the usability of the prover, since SML interpreters are now available on more platforms.

This grant has supported the work of the investigator and students in starting the construction of a body of mathematical theory adequate for applications, which continues under another ARO grant at the University of Idaho. This development work has also been the platform for user testing, which suggests that users of a wide range of levels of sophistication can learn to prove theorems with this system.

8 Appendix on Personnel

Personnel who have worked on the ARO grant have been the principal investigator, M. Randall Holmes, and a number of undergraduate research assistants. Karen Agnetta, Larry Campbell, Fongshing Lam, Michael Parvin, and Brian Mayer are or were Boise State undergraduate students who worked on the project. Of these, Michael Parvin made the greatest contribution, developing extensive proof scripts in propositional and predicate logic; Brian Mayer also made a considerable contribution (a `C++` program implementing part of the prover), but was mainly funded from another source (see the next section for details).

9 Appendix on Dissemination of Results

Two papers on the Mark2 work have been published in conference proceedings so far.

These are

1. "Untyped λ-calculus with relative typing", in Dezani and Plotkin, eds., *Typed Lambda-Calculi and Applications*, the proceedings of TLCA '95, Springer, 1995.

2. "Disguising recursively chained rewrite rules as equational theorems", in Hsiang, ed., *Rewriting Techniques and Applications*, the proceedings of RTA '95, Springer, 1995.

We are planning to submit a version of this final report for publication in a suitable journal.

A monograph, *Elementary Set Theory with a Universal Set* ([17]), shortly to be published by the Cahiers series of the Center for Logic in the Department of Philosophy at the Catholic University of Louvain-la-Neuve, Belgium, acknowledges the support of this grant with respect to one chapter, which is related to theoretical issues underlying the Mark2 research. Another publication on theoretical issues which will acknowledge the support of this grant is a survey of systems of first-order logic without bound variables on which I gave a preliminary report at the 1997 Joint Mathematics Meetings in San Diego.

Two other grant proposals were funded as offshoots of this work:

1. An REU (Research Experience for Undergraduates) award through NSF EPSCoR during summer 1995, supporting a BSU computer science undergraduate who implemented an important subset of the prover in C++.

2. (with Jim Alves-Foss of the University of Idaho): "Automated Reasoning using the Mark2 Theorem Prover", Army Research Office proposal no. P-36291-MA-DPS (funded by grant no. DAAH04-96-1-0397, starting date August 1, 1996)

Development of mathematical theories under Mark2 is supported by the latter grant; this work is continuing.

References

[1] Jim Alves-Foss and M. Randall Holmes, "Automated Reasoning using the Mark2 Theorem Prover", Army Research Office proposal no. P-36291-MA-DPS (funded by grant no. DAAH04-96-1-0397, starting date August 1, 1996)

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

[3] R. Constable and others, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Englewood Cliffs, 1986.

[4] H. B. Curry and R. Feys, *Combinatory Logic*, Vol. I, North Holland, Amsterdam, 1958.

[5] N. de Bruijn, "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.

[6] G. Dowek *at al.*, The Coq Proof Assistant User's Guide Version 5.6. Rapport Technique 134, INRIA, December 1991.

[7] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer, "IMPS: an interactive mathematical proof system", *Journal of Automated Reasoning*, vol. 11 (1993), pp. 213-48.

[8] M. J. C. Gordon and T. F. Melham, *Introduction to HOL: a theorem proving environment for higher order logic*, Cambridge University Press, 1993.

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

[10] M. Randall Holmes, "Systems of Combinatory Logic Related to Quine's 'New Foundations'", Ph.D. Dissertation, State University of New York at Binghamton, 1990.

[11] M. Randall Holmes, "Systems of Combinatory Logic Related to Quine's 'New Foundations'", *Annals of Pure and Applied Logic*, 53 (1991), pp. 103-33.

[12] Holmes, M. R. "The set-theoretical program of Quine succeeded, but nobody noticed". *Modern Logic*, vol. 4, no. 1 (1994), pp. 1-47.

[13] M. Randall Holmes, "`EFTTP`: an interactive equational theorem prover and programming language", Army Research Office proposal no. P-33580-MA-DPS (funded by grant no. DAAH04-93-G-0247).

[14] M. Randall Holmes, "Disguising recursively chained rewrite rules as equational theorems, as implemented in the prover `EFTTP` Mark 2", in *Rewriting Techniques and Applications* (proceedings of RTA '95), Springer, 1995, pp. 432-7.

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

[16] M. Randall Holmes, "A Functional Formulation of First-Order Logic 'With Infinity' Without Bound Variables", preprint, available from the author.

[17] M. Randall Holmes, *Elementary Set Theory with a Universal Set*, volume 10 of the Cahiers du Centre de logique, Academia, Louvain-la-Neuve (Belgium), 241 pages, to appear, ISBN 2-87209-488-1.

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

[19] Narciso Martí-Oliet and José Meseguer, "Rewriting Logic as a Logical and Semantic Framework", technical report SRI-CSL-93-05, SRI International, 1993.

[20] Larry Wos, et. al., *Automated Reasoning: introduction and applications*, 2nd ed., McGraw-Hill, 1992.

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