

# Discovering Theorems using GOEDEL: A Case Study

Johan Gijsbertus Frederik Belinfante

Georgia Institute of Technology, Atlanta, GA 30332-0160 (U.S.A.)  
belinfan@math.gatech.edu

**Abstract.** Combining an interactive symbolic manipulation program with a theorem prover allows one to discover theorems as well to prove them. The specific focus in this paper is on illustrating how to use the GOEDEL program, a Mathematica<sup>TM</sup> implementation of Gödel's algorithm for class formation, to help discover theorems about sets satisfying some property hereditarily. Similar techniques are applicable to other topics in set theory. Formal proofs of many of these theorems have been obtained using McCune's first order automated reasoning program **Otter**.

## 1 Introduction

In this paper we focus on illustrating how a symbolic manipulation program can help discover theorems in set theory. This is part of ongoing work (Belinfante, 1999a, 1999b and 2000b) on proving theorems of NBG class theory, building on prior work by Robert Boyer et al. (1986) and Art Quaipe (1992a, 1992b), using McCune's (1994) first order logic automated reasoning program **Otter**.

Our aim is not to announce stunning new theorems, but rather to explain the fairly prosaic techniques that were found to be useful. Further details, including related theorems, proof summaries and other pertinent information about each of the **Otter** proofs of the theorems mentioned here, as well as for several thousand other theorems in set theory, may be found on the author's website: <http://www.math.gatech.edu/~belinfan/research/>. A recent version of the GOEDEL program is also provided there.

The theorems to be discussed here are about sets satisfying some property hereditarily. For example, a set is said to be *hereditarily finite* if it is not only finite, but all its members are finite, and all their members in turn, and so on. Another example of interest concerns a characterization of the class **OMEGA** of all ordinal numbers. It is well-known (Monk 1969) that when the axiom of regularity holds, the class of ordinal numbers can be described as the class of hereditarily full sets. It was discovered that something similar (Theorem H-ON-3) holds without assuming the axiom of regularity.

## 2 Brief Description of the GOEDEL program

The GOEDEL program, originally developed (Belinfante, 1996 and 2000a) to help prepare input files for proofs in set theory using McCune's automated reasoning

program `Otter`, implements in Mathematica™ an algorithm Kurt Gödel (1940) presented in his proof of his basic class existence metatheorem schema.

Although the program was named after Gödel's finite axiomatization for class theory, the main ideas are older and stem from Bernays's reformulation (1937) of von Neumann's class theory. By syntactically analyzing the structure of a statement  $p(x)$ , the usual class formation constructor  $\{x \mid p(x)\}$  can be eliminated in favor of building classes in terms of a finite number of primitives, for example, the universal class  $V$ , the membership relation  $E$ , and seven other basic class constructors: `complement`, `domain`, `flip`, `rotate`, `pairset`, `cart`, `intersection`. Bernays (1991, page 64) points out that even fewer could be used.

Since the `GOEDEL` program itself has already been described elsewhere (Belinfante, 1996 and 2000a), only a brief indication of its general features will be given here. Some familiarity with Mathematica is assumed on the part of the reader. At the heart of the program is the algorithm for converting the customary definitions of classes into expressions built up from the primitive constructors. This algorithm is presented in the `GOEDEL` program as a series of definitions for a Mathematica function `class[x,p]`. The first argument  $x$ , which is assumed to be a set, must be either an atomic symbol, or an expression of the form `pair[u,v]` where  $u$  and  $v$  in turn are either atomic symbols or pairs, and so on. The second argument  $p$  is a statement which may involve the variables that appear in the expression  $x$ , as well as other variables that may represent arbitrary classes (not just sets). The statement may contain quantifiers, but all quantified variables must be sets. The quantifiers `forall` and `exists` used in the `GOEDEL` program are explicitly restricted to set variables.

Many rewrite rules must be added to produce compact definitions. With these added rewrite rules, Gödel's proof of termination no longer applies, but the `GOEDEL` program nonetheless has proved to be a practical ad hoc tool for formulating definitions and for simplifying statements of theorems.

Although it contains no systematic mechanism for carrying out deductions, the `GOEDEL` program does sometimes manage to prove statements by simplifying them to `True`. In addition, some rudimentary tools have been developed to coax the `GOEDEL` program to reveal new facts, as will be explained shortly.

### 3 A simple example and some history

From a user's standpoint, a most useful feature of the `GOEDEL` program is its ability to recognize that various different specifications of a class are equivalent. This use of `class` is illustrated by the following different descriptions of the class `FULL` of full sets.

```
class[x,forall[y,z,implies[and[member[y,z],member[z,x]],
    member[y,x]]]] = FULL
class[x,forall[y,implies[member[y,x],subclass[y,x]]]] = FULL
class[x,subclass[U[x],x]] = FULL
```

```
class[x,subclass[x,P[x]]] = FULL
class[x,full[x]] = FULL
```

Here and elsewhere in this paper, for the sake of brevity, an equation  $a = b$  is written to indicate that Mathematica input  $a$  produces output  $b$  for some version of the GOEDEL program. As facts discovered with one version are added as rewrite rules in later ones, the GOEDEL program is continually evolving.

Consider for example the following two examples involving `class`:

```
class[x,exists[y,and[member[x,y],full[y]]]] = U[FULL]
```

and

```
class[x,exists[y,and[subclass[x,y],full[y]]]]
      = image[inverse[S],FULL]
```

The current version of the GOEDEL program simplifies both of the above two expressions to the universal class  $V$ . One would have to go back to a version of the GOEDEL program as it was on 1999 February 12 to get the results stated above, because on the following day the rewrite rule

$$\text{image}[\text{inverse}[S],\text{FULL}] := U[\text{FULL}]$$

was added. An `Otter` proof of this equation had just been found on that day; this is Theorem FUL-IMS2 in the FULL\3 group. The proof is not complicated; it just uses the facts that if a set  $x$  is full, then so are the sum set  $U[x]$  and the power set  $P[x]$ . Although the equation  $U[\text{FULL}] = V$  had long ago been proved by hand, it was not added as a rewrite rule to the GOEDEL program until 2000 October 16, when an `Otter` proof of this equation was obtained. This proof found by `Otter` uses transfinite induction and a recursive definition of transitive closure.

The rewrite rules in the GOEDEL program can simplify statements as well as descriptions of classes, and in particular, can be used to eliminate quantifiers. Given any statement  $p$ , one can form the class `class[w,p]` where  $w$  is any variable that does not occur in the statement  $p$ . This class is the universal class  $V$  if  $p$  is true, and is the empty class when  $p$  is false. The Mathematica definition

$$\text{assert}[p_] := \text{Module}[\{w=\text{Unique}[]\}, \text{equal}[V, \text{class}[w,p]]]$$

thus produces a new statement equivalent to the original one. The occurrence of `class` here causes Gödel's algorithm to be invoked, the meaning of the statement  $p$  to be interpreted, and the rewrite rules in the GOEDEL program to be applied. The transformed statement need not be simpler than the statement one started with, but often it is. To improve readability of the output, further rewrite rules are sometimes used to convert the equations obtained with `assert` back to simpler nonequational statements.

## 4 Membership Rules

When one wants to extend the `GOEDEL` program to deal with a new concept, the first step is to add an appropriate membership rule. For example, for the power class constructor `P[x]`, the `GOEDEL` program contains the membership rule

```
member[x_,P[y_]] := and[member[x,V],subclass[x,y]]
```

Another example is the definition of the class `FULL` of all full sets:

```
member[x_,FULL] := and[member[x,V],subclass[U[x],x]]
```

Whenever a membership rule involves quantifiers, it is the author's practice to wrap it with `class` to force those quantifiers to be eliminated. A typical example is the following wrapped membership rule which is used to define the sum class constructor `U[x]`:

```
class[w_,member[x_,U[z_]]] := Module[{y = Unique[]},  
  class[w,exists[y,and[member[x,y],member[y,z]]]]
```

A second example is the definition of the subset relation `S`.

```
class[w_,member[x_,S]] :=  
  Module[{u = Unique[],v = Unique[]},class[w,exists[u,v,  
    and[equal[pair[u,v],x],subclass[u,v]]]]
```

In this case one needs variables in order to rewrite `x` as an ordered pair. It should be pointed out that in the `GOEDEL` program it is explicitly assumed that all quantified variables must refer to sets, and not to proper classes. For this reason it is not necessary to add `member[u,V]` and `member[v,V]` on the right side of this membership rule. For the benefit of Mathematica experts we remark that the wrapped membership rules must nonetheless be given as downvalues for `class`, and not as upvalues for `member`; attempting to do the latter would cause looping to occur.

In the case of the subclass relation `S` the `GOEDEL` program also contains another unwrapped membership rule

```
member[pair[u_,v_],S] :=  
  and[member[u,V],member[v,V],subclass[u,v]]
```

which applies only to ordered pairs. In this rule, it would have been safe to omit `member[u,V]`, but it would not be correct to omit `member[v,V]`. To rule out the possibility that adding such a second membership rule might lead to an incompatible definition for a particular class, it is wise not to add such a rule until it has been formally proven to be correct, either by using `Otter` or by using the `assert` mechanism of the `GOEDEL` program itself to discover the second rule.

## 5 Normalization Rules

Any class  $x$  is the class of all its members, a fact which is expressed in the `GOEDEL` program as a default rule:

```
class[u_,member[u_,x_]] := x /; And[FreeQ[x,u],AtomQ[u]]
```

This default rule must be placed after all specific wrapped membership rules in order for the specific rules to have a chance of being applied. This feature has one annoying side effect of which the user should be aware: one cannot just add wrapped membership rules on the fly in a Mathematica session, because Mathematica's built-in precedence rules for evaluation will cause the default rule to be applied instead of the rule that one has just added.

If the class  $x$  is replaced with some specific class, then it is as likely as not that `class[u_,member[u_,x_]]` will not simplify to  $x$ , a fact that one can work to one's advantage. If this expression does simplify to the same  $x$  with which one started, we say that the class  $x$  is *normalized*. Whether an expression is normalized or not depends of course on what simplification rules are present. Whatever this expression does simply to is in any case equal to the original expression  $x$ , thereby sometimes enabling one to discover an alternative formula for a particular class of interest.

To expedite this type of discovery, a separate file `TESTS.M` containing various useful Mathematica definitions such as

```
Normalize[x_] := Module[{w=Unique[]},member[w,V]=True;  
                        class[w,member[w,x]]]
```

```
Normality[x_] := (x == Normalize[x])
```

may be loaded along with the `GOEDEL` program itself. This file also contains variants of `Normality` in which one or two `assert`'s have been wrapped around `member[w,x]`. These variant tests are called `Renormality` and `Rerenormality`, respectively. The beauty of these tests is that one can specify exactly for which class a simpler formula is to be sought.

For binary and ternary relations, variants have been added which can exploit the membership rules that involve ordered pairs or ordered triples. The simplest of these is:

```
RelnNormality[x_] := Module[{u=Unique[], v=Unique[]},  
                            member[u,V]=True; member[v,V]=True;  
                            Equal[composite[Id,x],  
                            class[pair[u,v],member[pair[u,v],x]]]]
```

Again there are also variants with `assert` wrappers. Experience indicates that what works even better for many binary and ternary relations are still further variants which rely on vertical section rules instead of membership rules. The bare bones rule is called `VSNormality`:

```

VSNormality[x_] := Module[{u=Unique[], v=Unique[]},
  member[u,V]=True; member[v,V]=True;
  Equal[composite[Id,x], class[pair[u,v],
  member[v,image[x,singleton[u]]]]]]

```

Typically one barrages a relation of interest with a whole battery of such tests to flush out any interesting formulas that might prove useful.

When one does succeed in discovering some useful formula, one may add the new rule as a permanent new simplification rule, thereby enhancing the power of the GOEDEL program. Adding such rules has the beneficial effect of forcing simple expressions to simplify to themselves. One can of course take the first expression that comes along to enforce normalization, but doing so could lead to rather complicated rules of little intrinsic interest. In a few cases where one may have to resort to the expedient of adding rather complicated normalization rules, one may still hope that later on some simpler rules would be discovered. Our experience indicates that adding a complex normalization rule is better than adding no rule at all, and one often learns later how to break up complex rules into smaller pieces even after having temporarily put some complicated rule in place.

Since the GOEDEL program is to be used for discovery rather than proof, we do not hesitate to add facts proved using Otter. To avoid subtle errors, however, we avoid adding rules for which only a proof by hand is available because humans tend to gloss over uninteresting details, such as the requirement that some set be nonempty, or that some relation be thin. To avoid circularity, theorems discovered using with the GOEDEL program are never added to the usable list in Otter. Such facts are regarded as conjectures; of course, even mechanized proofs do not preclude the possibility of error (Belinfante 1997).

Finding formal proofs with Otter is generally more challenging than using GOEDEL to discover new facts to be proved. For this reason, the Otter proofs tend to lag behind what has been discovered using the GOEDEL program. Nonetheless, the GOEDEL program is no substitute for Otter because it does not produce readable proofs. One can of course try to find out how GOEDEL did its work by using Mathematica's built-in Trace procedure. This sometimes works, but often as not this yields a voluminous and rather unintelligible accounting of what took place.

## 6 A lambda calculus

A principal obstacle in using a first order theorem prover that relies on the clause language is that the Skolem functions that are introduced in the process of converting statements to clause form often lead to expressions of high weight. To avoid frequent intervention by hand to cope with this problem, it is often desirable to eliminate quantifiers over set variables. Minimizing the number of Skolem functions that need to be introduced often allows one to reduce the number of clauses needed, as well as the number of literals in a given clause, thereby greatly improving the readability of the statements of theorems and of

their proofs. An important tool for accomplishing this is to introduce whenever possible bonafide set-theoretic functions corresponding to the function symbols of first order logic. Thus, for example, in addition to introducing the function symbol `tc[x]` for the transitive closure of a class `x`, it is useful to introduce also the related set-theoretic function `TC` whose members are the ordered pairs `pair[x,tc[x]]`, where `x` is any set.

The basic constructor `VERTSECT` provides a standard way to obtain definitions for many functions. This enables one to define functions by specifying the result obtained when they are applied to an input. The basic idea is not limited to functions; any relation can be specified by giving a formula for its vertical sections. The vertical sections of a relation `z` are the family of classes

$$\text{class}[y, \text{member}[\text{pair}[x, y], z]] = \text{image}[z, \text{singleton}[x]].$$

Once one comes to realize the usefulness of working with vertical sections instead of points, it becomes natural to introduce the function which assigns these vertical sections:

$$\text{class}[\text{pair}[x, y], \text{equal}[y, \text{image}[z, \text{singleton}[x]]]] = \text{VERTSECT}[z]$$

Gödel's algorithm first converts the left side of this formula to the expression

$$\text{composite}[\text{Id}, \text{intersection}[\text{complement}[\text{composite}[E, \text{complement}[z]]], \text{complement}[\text{composite}[\text{complement}[E], z]]]].$$

What happens after this depends on `z`. If the class `z` is left unspecified, various normalization rules will simply convert this expression to `VERTSECT[z]`. If the class `z` is known to be a function with domain `V`, a different set of simplification rules will be applied, and one will obtain the function `composite[SINGLETON, z]`. If `z` is the inverse of the membership relation `E`, one obtains `Id`, and so on.

For many relations `z` the vertical sections need not be sets. The domain of `VERTSECT[z]` in general is the class of all sets `x` for which the vertical section `image[z, singleton[x]]` is also a set. Let us call a relation *thin* when all its vertical sections are sets. The axiom of replacement implies that functions are thin, and the sum class and power class axioms imply that `inverse[E]` and `inverse[S]` are thin, where `E` and `S` are the membership and the subset relations, respectively.

One can use `VERTSECT` to find a formula (Belinfante, 2000a) for any function from a formula for its application `A[image[f, singleton[x]]]`. This is done neatly in the `GOEDEL` program by introducing the Mathematica definition

$$\begin{aligned} \text{lambda}[x_, e_] &:= \\ \text{Module}\{\{y=\text{Unique}[\ ]\}, \text{VERTSECT}[\text{class}[\text{pair}[x, y], \text{member}[y, e]]]\} \end{aligned}$$

In addition to `VERTSECT`, it is convenient to introduce a related constructor `IMAGE`, defined by

$$\begin{aligned} \text{lambda}[u, \text{image}[x, u]] &= \\ \text{VERTSECT}[\text{composite}[x, \text{inverse}[E]]] &= \text{IMAGE}[x]. \end{aligned}$$

The constructor `IMAGE` does not in general preserve composites, but this does hold when the right hand factor is thin. While `IMAGE` preserves the global identity function, in general `IMAGE[id[x]]` is not an identity function, but it is nonetheless a useful function. From the `GOEDEL` program one learns:

$$\text{lambda}[w, \text{intersection}[x, w]] = \text{IMAGE}[\text{id}[x]].$$

## 7 Constructions for $H[x]$ and $tc[x]$

To illustrate how the `GOEDEL` program is used, two constructors that recently have been the subject of investigation will be discussed. The classes  $H[x]$  and  $tc[x]$  are respectively the largest full subclass of  $x$  and the smallest full class which contains  $x$ . Both of these descriptions do eventually emerge as theorems, but one must start with somewhat less transparent definitions. The problem here is that until some construction is given, it is not a-priori clear that a largest full subclass of  $x$  exists, nor that there is any smallest full class which contains  $x$ . In the `Otter` work, one begins in each case with a formula for a recursive construction of the class.

Intuitively, the transitive closure  $tc[x]$  of a class  $x$  is the union of  $x$ ,  $U[x]$ ,  $U[U[x]]$ , and so on. When  $x$  is a set, one might entertain the idea of using recursion to define these iterated sum classes, and then defining  $tc[x]$  as the sum class of the class whose members are  $x$ ,  $U[x]$ ,  $U[U[x]]$ , etc. This naive approach will not work when  $x$  is a proper class because in that case all of its iterated sum classes are also proper classes, and thus cannot be members of any class. For this reason, it is better to use recursion to construct a relation whose vertical slices are these iterated sum classes, and then to define  $tc[x]$  as the range of this relation. (For technical reasons, a slight variant of this was used in which the vertical slices are the partial sums  $x$ ,  $\text{union}[x, U[x]]$ , etc., but the basic idea is still the same.) The solution of the recursion equation for this relation is obtained as the union of partial solutions by analogy with standard proofs of the recursion theorem.

In this paper the focus is on what has been done since the author's work on transitive closure, so we concentrate mainly on  $H[x]$ . Consequently little will be said here about how various rules about  $tc[x]$  were discovered using the `GOEDEL` program, and just discuss how those rules were then used to find out facts about  $H[x]$ .

One can construct  $H[x]$  as the union of all full subsets of the class  $x$ . This definition, which is used in the `Otter` work makes no explicit mention of transitive closure:

$$U[\text{intersection}[\text{FULL}, P[x]]] = H[x]$$

Nevertheless, one still needs to use  $tc[x]$  to prove that this construction works, and so recursion still enters indirectly. For this reason, the  $H$  group of theorems about  $H[x]$  must be placed after the  $TC$  groups of theorems about transitive closure.

The theorems about  $H[x]$  listed below were all proved using McCune's automated reasoning program *Otter*. The theorems are formulated in the clause language, and in particular, *Otter*'s notation  $\neg$  for negation and  $\mid$  for disjunction is used here. Theorems flagged with an asterisk are (usually) added to the demodulator list. Complete details of these proofs are posted on the author's website; the proof summaries can be found there in the  $H$  group.

Theorems H-1 through H-6 were the first ones to be proved, first by hand, and then using *Otter*. Theorems H-1, H-2 and H-6, taken together, assert that  $H(x)$  is the largest full subclass of  $x$ . For the most part, the proofs *Otter* found resembled the hand-produced proofs, except for the order; the author had proved Theorem H-6 first, deducing H-4 and H-5 as corollaries, whereas *Otter* first proved H-4 and used it to prove H-6. The listing below includes some corollaries that were added later.

Under the additional hypothesis that  $x$  is a set, Theorem H-4 follows immediately from Theorem FUL-SC-9, one of the theorems about full sets that had been proved in the course of work (Belinfante 1999b) on ordinal number theory. Removing this additional hypothesis requires using facts about transitive closure. *Otter* found a proof of length 15 for Theorem H-4 on level 8.

```
% U:\H.USE                2001/03/25
list(usable).
% Definition
equal(U(intersection(FULL,P(x))),H(x)).           %*DEF-H

% Some examples
equal(intersection(FULL,P(FULL)),H(FULL)).        %*H-FULL-1
equal(U(H(FULL)),H(FULL)).                       %*H-FULL-2
equal(H(OMEGA),OMEGA).                           %*H-ON-1

% Basic theorems about H(x)
full(H(x)).                                       % H-1
equal(tc(H(x)),H(x)).                            %*H-1-TC
subclass(H(x),x).                                % H-2

% Characterization of the class OMEGA of all ordinals
equal(intersection(REGULAR,H(FULL)),OMEGA).      %*H-ON-3

% Connections between tc(x) and H(x)
-subclass(x,H(y)) | subclass(tc(x),y).          % H-SU-TC1
equal(image(inverse(TC),P(x)),P(H(x))).         %*H-TC-2
-member(x,H(y)) | subclass(tc(x),y).           % H-MEM-1

% More basic theorems
-subclass(x,y) | subclass(H(x),H(y)).           % H-3
-full(x) | equal(H(x),x).                       % H-4
```

```

% corollaries of Theorem H-4
equal(H(REGULAR),REGULAR).                %*H-4-REG
equal(H(tc(x)),tc(x)).                    %*H-4-TC

% idempotent property
equal(H(H(x)),H(x)).                      %*H-5

% the largest full subclass of x is H(x)
-full(x) | -subclass(x,y) | subclass(x,H(y)).    % H-6

```

Theorem H-TC-2 above was one of the discoveries made with the GOEDEL program coming on the heels of related discoveries involving the thin relation `inverse[TC]`. For example, a few days earlier, a number of membership rules for inverse images had been discovered using `assert`, among which was the following equation, which was promptly added as a new rewrite rule

```
assert[member[x,image[inverse[TC],y]]] = member[tc[x],y].
```

In the session which led to the discovery of H-TC-2, at an early point the `VSNormality` test had been applied to the function

```
composite[BIGCUP,IMAGE[id[FULL]],POWER] = HC
```

which takes any set `x` to `H[x]`, yielding a messy expression, whose exact nature need not concern us here.

```
In[]: composite[BIGCUP,IMAGE[id[FULL]],POWER] // VSNormality
```

```
Out[] = composite[BIGCUP,IMAGE[id[FULL]],POWER] == mess
```

The output equation was turned around and made into a temporary rewrite rule in order to cause the messy expression to be eliminated, should it ever come up again.

Later in that same session, the `VSNormality` test was applied again, this time to the relation `composite[inverse[S],HC]`. The actual manner in which the formula H-TC-2 popped up can be summarized as follows:

```
In[]: Map[U[image[# ,P[P[x]]]&,
           composite[inverse[S],BIGCUP,IMAGE[id[FULL]],POWER]
           // VSNormality]
```

```
Out[] = P[H[x]] == image[inverse[TC],P[x]]
```

Before the discovery of Theorem H-TC-2, two other connections between `H(x)` and `image(inverse(TC),P(x))` had been proved using `Otter`. Theorem H-TC-1 is an alternative description of `H(x)` using the function `TC`, and Theorem H-TC-3 solves exercise 9.5 on page 74 in a book by Thomas Jech (1978).

```
equal(U(image(inverse(TC),P(x))),H(x)).      % H-TC-1
equal(intersection(x,image(inverse(TC),P(x))),H(x)).    % H-TC-3
```

When Theorem H-TC-2 was added, it became a new demodulator, causing each of these other theorems first to be transformed, and then to be subsumed by  $\text{equal}(U(P(x)), x)$  and by Theorem H-PC-I, respectively.

```
% a demodulator
equal(intersection(x,P(H(x))),H(x)).           %*H-PC-I
```

If one assumes the axiom of regularity, the converse of Theorem H-PC-I holds:

```
-AxReg | -equal(intersection(x,P(y)),y) |
      equal(H(x),y).                             % RE-H
```

Consequently, assuming the axiom of regularity, one can characterize  $y = H(x)$  as the only solution of the equation

$$y = \text{intersection}(x, P(y)).$$

See for example Jech (1978).

In the course of proving these basic theorems about  $H(x)$ , several additional results were discovered by hand, such as Theorems H-I and H-PC listed below. Otter found a different proof for Theorem H-PC. Unlike the author's own proof, this proof used facts about transitive closure.

```
% converse of H-SU-TC1
-subclass(tc(x),y) | subclass(x,H(y)).          % H-SU-TC2
```

```
% the constructor H preserves intersection and power class
equal(intersection(H(x),H(y)),H(intersection(x,y))). %*H-I
equal(H(P(x)),P(H(x))).                             %*H-PC
```

```
% a generalization of epsilon induction
-subclass(intersection(x,P(y)),y) |
      subclass(intersection(REGULAR,H(x)),y).      % H-REG-SU
```

```
% converse of H-MEM-1
-member(x,y) | -subclass(tc(x),y) | member(x,H(y)). % H-MEM-2
end_of_list.
```

## 8 Membership rules for $H[x]$ and $tc[x]$

For the GOEDEL program, the constructions of  $H[x]$  and  $tc[x]$  are not used directly, but one does need to add appropriate membership rules.

Theorems H-MEM-1 and H-MEM-2 characterize  $H(x)$  as the class of all elements  $y$  of  $x$  for which  $tc(y)$  is contained in  $x$ . This characterization of  $H(x)$  is the basis for the membership rule used to define  $H[x]$  in the GOEDEL program.

To deal with the new functor  $H[x]$ , the following membership rule was added:

```
(* added 2001 March 15 *)
member[x_,H[y_]] := and[member[x,y],subclass[tc[x],y]]
```

This defines  $H[x]$  in terms of transitive closure  $tc[x]$ .

The membership rule used for  $tc[x]$  has a rather long history. In the author's *Otter* work he began with a recursive definition for transitive closure in order to prove the basic theorem that the transitive closure of a set is a set, which is obtained as a corollary of the equation  $U[FULL] = V$ . After proving those theorems using *Otter*, he decided as a shortcut to simply add the statement  $U[FULL] = V$  to the *GOEDEL* program. But one still wants the membership rule for  $tc[x]$  to remain valid whether  $x$  is a set or a proper class. For sets, one can characterize the transitive closure as the intersection of all full subsets that contain  $x$ . For a proper class, the transitive closure is the union of all transitive closures of its subsets. The existential quantifier involved in such a statement can be eliminated by applying *assert*, and one then arrives at the following quantifier-free membership rule for  $tc[x]$  currently used in the *GOEDEL* program:

```
(* added 2000 October 26 *)
member[z_,tc[x_]] :=
    and[member[z,V],not[subclass[P[x],image[inverse[S],
        intersection[FULL,P[complement[singleton[z]]]]]]]]
```

## 9 An application: Theorems about hereditarily finite sets

The HF group contains theorems about hereditarily finite sets. The main goal was to show that  $H[FINITE]$  is a model for set theory minus the axiom of infinity. For this one mainly needs to establish that  $H[FINITE]$  is preserved by various basic set theoretic constructions that occur in the Gödel axioms. (See Theorem 31 on page 97 in Jech 1978)

Much of this work has been done. It is perhaps worth commenting that several of the proofs depend explicitly on Quaipe's modification of Kuratowski's construction of an ordered pair. A typical example is the proof of the theorem HF-OP,

```
subclass(cart(H(FINITE),H(FINITE)),H(FINITE)).
```

This says that ordered pairs of hereditarily finite sets are hereditarily finite.

```
% HF.USE                                2001/03/27
list(usable).
% all full finite sets are hereditarily finite
subclass(intersection(FULL,FINITE),H(FINITE)).           % HF-FUL-1

% natural numbers are hereditarily finite
subclass(omega,H(FINITE)).                               % HF-OM

% closure under pairset and union
```

```

-member(pair(x,y),cart(H(FINITE),H(FINITE))) |
    member(pairset(x,y),H(FINITE)).           % HF-UP
-member(pair(x,y),cart(H(FINITE),H(FINITE))) |
    member(union(x,y),H(FINITE)).             % HF-U

% subsets of hereditarily finite sets are hereditarily finite
-member(x,H(FINITE)) | -subclass(y,x) |
    member(y,H(FINITE)).                       % HF-SU
% restatement of HF-SU as an equation without variables
equal(image(inverse(S),H(FINITE)),H(FINITE)). %*HF-HER

% closure under power set, cartesian product and singleton
-member(x,H(FINITE)) | member(P(x),H(FINITE)). % HF-PC
-member(pair(x,y),cart(H(FINITE),H(FINITE))) |
    member(cart(x,y),H(FINITE)).               % HF-CP
-member(x,H(FINITE)) | member(singleton(x),H(FINITE)). % HF-SS-1
% closure under ordered pairs
subclass(cart(H(FINITE),H(FINITE)),H(FINITE)). % HF-OP
% restatement of HF-PC without variables
subclass(image(POWER,H(FINITE)),H(FINITE)).    % HF-POW-1
% a demodulator
equal(U(H(FINITE)),H(FINITE)).                 %*HF-FUL-2

% closure under sum class, domain, range amd inverse
-member(x,H(FINITE)) | member(U(x),H(FINITE)). % HF-SC
-member(x,H(FINITE)) | member(D(x),H(FINITE)). % HF-DO
-member(x,H(FINITE)) | member(R(x),H(FINITE)). % HF-RA
-member(x,H(FINITE)) | member(inverse(x),H(FINITE)). % HF-IN

% converse of HF-SS-1
-member(x,V) | -member(singleton(x),H(FINITE)) |
    member(x,H(FINITE)).                       % HF-SS-2

```

One should always be on the lookout for reformulations of theorems as equations which can be made into demodulators because this helps combat combinatorial explosion. Eliminating variables also helps in this regard because variables can be instantiated in many ways. Two such rules were found here:

```

% restatements of HF-SC and HF-PC as demodulators
equal(image(BIGCUP,H(FINITE)),H(FINITE)).     %*HF-BC
equal(image(inverse(POWER),H(FINITE)),H(FINITE)). %*HF-POW-2
end_of_list.

```

Summaries of the proofs of all these theorems about hereditarily finite sets proved using Otter are posted in the HF group on the author's website.

## 10 Outlook and Conclusions

Mathematicians like to downplay the role of axiomatic set theory, but the plain fact is that there are few areas in modern mathematics where sets can be avoided, and to do so would artificially restrict mathematical practice. For automated reasoning programs to find favor in mathematical research, it is desirable that they can cope with set theory on a routine basis. Fortunately, much progress has been made toward mechanizing set theory, especially by Larry Paulson and his coworkers (1993, 1996), the Mizar group (1999), Formisano and Omodeo (1998), Megill (1997), Farmer (2001) and others, using diverse methods and axioms. The specific techniques discussed in the present paper are to be sure limited to the NBG formulation of set theory. The elimination of variables using `assert`, for example, would not have been possible without using proper classes. Because the NBG formalism is a conservative extension of ZF set theory, one may nonetheless view this formalism in principle (Kunen 1980) as just a particularly convenient notation.

Automated reasoning involves more than merely finding computer proofs of known theorems. In addition to proving theorems, other activities involved in mathematical reasoning also need to be automated, including the formulation of concise definitions, the discovery of useful theorems, the decision as to which equations should be made into rewrite rules and the organization of large collections of theorems. The `GOEDEL` program is an example of a tool that is proving to be an indispensable companion to `Otter` for proofs in set theory.

We have mentioned just a few of the more useful techniques currently being exploited to discover new theorems using the `GOEDEL` program. Space does not permit discussing other useful techniques, for example, tools based on controlling the order in which rewrite rules are to be applied. By comparing different orders of evaluation, new equations can be discovered, in the spirit of the familiar Knuth-Bendix completion procedure. As yet no serious attempt has been made to unify `GOEDEL` with `Otter` or another reasoning program to produce a unified proof environment, but this would certainly be worth doing.

While promising, our present abilities to prove mathematically interesting theorems in set theory using automated proof techniques still fall far short of what a skillful mathematician can do by hand. But if the enterprise succeeds, our collective efforts will have accomplished something extraordinarily useful, namely the construction of tools for mechanizing deduction in an area central to all of mathematics. The repercussions would be nothing short of revolutionary.

## References

- Belinfante, J. G. F., On a Modification of Gödel's Algorithm for Class Formation, Association for Automated Reasoning News Letter, No. 34 (1996) pp. 10–15.
- Belinfante, J. G. F., On Quaipe's Development of Class Theory, Association for Automated Reasoning Newsletter, No. 37 (1997) pp. 5–9.
- Belinfante, J. G. F., Computer Proofs in Gödel's Class Theory with Equational Definitions for Composite and Cross, Journal of Automated Reasoning, vol. 22 (1999) pp. 311–339.

- Belinfante, J. G. F., On Computer-Assisted Proofs in Ordinal Number Theory, *Journal of Automated Reasoning*, vol. 22 (1999) pp. 341–378.
- Belinfante, J. G. F., Gödel’s Algorithm for Class Formation, in *Automated Deduction – CADE-17*, edited by D. McAllester, June 2000, Lecture Notes in Artificial Intelligence, vol. 1831, pp. 132–147, Springer-Verlag, Berlin. (ISBN 3-540-67664-3)
- Belinfante, J. G. F., The Unifying Concept of Subvariance, in *FTP 2000, Third International Workshop on First-Order Theorem Proving*, St. Andrews, Scotland, July 2000, P. Baumgartner and H. Zhang, eds., pp. 56–67, Fachberichte Informatik, Universität Koblenz-Landau
- Bernays, P., A System of Axiomatic Set Theory – Part I, *Journal of Symbolic Logic*, vol. 2 (1937) pp. 65–77.
- Bernays, P., *Axiomatic Set Theory*, North Holland Publishing Co., Amsterdam. First edition: 1958. Second edition: 1968. Republished in 1991 by Dover Publications, New York.
- Boyer, R., Lusk, E., McCune, W., Overbeek, R., Stickel M. and Wos, L., Set Theory in First Order Logic: Clauses for Gödel’s Axioms, *Journal of Automated Reasoning*, vol. 2 (1986) pp. 287–327.
- Farmer, W. M., A Set Theory for Mechanized Mathematics, *Journal of Automated Reasoning*, vol. 26 (2001) pp. 269–289.
- Formisano, A. and Omodeo, E. G., An Equational Re-Engineering of Set Theories, presented at the FTP’98 International Workshop on First Order Theorem Proving (Nov. 23–25, 1998).
- Gödel, K., The Consistency of the Axiom of Choice and of the Generalized Continuum Hypothesis with the Axioms of Set Theory, Princeton Univ. Press, Princeton, 1940.
- Jech, T., *Set Theory*, Academic Press, Inc., San Diego, CA, 1978.
- Kunen, K., *Set Theory: An Introduction to Independence Proofs*, Elsevier Science Publishers B. V., 1980.
- McCune, W. W., Otter 3.0 Reference Manual and Guide, Argonne National Laboratory Report ANL–94/6, Argonne National Laboratory, Argonne, IL, Jan. 1994.
- Megill, N. D., *Metamath: A Computer Language for Pure Mathematics*, 1997.
- Monk, J. D., *Introduction to Set Theory*, McGraw-Hill Book Co., New York, 1969.
- Noël, P. A. J., Experimenting with Isabelle in ZF set theory, *Journal of Automated Reasoning*, vol. 10 (1993) pp. 15–58.
- Paulson, L. C., and Grabczewski, K., Mechanizing Set Theory, *Journal of Automated Reasoning*, vol. 17 (1996) pp. 291–323.
- Quaife, A., Automated Deduction in von Neumann-Bernays-Gödel Set Theory, *Journal of Automated Reasoning*, vol. 8 (1992) pp. 91–147.
- Quaife, A., Automated Development of Fundamental Mathematical Theories, Ph.D. thesis, Univ. of California at Berkeley, Kluwer Acad. Publishers, Dordrecht, 1992.
- Rubin, J. E., *Set Theory for the Mathematician*, Holden-Day, San Francisco, 1967.
- Rudnicki, P., and Trybulec, A., On equivalents of well-foundedness, *Journal of Automated Reasoning*, vol. 23 (1999) pp. 197–234.
- Tarski, A., and Givant, S., *A Formalization of Set Theory without Variables*, American Mathematical Society Colloquium Publications, vol. 41, Providence, RI, 1987.
- Wos, L., *Automated Reasoning: 33 Basic Research Problems*, Prentice Hall, Englewood Cliffs, NJ, 1988.
- Wos, L., The problem of finding an inference rule for set theory, *Journal of Automated Reasoning*, vol. 5 (1989) pp. 93–95.
- Wos, L., Overbeek, R., Lusk, E. and Boyle, J., *Automated Reasoning: Introduction and Applications*, Second Ed., McGraw Hill, New York, 1992.
- Wolfram, S., *The Mathematica™ Book*, Wolfram Media Inc., Champaign, IL, 1996.