
introduction

Art Quaife developed the arithmetic of natural numbers within the formal system of Peano arithmetic **PA**.

**"Art Quaife, Automated Development of
Fundamental Mathematical Theories, Kluwer Academic
Publishers, Dordrecht, the Netherlands, 1992.";**

In the system **PA** the only objects are natural numbers. Finite sets of natural numbers can be codified as natural numbers in this system, but since there are more sets of numbers than there are numbers, such a system cannot deal with infinite sets of numbers, and it would be quite awkward to develop other number systems, such as the integers, modular numbers, rational numbers, let alone the real and complex number systems.

The goal of this research project is to develop numbers as a subsystem of set theory, thereby making it possible to talk not only about individual numbers, but also about properties of the number systems themselves. For example, one would like to be able to prove that modular numbers form cyclic groups under addition, and that every finite cyclic group is isomorphic to such a number system.

In addition to studying the number systems, one would also like to be able to apply them to elementary problems of combinatorics, such as Lagrange's theorem in group theory: the number of elements in a subgroup of a finite group is a divisor of the number of elements in the group itself.

The goals of this research have not yet been fully realized. This talk should be viewed as a progress report.

Gödel's algorithm

In Gödel's class theory, every thing is a class. A class is a **set** if it belongs to some class. The primitives of Gödel's class theory are: the universal class **V**, the membership relation **E**, the class constructors **pairset**, **complement**, **intersection**, **domain**, **flip**, **rotate** and **cart**, the predicates **equal** and **member**, subject to various axioms, including the axioms of regularity and choice. There is no general class constructor of the form **class[x, p[x]]** , but the proof of a general class existence metatheorem provides an algorithm for constructing expressions for classes in terms of the primitives. This is done by syntactically analyzing the statement **p[x]**, which must be reducible to membership statements using logical connectives and quantifiers over sets.

Historical comment: the algorithm presented by Kurt Gödel had been illustrated in papers by Paul Bernays several years prior to the publication of Gödel's monograph on the consistency of the axiom of choice and the generalized continuum hypothesis. Gödel formalized it concisely and gave a proof of termination.

"Kurt Gödel, The Consistency of the Axiom of Choice and of the Generalized Continuum Hypothesis with the Axioms of Set Theory, Princeton University Press, 1940.";

class rules

The **GOEDEL** program contains a set of rewrite rules for **class** that amount to a modification of Gödel's proof of his metatheorem. The main departures from Gödel's work is that the Kuratowski construction of ordered pairs is not assumed. Instead, **pair** is adopted as an additional primitive, and his algorithm is modified by having statements about **pair** be reduced to ones involving **equal** rather than **member**. In addition, neither the axiom of regularity nor the axiom of choice need be assumed.

```
In[2]:= Begin["Goedel`Private`"];
```

A few examples of the class rules will be stated now. The complete set of class rules can be found in the **GOEDEL** program itself, which is an ASCII text file available on the first author's website.

```
In[3]:= "http://www.math.gatech.edu/~belinfan/";
```

The following **class** rule converts **not** to **complement**. The expression **class[x, True]** is needed here because **x** need not be an atomic symbol, but could involve **pair**.

```
In[4]:= FirstMatch[class[x_, HoldPattern[not[p_]]]]
```

```
Out[4]= class[x_, not[p_]] := intersection[
        complement[class[x, p]], class[x, True]]
```

Another **class** rule converts **or** to **union**.

```
In[5]:= FirstMatch[class[x_, HoldPattern[or[p_, q_]]]]
```

```
Out[5]= class[x_, or[p_, q_]] := union[class[x, p], class[x, q]]
```

Existential quantifiers are eliminated in terms of the primitive class constructor **domain**:

```
In[6]:= FirstMatch[class[x_, HoldPattern[exists[y_, p_]]]]
out[6]= class[x_, exists[y_, p_]] := domain[class[pair[x, y], p]]
```

ordered pairs

The **GOEDEL** program deviates from Gödel's algorithm in that Kuratowski's construction of the ordered pair is not assumed. Statements about ordered pairs are reduced to statements about equality instead of membership. The basic rule about ordered pairs is:

```
In[7]:= equal[pair[u, v], pair[x, y]]
out[7]= and[equal[set[u], set[x]], equal[set[v], set[y]]]
```

Note that all variables are wrapped with **singleton = set**. This rule could be further expanded:

```
In[8]:= equal[set[u], set[x]] // AssertTest
out[8]= equal[set[u], set[x]] ==
        or[and[not[member[u, V]], not[member[x, V]]], equal[u, x]]
```

The equality of **pair[u,v]** and **pair[x,y]** only implies **u = x** and **v = y** when all variables are sets.

definitions

Classes are often defined in the **GOEDEL** program, by specifying a suitable membership rule. If this rule involves quantifiers, it is wrapped with **class** to prevent the introduction of Skolem functions. For example, the rule that defines the class **omega** of all natural numbers is:

```
In[9]:= FirstMatch[class[x_, HoldPattern[member[y_, omega]]]]
Out[9]= class[z_, member[x_, omega]] := Module[{w =
  Unique[]}, class[z, forall[w, implies[and[member[0,
  w], subclass[image[SUCC, w], w]], member[x, w]]]]]
```

Complicated concepts are often defined by rules wrapped with **class** to avoid having their definitions expanded out unnecessarily.

the nat wrapper

When arithmetic is developed within class theory, one can avoid a plethora of numberhood literals **member[x, omega]** by the use of the **nat** wrapper, defined as follows:

```
In[10]:= member[x, nat[y]] // AssertTest
Out[10]= member[x, nat[y]] == and[member[x, y], member[y, omega]]
```

The set **nat[x]** is always a natural number, and may be thought of as a generic natural number.

```
In[11]:= member[nat[x], omega]
Out[11]= True
```

Results expressed in terms of the **nat** wrapper can be easily converted into statements involving numberhood literals, and vice-versa. The main tools for

doing so are **SubstTest**, the rule for equality substitution, and the following rewrite rule:

```
In[12]:= equal[x, nat[x]]
Out[12]= member[x, omega]
```

other wrappers

Using wrappers to replace literals permits one to formulate more succinct rewrite rules than would otherwise be possible. One of the first wrappers introduced in the **GOEDEL** program was **funpart**.

```
In[13]:= intersection[composite[Id, x], complement[composite[Di, x]]]
Out[13]= funpart[x]
```

This definition is due to Formisano and Omodeo.

"A. Formisano and E. G. Omodeo, An Equational Re-Engineering of Set Theories, presented at the FTP'98 International Workshop on First Order Theorem Proving, November 23-25, 1998";

The wrapper **funpart** satisfies this basic property:

```
In[14]:= equal[x, funpart[x]]
Out[14]= FUNCTION[x]
```

This permits one to formulate rewrite rules such as the following, which says that vertical sections of functions are singletons, or are empty.

```
In[15]:= image[funpart[x], set[y]]
Out[15]= set[APPLY[funpart[x], y]]
```

Here is another such rule:

```
In[16]:= member[image[x, set[y]], range[SINGLETON]]
```

```
out[16]= member[y, domain[funpart[x]]]
```

There is a wrapper **setpart** for sets, **eqv** for equivalence relations, **oopart** for one-to-one correspondences, **rff** for reflexive relations, **thinpart** for thin relations, **trv** for transitive relations, and so on. (A relation is **thin** if every vertical section is a set.)

constructors versus functions

The rule that defines the union $U[x]$ of a collection of sets is:

```
In[17]:= FirstMatch[class[x_, HoldPattern[member[y_, U[z_]]]]]
Out[17]= class[w_, member[x_, U[z_]]] := Module[{y = Unique[]},
  class[w, exists[y, and[member[x, y], member[y, z]]]]]
```

The corresponding function is **BIGCUP**.

```
In[18]:= lambda[x, U[x]]
Out[18]= BIGCUP
```

The constructor U should not be confused with the function **BIGCUP**. Functions are classes of ordered pairs, but constructors are not. The class $U[x]$ gives the union of any class x , not just for sets. Applying **BIGCUP** yields the union for sets, and is equal to V otherwise.

```
In[19]:= APPLY[BIGCUP, x]
Out[19]= union[complement[image[V, set[x]]], U[x]]
```

The connection between the constructor and the corresponding function is:

```
In[20]:= class[pair[x, y], equal[y, U[x]]]
Out[20]= BIGCUP
```

BIGCAP

A similar rule is used to define the intersection $\mathbf{A}[\mathbf{x}]$ of a class of sets. This intersection is not a set when $\mathbf{x} = \mathbf{0}$. In that case $\mathbf{A}[\mathbf{0}] = \mathbf{V}$ is the universal class, which is a proper class. Accordingly, the corresponding function **BIGCAP** is not total: its domain is the complement of the singleton of the empty set:

```
In[21]:= lambda[x, A[x]]
```

```
Out[21]= BIGCAP
```

```
In[22]:= domain[BIGCAP]
```

```
Out[22]= complement[set[0]]
```

The natural numbers form the least successor-invariant set that holds $\mathbf{0}$.

```
In[23]:= A[dif[invar[SUCC], P[complement[set[0]]]]]
```

```
Out[23]= omega
```

defined predicates

Non-primitive predicates, such as the binary predicate **subclass**, whose definitions involve quantifiers, are likewise wrapped with **class**.

```
In[24]:= FirstMatch[class[x_, HoldPattern[subclass[y_, z_]]]]
```

```
Out[24]= class[w_, subclass[x_, y_]] := Module[{u = Unique[]}, class[
    w, forall[u, implies[member[u, x], member[u, y]]]]]
```

Another example is the unary predicate **FUNCTION**.

```
In[25]:= FirstMatch[class[x_, HoldPattern[FUNCTION[y_]]]]
Out[25]= class[w_, FUNCTION[x_]] := Module[{z = Unique[]}, class[
  w, and[subclass[x, cart[V, V]], forall[z, or[not[
    member[z, x]], not[member[z, composite[Di, x]]]]]]]]]
```

assert

Any statement in Gödel's class theory can be rewritten as an equation. In the process, all quantifiers in the original statement are eliminated. To do this, one uses **assert**, which is defined in terms of **class** as follows:

```
In[26]:= ?? assert
assert[p] is a statement equivalent to p
  obtained by applying Goedel's algorithm to class[w,p].
  Applying assert repeatedly sometimes simplifies a statement.
assert[p_] := Module[{w = Unique[]}, equal[V, class[w, p]]]
```

As an illustration, consider the statement that a collection of sets is closed under binary intersections:

```
In[27]:= assert[forall[u, v, implies[and[member[u, x], member[v, x]],
  member[intersection[u, v], x]]]]
Out[27]= subclass[image[CAP, cart[x, x]], x]
```

The function **CAP** corresponds to binary intersections:

```
In[28]:= lambda[pair[x, y], intersection[x, y]]
Out[28]= CAP
```

equality substitution

The **GOEDEL** program contains a conditional rewrite rule that recognizes equality substitutions for any predicate **p**:

```
In[29]:= implies[and[equal[x, y], p[x]], p[y]]
```

```
Out[29]= True
```

If there are many equality literals in a statement, many attempts to use equality substitution could be attempted, causing excessive execution time. In such cases, an **equality** flag can be cleared to temporarily remove this conditional rewrite rule. Various other flags are available to temporarily remove certain conditional rewrite rules.

```
In[30]:= flags
```

```
cond = True, equality = True, simplify = True, unsafe = False
```

Comment. The **unsafe** flag affects rewrite rules for mathematically valid facts which as rewrite rules have various undesired properties, such as causing infinite looping. These conditional rewrite rules have been left in the program to serve as warnings.

SubstTest

Most proofs of theorems and derivations of new rewrite rules in the **GOEDEL** program involve using **SubstTest** to compare different ways to simplify an expression. The definition of **SubstTest** is simple:

```
In[31]:= ?? SubstTest
      SubstTest compares two different orders of evaluation
      SubstTest[f_, x_, r_] := f@@({x} /. r) == (f@@{x} /. r)
```

For the purpose of deriving theorems, the head **f** is taken to be **and**, the various steps of the derivation are schematically indicated by a sequence of generic clauses **x**, and the replacement rules **r** replace the generic clauses with specific clauses that make all but the last item in **x** true. The idea is to arrange it so that **({x} /. r)** becomes a list of the form **{True, True, ... , not[p]}** where **p/.r** is the theorem to be proved, and the expression **and[x]** is false. In one order of evaluation, one obtains **not[p]** and the other yields **False**, thereby disproving the negation of the desired theorem. As a practical matter, among the main contributors to execution time are the reduction of **and[x]** to **False**, which amounts to a check that the form of the argument is valid, and the possibility that a large number conditional rewrite rules are made attempting to simplify statements.

removing set variables and reification

The **class** rules provide a standard procedure for eliminating set variables. Doing so may lead to simpler formulas and new insights, but when complicated concepts are present, using the class rules may cause excessive execution time. Another technique, involving the use of reification rules, often fares better in such cases.

```
In[32]:= ?reify
```

```
reify[x,F[x]] is the relation of all pair[x,y] with y belonging to F[x]
```

For each outer constructor, there is a rule expressing the reification of all compound constructors **F[G[x]]**, in terms of the reification of the inner constructor. For example, when **F = complement**, this rule is:

```
In[33]:= reify[x, complement[G[x]]]
```

```
Out[33]= composite[Id, complement[reify[x, G[x]]]]
```

Comment. Since reifications are typically proper classes, this technique is not available for theories limited to sets.

equipollence

The equipollence relation is:

```
In[34]:= class[pair[x, y], exists[z,
and[ONEONE[z], equal[x, domain[z]], equal[y, range[z]]]]]
```

```
Out[34]= Q
```

The cardinality of a class is the least ordinal equipollent to it.

```
In[35]:= A[intersection[OMEGA, image[Q, set[x]]]]
```

```
Out[35]= card[x]
```

Since the axiom of choice is not assumed, there may not be any ordinal equipollent to \mathbf{x} , and in this case the cardinality is \mathbf{V} . The cardinality is an ordinal if there is an ordinal equipollent to \mathbf{x} .

```
In[36]:= member[card[x], OMEGA]
```

```
Out[36]= member[x, image[Q, OMEGA]]
```

If every set is equipollent to an ordinal, then the axiom of choice holds.

```
In[37]:= implies[equal[V, image[Q, OMEGA]], axch]
```

```
Out[37]= True
```

finite sets

Finiteness is defined by a descending chain condition: a set is finite if it is not a member of a set that has the property that each member has another member which is a proper subset.

```
In[38]:= class[x,
           not[exists[y, and[member[x, y], forall[u, implies[member[u, y],
           exists[v, and[member[v, y], propersubclass[v, u]]]]]]]]]
```

```
Out[38]= FINITE
```

Theorem: A set is finite if and only if its cardinality is a natural number.

```
In[39]:= member[card[x], omega]
```

```
Out[39]= member[x, FINITE]
```

The pigeonhole principle:

```
In[40]:= implies[and[ONEONE[x], member[x, FINITE],
  subclass[range[x], domain[x]], equal[range[x], domain[x]]]
Out[40]= True
```

iterate[x,y]

Explicit use of mathematical induction in the arithmetic for natural numbers can largely be circumvented by use of the construct **iterate[x,y]**.

```
In[41]:= FirstMatch[class[w_, member[z_, HoldPattern[iterate[x_, y_]]]]]
Out[41]= class[w_, member[x_, iterate[y_, z_]]] :=
  Module[{p = Unique[]}, class[w, exists[p, and[
    member[x, p], subclass[p, union[cart[set[0], z],
    composite[y, p, inverse[SUCC], id[omega]]]]]]]]
```

The class **iterate[x,y]** is a relation whose domain is either a natural number of the set **omega** of all natural numbers:

```
In[42]:= member[domain[iterate[x, y]], succ[omega]]
Out[42]= True
```

The vertical section of the relation **iterate[x,y]** at **0** is the class **y**. the vertical section at each natural number **z** determines the vertical section at the successor **succ[z]** by imaging with **x**.

```
In[43]:= image[iterate[x, y], set[succ[z]]]
Out[43]= image[x, image[iterate[x, y], set[z]]]
```

The main tool for using **iterate** is the uniqueness theorem:

```
In[44]:= implies[and[equal[composite[w, SUCC], composite[x, w]],
  equal[image[w, set[0]], y]],
  equal[composite[w, id[omega]], iterate[x, y]]]
Out[44]= True
```

addition and multiplication

The arithmetic of natural numbers was developed using the uniqueness of iteration. The laws of arithmetic are available in the **GOEDEL** program in the form of a set of rewrite rules. For example, addition is iterated succession:

```
In[45]:= APPLY[iterate[SUCC, set[nat[x]]], nat[y]]
```

```
Out[45]= natadd[nat[x], nat[y]]
```

The **iterate** construct here is the function **plus[x]** which adds **x** to any number:

```
In[46]:= iterate[SUCC, set[nat[x]]]
```

```
Out[46]= plus[nat[x]]
```

```
In[47]:= VERTSECT[reify[y, natadd[x, y]]]
```

```
Out[47]= plus[x]
```

Comment: the combination of **VERTSECT** with **reify** is equivalent to **lambda** for functions with a single argument. The use of **lambda** in this example would take much more time. For binary functions, a similar technique is available:

```
In[48]:= VERTSECT[reify[x, natadd[first[x], second[x]]]]
```

```
Out[48]= NATADD
```

The membership rule that defines the binary function **NATADD** is:

```
In[49]:= member[pair[pair[x, y], z], NATADD]
```

```
Out[49]= and[member[z, omega], member[pair[x, z], iterate[SUCC, set[y]]]]
```

For multiplication, a double iteration is used:

```
In[50]:= member[pair[pair[x, y], z], NATMUL] // AssertTest
Out[50]= member[pair[pair[x, y], z], NATMUL] == and[member[x, omega],
            member[pair[y, z], iterate[iterate[SUCC, set[x]], set[0]]]]
```

combinatorics

Two basic theorems about counting have been derived with the **GOEDEL** program. One is the rule for double-counting:

```
In[51]:= natadd[card[union[x, y]], card[intersection[x, y]]]
Out[51]= natadd[card[x], card[y]]
```

Despite first appearances this is only about finite sets. If either x or y is not finite, then this reduces to the uninteresting fact that $V = V$.

```
In[52]:= equal[V, natadd[card[x], card[y]]]
Out[52]= or[not[member[x, FINITE]], not[member[y, FINITE]]]
```

The other theorem is that the cardinality of a cartesian product of two finite sets is the product of their cardinalities.

```
In[53]:= implies[and[member[x, FINITE], member[y, FINITE]],
                equal[card[cart[x, y]], natmul[card[x], card[y]]]]
Out[53]= True
```

This can be restated without variables:

```
In[54]:= composite[CARD, CART, id[cart[FINITE, FINITE]]]
Out[54]= composite[NATMUL, cross[CARD, CARD]]
```

the factorial function

A recent example of a definition in terms of **iterate** is provided by the factorial function:

```
In[55]:= composite[SECOND,
  iterate[intersection[composite[inverse[SECOND], NATMUL],
    composite[inverse[FIRST], SUCC, FIRST]],
  cart[set[set[0]], set[set[0]]]]]
```

```
Out[55]= FACTORIAL
```

The uniqueness of **iterate** was used to establish these basic properties of this function:

```
In[56]:= APPLY[FACTORIAL, 0]
```

```
Out[56]= set[0]
```

The recursion relation is:

```
In[57]:= composite[FACTORIAL, SUCC]
```

```
Out[57]= composite[NATMUL, intersection[composite[inverse[FIRST], SUCC],
  composite[inverse[SECOND], FACTORIAL]]]
```

When expressed in terms of **APPLY**, the distributive law kicks in and makes for a somewhat messy result: This says $(x+1)! = x x! + x!$.

```
In[58]:= Map[A, ImageComp[FACTORIAL, SUCC, set[x]]]
```

```
Out[58]= natadd[APPLY[FACTORIAL, x], natmul[x, APPLY[FACTORIAL, x]]] ==
  APPLY[FACTORIAL, succ[x]]
```

divisibility and primes

The divisibility relation is defined by

```
In[59]:= composite[NATMUL, inverse[FIRST]]
```

```
Out[59]= DIV
```

The set of all multiples of x is **image**[DIV, set[x]], and the set of its divisors is **image**[inverse[DIV], set[x]]. Primes are defined as sets with exactly two divisors.

```
In[60]:= equal[succ[set[0]], card[image[inverse[DIV], set[x]]]]
```

```
Out[60]= member[x, PRIMES]
```

For example, 3 is prime because it has exactly two divisors:

```
In[61]:= card[image[inverse[DIV], set[succ[set[0]]]]]
```

```
Out[61]= succ[set[0]]
```

The fact that there are infinitely many primes was deduced by using the factorial function.

```
In[62]:= member[PRIMES, FINITE]
```

```
Out[62]= False
```

The least prime larger than a finite set of numbers x is **hull**[PRIMES, x]. In particular, the next prime after a given prime x is **hull**[PRIMES, succ[x]].

```
In[63]:= member[hull[PRIMES, x], PRIMES]
```

```
Out[63]= and[member[x, FINITE], subclass[x, omega]]
```

The least prime is **2**, the next is **3**, and after that comes **5**:

```
In[64]:= hull[PRIMES, 0]
```

```
Out[64]= succ[set[0]]
```

```
In[65]:= hull[PRIMES, succ[succ[set[0]]]]
```

```
Out[65]= succ[succ[set[0]]]
```

```
In[66]:= hull[PRIMES, succ[succ[succ[set[0]]]]]
Out[66]= succ[succ[succ[succ[set[0]]]]]
```

remainders

The reduction of a number x modulo a number y is defined as the least number obtainable from x by subtracting a multiple of y .

```
In[67]:= A[image[image[inverse[NATADD], set[x]], image[DIV, set[y]]]]
Out[67]= natmod[x, y]
```

For example, reducing **5** modulo **3** yields **2**:

```
In[68]:= natmod[succ[succ[succ[succ[set[0]]]]], succ[succ[set[0]]]]
Out[68]= succ[set[0]]
```

The following uniqueness theorem holds for remainders:

```
In[81]:= implies[and[member[pair[y, natsub[x, z]], DIV], member[z, y]],
  equal[z, natmod[x, y]]]
Out[81]= True
```

The authors rederived Quaife's theorems concerning remainders, and in some case went a bit further. For instance, Quaife's theorem (**DV22**) has this converse:

```
In[82]:= implies[member[pair[z, natsub[x, y]], DIV],
  equal[natmod[x, z], natmod[y, z]]]
Out[82]= True
```

modular arithmetic

The function that reduces modulo y is:

```
In[69]:= VERTSECT[reify[x, natmod[x, y]]]
```

```
Out[69]= modulo[y]
```

This function idempotent:

```
In[70]:= composite[modulo[x], modulo[x]]
```

```
Out[70]= modulo[x]
```

Art Quaife had studied modular arithmetic within the framework of Peano Arithmetic, without the benefit of set theory. The authors translated Quaife's clauses into the language used in the **GOEDEL** program. Some of Quaife's theorems become rather pretty when expressed with fewer variables:

```
In[71]:= composite[modulo[x], plus[y], modulo[x]]
```

```
Out[71]= composite[modulo[x], plus[y]]
```

```
In[72]:= composite[modulo[x], times[y], modulo[x]]
```

```
Out[72]= composite[modulo[x], times[y]]
```

associativity

A ternary relation x is associative if it satisfies a certain variable-free formulation of the associative law.

```
In[73]:= equiv[associative[x], and[subclass[x, cart[cart[V, V], V]],
    equal[composite[x, cross[x, Id]],
    composite[x, cross[Id, x], ASSOC]]] // not // not
```

```
Out[73]= True
```

Various examples are recognized by the **GOEDEL** program:

```
In[74]:= Select[NamedClasses, associative]
```

```
Out[74]= {0, CAP, CATOFUNS, CATORELN, COMPOSE, CUP,
    FIRST, INTADD, NATADD, NATMUL, SECOND, SYMDIF}
```

The function **CATOFUNS** is the partial binary composition for the category of sets. Other examples include addition and multiplication modulo x .

```
In[75]:= associative[composite[modulo[x], NATADD]]
```

```
Out[75]= True
```

```
In[76]:= associative[composite[modulo[x], NATMUL]]
```

```
Out[76]= True
```

theorems about associativity

Several general results about associative relations have been derived: the direct product of associative relations is associative:

```
In[77]:= implies[and[associative[x], associative[y]],
               associative[direct[x, y]]]
```

```
Out[77]= True
```

From any thin associative relation one can construct an associative function:

```
In[78]:= implies[and[thin[x], associative[x]],
               associative[composite[IMAGE[x], CART]]]
```

```
Out[78]= True
```

An example of this is the construction of the associative function for composition from the semigroupoid:

```
In[79]:= associative[composite[SWAP, RIF]]
```

```
Out[79]= True
```

```
In[80]:= composite[IMAGE[composite[SWAP, RIF]], CART]
```

```
Out[80]= COMPOSE
```

conclusions

Computers are can be used in mathematical reasoning in many ways:

1. finding proofs
2. finding counterexamples
3. verifying proofs
4. simplifying proofs
5. formulating definitions
- 6 simplifying expressions
7. discovering new facts
8. calculating

Unlike **Otter**, the **GOEDEL** program does not do searches, neither for proofs nor for counter-examples. The program is not intended to replace such automated reasoning programs, but only to serve as an aid in formulating definitions and theorems, and to help discovery new facts. It can verify proofs. Because many mathematical facts are built into the program in the form of rewrite rules, one can sometimes take a partial statement of a theorem and use the program to discover what further conditions need to be added to make the result valid.

a few unsolved problems

The research reported in this paper suggests many interesting challenges for further developing the **GOEDEL** program, including the following:

1. To derive the fact that $\text{card}[\mathbf{x}] = \mathbf{\omega}$ for any infinite set of natural numbers. In particular, to show $\text{card}[\mathbf{PRIMES}] = \mathbf{\omega}$.
2. To develop group theory and use it to help in the construction of the arithmetic of integers and rational numbers.
3. Can multiplication of integers be introduced as an automorphism of integer addition instead of using separate rules for positive and negative numbers?
4. Establish more interesting connections between counting and arithmetic, such as Lagrange's theorem for finite groups.
5. Develop the theory of greatest common divisors, and other facts that Quaipe proved within Peano Arithmetic.
6. Derive the fact that the axiom of choice implies that every set is equipollent to an ordinal.