Gödel’s Algorithm for Class Formation

Johan Gjisbertus Frederik Belinfante

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

Abstract. A computer implementation of Gödel’s algorithm for class formation in Mathematica™ is useful for automated reasoning in set theory. The original intent was to forge a convenient preprocessing tool to help prepare input files for McCune’s automated reasoning program Otter. The program is also valuable for discovering new theorems. Some applications are described, especially to the definition of functions. A brief extract from the program is included in an appendix.

1 Introduction.

Robert Boyer et al. (1986) proposed clauses capturing the essence of Gödel’s finite axiomatization of the von Neumann-Bernays theory of sets and classes. Their work was simplified significantly by Art Quaife (1992a and 1992b). About four hundred theorems of elementary set theory were proved using McCune’s automated reasoning program Otter. A certain degree of success has been achieved recently (1999a and 1999b) in extending Quaife’s work. Some elementary theorems of ordinal number theory were proved, based on Isbell’s definition (1960) of ordinal number, which does not require the axiom of regularity to be assumed.

An admitted disadvantage of Gödel’s formalism is the absence of the usual class formation \( \{ x \mid p(x) \} \) notation. Replacing the axiom schema for class formation are a small number of axioms for certain basic class constructions. Definitions of classes must be expressed in terms of two basic classes, the universal class \( V \) and the membership relation \( E \), and seven other basic class constructors: the unary constructors complement, domain, flip and rotate, and the binary constructors pairset, cart, intersection. Gödel also included an axiom for inverse, but it can be deduced from the others.

2 A brief description of the GOEDEL program.

As a replacement for the axiom schema for class formation, Kurt Gödel (1940) proved a fundamental Class Existence Metatheorem Schema for class formation. His proof of this metatheorem is constructive: a recursive algorithm for converting customary definitions of classes using class formation to expressions built out of the primitive constructors is presented, together with a proof of termination. An implementation of Gödel’s algorithm in Mathematica™ was created (1996)
to help prepare input files for proofs in set theory using McCune’s automated reasoning program Otter.

The likelihood of success in proving theorems using programs like Otter depends critically on the simplicity of the definitions used and the brevity of the statements of the theorems to be proved. To mitigate the effects of combinatorial explosion, one typically sets a weight limit to exclude complicated expressions from being considered. Although combinatorial explosion can not be prevented, the idea is to snatch a proof quickly before the explosion gets well under way.

Because one needs compact definitions for practical applications, and because the output of Gödel’s original algorithm is typically extremely complicated, a large number of simplification rules were added to the Mathematica implementation of Gödel’s algorithm. With the addition of simplification rules, Gödel’s proof of termination no longer applies. No assurance can be given that the added simplification rules will not cause looping to occur, but we have tested the program on a suite of several thousand examples, and it appears that it can be used as a practical tool to help formulate definitions and to simplify the statements of theorems. The GOEDEL program contains no mechanism for carrying out deductions, but it does sometimes manage to prove statements by simplifying them to True.

Much of the complexity of Gödel’s original algorithm stems from his use of Kuratowski’s definition for ordered pairs. The Mathematica implementation does not assume any particular contraction of ordered pairs, but instead includes additional rules to deal with ordered pairs. The self-membership rule in the original algorithm was modified because in our work on ordinal numbers the axiom of regularity is not assumed.

The stripped down version of the GOEDEL program presented in the Appendix omits many membership rules for defined constructors as well as most of the simplification rules. The modified Gödel’s algorithm is presented as a series of definitions for a Mathematica function class[x,y]. The first argument x, assumed to be the name of 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. It should be noted that Gödel did not allow both u and v to be pairs, but this unnecessary limitation has been removed to make the formalism more flexible. The second argument p is some statement which can involve the variables that appear in x, as well as other variables that may represent arbitrary classes (not just sets). The statement can contain quantifiers, but the quantified variables must be sets. The Gödel algorithm does not apply to statements containing quantifiers over proper classes. The quantifiers forall and exists used in the GOEDEL program are explicitly restricted to set variables.

A few simple examples will be presented to illustrate how the GOEDEL program is used. For convenience, Mathematica style notation will be employed, which does not quite conform to the notational requirements of Otter. For example, Mathematica permits one to define intersection to be an associative and commutative function of any number of variables. For brevity we write
a → b to mean that Mathematica input a produces Mathematica output b for some version of the GOEDEL program.

The functions FIRST and SECOND which project out the first and second components of an ordered pair, respectively, can be specified as the classes

\[
\text{class} \, \text{pair} \, \text{pair} \, x, y, z, \, \text{equal} \, z, x \mid \rightarrow \text{FIRST},
\text{class} \, \text{pair} \, \text{pair} \, x, y, z, \, \text{equal} \, z, y \mid \rightarrow \text{SECOND}
\]

Examples which involve quantifiers include the domain and range of a relation:

\[
\text{class} \, x, \, \text{exists} \, y, \, \text{member} \, \text{pair} \, x, y, z \mid \rightarrow \text{domain} \, z,
\text{class} \, y, \, \text{exists} \, x, \, \text{member} \, \text{pair} \, x, y, z \mid \rightarrow \text{range} \, z.
\]

It is implicitly assumed that all quantified variables refer to sets, but the free variable z here can stand for any class.

3 Eliminating flip and rotate.

Gödel's algorithm uses two special constructors \text{flip} \,[x] and \text{rotate} \,[x] which produce ternary relations. The ternary relation \text{flip} \,[x] is

\[
\text{class} \, \text{pair} \, \text{pair} \, u, v, w, \, \text{member} \, \text{pair} \, v, u, w, x
\]

while \text{rotate} \,[x] is

\[
\text{class} \, \text{pair} \, \text{pair} \, u, v, w, \, \text{member} \, \text{pair} \, v, w, u, x.
\]

Because these functors are not widely used in mathematics, it may be of interest to note that they could be eliminated in favor of more familiar ones. One can rewrite \text{flip} \,[x] as \text{composite} \,[x, \text{SWAP}] , where \text{SWAP} = \text{flip} \,[\text{Id}] is the relation

\[
\text{class} \, \text{pair} \, \text{pair} \, u, v, \, \text{pair} \, x, y, \, \text{and} \, \text{equal} \, u, y, \, \text{equal} \, v, x \mid \rightarrow \text{SWAP}.
\]

Note that the functions that project out the first and second members of an ordered pair are related by \text{SECOND} = \text{flip} \,[\text{FIRST}] and \text{FIRST} = \text{flip} \,[\text{SECOND}].

The general formula for \text{rotate} \,[x] is more complicated, but Gödel's algorithm actually only involves the special case where \text{x} is a Cartesian product. In this special case one has the simple formula,

\[
\text{rotate} | \text{cart} \,[x, y, z] | := \text{composite} \,[x, \text{SECOND}, \text{id} | \text{cart} \,[y, V]|].
\]

Using these formulas, the constructors \text{flip} and \text{rotate} could be completely eliminated from Gödel's algorithm, as well as from Gödel's axioms for class theory, if one instead takes as primitives the constructors \text{composite}, \text{inverse}, \text{FIRST} and \text{SECOND}. We have done so in the abbreviated version of the GOEDEL program listed in the Appendix. The function \text{SWAP} mentioned above, for example, could be defined in terms of these new primitives as

\[
\text{intersection} \,[\text{composite} \,[\text{inverse} \,[\text{FIRST}], \text{SECOND}],
\text{composite} \,[\text{inverse} \,[\text{SECOND}], \text{FIRST}] ] := \text{SWAP}.
\]
4 Equational set theory without variables.

The simplification rules in the GOEDEL program can be used not only to simplify descriptions of classes, but can also be induced to simplify statements. Given any statement \( p \), one can form the class \( \text{class}[w, p] \) 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. One can form a new statement equivalent to the original one by the definition

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

The occurrence of \text{class} causes Gödel's algorithm to be invoked, the meaning of the statement \( p \) to be interpreted, and the simplification rules in the GOEDEL program to be applied. While there can be no assurance the transformed statement will actually be simpler than the statement one started with, in practice it often is. For instance, the input

\[
\text{assert}[\text{equal}[\text{composite}[\text{cross}[x, y]], \text{DUP}]; \text{composite}[\text{DUP}, x]]
\]

produces the statement \text{FUNCTION}[\text{composite}[\text{Id}, x]] as output. To improve the readability of the output, in the current version of the GOEDEL program, rules have been added which may convert the equations obtained with \text{assert} back to simpler non-equational statements.

Since some theorem provers are limited to equational statements, it is of interest to reformulate set theory in equational terms. Alfred Tarski and Steven Givant (1987) have shown that all statements of set theory can be reformulated as equations without variables, somewhat reminiscent of combinatory logic. But whereas combinatory logic uses function-like objects as primitives, their calculus is based on the theory of relations. It has recently been proposed by Omodeo and Formisano (1998) that this formalism be used to recast set theory in a form accessible to purely equational automated reasoning programs. It is interesting to note that the \text{assert} mechanism in the GOEDEL program achieves the same objective. Any statement is converted by \text{assert} into an equation of the form \text{equal}[V, x]. If one prefers, one may also write this equation in the equivalent form \text{equal}[0, \text{complement}[x]]).

Another consequence of the \text{assert} process is that one can always convert negative statements into positive ones. For example, the negative statement \text{not}[\text{equal}[0, x]] is converted by \text{assert} into the equivalent positive statement \text{equal}[V, \text{image}[V, x]]. Thus it appears that at least in set theory it does not make too much sense to make a big distinction between positive and negative literals, because one can always convert the one into the other. Also, one can always convert a clause with several literals into a unit clause; the clause \text{or}[\text{equal}[0, x], \text{equal}[0, y]], for example, is equivalent to the unit clause

\[
\text{equal}[0, \text{intersection}[\text{image}[V, x], \text{image}[V, y]]].
\]

The class \text{image}[V, x] which appears in these expressions is equal to the empty set if \( x \) is empty, and is equal to the universal class \( V \) if \( x \) is not empty. This class
is quite useful for reformulating conditional statements as unconditional ones. Many equations in set theory hold only for sets and not for proper classes. For example, the union of the singleton of a class $x$ is $x$ when $x$ is a set, but is the empty set otherwise. This rule can be written as a single equation which applies to both cases as follows:

$$\text{union}(\text{singleton}(x), \text{intersection}(x, \text{image}(V, \text{singleton}(x))))$$

(This is in fact one of the thousands of rules in the GOEDEL program.) Although such unconditional statements initially appear to be more complex than the conditional statements that they replace, experience both with Otter and with the GOEDEL program indicates that the unconditional statements are in fact preferable. In Otter the unconditional rule can often be added to the demodulator list. In Mathematica, an unconditional simplification rule generally works faster than a conditional one.

When $\text{assert}$ is applied to a statement containing quantifiers, the statement is converted to a logically equivalent equation without quantifiers. All quantified variables are eliminated. What happens is that the quantifiers are neatly built into equivalent set-theoretic constructs like domain and composite. For example, the axiom of regularity is usually formulated using quantifiers as:

$$\text{implies}(\neg \text{equal}(x, 0), \exists u. \text{member}(u, x), \text{disjoint}(u, x))$$

When $\text{assert}$ is applied, this statement is automatically converted into the equivalent quantifier-free statement

$$\text{or}(\text{equal}(0, x), \neg \text{subclass}(x, \text{complement}(P)), \text{complement}(x))$$

In this case the quantifier was hidden in the introduced power class functor. Replacing $x$ by its complement, one obtains the following neat reformulation of the axiom of regularity:

$$\text{implies}(\text{subclass}(P, x), \text{equal}(x, V))$$

That is, the axiom of regularity says that the universal class is the only class which contains its own power class. When the axiom of regularity is not assumed, there may be other classes with this property. In particular, the Russell class $\text{RUSSELL} = \text{complement}(\text{fix}(E))$ has this property, a fact that is useful in the Otter proofs in ordinal number theory.

This reformulation of the axiom of regularity has the advantage over the original one in that its classification does not introduce new Skolem functions.

5 Functions, vertical sections and cancellation machines.

The process of eliminating variables and hiding quantifiers is facilitated by having available a supply of standard functions corresponding to the primitive constructors, as well as important derived constructors. For example, Quaife introduced the function $\text{SUCCE}$ corresponding to the constructor

$$\text{SUCCE}(x) := \text{union}(x, \text{singleton}(x))$$
so that the statement that the set $\omega$ of natural numbers is closed under
the successor operation could be written in the compact variable-free form as
the condition $\text{subclass}(\text{image}(\text{SUC.} \omega), \omega)$. This is just one of the many
techniques that Quaife exploited to reduce the plethora of Skolem functions that
had appeared in the earlier work of Robert Boyer, et al.

Replacing the function symbols of first order logic by bona fide set-theoretic
functions not only helps to eliminate Skolem functions, but also improves the
readability of the statements of theorems. A standard way to obtain definitions
for most of these functions is in terms of a basic constructor $\text{VERTSECT}$, enabling
one to introduce a lambda calculus for defining functions by specifying the re-
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

$$\text{image}_z(z \cdot \text{singleton}_x(x) = \text{class} \{ y \mid \text{member}(\text{pair}(x, y), z) \}. $$

One is naturally led to introduce the function which assigns these vertical
sections:

$$\text{VERTSECT}(z) = \text{class} \{ \text{pair}(x, y) \mid \text{equal}(y, \text{image}_z(z \cdot \text{singleton}_x(x))) \}.$$ 

(Formisano and Omodeo (1998) call this function $\forall(z)$. ) Gödel’s algorithm con-
verts this formula to the expression

$$\text{VERTSECT}(z) = \text{composite}(\text{Id}, \text{intersection}[
\ \text{complement} \{ \text{composite}(\text{E}, \text{complement}(z)) \},
\ \text{complement} \{ \text{composite}(\text{complement}(\text{E}, z)) \}].$$

Of course, for many relations $z$ the vertical sections need not be sets. The domain
of $\text{VERTSECT}(z)$ in general is the class of all sets $x$ for which $\text{image}_z(z \cdot \text{singleton}_x(x)$
is also a set. We call a relation thin when all vertical sections are sets. In addi-
tion to functions, there are many important relations, such as $\text{inverse}_E$ and
$\text{inverse}_E(\cdot)$, that are thin.

Using Otter, we have proved many facts about $\text{VERTSECT}$, making it unnec-
essary to repeat such work for individual functions.

When $f$ is a function, $\text{image}_f(z \cdot \text{singleton}_x(x)$ is a singleton, and one can
select the element in that singleton by applying either the sum class operation
$U$, as Quaife does, or by applying the unary intersection operation $A$ defined by

$$\text{class}(u, \forall v. \text{implies}(\text{member}(v, x), \text{member}(u, v))) \rightarrow A(x)$$

or equivalently,

$$\text{complement} \{ \text{image}(\text{complement}(\text{inverse}_E \cdot x) \rightarrow A(x).$$

The difference between using $U$ and $A$ only affects the case that $x$ is a proper
class. Nevertheless, using $A$ instead of $U$ in the definition of application has many
practical advantages.

For example, one can use $\text{VERTSECT}$ to obtain a formula for any function from
a formula for its application $A(\text{image}_f(z \cdot \text{singleton}_x(x))$. This can be done neatly
in the GOEDEL program by introducing the Mathematica definition
\begin{verbatim}
lambda[x_,e_]:=
  Module[{y=Unique[]}, VERTSECT[Class[Pair[x,y], member[y,e]]]]

This Mathematica function \texttt{lambda} satisfies:

\begin{verbatim}
  FUNCTION[f := True; lambda[x, A[image[f, singleton[x]]]] \rightarrow f.
\end{verbatim}

It should be noted that nothing like this works when one replaces \texttt{A} by \texttt{U} because \texttt{U} does not distinguish between 0 and \texttt{singleton}(0), whereas \texttt{A} does. For the constant function \( f := \text{cart}[x.\text{singleton}[0]], \) for example, one has \( U\text{image}[f, \text{singleton}[y]] \rightarrow 0, \) whereas

\begin{verbatim}
  A\text{image}[f, \text{singleton}[y]] \rightarrow
  \text{complement}_{\text{image}[V, \text{intersection}[x, \text{singleton}[y]]]].
\end{verbatim}

Because the formula for \( U\text{image}[f, \text{singleton}[y]] \) has lost all information about the domain \( x \) of the function \( f \), one cannot reconstruct \( f \) from this formula, but one can reconstruct \( f \) from the formula for \( A\text{image}[f, \text{singleton}[y]] \).

As examples of definitions obtained using \texttt{lambda} we mention the function \texttt{SINGLETON} which takes any set to its singleton,

\begin{verbatim}
  lambda[x, \text{singleton}[x]] \rightarrow \text{VERTSECT}[\text{Id}],
\end{verbatim}

and the function \texttt{POWER} which takes any set to its power set,

\begin{verbatim}
  lambda[x, P[x]] \rightarrow \text{VERTSECT}[\text{inverse}[S]].
\end{verbatim}

The function \texttt{VERTSECT}[x] itself satisfies

\begin{verbatim}
  lambda[w, \text{image}[x, \text{singleton}[w]]] \rightarrow \text{VERTSECT}[x].
\end{verbatim}

In addition to \texttt{VERTSECT}, it is also convenient to introduce a related constructor \texttt{IMAGE}, defined by

\begin{verbatim}
  \text{VERTSECT}[\text{composite}[x, \text{inverse}[E]]] := \text{IMAGE}[x].
\end{verbatim}

The function \texttt{IMAGE}[x] satisfies

\begin{verbatim}
  lambda[u, \text{image}[x, u]] \rightarrow \text{IMAGE}[x].
\end{verbatim}

The definition \texttt{IMAGE}[\text{inverse}[E]] := \text{BIGCUP} of the function \texttt{BIGCUP} which corresponds to the constructor \texttt{U}[x] was one of the first applications found for \texttt{IMAGE}. The function \texttt{IMAGE}[\text{inverse}[S]] is the hereditary closure operator, which takes any set \( x \) to its hereditary closure \texttt{image}[\text{inverse}[S], x]. This function is closely related to the \texttt{POWER} function mentioned earlier. The functions \texttt{IMAGE}[\text{FIRST}] and \texttt{IMAGE}[\text{SECOND}] take \( x \) to its domain and range, respectively, while \texttt{IMAGE}[\text{SWAP}] takes \( x \) to its inverse. The function \texttt{IMAGE}[\text{cross}[u, v]] takes \( x \) to \texttt{composite}[v, x, \text{inverse}[u]]. For example, the function that corresponds to the constructor \texttt{flip} is \texttt{IMAGE}[\text{cross}[\text{SWAP}, \text{Id}]].
The constructor **IMAGE** is not a functor in the category theory sense. The function **IMAGE**(x) does not in general preserve composites, but only when the right hand factor is thin:

\[ \text{domain} \text{\{VERTSECT\}}\{x\} := V; \]
\[ \text{IMAGE} \text{\{composite\}}\{x, t\} \rightarrow \text{composite} \text{\{IMAGE\}}\{x\}, \text{IMAGE}\{t\}. \]

**IMAGE** preserves the global identity function: \( \text{IMAGE}\text{\{Id\}} \rightarrow \text{Id} \); but in general \( \text{IMAGE}\text{\{id\}}\{x\} \) is not an identity function. It is nonetheless a useful function:

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

An important application of **VERTSECT** is to provide a mechanism for recovering a function \( f \) from a formula for \( \text{composite} \text{\{inverse\}}\{E\}, f \). One can use **VERTSECT** to cancel factors of \( \text{inverse} \text{\{E\}} \); for example, the Mathematica input

\[
\begin{align*}
\text{FUNCTION}\{f1\} := \text{True}; & \quad \text{FUNCTION}\{f2\} := \text{True}; \\
\text{domain}\{f1\} := V; & \quad \text{domain}\{f2\} := V; \\
\text{Map}\{\text{VERTSECT}, \text{composite}\{\text{inverse}\{E\}, f1\}\} & \Rightarrow \text{composite}\{\text{inverse}\{E\}, f2\}
\end{align*}
\]

produces the output \( f1 = f2 \). When the assumption about the domains of the functions are omitted, the results are slightly more complicated, but one nonetheless can obtain a formula for each function in terms of the other.

It is possible to use **VERTSECT** to construct other such cancellation machines which cancel factors of \( S, \text{inverse} \text{\{S\}} \) or **DISJOINT**. These machines were found to be quite useful in our investigations of the binary functions which correspond to the constructors **intersection**, **cart**, **union** and so forth.

6 Binary functions and proof by rotation.

Binary functions such as **CART**, **CAP**, **CUP**, corresponding to the constructors **cart**, **intersection**, **union** are important for obtaining variable-free expressions in many applications. To apply the **lambda** formalism to these functions, it is convenient to introduce the abbreviations

\[
\begin{align*}
\text{first}\{x\} := & \ A[\text{domain}[\text{singleton}\{x\}]]; \\
\text{second}\{x\} := & \ A[\text{range}[\text{singleton}\{x\}]].
\end{align*}
\]

One then has

\[
\begin{align*}
\text{lambda}\{x \text{. intersection}\{first\{x\}, second\{x\}\}\} & \rightarrow \text{CAP}; \\
\text{lambda}\{x \text{. union}\{first\{x\}, second\{x\}\}\} & \rightarrow \text{CUP}; \\
\text{lambda}\{x \text{. cart}\{first\{x\}, second\{x\}\}\} & \rightarrow \text{CART}.
\end{align*}
\]

(It should be noted that **first** and **second** here are technically different from the rather similar constructors **1st** and **2nd** introduced by Quaife.)

Although Gödel's **rotate** functor can be completely eliminated, nevertheless it does in fact have many desirable properties. For example, the **rotate** functor
preserves unions, intersections and relative complements, whereas \texttt{composite} preserves only unions. In the study of binary functions, the \texttt{rotate} constructor has turned out to be extremely useful. Often we can take one equation for binary functions and rotate it to obtain another.

The \texttt{SYMDIF} function corresponding to the symmetric difference operation is rotation invariant. Schroeder's transposition theorem can be given a succinct variable-free formulation as the statement that the relation

\[
\texttt{composite} \text{DISJOINT, IMAGE|SWAP|, COMPOSE}
\]

is rotation invariant, where \texttt{DISJOINT} is \texttt{classify|pair|x,y|, disjoint|x,y|}, and \texttt{COMPOSE} is the binary function corresponding to \texttt{composite}.

We mention three applications of these binary functions for defining classes. The class of all transitive relations can be specified as:

\[
\text{class}[x, \text{subclass}[\texttt{composite}(x,x,x) \rightarrow \texttt{fix}[\texttt{composite}[S, \text{COMPOSE, DUP}])].
\]

The class of all disjoint collections, specified as the input

\[
\text{class}[z, \forall [x,y], \text{implies}[\text{and}[\text{member}(x,z), \text{member}(y,z)],
\text{or}[\text{equal}(x,y), \text{disjoint}(x,y)])]]
\]

produces

\[
\text{fix}[\text{image}|\text{inverse}|\text{CART}, P|\text{union}|\text{DISJOINT}, \text{Id}][[]]
\]

as output. The class of all topologies, input as

\[
\text{class}[t, \text{and}[\text{subclass}[\text{image}[\text{BIGCUP}, P[t]], t],
\text{subclass}[\text{image}[\text{CAP}, \text{cart}[t,t]], t]]]
\]

produces the output

\[
\text{intersection}[
\text{complement}[\text{fix}[\text{composite}[\text{complement}[E], \text{BIGCUP}, \text{inverse}[S]]]],
\text{fix}[\text{composite}[S, \text{IMAGE}[\text{CAP}, \text{CART, DUP}]]].
\]

7 Conclusion.

Proving theorems in set theory with a first order theorem prover such as \texttt{Otter} is greatly facilitated by the use of a companion program \texttt{GOEDEL} which permits one to automatically translate from the notations commonly used in mathematics to the special language needed for the Gödel theory of classes.

Having an arsenal of set-theoretic functions that correspond to the function symbols of first order logic proves to be useful for systematically eliminating existential quantifiers and thereby avoiding the Skolem functions produced when formulas with existential quantifiers are converted to clause form. Although the main focus in this talk was on the use of the \texttt{GOEDEL} program to help find convenient definitions for all these functions, the \texttt{GOEDEL} program also permits one to
discover useful formulas that these functions satisfy. By adding these formulas as new simplification rules, the program has grown increasingly powerful over the years.

The GOEDEL program currently contains well over three thousand simplification rules, many of which have been proved valid using Otter. The simplification rules can be used not only to simplify definitions, but also to simplify statements. This power to simplify statements has led to the discovery of many new formulas, especially new demodulators. Experience with Otter indicates that searches for proofs are dramatically improved by the presence of demodulators even when they are not directly used in the proof of a theorem because they help to combat combinatorial explosion.

Appendix. An extract from the GOEDEL program.

Print[':Package Title: GOEDEL.M 2000 January 13 at 6:45 a.m. '];
(*
:Context: Goedel
:Mathematica Version: 3.0  Author: Johan G. F. Belinfante
:Summary: The GOEDEL program implements Goedel's algorithm for class
formation, modified to avoid assuming the axiom of regularity, and
Kuratowski's construction of ordered pairs.
:Sources: <description of algorithm, information for experts>
Kurt Goedel, 1939 monograph on consistency of the axiom of choice and
the generalized continuum hypothesis, pp. 9-14.
:Warnings: <description of global effects, incompatibilities>
0 is used to represent the empty set.
E is used to represent the membership relation.
:Limitations: <specific cases not handled, known problems>
The simplification rules are not confluent; termination is not assured.
There is no user control over the order that simplification rules are applied.
This stripped down version of GOEDEL.M.A23 lacks 95% of the simplification
rules needed to produce good output. Mathematica's builtin Tracing commands
are the only mechanism for discovering what rules were actually applied.
:Examples: Sample files are available for various test suites.
*)
BeginPackage["Goedel"]
and::usage = "and[x,y,...] is conjunction"
assert::usage = "assert[p] produces a statement equivalent to p by applying Goedel's
algorithm to class[v,p]. Applying assert repeatedly sometimes simplifies a statement."
cart::usage = "cart[x,y] is the cartesian product of classes x and y."
class::usage = "class[x,p] applies Goedel's algorithm to the class of all sets x
satisfying the condition p. The variable x may be atomic, or of the form pair[u,y],
where u and y in turn can be pairs, etc."
complement::usage = "complement[x] is the class of all sets that do not belong to x"
composite::usage = "composite[x,y,...] composites of x,y, ..."
domain::usage = "domain[x] is the domain of x"
E::usage = "E is the membership relation"
equal::usage = "equal[x,y] is the statement that the classes x and y are equal."
exists::usage = "exists[x,y,...]p means there are sets x,y,... such that p"
FIRST::usage = "FIRST is the function which takes pair[x,y] to y"
forall::usage = "forall[x,y,...]p means that p holds for all sets x,y,..."
Id::usage = "Id is the identity relation"
Id[x]::usage = "Id[x] is the restriction of the identity relation to x"
image::usage = "image[x,y] is the image of the class y under x"
intersection::usage = "intersection[x,y,...] is the intersection of classes x,y,..."
inverse::usage = "the relation inverse[x] is the inverse of x"
LeftPairY::usage = "LeftPairY is the function that takes x to pair[Y,x]"
member::usage = "member[x,y] is the statement that x belongs to y"
not::usage = "not[p] represents the negation of p"
or::usage = "or[x,y,...] is the inclusive or"
P::usage = "the power class P[x] is the class of all subsets of x"
pair::usage = "pair[x,y] is the ordered pair of x and y"
range::usage = "range[x] is the range of x"
RightPairY::usage = "RightPairY is the function that takes x to pair[x,Y]"
S::usage = "S is the subset relation"
SECOND::usage = "SECOND is the function that maps pair[x,y] to y"
singleton::usage = "singleton[x] has no member except x; it is 0 if x is not a set"
subclass::usage = "subclass[x,y] is the statement that x is contained in y"
U::usage = "the sup class U[x] is the union of all sets belonging to x"
union::usage = "union[x,y,...] is the union of the classes x,y,..."
V::usage = "the universal class"

Begin['Private'] (* begin the private context *)
(* definitions of auxiliary functions not exported *)
val listu[u_] := {u} // AtomQ[u]
val list[pair[u,v]] := Union[list[u],list[v]]
(* Is the expression x free of all variables which occur in y? *)
allsafeQ[x_,y_] := Apply[And,Map[FreeQ[x, #] && list[y]]]
(* definitions of exported functions *)
(* Rules that must be assigned before attributes are set. *)
and[p_] := p
or[p_] := p
Attributes[and] := {Flat, Orderless, OneIdentity}
Attributes[or] := {Flat, Orderless, OneIdentity}
composite[u_] := composite[Id, x]
intersection[u_] := x
union[u_] := x
Attributes[composite] := {Flat, OneIdentity}
Attributes[intersection] := {Flat, Orderless, OneIdentity}
Attributes[union] := {Flat, Orderless, OneIdentity}
not [True] := False (* Truth Table *)
not [False] := True

(* abbreviation for multiple quantifiers *)
exists [x, y, _, P] := exists [x, exists [y, P]]

(* elimination rule for universal quantifiers *)
forall [x, y, _] := not [exists [x, not [y]]]

(* basic rules for membership *)
member [u, 0] := False

(* Added to avoid assuming axiom of regularity. Goedel assumes member [x, 0] = 0. *)
class [u, member [x, _]] :=
  Module [{y, Unique []},
  class [v, exists [y, and [member [x, y], equal [x, y]]]]]
class [z, member [u, cart [x, y, _]]] :=
  Module [{u, Unique []},
  v, Unique []},
  class [z, exists [v, w, and [equal [pair [u, v], w], member [u, z], member [v, y]]]]]

member [u, v, cart [x, y, _]] :=
  and [member [u, z], member [v, y]]

member [u, v, complement [x, _]] :=
  and [member [u, v], not [member [u, x]]]

class [u, v, composite [x, _]] :=
  Module [{t, Unique []},
  u, Unique []},
  class [v, exists [u, v, and [equal [pair [u, v], z], member [pair [u, v], y]]]]

member [u, v, cross [x, y, _]] :=
  Module [{u, Unique []},
  v2, Unique []},
  class [z, exists [u1, u2, v1, v2, and [equal [pair [u1, u2], pair [v1, v2]],
  member [pair [u1, v1], z], member [pair [u2, v2], y]]}]

(* Goedel's definition 1.5 *)
class [u, v, member [x, domain [z]]] :=
  Module [{v, Unique []},
  class [v, exists [v, x, and [member [v, u], member [pair [v, u], z]]]]]
class [z, member [u, r]] :=
  Module [{u, Unique []},
  v, Unique []},
  class [z, exists [u, v, and [equal [pair [u, v], z]]]]]
class [z, member [u, Fin]] :=
  Module [{u, Unique []},
  v, Unique []},
  class [z, exists [u, v, and [equal [pair [u, v], z]]]]]
class [z, member [u, id [x]]] :=
  Module [{u, Unique []},
  class [z, exists [u, x, and [member [u, x], equal [pair [u, u], z]]]]]
class [z, member [u, image [x, _]]] :=
  Module [{u, Unique []},
  class [z, exists [u, x, and [member [v, u], equal [pair [v, u], z]]]]]

member [u, intersection [x, y, _]] :=
  and [member [u, x], member [u, y]]
class [z, member [u, inverse [x, _]]] :=
  Module [{u, Unique []},
  v, Unique []},
  class [z, exists [u, x, and [equal [pair [u, v], z]]]]]
class [z, member [u, LeftPair]] :=
  Module [{u, Unique []},
  v, Unique []},
  class [z, exists [u, v, and [equal [pair [u, v], z]]]]]

member [z, P [x, y]] :=
  and [member [z, x], subclass [y]]
class [u, member [v, pair [x, y]]] :=
  Module [{z, Unique []},
  class [z, exists [x, y, z, and [equal [pair [x, y], z], member [v, z]]]]]
class [u, member [v, range [z]]] :=
  Module [{u, Unique []},
  class [z, exists [u, v, and [member [v, u], member [pair [v, u], z]]]]]
class [z, member [v, RightPair]] :=
  Module [{u, Unique []},
  class [z, exists [u, v, and [equal [pair [u, v], z], equal [pair [v, u], z]]]]]
class [\(v\), member \([x, S]\)] := Module[{u = Unique[], v = Unique[]}, class[v, exists[u, v, and(equal[pair[u, v], x], subclass[u, v])]]

class [\(v\), member \([x, SECOND]\)] := Module[{u = Unique[], v = Unique[]}, class[v, exists[u, v, equal[pair[u, v], v]]]]

member[u, singleton[x]] := and(equal[u, x], member[u, y]]

member[u, union[x, y]] := or[member[u, x], member[u, y]]

class[v, member \([x, \bigcup(u)]\)] := Module[{y = Unique[]}, class[v, exists[y, and(member[x, y], member[y, z])]]]

class[v, subclass \([x, y]\)] := Module[{u = Unique[]}, class[v, and(exists[u, v, forall[u, or(not[member[u, x]], member[u, y])]]]]

(* translation of the rules in Goedel's 1939 monograph, pages 9-11 *)

class [\(x\), False] := 0

class [\(x\), True] := 0 /; AtomQ[x]

class [pair[u, v], True] := cart[class[u, True], class[v, True]]

class [u, member \([u, x]\)] := x /; And[AtomQ[u], AtomQ[x]]

(* special maneuver on top of page 10 of Goedel's monograph *)

class [u, member \([x, y]\)] := Module[{v = Unique[]},

class[u, exists[v, and(equal[x, v], member[v, y])]] /; FreQu[variablist[u, v]]

(* axiom B.1 membership relation *)

class [pair[u, v], member \([u, v]\)] := E /; And[AtomQ[u], AtomQ[v]]

(* axiom B.2 intersection *)

class [\(x\), and[p, q]] := intersection[class[x, p], class[x, q]]

class [\(x\), or[p, q]] := union[class[x, p], class[x, q]]

(* axiom B.3 complement *)

class [\(x\), not[p]] := intersection[complement[class[x, p]], class[x, True]]

(* axiom B.4 domain and Goedel's equation 2.6 on page 9 *)

class [\(x\), exists[p, y]] := domain[class[pair[x, y], p]]

(* axiom B.5 cartesian product *)

(* an interpretation of Goedel's equation 2.41 on page 9 *)

class [pair[u, v, p], := pair[cart[u, p], class[v, True]] /; allfree[p, v]

(* an interpretation of Goedel's equation 2.7 on page 9 *)

class [pair[u, v, p], := pair[cart[u, True], class[v, p]] /; allfree[p, u]

(* axiom B.6 inverse *)

class [pair[u, v, p], member \([u, v]\)] := inverse[E] /; And[AtomQ[u], AtomQ[v]]

(* Four rules to replace the rotation rules on Goedel's page 9: *)

class [pair[u, v, v, p], := composite[cart[pair[v, v], p], SECOND, id[cart[u, True, V]]] /; allfree[p, u]

class [pair[u, v, v, p], := composite[cart[pair[v, v], p], FIRST, id[cart[V, class[v, True]]]] /; allfree[p, v]

class [pair[u, v, v, p], := composite[id[cart[class[u, True], V]], inverse[SECOND], class[pair[v, v], p]] /; allfree[p, u]

class [pair[u, v, v, p], := composite[id[cart[V, class[v, True]]], inverse[FIRST], class[pair[v, u], p]] /; allfree[p, v]
(* new rules for equality *)

equal [u, u] := True

class [pair [u, v], equal [u, v]] := Id /; And[AtomQ [u], AtomQ [v]]
class [pair [u, v], equal [v, u]] := Id /; And[AtomQ [u], AtomQ [v]]
class [u, equal [y, y]] := intersection [singleton [y], class [x, True]] /; allfree [y, x]
class [x, equal [y, x]] := intersection [singleton [y], class [x, True]] /; allfree [y, x]

(* Goodall's Axiom 3 of Coextension. *)
class [u, equal [x, y]] :=
  intersection [class [v, subclass [x, y]], class [v, subclass [y, x]]] /;
  And[Or[Not[MemberQ [varlist [v, u]], Not[MemberQ [varlist [v, y]],
  Not[MemberQ [Head [x], pair]], Not[MemberQ [Head [y], pair]]]]

equal [pair [x, y], 0] := False
equal [pair [x, y], v] := False

(* equality of pairs *)
equal [pair [u, v], pair [x, y]] :=
  and[equal [singleton [u], singleton [x]],
  equal [singleton [v], singleton [y]]]

class [u, equal [pair [x, y], pair [x, y]]] := class [v, equal [u, v]] /;
  MemberQ [varlist [v, u]] || MemberQ [varlist [v, v]] || member [u, V] ||
  member [v, V]

(* flip equations involving a single pair to put pair on the left *)
equal [pair [x, y], v] := equal [y, x]

(* rules that apply when x or y is known not to be a set *)
pair [x, y] := pair [x, y] /; Not[MemberQ [varlist [x], x]] &
  Not[MemberQ [varlist [y], y]]

(* rule that applies when x does not occur in varlist [u] or when x occurs in x or y. *)
class [u, equal [pair [x, y], pair [x, y]]] :=
  Module [{xUnique = Unique []},
  class [v, exist [v, equal [pair [x, y], equal [v, v]]]] /;
  Not[MemberQ [varlist [u, x]], Not[FreeQ [x, y, z]]]

(* rule that applies when x does occur in varlist [v] and z does not occur in either x or y. *)
This rule only applies when x and y are known to be sets. *)
class [u, equal [pair [x, y], pair [x, y]]] := Module [{vUnique = Unique []},
  class [v, exist [v, equal [pair [x, y], equal [v, v]]]] /;
  And[MemberQ [varlist [v, x]], Not[member [x, V]]]

(* rule that applies when x does not know whether or not x is a set *)
class [u, equal [pair [x, y], pair [x, y]]] := Module [{vUnique = Unique []},
  class [v, exist [v, equal [pair [x, y], equal [v, v]]]] /;
  Not[MemberQ [varlist [u, x]], Not[FreeQ [x, y, z]]]

(* rule that applies when x does not know whether or not y is a set *)
class [u, equal [pair [x, y], pair [x, y]]] := Module [{xUnique = Unique []},
  class [v, exist [v, equal [pair [x, y], equal [v, v]]]] /;
  Not[MemberQ [varlist [u, y]], Not[FreeQ [x, y, z]]]

(* Rules for the functions LeftPairV and RightPairV *)
class [pair [u, v], equal [pair [v, u], v]] := LeftPairV
class [pair [u, v], equal [pair [v, u], v]] := RightPairV
class [pair [u, v], equal [pair [v, u], v]] := inverse [LeftPairV]
class [pair [u, v], equal [pair [v, u], v]] := inverse [RightPairV]

image [inverse [RightPairV], x] := 0 /; composite [id, x] == x
image [inverse [LeftPairV], x] := 0 /; composite [id, x] == x
class \(v\), equal [pair [\(v, y\), z]] := Module[v = Unique[]],
class [v, or and [not [member [y, V], equal [pair [v, y], z]]],
and [member [y, V], exists [v and [equal [pair [v, y], z], equal [v, y]]]]],
Not [allfree [y, z]]

class \(v\), equal [pair [z, V], z]] := Module[v = Unique[]],
class [v, or and [not [member [z, V], equal [pair [v, z], z]]],
and [member [z, V], exists [v and [equal [pair [v, z], z], equal [v, z]]]]],
Not [allfree [z, v]]

class \(v\), equal [pair [V, V], z]] := Module[v = Unique[]],
class [v, exists [v and [equal [pair [V, V], v], equal [v, z]]]]
Not [Member4 [\(v\), varlist [v]]]

(* assertions *)
assert [\(p\)] := Module[v = Unique[]], equal [V, class [v, p]]

(* a small sample of the thousands of simplification rules *)
cart \([x, 0]\) := 0
cart \([0, x]\) := 0
complement \([0]\) := V
complement [complement \([x]\)] := x
complement [union \([x, y]\)] := intersection [complement \([x]\), complement \([y]\)]
complement \([V]\) := 0

composite \([x, cart \([y, z]\)]\) := cart [\(y, image \([x, z]\)]
composite \([cart \([x, y]\), z]\) := cart \([image \([inverse \([x, z]\), y]\)]
composite \([Id, x, y]\) := composite \([x, y]\)
composite \([x, Id]\) := composite \([x, Id]\)
composite \([Id, Id]\) := Id

domain [cart \([x, y]\)] := intersection \([x, image \([V, y]\)]
domain [composite \([x, y]\)] := image \([inverse \([y]\), domain \([x]\)]
domain [Id] := V
domain [Id, x] := x
Id [\(V\)] := Id

image \([0, x]\) := 0
image \([x, 0]\) := 0
image [composite \([x, y]\), z] := image \([x, image \([y, z]\)]
image \([Id, x]\) := x
image \([id, x]\) := intersection \([x, y]\)
intersection [cart \([x, y]\), z] := composite \([id, y], z, id [x]\)
intersection \([V, x]\) := x

inverse \([0]\) := 0
inverse [cart \([x, y]\)] := cart \([y, x]\)
inverse [composite \([x]\)] := composite \([id, complement \([inverse \([x]\)]\)]
inverse [composite \([x, y]\)] := composite \([inverse \([y]\), inverse \([x]\)]
inverse [Id] := Id
inverse [inverse [\(x\)] := composite \([id, x]\)
range [Id] := V

union \([0, x]\) := x
End[ ]

(* and the private context *)
Protect [and, assert, cart, class, complement, composite, domain, E, equal, exists, FIRST, forall, Id, id, image, intersection, inverse, LeftPair, member, not, or, P, pair, range, RightPair, S, SECOND, singleton, subclass, U, union, V]
EndPackage[ ]

(* and the package context *)
References


