# case[p]

Johan G. F. Belinfante
2011 August 22

*In[1]:=* **SetDirectory["l:"]; << goedel.11aug18a**

```
:Package Title: goedel.11aug18a              2011 August 18 at 10:20 a.m.

Loading takes about twelve minutes, half that time due to builtin pauses.

It is now:  2011 Aug 22 at 11:8

Loading Simplification Rules

TOOLS.M is now incorporated in the GOEDEL program as of 2010 September 3

weightlimit = 40

Loading completed.

It is now:  2011 Aug 22 at 11:21
```

## summary

Although **reify** rules are often more efficient than **class** rules for eliminating variables, existing **reify** rules apply only to class expressions and not to statements about classes. To get around this limitation, a new constructor **case[p]** is introduced that passively assigns a class to any simple or compound statement **p** without immediately invoking **class** rules. One can use **Normality** to convert **case[p]** expressions to ones involving **image[V, x]**. Rewrite rules are derived to make this conversion automatic for statements involving a single positive literal, and to reduce **reify** for compound statements to those for the individual literals in that statement.

## definition

Definition.

*In[2]:=* **member[x_, case[p_]] := and[member[x, V], p]**

Theorem. An example showing the use of **Normality** to convert the constructor **case** to other class constructors.

*In[3]:=* **case[True] // Normality**

*Out[3]=* case[True] == V

*In[4]:=* **case[True] := V**

Theorem. A basic rule for extracting the statement **p** from the class **case[p]**.

```
In[5]:= equiv[equal[V, case[p]], p] // assert
Out[5]= True
In[6]:= equal[V, case[p_]] := p
```

## a normalization rule

The following rewrite rule prevents normalization tests from destroying the expression **case[p]** when **p** is a variable representing an unspecified statement.

Theorem.

```
In[7]:= case[p] // Normality
Out[7]= case[p] == class[$5, p]
In[8]:= class[x_, p_] := case[p] /; AtomQ[x] && FreeQ[p, x]
```

## complementation rules

Theorem. A rule for the complement of **case[p]**.

```
In[9]:= equal[case[not[p]], complement[case[p]]] // AssertTest
Out[9]= equal[case[not[p]], complement[case[p]]] == True
In[10]:= complement[case[p_]] := case[not[p]]
```

Theorem. An example showing how **case** rules can be derived by double complementation.

```
In[11]:= case[False] // DoubleComplement
Out[11]= case[False] == 0
In[12]:= case[False] := 0
```

Theorem.

```
In[13]:= SubstTest[equal, V, complement[t], t → case[p]]
Out[13]= equal[0, case[p]] == not[p]
In[14]:= equal[0, case[p_]] := not[p]
```

## information statement

An observation. The expression **case[p]** is equal to either **0** or **V**.

```
In[15]:= or[equal[0, case[p]], equal[V, case[p]]]

Out[15]= True
```

An information statement.

```
In[16]:= case::usage= "case[p] is V if p is true, and otherwise is empty"

Out[16]= case[p] is V if p is true, and otherwise is empty
```

## unions and intersections

The rewrite rules for unions and intersections will be oriented to produce case constructs involving compound statements, with the idea that those statements might be simplified automatically by existing rewrite rules.

Theorem. Eliminating unions of cases.

```
In[17]:= equal[union[case[p], case[q]], case[or[p, q]]] // AssertTest

Out[17]= equal[case[or[p, q]], union[case[p], case[q]]] == True

In[18]:= union[case[p_], case[q_]] := case[or[p, q]]
```

The rule for intersections of cases could also be derived in a similar way using **AssertTest,** but here instead it is derived using double complementation.

Corollary. Eliminating intersections of cases.

```
In[19]:= intersection[case[p], case[q]] // DoubleComplement

Out[19]= intersection[case[p], case[q]] == case[and[p, q]]

In[20]:= intersection[case[p_], case[q_]] := case[and[p, q]]
```

## reify rules for compound statements

In this section rewrite rules are derived that reduce **reify** expressions for **case[p]** to the special case that **p** is a single positive literal.

Theorem. Rule for negations.

```
In[21]:= SubstTest[reify, x, complement[f[x]], f[x] → case[p[x]]] // Reverse

Out[21]= reify[x, case[not[p[x]]]] == composite[Id, complement[reify[x, case[p[x]]]]]

In[22]:= reify[x_, case[not[p_]]] := composite[Id, complement[reify[x, case[p]]]]
```

Theorem. Rule for disjunctions.

```
In[23]:= SubstTest[reify, x, union[f[x], g[x]], {f[x] → case[p[x]], g[x] → case[q[x]]}] // Reverse

Out[23]= reify[x, case[or[p[x], q[x]]]] == union[reify[x, case[p[x]]], reify[x, case[q[x]]]]

In[24]:= reify[x_, case[or[p_, q_]]] := union[reify[x, case[p]], reify[x, case[q]]]
```

Theorem. Rule for conjunctions.

```
In[25]:= SubstTest[reify, x, intersection[f[x], g[x]],
           {f[x] → case[p[x]], g[x] → case[q[x]]}] // Reverse

Out[25]= reify[x, case[and[p[x], q[x]]]] ==
          intersection[reify[x, case[p[x]]], reify[x, case[q[x]]]]

In[26]:= reify[x_, case[and[p_, q_]]] := intersection[reify[x, case[p]], reify[x, case[q]]]
```

## image[V, – ] expressions

To reduce the number of **reify** rules needed for **case** expressions, it is helpful to automatically convert **case[p]** expressions involving a single positive literal to expressions involving **image[V, x]** whenever this is not too difficult. This is done for literals involving the binary predicates **member**, **subclass** and **equal**.

Theorem.

```
In[27]:= case[member[x, y]] // Normality

Out[27]= case[member[x, y]] == image[V, intersection[y, set[x]]]

In[28]:= case[member[x_, y_]] := image[V, intersection[y, set[x]]]
```

Theorem.

```
In[29]:= case[subclass[x, y]] // Normality

Out[29]= case[subclass[x, y]] == complement[image[V, intersection[x, complement[y]]]]

In[30]:= case[subclass[x_, y_]] := complement[image[V, intersection[x, complement[y]]]]
```

Theorem.

```
In[31]:= SubstTest[intersection, case[p], case[q], {p → subclass[x, y], q → subclass[y, x]}]

Out[31]= case[equal[x, y]] == intersection[complement[image[V, intersection[x, complement[y]]]],
           complement[image[V, intersection[y, complement[x]]]]]

In[32]:= case[equal[x_, y_]] :=
          intersection[complement[image[V, intersection[x, complement[y]]]],
           complement[image[V, intersection[y, complement[x]]]]]
```