e[x]

Johan G. F. Belinfante
2008 December 28

```
In[1]:= SetDirectory["l:"]; << goedel.08dec26a;<< tools.m

        :Package Title: goedel.08dec26a         2008 December 26 at 6:45 a.m.

        It is now:  2008 Dec 28 at 8:46

        Loading Simplification Rules

        TOOLS.M                                      Revised 2008 December 26

        weightlimit = 40
```

summary

The constructor **e[x]** is defined for any class **x**, but is mainly of interest for binary operations. The constructor **e[x]** is the unique neutral element of a binary operation if there is one. If a binary operation **x** has no neutral element, then **e[x] = V**.

normalization

The constructor **e[x]** is defined by a **class**-wrapped membership rule:

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

In[3]:= FirstMatch[class[t_, member[u_, HoldPattern[e[x_]]]]]

Out[3]= class[t_, member[x_, e[y_]]] := Module[{w = Unique[]},
          class[t, forall[w, implies[member[w, ids[y]], member[x, w]]]]]
```

Theorem. (Normalization.)

```
In[4]:= e[x] // Normality // Reverse

Out[4]= A[ids[x]] == e[x]

In[5]:= A[ids[x_]] := e[x]
```

a general lemma

The results in this section are valid for any class **x**. Some care has been taken in formulating these results that they do not result in looping when the variable is replaced by **binop[x]**.

Theorem.

```
In[18]:= Map[implies[empty[ids[x]], #] &, SubstTest[equal, V, A[t], t → ids[x]]] // Reverse
Out[18]= or[equal[V, e[x]], not[equal[0, ids[x]]]] == True
```

```
In[19]:= or[equal[V, e[x_]], not[equal[0, ids[x_]]]] := True
```

Theorem.

```
In[21]:= Map[implies[#, empty[ids[x]]] &, SubstTest[equal, V, A[t], t → ids[x]]] // Reverse
Out[21]= or[equal[0, ids[x]], not[equal[V, e[x]]]] == True
```

```
In[22]:= or[equal[0, ids[x_]], not[equal[V, e[x_]]]] := True
```

Theorem.

```
In[23]:= Map[or[empty[ids[x]], #] &, SubstTest[member, A[t], V, t → ids[x]]] // Reverse
Out[23]= or[equal[0, ids[x]], member[e[x], V]] == True
```

```
In[24]:= or[equal[0, ids[x_]], member[e[x_], V]] := True
```

Theorem.

```
In[27]:= Map[or[not[#], not[empty[ids[x]]]] &, SubstTest[member, A[t], V, t → ids[x]]] // Reverse
Out[27]= or[not[equal[0, ids[x]]], not[member[e[x], V]]] == True
```

```
In[28]:= or[not[equal[0, ids[x_]]], not[member[e[x_], V]]] := True
```

Corollary. Either **e[x]** is a set, or else **e[x]** equals **V**.

```
In[29]:= Map[not, SubstTest[and, implies[p1, p2], implies[p2, p3], not[implies[p1, p3]],
           {p1 → not[equal[V, e[x]]], p2 → not[empty[ids[x]]], p3 → member[e[x], V]}]] // Reverse
Out[29]= or[equal[V, e[x]], member[e[x], V]] == True
```

```
In[30]:= or[equal[V, e[x_]], member[e[x_], V]] := True
```

binary operations

Lemma. (Uniqueness of identities for a binary operation.)

```
In[6]:= Map[equal[#, cartsq[ids[binop[x]]]] &,
          SubstTest[composite, id[ids[t]], domain[t], id[ids[t]], t → binop[x]]]

Out[6]= or[equal[0, ids[binop[x]]], member[ids[binop[x]], range[SINGLETON]]] == True

In[7]:= (% /. x → x_) /. Equal → SetDelayed
```

Theorem.

```
In[8]:= SubstTest[equal, t, set[A[t]], t → ids[binop[x]]] // Reverse

Out[8]= equal[ids[binop[x]], set[e[binop[x]]]] == True

In[9]:= ids[binop[x_]] := set[e[binop[x]]]
```

Corollary. (Wrapper-free restatement of the lemma.)

```
In[11]:= SubstTest[implies, equal[x, binop[t]],
           or[equal[0, ids[x]], member[ids[x], range[SINGLETON]]], t → x] // Reverse

Out[11]= or[equal[0, ids[x]], member[ids[x], range[SINGLETON]], not[member[x, BINOPS]]] == True

In[12]:= or[equal[0, ids[x_]],
           member[ids[x_], range[SINGLETON]], not[member[x_, BINOPS]]] := True
```

Corollary. (Wrapper-free restatement of the theorem.)

```
In[13]:= SubstTest[implies, equal[x, binop[t]], equal[ids[x], set[e[x]]], t → x] // Reverse

Out[13]= or[equal[ids[x], set[e[x]]], not[member[x, BINOPS]]] == True

In[14]:= or[equal[ids[x_], set[e[x_]]], not[member[x_, BINOPS]]] := True
```

Corollary. (Uniqueness of identities for binary operations.)

```
In[16]:= Map[not, SubstTest[and, implies[p1, p3], implies[and[p2, p3], p4],
            not[implies[and[p1, p2], p4]], {p1 -> member[x, BINOPS], p2 → member[u, ids[x]],
             p3 -> equal[ids[x], set[e[x]]], p4 → equal[u, e[x]]}]] // Reverse

Out[16]= or[equal[u, e[x]], not[member[u, ids[x]]], not[member[x, BINOPS]]] == True

In[17]:= or[equal[u_, e[x_]], not[member[u_, ids[x_]]], not[member[x_, BINOPS]]] := True
```

Corollary. If a binary operation **x** has a neutral element, then **e[x]** is a neutral element.

```
In[34]:= Map[not, SubstTest[and, implies[p1, p2], implies[p2, p3],
            not[implies[p1, p3]], {p1 → member[x, BINOPS], p2 → equal[ids[x], set[e[x]]],
             p3 → or[member[e[x], ids[x]], equal[0, ids[x]]]}]] // Reverse

Out[34]= or[equal[0, ids[x]], member[e[x], ids[x]], not[member[x, BINOPS]]] == True

In[35]:= or[equal[0, ids[x_]], member[e[x_], ids[x_]], not[member[x_, BINOPS]]] := True
```

Corollary. If **x** is a binary operation, then either **e[x]** is a neutral element, or **e[x] = V**.

```
In[37]:= Map[not, SubstTest[and, implies[p1, p2], implies[p2, p3],
           not[implies[p1, p3]], {p1 → and[member[x, BINOPS], not[member[e[x], ids[x]]]],
            p2 → equal[0, ids[x]], p3 → equal[e[x], V]}]] // Reverse

Out[37]= or[equal[V, e[x]], member[e[x], ids[x]], not[member[x, BINOPS]]] == True

In[38]:= or[equal[V, e[x_]], member[e[x_], ids[x_]], not[member[x_, BINOPS]]] := True
```

Corollary.

```
In[43]:= Map[not, SubstTest[and, implies[p1, or[p2, p3]],
           implies[p2, p4], implies[p3, p4], not[implies[p1, p4]],
           {p1 → member[x, BINOPS], p2 -> equal[V, e[x]], p3 -> member[e[x], ids[x]],
            p4 → subclass[composite[x, RIGHT[e[x]]], Id]}]] // Reverse

Out[43]= or[not[member[x, BINOPS]], subclass[composite[x, RIGHT[e[x]]], Id]] == True

In[44]:= or[not[member[x_, BINOPS]], subclass[composite[x_, RIGHT[e[x_]]], Id]] := True
```

Corollary.

```
In[45]:= Map[not, SubstTest[and, implies[p1, or[p2, p3]], implies[p2, p4], implies[p3, p4],
           not[implies[p1, p4]], {p1 → member[x, BINOPS], p2 -> equal[V, e[x]],
            p3 -> member[e[x], ids[x]], p4 → subclass[composite[x, LEFT[e[x]]], Id]}]] // Reverse

Out[45]= or[not[member[x, BINOPS]], subclass[composite[x, LEFT[e[x]]], Id]] == True

In[46]:= or[not[member[x_, BINOPS]], subclass[composite[x_, LEFT[e[x_]]], Id]] := True
```

the empty binary operation

Theorem. The empty binary operation has no neutral element.

```
In[48]:= e[0] // Normality

Out[48]= e[0] == V

In[49]:= e[0] := V
```

examples in arithmetic

Theorem. (The number **0** is neutral for addition of natural numbers.)

```
In[50]:= e[NATADD] // Normality

Out[50]= e[NATADD] == 0

In[51]:= e[NATADD] := 0
```

Theorem. (The number **1 = set[0]** is neutral for multiplication of natural numbers.)

```
In[52]:= e[NATMUL] // Normality
Out[52]= e[NATMUL] == set[0]
In[53]:= e[NATMUL] := set[0]
```

Theorem. (The integer **plus[0] = id[omega]** is neutral for addition of integers.)

```
In[54]:= e[INTADD] // Normality
Out[54]= e[INTADD] == id[omega]
In[55]:= e[INTADD] := id[omega]
```

Theorem. (The integer **plus[set[0]] = composite[id[omega], SUCC]** is neutral for multiplication of integers.)

```
In[56]:= e[INTMUL] // Normality
Out[56]= e[INTMUL] == composite[id[omega], SUCC]
In[57]:= e[INTMUL] := composite[id[omega], SUCC]
```

examples in Boolean algebra

Theorem. The empty set is neutral for **CUP**.

```
In[58]:= e[CUP] // Normality
Out[58]= e[CUP] == 0
In[59]:= e[CUP] := 0
```

Theorem. The empty set is neutral for **SYMDIF**.

```
In[62]:= e[SYMDIF] // Normality
Out[62]= e[SYMDIF] == 0
In[63]:= e[SYMDIF] := 0
```

The empty string is neutral for concatenation of strings.

```
In[74]:= e[JOIN] // Normality
Out[74]= e[JOIN] == 0
In[75]:= e[JOIN] := 0
```

associative functions without neutral elements

Theorem. The function **CAP** has no neutral elements.

```
In[60]:= e[CAP] // Normality
Out[60]= e[CAP] == V
```

```
In[61]:= e[CAP] := V
```

Theorem. The function **COMPOSE** has no neutral elements.

```
In[67]:= e[COMPOSE] // Normality
Out[67]= e[COMPOSE] == V
```

```
In[68]:= e[COMPOSE] := V
```

Theorem. The function **FIRST** has no neutral elements.

```
In[69]:= e[FIRST] // Normality
Out[69]= e[FIRST] == V
```

```
In[70]:= e[FIRST] := V
```

Theorem. The function **SECOND** has no neutral elements.

```
In[71]:= e[SECOND] // Normality
Out[71]= e[SECOND] == V
```

```
In[72]:= e[SECOND] := V
```