pigeonhole principle for unary operations

Johan G. F. Belinfante
2012 March 2

```
In[1]:= SetDirectory["l:"]; << goedel.12mar01a
```

```
:Package Title: goedel.12mar01a                 2012 March 1 at 10:00 p.m.

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

It is now:  2012 Mar 2 at 6:41

Loading Simplification Rules

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

weightlimit = 40

Loading completed.

It is now:  2012 Mar 2 at 6:54
```

summary

A **unary operation** is a small function whose range is a subset of its domain. (A class is **small** if it is a set.) For short, it will be said that a unary operation is **onto** if its domain and range are equal. A **permutation** is a small one-to-one function whose domain and range are equal. The pigeon hole principle for unary operations is the statement that a finite unary operation is one-to-one if and only if it is onto. A variable-free form of this is already available in the **GOEDEL** program. In this notebook, additional forms are derived with variables, with and without **unop** and **fin** wrappers.

general results

In this section some general results are derived that do not require the finiteness hypothesis.

Theorem. Simplification rule.

```
In[9]:= equiv[and[member[x, UNOPS], subclass[x, cart[V, V]]], member[x, UNOPS]]
Out[9]= True
```

```
In[11]:= and[member[x_, UNOPS], subclass[x_, cart[V, V]]] := member[x, UNOPS]
```

Observation. The statement that a unary operation is onto is currently rewritten in terms of mappings.

```
In[109]:=
        and[member[x, UNOPS], equal[domain[x], range[x]]]

Out[109]=
        member[x, map[range[x], V]]
```

Theorem. A permutation is an onto unary operation.

```
In[108]:=
        or[member[x, map[range[x], V]], not[member[x, PERMS]]] // AssertTest

Out[108]=
        or[member[x, map[range[x], V]], not[member[x, PERMS]]] == True

In[112]:=
        or[member[x_, map[range[x_], V]], not[member[x_, PERMS]]] := True
```

Theorem. A one-to-one and onto unary operation is a permutation.

```
In[33]:= Map[implies[and[#, subclass[range[x], y]], member[x, PERMS]] &,
           SubstTest[member, x, intersection[u, v],
            {u → intersection[image[INVERSE, FUNS], image[inverse[DORA], Id]], v → FUNS}]]

Out[33]= or[member[x, PERMS], not[FUNCTION[inverse[x]]], not[member[x, map[range[x], y]]]] == True

In[34]:= or[member[x_, PERMS], not[FUNCTION[inverse[x_]]],
            not[member[x_, map[range[x_], y_]]]] := True
```

Theorem. A simplification rule for **unop[x]**.

```
In[44]:= SubstTest[member, unop[x], intersection[u, v],
           {u → intersection[image[INVERSE, FUNS], image[inverse[DORA], Id]], v -> UNOPS}]

Out[44]= and[equal[domain[unop[x]], range[unop[x]]], FUNCTION[inverse[unop[x]]] ==
           member[unop[x], PERMS]

In[45]:= and[equal[domain[unop[x_]], range[unop[x_]]], FUNCTION[inverse[unop[x_]]] :=
           member[unop[x], PERMS]
```

---

the pigeonhole principle

Theorem. A finite onto unary operation is a permutation.

```
In[23]:= Map[implies[and[#, subclass[range[x], y]], member[x, PERMS]] &, SubstTest[member, x,
           intersection[u, v], {u → intersection[FINITE, image[inverse[DORA], Id]], v -> UNOPS}]]

Out[23]= or[member[x, PERMS], not[member[x, FINITE]], not[member[x, map[range[x], y]]]] == True

In[24]:= or[member[x_, PERMS], not[member[x_, FINITE]],
            not[member[x_, map[range[x_], y_]]]] := True
```

Theorem.

```
In[51]:= Map[implies[#, member[unop[x], PERMS]] &, SubstTest[member, unop[x], intersection[u, v],
           {u → intersection[FINITE, image[inverse[DORA], Id]], v -> UNOPS}]]

Out[51]= or[member[unop[x], PERMS],
           not[equal[domain[unop[x]], range[unop[x]]]], not[member[unop[x], FINITE]]] == True

In[52]:= or[member[unop[x_], PERMS],
           not[equal[domain[unop[x_]], range[unop[x_]]]], not[member[unop[x_], FINITE]]] := True
```

---

the double wrapper unop[fin[x]]

Theorem. (Double wrapper introduction rule.) A basic property of the double wrapper **unop[fin[x]]**.

```
In[61]:= SubstTest[implies, and[subclass[u, v], member[v, FINITE]],
           member[u, FINITE], {u → unop[fin[x]], v → fin[x]}] // Reverse

Out[61]= member[unop[fin[x]], FINITE] == True

In[62]:= member[unop[fin[x_]], FINITE] := True
```

Theorem. An onto finite unary operation is a permutation.

```
In[64]:= SubstTest[or, member[unop[t], PERMS], not[equal[domain[unop[t]], range[unop[t]]]],
           not[member[unop[t], FINITE]], t → unop[fin[x]]] // Reverse

Out[64]= or[member[unop[fin[x]], PERMS],
           not[equal[domain[unop[fin[x]]], range[unop[fin[x]]]]]] == True

In[65]:= or[member[unop[fin[x_]], PERMS],
           not[equal[domain[unop[fin[x_]]], range[unop[fin[x_]]]]]] := True
```

Corollary. An onto finite unary operation is one-to-one.

```
In[72]:= Map[not, SubstTest[and, implies[p1, p2], implies[p2, p3],
            not[implies[p1, p3]], {p1 -> equal[domain[unop[fin[x]]], range[unop[fin[x]]]],
             p2 → member[unop[fin[x]], PERMS], p3 → FUNCTION[inverse[unop[fin[x]]]]}]] // Reverse

Out[72]= or[FUNCTION[inverse[unop[fin[x]]]],
           not[equal[domain[unop[fin[x]]], range[unop[fin[x]]]]]] == True

In[73]:= or[FUNCTION[inverse[unop[fin[x_]]]],
           not[equal[domain[unop[fin[x_]]], range[unop[fin[x_]]]]]] := True
```

Theorem. An one-to-one finite unary operation is onto.

```
In[82]:= Map[not, SubstTest[and, implies[p1, p2], implies[p2, p3], not[implies[p1, p3]],
           {p1 → FUNCTION[inverse[unop[fin[x]]]], p2 → member[unop[fin[x]], PERMS],
            p3 -> equal[domain[unop[fin[x]]], range[unop[fin[x]]]]}]] // Reverse

Out[82]= or[equal[domain[unop[fin[x]]], range[unop[fin[x]]]],
           not[FUNCTION[inverse[unop[fin[x]]]]]] == True
```

```
In[83]:= or[equal[domain[unop[fin[x_]]], range[unop[fin[x_]]]],
            not[FUNCTION[inverse[unop[fin[x_]]]]] := True
```

wrapper-free rules

Theorem. A form of the pigeonhole principle. An onto finite unary operation is one-to-one.

```
In[75]:= SubstTest[implies, equal[x, unop[fin[t]]],
            or[FUNCTION[inverse[x]], not[equal[domain[x], range[x]]]], t → x] // Reverse

Out[75]= or[FUNCTION[inverse[x]], not[equal[domain[x], range[x]]],
            not[member[x, FINITE]], not[member[x, UNOPS]]] == True

In[77]:= or[FUNCTION[inverse[x_]], not[equal[domain[x_], range[x_]]],
            not[member[x_, FINITE]], not[member[x_, UNOPS]]] := True
```

Theorem. A form of the pigeonhole principle. A one-to-one finite unary operation is onto.

```
In[86]:= SubstTest[implies, equal[x, unop[fin[t]]],
            or[equal[domain[x], range[x]], not[FUNCTION[inverse[x]]]], t → x] // Reverse

Out[86]= or[equal[domain[x], range[x]], not[FUNCTION[inverse[x]]],
            not[member[x, FINITE]], not[member[x, UNOPS]]] == True

In[87]:= or[equal[domain[x_], range[x_]], not[FUNCTION[inverse[x_]]],
            not[member[x_, FINITE]], not[member[x_, UNOPS]]] := True
```

Theorem. A finite one-to-one unary operation is a permutation.

```
In[90]:= Map[implies[#, member[x, PERMS]] &, SubstTest[member, x, intersection[u, v],
            {u -> intersection[FINITE, UNOPS], v -> image[INVERSE, FUNS]}]]

Out[90]= or[member[x, PERMS], not[FUNCTION[inverse[x]]],
            not[member[x, FINITE]], not[member[x, UNOPS]]] == True

In[94]:= or[member[x_, PERMS], not[FUNCTION[inverse[x_]]],
            not[member[x_, FINITE]], not[member[x_, UNOPS]]] := True
```

counterexamples

In this section, counterexamples are provided to show that the finiteness conditions cannot be omitted in either of the pigeonhole principles for unary operations.

Theorem. An example.

```
In[98]:= member[composite[id[omega], SUCC], PERMS] // AssertTest

Out[98]= member[composite[id[omega], SUCC], PERMS] == False
```

```
In[99]:= member[composite[id[omega], SUCC], PERMS] := False
```

Counterexample. A onto-to-one unary operation need not be a permutation.

```
In[100]:=
        or[member[x, PERMS], not[FUNCTION[inverse[x]]], not[member[x, UNOPS]]] /.
         x → composite[id[omega], SUCC]

Out[100]=
        False
```

Corollary. The class of one-to-one unary operations is not a subclass of the class of permutations.

```
In[102]:=
        Map[not, SubstTest[implies, and[member[u, v], subclass[v, w]], member[u, w],
           {u -> composite[id[omega], SUCC], v -> intersection[BIJ, UNOPS], w → PERMS}]] // Reverse

Out[102]=
        subclass[intersection[BIJ, UNOPS], PERMS] == False

In[103]:=
        subclass[intersection[BIJ, UNOPS], PERMS] := False
```

Theorem. The restriction of **BIGCUP** to natural numbers is a unary operation. (This is a one-point extension of the predecessor function for natural numbers.)

```
In[117]:=
        member[composite[BIGCUP, id[omega]], UNOPS] // AssertTest

Out[117]=
        member[composite[BIGCUP, id[omega]], UNOPS] == True

In[159]:=
        member[composite[BIGCUP, id[omega]], UNOPS] := True
```

Theorem. The restriction of **BIGCUP** to natural numbers is not one-to-one.

```
In[158]:=
        (SubstTest[subclass, composite[t, inverse[t]], Id, t -> union[funpart[x], id[set[0]]]]) /.
         x -> composite[id[omega], SUCC]

Out[158]=
        FUNCTION[composite[id[omega], inverse[BIGCUP]]] == False

In[160]:=
        FUNCTION[composite[id[omega], inverse[BIGCUP]]] := False
```

Lemma. A simplification rule for mapping statements about **BIGCUP** ∘ **id[ω]**.

```
In[165]:=
        member[composite[BIGCUP, id[omega]], map[x, y]] // AssertTest

Out[165]=
        member[composite[BIGCUP, id[omega]], map[x, y]] ==
         and[equal[omega, x], subclass[omega, y]]
```

```
In[166]:=
member[composite[BIGCUP, id[omega]], map[x_, y_]] :=
 and[equal[omega, x], subclass[omega, y]]
```

Counterexample. An onto unary operation need not be one-to-one.

```
In[167]:=
implies[and[member[x, UNOPS], equal[domain[x], range[x]]], FUNCTION[inverse[x]]] /.
 x -> composite[BIGCUP, id[omega]]
Out[167]=
False
```

Theorem. The function **BIGCUP** ∘ **id[**ω**]** is not a permutation.

```
In[172]:=
member[composite[BIGCUP, id[omega]], PERMS] // AssertTest
Out[172]=
member[composite[BIGCUP, id[omega]], PERMS] == False

In[173]:=
member[composite[BIGCUP, id[omega]], PERMS] := False
```

Corollary. An onto unary operation need not be a permutation.

```
In[174]:=
Map[not, SubstTest[implies, and[member[u, v], subclass[v, w]],
   member[u, w], {u -> composite[BIGCUP, id[omega]],
    v -> intersection[image[inverse[DORA], Id], UNOPS], w → PERMS}]] // Reverse
Out[174]=
subclass[intersection[UNOPS, image[inverse[DORA], Id]], PERMS] == False

In[175]:=
subclass[intersection[UNOPS, image[inverse[DORA], Id]], PERMS] := False
```