

equality rule for power[x]

Johan G. F. Belinfante
2003 May 4

```
In[1]:= << goedel52.r60; << tools.m

:Package Title: goedel52.r60      2003 May 4 at 6:30 p.m.

It is now: 2003 May 5 at 9:22

Loading Simplification Rules

TOOLS.M                          Revised 2003 April 1

weightlimit = 40
```

■ summary

An equality rule for **power[x]** is derived. As an elementary application of this rewrite rule, a previously-derived unit clause involving **funpart** is used to derive a binary clause which says that powers of functions are functions.

■ deriving equality substitution of powers from that for iterates

The following lemma will be needed:

```
In[2]:= equal[cross[Id, x], cross[Id, y]] // AssertTest

Out[2]= equal[cross[Id, x], cross[Id, y]] == equal[composite[Id, x], composite[Id, y]]

In[3]:= equal[cross[Id, x_], cross[Id, y_]] := equal[composite[Id, x], composite[Id, y]]
```

The starting point of the derivation of the equality rule for **power[x]** is this formula expressing **power** in terms of **iterate**:

```
In[4]:= iterate[cross[Id, x], Id]

Out[4]= power[x]
```

An equality substitution law for **iterate** implies a corresponding equality substitution law for **power**:

```
In[5]:= SubstTest[implies, equal[u, v], equal[iterate[u, w], iterate[v, w]],
  {u -> cross[Id, x], v -> cross[Id, y], w -> Id}]

Out[5]= or[equal[power[x], power[y]], not[equal[composite[Id, x], composite[Id, y]]]] == True

In[6]:= or[equal[power[x_], power[y_]], not[equal[composite[Id, x_], composite[Id, y_]]]] := True
```

The converse implication also holds:

```
In[7]:= SubstTest[implies, equal[u, v], equal[image[u, w], image[v, w]],
  {u -> power[x], v -> power[y], w -> singleton[singleton[0]]}]
```

```
Out[7]= or[equal[composite[Id, x], composite[Id, y]], not[equal[power[x], power[y]]]] == True
```

```
In[8]:= or[equal[composite[Id, x_], composite[Id, y_]], not[equal[power[x_], power[y_]]]] := True
```

That is:

```
In[9]:= equiv[equal[power[x], power[y]], equal[composite[Id, x], composite[Id, y]]]
```

```
Out[9]= True
```

This justifies adding the following rewrite rule:

```
In[10]:= equal[power[x_], power[y_]] := equal[composite[Id, x], composite[Id, y]]
```

■ an application to powers of functions

As an elementary application, a binary clause is derived from the following fact, eliminating the **funpart** wrapper.

```
In[11]:= FUNCTION[image[power[funpart[x]], singleton[y]]]
```

```
Out[11]= True
```

The first step is to relate the wrapper **funpart** to the predicate **FUNCTION**:

```
In[12]:= SubstTest[implies, equal[u, v], equal[image[u, w], image[v, w]],
  {u -> power[x], v -> power[funpart[x]], w -> singleton[y]}]
```

```
Out[12]= or[equal[image[power[x], singleton[y]], image[power[funpart[x]], singleton[y]]],
  not[FUNCTION[composite[Id, x]]]] == True
```

```
In[13]:= or[equal[image[power[x_], singleton[y_]], image[power[funpart[x_]], singleton[y_]]],
  not[FUNCTION[composite[Id, x_]]]] := True
```

The next step uses equality substitution for the predicate **FUNCTION**:

```
In[14]:= SubstTest[implies, and[equal[u, v], FUNCTION[u]], FUNCTION[v],
  {u -> image[power[funpart[x]], singleton[y]], v -> image[power[x], singleton[y]]}]
```

```
Out[14]= or[FUNCTION[image[power[x], singleton[y]]], not[equal[
  image[power[x], singleton[y]], image[power[funpart[x]], singleton[y]]]]] == True
```

```
In[15]:= or[FUNCTION[image[power[x_], singleton[y_]]],
  not[equal[image[power[x_], singleton[y_]],
  image[power[funpart[x_]], singleton[y_]]]] := True
```

The final step uses a chain of implications.

```
In[16]:= Map[not, SubstTest[and, implies[p1, p2],
  implies[p2, p3], implies[p3, p4], not[implies[p1, p4]],
  {p1 -> FUNCTION[x], p2 -> FUNCTION[composite[Id, x]],
  p3 ->
  equal[image[power[x], singleton[y]], image[power[funpart[x]], singleton[y]]],
  p4 -> FUNCTION[image[power[x], singleton[y]]]}]]]
```

```
Out[16]= or[FUNCTION[image[power[x], singleton[y]]], not[FUNCTION[x]]] == True
```

```
In[17]:= or[FUNCTION[image[power[x_], singleton[y_]]], not[FUNCTION[x_]]] := True
```

This clause says that if x is a function, then the y -th power of x is also a function. One does not need to add a literal requiring that y be a natural number because if this is not the case, the y -th power of x is the empty set. Since the empty set is a function, the clause holds whether or not y is a natural number.

■ eliminating the variable y

The variable y in the rule derived in the preceding section can be eliminated as follows:

```
In[18]:= SubstTest[class, y,
  implies[FUNCTION[x], FUNCTION[image[z, singleton[y]]], z → power[x]]
```

```
Out[18]= V == union[complement[fix[composite[inverse[power[x]], cross[Id, Di], power[x]]],
  image[V, fix[composite[x, inverse[x], Di]]],
  image[V, intersection[x, complement[cart[V, V]]]]]
```

```
In[19]:= Map[equal[V, #] &, %]
```

```
Out[19]= True == or[FUNCTION[rotate[inverse[power[x]]]], not[FUNCTION[x]]]
```

```
In[21]:= or[FUNCTION[rotate[inverse[power[x_]]]], not[FUNCTION[x_]]] := True
```

The following unit clause version of this result had been derived on a previous occasion:

```
In[24]:= FUNCTION[rotate[inverse[power[funpart[x]]]]]
```

```
Out[24]= True
```

■ comments

The use of **funpart** to eliminate the **FUNCTION** predicate is a useful technique for reducing the number of literals in a clause, but one needs to know how to go back when need be. This notebook illustrates the general technique needed to do this for a simple example. The **GOEDEL** program has less built-in support for equality substitution than does **Otter**. Accordingly, when **Otter** is used, one would expect that it should be much easier to go back and forth between clauses involving a **FUNCTION** literal and corresponding clauses involving **funpart**.

■ acknowledgement

The author originally learned about the usefulness of the concept of **funpart** for eliminating **FUNCTION** predicates in the course of studying a paper by Formisano and Omodeo. Their MAP formalism is quite different from Gödel's class theory, but many of their techniques also apply here.