iterate and functions

Johan G. F. Belinfante
2002 May 20; recreated 2003 May 22

```
In[1]:= << goedel52.n87; << tools.m

        :Package Title: GOEDEL52.n87           2002 May 20 at 2:45 p.m.

        It is now:  2003 May 22 at 10:17

        Loading Simplification Rules

        TOOLS.M                       Revised 2003 May 21

        weightlimit = 40
```

■ summary

In this notebook it is shown that if **x** is a function and if **y** is a singleton, then **iterate[x,y]** is a function. The wrappers **singleton** and **funpart** are used to avoid explicit mention of the two hypotheses in this theorem. The derivation is done in two steps. First, the theorem is derived for the special case of total functions, that is a function whose domain is the class **V** of all sets. The general case is obtained by extending an arbitrary function to a total function.

■ abbreviations

We will show by induction that the class **iterate[funpart[x],singleton[y]]** is a function . Some abbreviations may help make the derivation more readable. The first is a permanent abbreviation:

```
In[2]:= INDUCTIVE[x]

Out[2]= and[member[0, x], subclass[image[SUCC, x], x]]
```

The next abbreviation is a temporary one used in this notebook only:

```
In[3]:= ifs[x_, y_] := iterate[funpart[x], singleton[y]]
```

We begin with the special case when **funpart[x]** is total, and later remove this restriction.

■ First steps

From a thin−ness theorem we deduce:

```
In[4]:= SubstTest[implies, and[thin[u], member[v, V]],
            member[iterate[u, v], V],
            {u -> funpart[x], v -> singleton[y]}]

Out[4]= member[iterate[funpart[x], singleton[y]], V] == True
```

```
In[5]:= member[iterate[funpart[x_], singleton[y_]], V] := True

In[6]:= SubstTest[implies, and[member[z, u], subclass[u, v]], member[z, v],
            {u -> range[SINGLETON], v -> image[inverse[IMAGE[funpart[x]]], range[SINGLETON]]}]

Out[6]= or[and[member[z, V], member[image[funpart[x], z], range[SINGLETON]]],
          not[equal[V, domain[funpart[x]]]], not[member[z, range[SINGLETON]]]] == True

In[7]:= or[and[member[z_, V], member[image[funpart[x_], z_], range[SINGLETON]]],
          not[equal[V, domain[funpart[x_]]]], not[member[z_, range[SINGLETON]]]] := True
```

This statement is transformed when **z** is instantiated to the case of interest:

```
In[8]:= SubstTest[or, and[member[z, V], member[image[funpart[x], z], range[SINGLETON]]],
         not[equal[V, domain[funpart[x]]]], not[member[z, range[SINGLETON]]],
         z -> image[ifs[x, y], singleton[w]]]

Out[8]= or[member[image[funpart[x], image[iterate[funpart[x], singleton[y]], singleton[w]]],
           range[SINGLETON]], not[equal[V, domain[funpart[x]]]], not[member[
            image[iterate[funpart[x], singleton[y]], singleton[w]], range[SINGLETON]]]] == True

In[9]:= or[
          member[image[funpart[x_], image[iterate[funpart[x_], singleton[y_]], singleton[w_]]],
           range[SINGLETON]], not[equal[V, domain[funpart[x_]]]],
          not[member[image[iterate[funpart[x_], singleton[y_]], singleton[w_]],
            range[SINGLETON]]]] := True
```

This following fact is needed for the next step, whose goal is to eliminate the variable **w**.

```
In[10]:= SubstTest[funpart, composite[z, SUCC], z -> ifs[x, y]]

Out[10]= funpart[composite[funpart[x], iterate[funpart[x], singleton[y]]]] ==
          composite[funpart[iterate[funpart[x], singleton[y]]], SUCC]

In[11]:= funpart[composite[funpart[x_], iterate[funpart[x_], singleton[y_]]]] :=
          composite[funpart[iterate[funpart[x], singleton[y]]], SUCC]

In[12]:= Map[equal[V, #] &,
           SubstTest[class, w, or[member[image[f, image[z, singleton[w]]], range[SINGLETON]],
             not[equal[V, domain[f]]], not[member[image[z, singleton[w]], range[SINGLETON]]]],
            {f -> funpart[x], z -> ifs[x, y]}]] // Reverse

Out[12]= or[not[equal[V, domain[funpart[x]]]],
           subclass[image[SUCC, domain[funpart[iterate[funpart[x], singleton[y]]]]],
            domain[funpart[iterate[funpart[x], singleton[y]]]]] == True

In[13]:= or[not[equal[V, domain[funpart[x_]]]],
           subclass[image[SUCC, domain[funpart[iterate[funpart[x_], singleton[y_]]]]],
            domain[funpart[iterate[funpart[x_], singleton[y_]]]]] := True
```

■ application of mathematical induction

Mathematical induction is applied to the domain of **funpart[ifs[x,y]]**.

```
In[14]:= SubstTest[implies, INDUCTIVE[z], subclass[omega, z],
             z -> domain[funpart[ifs[x, y]]]]

Out[14]= or[not[member[y, V]],
           not[subclass[image[SUCC, domain[funpart[iterate[funpart[x], singleton[y]]]]],
             domain[funpart[iterate[funpart[x], singleton[y]]]]],
           subclass[omega, domain[funpart[iterate[funpart[x], singleton[y]]]]] == True
```

```
In[15]:= or[not[member[y_, V]],
            not[subclass[image[SUCC, domain[funpart[iterate[funpart[x_], singleton[y_]]]]],
              domain[funpart[iterate[funpart[x_], singleton[y_]]]]],
            subclass[omega, domain[funpart[iterate[funpart[x_], singleton[y_]]]]] := True
```

From this we deduce:

```
In[16]:= Map[not, SubstTest[and, implies[p1, p2],
            implies[and[p0, p2], p3], not[implies[and[p0, p1], p3]],
              {p0 -> member[y, V], p1 -> equal[V, domain[funpart[x]]],
             p2 -> subclass[image[SUCC, domain[funpart[ifs[x, y]]]], domain[funpart[ifs[x, y]]]],
             p3 -> subclass[omega, domain[funpart[ifs[x, y]]]]}]]

Out[16]= or[not[equal[V, domain[funpart[x]]]], not[member[y, V]],
            subclass[omega, domain[funpart[iterate[funpart[x], singleton[y]]]]] == True

In[17]:= or[not[equal[V, domain[funpart[x_]]]], not[member[y_, V]],
            subclass[omega, domain[funpart[iterate[funpart[x_], singleton[y_]]]]] := True

In[18]:= SubstTest[implies, and[subclass[u, v], subclass[v, w]], subclass[u, w],
              {u -> domain[funpart[ifs[x, y]]], v -> domain[ifs[x, y]], w -> omega}]

Out[18]= subclass[domain[funpart[iterate[funpart[x], singleton[y]]]], omega] == True

In[19]:= subclass[domain[funpart[iterate[funpart[x_], singleton[y_]]]], omega] := True

In[20]:= SubstTest[implies, and[subclass[u, v], subclass[v, w]], subclass[u, w],
              {u -> domain[ifs[x, y]], v -> omega, w -> domain[funpart[ifs[x, y]]]}]

Out[20]= or[FUNCTION[iterate[funpart[x], singleton[y]]],
            not[subclass[omega, domain[funpart[iterate[funpart[x], singleton[y]]]]]] == True

In[21]:= or[FUNCTION[iterate[funpart[x_], singleton[y_]]],
            not[subclass[omega, domain[funpart[iterate[funpart[x_], singleton[y_]]]]]] := True
```

We can now deduce a preliminary version of the main theorem. This statement still contains two unnecesary hypotheses that will be removed shortly.

```
In[22]:= Map[not, SubstTest[and, implies[and[p0, p1], p2],
            implies[p2, p3], not[implies[and[p0, p1], p3]],
                  {p0 -> member[y, V], p1 -> equal[V, domain[funpart[x]]],
          p2 -> subclass[omega, domain[funpart[ifs[x, y]]]],
                  p3 -> FUNCTION[ifs[x, y]]}]]

Out[22]= or[FUNCTION[iterate[funpart[x], singleton[y]]],
            not[equal[V, domain[funpart[x]]]], not[member[y, V]]] == True

In[23]:= or[FUNCTION[iterate[funpart[x_], singleton[y_]]],
            not[equal[V, domain[funpart[x_]]]], not[member[y_, V]]] := True
```

■ the empty function

It will be shown in this section that the hypothesis that **y** be a set is not needed. If **y** is not a set, then **singleton[y]** is the empty set. This is not a problem, because in this case iteration produces the empty function.

```
In[24]:= SubstTest[implies, and[equal[u, v], FUNCTION[v]], FUNCTION[u], v -> 0]

Out[24]= or[FUNCTION[u], not[equal[0, u]]] == True
```

```
In[25]:= or[FUNCTION[u_], not[equal[0, u_]]] := True

In[26]:= SubstTest[implies, equal[z, 0], FUNCTION[z], z -> iterate[x, singleton[y]]]

Out[26]= or[FUNCTION[iterate[x, singleton[y]]], member[y, V]] == True

In[27]:= or[FUNCTION[iterate[x_, singleton[y_]]], member[y_, V]] := True
```

This allows us to remove the unnecessary sethood hypothesis:

```
In[28]:= Map[not,
           SubstTest[and, implies[and[p1, p2], p3], implies[not[p1], p3], not[implies[p2, p3]],
                  {p1 -> member[y, V], p2 -> equal[V, domain[funpart[x]]],
                  p3 -> FUNCTION[ifs[x, y]]}]]

Out[28]= or[FUNCTION[iterate[funpart[x], singleton[y]]],
           not[equal[V, domain[funpart[x]]]]] == True

In[29]:= or[FUNCTION[iterate[funpart[x_], singleton[y_]]],
           not[equal[V, domain[funpart[x_]]]]] := True
```

■ the general case of a non−total function

The last step is to remove the hypothesis of totality. Every function can be extended to a total function. This can be done in more than one way. One convenient method is to use:

```
In[30]:= total[x_] := union[x, id[complement[domain[x]]]]

In[31]:= domain[total[x]]

Out[31]= V
```

The following step takes a while:

```
In[32]:= FUNCTION[total[funpart[x]]] // AssertTest

Out[32]= FUNCTION[union[funpart[x], id[complement[domain[funpart[x]]]]]] == True

In[33]:= FUNCTION[union[funpart[x], id[complement[domain[funpart[x]]]]]] := True

In[34]:= SubstTest[implies, equal[V, domain[funpart[z]]],
              FUNCTION[iterate[funpart[z], singleton[y]]], z -> total[funpart[x]]]

Out[34]= FUNCTION[
           iterate[union[funpart[x], id[complement[domain[funpart[x]]]]], singleton[y]]] == True

In[35]:= FUNCTION[iterate[
              union[funpart[x_], id[complement[domain[funpart[x_]]]]], singleton[y_]]] := True

In[36]:= SubstTest[implies, subclass[u, v], subclass[iterate[u, w], iterate[v, w]],
              {u -> funpart[x], v -> total[funpart[x]], w -> singleton[y]}]

Out[36]= subclass[iterate[funpart[x], singleton[y]],
           iterate[union[funpart[x], id[complement[domain[funpart[x]]]]], singleton[y]]] == True

In[37]:= subclass[iterate[funpart[x_], singleton[y_]], iterate[
              union[funpart[x_], id[complement[domain[funpart[x_]]]]], singleton[y_]]] := True
```

The final step is to use the monotonicity of **iterate**.

```
In[38]:= SubstTest[implies, and[subclass[u, v], FUNCTION[v]], FUNCTION[u],
             {u -> iterate[funpart[x], singleton[y]],
           v -> iterate[union[funpart[x], id[complement[domain[funpart[x]]]]], singleton[y]]}]

Out[38]= FUNCTION[iterate[funpart[x], singleton[y]]] == True

In[39]:= FUNCTION[iterate[funpart[x_], singleton[y_]]] := True
```