crossed iterates

Johan G. F. Belinfante
2003 May 16

```
In[1]:= << goedel52.r73; << tools.m

        :Package Title: goedel52.r73         2003 May 14 at 8:40 p.m.

        It is now:  2003 May 22 at 10:41

        Loading Simplification Rules

        TOOLS.M                     Revised 2003 May 21

        weightlimit = 40
```

■ summary

This notebook contains a formula which refers to two independent iterations being carried out simultaneously. The result is derived in two different ways. One uses a formula for the powers of a cross product. The other derivation is an application of iteration uniqueness. Both derivations make use of a standard tool called **SubstTest**:

```
In[2]:= ? SubstTest

        Global‘SubstTest

        SubstTest[f_, x__, r_] := f @@ ({x} /. r) == (f @@ {x} /. r)
```

This tool compares two methods of evaluating an expression: one method is to evaluate **f[x]** and then apply a set **r** of replacement rules. The other is to apply the replacement rules to **x** first, and then apply the head **f**. The double blank on the expression **x** allows one to use this tool when the head **f** is being applied to several arguments. This is the same idea that underlies Knuth−Bendix completion, but here one is doing it manually.

■ a direct derivation using powers of cross products

The direct derivation uses two results derived previously. One is the connection between **iterate** and **power**:

```
In[3]:= image[inverse[rotate[composite[inverse[power[w]], SWAP]]], z]

Out[3]= iterate[w, z]
```

This formula is admittedly a bit hard to remember, but one can rederive it quickly using an abstraction tool:

```
In[4]:= abstract[y, iterate[x, y]]

Out[4]= inverse[rotate[composite[inverse[power[x]], SWAP]]]
```

The other ingredient is a formula for powers of cross products:

```
In[5]:= power[cross[u, v]]

Out[5]= union[cart[singleton[0], id[complement[cart[V, V]]]],
          intersection[composite[cross[inverse[FIRST], inverse[FIRST]], power[u]],
           composite[cross[inverse[SECOND], inverse[SECOND]], power[v]]]]
```

One advantage of this method is that one does not need to know before hand the formula that is being derived. The whole derivation is done in a single step:

```
In[6]:= SubstTest[image, inverse[rotate[composite[inverse[power[w]], SWAP]]],
          z, {w -> cross[u, v], z -> cart[x, y]}] // Reverse

Out[6]= iterate[cross[u, v], cart[x, y]] == intersection[
          composite[inverse[FIRST], iterate[u, x]], composite[inverse[SECOND], iterate[v, y]]]
```

One could add this as a new rewrite rule:

```
In[7]:= iterate[cross[u_, v_], cart[x_, y_]] := intersection[
          composite[inverse[FIRST], iterate[u, x]], composite[inverse[SECOND], iterate[v, y]]]
```

This result can also be expressed more concisely as follows:

```
In[8]:= iterate[cross[u, v], cart[x, y]] == composite[cross[iterate[u, x], iterate[v, y]], DUP]

Out[8]= True
```

■ an alternate derivation using iteration uniqueness

The second method is more basic in that it does not require using any formulas about powers. The uniqueness of iteration is used instead. A disadvantage of this method is that one does need to know the formula that is being derived. The rewrite rule derived above is first removed, and then will be rederived using uniqueness of iteration.

```
In[9]:= iterate[cross[u_, v_], cart[x_, y_]] =.
```

The idea is to use iteration uniqueness:

```
In[10]:= implies[and[equal[composite[s, r], composite[r, SUCC]],
                 equal[image[r, singleton[0]], t]],
          equal[composite[r, id[omega]], iterate[s, t]]]

Out[10]= True
```

An obstacle in this derivation is getting around the following rewrite rule which holds for any function **ff**.

```
In[11]:= composite[cross[ff, ff], DUP]

Out[11]= composite[DUP, ff]
```

The tool needed here is **Assoc**. This tool applies the associative law to a composite of three factors. First this is done for the case of the successor function **SUCC**:

```
In[12]:= Assoc[cross[iterate[u, x], iterate[v, y]], cross[SUCC, SUCC], DUP]

Out[12]= composite[intersection[composite[inverse[FIRST], iterate[u, x]],
            composite[inverse[SECOND], iterate[v, y]]], SUCC] ==
          intersection[composite[inverse[FIRST], u, iterate[u, x]],
           composite[inverse[SECOND], v, iterate[v, y]]]

In[13]:= composite[intersection[composite[inverse[FIRST], iterate[u_, x_]],
            composite[inverse[SECOND], iterate[v_, y_]]], SUCC] :=
          intersection[composite[inverse[FIRST], u, iterate[u, x]],
           composite[inverse[SECOND], v, iterate[v, y]]]
```

A second application is required to remove a factor of **id[omega]**.

```
In[14]:= Assoc[cross[iterate[u, x], iterate[v, y]], id[cart[omega, omega]], DUP]

Out[14]= composite[intersection[composite[inverse[FIRST], iterate[u, x]],
            composite[inverse[SECOND], iterate[v, y]]], id[omega]] ==
          intersection[composite[inverse[FIRST], iterate[u, x]],
           composite[inverse[SECOND], iterate[v, y]]]

In[15]:= composite[intersection[composite[inverse[FIRST], iterate[u_, x_]],
            composite[inverse[SECOND], iterate[v_, y_]]], id[omega]] :=
          intersection[composite[inverse[FIRST], iterate[u, x]],
           composite[inverse[SECOND], iterate[v, y]]]
```

The final step compares two solutions of the same iteration problem, producing the same result as before.

```
In[16]:= SubstTest[implies, and[equal[composite[s, r], composite[r, SUCC]],
                 equal[image[r, singleton[0]], t]],
          equal[composite[r, id[omega]], iterate[s, t]],
             {r -> composite[cross[iterate[u, x], iterate[v, y]], DUP],
           s -> cross[u, v], t -> cart[x, y]}]

Out[16]= equal[intersection[composite[inverse[FIRST], iterate[u, x]],
            composite[inverse[SECOND], iterate[v, y]]], iterate[cross[u, v], cart[x, y]]] == True
```

■ comment

To prove a theorem one does not need two proofs. But sometimes a more complicated proof is helpful to point the way to more general situations. The technique used in the second derivation can in fact be generalized to more interesting cases involving non−independent simultaneous iterations. A typical example is found in the notebook **parabola.nb** where a function **PARABOLA** is defined by converting a recursion relation into a simultaneous iteration problem.