Mac Lane's definition of functor

Johan G. F. Belinfante
2011 December 27

```
In[1]:= SetDirectory["l:"]; << goedel.11dec27b
```

```
:Package Title: goedel.11dec27b              2011 December 27 at 12:15 noon

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

It is now:  2011 Dec 27 at 13:49

Loading Simplification Rules

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

weightlimit = 40

Loading completed.

It is now:  2011 Dec 27 at 14:2
```

summary

This notebook concerns the elimination of the morphism (arrow) variables in a definition of functor given by Saunders MacLane for arrows-only categories. This definition is found on page 13 of the following reference. The definition contains six conditions, most of which contain quantified variables for morphisms. Eliminating these variables yields six variable-free literals, two of which follow from the remaining four.

```
In[2]:= "Saunders Mac Lane, Categories for the
          Working Mathematician, Springer-Verlag, New York, 1971.";
```

This notebook is tutorial in the sense that although no new rewrite rules are derived, the exercise is nonetheless instructive. It is shown that the **case**-wrapped definition of functor introduced today in the notebook **casefunc.nb** satisfies all the Mac Lane conditions.

temporary declarations

It is fairly challenging to eliminate the arrow variables from Mac Lane's definition because of the multiple occurrence of ordered pairs and function application. It is generally faster to eliminate variables using **reify** instead of **class**, but the **GOEDEL** program currently lacks **reify** rules for ordered pairs, so the strategy will be to eliminate ordered pairs. The elimination is also made easier by fixing on a particular (generic) functor **f** from a category **g** to a category **h**, and to derive rewrite rules for these. To this end, the following temporary declarations are introduced.

```
In[3]:= FUNCTION[f] := True
```

```
In[4]:= category[g] := True
```

```
In[5]:= category[h] := True
```

The following lemmas will help to take advantage of conditional rewrite rules for functions. Categories are functions.

Lemma.

```
In[6]:= SubstTest[FUNCTION, cat[x], x → g] // Reverse
Out[6]= FUNCTION[g] == True
```

```
In[7]:= FUNCTION[g] := True
```

Lemma.

```
In[8]:= SubstTest[FUNCTION, cat[x], x → h] // Reverse
Out[8]= FUNCTION[h] == True
```

```
In[9]:= FUNCTION[h] := True
```

Lemma. An **APPLY** rule for **f**.

```
In[10]:= SubstTest[image, funpart[t], set[u], t → f] // Reverse
Out[10]= image[f, set[u]] == set[APPLY[f, u]]
```

```
In[11]:= image[f, set[u_]] := set[APPLY[f, u]]
```

Mac Lane's definition

The **arrows** (or **morphisms**) of an (arrows-only) category are the members of its range. A pair of arrows is **composable** if the pair belongs to the domain of the category.

Sauders Mac Lane defines a **functor** **f** from a category **g** to a category **h** to be a function taking arrows **u ∈ range[g]** to arrows **APPLY[f,u] ∈ range[h]**, carrying each identity of **g** to an identity of **h**, and each composable pair of arrows **pair[u,v] ∈ domain[g]** to a composable pair **pair[APPLY[f, u], APPLY[f,v]] ∈ domain[h]** with

APPLY[f, APPLY[g, pair[u,v]]] = APPLY[h, pair[APPLY[f, u], APPLY[f, v]]].

mapping assumptions

The assumption that **f** is a mapping from **range[g]** to **range[h]** translates to the following conditions on **f**.

```
In[12]:= domain[f] := range[g]
```

```
In[13]:= subclass[range[f], range[h]] := True
```

preservation of identities

The preservation of identities says:

```
In[14]:= or[member[APPLY[f, u_], ids[h]], not[member[u_, ids[g]]]] := True
```

Since the class **ids[g]** is a subclass of **range[g]**, the following fact is automatically recognized:

```
In[15]:= subclass[ids[g], domain[f]]

Out[15]= True
```

Theorem. (Eliminating the arrows variables using **reify**.)

```
In[16]:= Map[equal[V, domain[#]] &, SubstTest[reify, u,
           case[or[member[APPLY[funpart[t], u], ids[h]], not[member[u, ids[g]]]]], t → f]]

Out[16]= subclass[image[f, ids[g]], ids[h]] == True

In[17]:= subclass[image[f, ids[g]], ids[h]] := True
```

preservation of composability

The domain of a category is its **composability** relation.

Lemma. The domain of an arrows-only category is a subclass of the cartesian square of its range.

```
In[18]:= SubstTest[implies, category[x],
           subclass[domain[x], cart[range[x], range[x]]], x → g] // Reverse

Out[18]= subclass[domain[g], cart[range[g], range[g]]] == True

In[19]:= subclass[domain[g], cart[range[g], range[g]]] := True
```

The preservation of composability says:

```
In[20]:= or[member[pair[APPLY[f, u_], APPLY[f, v_]], domain[h]],
           not[member[pair[u_, v_], domain[g]]]] := True
```

There are two variables to be elimimated. The elimination process is speeded up by replacing the ordered pair with a single variable **t**.

Lemma.

```
In[21]:= Map[or[member[first[t], range[g]], #] &,
          SubstTest[implies, and[member[t, u], subclass[u, v]], member[t, v],
            {u -> domain[g], v -> cart[range[g], range[g]]}] // Reverse]

Out[21]= or[member[first[t], range[g]], not[member[t, domain[g]]]] == True

In[22]:= or[member[first[t_], range[g]], not[member[t_, domain[g]]]] := True
```

Lemma.

```
In[23]:= SubstTest[implies, and[member[t, u], subclass[u, v]], member[t, v],
            {u -> domain[g], v -> cart[range[g], range[g]]}] // Reverse // MapNotNot

Out[23]= or[member[second[t], range[g]], not[member[t, domain[g]]]] == True

In[24]:= or[member[second[t_], range[g]], not[member[t_, domain[g]]]] := True
```

Corollary.

```
In[25]:= Map[not,
           SubstTest[and, implies[p1, p2], not[implies[p1, p3]], {p1 → member[t, domain[g]],
             p2 → member[first[t], range[g]], p3 → member[first[t], V]}]] // Reverse

Out[25]= or[member[first[t], V], not[member[t, domain[g]]]] == True

In[26]:= or[member[first[t_], V], not[member[t_, domain[g]]]] := True
```

Lemma. A simplification rule.

```
In[27]:= equiv[and[member[t, domain[g]], member[first[t], V]], member[t, domain[g]]]

Out[27]= True

In[28]:= and[member[t_, domain[g]], member[first[t_], V]] := member[t, domain[g]]
```

Lemma. Replacing ordered pairs with a single variable.

```
In[29]:= SubstTest[implies, equal[t, PAIR[u, v]],
            or[member[pair[APPLY[f, u], APPLY[f, v]], domain[h]], not[member[t, domain[g]]]],
            {u → first[t], v → second[t]}] // Reverse // MapNotNot

Out[29]= or[member[pair[APPLY[f, first[t]], APPLY[f, second[t]]], domain[h]],
           not[member[t, domain[g]]]] == True

In[30]:= (% /. t → t_) /. Equal → SetDelayed
```

A further speedup of the elimination process is obtained by replacing the two occurrences of function application for $\mathbf{f}$ with a single application of the function $\mathbf{f} \otimes \mathbf{f}$. The following technical lemma is needed.

Lemma.

```
In[31]:= or[and[member[first[t], range[g]],
            member[pair[APPLY[f, first[t]], APPLY[f, second[t]]], domain[h]],
            member[second[t], range[g]]], not[member[t, domain[g]]]] // NotNotTest

Out[31]= or[and[member[first[t], range[g]],
            member[pair[APPLY[f, first[t]], APPLY[f, second[t]]], domain[h]],
            member[second[t], range[g]]], not[member[t, domain[g]]]] == True

In[32]:= (% /. t → t_) /. Equal→ SetDelayed
```

Theorem. Using **reify** to eliminate the variables in the preservation of composability condition.

```
In[33]:= Map[equal[V, domain[#]] &, SubstTest[reify, t,
           case[implies[member[t, domain[g]], member[APPLY[funpart[z], t], domain[h]]]],
           z → cross[f, f]]]

Out[33]= subclass[domain[g], composite[inverse[f], domain[h], f]] == True

In[34]:= subclass[domain[g], composite[inverse[f], domain[h], f]] := True
```

preservation of composition

The final requirement on the functor **f** is that it preserves composition for composable arrows.

```
In[35]:= or[equal[APPLY[f, APPLY[g, pair[u_, v_]]], APPLY[h, pair[APPLY[f, u_], APPLY[f, v_]]]],
           not[member[pair[u_, v_], domain[g]]]] := True
```

Lemma. Some ordered pairs are replaced with a single variable.

```
In[36]:= (SubstTest[implies, and[equal[t, s], equal[s, pair[u, v]]],
             or[equal[APPLY[f, APPLY[g, t]], APPLY[h, pair[APPLY[f, u], APPLY[f, v]]]],
              not[member[t, domain[g]]]], s → PAIR[u, v]] /.
            {u → first[t], v → second[t]}) // Reverse // MapNotNot

Out[36]= or[
           equal[APPLY[f, APPLY[g, t]], APPLY[h, pair[APPLY[f, first[t]], APPLY[f, second[t]]]]],
           not[member[t, domain[g]]]] == True

In[37]:= (% /. t → t_) /. Equal→ SetDelayed
```

Because ordered pairs of morphisms have been replaced with a single variable **t**, another **APPLY** rule for **g** is needed.

Lemma.

```
In[38]:= SubstTest[image, funpart[z], set[t], z → g] // Reverse

Out[38]= image[g, set[t]] == set[APPLY[g, t]]

In[39]:= image[g, set[t_]] := set[APPLY[g, t]]
```

Lemma.

```
In[42]:= SubstTest[implies, equal[z, pair[APPLY[f, first[t]], APPLY[f, second[t]]]],
           or[equal[APPLY[f, APPLY[g, t]], APPLY[h, z]], not[member[t, domain[g]]]],
           z → PAIR[APPLY[f, first[t]], APPLY[f, second[t]]]] // Reverse

Out[42]= or[equal[APPLY[f, APPLY[g, t]],
            APPLY[h, PAIR[APPLY[f, first[t]], APPLY[f, second[t]]]]], not[member[t, domain[g]]],
           not[member[first[t], range[g]]], not[member[second[t], range[g]]] == True

In[43]:= (% /. t → t_) /. Equal→ SetDelayed
```

Lemma. Cleaning up the preceding result.

```
In[44]:= Map[not, SubstTest[and, implies[p1, p2], implies[p1, p3], not[implies[p1, p4]],
            {p1 -> member[t, domain[g]], p2 -> member[first[t], range[g]],
             p3 -> member[second[t], range[g]], p4 -> equal[APPLY[f, APPLY[g, t]],
               APPLY[h, PAIR[APPLY[f, first[t]], APPLY[f, second[t]]]]]}]] // Reverse

Out[44]= or[
           equal[APPLY[f, APPLY[g, t]], APPLY[h, PAIR[APPLY[f, first[t]], APPLY[f, second[t]]]]],
           not[member[t, domain[g]]]] == True

In[45]:= (% /. t → t_) /. Equal→ SetDelayed
```

Theorem. Using **reify** to eliminate the variable **t**.

```
In[46]:= Map[equal[V, domain[#]] &, SubstTest[reify, t, case[
            implies[member[t, domain[g]], equal[APPLY[funpart[u], t], APPLY[funpart[v], t]]],
           {u → composite[f, g], v → composite[h, cross[f, f]]}]]

Out[46]= subclass[domain[g], fix[composite[inverse[g], inverse[f], h, cross[f, f]]]] == True

In[47]:= % /. Equal→ SetDelayed
```

Theorem. A simpler variable-free statement for the preservation of composition.

```
In[48]:= SubstTest[subclass, domain[funpart[x]],
           fix[composite[inverse[funpart[x]], funpart[y]]],
           {x → composite[f, g], y → composite[h, cross[f, f]]}]

Out[48]= subclass[composite[f, g], composite[h, cross[f, f]]] == True

In[49]:= subclass[composite[f, g], composite[h, cross[f, f]]] := True
```

a functor definition

A formal definition is in order. The condition that **x** and **y** be categories will be dropped for simplicity. The above analysis of Mac Lane's definition of functor suggests that a predicate **functor[t, x, y]** be defined as an abbreviation for the following conjunction of six literals.

```
and[FUNCTION[t], equal[domain[t], range[x]], subclass[range[t], range[y]],
 subclass[image[t, ids[x]], ids[y]], subclass[domain[x], composite[inverse[t], domain[y], t]],
 subclass[composite[t, x], composite[y, cross[t, t]]]]
```

These six conditions are not all independent of each other. This definition can be simplified by omitting the third and fifth literals. In the notebook **casefunc.nb** the following simpler **case**-wrapped definition was introduced.

```
In[50]:= case[functor[t, x, y]]

Out[50]= case[and[equal[domain[t], range[x]], FUNCTION[t],
           subclass[composite[t, x], composite[y, cross[t, t]]],
           subclass[image[t, ids[x]], ids[y]]]]
```

Theorem. The **case**-wrapped definition of functor satisfies all six conditions required by Mac Lane.

```
In[51]:= implies[functor[t, x, y], and[FUNCTION[t], equal[domain[t], range[x]],
             subclass[range[t], range[y]], subclass[image[t, ids[x]], ids[y]],
             subclass[domain[x], composite[inverse[t], domain[y], t]],
             subclass[composite[t, x], composite[y, cross[t, t]]]]] // not // not

Out[51]= True
```