# Associative Relations

Based on a talk presented 2003 July 11 at the AWARD2003 workshop at Argonne.

*Johan G. F. Belinfante*
*2003 July 17*

```
In[1]:=  << goedel52.s56; << tools.m

        :Package Title: goedel52.s56       2003 July 17 at 5:05 p.m.

        It is now:  2003 Jul 18 at 15:39

        Loading Simplification Rules

        TOOLS.M                      Revised 2003 July 8

        weightlimit = 40
```

## ■ eliminating quantifiers

The dominant theme at the AWARD 2003 workshop was the study of single axioms for a variety of logical and algebraic systems. **Otter** is an extremely powerful tool for studying such questions. A natural question is whether **Otter** could also be used to reason about algebraic systems as well as reasoning within an algebraic system. An example is the proof of Lagrange’s theorem in group theory; the reasoning involves not only the axioms for group theory, but also such notions as subgroups, cosets, counting and arithmetic. A natural idea would be to tackle such studies by extending the work of Boyer, et al., Quaife and the author, based on Gödel’s class theory.

Semigroups come to mind as one of the simplest algebraic systems. A semigroup can be regarded as a binary function **f** whose domain is a cartesian square **cart[s, s]** and whose range is contained in **s**, with the associative law as the sole axiom. The associative law is usually stated in terms of applications of the function **f**, but for the present discussion it is more convenient to rewrite the associative law in terms of images of singletons:

```
In[2]:=  image[f, cart[image[f, cart[singleton[u], singleton[v]]], singleton[w]]] ==
           image[f, cart[singleton[u], image[f, cart[singleton[v], singleton[w]]]]];
```

This equation is to hold for all **u**, **v** and **w** belonging to **s**. The restriction of the quantifiers to **s** is not essential because the domain information is already contained in the function **f**. If any of the variables **u**, **v** or **w** fails to belong to **s**, then the images are empty and the above equation reduces to the tautology **0 = 0**. A natural starting point for the theory of semigroups would be to define a function **f** to be associative if and only if the above law holds. When **Otter** is used, the only−if part of the definition of associativity produces Skolem functions. (Of course, one does not encounter these Skolem functions when all one is doing is deriving consequences of the associative law.)

## ■ reification

One of the pleasant features of Gödel’s class theory is that all quantifiers over set−variables can be eliminated. Since the variables **u**, **v** and **w** in the definition of associativity all refer to sets, these variables can be eliminated to obtain a quantifier−free definition of associativity. The technical means for doing this in a nice way is to use a technique that the author calls reification. The idea is to assign an object (class) to each class−constructor. Consider, for example, the power class contructor **P[x]**. Corresponding to this constructor, there is a function **POWER** that takes each set to the corresponding power set:

```
In[3]:= class[pair[x, y], equal[y, P[x]]]

Out[3]= POWER
```

This idea works fine for many constructors, but not when the constructor takes a set to a proper class. For example, the complement of a set is a proper class, and so there is no useful function that corresponds to it:

```
In[4]:= class[pair[x, y], equal[y, complement[x]]]

Out[4]= 0
```

The idea of reification is to replace **equal** by **member**. Of course, one no longer gets a function, but one does obtain a relation that captures all of the information contained in the constructor that pertains to the case when the constructor is applied to sets. (One does lose the information about the constructor when it is applied to proper classes.)

```
In[5]:= class[pair[x, y], member[y, P[x]]]

Out[5]= inverse[S]

In[6]:= class[pair[x, y], member[y, complement[x]]]

Out[6]= composite[Id, complement[inverse[E]]]
```

In general, **reify[x, F[x]]** denotes the result of reifying a constructor **F**. For example:

```
In[7]:= reify[x, P[x]]

Out[7]= inverse[S]
```

An interesting feature of reification is that the reification of a composite constructor **F[G[x]]** can be expressed in terms of the reification of the inner constructor. The formula for this is different for each outer constructor. For example:

```
In[8]:= reify[x, complement[G[x]]]

Out[8]= composite[Id, complement[reify[x, G[x]]]]

In[9]:= reify[x, P[G[x]]]

Out[9]= inverse[LB[reify[x, G[x]]]]
```

## ■ the associative law without variables

When reification is applied to the middle variable **v** in the associativity condition, one obtains a statement with one fewer variable. It says that left multiplication by **u** commutes with right multiplication by **w**. (For convenience, application of the function **f** will be called multiplication.)

```
In[10]:= Map[reify[v, #] &,
          image[f, cart[image[f, cart[singleton[u], singleton[v]]], singleton[w]]] ==
           image[f, cart[singleton[u], image[f, cart[singleton[v], singleton[w]]]]]]

Out[10]= composite[f, RIGHT[w], f, LEFT[u]] == composite[f, LEFT[u], f, RIGHT[w]]
```

The functions **LEFT[u]** and **RIGHT[w]** are defined as follows:

```
In[11]:= class[pair[x, pair[y, z]], and[equal[y, u], equal[z, x]]]

Out[11]= LEFT[u]

In[12]:= class[pair[x, pair[y, z]], and[equal[y, x], equal[z, w]]]

Out[12]= RIGHT[w]
```

The composite function **composite[f, LEFT[u]]** is left−multiplication by **u** and **composite[f, RIGHT[w]]** is right−multiplication by **w**.

The elimination of variables can be continued. The results are improved by judicious applications of **flip**, **rotate** and **inverse**:

```
In[13]:= Map[rotate[inverse[reify[u, #]]] &,
          composite[f, RIGHT[w], f, LEFT[u]] == composite[f, LEFT[u], f, RIGHT[w]] ]

Out[13]= composite[f, RIGHT[w], f, id[cart[V, V]]] ==
          composite[f, cross[Id, composite[f, RIGHT[w]]]]

In[14]:= Map[flip[rotate[inverse[reify[w, #]]]] &, composite[f, RIGHT[w], f, id[cart[V, V]]] ==
           composite[f, cross[Id, composite[f, RIGHT[w]]]]]

Out[14]= composite[f, cross[composite[f, id[cart[V, V]]], Id]] ==
          composite[f, cross[Id, f], ASSOC]
```

Because **f** is a binary function, this equation can be cleaned up by noticing that **composite[f, id[cart[V, V]] = f**. The associative law in this form also makes sense when **f** is not single−valued. The predicate **associative** in the **GOEDEL** program is defined as follows:

```
In[15]:= associative[x] // AssertTest

Out[15]= associative[x] ==
          and[equal[composite[x, cross[x, Id]], composite[x, cross[Id, x], ASSOC]],
           subclass[x, cart[cart[V, V], V]]]
```

It would be natural to call **x** an associative relation when this condition holds. (Remark: this term has other meanings in psychology and elsewhere.) No restrictions are placed on the domain other than that it is contained in the class of ordered pairs. The class **x** need not be a set.

## ■ some examples

Some familiar functions that are associative come to mind: most of these are proper classes.

```
In[16]:= Select[NamedClasses, associative]

Out[16]= {0, CAP, COMPOSE, CUP, FIRST, NATADD, NATMUL, SECOND, SYMDIF}
```

The functions **NATADD** and **NATMUL** are addition and multiplication of natural numbers. The domains of all these are cartesian squares:

```
In[17]:= Map[domain, {0, CAP, COMPOSE, CUP, FIRST, NATADD, NATMUL, SECOND, SYMDIF}]

Out[17]= {0, cart[V, V], cart[V, V], cart[V, V], cart[V, V],
          cart[omega, omega], cart[omega, omega], cart[V, V], cart[V, V]}
```

The domain of an associative function need not be a cartesian square. A simple example is the inverse of the duplication function:

```
In[18]:= class[pair[pair[x, y], z], and[equal[x, y], equal[y, z]]]

Out[18]= inverse[DUP]

In[19]:= {associative[inverse[DUP]], FUNCTION[inverse[DUP]], domain[inverse[DUP]]}

Out[19]= {True, True, Id}
```

Another example is the so−called pair−groupoid, studied by W. Brandt in the 1920's. The idea is to define a multiplication by **(a, b) (b, c) = (a, c)**. The function that does this is:

```
In[20]:= class[pair[pair[pair[u, v], pair[w, x]], pair[y, z]],
          and[equal[v, w], equal[u, y], equal[x, z]]]

Out[20]= composite[RIF, cross[SWAP, SWAP]]
```

This is indeed an associative function. Its domain is not a cartesian square.

```
In[21]:= Map[{associative[#], FUNCTION[#], domain[#]} &, {composite[RIF, cross[SWAP, SWAP]]}]

Out[21]= {{True, True, composite[inverse[FIRST], SECOND]}}
```

The rotation−invariant function **RIF** that appears here has many other important applications. It has the property of converting cartesian products into composites:

```
In[22]:= image[RIF, cart[x, y]]

Out[22]= composite[inverse[y], inverse[x]]
```

Padmanabhan pointed out another example of an associative function:

```
In[23]:= class[pair[pair[pair[u, v], pair[w, x]], pair[y, z]], and[equal[u, y], equal[x, z]]]

Out[23]= cross[FIRST, SECOND]
```

This is also an associative function.  Its domain is a cartesian square.

```
In[24]:= Map[{associative[#], FUNCTION[#], domain[#]} &, {cross[FIRST, SECOND]}]

Out[24]= {{True, True, cart[cart[V, V], cart[V, V]]}}
```

## ■ some theorems about associative relations

Only a few theorems have been derived so far about associative relations.  The flip of an associative relation is associative:

```
In[25]:= associative[composite[x, SWAP]]

Out[25]= associative[composite[x, id[cart[V, V]]]]
```

The direct product of associative relations is associative:

```
In[26]:= implies[and[associative[x], associative[y]],
          associative[composite[cross[x, y], TWIST]]]

Out[26]= True
```

The direct product is defined as follows:

```
In[27]:= class[pair[pair[pair[u1, u2], pair[v1, v2]], pair[w1, w2]],
              and[member[pair[pair[u1, v1], w1], x], member[pair[pair[u2, v2], w2], y]]]

Out[27]= composite[cross[x, y], TWIST]
```

Associated with any associative relation are left and right divisibility relations, and each of these is transitive.

```
In[28]:= implies[associative[x], TRANSITIVE[composite[x, inverse[FIRST]]]]

Out[28]= True

In[29]:= implies[associative[x], TRANSITIVE[composite[x, inverse[SECOND]]]]

Out[29]= True
```

## ■ semigroups, quasigroups and groups

The statement that **f** is a semigroup on **s** can be formulated as:

```
In[30]:= semigroup[f_, s_] :=
          and[associative[f], FUNCTION[f], equal[domain[f], cart[s, s]], subclass[range[f], s]]
```

For example:

```
In[31]:= semigroup[NATADD, omega]

Out[31]= True
```

The definition of quasigroup does not involve associativity. The definition says that **f**, **rotate[f]** and **rotate[rotate[f]]** are all functions with domain **cart[s,s]**.

```
In[32]:= quasigroup[f_, s_] :=
          and[FUNCTION[f], FUNCTION[rotate[f]], FUNCTION[rotate[composite[f, SWAP]]],
                 equal[domain[f], cart[s, s]], equal[composite[f, inverse[FIRST]], cart[s, s]],
           equal[composite[f, inverse[SECOND]], cart[s, s]]]
```

The fact that **range[f] = s** can be deduced as a theorem. Note that the empty set satisfies this condition:

```
In[33]:= quasigroup[0, 0]

Out[33]= True
```

The traditional definition of group requires it to be nonempty. This condition is independent of the other requirements in the definition of a group.

```
In[34]:= group[f_, s_] := and[associative[f], quasigroup[f, s], not[equal[0, s]]]
```

This definition of group does not require that **s** be a set. The symmetric difference function defines a group on the universal class, for example:

```
In[35]:= group[SYMDIF, V]

Out[35]= True
```

## ■ abstraction

Although the predicate **associative** will probably suffice for most algebraic studies, it is also possible to obtain an explicit formula for the class of all associative relations. To do this, it is useful to rewrite the associative law in an unusual way which transforms it into a statement about the cartesian square of **f**. The chief tool here is abstraction, which is defined in terms of reification. The process is vaguely analogous to abstraction in lambda calculus.

```
In[36]:= Begin["Goedel‘Private‘"];

In[37]:= ?? abstract

        abstract[x,F[x]]  yields a class  y  satisfying  F[x] = image[y,x]  if such a class exists

        abstract[x_, y_] := composite[reify[x, y], SINGLETON]
```

A simple example is the sum−class constructor:

```
In[38]:= class[z, exists[y, and[member[z, y], member[y, x]]]]

Out[38]= U[x]
```

Abstraction yields:

```
In[39]:= abstract[x, U[x]]

Out[39]= inverse[E]
```

One needs to verify that this is correct because the **GOEDEL** program does not investigate whether **U[x]** can in fact be written as an image. It can:

```
In[40]:= image[inverse[E], x]

Out[40]= U[x]
```

For binary constructors, one can iterate this procedure. Consider the cartesian product constructor, for example:

```
In[41]:= abstract[x, abstract[y, cart[x, y]]]

Out[41]= composite[id[inverse[SECOND]], inverse[SECOND], inverse[FIRST]]
```

It works:

```
In[42]:= image[image[composite[id[inverse[SECOND]], inverse[SECOND], inverse[FIRST]], x], y]

Out[42]= cart[x, y]
```

For **composite**, the rotation−invariant function **RIF** makes its appearance:

```
In[43]:= abstract[x, abstract[y, composite[x, y]]]

Out[43]= composite[cross[Id, SWAP], inverse[RIF]]

In[44]:= image[image[composite[cross[Id, SWAP], inverse[RIF]], x], y]

Out[44]= composite[x, y]
```

In general, for a binary constructor, one can abstract on the two variables in either order. If the process works in one order, it also works in the other order. It is easy to relate the one order to the other:

```
In[45]:= Map[abstract[x, abstract[y, #]] &, image[image[u, x], y] == image[image[v, y], x]]

Out[45]= composite[id[cart[V, V]], u] == inverse[rotate[composite[inverse[v], SWAP]]]
```

The identity on the left side is not essential, both here and below. The result obtained can be verified directly:

```
In[46]:= image[image[inverse[rotate[composite[inverse[v], SWAP]]], x], y]

Out[46]= image[image[v, y], x]
```

In addition, one can also replace the inner **image** with **cart**:

```
In[47]:= Map[abstract[x, abstract[y, #]] &, image[image[u, x], y] == image[w, cart[x, y]]]

Out[47]= composite[id[cart[V, V]], u] == composite[SWAP, inverse[rotate[composite[w, SWAP]]]]
```

One can solve for **w**:

```
In[48]:= Map[rotate[inverse[#]] &, %]

Out[48]= rotate[inverse[u]] == composite[w, id[cart[V, V]]]
```

The result can be verified directly:

```
In[49]:= image[rotate[inverse[u]], cart[x, y]]

Out[49]= image[image[u, x], y]
```

## ■ the class of all associative relations

The expressions that appear in the associative law can not be written as image of anything, but one can apply abstraction after replacing one of the variables on each side by a different variable. In other words, one considers the following generalization of the associative law:

```
In[50]:= composite[x, cross[y, Id]] == composite[x, cross[Id, y], ASSOC];
```

Abstract on **x** and **y** and apply **rotate** and **inverse**. The upshot is that these formulas can be derived:

```
In[51]:= image[composite[SWAP, RIF,
           cross[Id, composite[cross[SWAP, Id], inverse[RIF]]]], cart[x, y]]

Out[51]= composite[x, cross[y, Id]]

In[52]:= image[composite[SWAP, RIF,
           cross[Id, composite[cross[inverse[ASSOC], SWAP], inverse[RIF]]]], cart[x, y]]

Out[52]= composite[x, cross[Id, y], ASSOC]
```

Setting **y** equal to **x**, one finds that the associative law can now be written in the form of a condition on the cartesian square:

```
In[53]:= image[u, cart[x, x]] == image[v, cart[x, x]];
```

From this observation it is not hard to come up with the following formula for the class of associative relations:

```
In[54]:= intersection[fix[image[inverse[CART], fix[composite[inverse[IMAGE[
               composite[SWAP, RIF, cross[Id, composite[cross[SWAP, Id], inverse[RIF]]]]]],
             IMAGE[composite[SWAP, RIF, cross[Id, composite[cross[inverse[ASSOC], SWAP],
                 inverse[RIF]]]]]]]]]], P[cart[cart[V, V], V]]]

Out[54]= ASSOCIATIVE
```

For example, the addition and multiplication functions for natural number arithmetic are members of this class:

```
In[55]:= member[NATADD, ASSOCIATIVE]

Out[55]= True

In[56]:= member[NATMUL, ASSOCIATIVE]

Out[56]= True
```

## ■ discussion about combinators

In the discussion following this talk the question was raised about the relation of **RIF** to the combinator **B**. The abstraction process does not allow one to find a full model for the lambda calculus with **image** as application and proper classes as combinators. There are no classes that behave like the combinators **S** and **W**, or anything that involves duplicating an argument. One can however find a class that behaves like the combinator **B**. To discover it, one abstracts on the defining equation:

```
In[57]:= Map[abstract[x, abstract[y, abstract[z, #]]] &,
          image[image[image[B, x], y], z] == image[x, image[y, z]]]

Out[57]= composite[id[cart[V, cart[V, V]]], B] == composite[cross[Id, SWAP], inverse[RIF]]
```

Ignore the identity for the moment, and make this into a definition:

```
In[58]:= B := composite[cross[Id, SWAP], inverse[RIF]]
```

This combinator−like relation has the following propertties. It is not a function, but it is the inverse of one:

```
In[59]:= FUNCTION[inverse[B]]

Out[59]= True
```

The first image produces **cross**.

```
In[60]:= image[B, x]

Out[60]= cross[Id, x]
```

The second yields **composite**.

```
In[61]:= image[image[B, x], y]

Out[61]= composite[x, y]
```

The third image gives back the defining equation for **B**.

```
In[62]:= image[image[image[B, x], y], z]

Out[62]= image[x, image[y, z]]
```

The identities that were ignored above can be absorbed into **B**.

```
In[63]:= composite[id[cart[V, cart[V, V]]], B] == B

Out[63]= True
```

Similar results can be obtained for **I**, **K**, **C**, and various other combinators, but not the mockingbird, etc. In the case of **K = inverse[SECOND]**, one needs to restrict the variables in the usual defining equation to nonempty classes:

```
In[64]:= equal[image[image[inverse[SECOND], x], y], x]

Out[64]= or[equal[0, x], not[equal[0, y]]]
```

Suppose one defines a predicate **cancellative** by removing the domain conditions from the definition of quasigroup:

```
In[65]:= cancellative[f_] :=
          and[FUNCTION[f], FUNCTION[rotate[f]], FUNCTION[rotate[composite[f, SWAP]]]]
```

Some examples :

```
In[66]:= Select[BinaryFuns[NamedClasses], cancellative]

Out[66]= {0, ASSOC, KURA, NATADD, RIF, ROT, SWAP, SYMDIF, TWIST}
```

The inverses of **B**, **C = composite[SWAP, inverse[ASSOC], cross[Id, SWAP]]** , and **T = image[C, Id] = inverse[rotate[-SWAP]]** are all cancellative. But **inverse[K]** is not. The relations **rotate[inverse[K]]** and **rotate[inverse[B]]** are associative, but **rotate[inverse[T]]** is not. Another interesting observation: **C** is its own inverse.