Towards a Theory of Moa
proposal for a PhD Thesis

Maarten M. Fokkinga

Version of January 31, 2001 (14:08)

This note proposes an approach to a "Theory of Moa" — the completion of which is deemed worth (and proposed as) a full PhD thesis. The objective of the theory is threefold: first it should provide 'insight' in the principles of Moa (possibly leading to improvements in the documentation, explanations, and so on), second it should provide a formal framework to formulate and prove various claims and properties of Moa, and third it should suggest generalisations (possibly leading to an improved or more general system). In order to achieve the right level of abstraction and generality, we will use some methods and notions from category theory; using category theory is *not* an aim in itself.

A Introduction

1 Monet. Whereas in traditional relational database applications, data is most often needed "by rows at a time" (of one person all its attributes), there also exist database applications, such as datamining and information retrieval, where data is needed "by entire columns at a time" (of all persons just one attribute). For such applications the main-memory database system Monet [3, 4] has been developed; it is optimised for manipulating (traversing and aggregating) entire columns at a time. In fact, Monet has only *binary* tables, so that fetching a column is fetching a table (and this is manipulated entirely in main memory, and stored consecutively on disk).

2 Moa. Moa [4] is a language (a datamodel, to be used at the "logical" level in between the "external" end-user level and the low-level "physical" level of Monet) for defining data representations that exploit Monet's capabilities of efficiently traversing and aggregating entire columns. (The acronym Moa stands for Magnum Object Algebra, where Magnum is/was a Dutch national project in which CWI, UT, TUE, and UvA participated.) Moa is extensible in the sense that the user may define new "basic" structures and their representation in terms of binary tables, and the system then takes care of arbitrary combinations of those structures. In particular, the Moa system *flattens* each nested collection (set, list, bag) to a single, unnested, collection, so that nested traversals are exchanged for single traversals, thus avoiding the famous inefficiency of nested loops. The combination of Moa with Monet has been used with surprisingly good performance in a GIS application [3].

3 Aim. This note proposes an approach to a "Theory of Moa"; the completion of the theory is deemed worth (and proposed as) a full PhD thesis. The objective of the theory is threefold: first it should provide 'insight' in the principles of Moa (possibly leading to improvements in the documentation, explanations, and so on), second it should provide a formal framework to formulate and prove various claims and properties of Moa, and third it should suggest generalisations (possibly leading to an improved or more general system). For the latter we think of a redo of the optimising rewriter, the implementation of modules, and possibly some new value representations. In order to achieve the

right level of abstraction and generality, we will use some methods and notions from category theory [2]. Using category theory is *not* an aim in itself, and we are ready to work non-categorically where appropriate.

4 Category theory. Category theory is a language (consisting of notions and accompanying axioms and theorems) in which various diverse fields of mathematics and computing science can be described in a uniform way. In order to achieve the right level of abstraction (that reveals the similarities between seemingly different and unrelated fields of mathematics or computing science), the language may feel as a straight-jacket when used for the first time in one specific field (programming, say). For example, when describing functional programming, category theory requires to express oneself entirely at the function level, not using explicit arguments and results.

Formulated in a programmers jargon, category theory has only the following concepts (and we will adhere to these in the sequel): *types* $(a, b, c \ldots)$ and *typed total functions* $(f, g, h \ldots$, typed like $f\colon a \to b)$, and a *composition* operation $\circ$ and specific functions $id_a\colon a \to a$ for each type a, such that the following axioms hold true:

$$f\colon a \to b \ \wedge\ g\colon b \to c \ \Rightarrow\ g \circ f \text{ exists } \wedge \ g \circ f\colon a \to c$$

and, whenever the compositions exist (and omitting the subscripts):

$$f \circ (g \circ h) = (f \circ g) \circ h$$
$$f \circ id = f = id \circ f$$

All other concepts in category theory must be defined in terms of these concepts (so that conventional concepts might not be recognised easily at first sight). Note that the concepts of being a 'member' of a type, and of 'application' of a function to an 'argument' do not occur: $a, b, c \ldots$ and $f, g, h \ldots$ may be interpreted quite different from the programmers' intuition about types and typed total functions! For example, $a, b, c, \ldots$ may be interpreted as numbers and $f\colon a \to b$ may be interpreted as a witness of the fact that a is at most b; then the implication '$f\colon a \to b$ and $g\colon b \to c$ implies $g \circ f\colon a \to c$' expresses transitivity of the ordering on numbers. Many more interpretations are possible.

In the sequel, we will use category theory as a source of inspiration rather than as an objective in itself. We shall use the conventional programmer's jargon and intuition, but take care not to exceed the spirit of category theory. Our *use* of category theory is mainly in the derived concepts it provides (in particular the notion of *structure* that we introduce below). When it comes to proofs of properties, we will certainly benefit further of our categorical approach.

B The notion of 'structure'

5 Notation. Throughout the paper we let $a, b, \ldots$ vary over types and $f, g, \ldots$ vary over typed total functions. For Cartesian product we use the following notation:

$$a \times b = \{x\colon a,\ y\colon b \bullet (x, y)\}$$
$$f \times g = \lambda(x, y) \bullet (f\,x, g\,y)$$

So $f \times g$, applied to a tuple of type $a \times b$, subjects the a-constituent of the tuple to f and the b-constituent to g. When $f\colon a \to b$ and $g\colon c \to d$, then $f \times g\colon a \times c \to b \times d$.
For the type of sets we use the following notation:

$$\mathbb{S}a = \text{the type of sets like "}\{x_0, \ldots, x_{n-1}\}\text{", where each } x_i \text{ is of type } a$$
$$\mathbb{S}f = \lambda\,\{x_0, \ldots, x_{n-1}\} \bullet \{f\,x_0, \ldots, f\,x_{n-1}\}$$

So $\mathbb{S}f$, applied to a set of type $\mathbb{S}a$, subjects each a-constituent of the set to f; some people write $\mathbb{S}f$ as $f*$ and call it the "map f". When $f\colon a \to b$, then $\mathbb{S}f\colon \mathbb{S}a \to \mathbb{S}b$.

6 Structure. Consider, as a motivating example, the type $a \times b$ (the type of tuples from a and b) or $\mathbb{S}a$ (the type of sets over a). What property makes that these types are *structured*? The answer is: the existence of a way to manipulate the constituents without changing the structure. For $a \times b$ it is $f \times g$ that manipulates the constituents only,

and for $\mathbb{S}a$ it is $\mathbb{S}f$ that manipulates the constituents while not affecting the structure. The fact that only the constituents are affected and not the structure, is captured by the following property:

$$\begin{array}{lcl} (f \times g) \circ (f' \times g') & = & (f \circ f') \times (g \circ g') \\ id_a \times id_b & = & id_{a \times b} \end{array}$$

resp.,

$$\begin{array}{lcl} \mathbb{S}f \circ \mathbb{S}f' & = & \mathbb{S}(f \circ f') \\ \mathbb{S}\, id_a & = & id_{\mathbb{S}a} \end{array}$$

Thus, a structure not only defines, for given types, a structured type, but also comes equipped with a way to manipulate the constituents without changing the structure. This leads to the following definition.

A *structure* F (the categorical term is *functor*) consists of a mapping of types and a mapping of typed total functions, both denoted by the same name F, satisfying these properties:

$$\begin{array}{lcl} a \text{ is a type} & \Rightarrow & F\,a \text{ is a type} \\ f\colon a \to b & \Rightarrow & F f\colon F\,a \to F\,b \end{array}$$

and

$$\begin{array}{lcl} F(f \circ g \circ \cdots \circ h) & = & F f \circ F g \circ \cdots \circ Fh \\ F\, id & = & id \end{array}$$

(The latter line may be considered a special instance of its preceding line. Written in full the equation reads: $F\, id_a = id_{Fa}$ for all types a.) So, Fa is "the structured type" built out of values of type a, and F itself is "the structure". Function Ff is the function that subjects every constituent of structured type Fa to function f thereby resulting in a structured type Fb. We let $F, G, \ldots$ vary over structures.

Here are some examples of structures:

I — the trivial structure:
$\mathrm{I}a = a$
$\mathrm{I}f = f$

$\mathbb{I}$ — the structure of pairs of equal type:
$\mathbb{I}\,a = a \times a$
$\mathbb{I}f = f \times f = \lambda(x, y) \bullet (f\,x, f\,y)$

$\mathbb{S}$ — the structure of sets; recall:
$\mathbb{S}\,a =$ the type of sets $\{x_0, \ldots, x_{n-1}\}$, $(x_i\colon a)$
$\mathbb{S}f = \lambda\,\{x_0, \ldots, x_{n-1}\} \bullet \{f\,x_0, \ldots, f\,x_{n-1}\}$

$\mathbb{L}$ — the structure of lists:
$\mathbb{L}\,a =$ the type of lists $[x_0, \ldots, x_{n-1}]$, $(x_i\colon a)$
$\mathbb{L}f = \lambda\,[x_0, \ldots, x_{n-1}] \bullet [f\,x_0, \ldots, f\,x_{n-1}]$

Note that the composition of structure F with structure G is a structure again: an F-structure of G-structures. All these example structures are unary; binary structures also exist, and Cartesian product is the prime example. For the sake of simplicity we mainly consider unary structures here (and thus skip the definition for binary and n-ary structures).

Notation. For structures, we use juxtaposition for both application to a type and composition with another structure, and, since $F(G\,a) = (F\,G)a$, we leave out the parentheses, thus writing $F\,G\,a$. We also use the *section* notation from functional programming languages: $(a\times)$ is a unary structure, mapping type b to $a \times b$ and typed total function $f\colon b \to b'$ to $id_a \times f\colon a \times b \to a \times b'$.[1]

7 Regular structures. The notion of structure defined above is very general, maybe too general to be useful. A less general set of structures are the ones that are built by composition from some given elementary structures (like I and $\times$): the so-called *regular* structures. I suspect that regularity will play an important role in the theory of Moa, but for simplicity I skip the definition of regularity here. However, below it might be the case that sometimes we have to restrict ourselves to regular structures, so that it is valid to decompose them into compositions of elementary structures.

8 Datatypes. A datatype definition defines not only constructors (or destructors) for the new type, but also a structure in the sense explained above.

To illustrate this, consider the following datatype definition of binary trees with values of type a in the tips, and constructors *tip* and *join*:

[1] The definition of $(a\times)$ can be written as one equation '$(a\times)x = a \times x$' (for both types and functions x), if we convene to denote both a type and the identity on that type by the same name. The resulting syntactic ambiguity in '$F\,a$' isn't present semantically since $F\,id_a = id_{Fa}$.

datatype $\mathbb{T}\,a$ **with** *fold* **has constructors:**

$tip\colon a \to \mathbb{T}\,a$

$join\colon \mathbb{T}\,a \times \mathbb{T}\,a \to \mathbb{T}\,a$

(Combinator *fold* is explained in a moment.) This declaration defines not only constructors *tip* and *join*, but also a structure $\mathbb{T}$: it is implicit in the declaration that for $f\colon a \to b$ we have $\mathbb{T}f\colon \mathbb{T}a \to \mathbb{T}b$ with the effect that $\mathbb{T}f$ subjects each tip of an a-tree to f, thus resulting in a b-tree. Moreover, structure $\mathbb{T}$ is regular. We omit further details.

Combinator *fold* is the only primitive means to define *inductive* functions on $\mathbb{T}\,a$. For example, the sum of all the tips in an *int*-tree is defined by:

$$sum \quad = \quad fold\ (id,\, +)$$

Operationally, $fold\,(f, \oplus)$ applied to a tree replaces every *tip*-node by f, and every *join*-node by $\oplus$ (and then evaluates the resulting expression).

(Actually, the datatype of lists $\mathbb{L}$ results from the datatype of trees $\mathbb{T}$ by imposing associativity of *join*, and the datatype of sets $\mathbb{S}$ results from $\mathbb{T}$ by imposing associativity, commutativity and absorptivity of *join*. Datatypes with equations for the constructors also fit into our categorical framework.)

9 Collection. Some structures might be considered a "collection", like the structure of sets, $\mathbb{S}$. In informal explanations about Moa I've come across the notion of collection many times. However, to my surprise, it turns out in §17 that this notion doesn't make much sense; in particular, $\mathbb{I}$ is as much a collection as $\mathbb{S}$ is.

C The representation of values

The essence of Moa is the representation of values by binary tables in such a way that nested sets are flattened, thus enabling nested loops to be implemented by single, unnested, loops. That is what we are going to explain in this section — without too much formality.

10 Notation. Let *oid* be the type of so-called object identities; we let i, j vary over *oid*. Let *val* denote the type of basic, unstructured, values. We assume that *oid*, *int*, and *bool* are subsets of *val*.

11 BAT. A *bat* (binary association table) is a relation, often a function, from *oid* to *val*:

$$bat \quad = \quad oid \leftrightarrow val \quad (= \ \mathbb{P}(oid \times val)\ )$$

Our visualisation of a typical value of type *bat* is:

i_0	v_0
$\vdots$	$\vdots$
i_{n-1}	v_{n-1}

We will frequently encounter a bat whose domain is a single oid, and whose range is a set of oid's:

i	i_0
$\vdots$	$\vdots$
i	i_{n-1}

Of course, in a practical implementation such a bat may be more efficiently stored as the tuple $(i, \{i_0, \ldots, i_{n-1}\})\colon oid \times \mathbb{S}\,oid$. For the sake of elegance in our formula's we refrain from doing that.

12 Flattening. The representation of a 'nested collection' of type $\mathbb{S}\mathbb{S}\cdots\mathbb{S}(int \times bool)$, say, will be something of type $bat\times(bat\times\cdots\times(bat\times(bat\times bat)))$, in such a way that the part '$(bat \times bat)$' contains *all* tuples of all innermost sets. Thus the *nesting* of several $\mathbb{S}$'s is *flattened*. The flattening is done to improve the efficiency of a manipulation with all innermost components from $\mathbb{S}\mathbb{S}\cdots\mathbb{S}\,(int\times bool)$ simultaneously. Moreover, each set on each level is *also* present in such a way that a simultaneous manipulation with all components of fixed, intermediate, level can be done quite efficiently too.

13 The algorithm. We explain the representation (called 'the bat representation') by sketching the construction of the representation for values of type $\mathbb{S}\mathbb{S}\cdots\mathbb{S}(int \times bool) \times \mathbb{S}\cdots\mathbb{S}\,int$ by a series of steps, proceeding inside-out. Values of structured types are represented with their own *handle*, an oid, which is a kind of indirection. For simplicity in the pictures, we choose to represent values of basic types

without a handle. Representing basic values with an indirection, too, gives presumably nicer formulas but some pictures below would become too large to fit on a line.

Step 0. The bat representation of a basic value x: *int* is the value itself:

$$x$$

So, values of type *int* are represented by values of type *int*. Similarly for boolean values.

Step 1. The bat representation with handle i of a tuple (x, y): *int* $\times$ *bool* consists of two bats, associating i to the representations of x and y, respectively:

$$\left(\begin{array}{c|c} \hline i & x \end{array}\ ,\ \begin{array}{c|c} \hline i & y \end{array} \right)$$

So, values of type *int* $\times$ *bool* are represented by values of type *bat* $\times$ *bat*.

Step 2. Consider the set $\{(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})\}$ of type $\mathbb{S}$(*int* $\times$ *bool*). Take the bat representations of the individual elements of the set, say with handles $i_0, \ldots, i_{n-1}$:

$$\begin{array}{l} \left(\begin{array}{c|c} \hline i_0 & x_0 \end{array}\ ,\ \begin{array}{c|c} \hline i_0 & y_0 \end{array} \right) \\ \vdots \\ \left(\begin{array}{c|c} \hline i_{n-1} & x_{n-1} \end{array}\ ,\ \begin{array}{c|c} \hline i_{n-1} & y_{n-1} \end{array} \right) \end{array}$$

Then the bat representation with handle i of the set as a whole is obtained by adding a new bat in front (associating i to the handles of the individual elements) and uniting the left components, and also the right components, of the bats of the individual elements:

$$\left(\begin{array}{c|c} \hline i & i_0 \\ \vdots & \vdots \\ i & i_{n-1} \end{array}\ ,\ \left(\begin{array}{c|c} \hline i_0 & x_0 \\ \vdots & \vdots \\ i_{n-1} & x_{n-1} \end{array}\ ,\ \begin{array}{c|c} \hline i_0 & y_0 \\ \vdots & \vdots \\ i_{n-1} & y_{n-1} \end{array} \right) \right)$$

So, values of type $\mathbb{S}$(*int* $\times$ *bool*) are represented by values of type *bat* $\times$ (*bat* $\times$ *bat*).

Step 3. Consider a set of type $\mathbb{S}\mathbb{S}$(*int* $\times$ *bool*), with the set of the previous step as one of its elements. Then the bat representation with handle j is, again, obtained by adding a new bat (associating j to the handles of the individual elements), and, *per component in the representation*, uniting the bats of all elements:

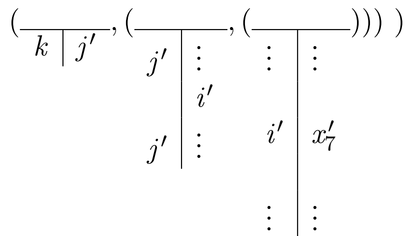
$$\left(\begin{array}{c|c} \hline j & \vdots \\ \vdots & i' \\ j & i \\ \vdots & i'' \\ j & \vdots \end{array}\ ,\ \left(\begin{array}{c|c} \hline \vdots & \vdots \\ i & i_0 \\ \vdots & \vdots \\ i & i_{n-1} \\ \vdots & \vdots \end{array}\ ,\ \left(\begin{array}{c|c} \hline \vdots & \vdots \\ i_0 & x_0 \\ \vdots & \vdots \\ i_{n-1} & x_{n-1} \\ \vdots & \vdots \end{array}\ ,\ \begin{array}{c|c} \hline \vdots & \vdots \\ i_0 & y_0 \\ \vdots & \vdots \\ i_{n-1} & y_{n-1} \\ \vdots & \vdots \end{array} \right) \right) \right)$$

Note that, if the component set is empty ($n = 0$), handle i disappears from the second bat, but it stays present in the first one! So, values of type $\mathbb{S}\mathbb{S}$(*int* $\times$ *bool*) are represented by values of type *bat* $\times$ *bat* $\times$ (*bat* $\times$ *bat*).

Step 4. It may be clear by now how to proceed: each $\mathbb{S}$ in the value type gives a (*bat*$\times$) in the representation type. Thus values of type $\mathbb{S}^m$ (*int* $\times$ *bool*) are represented by values of type (*bat*$\times$)^m (*bat* $\times$ *bat*).

Step 5. Finally consider a tuple structure at the outermost level: $\mathbb{S}^m$ (*int* $\times$ *bool*) $\times$ $\mathbb{S}^n$ *int*. The bat representation of the tuple with handle k consists of a tuple of bats (associating k to the handles of the left and right argument, respectively), each component being accompanied by the representation of the respective argument. The picture below is for the case that $m = n = 2$:

$$\left(\left(\begin{array}{c|c} \hline k & j \end{array}, \left(\begin{array}{c|c} \hline j & \vdots \\ \vdots & i \\ j & \vdots \end{array}, \left(\begin{array}{c|c} \hline \vdots & \vdots \\ i & i_3 \\ \vdots & \vdots \end{array}, \left(\begin{array}{c|c} \hline \vdots & \vdots \\ i_3 & x_3 \\ \vdots & \vdots \end{array}, \begin{array}{c|c} \hline \vdots & \vdots \\ i_3 & y_3 \\ \vdots & \vdots \end{array} \right) \right) \right) \right), \right.$$



Thus values of $\mathbb{S}^m\,(int \times bool) \times \mathbb{S}^n\,int$ are represented by values of type $(bat\times)(bat\times)^m\,(bat\times bat) \times (bat\times)(bat\times)^n\,()$. The right component might be a bit unexpected; it is caused by the fact that values of basic type, like x'_7, are represented by themselves and not with an indirection, like (x_3, y_3) having handle i_3. Type $()$ is the so-called unit type, sometimes called *void*; it holds true that $bat \times ()$ equals or is isomorphic to just *bat*.

14 Labelling. In the previous type $(bat\times)(bat\times)^m\,(bat\times bat) \times (bat\times)(bat\times)^n\,()$, some factors $(bat\times)$ originate from structure $\mathbb{S}$ whereas others originate from $\times$. In order to know how to interpret each factor $(bat\times)$, we should have labelled various constituents in the representation. Taking the structure names themselves as label, and writing them as a prefix superscript, the representation of the value discussed in Step 5 above then reads as in Figure 1. Alternatively, it suffices to label the entire value only, with the full type from which it originates.

15 Formalisation. The sketch above is clear enough to know, for each particular case, what the bat representation of a value will be if its type is built from basic types, $\times$, and $\mathbb{S}$ alone. A precise and elegant (one-line!) formulation of the representation algorithm is needed when properties of the representation have to be proven formally. Since the latter is not our intention here, we omit the formal representation algorithm.

D Generalisation—exploiting categorical notions

16 Research issues. Now our categorical notion of *structure* comes into play. Suppose we have a value whose type is built from arbitrary structures F rather than the fixed $\times$ and $\mathbb{S}$; how does its representation look like? Does $\mathbb{S}$ play a particular role, or can it be treated as any other structure? What about structures F for which the user has a particular representation structure F' in mind? Can a structure F have two representations F' and F''? What about disjoint union + (and arbitrary binary structures) instead of the particular binary structure $\times$? Should the unary structure $\mathbb{I}$ be treated the same way as the binary structure $\times$, or can we do better (and, if so, in what sense)? What functions can be computed efficiently on a bat representation? How can we characterise those functions? What can we say about the improved efficiency (on a suitable level of abstraction)?

For really general answers category theory might provide helpful concepts. By way of experiment, to show how categorical notions might help (at least, they did inspire me!), we provide initial answers to most of these questions, thereby presenting some ideas that are currently (July 2000) new to the Moa community.

17 On the notion of collection. I claim that arbitrary structures can be treated in the spirit of "the collection structure" $\mathbb{S}$. I will demonstrate this by re-doing the example of the previous section, but now with $\mathbb{I}$ instead of $\times$ at the outermost level. Recall the bat representation of two values of type $\mathbb{S}^m\,(int \times bool)$ in type $(bat\times)^m(bat \times bat)$ with $m = 2$ as given in Figure 2. Recall that the representation of the *tuple* of such values was obtained by a tuple of new bats (associating the new handle to j and j', respectively), *each component* being accompanied by the representation of the respective argument.

Now we propose (or rather, we *define*) the representation with handle k to be a new pair (again associating the new handle k to j and j', as before), *as a whole* accompanied by the componentwise union of the representations of the argument values (which makes sense since they are of the same type — thanks to the use of $\mathbb{I}$ instead of $\times$): see Figure 3. Thus values of type $\mathbb{I}\,\mathbb{S}^m\,(int\times bool)$ are represented in $(\mathbb{I}\,bat\times)\,(bat\times)^m\,(bat \times bat)$ rather than in

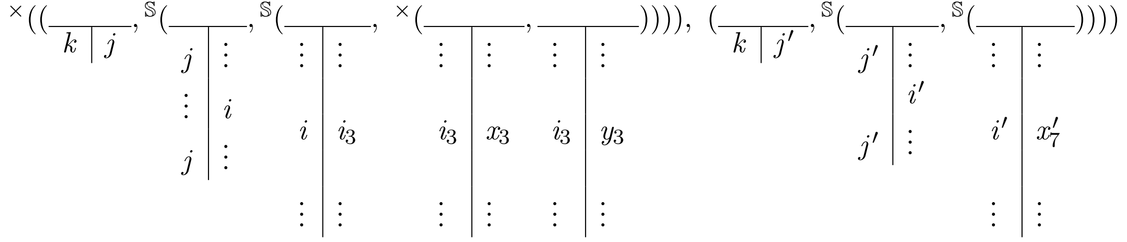


Figure 1: The labelled value representation in Step 5.

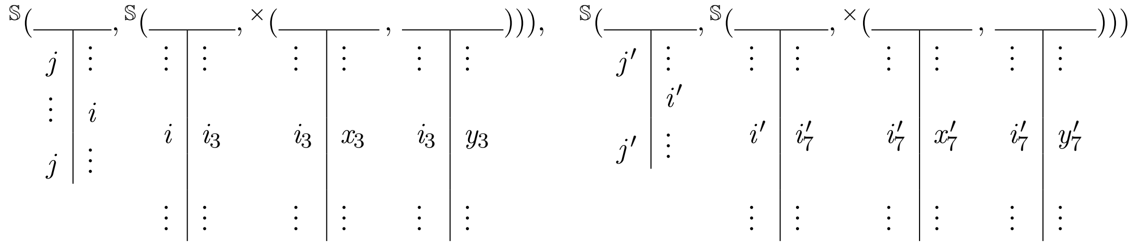


Figure 2: The representation of two equally typed values.

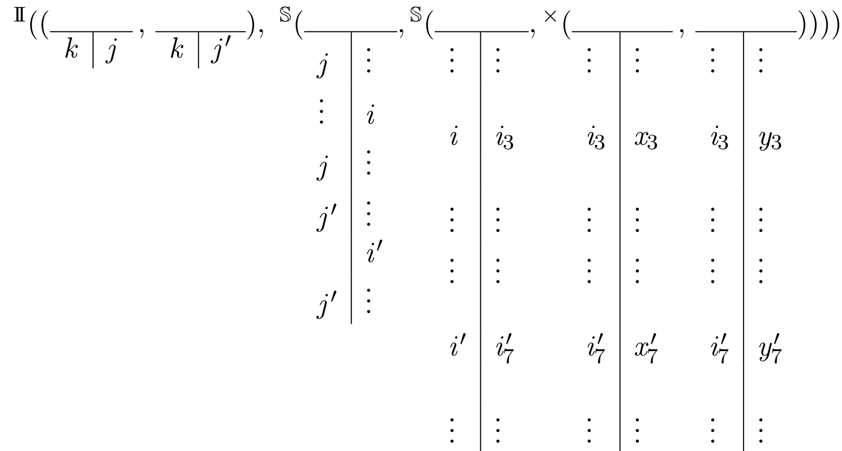


Figure 3: The representation of the pair of values from Figure 2.

$\mathbb{I}\!\mathbb{I}\,(bat\times)\,(bat\times)^m\,(bat\times bat)$. The advantage over the latter representation is that a manipulation with *all* innermost components simultaneously can be done in one go!

It appears that this trick can be done with arbitrary unary structure F. In other words, values of type $F\,\mathbb{S}^m\ldots$ will be represented in $(Fbat\times)(bat\times)^m\ldots$ rather than in $F(bat\times)(bat\times)^m\ldots$, thereby facilitating simultaneous manipulations of all innermost components in one go. So, there is no need for a notion of "collection"; the crux is that for a representation of a value of an n-ary structure $F(a_0,\ldots,a_{n-1})$ all bats of the a_0-components are united, and similarly for a_1 through a_{n-1}.

Generalising a bit further, we have that each elementary F in a (regular?!) structure gives rise to $(F'\,bat\times)$ in the representation. The representation structure F' may be taken equal to F (as we did for $\mathbb{I}\!\mathbb{I}$), but the user of the system may also define his own way of representing F-structured values (as we will do for $\mathbb{T}$, below in §19). For the set structure $\mathbb{S}$, the representation structure $\mathbb{S}'$ equals I: sets are flattened by default.

18 Dualisation. Roughly said, *dualisation* is the systematic interchange of the source and target compoment in each type, and of the left and right operand in each composition. Dualisation of a categorically defined concept gives another categorically defined concept, and dualisation preserves the validity of categorically provable statements. Without further elaboration of the principle, we only consider one example here: Cartesian product and disjoint union are dual to each other. Without giving any argument, the fact that they are dual to each other might be clear from the following declarations:

datatype $a \times b$ **with** $_ \vartriangle _$ **has destructors:**

$exl\colon a \times b \to a$

$exr\colon a \times b \to b$

datatype $a + b$ **with** $_ \triangledown _$ **has constructors:**

$inl\colon a \to a + b$

$inr\colon b \to a + b$

These declarations, under the interpretation of types-as-sets, amount to the following definitions. For Cartesian product:

$$\begin{array}{lcl} a \times b &=& \{x\colon a, y\colon b \mid (x,y)\} \\ f \times g &=& \lambda(x,y) \bullet (f\,x, g\,y) \\ f \vartriangle g &=& \lambda \quad x \quad \bullet (f\,x, g\,x) \\ exl &=& \lambda(x,y) \bullet x \\ exr &=& \lambda(x,y) \bullet y \end{array}$$

Functions exl and exr are the extractions of the left and right component. For disjoint union:

$$\begin{array}{lcl} a + b &=& \{x\colon a \bullet (0,x)\} \cup \{y\colon b \bullet (1,y)\} \\ f + g &=& (\lambda(0,x) \bullet (0, f\,x)) \cup (\lambda(1,y) \bullet (1, g\,y)) \\ f \triangledown g &=& (\lambda(0,x) \bullet f\,x) \ \cup \ (\lambda(1,y) \bullet g\,y) \\ inl &=& \lambda\, x \bullet (0,x) \\ inr &=& \lambda\, y \bullet (1,y) \end{array}$$

Functions inl and inr are the injections into the left and right summand of the union. Function $f \triangledown g$ is a case distinction with branches f and g. However, it is *only* by considering the proper categorically expressed definitions of these concepts that the duality may become really clear; we omit that for brevity. We have already defined $\mathbb{I}\mathbb{I}\, x = x \times x$ (for both types x and functions x). Similarly, we define structure $\mathit{2}$ by $\mathit{2}x = x + x$ (for both types and functions x).

We now show informally how the principle of dualisation suggests a representation for $\mathit{2}$. To this end, first observe that $x = (3,7)\colon \mathbb{I}\mathbb{I}\, int$ is represented by two bats x', x'' in such a way that $\{x\} = \mathrm{ran}(x' \vartriangle x'')$:

$$\begin{array}{c|c} \hline i & 3 \end{array} \qquad \begin{array}{c|c} \hline i & 7 \end{array} \qquad \{x\} = \mathrm{ran}(x' \vartriangle x'')$$

Dualising the latter right-hand side gives $\{x\} = \mathrm{dom}(x' \triangledown x'')$. This suggests that the representation with handle i of $x = inl\,3\colon \mathit{2}\,int$ consists of the following two bats x' and x'':

$$\begin{array}{c|c} \hline 3 & i \end{array} \qquad \begin{array}{c|c} \hline \ & \ \end{array} \qquad \{x\} = \mathrm{dom}(x' \triangledown x'')$$

Maybe more convincing is the representation of a set, say $x = \{inl\,3, inr\,5, inl\,7\}\colon \mathbb{S}\,\mathit{2}\,int$ by the following two bats x' and x'':

$$\begin{array}{c|c} \hline 3 & i \\ 7 & i \end{array} \qquad \begin{array}{c|c} \hline 5 & i \end{array} \qquad x = \mathrm{dom}(x' \triangledown x'')$$

Indeed, as expected, $x = \mathrm{dom}(x' \triangledown x'')$. Note also that a manipulation with all elements in the left summand of x (here: $inl\,3$ and $inl\,7$) simultaneously, can be done in one go, and similarly for those in the right summand (here: only $inr\,5$).

It seems that with this representation, structure $\mathit{2}$ can be dealt with in the general setting of bat representations.

19 Binary trees. (This paragraph is hardly inspired by or dependent on category theory.) Consider the following value of type $\mathbb{T}$, the type of binary trees having values only at the tips; the parenthesized subscripts define the handles of the join and tip nodes that we use later:

$$join_{(i_0)} \begin{cases} join_{(i_1)} \begin{cases} tip_{(i_a)}\ a \\ tip_{(i_b)}\ b \end{cases} \\ join_{(i_2)} \begin{cases} join_{(i_3)} \begin{cases} tip_{(i_c)}\ c \\ tip_{(i_d)}\ d \end{cases} \\ tip_{(i_e)}\ e \end{cases} \end{cases}$$

On intuitive grounds, we propose the following bat representation with handle i for that tree:

nodes		depth		parent		tips		value		index	
i	i_0	i_0	0	i_1	i_0	i	i_a	i_a	a	i_a	1
i	i_1	i_1	1	i_2	i_0	i	i_b	i_b	b	i_b	2
i	i_2	i_2	1	i_3	i_2	i	i_c	i_c	c	i_c	3
i	i_3	i_3	2	i_a	i_1	i	i_d	i_d	d	i_d	4
i	i_a	i_a	2	i_b	i_1	i	i_e	i_e	e	i_e	5
i	i_b	i_b	2	i_c	i_3						
i	i_c	i_c	3	i_d	i_3						
i	i_d	i_d	3	i_e	i_2						
i	i_e	i_e	2								

The conventional manipulations with trees can now be done efficiently (or without losing too much efficiency) with the Monet engine.

Taking $\mathbb{VI}$ to be the 6-fold Cartesian product, so that $\mathbb{VI}x = x \times x \times x \times x \times x \times x$, we have that each $\mathbb{T}$ in a value type gives rise to $(\mathbb{VI}\mathit{bat}\,\times)$ in the representation type. This is an efficient user-defined alternative to the default representation in which a $\mathbb{T}$ in a value type gives rise to $(\mathbb{T}\mathit{bat}\,\times)$ in the representation type.

20 Efficiently computable functions. Apart from representing *values* by bats, it is also Moa's task to implement value *manipulations* in terms of the bat representation. Two value manipulations in particular can be implemented very efficiently: flattening and nested loops. This is because by construction the bat representation of a nested set contains a flattened form of the set; see Step 2 and 3 of §13 and see also §17. We will elaborate this remark a little, starting with precise definitions of nesting and flattening.

A *nested loop* is a function of the form $\mathbb{SS}f$. The straightforward implementation of such a nested loop traverses a set of sets, applying f to each element of each set of the set-of-sets. *Flattening*, denoted $\cup/$, is the operation that transforms a set of sets to one big set, containing precisely all elements of the constituent sets:

$$\cup/ \quad = \quad \lambda\,\{x_0,\ldots,x_{n_1}\} \bullet x_0 \cup \ldots \cup x_{n-1}$$

Nested loops and flattening are related by the following law:

$$\mathbb{S}f \circ \cup/ \quad = \quad \cup/ \circ \mathbb{SS}f$$

Reading from right to left, the law says how to eliminate a nested loop. In the context of Monet, the obtained efficiency improvement is that the overhead for dealing with nesting has gone (some bookkeeping costs but especially the costly retrieval of data that is stored non-consecutively on disk). A single loop '$\mathbb{S}f$' can be executed very efficiently since Monet is optimised for traversing bats. Note, moreover, that the cost for flattening is null (since the bat representation contains flattenings by construction).

The fact that flattening and nesting of loops come for free, in the bat representation, facilitates several more efficiency improvements, involving aggregations, as we show now. The *aggregation* combinator $_/$ transforms a binary operation $\oplus$ to a unary operation $\oplus/$ on sets, which $\oplus$-s all elements together:

$$\oplus/ \quad = \quad \lambda\,\{x_0,\ldots,x_{n-1}\} \bullet x_0 \oplus \cdots \oplus x_{n-1}$$

Here it is required that operation $\oplus$ is of type $a \times a \to a$ for some type a (and that $\oplus$ is associative, commutative and absorptive); the typing is then $\oplus/ \colon \mathbb{S}\,a \to a$. A special case is flattening, $\cup/ \colon \mathbb{SS}\,a \to \mathbb{S}\,a$. An important law for aggregation says that f can be shifted down through a $\oplus$-aggregation (while changing it to a $\otimes$-aggregation) provided that it can already be shifted through operation $\oplus$ itself down to the arguments (while changing the operation similarly to $\otimes$):

$$f \circ \oplus/ = \otimes/ \circ \mathbb{S}f \quad \text{provided} \quad f \circ (\oplus) = (\otimes) \circ \mathbb{II}f \quad \text{and} \quad f(\nu_\oplus) = \nu_\otimes$$

Here, $\nu_\oplus$ stand for the neutral element of operation $\oplus$. For example, we have $2^{k+\cdots+m+n} = 2^k \times \cdots \times 2^m \times 2^n$, since $2^{m+n} = 2^m \times 2^n$ and $2^1 = 0$. The law has the following special cases, the first one of which we've already seen above:

$$\begin{aligned}
\mathbb{S}f \circ \cup/ &= \cup/ \circ \mathbb{SS}f \\
\oplus/ \circ \cup/ &= \cup/ \circ \mathbb{S}(\oplus/) \\
\cup/ \circ \cup/ &= \cup/ \circ \mathbb{S}(\cup/) \\
(p\lhd) \circ \cup/ &= \cup/ \circ \mathbb{S}(p\lhd)
\end{aligned}$$

In the latter line, $p\triangleleft$ is the function that selects all elements of a set that satisfy predicate p. When read from right to left, these special cases show a way to eliminate nested loops, nested aggregations, nested flattening, and nested selection, thus improving the efficiency of the function's implementation. It is Moa's task to apply these optimisations.

The definition of the aggregation combinator $_/$ given above was specific for sets, but can be generalised for (almost) any datatype. For example, consider the definition of the tree structure $\mathbb{T}$ by means of the datatype declaration in §8. Here we can also define an aggregation combinator $_/$:

$$\oplus/ \;=\; \mathit{fold}\,(\mathit{id}, \oplus)$$

Again it is required that $\oplus\colon a \times a \to a$ for some type a, and then we have $\oplus/\colon \mathbb{T}\,a \to a$. Remember, in this way we have already defined the aggregated sum over binary trees, and the special case is now $\mathit{join}/\colon \mathbb{T}\,\mathbb{T}\,a \to \mathbb{T}\,a$, which flattens a tree of trees to one big tree. This phenomenon is quite general; a wide class of datatypes come equipped with an aggregation combinator.

(The following claim is possibly wrong...) Apart from the flattening that comes for free in the bat representation, we also have that "mapping over a structure" is very efficient because of the internals of Monet's engine. This is expressed in our theory as follows. Consider an arbitrary (regular?) structure $F = GH$, possibly with $G = F$ or $G = I$, and an arbitrary function $f\colon Ha \to b$. Then the "mapping" $Gf\colon Fa \to b$ (doing f on each Ga-constituent of an Fa-value) can be computed efficiently on the bat representation: the overhead for doing the G-combinator is almost nought (in Monet).

21 Structural transformations. Let F and G be arbitrary structures. A *structural transformation* t from F to G, denoted $t\colon F \dot{\to} G$ (and called *natural transformation* in category theory), is a function from $F\,a$ to $G\,a$, for all types a (hence called 'polymorphic'), that *only reshapes* its argument, independent of the particular *values* of the argument's constituents. To formalise this latter property, recall that Ff and Gf simply apply f to each constituent of their arguments. Therefore, the property says that Ff before the transformation has the same effect as Gf afterwards. So, the two defining properties for being a structural transformation read:

$$\begin{array}{ll} t\colon F\,a \to G\,a & \text{for all } a \\ t \circ Ff \;=\; Gf \circ t & \text{for all } f \end{array}$$

I think that structural transformations will play an important role in the formal proofs of correctness claims about the representation. Indeed, if F in the type of a value gives rise to $(F'\,\mathit{bat}\,\times)$ in the representation type, then we expect that this is only correct if there is a structural transformation from F to $(F'\,\mathit{bat}\,\times)$. Manipulations on values can thus be translated to manipulations on the bat representations.

E Conclusion

Category theory provides several notions (and a lot of knowledge about those notions) that seem to be just the right ones for a Theory of Moa. Much work need to be done to elaborate the ideas given so far. And there a lot more aspects of Moa and the Moa-Monet combination that deserve study.

22 Related work. A lot of work has already been done on Moa and the Moa-Monet combination. However, some of this has not been published, but is only documented in internal working documents. Foremost of all should be mentioned the Moa system itself; designed by Annita Wilschut, it runs with a graphical interface under Unix, with Jan Flokstra as main implementor. Moa is described as a layer on top of Monet by Boncz *et al.* [4]. Several draft papers were under construction by Annita Wilschut [18, 16, 17]. A description of a typing system for Moa is under construction by Maurice van Keulen [11].

The categorical approach to program construction originates from work by Bird and Meertens since 1987, and later Backhouse and his team; the approach is described in a semi-formal setting by Meijer *et al.* [14], and more formally in a series of PhD theses by Malcolm [12, 13], Fokkinga [7, 5, 6, 8], de Moor [15], Jeuring [10]. Recently Grust [9]

has followed this approach for studying query optimisation, and a categorical theory for Moa will have close connection with his work. The categorical approach to programming has by now evolved to *generic* programming [1].

References

[1] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. In S. Doaitse Swierstra, Pedro R. Henriques, and Jose N. Oliveire, editors, *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1999. Obtainable by `http://www.cs.uu.nl/~johanj/publications/portugal.ps`.

[2] M. Barr and C. Wells. *Category Theory for Computing Science.* Prentice Hall, 1990.

[3] P. A. Boncz, W. Quak, and M. L. Kersten. Monet and its Geographical Extensions: a Novel Approach to High-Performance GIS Processing. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, volume 1057 of *Lecture Notes in Computer Science (LNCS), Springer-Verlag*, pages 147–166, Avignon, France, June 1996. Obtainable via `http://www.cwi.nl/htbin/ins1/publications`.

[4] P. A. Boncz, A. N. Wilschut, and M. L. Kersten. Flattening an Object Algebra to Provide Performance. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 568–577, Orlando, FL, USA, February 1998. Obtainable via `http://www.cwi.nl/htbin/ins1/publications`.

[5] Maarten M. Fokkinga. Datatype laws without signatures. *Mathematical Structures in Computer Science*, 6(1):1–32, 1996. Obtainable by `ftp://ftp.cs.utwente.nl/pub/doc/Parlevink/fokkinga/mmf91h.ps.Z`.

[6] M.M. Fokkinga. Calculate categorically! *Formal Aspects of Computing*, 4(4):673–692, 1992. Obtainable by `ftp://ftp.cs.utwente.nl/pub/doc/Parlevink/fokkinga/mmf91j.ps.Z`.

[7] M.M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992. Obtainable by `ftp://ftp.cs.utwente.nl/pub/doc/Parlevink/fokkinga/mmfphd.ps.Z`.

[8] M.M. Fokkinga. Abstracte datatypen en categorie-theorie. Memoranda Informatica 94-32, University of Twente, June 1994. Obtainable by `ftp://ftp.cs.utwente.nl/pub/doc/Parlevink/fokkinga/mmf94e.ps.Z`.

[9] Torsten Grust. *Comprehending Queries.* PhD thesis, Universität Konstanz, Germany, September 1999. (Ask `Torsten.Grust@uni-konstanz.de` for a copy).

[10] J.T. Jeuring. *Theories for Algorithm Calculation.* PhD thesis, Dept Computer Science, Utrecht University, The Netherlands, 1993.

[11] Maurice van Keulen. A formalisation of MOA. Working document, October 1999.

[12] G. Malcolm. *Algebraic Data Types and Program Transformation.* PhD thesis, University of Groningen, The Netherlands, 1990.

[13] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–280, September 1990.

[14] E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA91: Functional Programming Languages and Computer Architecture*, volume 523 of *Lect. Notes in Comp. Sc.*, pages 124–144. Springer Verlag, 1991. Obtainable by `http://www.cs.utwente.nl/~fokkinga/mmf91m.ps`.

[15] O. de Moor. *Categories, Relations and dynamic programming.* PhD thesis, Oxford University Computing Laboratory, Programming Research Group, April 1992. Also: Technical Monograph PRG-98.

[16] Annita Wilschut. MOA, an extensible data Model and Object algebra. Working document, October 1998.

[17] Annita Wilschut and Peter Apers. MOA: an open complex model to efficiently support a wide variety of database applications. Working document, March 1999.

[18] Annita N. Wilschut and Jan Flokstra. Flattening an object datamodel to provide functionality: the implementation of topology in Magnum. Working document, February 1998.