Object Oriented Mathematics

Klaus Grue*

December 1, 1995

Abstract

This paper shows that OO principles can be used to enhance the rigour of mathematical notation without loss of brevity and clarity.

It is well known that traditional mathematical notation is not completely formal. This is so because mathematicians and other users of mathematical notation tend to sacrifice exactness to obtain brevity and clarity. The mathematician thereby leaves to the reader to guess the meaning of each formula presented based on the written and unwritten rules of the particular field of research. This works perfectly well for communication between researchers in the same field, but may be an obstacle for communication between researchers form different fields or for newcomers such as students. The lack of rigour in mathematical notation may also be an obstacle when mathematical phenomena are to be simulated on computers, where the programmer has to fill out the gaps in the notation.

It is generally believed that complete formal rigour leads to an explosion in the size of formulas. The present paper, however, shows that a great amount of ambiguity can be removed from mathematical notation by keeping the notation but defining mathematical objects as objects in the OO sense. Examples from several different fields will be given.

Contents

*DIKU, University of Copenhagen, Denmark

1 Introduction

Computers demand unambiguous notation, programmers want brevity and maintainers of software want clarity. Users of mathematical notation demand brevity and clarity, but the demand for unambiguous notation is less pressing in mathematics. Hence, development of unambiguous notation has mainly occurred in the fields of mathematical logic and computer science, with computer science as the place where unambiguity is most needed. It is therefore no surprise that computer science has developed a number of formalisms (i.e. programming languages) that are completely unambiguous.

In many respects, mathematical notation is superior to computer science ditto. This is so because mathematics has had longer time to develop its notation and because computer science has been restricted to a character set with 96 characters and typewriters that could merely arrange those characters as simple, linear strings. The present paper aims at combining the best from the two worlds.

Such a combination of notation from two worlds will necessarily offend both worlds as not all properties of each world will be included or even appreciated.

Section 2 will present a number of ambiguities that appear in contemporary mathematical notation. Section 3 will present the choices that were made in the development of the notation. Section 4 develops the notation itself.

2 Examples of problems

2.1 Sum

A term like

$$A + B$$

may denote many different things. If A and B are integers, then the $+$ denotes integer addition, and if A and B are matrices, then the $+$ denotes matrix addition. Similarly, a statement like

$$A + B = B + A$$

may denote many different things, depending on what A and B might be. if $\mathbf{Z}$ and $\mathbf{M}(m, n)$ denote the sets of integers and m by n real matrices, respectively, then

$$\forall A, B{\in}\mathbf{Z}{:}\, A + B = B + A$$

states that addition of integers commutes whereas

$$\forall A, B{\in}\mathbf{M}(2,2){:}\, A + B = B + A$$

states that addition of 2 by 2 matrices commutes. In general, in the statement

$$\forall A, B{\in}S{:}\, A + B = B + A$$

the kind of addition in play depends on the set S. In OO terms this dependency is easy to explain: An integer like 2 or 117 must contain somewhere inside it an addition operation, and in a term like

$$2 + 117$$

the addition operation is fetched from 2 or 117. In a term like

$$\forall A, B{\in}\mathbf{Z}{:}\, A + B = B + A$$

all elements of $\mathbf{Z}$ contain an operation that does integer addition. This is in contrast to the traditional mathematical approach where addition is attached to the set $\mathbf{Z}$ rather than to each of its elements. As an example, in the traditional approach one might study the structure $(\mathbf{Z}, +)$ tacitly meaning that the given operation should be used for addition of integers in the formulas appearing afterwards.

2.2 Product

The product operator $\times$ yeilds more striking results in that

$$A \times B$$

denotes vector cross product if A and B are 3-dimensional vectors and Cartesian product if they are sets. Furthermore, $A \times B$ denotes the ordinary product of A and B if A and B are integers or real numbers. As examples,

$$\forall A, B{\in}\mathbf{Z}\colon A \times B = B \times A$$

is true whereas

$$\forall A, B{\in}\mathbf{R}^3\colon A \times B = B \times A$$

is false. In the former statement, $\times$ denotes integer multiplication, and in the latter it denotes vector cross product. Again, in OO terms, there is no problem in describing what happens: Integers as well as vectors have a cross product operation, but that of integers is ordinary integer multiplication and that of vectors is vector cross product.

2.3 Juxtaposition

If v is a vector and T is a transformation from vectors to vectors, then

$$2v$$

denotes the product of 2 and v whereas

$$Tv$$

denotes T applied to v. This indicates that objects must have a juxtaposition operation and that AB denotes the juxtaposition operation of A applied to A and B. The juxtaposition operation of integers is integer multiplication whereas that of functions is functional application. Note that this does of course not mean that 23 means two times three. Digits are "synthetic "characters in the sense that if they are placed right next to each other, then they synthesise a new token rather than being combined by the juxtaposition operation. In natural language and traditional computer science, also alphabetic characters are synthetic. In the present paper, roman characters are synthetic and italics are not so that $\log(x)$ is the function "log" applied to x whereas $log(x)$ is the juxtaposition of l, o, g, and x.

2.4 Arrows

If A and B are sets then

$$A \rightarrow B$$

denotes the set of functions from A into B. In many expositions, if U and V are vector spaces, then

$$U \rightarrow V$$

denotes the vector space of linear functions from U to V. In OO terms it is still easy to describe what happens: Sets are sets of objects, but sets are also objects in their own right and may contain operations that operate on the set as a whole. The arrow operation is an example of an operation on a set. The arrow operation is also reasonable to define for structured sets like vector spaces. In category theory arrows are also defined for structures that have not resemblance with sets.

2.5 Functional objects

Functions are objects just like sets. As an example, if U and V are vector spaces and if $f, g \in (U \rightarrow V)$ then

$$f + g$$

denotes the function h for which $h(x) = f(x) + g(x)$. This can be achieved by defining the $\rightarrow$ operation of vector spaces such that when U and V are vector spaces, all elements of $U \rightarrow V$ have $+$ operations that behave as above. Hence, it is reasonable to let the functions in $U \rightarrow V$ have $+$ operations so it is reasonable to let the functions in $U \rightarrow V$ be objects.

Another example is composition. If U, V and W are vector spaces, if $f \in U \rightarrow V$ and $g \in V \rightarrow W$, then

$$g \circ f \in U \rightarrow W$$

The composition operation $\circ$ has to be an operation of f or g. Composition is more than just functional composition as can be seen in

$$\forall f, f' {\in} U \rightarrow V \forall g {\in} V \rightarrow W \colon g \circ (f + f') = (g \circ f) + (g \circ f')$$

In this statement, the second plus operation has to be an operation of $g \circ f$ or $g \circ f'$ which indicates that composition not merely composes the functions but also defines an addition operation for the resulting object.

(Again, in category theory, composition is also defined for structures that have no resemblance with functions).

2.6 Summary

The above examples show that if numbers, sets, functions etc. are interpreted as objects in the OO sense, then a lot of the ambiguity in ordinary mathematical notation disappears. It remains to justify that this approach is at all mathematically sound.

3 Choices of notation

3.1 Strength of foundation

The development of an object oriented mathematics can be based on a strong foundation such as set theory [14] or map theory [9] or a weak one like first order predicate calculus [14], category theory [3] or lambda calculus [5, 2, 16, 17] ("strong" and "weak" with respect to consistency power). In a weak foundation it is possible to prove statements like "if real numbers exist, then addition of real numbers commute". In strong foundations it is furthermore possible to prove statements like "real numbers do exist".

The only drawback of stronger theories is that their consistency is harder to prove (actually, this is the formal definition of "stronger"). In the following, ZFC set theory (and the existence of an inaccessible ordinal [14]) is assumed consistent, and under this assumption, strong theories have benefits only. In this light it is chosen to base object oriented mathematics on a strong theory.

Set theory is a strong foundation building on first order predicate calculus and map theory is a strong foundation building on lambda calculus. Attempts have been made to establish a strong theory based on category theory [12]. Category theories that include set theory are of course strong, but they just loan power from set theory and do not really add anything to set theory. The object oriented mathematics presented here includes category theory in a natural way, but is not based on it.

3.2 Computability

Set and map theory have the same formal power but map theory has an advantage that could be expressed as a slogan like "computable functions are computable". As an example, if the definition of integer addition in set theory is written out in its full horror, then it contains a large number of universal and existential quantifiers and contains absolutely no indication of how two integers can actually be added. The definition of integer addition in map theory is not particularly easy to read, but it has the advantage that it contains no quantifiers and is directly executable by machine. More generally, arbitrary recursive definitions are legal in map theory where set theory merely allows restricted recursion schemes.

The object oriented mathematics presented in the following is based on map theory and has the advantage that computable functions are directly computable. Computation may even be efficient provided that elementary operations like addition are not actually executed on basis of their definitions but are executed directly by hardware.

A description of the version of map theory used here is given in [7]. A description that aims at first year university students is given in [10]. The original version of map theory was given in [9] and a semantic description is given in [4].

3.3 Notational freedom

A tradition of mathematics is that each author has notational freedom. On the contrary, computer science has the tradition that the equivalent of an author, namely a programmer, can merely choose names of variables and functions. Furthermore, mathematicians arrange their formulas as two-dimensional patterns with subscripts, superscripts and more advanced arrangements of characters such as matrices. Programmers can merely arrange their programs as simple linear strings of characters. These limitations in computer science are due to the lack of laser printers or similar in the early days of computer science and have no justification today (see [10] for a language that offers notational freedom to the programmer).

3.4 Operations

The central contribution from OO to object oriented mathematics is the idea to include operations in objects where the traditional approach in mathematics is to associate operations to sets of objects. This central contribution can be stated in few words and may seem simple, even trivial, but that is normal in fundamental mathematics. As an example, the notion of a set is even simpler but is however quite important. Even though the central contribution is simple, it is still non-trivial to work out the details. The reward of working out the details is that mathematical notation becomes less ambiguous, category theory is captured by the resulting theory, and it becomes possible to integrate teaching in OO with teaching of mathematics. Since the object oriented mathematics presented here is based on map theory which in turn is based on lambda calculus, it furthermore becomes possible to integrate teaching in OO and mathematics with teaching of functional programming.

3.5 Types

All attempts so far to introduce strong typing in fundamental mathematics have been notorious failures (though influential failures) [19, 15]. Hence, no strong typing scheme will be enforced upon object oriented mathematics. The typing in object oriented mathematics will be more like that in smalltalk [1] and lisp [13, 18]. In programming languages, types are convenient for catching programming errors and for speeding up the resulting code, as is the rationale for types e.g. in [11]. Hence, types should be seen as a practical convenience rather than a fundamental concept, and types seems to play no role in fundamental mathematics.

3.6 Inheritance

The notion of inheritance will not play a central role in object oriented mathematics because of the lack of strong typing. Inheritance will, however, occur in the form of functions on objects, and will be of great convenience when constructing objects later in this paper.

3.7 State

Objects in OO can have a "state" and, correspondingly, objects in object oriented mathematics have a "value". However, the value of an object in object oriented mathematics cannot change with time. As an example, the result of $2+3$ is not that the value of 2 changes to 5. Rather, the result of $2+3$ is a new object whose value of 5.

This is a general and quite convenient property of mathematics that objects do not change value with time. Even temporal logic is time independent in the sense that e.g. a theorem in temporal logic is true forever even if the theorem talks about temporal relationships. At the end of the paper, the standard example with a bank account is worked out in object oriented mathematics just to show that object oriented mathematics can deal with more programming like topics than matrices and vector spaces. However, as will be seen, withdrawing money from a bank account results in a bank account with a new value rather than changing the old value.

4 Notation

4.1 Basics

This section presents a number of concepts that are available in map theory. All concepts in map theory are ultimately defined from five basic concepts as described in [7, 10]. In the present paper, a long list of concepts like pairs, integers etc. will be taken as granted.

In this section, a distinction will be made between "OO constructs" and "raw constructs". As an example,

117

will denote an OO integer which has a value, a type, and a number of operations, whereas

$$\underline{117}$$

will denote a raw integer which has no such structure. Later, a "value" operation will be introduced and, as an example, the value of the OO integer 117 will be the raw integer $\underline{117}$:

$$\mathrm{value}(117) \equiv \underline{117}$$

Here, $\equiv$ denotes raw equality of map theory as opposed to the $=$ operation that comes with most objects. As another example, $x +_{\mathbf{Z}} y$ will be assumed to denote raw addition of raw integers whereas $x + y$ will be defined later to denote general addition:

$$\begin{aligned} 2+3 &\equiv 5 \\ \underline{2} +_{\mathbf{Z}} \underline{3} &\equiv \underline{5} \end{aligned}$$

4.2 Logic

$\underline{T}$ and $\underline{F}$ will denote raw truth and raw falsehood. In map theory, truth and falsehood are values just like e.g. integers and differential manifolds. The raw connectives $\underline{\wedge}$, $\underline{\vee}$, $\underline{\neg}$, $\underline{\Rightarrow}$ and $\underline{\Leftrightarrow}$ have their usual properties such as

$$\underline{T} \mathbin{\underline{\wedge}} \underline{F} \equiv \underline{F}$$

and

$$\underline{\neg F} \equiv \underline{T}$$

Likewise, the raw quantifiers $\underline{\forall}$ and $\underline{\exists}$ have their usual properties such as

$$(\underline{\forall} x \underline{\in} \underline{\mathbf{Z}} \underline{\exists} y \underline{\in} \underline{\mathbf{Z}} : y =_{\mathbf{Z}} x +_{\mathbf{Z}} \underline{2}) \equiv \underline{T}$$

where $=_{\mathbf{Z}}$ is equality on integers. The raw if-then-else construct

$$p \left\langle \begin{matrix} x \\ y \end{matrix} \right.$$

has the properties

$$\underline{T} \left\langle \begin{matrix} x \\ y \end{matrix} \right. = x \text{ and } \underline{F} \left\langle \begin{matrix} x \\ y \end{matrix} \right. = y$$

4.3 Bottom and exception

$\perp$ is the value of an infinitely looping expression and $\bullet$ is the value of an expression that gives rise to a "run time error" within finite time. As an example, if

$$n! \doteq n = 0 \left\langle \begin{matrix} 1 \\ n \cdot (n-1)! \end{matrix} \right.$$

then $(-1)! \equiv \perp$ since computation of $(-1)!$ loops indefinitely. On the contrary, the value of $1/0$ is $\bullet$. The value $\bullet$ is an OO value as will be seen later. The value $\perp$ has no sympathetic properties.

The general properties of the raw if-then-else construct are

$$\underline{T} \left\langle \begin{matrix} x \\ y \end{matrix} \right. \equiv x \qquad \perp \left\langle \begin{matrix} x \\ y \end{matrix} \right. \equiv \perp \quad \text{and} \quad p \left\langle \begin{matrix} x \\ y \end{matrix} \right. \equiv y \text{ if } p \not\equiv \underline{T} \text{ and } p \not\equiv \perp$$

4.4 Binary trees

The construct $x\text{'}y$ denotes the raw pair of x and y (as opposed to the OO pair defined later). The operations $z^{\underline{h}}$ and $z^{\underline{t}}$ denote the head and tail of z, respectively. In particular,

$$(x\text{'}y)^{\underline{h}} \equiv x \qquad (x\text{'}y)^{\underline{t}} \equiv y$$

A construct is said to be a raw binary tree if it is built up from raw truth and raw pairs only. As an example,

$$((\underline{T}\text{'}\underline{T})\text{'}(\underline{T}\text{'}(\underline{T}\text{'}\underline{T})))$$

is a raw binary tree. Equality $=_B$ on raw binary trees is defined by

$$x =_B y \doteq x \left\langle \begin{matrix} y \left\langle \begin{matrix} \underline{T} \\ \underline{F} \end{matrix} \right. \\ y \left\langle \begin{matrix} \underline{F} \\ x^{\underline{h}} =_B y^{\underline{h}} \,\underline{\land}\, x^{\underline{t}} =_B y^{\underline{t}} \end{matrix} \right. \end{matrix} \right.$$

This is an example of a recursive definition. Arbitrary recursive definitions are legal in map theory. Note that the definition makes use of

$$p \left\langle \begin{matrix} x \\ y \end{matrix} \right. \equiv y \text{ if } p \not\equiv \underline{T} \text{ and } p \not\equiv \perp$$

where it should be noted that no pair $(r\text{'}s)$ equals $\underline{T}$ or $\perp$.

Binary trees will be used to represent many different things such as $\underline{F}$, $\bullet$ and raw integers. $\underline{T}$ is by definition a binary tree.

4.5 Arrays

The concept of a "raw array" and the constructs $\langle\rangle$, $x\langle y\rangle$ and $x\langle y \mapsto z\rangle$ have the following properties for all raw arrays x, binary trees y, y' and arbitrary values z:

$\langle\rangle$ is a raw array
$x\langle y \mapsto z\rangle$ is a raw array
$x\langle y \mapsto z\rangle\langle y\rangle = z$
$x\langle y \mapsto z\rangle\langle y'\rangle = x\langle y'\rangle \quad \text{if } y \neq_B y'$

As an example, let

$$a \doteq \langle\rangle\langle \underline{0} \mapsto \underline{10}\rangle\langle \underline{1} \mapsto \underline{11}\rangle\langle \underline{2} \mapsto \underline{12}\rangle$$

The construct a is a raw array according to the rules above (since raw integers are binary trees). Among other, the array a has the properties

$a\langle\underline{0}\rangle$	$\underline{10}$
$a\langle\underline{1}\rangle$	$\underline{11}$
$a\langle\underline{2}\rangle$	$\underline{12}$
$a\langle\underline{3}\rangle$	$\underline{T}$

For a more detailed description of raw arrays, see [8].

4.6 Strings

For all constructs $\mathcal{A}$, $[\mathcal{A}]$ denotes a representation of the expression $\mathcal{A}$ (a "Gödel number" [14], except that in this paper, $[\mathcal{A}]$ is a raw binary tree; the exact encoding is irrelevant). As an example,

$$2 + 2 \equiv 4$$

whereas

$$[2 + 2] \not\equiv [4]$$

where the former states that $2+2$ and 4 are equal (have the same value) whereas the latter states that $2 + 2$ and 4 are different expressions. Constructs like $[+]$ and x^y are also legal and also denote binary trees. Constructs like $[x + y]$ and $[+]$ will be referred to as "strings".

4.7 Raw functions

The construct $\hat{x}\mathcal{A}.$ denotes the raw function that maps x to $\mathcal{A}$, and $f\underline{(}x\underline{)}$ denotes the raw function f applied to x. As an example, if

$$f \equiv \hat{x}x +\mathbf{z}\ \underline{2}.$$

then

$$f\underline{(}\underline{3}\underline{)} \equiv \underline{3} +\mathbf{z}\ \underline{2} \equiv \underline{5}$$

The construct $\hat{x}\mathcal{A}.$ is approximately the original notation for lambda abstraction $\lambda x.\mathcal{A}$, but Church had no $\hat{}$ on his typewriter, so he used the letter most similar to $\hat{}$ which was λ.

In map theory any raw function differs from $\underline{T}$ and $\bot$.

4.8 Objects

Objects are arrays. If a is an object then, among other, $a\langle\underline{T}\rangle$ is the raw value of a and $a\langle\underline{F}\rangle$ is the type of a:

$$\begin{array}{lcl} \text{value}(a) & \doteq & a\langle\underline{T}\rangle \\ \text{type}(a) & \doteq & a\langle\underline{F}\rangle \end{array}$$

As examples,

$$\begin{array}{lcllcl} \text{value}(T) & \equiv & \underline{T} & \text{type}(T) & \equiv & \underline{0} \\ \text{value}(F) & \equiv & \underline{F} & \text{type}(F) & \equiv & \underline{0} \\ \text{value}(117) & \equiv & \underline{117} & \text{type}(117) & \equiv & \underline{1} \\ \text{value}(\bullet) & \equiv & \underline{T} & \text{type}(\bullet) & \equiv & \underline{2} \end{array}$$

The assignment of types to objects is arbitrary. The purpose of having types is that it makes possible to define a computable construct $x =_{\mathcal{O}} y$ which states "the objects x and y have the same value and type":

$$x =_{\mathcal{O}} y \doteq \text{value}(x) =_B \text{value}(y) \mathbin{\underline{\wedge}} \text{type}(x) =_B \text{type}(y)$$

The object $\bullet$ is particularly easy to define:

$$\bullet \doteq \langle\rangle\langle\underline{F} \mapsto \underline{2}\rangle$$

4.9 Operations

If a is an object then

$$a\langle\underline{1}\text{'}[+]\rangle$$

is a plus operation on a applicable when a is the first argument of the sum and

$$a\langle\underline{2}\text{'}[+]\rangle$$

is applicable when a is the second. As an example,

$$117\langle\underline{1}\text{'}[+]\rangle$$

is an integer plus operation and

$$117\langle\underline{1}\text{'}[+]\rangle\underline{(}117\underline{)(}118\underline{)} \equiv 117 + 118 \equiv 235$$

Likewise,

$$117\langle\underline{2}\text{'}[+]\rangle$$

is an integer plus operation and

$$117\langle\underline{2}\text{'}[+]\rangle\underline{(}118\underline{)(}117\underline{)} \equiv 118 + 117 \equiv 235$$

Similarly, $a\langle\underline{1}\text{'}[\,]\rangle$ and $a\langle\underline{2}\text{'}[\,]\rangle$ are the juxtaposition operations of a. As an example, $117\langle\underline{1}\text{'}[\,]\rangle$ is integer multiplication so that if $x \equiv 3$ then

$$2x \equiv 2\langle\underline{1}\text{'}[\,]\rangle\underline{(}2\underline{)(}x\underline{)} \equiv 6$$

Integers have no operation for the situation where the integer is the second argument to a juxtaposition, which is indicated by the operation being $\underline{T}$ instead of a raw function:

$$117\langle\underline{2}\text{'}[\,]\rangle \equiv \underline{T}$$

4.10 Pedagogical ordering

Sums like $2 + \pi$ and $\pi + 2$ where 2 is an integer and π a real number posses a particular problem. It is customary to introduce integers before real numbers which gives a "pedagogical ordering" on the two concepts: Integers come before real numbers. It is reasonable to define the addition operation on integers before introducing real numbers, and therefore the addition operation on integers cannot add an integer to a real number. On the contrary, integers are known when addition of real numbers is defined, and therefore it is reasonable to define real addition such that it can cope with integers also (as will be seen later, the definition of real addition is simplified by introducing a "canonical conversion", i.e. type conversion, from integers to real numbers).

In the mathematical world, a text may be said to be "pedagogical" and in the world of computer science, a program may be said to be "readable" or "maintainable". Pedagogical, readable and maintainable are of course synonyms in this sense. A major advantage of OO is that it allows a proper pedagogical ordering of concepts in programs as well as mathematics. A common lie in teaching of OO as well as mathematics is to identify this pedagogical ordering with the temporal order in which the programmer/mathematician invents the concepts.

4.11 The definition of $x + y$

Now let $+^1_{\mathbf{Z}}$, $+^2_{\mathbf{Z}}$, $+^1_{\mathbf{R}}$, and $+^2_{\mathbf{R}}$ denote integer and real addition, i.e.

$$\begin{array}{lcl}
x +^1_{\mathbf{Z}} y & \equiv & 117\langle \underline{1}\text{'}[+]\rangle \underline{(}x\underline{)}\underline{(}y\underline{)} \\
x +^2_{\mathbf{Z}} y & \equiv & 117\langle \underline{2}\text{'}[+]\rangle \underline{(}x\underline{)}\underline{(}y\underline{)} \\
x +^1_{\mathbf{R}} y & \equiv & \pi\langle \underline{1}\text{'}[+]\rangle \underline{(}x\underline{)}\underline{(}y\underline{)} \\
x +^2_{\mathbf{R}} y & \equiv & \pi\langle \underline{2}\text{'}[+]\rangle \underline{(}x\underline{)}\underline{(}y\underline{)}
\end{array}$$

The operation $x +^2_{\mathbf{Z}} y$ should only be used if y is an integer. The operation inspects the type of x, and if it is an integer, then it adds x and y. If x is no integer then the operation gives op and returns $\underline{T}$. In other words, $x +^2_{\mathbf{Z}} y$ can be defined by

$$x +^2_{\mathbf{Z}} y \doteq \mathrm{type}(x) =_B \underline{1} \left\langle \begin{array}{l} y\langle \underline{T} \mapsto \mathrm{value}(x) +_{\mathbf{Z}} \mathrm{value}(y)\rangle \\ \underline{T} \end{array} \right.$$

The definitions of $+^2_{\mathbf{Z}}$, $+^1_{\mathbf{R}}$, and $+^2_{\mathbf{R}}$ are similar though the latter two are more complicated.

It is now possible to define $x + y$:

$$\begin{array}{lcl}
x \text{ else } y & \doteq & x \left\langle \begin{array}{l} y \\ x \end{array} \right. \\
x + y & \doteq & x\langle \underline{1}\text{'}[+]\rangle \underline{(}x\underline{)}\underline{(}y\underline{)} \text{ else } y\langle \underline{2}\text{'}[+]\rangle \underline{(}x\underline{)}\underline{(}y\underline{)} \text{ else } \bullet
\end{array}$$

As an example, $\pi + 2$ is computed as follows:

$$\begin{array}{lcl}
\pi + 2 & \equiv & \pi\langle \underline{1}\text{'}[+]\rangle \underline{(}\pi\underline{)}\underline{(}2\underline{)} \text{ else } 2\langle \underline{2}\text{'}[+]\rangle \underline{(}\pi\underline{)}\underline{(}2\underline{)} \text{ else } \bullet \\
 & \equiv & \pi +^1_{\mathbf{R}} 2 \text{ else } \pi +^2_{\mathbf{Z}} 2 \text{ else } \bullet \\
 & \equiv & \pi +^1_{\mathbf{R}} 2 \\
 & \equiv & 5.14159\cdots
\end{array}$$

Likewise, $2 + \pi$ is computed by

$$\begin{array}{rcl}
2 + \pi & \equiv & 2\langle\underline{1}\text{'}[+]\rangle\underline{(}2\underline{)(}\pi\underline{)} \text{ else } \pi\langle\underline{2}\text{'}[+]\rangle\underline{(}2\underline{)(}\pi\underline{)} \text{ else } \bullet \\
& \equiv & 2 +^{1}_{\mathbf{Z}} \pi \text{ else } 2 +^{2}_{\mathbf{R}} \pi \text{ else } \bullet \\
& \equiv & \underline{T} \text{ else } 2 +^{2}_{\mathbf{R}} \pi \text{ else } \bullet \\
& \equiv & 2 +^{2}_{\mathbf{R}} \pi \\
& \equiv & 5.14159\cdots
\end{array}$$

As can be seen, $x + y$ first tries to apply the plus operation of x, and if that fails, it tries to apply the plus operation of y. If both attempts fail, the result is an exception $\bullet$.

The computations above are not quite accurate in that the right argument of "else" is only computed if the first equals $\underline{T}$.

All operations are defined like $x+y$ above. An operation like $x \in y$, however, should have the variation that it tries the $\in$-operation of y before that of x.

4.12 Example: quantum mechanics

Even functional application (juxtaposition) should try to look for a juxtaposition operation on the second argument. As an example, if f is a real function and x is an "observable" in quantum mechanics, then it is x, not f, that has an operation that tells how f should be applied to x. Quantum mechanics is a particularly good example of a field in which the notation is so heavily overloaded that the notation is a major obstacle to understand the theory. Furthermore, quantum mechanics is a particularly good example for illustrating the benefits of an object oriented mathematics, but space does not permit a full treatment.

In quantum mechanics, things like position, time, momentum and kinetic energy are "observables". As an example of a formula, the energy H of a particle with mass m and momenta p_x, p_y and p_z is (c.f. [6], Section 30):

$$H \equiv c\sqrt{m^2c^2 + p_x^2 + p_y^2 + p_z^2}$$

In this formula, the real squaring and real square root functions are applied to observables, but observables are not real numbers. The operation for applying real functions to observables is defined in Section 11 of [6] and is a good example of pedagogical ordering: There is no need to know about observables when defining ordinary functional application. The notation in [6] is a good example of heavily overloaded notation that can be made precise by object oriented mathematics.

4.13 Canonical conversions

In mathematics as well as computer science, it is customary to "identify" objects with each other. As an example, an integer may serve as a real number. This will be represented by a "canonical conversion" operation $x{:}y$ whose value has the same type as y and represents the same value as x. As an example,

$$2{:}\pi$$

is the real number 2. If the conversion is impossible, then the value is $\underline{T}$. As an example,

$$\pi{:}2 \equiv \underline{T}$$

The canonical conversion operation is treated like any other operation.

As an example of use, $x +^1_{\mathbf{R}} y$ tries to convert y to a real number before addition:

$$\begin{array}{lcl} x +^1_{\mathbf{R}} y & \doteq & x \tilde{+}^1_{\mathbf{R}} (y{:}x) \\ x \tilde{+}^1_{\mathbf{R}} z & \doteq & z \left\langle \begin{array}{l} \underline{T} \\ x\langle \underline{T} \mapsto \mathrm{value}(x) +_{\mathbf{R}} \mathrm{value}(y) \rangle \end{array} \right. \end{array}$$

Canonical conversions are useful in many situations. As an example, if S is a set and U a vector space, then $U{:}S$ would be the vector space U interpreted as a set. Further, if U° denotes the dual vector space of U, then U and $U^{\circ\circ}$ are canonically isomorphic. Hence, if $x \in U$ and $y \in U^{\circ\circ}$, then $x{:}y$ should be the element of $U^{\circ\circ}$ corresponding to x and $y{:}x$ should be the element of U corresponding to y.

4.14 Example: a category of vector spaces

A possible representation/implementation of vectors and vector spaces is outlined in the following. Space does of course not allow a complete treatment. Hence, the presentation will take the form of a specification that states things like that a vector "shall" have certain operations with certain properties.

As will be seen, the vector spaces form a category, actually a Cartesian closed one. In the category, $U \rightarrow V$ will be the vector space of linear maps from U to V. Another category would be obtained by letting $U \rightarrow V$ be the vector space of all maps from U to V.

For vector spaces U and V, $U \times V$ will be the Cartesian product of U and V. Vector spaces W that can be expressed on the form $U \times V$ will be called product vector spaces. Vector spaces W that can be expressed on the form $U \rightarrow V$ will be called function vector spaces.

Any vector space has an associated body of scalar values. As examples, the body of a real vector space is the body of real numbers and that of a complex vector space is the body of complex numbers. Any vector shall have a body operation so that body(x) is the body of scalars that x can be multiplied with. Furthermore, any vector x shall have a right juxtaposition operation so that if $a \in \mathrm{body}(x)$ then ax is the vector x multiplied by the scalar a.

Any vector shall of course have value, type and $=$ operations, but the details are not particularly interesting here.

Any vector shall have a pair operation. As an example, if x and y are vectors with the same body, then (x, y) shall be a vector. If $z = (x, y)$ then body(z) = body(x) = body(y), value(z) = value(x)'value(y), and type(z) = [vector]'[,]'type(U)'type(V)'$\underline{T}$. Furthermore, z shall have operations z^h and z^t such that $(x, y)^h = x$ and $(x, y)^t = y$. The plus and juxtaposition operations of z shall be $\hat{u}\hat{v}(u^h + v^h, u^t + v^t)..$ and $\hat{a}\hat{x}(ax^h, ax^t)..$, respectively.

Any vector x shall have a canonical conversion operation with the property that $0{:}x$ equals the zero vector of the same kind as x, i.e. 0 times x. Any vector space U shall have a 0 operation such that 0_U is the zero vector of U.

A vector space shall have a value which is a set of vectors, a type, an $=$ operation and a body operation. The vector space shall have a $\in$ operation so that $x \in U$ means $x \in$ value(U). If U is a vector space and S is a set, then $U{:}S$ shall be equal to value(U), i.e. the vector space U interpreted as a set.

Vector spaces shall have a $\times$ operation such that if U and V have the same body then $U \times V$ is a vector space W such that value(U) $= \{(x, y) | x \in U \wedge y \in V\}$, type($W$) = [vspace]'[$\times$]'type($U$)'type($V$)'$\underline{T}$, and body($W$) = body($U$) = body($V$). The product vector space W shall have operations W^H and W^T so that $W^H = U$ and $W^T = V$.

Vector spaces shall have a $\rightarrow$ operation such that if U and V have the same body then $U \rightarrow V$ is a vector space W such that type(W) = [vspace]'[$\rightarrow$]'type(U)'type(V)'$\underline{T}$ and body(W) = body(U) = body(V). value(W) shall contain one vector for each linear map from U to V, and those vectors shall have a left juxtaposition operation that denotes functional application. As an example, if $f \in U \rightarrow V$ and $x \in U$, then $fx = f\langle\underline{1}\text{'}[\,]\rangle\underline{(}f\underline{)(}x\underline{)} \in V$ and fx denotes f applied to x.

Functional vector spaces $U \rightarrow V$ shall have a Dom and a Rng operation such that Dom($U \rightarrow V$) $= U$ and Rng($U \rightarrow V$) $= V$. Functional vectors $x \in U \rightarrow V$ shall have a dom and a rng operation such that dom(x) $= U$ and rng(x) $= V$.

All vectors x whatsoever shall have a domain operation such that domain(x) is the vector space of which x is a member, so if $x \in$ value(U) then domain(x) $= U$. In consequence, no two vector spaces can share vectors. As an example, if U' is a subspace of U, then the elements of U' shall have domain U' and those of U shall have domain U, which prevents U and U' to have any vectors in common. There shall of course be canonical conversions between vectors of U and U' if U' is a subspace of U so that elements of U' are identified with elements of U.

Having a domain operation simplifies the $\mapsto$ operation: All vector spaces U shall have a $\mapsto$ operation such that

$$x{\in}U \mapsto \mathcal{A}$$

has the following properties: Let $f = \hat{x}\mathcal{A}.$ and let $V =$ domain($f0_U$). If domain(fx) $= V$ for all $x \in U$, and if there is an element g of $U \rightarrow V$ such that

$fx = gx$ for all $x \in U$, then $x \in U \mapsto \mathcal{A}$ denotes this (unique) g. Otherwise, $x{\in}U \mapsto \mathcal{A}$ shall equal $\bullet$.

It is now possible to define id_U and $\circ$:

$$\begin{array}{lcl} \mathrm{id}_U & \doteq & x{\in}U \mapsto x \\ f \circ g & \doteq & x{\in}\mathrm{dom}(g) \mapsto f(gx) \end{array}$$

These definitions of id_U and $f \circ g$ together with Dom and Rng turn the collection of vector spaces into a category. The elements of $U \rightarrow V$ have some resemblance with functions and $f \circ g$ has some resemblance with functional composition, but the elements of $U \rightarrow V$ and the construct $f \circ g$ have much more structure than functions and functional composition. This is a typical situation in category theory, but there are also examples where elements of $U \rightarrow V$ has no resemblance with functions.

As an example of use, define

$$\begin{array}{lcl} \mathrm{base}(U) & \doteq & \mathrm{body}(U){:}U \\ U^\circ & \doteq & U \rightarrow \mathrm{base}(U) \end{array}$$

$\mathrm{base}(U)$ is the body of U interpreted as a vector space and U° is the dual of U. Now define

$$c \doteq x{\in}U \mapsto y{\in}U^\circ \mapsto yx$$

The function vector c is of type $U \rightarrow (U^\circ \rightarrow \mathrm{base}(U))$ so c is of type $U \rightarrow U^{\circ\circ}$. The function c is the canonical isomorphism between U and $U^{\circ\circ}$ which shows that two applications of the dual operation essentially yields the original vector space. This is an example of use of $\mapsto$. As another example, define

$$c' \doteq f{\in}U{\rightarrow}V \mapsto x{\in}V \rightarrow \mathrm{base}(V) \mapsto x \circ f$$

The function vector c' is of type $(U \rightarrow V) \rightarrow (V^\circ \rightarrow U^\circ)$ and is the canonical isomorphism between $U \rightarrow V$ and $V^\circ \rightarrow U^\circ$. As a last example, define

$$c'' \doteq f{\in}U{\times}V{\rightarrow}W \mapsto x{\in}U \mapsto y{\in}V \mapsto f(x, y)$$

The function vector c'' is the canonical isomorphism $(U \times V \rightarrow W) \rightarrow (U \rightarrow (V \rightarrow W))$.

The canonical conversion functions of vectors shall implement these canonical isomorphism so that if e.g. $x \in U$ and $y \in U^{\circ\circ}$, then $x{:}y = u{\in}U^\circ \mapsto ux$. Likewise, $y{:}x = \varepsilon\hat{v}v{\in}U \wedge y = cv.$ where ε is the "such that" operator of map theory and where $\varepsilon\hat{v}v \in U \wedge y = cv.$ reads "a v such that $v \in U$ and y equals cv".

4.15 Example: bank accounts

As the last example, the classical (and in OO papers almost unavoidable) example with a bank account will be treated. A simple bank account in the present setting will be represented by an object with a value, a type (which could arbitrarily be set of $\underline{118}$, a deposit operation and a withdraw operation. The deposit operation will be denoted $+$ and withdrawal will be denoted $-$ so that if x is a bank account and y is an integer (denoting an amount of cents) then $x+y$ is a new bank account whose deposit is y greater than the deposit of x and similarly for $x-y$. If y is negative then $x+y$ and $x-y$ equal $\bullet$. If y is greater than the deposit of x then $x-y$ also equals $\bullet$. The definition of a bank account of zero deposit reads:

$$\begin{array}{lcl}
\mathrm{zerobankaccount}' & \doteq & \langle\rangle\langle\underline{1}\text{'}[+] \mapsto \mathrm{deposit}\rangle\langle\underline{1}\text{'}[-] \mapsto \mathrm{withdraw}\rangle \\
\mathrm{zerobankaccount} & \doteq & \mathrm{zerobankaccount}'\langle\underline{T} \mapsto 0\rangle\langle\underline{F} \mapsto \underline{118}\rangle \\
\mathrm{deposit} & \doteq & \hat{x}\hat{y} y \geq 0 \left\langle \begin{array}{l} x\langle\underline{T} \mapsto \mathrm{value}(x)+y\rangle \\ \bullet \end{array} \right. .. \\
\mathrm{legal}(x,y) & \doteq & y \geq 0 \wedge \mathrm{value}(x) - y \geq 0 \\
\mathrm{withdraw} & \doteq & \hat{x}\hat{y}\mathrm{legal}(x,y) \left\langle \begin{array}{l} x\langle\underline{T} \mapsto \mathrm{value}(x)-y\rangle \\ \bullet \end{array} \right. ..
\end{array}$$

Next, consider an object that represents a person. Person objects will be constructed such that their value is unused and their type is arbitrarily set to $\underline{119}$. A person will have a name and an address together with two operations "setname" and "setaddress". In a person object x, the name and address will be kept in $x\langle\underline{0}\text{'}[\mathrm{name}]\rangle$ and $x\langle\underline{0}\text{'}[\mathrm{address}]\rangle$, respectively. The definition of person with no name and address reads:

$$\begin{array}{lcl}
\mathrm{noperson} & \doteq & \langle\rangle\langle\underline{F} \mapsto \underline{119}\rangle\langle\underline{1}\text{'}[\mathrm{setname}] \mapsto \mathrm{setn}\rangle\langle\underline{1}\text{'}[\mathrm{setaddress}] \mapsto \mathrm{seta}\rangle \\
\mathrm{setn} & \doteq & \hat{x}\hat{y}x\langle\underline{0}\text{'}[\mathrm{name}] \mapsto y\rangle.. \\
\mathrm{seta} & \doteq & \hat{x}\hat{y}x\langle\underline{0}\text{'}[\mathrm{address}] \mapsto y\rangle..
\end{array}$$

Section 4.5 introduced raw arrays and a number of operations on arrays. Now, one more operation on arrays, $\underline{\&}$, will be introduced. Whenever x and y are raw arrays, $x \mathbin{\underline{\&}} y$ will be a new raw array z with the following property for all binary trees t:

$$\begin{array}{ll}
z\langle t\rangle = y\langle t\rangle & \text{if } y\langle t\rangle \neq \underline{T} \\
z\langle t\rangle = x\langle t\rangle & \text{otherwise}
\end{array}$$

In other words, $x \mathbin{\underline{\&}} y$ combines the associations of x and y such that the associations in y take precedence.

Having this operation, it is easy to combine a person object and a bank account object into a bank account with owner. As an example,

$$\mathrm{noperson} \mathbin{\underline{\&}} \mathrm{zerobankaccount}$$

is a bank account with a deposit of zero and no name and address, but which has a deposit and withdraw operation as well as a setname and a setaddress operation.

The description of bank accounts above is a mathematical one but, by virtue of the computability of a subset of map theory, the above definitions are also machine computable. Among other, this gives an opportunity to establish a link between mathematics and object oriented programming in an undergraduate curriculum. Mathematical bank accounts of course have the peculiar property that any operation on a bank account gives a new bank account rather than modifying the old one. Treatment of temporal bank accounts in which the deposit is a function of time is not particularly complicated but nevertheless out of the scope of the present paper.

5 Conclusion

An "object oriented mathematics" system has been introduced. The major benefit of such a system is that it allows an exact yet notationally compact treatment of notions that are not normally treated in a mathematically precise way. Examples are given drawn from banking, quantum mechanics, category theory and tensor algebra. Possible uses of the system is to ease communication between different areas and, in particular, to make undergraduate curricula more coherent by allowing a consistent notation to be used for many different fields such as object oriented programming, functional programming and mathematics.

References

[1] Author. *Smalltalk*. Publisher, Year.

[2] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and The Foundation of Mathematics*. North-Holland, 1984.

[3] M. Barr and C. Wells. *Category Theory for Computing Science*. Series in Computer Science. Prentice-Hall, 1990.

[4] C. Berline and K. Grue. A simple semantic consistency proof for map theory, based on ξ-denotational semantics. To Appear.

[5] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[6] P. A. M. Dirac. *The Principles of Quantum Mechanics*, volume 27 of *The International Series of Monographs on Physics*. Oxford Science Publications, fourth edition, 1958.

[7] K. Grue. Map theory 93. To appear.

[8] K. Grue. Arrays in pure functional programming languages. *Lisp and Symbolic Computation*, 1(2):105–113, 1989.

[9] K. Grue. Map theory. *Theoretical Computer Science*, 102(1):1–133, July 1992.

[10] K. Grue. Mathematics and computation. Available from the author, contact grue@diku.dk, 1994.

[11] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald Programming Language Report. Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, October 1987. (Revised August 1988).

[12] Lawvere. The category of categories as a foundation for mathematics. In *Proceedings of the Conference on Categorical Algebra, La Jolla 1965*, pages 1–21. Springer, 1966.

[13] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, pages 184–195, 1960.

[14] E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth and Brooks, 3. edition, 1987.

[15] W. V. Quine. New foundations for mathematical logic. *Amer. Math. Monthly*, 44:70–80, 1937.

[16] D. Scott. *λ-Calculus and Computer Science Theory*, volume 37 of *Lecture Notes in Computer Science*, chapter Combinators and classes, pages 1–26. Springer-Verlag, 1975.

[17] D. Scott. *λ-Calculus and Computer Science Theory*, volume 37 of *Lecture Notes in Computer Science*, chapter Some philosophical issues concerning theories of combinators, pages 346–366. Springer-Verlag, 1975.

[18] Guy L. Steele. *Common Lisp—The Language*. Digital Press, second edition, 1990.

[19] A.N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, 1910.