МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2005
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2005
Proceedings of the Thirty Fourth Spring Conference of
the Union of Bulgarian Mathematicians
Borovets, April 6–9, 2005

AN EXPERIMENT IN PROGRAM ANNOTATION

Boyko B. Bantchev

It has been for a long time, and it still remains an open question how to annotate programs. We discuss the problem and describe a notational system of our own invention that we have been using for some time. Although simple, it proves to be very useful.

Defining what we aim at. Computer programs are very complicated formal objects. Often it is difficult not only to reason about a program but even to understand, by reading, what it does. That is why it is important to properly annotate code where necessary. By 'annotating' we mean documenting that states program's requirements and obligations and clarifies its working.

Of course, efforts should be exerted to make programs as self-evident as possible, but this has its limits. Even very short programs, no matter how carefully written, may be impossible to comprehend without accompanying explanations; this is inevitable. The more algorithmically complex a program is, the more it needs annotation.

How do we write code annotations? We certainly want to be as explicit as possible but without being too wordy, for then the code itself would be lost in the accompanying explanations and the effort of making them would be doubtful. Hence, what we need is a formal notation that we could use along with our natural language.

Such a formal notation can most naturally be provided by the written tradition of mathematics. However, for various reasons that will be mentioned later on, borrowing from traditional mathematical notation (TMN) without modifying it may not be just what we need.

The question of what and how to write is heavily dependent on the related question of reading it. So, how do we read code annotations? In the simplest case, annotations reside in the file with the program itself, and are being read as they are, e. g. using a text editor. If other media are provided, it becomes possible to use more sophisticated notation, where different fonts, special symbols etc. play role. We mention several approaches in the following section.

Some approaches to program annotation. It is remarkable that some of the most eminent computer scientists of the past century took serious pains to devise systems that would allow them to explain programs or reason about them! They realized that notation is an important tool of thinking and not only a means of its expression.

D. Knuth proposed a method for writing programs together with their documentation which he called *literate programming* [1]. The programmer creates chunks of text where code and documentation are intermixed. Then he, by use of a special transforming program, extracts code that he can run through the compiler, while, using another transforming program, from the same chunk he extracts a documentation file that can be TEXed. Parts of the program code can be put in the documentation as needed.

This method is attractive with its bringing the whole power of the TEX document production system to the programmer: the latter can not only format his text beautifully, but also write formulae and draw diagrams. The drawback is the necessity to know how to use and to actually use some additional programs, TEX itself included. At present, this is still a hurdle for most programmers.

While Knuth's idea was to embed code in typographically rich documents, thus making possible to use TMN for documenting programs, others' efforts were directed toward improving the formal (mathematical) notation itself. E. Dijkstra, whose manuscripts were almost all really hand-written by him (all of them are available in 'electronic' form at [2]) devised his own notation for mathematical writing, which he used to derive programs and to reason about them. In this respect, it can be seen as a system for program annotation. Dijkstra wrote a partial rationale for his notation [3]. Our own notation, that we describe in the next section, borrows much from Dijkstra's.

K. Iverson, earlier than all—in the 60s—recognized the inadequacies of the TMN for computing-related use, and created a revolutionary new language to use in its stead ([4] and many of his other works). That language eventually grew into a programming language (APL) that itself spawned successors (A+, J, K). Iverson's work was deeply influential to programming, especially to the functional kind of it.

Other notational systems also exist or keep emerging.

Z [9] is a rich, well respected notation for formal specification of programs. Certainly it can be used to annotate programs, provided mathematical writing is possible.

MATHS [5] is a notation for reasoning about programs, which its author uses in his development of methodology for software engineering. As it only uses plain text (no special symbols, all writing is linear), it can be adopted as a vehicle for inline program annotation.

Finally, there are several attempts at creating languages that are able to express mathematical content and can be rendered in a graphically appropriate (more or less traditional) way in the Web. For example, the WWW Consortium proposed the XML-based language MathML for representing mathematical content in the Web [8]. OpenMath [6] is a somewhat similar project, but more oriented toward content (as opposed to syntax, which is the primary concern of MathML). MINSE [7] is another hierarchical and using plain text (but not XML-based) language for expressing mathematics that can be, by means of special software residing on a server, rendered as conventional mathematical formulae in HTML pages.

All such Web-targeted notations, to the extent they are really supported by appropriate software, are attractive candidates for program annotating tools, in view of the nowadays omnipresence of the Web and browsers for it.

A rationale and a proposal. One particular reason that makes us want to annotate programs is when those programs are used for teaching algorithms to other people.

For us, that was the main stimulus to start a search for a good program annotation system.

The said situation is characterized by some additional conditions that must be taken care of. If we wish not to rely on any external tool for annotating our algorithms, it is best of all that those annotations be presented in the simplest form: as a linear, ordinary symbol text. That is, all the text is plain strings of characters (no multilayered formulae, subscripts etc.) and those characters are the ones that we find on our keyboards. The latter precludes using $\sqrt{\ }$, $\sum$, greek alphabet etc. Thus it is possible to keep annotations within the program code itself, in the form of comments.

It follows that we need a (semi-)formal, mathematical notation but representable as plain text. In fact, we are compelled to deviate from TMN not only because of the representational restrictions that we willingly adopt. Not less important is that TMN is not fully adequate to represent some objects and notions, essential for programming, such as sequences and operations on them.

One more reason to abstain from directly borrowing from TMN is that the latter is partly arbitrary and irregular, as well as inexplicit and ambiguous: features we would better get rid of.

For example, in TMN, most monadic operators are prefixed to their arguments, but some are not. The factorial (!) is a suffix, and absolute value ($|\,|$), floor ($\lfloor\,\rfloor$) and square root ($\sqrt{\ }$) have their specific ways of representation.

We are not more fortunate with the dyadic operators. Some of them are infix (addition, subtraction), others ($\sqrt[x]{y}$) are idiosyncratic, and still others do not have symbols to represent them at all. TMN represents multiplication, and often function application, as juxtaposition. This can lead to ambiguity or impossibility of expression. (What is fx? Is it $f \times x$, $f(x)$ or just a two-letter name? Or how do we tell whether p in $p\,(q + r)$ is a function applied to an argument or a number multiplied to another one?) Equally bad is the lack of a symbol for raising to a power, for then we are unable to tell, in say a^i, a power from a superscript for summation.

TMN is ambiguous not only through being inexplicit, as the above examples show, but also by denoting different objects with the same symbols. An example of this is the braces notation ($\{\}$), used to denote both the fractional part of a number and a set enumeration (so we cannot distinguish between the former and a singleton).

The traditional mathematician does not usually feel bothered by idiosyncrasies or irregularities of the notation he uses, and can remove ambiguity by creating context, which in most cases he does informally. The programmer, annotating a piece of code, however, must be as succinct as possible, so he must use natural language sparingly.

To summarize, the language that we need for annotating code must be

- sufficiently formal: most of what we would want to write must be expressible in formulae,
- human-readable,
- adequate enough for description of algorithms and related objects,
- unambiguous,
- succinct, i.e. allow laconic expression,
- written as linear, ordinary text.

The listed features rule out most of the known notations, for them being either too much oriented towards TMN, or because they are not intended for human processing, or because they are weak at describing computing notions. This includes most of the above mentioned approaches. Others, such as Iverson's notation, are worth borrowing from, but as a whole are both restrictive and difficult in use for our purposes.

Our experience has lead us to a form of writing that is mostly similar to the one of Dijkstra, with several notable changes and additions. In the following, we take a brief tour of what it comprises.

Overview. The types of data that can be dealt with include numbers, booleans and sets—both unordered and ordered. Ordered sets are so important for programming that we prefer to use a separate word for them—sequences—leaving the word 'set' for the unordered case. Besides arithmetical operations, comparisons and set-theoretic operations, there are some known functions and some special forms of function applications. Names can be bound to values or to newly defined functions. We also use existential and universal quantifiers as needed.

Here is a summary of the basic operations and functions.
Arithmetic: `+`, `-`, `*`, `/`, with the usual meaning. In addition, we use `-` and `/` also for negation and reciprocal, respectively.
Numeric comparisons: `<`, `>`, `<=`, `>=`, and `=` and `~=`, the latter two being used for expressing equality or inequality of non-numeric values as well as numeric.
Boolean operators: `,` (conjunction), `|` (disjunction), `~` (negation) and `=>` (implication).
Functions: `abs`, `min`, `max`, `ceiling`, `round` etc. Also, type-checking predicates, such as `int`, `real`, `set` and `seq`, for integers, reals, sets and sequences.
Binding is denoted with a colon, in the form

name `:` *expression* .

or, if a name is already bound, that binding can be changed by a 'compound assignment' operation, denoted by *operator*`:`, e. g. `+:` .

Multiple binding is possible through pattern matching (see below).

The colon is also used to place a *constraint* on a value. The general form of this is

value `(`*scope*`)` `:` *boolean expression*

where *scope* can be the name of a data type, or that of a set, or a boolean expression. The short forms *value*`(`*scope*`)` and *value* `:` *boolean expression* are also admitted. Often, the *value* is just a name. Examples of constrained expressions are

```
x(real)
z(3<=z<n):'z prime'
(x*y):(x+y=n)  .
```

Note that boolean expressions can be specified informally, as in the second line above. Note also that the parentheses `( )` have no role in our notation other than to show a scope or to delimit an expression in the context of a larger one, as in the third line above.

The universal and existential quantifiers are written

`A.`*name*`(`*scope*`)`:*boolean expression*
`E.`*name*`(`*scope*`)`:*boolean expression*

The *name* is the bound variable. The *scope* and *boolean expression* parts have the same meaning as for the constrained expressions above, and similarly the scope (and only it) can be omitted. For example, the expression `A.x(J,x>0):x>5` is true iff all positive elements of `J` are greater than 5.

Function application. It is important for the consistency of a notation that all operations have visually explicit representations. Function application is no exception, and we, following Dijkstra, use the `.` (dot) operator to denote it. Thus, we write e. g. `abs.x` or `abs.(i-j)`.

`//` is our operator of functional composition. It is associative and binds stronger that the `.` operator, which itself is left-associative, so e. g. `f//g//h.x.y` reads `((f//g//h).x).y`.

Functions can be applied partially—a useful concept first introduced by Schönfinkel and Curry and now present in the functional programming languages. For example, `+.2` is the function that increments a single argument by 2, so `+.2.z` is the same as `2+z`. One thing partial application is convenient for is function definition. For the just given example, we could have defined `add2` with `add2: +.2` and then used it as in `add2.z`. In general, partial application is a useful form of function abstraction that provides a way to generate new functions from given ones.

Other forms of function application arise in connection with set-valued arguments, as will be seen in the next subsection.

Sets and sequences. Set union, intersection and complement are represented in our notation as `\/`, `/\` and `~` (same as boolean negation). Set difference is `\`.

The operators `+` and `-` add/subtract a value to/from a set. (These are different from `\/` and `\`: `{a,b,c}\/{b}` is `{a,b,c}` while `{a,b,c}+{b}` is `{a,b,c,{b}}`.)

The operators `@`, `<@` and `<=@` check for element inclusion and strict and non-strict set inclusions. The `#` operator gives the cardinality of a set (or the length of a sequence).

We use curly braces for explicitly enumerated sets, e. g. `{i,j,{},{m,n}}`. The empty set is `{}`. A constrained expression used inside braces provides a set generator, so `{x(M):~set.x}` denotes the set of those elements of `M` that are themselves not sets.

There are also incomplete forms of set enumeration, such as:

{.}, {,}, {,,} etc.	sets of exactly one, two, three etc. elements
{...}	a set of any, including 0, number of elements
{,...}	a set of at least one element
{z,...}	a set of one or more elements including z
{[10]}	a set of exactly 10 elements.

The notation for sequences is similar to the one for sets but angular brackets `< >` are used. However, there are much more forms for explicit enumeration of sequences than there are for sets. Here are some examples:

`<z,...>`	a sequence that starts with `z`
`<...,x,y>`	a sequence that ends with `x` and `y`
`<[2],k,[5]>`	a sequence of exactly 8 elements, where the third one is `k`
`<,p,,q,...,7,>`	a sequence of at least six elements where the second and the fourth ones are `p` and `q`, and the last but one is `7`.

The structural patterns like these can be nested in order to reflect data structure at deeper levels.

An important use of these patterns is in providing a means for structure-lead, multiple-name binding. When used on the left-hand side of a binding operator, a structural pattern must not contain any literal values, only names. These names then get bound by pattern matching—a mechanism resembling the one present in some functional programming languages. For example, `<x,<,z>,...> : <{12,a},<8,b>,34,{18}>` is equivalent to `x:{12,a}; z:b` (all other values on the right are discarded in the matching process).

Special notation is provided for arithmetic progressions and index sequences. An expression, such as `[1,20]`, denotes the sequence of the integers from 1 to 20, while `[2,5,16]` is equivalent to `<2,5,8,11,14>`.

An integer progression with a step 1 is called an index sequence. Index sequences can be semi-open, such as in `[10,20)`, where the term `20` is excluded.

Index sequences can be used to index other sequences. For example, `a[0,n)` denotes a sequence `a` whose entries are numbered from `0` to `n`. Individual entries in an indexed sequence are referred to using the operator for function application, e. g. `a.3`, `a.k`, `a.(2*j+1)`.

An indexed sequence can be constrained by the values of its elements, or by their indices, or both. For example, in a sequence `a[0,n)`, a segment of length `q-p` such that any two of its elements are within distance `d` is defined as:

```
a[p,q): (0<=p<q<=n, A.<i,j>(p<=i<j<q): abs.(a.i-a.j)<=d) .
```

On sets and sequences, function application has a special meaning: it is what is sometimes called a *reduction*. For example, `+.{a,b,c}` and `+.s` (provided `s` is a set) are the sums of `{a,b,c}` and of `s`, respectively. Any binary operator is equally applicable in place of `+`. There is also a form of reduction with `.\` instead of `.`, the difference being that on sequences `.\` applies from right to left. Thus, if `q[1,n]` is a sequence, `-.\q` is the alternating sum `q.1-q.2+...+(-1)^(n-1)*q.n`.

Besides reduction, a *mapping* or *zipping* operation can be applied on a set or a sequence. A mapping operation is denoted similarly to function application, but with `./` in place of `.` . It transforms each element of a set or a sequence by applying some operator on it; thus `op./s`, where `op` is an operator and `s` is a set, is the same as `{op.x(x@s)}`, and similarly for sequences.

Zipping is written `||` or `|`*op*`|`, where *op* is a dyadic operator. The first form just 'zips' two sequences into a sequence of pairs, or produces a Cartesian product of two sets. The second form applies *op* to each such pair, thus producing a result of the same shape as its arguments.

For example, `{1,2,3}||{x,y}` gives `{<1,x>, <1,y>, <2,x>, <2,y>, <3,x>, <3,y>}`, while `<1,2,3>||<x,y,z>` gives `<<1,x>, <2,y>, <3,z>>`, and `<1,2,3> |+| <x,y,z>` amounts to `<1+x,2+y,3+z>`. As another example, consider `|+|.q` which does a reduction with `+`: provided `q` is a sequence of sequences, the result is the element-wise sum of the `q`'s entries.

As a final example, we present the description of Prim's algorithm for finding a minimum-weight spanning tree of a weighted graph. Starting from a single-vertex tree, the algorithm repeats the following: add a minimum-weight edge among those that augment the tree (an edge augments a tree when it is linked to precisely one of the tree's vertices). It stops when the tree includes all the vertices of the given graph. We assume that the graph has a set of vertices `V` and a set of edges `E`, and the result is the set of edges `T` (the set of vertices of the tree `T` need not be represented as it is `V`).

```
(initialize)
   T:{};  N:v@V
(repeat)
   s: {{x,y}(E):(x@N,~y@N)}
   e(s): weight.e=min.weight/.s
   T+: e;  N\/: e
(until)
   N=V
```

Concluding remarks. The system for program annotating that we present is an attempt to systematically design a notation for the stated purposes. It is one that we apply and try to improve according to our perception of the necessities of the programmer's work.

There are several directions for improvement that we currently observe. One of them is extending the notation to include specific means for describing hierarchies (rooted trees). We are tempted to think that it is possible to describe basic operations and structural patterns in hierarchies in a similar way that we do with sets and sequences. Another, rather challenging problem is to find a way to embed domain-specific notations. For example, a vector algebra embedding would make possible to annotate geometric algorithms using the adequate language.

REFERENCES

[1] The literate programming web site. `http://www.literateprogramming.com`.

[2] Archive of manuscripts of Edsger W. Dijkstra. `http://www.cs.utexas.edu/users/EWD`.

[3] E. W. Dijkstra. The notational conventions I adopted, and why, EWD 1300. *Formal Aspects of Computing*, **14**, *2* (2002), 99-107.

[4] K. E. Iverson. Notation as a tool of thought. (1979 ACM Turing Award Lecture) *Comm. ACM*, **23**, *8*, (1980), 444–465.

[5] R. J. Botting. MATHS web site. `http://www.csci.csusb.edu/dick/maths`.

[6] OpenMath project web site. `http://www.openmath.org/cocoon/openmath`.

[7] MINSE project web site. `http://lfw.org/math`.

[8] MathML web site. `http://w3.org/Math`.

[9] J. M. Spivey. The Z Notation: A Reference Manual, 2$^{\text{nd}}$ ed. Oriel College, Oxford, 1998

Boyko B. Bantchev
Institute of Mathematics and Informatics
Acad. G. Bontchev Str., Bl. 8
1113 Sofia, Bulgaria
e-mail: `bantchev@math.bas.bg`

ОПИТ ПО АНОТИРАНЕ НА ПРОГРАМИ

Бойко Бл. Банчев

В статията се обсъжда един отдавнашен нерешен проблем: как да се анотират програми. Описва се формализъм, въведен и използван от известно време от автора като средство за анотиране. Макар и прост, той се оказва твърде полезен.