Joy: Forth's Functional Cousin

Manfred von Thun

9th October 2001

1 Synopsis of the language Joy

The language Joy is a purely functional programming language. Whereas all other functional programming languages are based on the application of functions to arguments, Joy is based on the composition of functions. All such functions take a stack as argument and produce a stack as value. Consequently much of Joy looks like ordinary postfix notation. However, in Joy a function can consume any number of parameters from the stack and leave any number of results on the stack. The concatenation of appropriate programs denotes the composition of the functions which the programs denote. One of the datatypes of Joy is that of quoted programs, of which lists are a special case. Some functions expect quoted programs on top of the stack and execute them in many different ways, effectively by dequoting. So, where other functional languages use abstraction and application, Joy uses quotation and combinators – functions which perform dequotation. As a result, there are no named formal parameters, no substitution of actual for formal parameters, and no environment of name-value pairs. Combinators in Joy behave much like functionals or higher order functions in other languages, they minimise the need for recursive and non-recursive definitions. Because there is no need for an environment, Joy has an exceptionally simple algebra, and its programs are easily manipulated by hand and by other programs. Many programs first construct another program which is then executed.

This paper is intended as an introduction for Forth programmers.

2 Introduction

To add two integers, say 2 and 3, and to write their sum, you type the program

```
2  3  +
```

This is how it works internally: the first numeral causes the integer 2 to be pushed onto a stack. The second numeral causes the integer 3 to be pushed on top of that. Then the addition operator pops the two integers off the stack and pushes their sum, 5. So the notation looks like ordinary postfix. The Joy processor reads programs like the above until they are terminated by a period. Only then are they executed. In the default mode the item on the top of the stack (5 in the example) is then written to the output file, which normally is the screen.

To compute the square of an integer, it has to be multiplied by itself. To compute the square of the sum of two integers, the sum has to be multiplied by itself. Preferably this should be done without computing the sum twice. The following is a program to compute the square of the sum of 2 and 3:

```
2  3  +  dup  *
```

After the sum of 2 and 3 has been computed, the stack just contains the integer 5. The `dup` operator then pushes another copy of the 5 onto the stack. Then the multiplication operator replaces the two integers by their product, which is the square of 5. The square is then written out as 25. Apart from the `dup` operator there are several others for re-arranging the top of the stack. The `pop` operator removes the top element, and the `swap` operator interchanges the top two elements. There are several other operators that do similar things. This idea is of course familiar to Forth programmers.

Unlike Forth, Joy has a data type of lists. A *list* of integers is written inside square brackets. Just as integers can be added and otherwise manipulated, so lists can be manipulated in various ways. The following `concat`enates two lists:

```
[1 2 3]  [4 5 6 7]  concat
```

The two lists are first pushed onto the stack. Then the `concat` operator pops them off the stack and pushes the list `[1 2 3 4 5 6 7]` onto the stack.

Joy makes extensive use of *combinator*s, even more so than most functional languages. In mainstream functional languages combinators take as arguments (lambda abstractions of) functions. In Joy combinators are like operators in that they expect something specific on top of the stack. Unlike operators they execute what they find on top of the stack, and this has to be the *quotation* of a program, enclosed in square brackets. One of these is a combinator for `map`ping elements of one list via a function to another list. Consider the program

```
[1 2 3 4]  [dup *]  map
```

It first pushes the list of integers and then it pushes the quoted squaring program onto the stack. The `map` combinator then removes the list and the quotation and constructs another list by applying the program to each member of the given list. The result is the list

```
[1 4 9 16]
```

which is left on top of the stack.

In *definition*s of new functions, as in Forth, no formal parameters are used, and hence there is no substitution of actual parameters for formal parameters. After the following definition

```
square   ==   dup  *
```

the symbol `square` can be used in place of `dup *` . As in most other programming languages, definitions may be recursive.

3 Data types of Joy

The data types of Joy are divided into simple and aggregate types. The *simple* types comprise integers, floating point, characters and the truth values. The aggregate types comprise sets, strings, lists and files. Literals of any type cause a value of that type to be pushed onto the stack. There they can be manipulated by the general stack operations such as `dup`, `pop` and `swap` and a few others, all familiar in Forth. Or they can be manipulated by operators specific to their type.

Integers and floats are written in decimal notation. The usual binary operations and relations are provided. Operators are written after their operands. Binary operators remove two values from the top of the stack and replace them by the result. Unary operators are similar, except that they only remove one value. In addition to the integer type there are also characters and truth values (Boolean), both with the usual operations.

The *aggregate* types are the unordered type of sets and the ordered types of strings and lists. Aggregates can be built up, combined, taken apart and tested for membership. A *set* is an unordered collection of zero or more small integers, 0.. 31. Literals of type set are written inside curly braces, like this `{3 7 21}`. The usual set operations are available.

A *string* is an ordered sequence of zero or more characters. Literals of this type string are written inside double quotes, like this: "Hello world".

A *list* is an ordered sequence of zero or more values of any type. Literals of type list are written inside square brackets, like this: `[peter paul mary]` or `[42 true {2 5}]`. Lists can contain mixtures of values of any types. In particular they can contain lists as members, so the type of lists is a recursive data type. The usual list operations are provided. Matrices are represented as lists of lists, and there is a small library for the usual matrix operations. To a large extent the operators on the structured types are identical. Lists are implemented using linked structures, and the stack itself is just a list. This makes it possible to treat the stack as a list and vice versa.

4 Quotations and Combinators

Lists are really just a special case of *quoted program*s. Lists only contain values of the various types, but quoted programs may contain other elements such as operators and some others that are explained below. A *quotation* can be treated as passive data structure just like a list. For example,

```
[ +  20  *  10  4  - ]
```

has size `6`, its second and third elements are `20` and `*`, it can be reversed or it can be concatenated with other quotations. But passive quotations can also be made active by *dequotation*.

If the above quotation occurs in a program, then it results in the quotation being pushed onto the stack - just as a list would be pushed. There are many other ways in which that quotation could end up on top of the stack, by being concatenated from its parts, by extraction from a larger quotation, or by being read from the input. No matter how it got to be on top of the stack, it can now be treated in two ways: passively as a data structure, or actively as a program. The square brackets prevented it from being treated actively. Without them the program would have

been executed: it would expect two integers which it would add, then multiply the result by 20, and finally push 6, the difference between 10 and 4. A combinator expects a quotation on the stack and executes it in a way that is different for each combinator. One of the simplest is the `i` combinator. Its effect is to execute a single program on top of the stack, and nothing else. Syntactically speaking, its effect is to remove the quoting square brackets and thus to expose the quoted program for execution. Consequently the following two programs are equivalent:

```
[ +  20  *  10  4  - ]  i
  +  20  *  10  4  -
```

The `i` combinator is mainly of theoretical significance, but it is used occasionally. The many other combinators are essential for programming in Joy.

Some combinators require that the stack contains values of certain types. Many are analogues of higher order functions familiar from other programming languages: `map`, `filter` and `fold`. The `map` has already been explained.

Another combinator that expects an aggregate is the `filter` combinator. The quoted program has to yield a truth value. The result is a new aggregate of the same type containing those elements of the original for which the quoted program yields `true`. For example, the quoted program `['Z >]` will yield truth for characters whose numeric values is greater than that of `Z`. Hence it can be used to remove upper case letters and blanks from a string. So the following evaluates to `"ohnmith"`:

```
"John Smith"   ['Z >]   filter
```

Sometimes it is necessary to add or multiply or otherwise combine all elements of an aggregate value. The `fold` combinator can do just that. It requires three parameters: the aggregate to be folded, the quoted value to be returned when the aggregate is empty, and the quoted binary operation to be used to combine the elements. In some languages the combinator is called reduce (because it turns the aggregate into a single value), or insert (because it looks as though the binary operation has been inserted between any two members). The following two programs compute the sum of the members of a list and the sum of the squares of the members of a list. They evaluate to 10 and 38, respectively.

```
[2 5 3]  0  [+]  fold
[2 5 3]  0  [dup * +]  fold
```

5 Intermezzo: Lambda abstraction and application

The squaring function can be defined in just about any programming language. In infix notation the external definition might look similar to this:

```
square(x)   ==   x  *  x
```

Internally, and possibly externally too, the definition would be

```
square   ==   Lx :  x * x
```

where "Lx" is sometimes written "\x" or "lambda x" of "fn x". The expression on the right side of the definition then is to be read as "the function of one argument x which yields the value x * x". An expression like that is known as a *lambda abstraction.* Almost all programming languages use such a construct at least implicitly.

Such a definition would be used as in the following evaluation, where `square` is being *applied* (here using an explicit `@` infix operator) to the argument 2:

```
      square  @  2
(Lx : x * x)  @  2
      2 * 2
        4
```

In the second line `square` is being replaced by its definition. This sets up an environment in which `x = 2`, in which the formal parameter x has been given the value of the actual parameter 2. An environment is a collection of associations of formal and actual parameters. In all but the simplest cases the environment will contain several such associations. In the third line this environment is used to replace the formal parameter x by the actual parameter 2. All this is completely hidden from the user when the first style of definition is used.

The two operations of lambda abstraction and application are complementary. They form the heart of the lambda calculus which underlies all the mainstream functional and imperative languages. Joy eliminates abstraction and application and replaces them by program quotation and function composition.

6 Doing without abstraction and environments

The previous definitions of the squaring function as a lambda abstraction might be written

```
square   ==   Lx :  x  x  *
square   ==   Lx :  x  dup  *
```

For the second definition the right hand side has to be read as "the stack function for stacks in which there is a top element `x` and which for stacks without that top element yields the same result as `x dup *`". But one might also write without the explicit lambda abstraction as

```
x  square   ==   x  x  *
x  square   ==   x  dup  *
```

In the last of these both sides start with the formal parameter x, and otherwise there are no further occurrences of x on either side. Would it be possible for the two occurrences on the left and right to "cancel out", so to speak? Yes indeed, and now the definition looks like this:

```
square   ==   dup  *
```

The mainstream imperative languages have a *state* of associations between assignable variables and their current values. The values are changed by assignments during the run of the program. These languages also have an environment of formal / actual parameter associations

that are set up by calls of defined functions or procedures. Purely functional languages have no state. But the mainstream functional languages are based on the lambda calculus, and hence they have an environment. The purely functional language Joy has no state and no environment. The imperative language Forth has both.

As in other languages, definitions can be recursive in Joy. In the first line below is a recursive definition of the factorial function in one of many variants of conventional notation. In the second line is a recursive definition in Joy.

```
factorial(x)  =  if x = 0 then 1 else x * factorial(x - 1)
factorial  ==  [0 =] [pop 1] [dup 1 - factorial *] ifte
```

Again the Joy version does not use a formal parameter `x`. It works like this: The right side of the definition pushes three quotations, called the if-part, the then-part and the else-part. Then the if-then-else combinator `ifte` sets these aside and exposes the numeric argument that is now on top of stack. >From here on it behaves just like conditional expression in the conventional version.

7 Recursive Combinators

If one wanted to compute the list of factorials of a given list, this could be done by

```
[ factorial ]  map
```

But this relies on an external definition of factorial. It was necessary to give that definition explicitly because it is recursive. If one only wanted to compute factorials of lists of numbers, then it would be a minor nuisance to be forced to define factorial explicitly just because the definition is recursive.

A high proportion of recursively defined functions exhibit a very simple pattern: There is some test, the if-part, which determines whether the ground case obtains. If it does, then the non-recursive

Joy has a useful device, the `linrec` combinator, which allows computation of anonymous functions that *might* have been defined recursively using a linear recursive pattern. Whereas the `ifte` combinator requires three quoted parameters, the `linrec` combinator requires four: an if-part, a then-part, a rec1-part and a rec2-part. Recursion occurs between the two rec-parts. For example, the factorial function could be computed by

```
[null]  [succ]  [dup pred]  [*]  linrec
```

There is no need for a definition, the above program can be used directly. To compute the list of factorials of a given list of numbers the following can be used:

```
[ [null]  [succ]  [dup pred]  [*]  linrec ]   map
```

In many recursive definitions there are two recursive calls of the function being defined. This is the pattern of *binary recursion*, and it is used in the usual definitions of quicksort and of the Fibonacci function. In analogy with the `linrec` combinator for linear recursion, Joy has a `binrec` combinator for binary recursion. The following will *quicksort* a list whose members can be a mixture of anything except lists.

```
[small]  []  [uncons [>] split]  [swapd cons concat]  binrec
```

8 Mathematical Foundations of Joy

Joy programs are built from smaller programs by just two constructors: *concatenation* and *quotation*.

Concatenation is a binary constructor, and since it is associative it is best written in infix notation and hence no parentheses are required. Since concatenation is the only binary constructor of its kind, in Joy it is best written without an explicit symbol.

Quotation is a unary constructor which takes as its operand a program. In Joy the quotation of a program is written by enclosing it in square brackets. Ultimately all programs are built from atomic programs which do not have any parts.

The semantics of Joy has to explain what the atomic programs mean, how the meaning of a concatenated program depends on the meaning of its parts, and what the meaning of a quoted program is. Moreover, it has to explain under what conditions it is possible to replace a part by an equivalent part while retaining the meaning of the whole program.

Joy programs denote functions which take one argument and yield one value. The argument and the value consist of several components. The principal component is a *stack*, and the other components are not needed here. Much of the detail of the semantics of Joy depends on specific properties of programs.

However, central to the semantics of Joy is the following, which also holds for the purely functional fragment of Forth:

> The concatenation of two programs denotes the composition of the functions denoted by the two programs.

Function composition is associative, and hence denotation maps the associative syntactic operation of program concatenation onto the associative semantic operation of function composition.

One part of a concatenation may be replaced by another part denoting the same function while retaining the denotation of the whole concatenation.

One quoted program may be replaced by another denoting the same function only in a context where the quoted program will be dequoted by being executed. Such contexts are provided by the *combinator*s of Joy. These denote functions which behave like higher order functions in other languages. There is no equivalent mechanism in Forth.

The above may be summarised as follows: Let `P`, `Q1`, `Q2` and `R` be programs, and let `C` be a combinator. Then this principle holds:

```
IF          Q1      ==      Q2
THEN     P  Q1  R   ==   P  Q2  R
AND        [Q1] C   ==     [Q2] C
```

9 Joy Algebra

The principle is the prime rule of inference for the *algebra of Joy* which deals with the equivalence of Joy programs, and hence with the identity of functions denoted by such programs. A few laws

in the algebra can be expressed without combinators, but most require one or more combinators for their expression. Concatenation of Joy programs denote the composition of the functions which the concatenated parts denote. Hence if `Q1` and `Q2` are programs which denote the same function and `P` and `R` are other programs, then the two concatenations `P Q1 R` and `P Q2 R` also denote the same function. In other words, programs `Q1` and `Q2` can replace each other in concatenations. This can serve as a *rule of inference* for *rewriting*.

As premises one needs axioms such as in the first three lines below, and definitions such as in the fourth line:

```
(+)              2  3  +   ==   5
(dup)            5  dup   ==   5  5
(*)              5  5  *   ==   25
(def square)     square  ==  dup *
```

A derivation using the above axioms and the definition looks like this:

```
                 2  3  +  square
          ==     5  squar                    (+)
          ==     5  dup  *                   (def square)
          ==     5  5  *                     (dup)
          ==     25                          (*)
```

The comments in the right margin explain how a line was obtained from the previous line. The derivation shows that the expressions in the first line and the last line denote the same function, or that the function in the first line is identical with the function in the last line.

Consider the following equations in infix notation: The first says that multiplying a number `x` by 2 gives the same result as adding it to itself. The second says that the `size` of a `reversed` list is the same as the `size` of the original list.

```
2 * x  =  x + x                size(reverse(x))  =  size(x)
```

In Joy the same equations would be written, *without variables*, like this:

```
2  *   ==   dup  +              reverse  size   ==   size
```

Other equivalences express algebraic properties of various operations. For example, the predecessor `pred` of the successor `succ` of a number is just the number itself. The conjunction `and` of a truth value with itself gives just the truth value. The less than relation `<` is the converse of the greater than relation `>`. Inserting a number with `cons` into a list of numbers and then taking the `sum` of that gives the same result as first taking the sum of the list and then adding the other number.

In conventional notation these are expressed by

```
pred(succ(x))  =  x          x and x  =  x
x < y  =  y > x              sum(cons(x,y))  =  x + sum(y)
```

In Joy these can be expressed *without variables*, using the *identity function* `id`:

```
succ  pred   ==   id        dup  and   ==   id
<  ==   swap >              cons  sum   ==   sum  +
```

Some properties of operations have to be expressed by combinators. One of these is the `dip` combinator which expects a program on top of the stack and below that another value. It saves the value, executes the program on the remainder of the stack and then restores the saved value.

In the first example below, the `dip` combinator is used to express the associativity of addition. Another combinator is the `app2` combinator which expects a program on top of the stack and below that two values. It applies the program to the two values. In the second example below it expresses one of the De Morgan laws. In the third example it expresses that the `size` of two lists `concat`enated is the sum of the `size`s of the two concatenands. The last example uses both combinators to express that multiplication distributes (from the right) over addition. (Note that the program parameter for `app2` is first constructed from the multiplicand and `*`.)

```
[+]  dip  +   ==   +  +
and  not   ==   [not]  app2  or
concat  size   ==   [size]  app2  +
[+]  dip  *   ==   [*]  cons  app2  +
```

10 Joy and Forth

The similarities and differences between Joy and Forth are striking and profound. They have been discussed in the mailing list, which can be found at `http://groups.yahoo.com/group/concatenative`.

The main Joy page is at `http://www.latrobe.edu.au/philosophy/phimvt/joy.html`.