Rationale for Joy, a functional language

by Manfred von Thun

Abstract: Joy is a high-level purely functional programming language which is not based on the application of functions but on the composition of functions. This paper gives a rationale for Joy by contrasting it with with other paradigms of functional languages. Joy differs from lambda calculus languages in that it has no variables and hence no abstraction. It differs from the combinatory calculus in that it does not use application. It differs from the categorical languages in *uniformly* using an untyped stack as the argument and value of the composed functions. One of the datatypes is that of programs, and the language makes extensive use of this, more than other reflective languages. The paper gives practical and theoretical introductions to various aspects of the language.

Keywords: lambda calculus, combinatory logic, lambda abstraction, function application, function composition, postfix notation

Introduction

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. Joy has a rich but simple algebra, and its programs are easily manipulated by hand and by other programs.

Joy is an attempt to design a high level programming language which is based on a computational paradigm that is different from the three paradigms on which existing functional languages are based. Two of these paradigms have been criticised in the literature, and Joy aims to overcome the complexity of the third by a simpler mechanism.

The remainder of this paper is organised as follows. The next two section of the paper assume some familiarity with three paradigms: the lambda calculus, combinatory calculus and, to a lesser extent, the basic notions of category theory. The purpose of these sections is to contrast Joy with these paradigms and to motivate the departure. The other sections are very specific to Joy and hence mostly self-contained. One section is a short tutorial introduction, another a discussion of theoretical aspects. The concluding section gives a more detached perspective.

Background of functional languages

All natural languages and most artificial languages contain as a component a functional language which allows expressions to be built up from individual symbols together with functional symbols. In appropriate interpretations the expressions have a value which is an individual. Even statements can be considered to belong here, provided we take predicates to be functions which yield a truth value. Sometimes one needs higher order functions, often called functionals or combinators which can take other functions as parameters. Higher order functions can be handled in the lambda calculus. Here functional expressions are built from

variables and constants by just two constructions. One of these is *lambda abstraction*: if `(..x..)` is an expression containing a variable, then its lambda abstraction is generally written λx`(..x..)` and pronounced "the function of one parameter x which yields the result `(..x..)`". The other construction is *application*, written in infix: if `f` is a function, then `f @ x` is the result of applying the function to x. Functions of several parameters are still a nuisance because one has to write `g @ <x,y></x,y>` and `h @ <x,y,z></x,y,z>` and so on. There is a useful device called currying, Generally attributed to Curry, but freely acknowledged by him to be due to Sch\"{o}nfinkel (1924). The term "Sch\"{o}nfinkeling" never caught on. By currying all functions can be taken to be unary. The binary application operation is still written in infix notation, and by making it left-associative some parentheses can be saved. Furthermore, since it is the *only* binary operation, the @ symbol is simply left out.

The notation makes the expression

```
+ 2 3
```

potentially ambiguous. On one reading it is a prefix expression, entirely equivalent to the infix `2 + 3` or the postfix `2 3 +`, with + as the binary operator and the two numerals as operands. On another reading it is a nested infix expression with binary application suppressed between the two numerals as the main operator and a similar subordinate suppressed operator between the curried + and the 2. In practice there is no confusion because the context disambiguates, particularly in nested expressions. Prefix never needs parentheses, that is why it was invented by Polish logicians. Applicative notation needs parentheses in more complex expressions, see section 3. Syntax aside, there is a world of difference in the semantics between prefix and *applicative* notation. A similar ambiguity will arise later. (As yet another applicative notation, to eliminate parentheses completely, Quine in his foreword to the Sch\"{o}nfinkel (1924) reprint suggested using prefix for application, thus: `@fx`, `@@gxy` and so on.)

The lambda calculus is an immensely powerful system. It comes as a surprise that it is Turing complete, that all *computable function*s can be expressed in the lambda calculus with just variables, abstraction and application, and can then be computed by reduction. Even numbers can be represented, as Church numerals, and similarly for lists. However, any efficient implementation will need constants, and all practical programming languages based on the lambda calculus provide them. This includes the older language Lisp and its descendants, based on the untyped lambda calculus, and also the newer languages ML % ML, Miranda\footnote{ % "Miranda" is a trademark of Research Software Ltd.} and Haskell, based on a typed lambda calculus with parametric polymorphism. Central to all of them are the lambda calculus operations of abstraction and application.

The lambda calculus is a purely syntactic calculus, and its rules have to do with simplifying expressions. The few rules are deceptively simple but are in fact difficult to use correctly in real examples. The main difficulty arises from the variables which get in each other's way. Their purpose is to steer arguments into the right position. Can one do without variables, to make things easier for people or for computers, and still steer arguments into the right position? Brus *et al* (1987 p 364) write "if one wants to have a computational model for functional languages which is also close to their implementations, pure lambda calculus is not the obvious choice anymore".

One alternative is combinatory calculus, also called *combinatory logic* because of its origin. The key idea came from Sch\"{o}nfinkel (1924) but was greatly expanded in Curry and Feys (1958). Variables can indeed be eliminated completely, provided some appropriate higher order functions or combinators are introduced. Most such systems use as their basis a translation scheme from the lambda calculus to a combinatory calculus which only *needs* two combinators, the S combinator and the K combinator. Abstraction is an operation in the *object language*, the lambda calculus. In combinatory logic this operation is replaced by an operation in the *metalanguage*. This new operation is called bracket abstraction, essentially a compilation. Since all lambda calculus expressions can be compiled in this manner, the language of combinators is again Turing complete. The simple compilation scheme yields translations whose length can be exponential in the length of the source expression. Using additional combinators it is possible to produce translations of acceptable lengths. The

combinators S and K can be used to define all other combinators one might desire, or even on their own to eliminate variables and hence lambda abstractions `lambda x (..x..)`. Even recursion can be handled with the "paradoxical" Y combinator which is equivalent to a (hideous) expression just in S and K. A similar y combinator in Joy is discussed in section 5. Y and y cannot be given a finite type, so they are not definable in typed languages. Joy, like Lisp, is untyped, hence it requires runtime checks.

So we can do without abstraction but with application together with first and higher order functions. The resultant system is simpler, but because it is so low level, it has never been proposed as a programming language. However it did inspire Backus (1978) in his Turing award lecture where he introduced his FP system, short for "Functional Programming system". Central to the language are *functional forms*, a small, fixed and unextendible collection of combinators operating on unary functions. A more recent reference is Backus, Williams and Wimmers (1990). Backus acknowledges a dept to combinatory logic, but his aim was to produce a variable free notation that is amenable to simple algebraic manipulation by people. Such ease should produce clearer and more reliable programs.

Motivating foundation for Joy

Like the various lambda calculus languages, the low level combinatory calculus and the higher level language FP still use application of functions to their arguments. However, as Meertens (1989 p 71) writes, "The basic problem is that the basic operation of the classical combinator calculus (and also of the closely related lambda calculus) is application instead of composition. Application has not a single property. Function composition is associative and has an identity element (if one believes in the 'generic' identity function)." Of course application is substitutive, identical arguments yield identical results, hence if `f = g` and `x = y` then `f @ x = g @ y`. But the substitutivity property is shared with all other functions. Meertens later (p 72) speaks of "the need of a suitable system of combinators for making functions out of component functions without introducing extra names in the process. Composition should be the major method, and not application." This is in fact done in category theory for the (concrete) category of functions, their compositions and their types. Like Backus, Meertens develops a system of combining functions that is more suitable to formal manipulation than the classical combinators.

Consider a long expression, here again written explicitly with the application operator @. Note the need for parentheses.

```
square @ (first @ (rest @ (reverse @ [1 2 3 4])))   -->  9
```

All the functions are unary, and unary functions can be composed. The composition of unary functions is again a unary function, and it can be applied like any other unary function. Let us write composition with an infix dot " . ". The composition can be applied to the original list:

```
(square . first . rest . reverse)  @  [1 2 3 4]     -->  9
```

One might even introduce definitions in the style of the first line, and then write as in the second line:

```
second = first . rest          second-last = second . reverse
  (square . second-last)  @  [1 2 3 4]                  -->  9
```

This and also the preceding definitions would not make sense with application instead of composition. Importantly, a definition can be used to textually replace the *definiendum* by its *definiens* to obtain the original long composition expression. This is because the textual operation of compounding several expressions to make a larger one is mapped onto the semantic operation of composing the functions denoted by the expressions. The textual replacement is not possible in the original applicative expression because the parentheses get in the way.

Substitutivity is a highly desirable property for algebraic manipulation. The only trouble is that the resultant composition expression still has to be applied to an argument, the list `[1 2 3 4]`. If we could elevate that list to the status of a function, we could eliminate application entirely from the expression and write

```
square . first . rest . reverse . [1 2 3 4]          -->  9
```

The numeral 9 would also need to denote a function. Can this be done?

Indeed it can be. We just let numerals and list constants denote functions which take a fixed dummy argument, written ?, as argument and return a number or a list as value. So we should now write

```
(square . first . rest . reverse . [1 2 3 4]) @ ?   -->   9 @ ?
```

We just have to pretend that `@ ?` is not there, that everything is a function anyhow. The dummy argument device is routinely used in the category of functions, and the pretense is argued to be a superior view.

All this works well with unary functions, but how is one to deal with functions of several arguments? In category theory there is the notion of *products*, and in the category of functions it is a way of dealing with interrelated pairs --- function pairs to produce value pairs of a type pair. (Backus in his FP used a similar mechanism, *construction* which used function tuples to produce value tuples. But the function tuples ultimately need the application operation to produce the value tuple.) Two important *projection* functions are needed for picking the first and second from a pair (`car` and `cdr` in Lisp). Pairs would seem to be the obvious way to handle binary functions. But this reintroduces pairs (of functions) whereas in the lambda calculus pairs (of arguments) were so elegantly eliminated by currying. In category theory there is also the notion of *exponentials*, and in the category of functions they are a way of dealing with the interrelation between the type of functions, the type of their arguments and the type of values. Two important functions are needed: explicit currying and explicit application (`apply` in Lisp). This makes such Cartesian closed categories second order. They are a computationally equivalent alternative to the (typed) lambda calculus and to combinatory calculus. So these categorical languages can handle functions of several argument and all higher order functions.

Barr and Wells (1990 Chapter 6) give an example of a simple lambda expression with variables contrasted with first a complicated looking and then a reformulated categorical equivalent formula. Here the steering of arguments into the right place is essentialy done by the projection functions. Category theory has given rise to another model of computation: the CAM or Categorical Abstract Machine, described for example in Cousineau *et al* (1987). The machine language of the CAM is very elegant, but programs in that language look as inscrutable as low level programs in other machine languages. The language is of course suitable as the target language for compilation from any functional language. For more recent references, including to exciting hardware applications, see Hains and Foisy (1993).

Many categorical concepts have been successfully used in otherwise applicative languages, such as the polymorphically typed Haskell, see the recent Bird and de Moor (1997) for the now mature theory and for many references. Compact "pointfree" definitions in the style of `second-last` above are used routinely, but many need additional operators, even application, for example (p 10):

```
length  =  sum . listr one            where  one a  =  1
```

Note the implicit application between `listr` and one and again between one and a in the local `where` definition. The whole definition may be read as: to compute the length of a list, let one be the function which ignores its argument (a) and always returns 1, use this function to map (`= listr`) the given list to produce a list of just 1s, then take the sum of those.

At least for handling functions of several arguments categorical concepts are rather heavy artillery. Are there other ways? Consider again the runnning example. Written in plain prefix notation it needs no parentheses at all:

```
square  first  rest  reverse  [1 2 3 4]            -->  9
```

An expression with binary operators such as the infix expression `((6 - 4) * 3) + 2` is written in prefix notation also without parentheses as

```
+     *     -     6     4     3     2          -->  8
```

(Note in passing that the four consecutive numerals look suspiciously like a list of numbers.) We now have to make sense of the corresponding compositional notation

```
(+  .  *  .  -  .  6  .  4  .  3  .  2)  @  ?    -->  8 @ ?
```

Clearly the "2-function" is applied to the dummy argument. But the other "number functions" also have to be applied to something, and each value produced has to be retained for pairwise consumption by the binary operators. The order of consumption is the *reverse* of the order of production. So there must be a stack of intermediate values which first grows and later shrinks. The dummy argument ? is just an empty stack.

What we have gained is this: The expression denotes a composition of unary stack functions. Literal numerals or literal lists denote unary functions which return a stack that is just like the argument stack except that the number or the list has been pushed on top. Other symbols like `square` or `reverse` denote functions which expect as argument a stack whose top element is a number or a list. They return the same stack except that the top element has been replaced by its square or its reversal. Symbols for what are normally binary operations also denote unary functions from stacks to stacks except that they replace the top two elements by a new single one. It is helpful to reverse the notation so that compositions are written in the order of their evaluation. A more compelling reason is given in the next section.

This still only has composition as a second order stack function. Others are easy enough to introduce, using a variety of possible notations. But now we are exactly where we were at the beginning of section 2: The next level, third order stack functions, calls for a lambda calculus with variables ranging over first order stack functions. The variables can be eliminated by translating into a lean or rich combinatory calculus. Application can be eliminated by substituting composition of second order stack functions, and so on. This cycle must be avoided. (But maybe the levels can be collapsed by something resembling reducibility in Russell's theory of types?)

In reflective languages such as Lisp, Snobol and Prolog, in which *program = data*, it is easier to write interpreters, compilers, optimisers and theorem provers than in non-reflective languages. It is straightforward to define higher order functions, including the Y combinator.

Backus (1978) also introduces another language, the reflective FFP system, short for "Formal Functional Programming system". In addition to objects as in FP there is now a datatype of *expressions*. In addition to the listforming constructor as in FP there is now a new binary constructor to form *application*s consisting of an operator and an operand. So expressions can be built up, but they cannot be taken apart, and hence FFP is not fully reflective, *program = $1/2$ data*. One semantic rule, *metacomposition*, is immensely powerful. It can be used to define arbitrary new functional forms, including of course the fixed forms from FP. The rule also makes it possible to compute recursive functions without a recursive definition. This is because in the application the function is applied to a pair which includes the original list operand which in turn contains as its first element the expression denoting the very same function. The method is considerably simpler than the use of the Y combinator. A mechanism similar to metacomposition is possible in Joy, see section 6.

Joy is also reflective. As noted in passing earlier, expressions which denote compositions of stack functions which push a value already look like lists. Joy extends this to arbitrary expressions. These are now called quotations and can be manipulated by list operations. The effect of higher order functions is obtained by first order functions which take such quotations as parameters.

A little tutorial on Joy

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. In the default mode there is no need for an explicit output instruction, so the numeral 5 is now written to the output file which normally is the screen. Joy has the usual arithmetic operators for integers, and also two other simple datatypes: truth values and characters, with appropriate operators.

The example expression above is potentially ambiguous. On one reading it is a postfix expression, equivalent to prefix or infix, with binary + as the main operator. On another reading it is a nested infix expression, with either of the two suppressed composition operators as the main operator. In practice there is no confusion, but there is a world of difference in the semantics.

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. These operators do not make sense in true postfix notation, so Joy uses the second reading of the ambiguous expression mentioned above.

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 concatenates 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. There it may be further manipulated or it may be written to the output file. Other list operators are `first` and `rest` for extracting parts of lists. Another is `cons` for adding a single element, for example `2 [3 4] cons` yields `[2 3 4]`. Since `concat` and `cons` are not commutative, it is often useful to use `swoncat` and `swons` which conceptually perform a `swap` first. Lisp programmers should note that there is no notion of dotted pairs in Joy. Lists are the most important sequence types, the other are strings of characters. Sequences are ordered, but there are also sets (currently only implemented as wordsize bitstrings, with the obvious limitations). Sequences and sets constitute the aggregate types. Where possible operators are overloaded, so they have some *ad hoc* but still somewhat systematic polymorphism.

Joy makes extensive use of combinators. These are like operators in that they expect something specific on top of the stack. But 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 the quoted 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. The `map` combinator also works for strings and sets. Similarly, there are a `filter` and a `fold` combinator, both for any aggregate.

The simplest combinator is `i` (for 'interpret'). The quotation parameter `[dup *]` of the `map` example can be used by the `i` combinator to square a single number. So `[dup *] i` does exactly the same as just `dup *`. Hence `i` undoes what quotation did, it is a dequotation operator, just like `eval` in Lisp. All other combinators are also dequotation operators. But now consider the program `1 2 3` and its quotation `[1 2 3]`. The program

pushes three numbers, and the quotation is just a list literal. Feeding the list or quotation to `i` pushes the three numbers. So we can see that lists are just a special case of quotations.

The familiar list operators can be used for quotations with good effect. For example, the quotation `[* +]`, if dequoted by a combinator, expects three parameters on top of the stack. The program `10 [* +] cons` produces the quotation `[10 * +]` which when dequoted expects only two parameters because it supplies one itself. The effect is similar to what happens in the lambda calculus when a curried function of three arguments is applied to one argument. As mentioned earlier, a similar *explicit* application operation is available in FFP. The device of constructed programs is very useful in Joy, and the resultant simple notation is another reason for writing function composition in diagrammatic order.

Combinators can take several quotation parameters. For example the `ifte` or if-then-else combinator expects three in addition to whatever data parameters it needs. Third from the top is an if-part. If its execution yields truth, then the then-part is executed, which is second from the top. Otherwise the else-part on top is executed. The order was chosen because in most cases the three parts will be pushed just before `ifte` executes. For example, the following yields the absolute value of an integer, note the empty else-part.

```
[0 <]  [0 swap -]  []  ifte
```

Sometimes it is necessary to affect the elements just below the top element. This might be to add or swap the second and third element, to apply a unary operator to just the second element, or to push a new item on whatever stack is left below the top element. The `dip` combinator expects a quotation parameter (which it will consume), and below that one more element. It saves that element away, executes the quotation on whatever of the stack is left, and then restores the saved element. So `2 3 4 [+] dip` is the same as `5 4`. This single combinator was inspired by several special purpose optimising combinators `S'`, `B'` and `C'` in the combinatory calculus, see Peyton Jones (1987, sections 16.2.5 and 16.2.6).

The stack is normally just a list, but even operators and combinators can get onto it by e.g. `[swap] first`. Since the stack is the memory, in Joy *program = data = memory*. The stack can be pushed as a quotation onto the stack by `stack`, a quotation can be turned into the stack by `unstack`. A list on the stack, such as `[1 2 3 4]` can be treated temporarily as the stack by a quotation, say `[+ *]` and the combinator `infra`, with the result `[9 4]`.

In definitions of new functions no formal parameters are used, and hence there is no substitution of actual parameters for formal parameters. Definitions consist of a new symbol to be defined, then the == symbol, and then a program. After the first definition below, the symbol `square` can be used in place of `dup *` .

```
square   ==   dup  *
  size     ==   [pop 1]  map  sum
```

The second definition is the counterpart of the definition of `length` in Bird and de Moor (1997 p 10) mentioned in the previous section, except that it is called `size` because it also applies to sets. (Note that no local definition of one is needed.) As in other programming languages, definitions may be recursive, but the effect of recursion can be obtained by other means. Joy has several combinators which make recursive execution of programs more succinct.

One of these is the `genrec` combinator which takes four program parameters in addition to whatever data parameters it needs. Fourth from the top is an if-part, followed by a then-part. If the if-part yields `true`, then the then-part is executed and the combinator terminates. The other two parameters are the rec1-part and the rec2part. If the if-part yields `false`, the rec1-part is executed. Following that the four program parameters and the combinator are again pushed onto the stack bundled up in a quoted form. Then the rec2-part is executed, where it will find the bundled form. Typically it will then execute the bundled form, either with `i` or with app2, or some other combinator. The following pieces of code, *without any definitions*, compute the factorial, the (naive)

Fibonacci and quicksort. The four parts are here aligned to make comparisons easier.

```
  [null ] [succ] [dup pred         ] [i *                    ] genrec
    [small] [    ] [pred dup pred   ] [app2 +                   ] genrec
    [small] [    ] [uncons [>] split] [app2 swapd cons concat] genrec
```

The overloaded unary predicate `null` returns `true` for `0` and for empty aggregates. Similarly `small` returns `true` for integers less than 2 and for aggregates of less than two members. The unary operators `succ` and `pred` return the successor and predecessor of integers or characters. The aggregate operator `uncons` returns two values, it undoes what `cons` does. The aggregate combinator `split` is like `filter` but it returns two aggregates, containing respectively those elements that did or did not pass the test. The `app2` combinator applies the same quotation to two elements below and returns two results. The `swapd` operator is an example of having to shuffle some elements but leaving the topmost element intact. This operator swaps the second and third element, it is defined as `[swap] dip`. Of course the factorial and Fibonacci functions can also be computed more efficiently in Joy using *accumulating parameter*s.

Two other general recursion combinators are `linrec` and `binrec` for computing linear recursion and binary recursion without having to introduce definitions. Both have essentially the same kinds of four parts as `genrec`, except that the recursion occurs automatically between the rec1-part and the rec2-part. The following is a small program which takes one sequence as parameter and returns the list of all permutations of that sequence. For example, from sequences of 4 elements such as the string `"abcd"`, the heterogeneous list `[foo 7 'A "hello"]` or the quotation `[[1 2 3] [dup *] map reverse]` it will produce the list of 24 permutation strings or lists or quotations.

```
1               [ small ]
  2               [ unitlist ]
  3               [ uncons ]
  4.1             [ swap
  4.2.1             [ swons
  4.2.2.1             [ small ]
  4.2.2.2             [ unitlist ]
  4.2.2.3             [ dup unswons [uncons] dip swons ]
  4.2.2.4             [ swap [swons] cons map cons ]
  4.2.2.5             linrec ]
  4.3               cons map
  4.4               [null] [] [uncons] [concat] linrec ]
  5               linrec.
```

The `unitlist` operator might have been defined as `[] cons`. The `unswons` operator undoes what `swons` does, and it might have been defined as `uncons swap`. An essentially identical program is in the Joy library under the name `permlist`. It is considerably shorter than the one given here because it uses two subsidiary programs which are useful elsewhere. One of these is `insertlist` (essentially 4.2) which takes a sequence and a further potential new member as parameter and produces the list of all sequences obtained by inserting the new member once in all possible positions. The other is `flatten` (essentially 4.4) which takes a list of sequences and concatenates them to produce a single sequence. The program given above is an example of a non-trivial program which uses the `linrec` combinator three times and the `map` combinator twice, with constructed programs as parameters on both occasions. Of course such a program with local definitions for `insertlist` and `flatten` can be written in any lambda calculus notation. But in Joy, as in other pointfree notations, no local *definitions* are needed, one simply takes the *bodies* of the definitions and inserts them textually.

The semantics of Joy can be expressed by two functions EvP and Eva, whose types are:

```
EvP : PROGRAM * STACK  ->  STACK  (evaluate concatenated program)
  EvA :    ATOM * STACK  ->  STACK  (evaluate atomic program)
```

In the following a Prolog-like syntax is used (but without the comma separator): If R is a (possibly empty) program or list or stack, then `[F S T | R]` is the program or list or stack whose first, second and third elements are F, S and T, and whose remainder is R. The first two equations express that programs are evaluated sequentially from left to right.

```
EvP( [] , S)  =  S
  EvP( [A | P] , S)  =  EvP( P , EvA(A, S))
```

The remaining equations concern atomic programs. This small selection is restricted to those literals, operators and combinators that were mentioned in the paper. The exposition also ignores the data types character, string and set.

```
(Push literals:)
    EvA( numeral , S)  =  [number | S]  (e.g.  7  42  -123 )
    EvA( true    , S)  =  [true   | S]  (ditto "false")
    EvA( [..]    , S)  =  [[..]   | S]  ([..] is a list or quotation)
  (Stack editing operators:)
    EvA( dup   , [X   | S])  =  [X X | S]
    EvA( swap  , [X Y | S])  =  [Y X | S]
    EvA( pop   , [X   | S])  =         S
    EvA( stack ,        S )  =  [S   | S]
    EvA(unstack, [L    | S])  =  L          (L is a quotation of list)
  (Numeric and Boolean operators and predicates:)
  EvA( +   , [n1 n2 | S])  =  [n | S]    where n = (n2 + n1)
  EvA( -   , [n1 n2 | S])  =  [n | S]    where n = (n2 - n1)
  EvA( succ, [n1    | S])  =  [n | S]    where n = (n1 + 1)
  EvA( <   , [n1 n2 | S])  =  [b | S]    where b = (n2 < n1)
  EvA( and , [b1 b2 | S])  =  [b | S]    where b = (b2 and b1)
  EvA( null, [n     | S])  =  [b | S]    where b = (n = 0)
  EvA(small, [n     | S])  =  [b | S]    where b = (n < 2)
(List operators and predicates:)
  EvA( cons  , [ R   F  | S])  =  [[F | R] | S]
  EvA( first , [[F | R] | S])  =  [F       | S]
  EvA( rest  , [[F | R] | S])  =  [     R  | S]
  EvA( swons , [ F   R  | S])  =  [[F | R] | S]
  EvA( uncons, [[F | R] | S])  =  [R F | S]
  EvA( null  , [L | S])  =  [b | S]    where b = (L is empty)
  Eva( small , [L | S])  =  [b | S]    where b = (L has < 2 members)
(Combinators:)
  Eva( i   , [Q   | S])  =  EvP(Q, S)
  EvA( x   , [Q   | S])  =  EvP(Q, [Q | S])
  EvA( dip , [Q X | S])  =  [X | T]     where EvP(Q, S) = T
  EvA(infra, [Q X | S])  =  [Y | S]     where EvP(Q, X) = Y
  EvA( ifte, [E T I | S])  =
    if EvP(I, S) = [true | U]                (free U is arbitrary)
      then EvP(T, S)  else EvP(E, S)
  EvA( app2, [Q X1 X2 | S])  =  [Y1 Y2 | S]
```

```
          where EvP(Q, [X1 | S]) = [Y1 | U1]   (U1 is arbitrary)
            and EvP(Q, [X2 | S]) = [Y2 | U2]   (U2 is arbitrary)
EvA( map , [Q [] | S])  =  [[] | S]
EvA( map , [Q [F1 | R1] | S])  =  [[F2 | R2] | S]
          where EvP(Q, [F1 | S]) = [F2 | U1]
            and EvA( map, [Q R1 | S]) = [R2 | U2]
EvA( split , [Q [] | S])  =  [[] [] | S]
EvA( split , [Q [X | L] | S]  =
  (if EvP(Q, [X | S]) = [true | U]
          then [FL [X | TL] | S]  else [[X | FL] TL | S])
       where EvA( split, [Q L | S]) = [TL FL | S]
EvA( genrec , [R2 R1 T I | S])  =
  if EvP(I, S) = [true | U]  then EvP(T, S)
  else EvP(R2, [[I T R1 R2 genrec] | W])
          where EvP(R1, S) = W
EvA( linrec, [R2 R1 T I | S])  =
  if EvP(I, S) = [true | U]  then EvP(T , S)
  else EvP(R2, EvA(linrec, [R2 R1 I T | W]))
          where EvP(R1, S) = W
EvA( binrec, [R2 R1 T I | S])  =
  if EvP(I, S) = [true | U]  then EvP(T, S)
  else EvP(R2, [Y1 Y2 | V])
          where EvP(R1, S) = [X1 X2 | V]
            and EvA(binrec, [R2 R1 T I X1 | V]) = [Y1 | U1]
            and EvA(binrec, [R2 R1 T I X2 | V]) = [Y2 | U2]
```

Theory of Joy

In any functional language expressions can be evaluated by stepwise rewriting. In primary school we did this with arithmetic expressions which became shorter with every step. We were not aware that the linear form really represents a tree. The lambda calculus has more complicated rules. The beta rule handles the application of abstractions to arguments, and this requires possibly multiple substitutions of the same argument expression for the multiple occurrences of the same variable. Again the linear form represents a tree, so the rules transform trees. The explicit substitution can be avoided by using an environment of name-value pairs, as is done in many implementations. In the combinatory calculus there is a tree rule for each of the combinators. The S combinator produces duplicate subtrees, but this can be avoided by sharing the subtree. Sharing turns the tree into a directed acyclic graph, but it gives lazy evaluation for free, see Peyton Jones (1987, Chapter 15). Rewriting in any of the above typically allows different ordering of the steps. If there are lengthening rules, then using the wrong order may not terminate. Apart from that there is always the question of efficiently *finding* the next reducible subexpression. One strategy, already used in primary school, involved *searching* from the beginning at every step. This can be used in prefix, infix and postfix forms of expression trees, and in the latter form the search can be eliminated entirely. Postfix ("John Mary loves") is used in ancient Sanscrit and its descendants such as modern Tibetan, in subordinate clauses in many European languages, and, would you believe, in the Startrek language Klingon. Its advantage in eliminating parentheses entirely has been known ever since Polish logicians used prefix for that same purpose. It can be given an alternative reading as an infix expression denoting the compositions of unary functions. Such expressions can be efficiently evaluated on a stack. For that reason it is frequently used by compilers as an internal language. The imperative programming language Forth and the typesetting language Postscript are often said to be in postfix, but that is only correct for a small fragment.

In the following example the lines are doubly labelled, lines a) to f) represent the stack growing to the right followed by the remainder of the program. The latter now has to be read as a sequence of instructions, or equivalently as denoting the composition of unary stack functions.

```
1   a)                                  2   3   +   4   *
        b)      2                         3   +   4   *
        c)      2   3                     +   4   *
  2     d)      5                         4   *
        e)      5   4                     *
  3     f)      20
```

If we ignore the gap between the stack and the expression, then lines a) to c) are identical, and lines d) and e) are also identical. So, while the stack is essential for the semantics and at least useful for an efficient implementation, it can be completely ignored in a rewriting system which only needs the three numbered steps. Such a rewriting needs obvious axioms for each operator. But it also needs a rule.

Program concatenation and function composition are associative and have a (left and right) unit element, the empty program and the identity function. Hence meaning maps a syntactic monoid into a semantic monoid. 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 a rewriting system because identity of functions is of course already an equivalence.

To illustrate rewriting in Joy, the "paradoxical" Y combinator for recursion in the lambda calculus and combinatory logic has a counterpart in Joy, the y combinator defined recursively in the first definition below. Then that needs to be the only recursive definition. Alternatively it can be defined without recursion as in the second definition.

```
y   ==  dup  [[y] cons]  dip  i
  y   ==  [dup cons]  swap  concat  dup  cons  i
```

The second definition is of greater interest. It expects a program on top of the stack from which it will construct another program which has the property that if it is ever executed by a combinator (such as `i`) it will first construct a replica of itself. Let `[P]` be a quoted program. Then the rewriting of the initial action of y looks like this:

```
1       [P]  y
  2   ==  [P]  [dup cons]  swap  concat  dup  cons  i        (def y)
  3   ==  [dup cons]  [P]  concat  dup  cons  i             (swap)
  4   ==  [dup cons P]  dup  cons  i                        (concat)
  5   ==  [dup cons P]  [dup cons P]  cons  i               (dup)
  6   ==  [[dup cons P] dup cons P]  i                      (cons)
  7   ==  [dup cons P]  dup  cons  P                        (i)
  8   ==  [dup cons P]  [dup cons P]  cons  P               (dup)
  9   ==  [[dup cons P] dup cons P]  P                      (cons)
```

What happens next depends on P. Whatever it does, the topmost parameter that it will find on the stack is the curious double quotation in line 9. (It is amusing to see what happens when `[P]` is the empty program `[]`, especially lines 6 to 9. Not so amusing is `[i]`, and worse still is `[y]`).

Actual uses of y may be illustrated with the factorial function. The first line below is a standard recursive definition. The second one uses uses y to perform anonymous recursion. Note the extra two pops and the extra `dip` which are needed to bypass the quotation constructed by line 9 above. The third definition is

discussed below. In the three bodies the recursive step is initiated by f1, i and x respectively.

```
f1  ==    [    null]  [    succ]  [ dup pred      f1 *]  ifte
  f2  ==  [ [pop null]  [pop succ]  [[dup pred] dip i  *]  ifte ]  y
  f3  ==  [ [pop null]  [pop succ]  [[dup pred] dip x  *]  ifte ]  x
```

But the y combinator is inefficient because every recursive call by i in the body *consumes* the quotation on top of the stack, and hence has to first replicate it to make it available for the next, possibly recursive call. The replication steps are the same as initial steps 6 to 9. But all this makes y rather inefficient. However, first consuming and then replicating can be avoided by replacing both y and i in the body of f2 by something else. This is done in the body of f3 which uses the simple x combinator that *could* be defined as dup i. Since the definitions of f2 and f3 are not recursive, it is possible to *just use the body* of either of them and textually insert it where it is wanted. The very simple x combinator does much the same job as the (initially quite difficult) metacomposition in FFP, see Backus (1978 section 13.3.2), which provided the inspiration. A simple device similar to x can be used for anonymous mutual recursion in Joy. The need to bypass the quotation by the pops and the dip is eliminated in the genrec, binrec and linrec combinators discussed in the previous section, and also in some other special purpose variants. In the implementation no such quotation is ever constructed.

To return to rewriting, Joy has the extensional composition constructor concatenation satisfying the substitution rule. Joy has only one other constructor, quotation, but that is *intensional*. For example, altough the two stack functions succ and 1 + are identical, the quotations [succ] and [1 +] are not, since for instance their sizes or their firsts are different. However, most combinators do not examine the insides of their quotation parameters textually. For these we have further substitution rule: If Q1 and Q2 are programs which denote the same function and C is such a combinator, then [Q1] C and [Q2] C denote the same function. In other words, Q1 and Q2 can replace each other in quotations embedded in suitable combinator contexts. Unsuitable is the otherwise perfectly good combinator rest i. For although succ and 1 + denote the same function, [succ] rest i and [1 +] rest i do not.

The rewriting system gives rise to a simple algebra of Joy which is useful for programming and possibly for optimisations, replacing complex programs by simpler ones. In the first line below, consider the two equations in conventional 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 in Joy algebra. The second line gives the same equations *without variables* in Joy.

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

Other equivalences express algebraic properties of various operations. For example the predecessor function is the inverse of the successor function, so their composition is the identity function id. The conjunction function and for truth values is idempotent. 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.

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

Some properties of operations have to be expressed by combinators. In the first example below, the dip combinator is used to express the associativity of addition. In the second example below app2 expresses one of the De Morgan laws. In the third example it expresses that size is a homomorphism from lists with concatenation to numbers with addition. 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  +
```

The sequence operator reverse is a purely structural operator, independent of the nature of its elements. It does not matter whether they are individually replaced by others before or after the reversal. Such structural functions are called natural transformations in category theory and polymorphic functions in computer science. This is how naturality laws are expressed in Joy:

```
[reverse]  dip  map   ==   map  reverse
  [rest]  dip  map   ==   map  rest
  [concat]  dip  map   ==   [map]  cons  app2  concat
  [cons]  dip  map   ==   dup  [dip]  dip  map  cons
  [flatten]  map  flatten   ==   flatten  flatten
  [transpose] dip [map] cons map  ==  [map] cons map transpose
```

A matrix is implemented as a list of lists, and for mapping it requires mapping each sublist by [map] cons map. Transposition is a list operation which abstractly interchanges rows and columns.

Such laws are proved by providing dummy parameters to both sides and showing that they reduce to the same result. For example

```
M  [P]  [transpose]  dip  [map]  cons  map
  M  [P]  [map]  cons  map  transpose
```

reduce, from the same matrix M, along two different paths with two different intermediate matrices N1 and N2, to a common matrix 0.

To show that Joy is Turing complete, it is necessary to show that some universal language can be translated into Joy. One such language is the untyped lambda calculus with variables but without constants, and with abstraction and application as the only constructors. Lambda expressions can be translated into the SK combinatory calculus which has no abstraction and hence is already closer to Joy. Hence it is only required to translate application and the two combinators S and K into Joy counterparts. The K combinator applicative expression K y x reduces to just y. The S combinator applicative expression S f g x reduces to (f x) (g x). The reductions must be preserved by the Joy counterparts, with quotation and composition as the only constructors. The translation from lambda calculus to combinatory calculus produces expressions such as K y and S f g, and these also have to be translated correctly. Moreover, the translation has to be such that when the combinatory expression is applied to x to enable a reduction, the same occurs in the Joy counterpart.

Two Joy combinators are needed, k and s, defined in the semantics by the evaluation function EvP for atoms:

```
EvA( k, [Y X | S])  =  EvP(Y, S)
  EvA( s, [F G X | S])  =  EvP(F,[X | T])  where EvP(G, [X | S]) = T
```

In the above two clauses Y, F and G will be passed to the evaluation function EvP for programs, hence they will always be quotations. The required translation scheme is as in the first line below, where the primed variables represent the translations of their unprimed counterparts.

```
K y   =>    ['y] k          S f g   =>    ['g] ['f] s
  K y x => 'x ['y] k          S f g x => 'x ['g] ['f] s
```

The second line shows the translations for the combinatory expression applied to x. In both lines the intersymbol spaces denote application in the combinatory source and composition in the Joy target. This is what Meertens (1989 p 72) asked for in the quote early in section 3. Since x is an argument, its translation `'x` has to push something onto the Joy stack.

Alternatively, s, k and others may be variously defined from

```
 k == [pop ] dip i            s == cons2 b
   c == [swap] dip i        cons2 == [[dup] dip cons swap] dip cons
   w == [dup ] dip i         cons == dup cons2 pop
   b == [i   ] dip i            y == [dup cons] swap [b] cons cons i
  id == [] i                    x == dup i
   i == dup dip pop         twice == dup b
```

The reduction rule for K requires that `'y ['x] k` reduce to `'x` which is `'y ['x] [pop] dip i`. The reduction rule for S requires `'x ['g] ['f] s` to reduce to `['x 'g] ['x 'f] b`, where consing the `'x` into the two quotations can be done by `cons2`. The definition of y is different from the one given earlier which relied on `b == concat i`. So, apart from the base s and k, another more Joy-like base is pop, swap, dup, the sole combinator `dip`, and either `cons` or `cons2`. Because of x or y, no recursive definitions are ever required. Since conditionals translated from the lambda calculus are certain to be cumbersome, a most likely early addition would be `ifte` and truth values. Instead of Church numerals there will be Joy numerals. For efficiency one should allow constants such as decimal numerals, function and predicate symbols in the lambda calculus, and translate these unchanged into SK or a richer calculus, then unchanged into Joy but in postfix order.

So far lists and programs can only be given literally or built up using `cons`, they cannot be inspected or taken apart. For this we need the `null` predicate and the `uncons` operator. Then `first` and `rest` can be defined as `uncons pop` and `uncons swap pop`. Other list operators and the `map`, `fold` and `filter` combinators can now be defined without recursion using x or y.

Concluding remarks

In all aspects Joy is still in its infancy and cannot compete with the mature languages.

Various extensions of Joy are possible. Since the functions are unary, they might be replaced by binary relations. This was done in an earlier but now defunct version written in Prolog which gave backtracking for free. Another possible extension is to add impure features. Joy already has `get` and `put` for explicit input and output, useful for debugging, it has `include` for libraries, a `help` facility and various switches settable from within Joy. There are no plans to add fully blown imperative features such as assignable variables. However, Raoult and Sethi (1983) propose a purely functional version for a language of unary stack functions. The stack has an everpresent extra element which is conceptually on top of the stack, a state consisting of variable-value pairs. Most activity of the stack occurs below the state, only the *purely functional* `fetch X` and `assign Y` perform a read or write from the state and and they perform a push or a pop on the remainder of the stack. The authors also propose uses of *continuations* for such a language. Adapting these ideas to Joy is still on the backburner, and so are many other ideas like the relation of the stack to linear logic and the use of categorial grammars (nothing to do with category theory) for the rewriting. Since the novelty of Joy is for programming in the small, no object oriented extensions are planned beyond a current simple device for hiding selected auxiliary definitions from the outside view.

For any algebra, any relational structure, any programming language it is possible to have alternative sets of primitives and alternative sets of axioms. Which sets are optimal depends on circumstances, and to evolve optimal sets takes time. One only needs to be reminded of the decade of discussions on the elimination of `goto` and the introduction of a small, orthogonal and complete set of primitives for flow of control in imperative languages. The current implementation and library of Joy contain several experimental operators and combinators whose true value is still unclear. So, at present it is not known what would be an *optimal* set

of primitives in Joy for writing programs.

It is easy enough to eliminate the intensionality of quotation. Lists and quotations would be distinguished textually, and operators that build up like `cons` and `concat` are allowed on both. But operators which examine the insides, like `first`, `rest` and even `size` are only allowed on lists. It is worth pointing out that the earlier list of primitives does not include or derive them. Now the substitution rule for quotation is simply this: if Q1 and Q2 denote the same program, then so do `[Q1]` and `[Q2]`. But maiming quotation in this manner comes at a price --- compilers, interpreters, optimisers, even pretty-printers and other important kinds of program processing programs become impossible, although one remedy might be to "certify and seal off" quotations after such processing. As G\"{o}del showed, any language that has arithmetic can, in a cumbersome way, using what are now called G\"{o}del numbers, talk about the syntax of any language, including its own. Hofstadter (1985 p 449) was not entirely joking when, in response to Minsky's criticism of G\"{o}del for not inventing Lisp, he was tempted to say 'G\"{o}del *did* invent Lisp'. But we should add in the same tone 'And McCarthy invented `quote`. And he saw that it *was* good'.


```
J.W. Backus.
    Can programming be liberated from the von {N}eumann style? a
    functional style and its algebra of programs.
    {\it Communications of the ACM}, 21(8):613, 1978.

J.W. Backus, J. Williams, and E.W. Wimmers.
    An introduction to the programming language {FL}.
    In D.A. Turner, editor, {\it Research Topics in Functional
    Programming}, page~219, Addison Wesley, 1990.

M. Barr and C. Wells.
    {\it Category Theory for Computer Science}.
    Prentice Hall, 1990.

R. Bird and O. de~Moor.
    {\it Algebra of Programming}.
    Prentice Hall, 1997.

T.H. Brus, M.C.J.D. van~Eekelen, M.O. van~Leer, and M.J. Plasmejer.
    Clean --- a language for functional graph rewriting.
    In G. Kahn, editor, {\it Functional Programming Languages and
    Computer Architecture}, page~367, Springer: LNCS vol. 272, 1987.

G. Cousineau, P.-L. Curien, and M. Mauny.
    The categorical abstract machine.
    {\it Science of Computer Programming}, 9:203, 1987.

H. Curry and R. Feys.
    {\it Combinatory Logic}.
    Volume~1, North Holland, 1958.

G. Hains and C. Foisy.
    The data-parallel categorical abstract machine.
    In A. Bode, M. Reeve, and G. Wolf, editors, {\it PARLE '93 Parallel
    Architectures and Languages Europe}, page~56, Springer: LNCS vol. 694, 1993.
```

D. Hofstadter.
{\it Metamagical Themas: Questing for the Essence of Mind and Pattern}.
Basic Books, 1985.

L. Meertens.
Constructing a calculus of programs.
In J.L.A. {van de Snepscheut}, editor, {\it Mathematics of Program Construction}, page~66, Springer: LNCS vol. 375, 1989.

S.L. PeytonJones.
{\it The Implementation of Functional Languages}.
Prentice Hall, 1987.

J.-C. Raoult and R. Sethi.
Properties of a notation for combining functions.
{\it J. Assoc. for Computing Machinery}, 30:595, 1983.

M. {Sch\"{o}nfinkel}.
On the building blocks of mathematical logic.
In J. van~Heijenoort, editor, {\it From Frege to {G\"{o}del}}, page~357, Harvard University Press, 1967.
English Translation from the German original. Includes foreword by W.V.O. Quine.