Object-oriented programming as the end of history in programming languages

M.H. van Emden
Department of Computer Science
University of Victoria
P.O. Box 3055, Victoria, B.C., V8W 3P6 Canada

ABSTRACT: In the past, the invention of a new programming paradigm led to new programming languages. We show that C^{++} is a perfectly adequate dataflow programming language, given some suitable definitions. Reasons are mentioned for believing that this is also the case for logic and functional programming. We conclude that object-oriented programming may remove the need for languages motivated by specific programming paradigms.

Introduction

The earliest programming languages were machine-oriented, as the contemporary assemblers still are. The next step was toward high-level languages, such as Fortran, Cobol, and Algol. From our current perspective we call these "procedure-oriented". The main events after that were languages centered around a programming paradigm: functional, dataflow, and logic programming languages.

I take as definition of programming paradigm: a way of looking at a programming problem that makes a certain class of problems much easier than in other such ways. Thus a programming paradigm is recognized as such before a language is specifically designed for it. In the case of functional, dataflow and logic programming, such languages have been implemented.

Object-oriented programming (OOP) plays the distinct role of meta-paradigm. Accordingly, I do not try to give it a place in the above historical development. In this paper I show that OOP subsumes dataflow programming languages. I sketch an argument that it can do the same for functional and logic programming. If we jump to the conclusion that future paradigms can also be incorporated in OOP, then OOP is indeed the end of history in programming languages. This is not to say that any of the existing OOP languages will not be superseded: many details in these languages are still controversial.

Dataflow programming

In the dataflow paradigm all computation happens in a network consisting of nodes connected by unidirectional datapipes. Thus each node has zero or more input pipes (when the output end of the pipe is connected to the node), and zero or more output pipes (when the input end of the pipe is connected to the node). A pipe is used by repeatedly placing data items at its input end. These items can be retrieved from its output end in the same order in which they entered at the opposite end. Abstractly viewed, the pipes behave like the abstract data structure referred to as a *queue*.

The dataflow paradigm is justified by the class of problems that it makes easy to solve. An important subclass is Structured Systems Analysis (also called Structured Analysis and Design Technique), which is widely applied in commercial dataprocessing [2]. However, Structured Systems Analysis is a world unto itself, and is apparently not aware of the larger context of dataflow programming.

Structured Systems Analysis discovered dataflow by inspired thinking about commercial dataprocessing applications. Ashcroft and Wadge [8] arrived at the idea starting from studies in semantics of programming languages. Dennis, Arvind and others at MIT have arrived at the dataflow paradigm from computer architecture [7].

As early as 1977 the paradigm was already sufficiently compelling that a programming language was designed to make dataflow programming as natural as possible [4]. The paper just mentioned also contains some simple and widely appealing examples showing the paradigm at its best. Note that the paradigm was independently arrived at from disparate areas: commercial applications, semantics, and architecture. This suggests that it's "real" in some sense.

Hamming's problem solved in dataflow A good introduction to the dataflow paradigm is *Hamming's problem* [4]:

> to print out in increasing order all positive integers that have no prime factors other than 2, 3, or 5.

This problem is attributed to R.W. Hamming by Dijkstra [1], who provided an ingenious solution. It is more efficient,

0-7803-3905-3/97/$10.00©1997 IEEE

but is considerably harder to understand than the dataflow version given by Kahn and McQueen [4].

One approach to a solution starts with the observation that the infinite sequence x of numbers required by Hamming's problem satisfies the following equation:

$$x = 1 \circ merge(merge(t_2(x), t_3(x)), t_5(x))$$

where

- *merge* is the result of merging its two sorted input sequences into a sorted output sequence, suppressing duplicates
- t_2 is result of multiplying its input sequence number by number by 2; similarly for t_3 and t_5
- the meaning of $\circ$ is defined by $u \circ v$ being the result of prefixing the sequence v by the individual number u

The question whether the solution to Hamming's problem is the *only* solution to the equation is addressed by the methods developed in Kahn [3].

To turn this observation into a dataflow network, we take the above equation with a complex expression and turn it into a system of simple equations by introducing auxiliary variables. We do this in two steps. In the first step, we get rid of nested expressions:

$$\begin{array}{ccc} a = t_2(x) & b = t_3(x) & d = t_5(x) \\ c = merge(a,b) & x_1 = merge(c,d) & \\ x = 1 \circ x_1 & & \end{array}$$

The resulting simple equations can, if considered in isolation, each be translated directly into a node of a dataflow network. Each of a, b, c, d, and x_1 correspond to a datapipe because, in the above set of equations, they have exactly one occurrence in a left-hand side and exactly one occurrence in a right-hand side. An occurrence on the left-hand (right-hand) side corresponds to the output (input) side of the datapipe.

However, x has too many occurrences in right-hand sides. We can avoid this problem by making these occurrences into different variables, say f, g, and h. But how do we tell that these are the same sequence? To do that, we introduce a node type, with one input and two output pipes, that outputs two identical copies of each item that it removes from the input pipe. Let us call this node *split*.

$$\begin{array}{ccc} i = x & h = x & \\ f = i & g = i & \\ a = t_2(f) & b = t_3(g) & d = t_5(h) \\ c = merge(a,b) & x_1 = merge(c,d) & \\ x = 1 \circ x_1 & & \end{array}$$

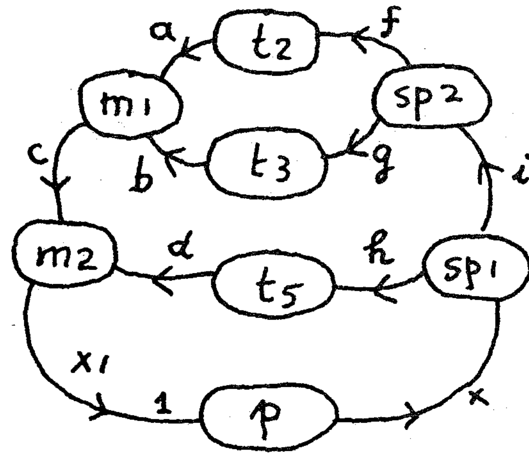

Figure 1: Dataflow network for Hamming's problem in the initial state, where all pipes are empty except a 1 in `x1`.

The entire top two lines each correspond each to a *split* node; the remaining six equations correspond to a node each. This makes eight nodes in all. See the network diagram in Figure 1.

An object-oriented implementation of Hamming's problem In the object-oriented solution of any problem, we are advised to consider the nouns of the specification as candidates for classes.

In any dataflow network, particularly conspicuous nouns are *node* and *datapipe*. As observed before, datapipes behave like queues, a commonly used class. In the C++ code in Figure 2 we can see a queue being created, each with maximum size 10, for each datapipe in the diagram.

According to the advice just mentioned, we should consider a class suitable for creating all required nodes as instances. The advice, though a good first approximation, needs some refinement. This is because objects of the same class should have states of the same form, though not necessarily of the same content. The state of a node object includes the states of the abutting datapipes. And of course, instances of the same class should have the same behaviour.

These considerations suggest that the *merge* nodes are instances of the same class (`merge`; the individual instances are `m1` and `m2`), as are the *split* nodes (of the class `split`, with instances `sp1` and `sp2`), as are the nodes t_2, t_3, and t_5 (of the class `times`, with instances `t1`, `t2`, and `t3`).

Lines 3–5 create instances of class *queue* to act as datapipes, each with a maximum size of 10. Given these pipes, lines 6–10 create the nodes, with the correct pipe connections. These lines seem the most succinct possible textual representation of the diagram in Figure 1. In so far as this is true, C++ comes close to the best possible dataflow

```
void main() {                               //01
const int maxTimes = 50;                    //02
queue a(10), b(10), c(10), d(10),           //03
      x1(10), x(10),                        //04
      f(10), g(10), h(10), i(10);           //05
merge m1(&a,&b,&c), m2(&c,&d,&x1);          //06
times t2(2,&f,&a), t3(3,&g,&b),             //07
      t5(5,&h,&d);                          //08
split sp1(&x,&h,&i), sp2(&i,&f,&g);         //09
print p(&x1,&x);                            //10
                                            //11
x1.add(1);                                  //12
for (int j=1; j < maxTimes; j++){           //13
    m1.run(); m2.run();                     //14
    t2.run(); t3.run(); t5.run();           //15
    sp1.run(); sp2.run(); p.run();          //16
  }
}//void main()
```

Figure 2: C++ code for the dataflow network for Hamming's problem. The node `print` has been added to allow the solution to be printed.

programming language.

Lines 3–10 *create* the dataflow network; they do not cause it to execute its computation. The attraction of the dataflow paradigm is to avoid the difficulty of conventional programming, namely to ensure that events happen in the right sequence. To execute a dataflow network, each node executes, independently of the others, the following simple computation:

> If any of the input datapipes is empty, or if any of the output datapipes is full, do nothing. Otherwise, remove the next item from each of the input pipes, perform on them the specialized computation characteristic for the type of node, and place the results, if any, on the output pipes.

The computation just described is invoked by a method called **run**, which is defined for each of the classes `merge`, `times`, and `split`.

To execute the entire network, one invokes the run method for each node, as in lines 14–16 of Figure 2. Typically several of these invocation have no effect because of full output or empty input pipes. But if the network can do anything at all, then at least one node will do something. In large networks it is worth optimizing the invocations of the **run** method. One can keep track of which nodes are blocked. A blocked node connected to a pipe of which the content changed may no longer be blocked and becomes a candidate for being **run**. Such an optimization is reminiscent of the constraint propagation algorithm of D. Waltz [9].

Note that in lines 14–16 the order of the statements does not matter: in dataflow it matters less what order things are done in than in conventional programming.

Space limitations prevent us from listing the entire program, which is about a hundred lines, including the queue implementation. We just add a representative class definition:

```
class times {
private:
  int mult; queue *in, *out;
public:
  times(int Mult, queue *In, queue *Out) {
    mult = Mult; in = In; out = Out;
  }
  void run() {
    if (in->empty() || out->full()) return;
    out->add(mult*(in->next()));
    in->remove();
  }
};
```

Other paradigms

Functional programming In procedure-oriented languages, numbers are privileged in that one can (1) give them names, (2) assign them to variables, and (3) return them as function results. In these languages, functions are underprivileged in that they share with numbers property (1), but not (2) and (3). The motivation for functional programming languages is to accord to them all the privileges of numbers, and thus make them "first-class objects", as was the going terminology in functional programming [6].

In OOP one can make functions into objects. This is exploited in the function objects of STL (the Standard Template Library [5] being included in the C^{++} standard). I don't want to suggest that STL has the power of a functional programming language. But that is because the designers did not share the motivation of the designers of functional programming languages. They had different objectives, in which function objects play only a minor role. But STL does suggest that the ideas for functional programming can be implemented in C^{++}. See the generic `accumulate` algorithm [5], which is a standard functional programming example.

Logic programming The main difficulty in a high-level implementation in C of a logic programming language like Prolog lies in control. Forward computations in Prolog

map neatly on the function-call control of C. But backtracking is difficult to implement when forward computation is done by conventional function calls. C^{++} has, in addition to the standard control of C, exception handlers. If each of the clauses for the same predicate is associated with an exception handler rather than a function, then the Prolog-style control is implemented in C^{++} in a natural way.

Concluding remarks

One wants to get all the help one can in making programming problems easier. This suggests being able to choose the right paradigm, even if that means switching languages.

However, in running a software-development shop, one wants to avoid the inefficiencies in training and documentation caused by not standardizing on a single language. Moreover, paradigm-oriented languages like Prolog and functional programming languages share with other languages a good deal of what I call "mundane infrastructure": how your write *less than or equal to*, strings, characters, in what lengths integers or floating-point numbers come, what mathematical functions are available, and so on. However valuable the paradigm in question may be, it does not add anything of value to the mundane infrastructure. Hence availing oneself of the advantages of the paradigm comes at present with a considerable cost in time, tedium, and frustration when one switches to a capriciously different mundane infrastructure.

Such a switch could be avoided by standardizing it and re-standardizing Prolog, Lisp, ML, ..., accordingly. Given the Herculean labors required so far for more limited standardization, this seems unlikely. It is more attractive to use the paradigms via a class library in an OOP language that one is using anyway, and thus to adopt the mundane infrastructure of that language.

In recent years the dilemma has been resolved almost universally by giving up on the advantage of working in the paradigm that is right for the problem: most shops, or even entire companies have standardized on a single language[1].

Although much lip service has been paid to the virtues of object-oriented programming, its full potential seems to be unknown. It is this: that we no longer have the dilemma. A fuller understanding of both object-oriented programming and the paradigm of interest allows one to write a suitable class library so that one can have the advantages of the paradigm of choice within the single programming language that organizational efficiency demands.

[1] But it's scary to see the stampedes resulting from everyone starting to do what everyone else is doing. The stampede into C^{++} has hardly subsided when another one into *Java* gets under way.

C^{++} has been widely criticized for its many shortcomings. These criticisms strengthen my case: I claim that C^{++} is *good enough* to implement valuable programming paradigms. As is to be hoped, C^{++} will be superseded by a better language. This paper provides a criterion for what is to count as "better". History may not, after all, have ended in programming languages.

References

[1] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[2] Chris Gane and Trish Sarson. *Structured Systems Analysis.* Prentice-Hall, 1979.

[3] Gilles Kahn. The semantics of a simple langauge for parallel processing. In *Information Processing 74.* North Holland, 1974. Proceeding of the IFIP Congress.

[4] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77.* North Holland, 1977. Proceeding of the IFIP Congress.

[5] David R. Musser and Atul Saini. *C++ Programming with the Standard Template Library.* Addison-Wesley, 1996.

[6] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey approach to Programming Language Theory.* MIT Press, 1977.

[7] A.H. Veen. Dataflow machine architecture. *Computing Surveys*, pages 365–396, 1986.

[8] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Language.* Academic Press, 1985.

[9] D. Waltz. Understanding line drawings in scenes with shadows. In Patrick Henry Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.