Full Metal Jacket: A Pure Visual Dataflow Language Built on Top of Lisp.

Donald Fisk
England
United Kingdom
hibou at onetel dot com

ABSTRACT

Full Metal Jacket is a general-purpose visual dataflow language currently being developed on top of Emblem, a Lisp dialect strongly influenced by Common Lisp but smaller and more type-aware, and with support for CLOS-style object orientation, graphics, event handling and multi-threading.

Methods in Full Metal Jacket Jacket are directed acyclic graphs. Data arriving at ingates from the calling method flows along edges through vertices, at which it gets transformed by applying Emblem functions or methods, or methods defined in Full Metal Jacket, before it finally arrives at outgates where it is propagated back upwards to the calling method.

The principal difference between Full Metal Jacket and existing visual dataflow languages such as Prograph is that Full Metal Jacket is a *pure* dataflow language, with no special syntax being required for control constructs such as loops or conditionals, which resemble ordinary methods except in the number of times they generate outputs. This uniform syntax means that, like Lisp and Prolog, methods in Full Metal Jacket are themselves data structures and can be manipulated as such.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Data-flow languages*

General Terms

Languages

Keywords

Language design, dataflow, visual programming, Lisp

1. INTRODUCTION

To date, many hundreds of different programming languages have been developed. However, the space of possible programming languages has not been evenly explored. Historically, most of the effort has centred around Algol derivatives, i.e. structured imperative languages recently augmented with object orientation (e.g. C++, Java, C#). Other language categories where effort has been concentrated include functional languages, object oriented languages, and the so-called scripting languages. (Common Lisp could be categorized either as functional or multi-paradigm.) It is the author's view that there is now limited scope for improvement within these categories, so it has become worthwhile to investigate alternatives.

One property all these categories share is that they are text-based, and consequently one-dimensional. However, algorithms in general can have several computations occurring concurrently, and therefore are arguably more naturally represented in two dimensions, as directed graphs. Visual dataflow languages, in which programs are represented in this form, have existed for some time, but there are relatively few of them and it is the author's opinion that there is scope for improvement over existing visual dataflow languages.

Full Metal Jacket, a visual dataflow language, is currently being implemented in Emblem (a Lisp dialect developed by the author), principally as an exercise in programming language design, but also to prove (or test) the usability (and *habitability*) of Emblem. Although Full Metal Jacket is still in a very early stage of development, and it remains uncertain whether it will prove to be useful in its own right, some important conclusions and design decisions about visual dataflow programming have been made, which are the subject of this paper.

2. HISTORICAL PERSPECTIVE

The author's interest in dataflow dates back to the early 1980s. At the time, the Japanese had just announced their Fifth Generation Project [13], and in particular the proposed adoption of highly parallel machine architectures based around the concept of dataflow. Such architectures had already been the subject of active research in the West for several years, a working prototype having been built at the University of Manchester [7]. Other research included a dataflow machine simulator developed at Burroughs in Cumbernauld by Stoyan Kableshkov [9], and work at MIT [14].

The high-level languages used or proposed for dataflow machines at the time were not a particularly good fit for the hardware. The Manchester Dataflow Group selected SISAL, the Japanese planned to use a variant of Prolog. However, although mapping of these onto dataflow hardware was by no means trivial, no attempt was made as part of the Fifth

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ILC2003 2003 New York, USA
Copyright 2003 Donald Fisk .

Generation Project to develop a graphical language in which the dataflow could be represented explicitly, in large part no doubt due to the high cost of good graphics terminals at the time.

Since the start of the Fifth Generation Project, Moore's Law has remained valid, so processors are now roughly 1000 times faster with comparable improvements in memory size, as well as improvements in graphical displays. This meant that there was a reduction in pressure to develop highly parallel computers, and as a result research into dataflow computer architectures ceased to be fashionable. However, recently there has been a considerable resurgence of interest in parallel distributed computing centred around The Grid [6].

Independently of the Fifth Generation Project and other dataflow computer projects, some dataflow programming languages, both general purpose (e.g. Prograph [4]) and special purpose (e.g. Labview [10]), were developed [8].

3. THE IMPLEMENTATION LANGUAGE

Full Metal Jacket is built on top of Emblem, a Lisp implementation, and most of its power resides in this underlying Lisp. Emblem has four design goals:

- to minimize the effort required to implement Full Metal Jacket;
- to be embeddable within Full Metal Jacket;
- to be small;
- not to diverge too much from Common Lisp.

Because there is a close correspondence between constructs in the two languages, it has been possible to keep Full Metal Jacket small, with most of the overall development effort going into writing Emblem itself. In particular, there is no need to provide separate library code for Full Metal Jacket where Emblem's library code can be used instead. It is possible to build a system in a mixture of the two languages, and call Emblem from Full Metal Jacket and vice-versa.

However, Full Metal Jacket is more than just a visual programming interface for an underlying Lisp – as will be shown below, it is a programming language in its own right. Instead, it was Emblem which was originally designed to be the underlying Lisp for Full Metal Jacket (rather than out of any dissatisfaction with Common Lisp), analogous to the Lisps embedded in Emacs, AutoCAD and Abuse. Emblem has since proven useful in its own right.

4. THE COMPUTATIONAL MODEL

In Full Metal Jacket, programs are directed graphs, i.e. vertices with attached methods or functions, connected by edges. Each vertex has one or more inputs and zero or more outputs. An edge connects a vertex output on one vertex to a vertex input on another vertex.

Values flows along edges from tail (a vertex output) to head (a vertex input), and when all the inputs of a vertex have received values, they are normally consumed, and its method or function is applied to them to produce output values, which are then propagated onwards to another vertex's inputs. Values can therefore be thought of as flowing along edges and being transformed at vertices.

Both vertex inputs and vertex outputs are typed, the types corresponding to the types of the underlying method or function. It is impossible (i.e. prevented by the graph editor) to connect a vertex output to a vertex input of an incompatible type.

Function or methods can be defined as directed graphs whose ingates represent the function or method's lambda list, and whose outgates represent the values returned. Such directed graphs are called λ-graphs. Each ingate and outgate has a single input and output. The outputs of ingates are connected to inputs of vertices, and the outputs of vertices can be connected to the inputs of either other vertices or outgates.

Because methods can be called in several places, and can be called recursively, it is necessary to tag values so that only corresponding input values (those with the same tag) are provided as arguments when the underlying function or method is called. Tags have integer values and are generated when a λ-graph is entered, for use within the λ-graph.

A problem arises with respect to the representation of conditional and iterative evaluation. Two solutions are being adopted with the choice, which depends on the details of the computation being performed, left to the programmer. The first solution is completely general and is similar to what is done in Smalltalk. This is to encapsulate conditionals and iteration in higher order functions or methods implemented in Emblem, e.g.

```
(defun =if= ((test Bool) (then Fun) (else Fun))
  (returns Any)
  (funcall (if test then else)))

(defun =while= ((test Fun) (body Fun))
  (returns)
  (do ()
      ((not (funcall test)))
      (funcall body)))
```

An example of its use in Full Metal Jacket is shown in Figure 1. In Emblem, this would be written as

```
(defun fac ((n Int))
  (returns Int)
  (=if= (zerop n)
        (lambda () 1)
        (lambda () (* n (fac (sub1 n))))))
```

The second approach involves generalizing the behaviour of the vertices so that they do not all produce exactly one set of output values for each set of input values they consume. For example, `when` only outputs if one of its inputs receives the value `TRUE`, `unless` if one of its inputs receives `FALSE`, `count` repeatedly outputs an integer counter from a lower bound to an upper (or vice-versa), and `collect` repeatedly receives inputs (e.g. from `count`) and processes them until a condition is satisfied (see Figure 2 for an illustration of its use). This can also be used to retrieve values from a producer, such as a queue or file. This second approach is similar to Common Lisp series, and like series, is not completely general.

Both the above approaches are pure dataflow, in contrast to those taken in Prograph and Labview.

4.1 Implications for the Implementation Language

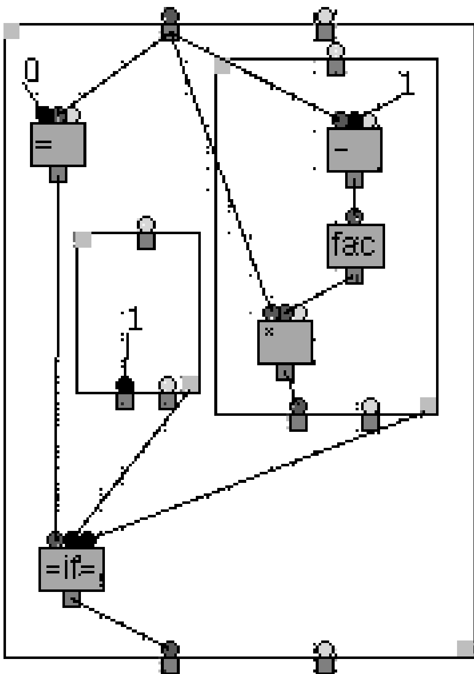

Figure 1: Recursive factorial using a Smalltalk-style conditional.

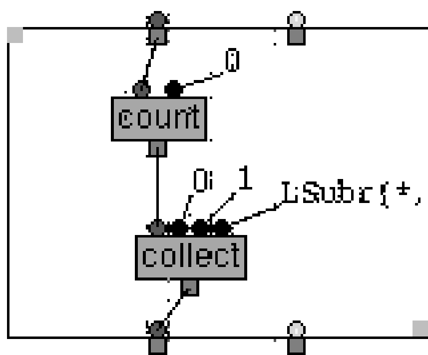

Figure 2: Iterative factorial. `count` outputs repeatedly, `collect` accumulates the values.

The computational model imposes, or at least makes desirable, certain constraints on the implementation language:

- all arguments and return values should have declarable types (undeclared types defaulting to `Any`), accessible to the programmer;
- types should be unambiguous, e.g. FALSE and '() should be distinguishable;
- multi-threading (with non-blocking I/O) should be supported.

Multiple values and additional arguments are also requirements.

5. THE PROGRAMMER INTERFACE

In designing a programmer interface for a visual programming language, it is important to ensure that expressing an algorithm in it is less effort than typing in the same algorithm in a text-based language.

In practice, this means minimizing the number of mouse gestures and keystrokes required, and ensuring similar actions are used to perform similar operations.

To define a new function, you double click on a window, then enter the name of the function into a dialog box. A window is displayed, containing an empty λ-graph, in which the function can be defined.

To add a gate (ingate or outgate), you click on a special gate called a *rest gate*. A rest ingate also functions to accept the remaining arguments of the function being defined. To assign a type to a gate, you right click on its input and then click to select the class from the class hierarchy, which is displayed in a separate window. The class hierarchy is displayed as a directed graph using the same widget set as is used in programs.

To connect an output of an ingate or a vertex to an input of a vertex already in the λ-graph or an outgate, you press the mouse on the output and drag it in the general direction of the input. When a line appears connecting the desired pair, you release the mouse. Normally, you can only connect to an input of the same type or a more general type. If the vertex you intend to connect to has not yet been added to the λ-graph, you *right click* on the output, and a new window opens containing all the vertices you can connect to (i.e. which can accept a value from the output). You then drag the vertex you want into the λ-graph. If there is no ambiguity as to which of its inputs can be connected to, the connexion is made automatically.

To add a λ-graph, you press the mouse on an empty point on the window, drag south-east and release.

Sometimes it is required that an input to a vertex have a constant value, and this is entered by pressing the mouse on the input, dragging up and releasing it. The value can then be entered into, or selected from, a dialog box. This is a special case of a *sticky input* – one whose value is *not* consumed when the outputs are produced. It is also possible to make an input sticky whose value arrives from another vertex, by clicking on the input. It is possible to connect a input which accepts functions to a λ-graph, by dragging back from the input to a corner of the λ-graph.

A vertex's documentation string is displayed by hovering over it with the mouse. This is also used to display the types of inputs and outputs.

An object can be deleted by pressing the mouse button on it and dragging it to an icon, and releasing it. Objects which depend on the object being deleted (e.g. edges joined to a vertex being deleted) are also deleted.

An important part of the programmer interface consists of the concepts which have to be understood to work within the language. As it is a good design principle not to increase the number of concepts unnecessarily (Occam's razor), the concepts of Full Metal Jacket are inherited from Lisp and graph theory.

5.1 Implications for the Implementation Language

The programmer interface outlined above imposes certain further constraints on the implementation language:

- good graphics and event handling;
- object orientation. Since it makes the programmer interface more complicated to have privileged arguments, multiple dispatch is preferable to message passing.
- the set of all functions and methods taking an argument of a given type should be accessible to the programmer.
- in addition to arguments and return values, documentation strings should also be accessible to the programmer;

The graphics system used for implementing Full Metal Jacket is Mistletoe, which was implemented as part of Emblem on top of Xlib. In Mistletoe, shapes are normally represented as objects, and contain the code that determines their behaviour in response to mouse gestures and other events.

6. PROGRAMS AS DATA

Programs in Full Metal Jacket are directed graphs. In the process of providing the ability to enter and edit programs, graph processing primitives had to be implemented. And since directed graphs are a completely general data structure, it makes sense to provide access to those primitives to Full Metal Jacket programmers, so that they can use them to manipulate *data* in the form of directed graphs, just as `car`, `cdr`, `cons`, etc. are available to Lisp programmers to manipulate lists.

This raises the possibility of adding macros to Full Metal Jacket. A macro's definition would contain code which generates code to replace invocations of the macro, just like Lisp macros except for the type of data structure they operate on. It might also be possible to automatically generate fragments of code at run time.

One disadvantage directed graphs have over (singly linked) lists is that adding an edge or vertex (the equivalent of `cons`), to be side-effect free, requires graphs to be represented as a list of vertices and edges. This means that executing a graph involves frequent list traversals to deliver data to its intended destination. The alternative, which is adopted in the current implementation, is to represent graphs as graphs, and live with side-effects. The expectation is that most graph modifying operations are done during graph construction where side-effects are not a problem.

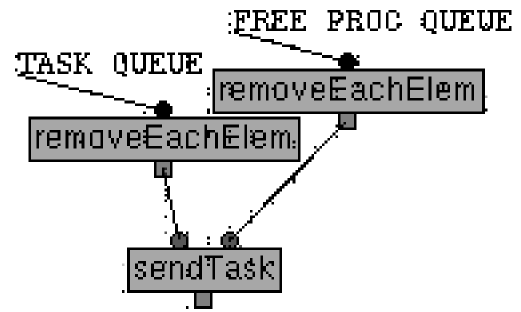

Figure 3: Allocation of tasks to other processes by the master process.

7. PARALLEL PROCESSING

Programs which do Multiple Instruction, Multiple Data (MIMD) parallel processing in conventional languages, using message passing systems such as MPI, are troublesome to implement. Much of the responsibility for synchronization and load balancing rests with the programmer.

With dataflow, this responsibility can be offloaded onto the system developer. Synchronization is handled automatically: a vertex waits until it has received all of its inputs before executing. Load balancing can be achieved with the use of an intelligent scheduler.

A possible distributed scheduling algorithm is outlined in the dataflow illustrated in Figures 3, 4 and 5. In Figure 3, two queues are repeatedly read from, one containing the next task to be run, the other containing the identity of a processor which is ready to run it. If a queue empties, `removeEachElem` blocks, and if values arrive at `sendTask` at different rates, the extra values are simply queued. Figure 4 illustrates how the free processor queue is filled. Most of the complexity in processing is hidden in `deliverValues`. In Figure 5, a process repeatedly receives tasks from the master process, executes them and returns the results to the master process. The processes can be distributed and/or running on separate processors.

8. ARTIFICIAL INTELLIGENCE

There are two complementary approaches to AI that have achieved some degree of success. The first, sometimes called Good Old Fashioned AI (GOFAI), relies on making inferences from aggregations of symbolic expressions, and consequently is best implemented in Lisp or Prolog. A good example of GOFAI is the work done by Doug Lenat on AM [5], Eurisko [11], and Cyc [12]. Those working on GOFAI could be criticized for designing "brains in vats" with no direct contact with the physical world.

The second approach (subsumption architecture) functions at a lower level, and most of the effort has gone into modelling the non-cognitive aspects of behaviour, and into building robots. A good example of subsumption architecture is the work done by Rodney Brooks on various robots [2], including Cog [3]. (Neural networks are also usually considered AI and also function at a "precognitive" level.) Those working on subsumption architecture or neural networks live in hope that somehow cognition will "emerge"

from these systems.

It would be worthwhile investigating whether a combination of the two approaches might work better than either alone, and whether use of a dataflow language such as Full Metal Jacket might be the best way to combine them. Subsumption architectures depend very much on modules which perform processing on inputs received either from sensors or other modules, and which sending outputs either to other modules or to actuators (see, for example [1]). In other words, they *already* perform dataflow, so a language like Full Metal Jacket seems ideally suited. GOFAI can also be done because Full Metal Jacket maps onto Lisp.

But perhaps the most important point is this: the brain is a very large dataflow computer.

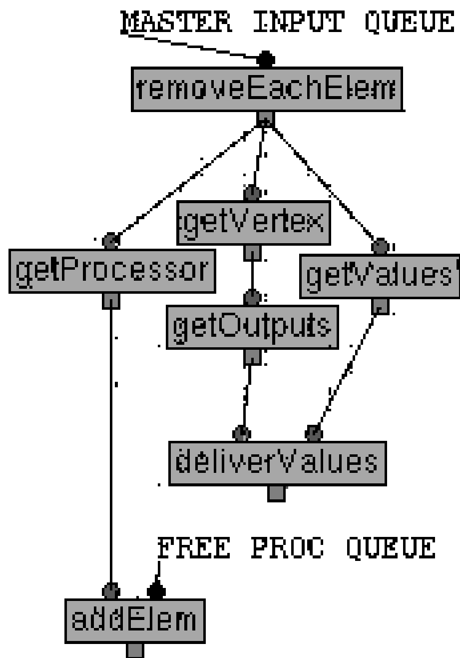

Figure 4: Receipt of results by the master process from other processes, and their delivery to their destination vertices.

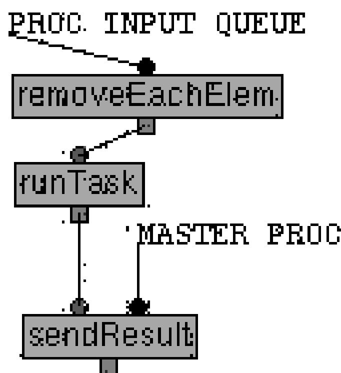

Figure 5: Processing of tasks by other processors.

9. CONCLUSIONS

Although development of Full Metal Jacket is in a very early stage and firm conclusions about it cannot be reached yet, the combination of dataflow and Lisp in the same system, and the potential use of directed graphs as a data structure, seem promising.

The project has already proven to be a useful testing ground for ideas about design of dataflow systems, graphical user interfaces and Lisp; in particular, it has demonstrated the value of CLOS-style object orientation and multi-threading. Use of Mistletoe, with graphical objects containing the code which determines their behavior, has enabled the code to be more modular and better structured.

An important lesson for the Lisp community is that it is important to build graphics, multi-threading, and other requirements of modern computing systems into Lisp distributions for programmers to customize for their own purposes.

10. REFERENCES

[1] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1984.

[2] R. A. Brooks. Fast, cheap and out of control: A robot invasion of the solar system. *Journal of the British Interplanetary Society*, 42:478–485, 1989.

[3] R. A. Brooks, C. Breazeal, M. Marjanovic, B. Scassellati, and M. M. Williamson. The cog project: Building a humanoid robot. *Lecture Notes in Computer Science*, 1562:52–87, 1999.

[4] P. T. Cox. Prograph: a step towards liberating programming from textual conditioning. In *Proceedings of the 1989 IEEE Workshop on Visual Programming*, 1989.

[5] R. Davis and D. B. Lenat. *Knowledge-Based Systems in Artificial Intelligence*. McGraw-Hill, 1982.

[6] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150:1–??, 2001.

[7] J. R. Gurd, C. C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28:34–52, 1985.

[8] D. D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languagse and Computing*, 3:69–101, 1992.

[9] S. O. Kableshkov. *Anthropocentric Approach to Computing and Reactive Machines*. John Wiley and Sons Ltd, 1983.

[10] J. Kodosky. Visual programming using structured dataflow. In *Proceedings of the IEEE Workshop on Visual Languages*, 1991.

[11] D. B. Lenat. Eurisko: A program that learns new heuristics and domain concepts. *Artificial Intelligence*, 21:61–98, 1983.

[12] D. B. Lenat. Cyc: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11), November 1995.

[13] T. Moto-oka, editor. *Fifth Generation Computer Systems*. North-Holland Publishing Company, 1983.

[14] G. M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor. LCS TR-432*. PhD thesis, MIT, Laboratory for Computer Science, August 1988.