Keyboard? How quaint. Visual Dataflow Implemented in Lisp.

Donald Fisk
Barnet
Herts
England
hibou@onetel.com

ABSTRACT

Full Metal Jacket is a general-purpose, homoiconic, strongly-typed, pure visual dataflow language, in which functions are represented as directed acyclic graphs. It is implemented in Emblem, a bytecode-interpreted dialect of Lisp similar to, but simpler than, Common Lisp. Functions in Full Metal Jacket can call functions in Emblem, and vice-versa. After a brief language description, this paper describes the interpreter in detail, how iteration is handled, how the editor handles type checking interactively, and how it detects race conditions.

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

In recent years, many new programming languages (e.g. GoLang, Swift, Rust, Hack, etc.) have been released. As none of these represents a radical departure from already existing languages, it is understandable that the announcement of the development of yet another programming language might be greeted with a degree of skepticism.

However, Full Metal Jacket *is* fundamentally different from any mainstream programming language (including Lisp), in both its outward appearance and its internal computational model. Instead of being expressed in text, its programs are directed acyclic graphs drawn on a two-dimensional canvas, and computations are run, whenever possible, in parallel, with no central flow control. It is strongly typed yet it does not have any variables, and has iteration but does not have any loops. Programmers do not have to learn "yet another syntax": its syntax is not much more complex than Lisp's, and it has an integrated editor which not only is simple to use, but also prevents syntax and type errors. Editing is mostly done with the mouse, by dragging and dropping program elements, and then joining them up. Early work on Full Metal Jacket is described in [5].

Work on Full Metal Jacket only recently resumed after a long break. In the meantime, the underlying Lisp dialect, Emblem, has, however, undergone substantial improvements which have made Full Metal Jacket implementation more straightforward. These include better event handling, X11 graphics programming, and object orientation. Changes to Emblem since [5] have, in general, moved it closer to Common Lisp, except that the object system has been simplified.

At present, Full Metal Jacket is interpreted. Ideally, it would compile onto a dataflow machine if a suitable one existed. (A prototype was built at the University of Manchester [6]. Others were designed by Kableshkov [9] and Papadopoulos [12].) Alternatively, it could be compiled onto a more conventional architecture, such as X86 or X86-64. Full Metal Jacket does go one step further than Lisp in facilitating compilation: not only is the parsing step trivial, optimizations based upon dataflow are also made more straightforward.

When a program runs, values can be thought of as flowing downwards along the edges which connect vertices, where they are transformed. Some vertices contain nested enclosures, in which data flows from ingates, down through vertices, towards outgates.

The intention is for the language to interoperate with Lisp, rather than to replace it. There is some evidence [8] that, although dataflow excels at coarse-grained parallelism, its high overhead makes it less suitable for parallelism at the instruction level.

In this article, following some examples of simple programs, the interpreter is described in some detail. Iteration, type inference, and race condition detection are also covered.

2. RELATED WORK

Full Metal Jacket is not the only visual dataflow language. Three others have been developed which have seen widespread use, namely Prograph [2] which is general-purpose, MAX/MSP [3] [4], which is designed for music and multimedia, and LabVIEW [10] which is designed for programming instruments and devices. Another system, Plumber [1], has been implemented in Lisp. [8] contains a recent survey of many such languages.

There are significant differences between Prograph and Full Metal Jacket: in Prograph, type checking is not done until run time; Prograph places more emphasis on object orientation; *then* and *else* clauses (and cases) are in separate windows; there is a 'syncro' edge which enforces execution

order; and garbage collection is by reference counting.

LabVIEW uses two separate windows: a front panel for user interface objects (these are equivalent to Full Metal Jacket's constants), and a block diagram for code. It also requires extra constructs for iteration and conditional code.

Max/MSP differs from Full Metal Jacket in that it is explicitly object-oriented (with dataflow as message passing); allows feedback; does not comfortably support recursion; execution order is sensitive to program layout; and triggers are sometimes needed to guarantee execution order. Of these three systems, Max/MSP is the least similar to Full Metal Jacket.

3. A BRIEF DESCRIPTION OF THE LANGUAGE

The simplest program, shown in Figure 1, displays the constant "Hello, world!" in a dialog box.

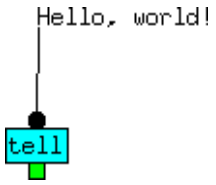

Figure 1: Hello, world!

In the program shown in Figure 2, $2.5sin(0.6) + 5.4$ is calculated.

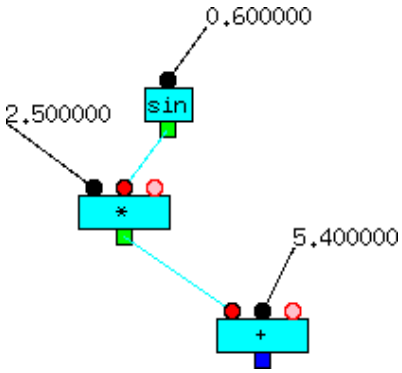

Figure 2: A simple calculation

The dark blue square indicates that the result of + should be output to a dialog box. Vertex inputs are shown as circles and are usually red, and vertex outputs are are shown as squares and are usually green. The edge colors are configurable, and give some indication of type. Here, *cyan* ≡ *Real*.

The third inputs of `*` and `+` are called extra inputs. An extra input allows additional arguments to be added by clicking on it, and can also be used like Common Lisp's `&rest` argument, in which case it will be the head of an edge.

Figure 3 shows the code for `myAppend`, the canonical recursive append. The surrounding square is an enclosure. Ingates are at the top and outgates are at the bottom.

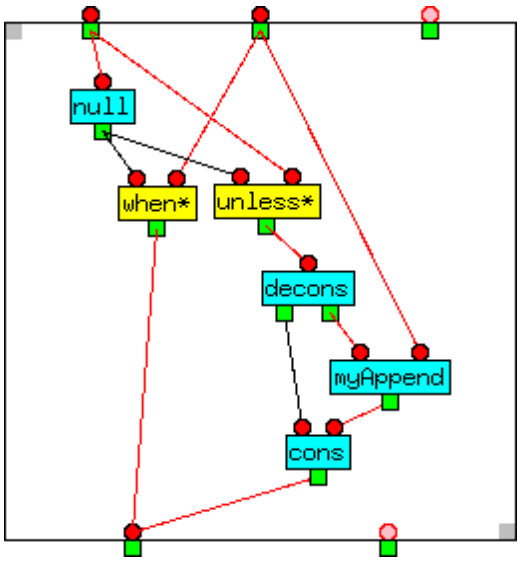

Figure 3: Recursive append.

`when*` outputs its second argument if its first is `T`, otherwise it does not output anything. `unless*` does the opposite. So, if `myAppend`'s first argument is `NIL`, `when*` outputs `myAppend`'s second argument. Otherwise, `unless*` outputs `myAppend`'s first argument, and `decons` splits it into its *car* and *cdr*. The *cdr* and `myAppend`'s second argument then become the inputs to `myAppend`, which is called recursively. The *car* and the result of the recursive call are then *cons*ed and returned.

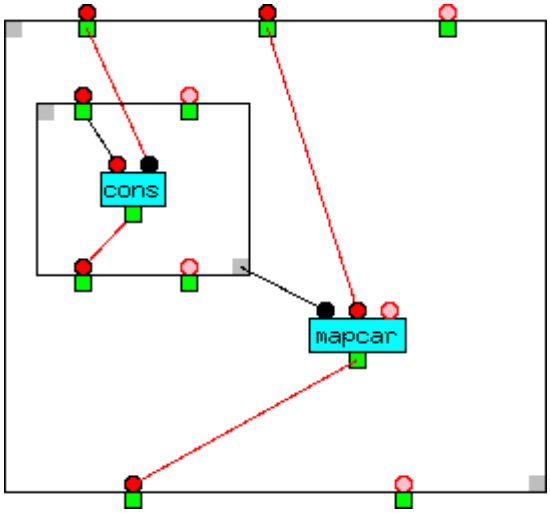

Figure 4: Use of a constant enclosure.

Figure 4 shows the code for `foo`, and is the equivalent of the Lisp

```
(defun foo (x y) (mapcar (lambda (z) (cons z x)) y))
```

The inner enclosure is a constant of the first argument of `mapcar`, and is called as often as the length of `foo`'s (and

mapcar's) second argument, yet it captures foo's first argument precisely once. To make it available for more than one call, it has been made sticky, which is why, like inputs with constants, it is shown in black. Sticky values behave like constants, but change when a new value reaches their input.

4. THE INTERPRETER

A simplified version of the interpreter code, written in Emblem, follows.

4.1 Interpreter Data Structures

Full Metal Jacket has four fundamental classes of objects: Vertex, Constant, Edge, and Enclosure. three other classes: Input, Output, and Gate, are used in constructing them.

A vertex has inputs, a function, and outputs. When a vertex has values with the same tag[1] on all of its inputs, it applies its function to those values. The values returned by the function are then sent with the same tag from the outputs.

```
(defclass Vertex (Any)
  (function type Fun accessor functionOfVertex)
  (inputs type List accessor inputsOfVertex)
  (outputs type List accessor outputsOfVertex))

(defclass Socket (Any)
  (edges type List accessor edgesOfSocket initForm '()))
```

An input usually has a queue of tagged values, but may instead have a sticky value, which is not consumed when the vertex's function is called, but changes when a different value is received, unless it is a constant.

```
(defclass Input (Socket)
  (vertexOrGate type Any accessor vertexOrGateOfInput)
  (taggedValueQueue type Queue accessor taggedValueQueueOfInput)
  (stickyValue type Any accessor stickyValueOfInput)
  (hasStickyValue type Bool accessor inputHasStickyValueP
                  initForm NIL))

(defclass Output (Socket))
```

A constant is a value attached to an vertex's input, which does not change between function calls.

```
(defclass Constant (Any)
  (input type Input accessor inputOfConstant))
```

An edge connects an output (its tail) to an input (its head). Values notionally flow along edges from output to input.

```
(defclass Edge (Any)
  (output type Output accessor outputOfEdge)
  (input type Input accessor inputOfEdge))
```

Enclosures, which are an exact analogue of Lisp's lambdas, are used in defining new functions, and also used within functions, to provide local scope. Each enclosure has ingates and outgates. Within an enclosure, the outputs of ingates are connected to inputs of vertices, and outputs of vertices are connected to the inputs of other vertices, or the inputs of outgates.

```
(defclass Enclosure (Any)
  (ingates type List accessor ingatesOfEnclosure
           initForm '())
  (outgates type List accessor outgatesOfEnclosure
            initForm '())
  (tag type Int accessor tagOfEnclosure initForm 0))
```

Ingates and outgates are gates.

```
(defclass Gate (Any)
  (enclosure type Enclosure accessor enclosureOfGate)
  (input type Input accessor inputOfGate)
  (output type Output accessor outputOfGate))
```

4.2 Interpreter Code

Vertices are executed asynchronously.

As soon as a vertex is ready to be executed by executeVertex, it is added to the task queue. In the current system, tasks are run by runNextTask in the order they appear on it.

```
(setf TASK_QUEUE (new Queue))


(defmacro makeTask (vertex tag args) `(list ,vertex ,tag ,args))


(alias vertexOfTask car)
(alias tagOfTask cadr)
(alias argsOfTask caddr)

(defun runNextTask ()
  (let ((task (takeOffQueue TASK_QUEUE)))
    (if task
        (executeVertex (vertexOfTask task)
                       (tagOfTask task)
                       (argsOfTask task))
      (write "TASK_QUEUE is empty!" $))))

(defun executeVertex (vertex tag argList)
  (do ((args (mvList (apply (functionOfVertex vertex)
                            argList))
             (cdr args))
       (outputs (outputsOfVertex vertex) (cdr outputs)))
      ((null outputs))
      (do ((edges (edgesOfSocket (car outputs))
                  (cdr edges)))
          ((null edges))
          (sendValueToInput (inputOfEdge (car edges))
                            tag
                            (car args)))))
```

Values must be tagged.

Values intended as arguments of a vertex's function are sent to its inputs asynchronously, and possibly out of order, by sendValueToInput, so it is essential to distinguish which values belong to which invocation of the function. This is achieved by accompanying each value intended for the same invocation with the same unique tag, which can be an integer. Because tagged values have to be queued at the input if

[1]Tags, which are used to distinguish different computations which share the same vertex, are explained in detail in Subsection 4.2.

the vertex is not ready to receive them, each input requires a queue for holding them. When every input is found by `everyInputHasAValueP` to have a value with the same tag, the vertex is ready to be executed. Then `extractValuesFromInputs` gets the input values, and `putOnQueue` adds a task to the task queue.

```
(defun sendValueToInput (inputOfDest tag value)
  ;; If the destination input has a sticky value, change it
  ;; to the new value.
  (if (inputHasStickyValueP inputOfDest)
      (setf (stickyValueOfInput inputOfDest) value)
    ;; Otherwise, tag the value and add it to end of
    ;; the input's tagged value queue.
    (putOnQueue (taggedValueQueueOfInput inputOfDest)
                (cons tag value)))
  ;; If the destination input belongs to a vertex,
  ;; rather than a gate, and it has values for all inputs
  ;; with the tag, schedule the vertex to be run
  ;; with those inputs.
  (let ((dest (vertexOrGateOfInput inputOfDest)))
    (when (and (instanceOf dest Vertex)
               (everyInputHasAValueP (inputsOfVertex dest)
                                     tag))
      (putOnQueue TASK_QUEUE
                  (makeTask dest
                            tag
                            (extractValuesFromInputs
                             (inputsOfVertex dest)
                             tag))))))

(defun everyInputHasAValueP (inputs tag)
  (every (lambda (input)
           (or (and (eq (classOf input) ExtraInput)
                    (null (edgesOfSocket input)))
               (inputHasStickyValueP input)
               (assoc tag
                      (elemsOfQueue (taggedValueQueueOfInput
                                     input)))))
         inputs))

;;; This should only be called after verifying
;;; that the values are actually present.   The use of mapcan
;;; permits the use of an extra arg.
(defun extractValuesFromInputs (inputs tag)
  (mapcan (lambda (input)
            (if (inputHasStickyValueP input)
                (list (stickyValueOfInput input))
              (let ((taggedValue
                     (assoc tag (taggedValueQueueOfInput input))))
                (cond (taggedValue
                       (deleteFromQueue (taggedValueQueueOfInput
                                         input)
                                        taggedValue)
                       (list (cdr taggedValue)))
                      ;; If there is no tagged or sticky value,
                      ;; no value should be extracted.
                      (T NIL)))))
          inputs))
```

Arguments must be imported into enclosures synchronously.

It might be thought that, if a vertex's function is defined as an enclosure, it would be possible to import individual input values into the enclosure through their corresponding ingate as soon as they arrive at the calling vertex, without waiting for the other argument values to arrive. However, in general, it is not possible, as when the function is recursively defined, such a value could be transmitted immediately to a vertex with the same enclosure as its definition. This value would then be imported into the enclosure via the same ingate and arrive at the vertex again, and again, and again. For example, this would occur in `myAppend` (Figure 3). Therefore, it is necessary to wait until all the arguments for a given invocation, and therefore with the same tag, are available before startng the enclosure call, although this might delay execution at a few vertices. As it also has to be done when vertex's function is encoded in Lisp, the same mechanism can be used in both cases.

```
(defun importArgsIntoEnclosure (enclosure tag args)
  (mapc (lambda (arg ingate)
          (do ((edges (edgesOfSocket (outputOfGate ingate))
                      (cdr edges)))
              ((null edges))
              (sendValueToInput (inputOfEdge (car edges))
                                tag
                                arg)))
        args
        (ingatesOfEnclosure enclosure)))
```

The tags used in an enclosure are a property of the enclosure and are unique to the enclosure's invocation.

An invocation results in data flowing through an enclosure, or outside any enclosure in the case of top-level computations (i.e. those taking place in the sandbox, Full Metal Jacket's equivalent of Lisp's `read-eval-print` loop.) Each invocation has its own tag for use in that enclosure (or sandbox), supplied on entry.

While values are flowing through an enclosure during a particular invocation, they must all have the same tag, unique to the invocation, in order for the vertices they encounter to execute each time with the correct input data.

During an invocation, the same function might be called from different vertices, each resulting in a different invocation of the same enclosure. In order to keep the computations for the different invocations separate, their values must be provided with different tags when the enclosure is entered, for use during their time in the enclosure. When the enclosure is exited and the value(s) returned to the calling vertex, the original tag on the data received by the vertex's inputs is restored.

As computations within different enclosures are physically separate, there is no problem if they occasionally have the same tag provided that tags are properties of their enclosure.

In `applyEnclosure`, each argument is given a new tag, unique to the invocation, then transmitted along each edge leading from its corresponding ingate.

`applyEnclosure` then waits until a value with the same tag appears at each outgate, before returning those values.

The values returned are then transmitted along each edge leading from the corresponding vertex output.

```
(defun applyEnclosure (enclosure args)
  (let ((newTag (incf (tagOfEnclosure enclosure))))
    (importArgsIntoEnclosure enclosure newTag args)
    ;; Wait for the values to appear at each outgate.
    (do ((inputs (mapcar inputOfGate
                         (outgatesOfEnclosure enclosure))))
        ((everyInputHasAValueP inputs newTag)
         (valuesList (extractValuesFromInputs inputs
                                              newTag)))
        ;; Not there?   Find something else to do.
        (runNextTask))))
```

Functions defined as enclosures in Full Metal Jacket can be called from Lisp.

When a tagged value is sent to a vertex's input, it is put on its tagged value queue. Then, if all the inputs either have an attached constant, or a value in their queue with the same tag, those values are removed from their respective queues to comprise, along with any constants, the argument list. The vertex's function is then applied to the argument list, by calling `apply` if the function was written in Lisp.

If the function was implemented as an enclosure, `applyEnclosure`, Full Metal Jacket's analogue of `apply`, is called instead. The arguments to `applyEnclosure` are the enclosure and the enclosure's inputs.

For example, the definition in Emblem of `myAppend`, generated automatically when the enclosure is saved, is

```
(defun myAppend (x y)
  (applyEnclosure (get 'myAppend 'enclosure) (list x y)))
```

5. ITERATION

`when*` and `unless*` are unusual because, unlike functions, they do not always output a value exactly once when invoked. This suggests that the function concept might be generalized to include, in addition to ordinary functions, not only boolean operators like `when*` and `unless*`, but also emitters, which can output more than once per invocation, and collectors, which can accept inputs more than once before outputting.

Figure 5 illustrates the function `iterReverse`, which reverses its first argument onto its second. If its inputs are `'(a b c)` and `NIL`, the values at various stages of the computation are shown in Table 1.

Output of `strip*`	2nd Input	3rd Input of `collectUntil*`	Accumulator of `collectUntil*`	Output
`(a b c)`	`a`	`NIL`	`NIL` `(a)`	
`(b c)`	`b`	`NIL`	`(b a)`	
`(c)`	`c`	`T`	`(c b a)`	`(c b a)`

Table 1: Iteration in `iterReverse`

Emitters and collectors typically work in tandem, with emitters sending values and collectors receiving those values, or data computed from them, and accumulating them in some manner, until a boolean input changes value, at which point they output their result. Here, `strip*` repeatedly sends a list, stripping off one element at a time, and, after initializing its accumulator to `NIL`, `collectUntil*` receives the `car` of each of these lists, and `cons`es it onto its accumulator (a local storage area), outputting its value after `strip*` has sent all its outputs.

As tagged values are queued, when an emitter is connected to a collector, the collector will process values in the order they were sent by the emitter. This ensures that there is no problem in the collector repeatedly receiving values with the same tag, provided care is taken that it receives the same number of values on its other inputs, except when they are sticky.

The emitters and collectors available to the programmer are still, at present, not settled, and it will require further experimentation before a final selection can be made. One option is to make them compatible with Waters's Series or Curtis's Generators and Gatherers [13], to which they bear some similarity.

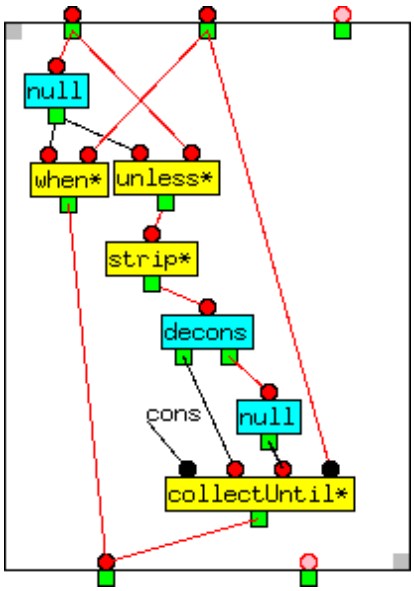

Figure 5: Iterative Reverse.

6. TYPES AND TYPE INFERENCE

Like ML and Haskell, Full Metal Jacket has a Hindley-Milner type system [7] [11], but types are inferred incrementally, and entirely interactively, when edges and constants are added or deleted in the editor, in the course of function definition.

At present (but see subsection 8.1), types in Full Metal Jacket must be defined in Emblem, and are not yet fully integrated into Emblem's object system. Arguments and return values of functions in Emblem, and therefore inputs and outputs of vertices in Full Metal Jacket, are typed.

Composite types can be parameterized, allowing their element types to be specified or inferred, e.g. a list of unknown items can be declared as `(List ?x)` and a list of integers as `(List Int)`.

Examples of type definitions are:

```
(deftype (List ?x) (or NIL (Pair ?x (List ?x))))
(deftype (AList ?x ?y) (List (Pair ?x ?y)))
(deftype (Bag ?x) (AList ?x Int))
(deftype (TaggedValueQueue ?x) (Queue (Pair Int ?x)))
```

The types of inputs and outputs are displayed in Full Metal Jacket's editor when the pointer is placed over them. Whenever an attempt is made to connect an output, or add a

constant, to an input, the types are checked, and unless the output's type matches the input's type, the programmer is prevented from making the connexion or adding the constant. Successful matches can result in type variables being bound (by means of a process similar to Prolog's unification), or types made more specific. Type matching and inference occur while the program is being edited, every time an edge is added, and also when one is deleted, in which case the type variable might again become undetermined.

6.1 Type Matching Algorithm

The basic type-matching algorithm matches an edge's output type to its input type, resulting in either a list of bindings for the variables contained in the types if the match is successful, or the symbol FAIL.

The cases are:

A. Match of two type constants. This results in either success (`NIL`) if the output type is equal to or a subtype of the input type, or failure (`FAIL`) otherwise.

B. Match with a type variable. The match results in a new binding unless the match is with the same variable, in which case `NIL` is returned.

C. Match of two function types. The match is then applied to the argument and return types.

D. Match with the same parameterized type. The parameters are matched recursively.

E. Match with a different parameterized type. If one is a subtype of the other, the match is repeated using the more specific type's definition, after the appropriate variable substitutions. Otherwise, the match fails.

Case	Output type	Input type	Bindings
A	`List`	`Str`	`FAIL`
A	`Str`	`Str`	`NIL`
B	`Str`	`?x`	`?x` → `Str`
B	`(List ?x)`	`?y`	`?y` → `(List ?x)`
B	`?x`	`?x`	`NIL`
B	`?x`	`?y`	`?y` → `?x`
D	`(List Int)`	`(List ?x)`	`?x` → `Int`
E	`(Bag Sym)`	`(List (Pair ?y ?x))`	`?x` → `Int` `?y` → `Sym`

Table 2: Type matching examples

The output of the matcher is the list of bindings for the type variables of the edge's output and input. (See Table 2.)

6.2 Type Inference

Type inference is performed whenever an edge or constant is connected to an output, as follows:

A. If there are any type variables shared between an edge's output and input, unique variables are substituted for them before matching. For example, if output type `(Bag ?x)` and input type `(List (Pair ?y ?x))` are matched, the type variables are renamed, giving (e.g.) `(Bag ?17)` and `(List (Pair ?18 ?19))`.

B. Matching is then performed. The bindings returned by the match algorithm described above are `?19` → `Int`, `?18` → `?17`.

C. The bindings for the output and input types are then separately extracted, giving `NIL` and `?19` → `Int`, `?18` → `?17` respectively.

D. Finally, the original names replace the substitute names, giving `NIL` and `?x` → `Int`, `?y` → `?x` respectively.

E. Type variables are shared among a vertex's input and output types, so if one becomes bound to a particular value on an input or output, it also becomes bound to the same value on all other inputs and outputs on the same vertex.

F. When the type of an input or output changes, type inference is applied along the edges connected to them.

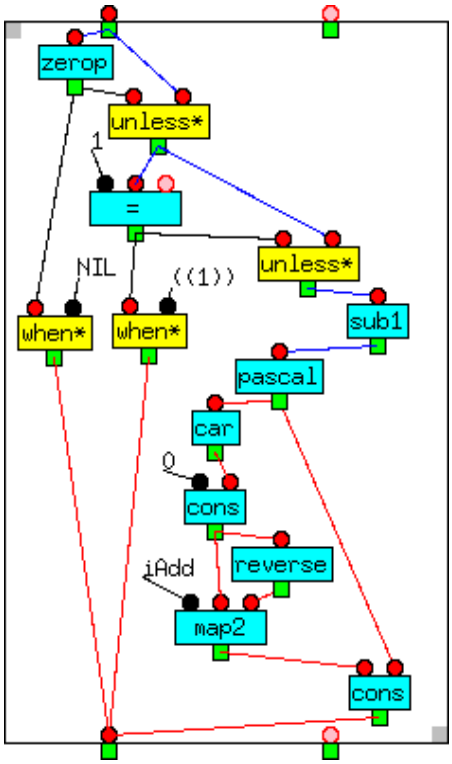

Figure 6: Code for Pascal's triangle.

6.3 An Example

`pascal`, shown in Figure 6, computes Pascal's triangle. For example, `(pascal 5)` returns `((1 4 6 4 1) (1 3 3 1) (1 2 1) (1 1) (1))`.

`pascal` takes an `Int` as input. Here, the types of the values are inferred statically.

Given the vertex input and output types shown in Table 3, the edge types to be matched will be those shown in Table 4. Together with the types of the constants (shown in Table

5), the types of all the variables can be inferred, and this is shown in Table 6.

Input Types	Function	Output Types
`Int`	`pascal`	`?a`
`(List ?b)`	`car`	`?b`
`?c` `(List ?c)`	$\texttt{cons}_a$	`(List ?c)`
`(List ?d)`	`reverse`	`(List ?d)`

`(?e ?f)` $\rightarrow$ `(?g)`
`(List ?e)`
`(List ?f)`
	`map2`	`(List ?g)`

`?h` `(List ?h)`	$\texttt{cons}_b$	`(List ?h)`

Table 3: Vertex types in `pascal`

Function	Output Type	Input Type	Function
`pascal`	`?a`	`(List ?b)`	`car`
`car`	`?b`	`(List ?c)`	$\texttt{cons}_a$
$\texttt{cons}_a$	`(List ?c)`	`(List ?d)`	`reverse`
$\texttt{cons}_a$	`(List ?c)`	`(List ?e)`	`map2`
`reverse`	`(List ?d)`	`(List ?f)`	`map2`
`map2`	`(List ?g)`	`?h`	$\texttt{cons}_b$
`pascal`	`?a`	`(List ?h)`	$\texttt{cons}_b$

Table 4: Edge types in `pascal`

Value	Type	Input Type	Fn.
`0`	`Int`	`?c`	$\texttt{cons}_a$
`iAdd`	`(Int Int)` $\rightarrow$ `(Int)`	`(?e ?f)` $\rightarrow$ `(?g)`	map2

Table 5: Constant types in `pascal`

Type Variable	Type
`?a`	`(List (List Int))`
`?b`	`(List Int)`
`?c`	`Int`
`?d`	`Int`
`?e`	`Int`
`?f`	`Int`
`?g`	`Int`
`?h`	`(List Int)`

Table 6: Inferred types in `pascal`

This, reassuringly, is consistent with the types returned in the other two paths through the `pascal` function, which return `NIL` and `'((1))`.

7. RACE CONDITION DETECTION

Race conditions occur when two or more values are written to the same memory location. The final value then depends on the order in which they are written, which is unsatisfactory. In Full Metal Jacket, if more than one edge is connected to the same input, a potential race condition occurs. Whether it is a true race condition can be detected fairly straightforwardly.

Data follows one out of one or more mutually exclusive *streams* through an enclosure. The stream containing a given vertex can be found by searching downstream from it along the edges leaving its outputs, and then back upstream at each vertex encountered, along *its* edges, marking the vertices as we go. All those vertices, and any edges connecting them, are then on the same stream. If data flow through a vertex, data also must be flowing through other vertices in the same stream on the same enclosure invocation. When two or more edges converge on the same input, and they are from vertices in the same stream, there is a race condition, and one of the edges should be disallowed.

The stream detection mechanism also solves a well-known problem with visual programming: how to remove clutter. Functions do not have to become very large before they become difficult to read, due to many edges crossing each other. Within the editor, clicking on a particular vertex hides all the vertices and edges except those in the same stream.

A further use for stream detection is in detecting gaps in unit test coverage.

8. CONCLUSIONS AND FUTURE WORK

Development of Full Metal Jacket is still incomplete. While it is already possible to implement and run some small programs containing nested function calls, others programs have been found to be difficult to implement without important features still absent from the language: in particular, the ability to extend the type system from within the language, and a more comprehensive set of emitters and collectors.

The ability to take advantage of the language's homoiconicity should also be added. Other features missing from Full Metal Jacket's environment include a compiler (initially, generating Emblem bytecode, for programmer-selected functions), and interactive debugging capabilities.

Learning to think in a dataflow language should not be considered a problem, but a worthwhile challenge, also present in any other programming paradigm switch.

8.1 Type and Class Hierarchy Extension

At present, the Emblem class hierarchy is displayed as a tree (Emblem has *single* inheritance), with each class displayed as a vertex, but this cannot yet be extended from inside Full Metal Jacket. Types and classes should ideally be merged, with the system capable of handling the two different ways of extending them: adding fields to objects, and abstracting and making types more specific. For example, the class hierarchy contains

`Any` $\rightarrow$ `Graphical` $\rightarrow$ `Shape` $\rightarrow$ `Rectangle`
`Any` $\rightarrow$ `Queue`

These are defined by adding extra fields. It should also be possible to define a type hierarchy, which would contain (see Section 6)

`Pair` $\rightarrow$ `List` $\rightarrow$ `AList` $\rightarrow$ `Bag`
`Queue` $\rightarrow$ `TaggedValueQueue`

The subtypes above share the underlying data structure of

their parent types, the `cons`-cell and `Queue` respectively, but a `List` requires its `cdr` also to be of type `List`, an `AList` requires all of its elements to be `Pair`s, and a `Bag` requires the `cdr` of each of its elements to be `Int`. Similarly, a tagged value queue is a `Queue` of `Pair`s, of which the `car` is an `Int`. (A `Queue` is an object containing a pointer to a list of elements, and a pointer to their last `cons`-cell. Alternatively, it could have been implemented by subtyping `Pair`.)

It will be noticed that Emblem's `typedef` macro (see Section 6) resembles Scheme's define macro when that is used to define functions, suggesting that types be defined graphically in Full Metal Jacket as enclosures, with more primitive types as vertices and type variables as gates. This hints at a deep equivalence between code and data.

8.2 Homoiconicity

Until now, Lisp and Prolog, and only those languages, have been meaningfully homoiconic, i.e. able to treat code written in them as data, transform it, and reason about it, and to treat data as code, and execute it. Lisp macros are the most widespread use of this, but more generally, code can be generated on the fly (data becomes code) or reasoned about (code becomes data).

A Full Metal Jacket program is a directed graph, which is the most general data structure. The intention is to exploit this feature, similar to how Lisp uses lists, and Prolog uses terms, to represent both programs and data. However, it is less straightforward: directed graph elements have a more complex structure, and are more specific to code. While this does not preclude the incorporation of *macros* into the language, use of the same structures for *data* might seem less natural. Vertices, edges and enclosures can, however, be generalized to objects without the program-specific fields (such as tagged value queues and types), and then specialized by restricting inputs and outputs each to one. This has already been done with classes, so the class hierarchy can be displayed as a tree.

9. REFERENCES

[1] Seika Abe. Plumber - A Higher Order Data Flow Visual Programming Language in Lisp *International Lisp Conference*, 2012.

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

[3] Peter Elsea. Max and Programming (July 2007) Retrieved 3rd March 2015 from `http://peterelsea.com/Maxtuts_advanced/Max&Programming.pdf`

[4] Peter Elsea. Messages and Structure in Max Patches (February 2011) Retrieved 3rd March 2015 from `http://peterelsea.com/Maxtuts_advanced/Messages%20and%20Structure.pdf`

[5] Donald Fisk. Full Metal Jacket: A Pure Visual Dataflow Language Built on Top of Lisp. *International Lisp Conference*, 2003.

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

[7] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[8] W. M. Johnston, J.R. Paul Hanna, and R.J. Millar. Advances in Dataflow Programming Languages. *ACM Computing Surveys, Vol. 36, No. 1*, March 2004.

[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] Milner, R. A Theory of Type Polymorphism in Programming *Journal of Computer and System Science*, 17:348–374, 1978.

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

[13] Guy Steele. *Common Lisp: The Language. Second Edition Digital Press*, 1990.