Motivation for the Crow Programming Language

Taybin Rutkin

November 6th, 1999

1 Introduction to Message Driven Computing

Message Driven Computing is one of many paradigms for parallel computing. It is based on the idea that all communication between computational events is by means of message passing [1]. Messages are passed between locations or left at the sending location. A collection of messages at a location, which match a pattern, can trigger a computational event. This event will then consume some or all of the messages, and send messages of its own.

1.1 What is interesting about MDC

MDC is at a lower level than the other paradigms. It is possible to change the grain size of computational events in order to increase the parallelism. Christopher writes that he observed a difference of over 9000% in the run time when the grain size in an algorithm is changed.

Another reason MDC is lower level is that other programming languages can be encoded in MDC[1]. Christopher [2] writes that languages such as FP[2] have been translated already. Translating a lower level sequential language like C would be difficult, but a language like Pascal would merely be messy[3].

What I found more interesting though was the ease with which techniques from other parallel paradigms could be translated into MDC. Christopher mostly describes Actors, another paradigm, but he mentions other techniques such as futures, I-Structures, and active messages[4]. He writes that locations often metamorphose from Actors to processes and back to Actors.

[1] The language used for the encodings is LLMDC/C, or Low Level MDC built atop C

[2] FP is a side-effect free, combinator style language, described in: "Can programming be liberated from the von Neumann style?", John Backus, Communications of the ACM, 21, 8, pp.613–641, 1978.

[3] C is difficult because of its pointers. Finding a way to correctly implement them when they are spread across separate computers would be very painful.

[4] These are different messaging paradigms.

1.2 Overview of LLMDC/C

LLMDC/C, or MDC-90, as it is more commonly called, is the MDC paradigm implemented as an extension to C. It is implemented [4] with a preprocessor and a run time system.

There are several additional operators, data types, and functions I leave out. I just keep enough to understand the rudiments of the language.

1.2.1 Locations

A location is a named place where messages are sent and computation occurs. It is denoted as [S, X, Y] where S is a symbol and X and Y are unsigned long integers. The X and Y are used to differeniate the different locations from each other. They are a handy way of constructing loops to affect each location. A location is assigned to a node in a multicomputer. The runtime system will do that automatically, but there is a method for the programmer to decide what node goes where. This is used to build hypercubes and other structures with the nodes.

1.2.2 Symbols

A SYMBOL is a new data type. In MDC-90, the analog to a pointer is a symbol. Variables may be declared of type SYMBOL, and new SYMBOL values may be created by calling the function Symbol.

Symbols are unique values, that are used to construct names of locations. The symbol corresponds to the name of a potentially distributed object[5].

1.2.3 Messages

Messages are declared similar to **struct**s.

message msg_id { *members* };

They can also be declared with no members, or as equivalent to a previously defined message. Members can be scalars or fixed size arrays only. Pointers, structures, and unions are excluded. The allowable data types are

[5]By potentially distributed object, I do not mean that the object is broken up and distributed, but rather that, based on the messages sent to the object, it may at some future point be distributed.

char, int, float, and double, which were taken from C. In addition, MDC-90 adds the following types: SYMBOL, LOCUS, PORTAL, MSGTYPE, CLASS and ANY. The non-C types are explained below.

1.2.4 Behaviors

A behavior is a pattern and a procedure body. When a pattern is matched at a location, the procedure may be executed. There are so many combinations for behaviors that it is difficult to give an overall example. The documentation for MDC-90 instead gives the context free grammar.

behavior *pattern* `=>` { *procedure body* }[6]

There are three parts of patterns that can be defined. They are the message component list, the relation operator, and the precedence.

Message Component List The syntax for this is

> msg_id *id1, id2, . . . , idn* ;
>
> This declares *id1, id2, . . . , idn* to be of type msg_id. The pattern only matches if there are at least *n* message waiting at the location. The *n* messages will then be bound to the identifiers and if the full pattern matches, be sent to the behavior body.

Relation Operator The syntax is

> msg_id *relop* number ;
>
> This pattern element will match only if the number of messages at the location with the msg_id bears the relop to number. The permissible relops are `== != > < >= <=`.

Precedence The syntax is

> **precedence** *value* ;
>
> This gives the precedence of the behavior. In the event that more

[6] `=>` is an operator in MDC-90. I do not use it in the meta-language/context-free grammer sense.

than one pattern matches, the pattern with the highest precedent will be launched. The *number* can be as high as 255.

So a full example could look like this

behavior msgType1 m1, m2 ; msgType2 == 0 ; **precedence** 100 =>
{ ... }

According to the grammar, the component list and the relation can be chained onto each other multiple times. So very complex patterns can be created. This makes the likelihood of two or more patterns matching very high, so the precedence number is necessary.

1.2.5 Classes

Classes in MDC-90 are different from the OOP sense of class. A SYMBOL is created to be of a particular class. A location is of the same class as its symbol. Behaviors are also declared to be associated with particular classes, and a behavior can only be executed at a location of some class with with it is associated. A class is declared

class *class_name1, ...* :

This declares that all behaviors encountered up to the next **class** declaration, or end of the program, will be associated with the classes named in the list. This way, only certain behaviors are linked with certain classes[7].

There is also the **subclass** keyword.

subclass class1 => class2 ;

This declares class1 and class2 to be classes. It also says that class1 is a subclass of class2. Class1 inherits all behaviors from class2.

1.2.6 Message Control

To send a message to a location, one would use the following form.

send *message_expression* **to** [*expr1, ...*] ;

[7] Apparently, messages are associated in the same way.

The *message_expression* is an expression that evaluates to a message. The message is then sent to the location whose name is constructed from expression expr1 and subsequent expressions.

The **leave** statement works as a send to oneself. It adds a message to the operating locations message list.

A message that has already been received can be hidden from the message list with the **delay** statement. This removes it from pattern matching until the **undelay** statement replaces it.

1.2.7 Miscellaneous

Execution in MDC-90 is started by a special behavior.

initial behavior => { *body* }

The body is a function body. It receives *argc* and *argv* just like the C function main().

2 Icon

Icon is a general purpose programming language descended from SNOBOL. Icon came about in 1976 with the idea that the control structures used in pattern matching could be integrated with conventional control structures to build a more coherent and powerful language [6]. Since then it has been implemented in Ratfor, C, Java, and RTL[8].

2.1 Icon Features

These are some of the features that make Icon interesting. There are some other features, such as Icon's string scanning expression, that I do not describe. These, however interesting, are not directly applicable to parallelism and did not affect my choice of Icon.

[8]The Run Time Language is a superset of C with higher level features to support some of Icon's semantics. It was developed for the purpose of implementing an optimizing compiler for Icon [11].

2.1.1 Generators

In Icon, an expression either returns a value, fails, or returns multiple values [3][9]. The expressions that can return multiple values are called generators. Icon has several operators that will evaluate an expression and return multiple values.

An example of this is the find function. It takes two arguments, a string and a substring, and returns the character number of where the substring starts[10]. The first time find is called, it returns the first position. If called again from within the same expression, it returns the next position. This continues until it can not locate any more substrings and fails. So this behavior can be used like this:

world := "Hello World"
every write(find("o", world))

This produces the output:
5
8
By itself, the find would produce 5. The keyword **every** causes a generator to produce every possible result.

2.1.2 Goal-Directed Evaluation

Failure during the evaluation of an expression causes it to backtrack to look for other possibilities [7]. This way, failure of one part of an expression does not necessarily cause the entire expression to fail. An example using the example for generators:

if find("o", world) > 7 **then** write("Success")

The find produces a 5 and suspends and the 7 produces 7. The two are then compared and the comparison fails. So the interpreter backtracks to the find which then produces 8. The expression then succeeds. If the comparison was against 3 for example, the evaluation would not have backtracked.

This can get much more complex with multiple generators, complicated expressions, and many operators tying them all together. Griswold says that

[9] Oddly enough, Thomas Christopher, the creator of MDC, is also big in the Icon world. I did not make the connection though until I looked at his *Curriculum Vita*.

[10] Actually, it returns the slice number. But that is another wholly different topic on Icon's string handling. Something which is not germane to parallelism

in practice it is not a very expensive operation[11] despite appearances. The main advantage is very compact programming and very expressive comparisons.

2.1.3 Miscellaneous

Typeless Variables Variables do not have types. Instead, the value determines the type. There are no type declarations either. A value with one type can be converted into a different type at runtime. So a string that consisted of numbers would automaticly be converted to an integer or long if it was necessary [9]. It still has error checking, however. If a value cannot be converted in a meaningful way, the program is terminated and a diagnostic message is printed.

High Level Data Types Icon supports Lists, Sets, Tables, and Strings. This includes the appropriate function support and dynamic type conversion where applicable.

Garbage Collection Memory is dynamically allocated and reclaimed. A string can take up as much space as the machine permits [5].

Libraries Icon has a large collection of libraries, including cross platform support for windowing and graphics [8]. I would like to maintain compatibility with these.

2.2 Why I chose Icon instead of a different language

There were several languages that I considered before settling on Icon. I never considered Java but several people have asked me why I did not decide to extend it. Another language that I was playing with at the time was Python.

2.2.1 Java

One reason I did not want to work on Java is because it already has multitasking support. The problems of concurrency and OOP seems to have a useful solution in Java already. Adding MDC would be redundant.

Compared to Icon, I do not find Java very interesting. Mostly, I wanted to work on a procedural language and stay away from the complexities of OOP. For the same reason I did not look at any functional languages. There

[11] Memory-wise and speed-wise

is one interesting similarity between Java and Icon, however. They both use a byte compiled/virtual machine architecture.

2.2.2 Python

Feature-wise, Python is very similar to Icon. They are both dynamically typed. They both support high level data types. One reason I chose to stay away from Python is that it has OO features and I had been looking for a more procedural language. Another is that Python is interactive and I want a more traditional write/compile cycle.

2.3 Features of Icon that will enhance MDC

I think that backtracking would allow for some interesting techniques when mixed with behaviors and pattern matching. Generators will also add much complexity but I think that they too can be used for novel solutions.

3 Crow

Crow[12] will be MDC on top of Icon. Having high level data types will make programming much more enjoyable and easier.

I think that restricting programmers to one true paradigm is pointless. What I want to do is allow a programmer to write a traditional Icon program, and where necessary, use MDC if it made the algorithm more clear. One example is a graphical application that needs to update several separate windows. Each window has a separate process to update them concurrently[13]. I think that MDC will be useful for the main program loop to communicate with the separate processes.

As a real world example, I wrote a graphical sort[14] program in Icon as a learning problem. One problem that showed up is that the sorting takes place outside of the event loop. So it is impossible to halt the sort once it has started without resorting to killing the program. It seems that a parallel solution would resolve this gracefully. The event loop would still be running separately from the sort algorithm. Then if the user wanted to halt a sort, a

[12] I named it Crow after a character from the TV show MST3K. This is how Guido Van Rossum named Python. Sounds better than HLMDC/Icon I think.

[13] There is another project called MT Icon that might appear similar at first. However, the two are very different. The author himself states that MT Icon is not a concurrent programming language [10]. Instead it is just multiple, tightly-coupled Icon programs that operate in the same memory space and can communicate with each other.

[14] This was completely unnecessary. Icon has a sort function for linked lists built in.

message is sent to the sort, which would be running as a behavior, to stop. This seems to be an elegant solution to the problem. It would also allow other program activity to take place during the sort.

I am aware that most event driven applications have this problem and solve it without using parallel languages. But this seems to be an easy and elegant solution.

3.1 Goals for Crow

I think that Crow will be useful for a large group of different programming problems. Crow will have the generality of Icon built in with the additional paradigm of MDC. So Icon's original problem set is included. On top of that, Crow will also do parallel applications. I think that this combination will make it easy to take non–parallel algorithms and distribute them across systems in order to speed them up.

3.2 Additions to Icon

These are the new data types, operators, and functions that I will add to Icon. It appears to be fairly easy to add additional data types to Icon, judging from the implementation. The source code is very modular, and many features could be taken out or added without affecting the other features.

3.2.1 Additional Datatypes

Locations These would correspond to the locations in MDC-90. In MDC-90, one can also specify what node a location is assigned to. For Crow, giving that option would violate the high level spirit of Icon. Denying this option also would make the program more portable. The location could be implemented as a process on the local computer or on a separate node in a multicomputer.

Behaviors These would be in addition to the procedures that Icon already has. They should use a similar syntax to procedure declarations except that they would contain the pattern matching information as well.

Messages This could be a data type that other types would be converted to at runtime. This would only be necessary if messages needed special handling in the implementation. Otherwise, the message datatype is not needed.

Portals Because of Icon's runtime type inference and conversion, a portal type will not be necessary.

3.2.2 Additional Operators

Icon already has very many operators. So I am hesitant to add any more. I would not want to add any more reserved words either. I can add functions to provide the functionality however. So instead of

send *message* **to** *location*

Crow could use send(*message*, *location*) which would work the same. The MDC-90 syntax could not be kept anyway, since **to** has a very different meaning in Icon.

3.2.3 Additional Keywords

Templates I am considering a template mechanism to help write behaviors that are very similar to each other.

Atoms Atoms would be a built in semaphore that would lock variables. This seems essential for a concurrent language.

Classes MDC-90's class system is too unwieldy for larger systems. Crow would use a simpler scheme to link behaviors with locations.

Initial Behavior It seems that for compatibility and greater flexibility, it would be better to not have an **initial behavior**. Instead, Icon's **procedure main** will be used to set up the initial program state.

References

[1] Thomas W. Christopher. *Experience With Message Driven Computing and the Language LLMDC/C*, chapter 2, pages 15–28. Illinois Institute of Technology, 1990. `http://www.cs.iit.edu/~mdc/docs/mdc/exllmdc.ps`.

[2] Thomas W. Christopher. *Message Driven Computing and its Relationship to ACTORS*, chapter 3, pages 41–46. Illinois Institute of Technology, 1994. `http://www.cs.iit.edu/~mdc/docs/mdc/objorint.ps`.

[3] Thomas W. Christopher. *Icon Programming Language Handbook*. Tools of Computing, beta edition, 1996. http://www.toolsofcomputing.com/iconprog.pdf.

[4] Thomas W. Christopher and Gregory A. Freitag. *MDC-90 Language Description*. Illinois Institute of Technology. http://www.cs.iit.edu/~mdc/mdc90_manual.html.

[5] Ralph E. Griswold. String allocation in Icon. Technical report, Department of Computer Science, The University of Arizona, May 1996. http://www.cs.arizona.edu/icon/docs/ipd277.htm.

[6] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Princeton University Press, 1986.

[7] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Peer-To-Peer Communications, 3rd edition, 1997.

[8] Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend. *Graphics Programming in Icon*. Peer-To-Peer Communications, 1998.

[9] Ralph E. Griswold and Kenneth Walker. Type inference in the Icon programming language. Technical report, Department of Computer Science, The University of Arizona, March 1996. http://www.cs.arizona.edu/icon/ftp/doc/tr93_32.pdf.

[10] Clinton L. Jeffery. *The MT Icon Interpreter*, November 1993. http://www.cs.arizona.edu/icon/docs/ipd169.htm.

[11] Kenneth Walker. The run-time implementation language for Icon. Technical report, Department of Computer Science, The University of Arizona, March 1994. http://www.cs.arizona.edu/icon/docs/ipd261.ps.