An Overview of the Crow Programming Language

Taybin Rutkin

May 3, 2000

1 Messaging Algorithms

One of the biggest puzzles in the field of concurrency is how to write concurrent algorithms. One of the solutions is to try and split algorithms into smaller parts that can be executed simultaneously. Since these mini-algorithms remain part of a larger algorithm, but presumably scattered across a distributed computer, they need to communicate with each other. This is done through *messages*. Messages are just the name for the unit of communication.

There are several messaging paradigms. They differ in the contents of the message, the semantics of the messaging system, or both. ACTORS, a messaging language, uses a single queue for all of its messages and its messages contain a parametrized list of arguments [2]. Each procedure then builds up a backlog of parameters and executes them when it can. In this way, concurrency is achieved.

MDC[1], which was influenced by ACTORS, uses several queues, pattern checking, and its messages contain data structures [1]. Instead of having procedures which have backlogs of parameters to execute over, it has *locations*. A location can be thought of as an abstract point that contains several procedures, here called *behaviors*, and has several queues instead of just one, as in the case of ACTORS. There is a one queue for each message type. The location looks at the queues and checks for *patterns*. A pattern is a combination of relational operators that check for a certain number of messages in each

[1]Message Driven Computing.

queue. Each behavior is defined by its pattern. When a pattern matches, its behavior is executed.

The behavior then makes use of the messages that matched the pattern. This can be compared to AMDC[2], the successor to MDC. AMDC's messages contain the procedure to be executed and the location contains the data to be processed [3]. The data was placed there by a previous message.

Crow uses the MDC paradigm because it was the closest to my vision of parallel computing. I thought that a good way to do parallel programming would be to have blocks of code that would communicate with other blocks and tell them when they could run or what was necessary for them to run. MDC was the closest paradigm that fit what I was thinking of.

2 Icon

The problem I had with MDC-90[3] is that its algorithms were obfuscated by its C roots [5]. Two much space was taken up with memory allocation and the syntax was rather unfriendly. It seemed that the algorithms could be more understandable if a more concise, more high-level[4] language was used.

There were several languages that fit the requirements, most notably Python [9], but the Icon programming language seemed to be the best choice [7]. In addition to the high-level data structures, of which Icon has plenty, it supported several unusual control structures. These unusual control structures included *generators* and *backtracking*.

A generator is an expression that evaluates to one or more values. The simplest example is the `to` keyword. The expression `1 to 5` represents all the numbers from 1 to 5. By prepending the `every`[5] keyword to the expression, it causes the machine to evaluate the expression, save its state, and then evaluate the expression again [6]. So a complete example, `every i := 1 to 5`, causes `i` to be assigned to 1 through 5. This may not appear to be immediately useful, but by appending the `do` keyword, followed by another expression, the equivalent of `a for`-loop can be created.

[2]Active Message Driven Computing.

[3]MDC-90 is the implementation of the MDC paradigm. It is a superset of the C language.

[4]By high-level, I mean a languge with garbage collection and high level datatypes, such as lists, built into the language.

[5]The `every` keyword causes all possible results to be generated.

This feature allows very concise programs to be written. The `!` prefix operator generates all the nodes in a list [4].[6] So the one line expression, `every !somelist := 0`, initializes every node of `somelist` to zero. There are several generators built into the language and procedures can be turn into generators by using the keyword `suspend` instead of `return`.

Generators allow for another one of Icon's powerful features. There are two variants of backtracking: *control backtracking* and *data backtracking.* Expanding on the generator's example, `i := 1 to 5 & i > 3`, demonstrates control backtracking.[7] This expression chains two sub-expressions[8] together, forcing both to succeed for the entire expression to succeed.

The first sub-expression, `i := 1 to 5`, assigns 1 to `i` and succeeds[9]. But the second sub-expression fails because `i` is smaller than 3. So execution returns to the first sub-expression. This continues until both sub-expressions have succeeded. The result of the expression is that 4 is assigned to `i`. Control backtracking allows very expressive control expressions. It also allows for *goal-directed evaluation.* This is where a range of data is scanned to see if certain parts fit a criteria.

But what if the second sub-expression was `i > 7` instead of `i > 3`? In that case, 5 is assigned to `i`. This is because the assignment in `i := 1 to 5` is not reversible. The answer to this problem is data backtracking, wherein if an expression fails, any assignments are reversed and the variables are restored to their original states. This is done with `<-`, the reversible assignment operator. So the old example is redone: `i <- 1 to 5 & i > 7`. In this case, `i` is restored to its previous state each time the expression fails. If the expression succeeded at any point, then the assignment would remain. This allows one to write side-effect free expressions.

Generators and backtracking made Icon a very interesting language.[10] It seemed that those features, combined with MDC, would present a very attractive method for writing concurrent applications.

[6] It works on other types of structures as well (ie. strings, tables, records).

[7] There are more succient ways to write this expression, but they require other operators.

[8] Coming from a C background, one might think that the two expressions were `1 to 5` and `i > 3`. Not so. They are `i := 1 to 5` and `i > 3`.

[9] Icon, because of its logical features, uses *success* versus *failure* instead of *true* versus *false.* No simple boolean logic here, folks.

[10] Even if Icon was not a high-level language, I found its syntax much nicer and more consistent than C's syntax.

```
message print(text)

class OUTPUT
behavior(print p1)
    write(p1.text || " " || here.X)
end

procedure main()
    local bottle, s
    bottle := print("Hello World")
    s := symbol("OUTPUT")
    every i := 1 to 3 do
        send(bottle, <s,i>)
end
```

Figure 1: A simple helloworld to show some semantics.

3 Crow

After deciding on Icon as a base, I added the features that MDC-90 gave to C to Icon. There were some features that I left out that were redundant with Icon's abilities. Since one of my goals was to make MDC-90 into a high-level language, I also left out some features that were too low-level. This included specifying where in a supercomputer a node was located and an additional specifier for a location.

3.1 Semantics

A location is specified through a special syntax: `<S,X>`.[11] The `S` is a unique *symbol* which was returned from the `symbol()` procedure. The symbol's *classname* is passed to `symbol()`. A class is how behaviors are grouped together. This is how each location is linked with a group of behaviors. The `X` is an integer. In the current implementation, it can be as large as 255.

The combination of `S` and `X` form a location. It is enough to change either the `S` or the `X` to create a unique location. So one symbol with different `X`'s

[11]In MDC-90, there was also a `Y`, so the specifier looked like `[S,X,Y]`. I left it out for simplicity.

forms multiple locations. The locations are all of the same class though. Figure 1 demonstrates this.

Classes can also inherit behaviors from superclasses. The syntax for this was lifted from Idol, an object-oriented version of Icon [8]. I am not sure how strong the relation is between MDC classes and OOP classes. This is because MDC classes do not have a single data structure that they are members of and act upon. They are grouped around a collection of queues.

The `symbol()` procedure shows how I tried to integrate MDC with Icon. `symbol()` is a generator that will return a new symbol whenever it is evaluated. So a possible use of this is: `every !somelist := symbol("someclass")`. Every node in `somelist` will be assigned a symbol that points to a location of type `someclass`.

Patterns are part assignment and part relational operator. Figure 1 shows an example. The behavior's pattern succeeds if there is one or more `print` messages. If there are, the first one is removed from the queue and is assigned to `p1`. If the programmer did not want an assignment to take place, the line `behavior(print >= 1)` would have worked. In this case, the `print` message would have stayed on the queue. Since patterns can overlap, there is also a `precedence` keyword which is followed by an integer. In the case of overlapping patterns, the one with the larger precedence is executed.

Messages are sent through the `send()` procedure. `send()` takes two arguments: a destination, and a message. A destination can be either a location or a *portal*. A portal is a method of indirection for locations. A portal is made of two values: a destination, and a message type.[12] When a portal is used in place of a location, the message being sent is converted to the message type of the portal and then redirected to the portal's location. The use for this is subtle, but there is a need for location indirection.

3.2 Implementation

The current implementation of Crow only simulates concurrency. It loops through a hash table of locations, checking to see if any patterns at each location match.[13]

The output for figure 1 is:

[12]Very similar to the `send()` procedure.

[13]As an optimazation, it actually loops through a list of locations that have had messages sent to them. This list, generally, is shorter than the hash table.

```
Hello World 1
Hello World 2
Hello World 3
```

Because of the implementation of the runtime system, the output is in an orderly manner. This is not a promise for all implementations. Before I added the optimization to the loop, the results were unordered. The order is a happy side-effect of the optimization.

Locations are implemented as integers. The symbol is also an integer that is bitwise-or'd with the `X`. This provides a unique integer that can be used as a key in the hash table. This did cause some problems where the classname had to be known though. The solution was to assign a range of integers to each classname. The first eight bits were reserved for `X`. After that, the remaining 24 bits were divided between however many classes there were. Then, to find what class a symbol represents, a bitwise-and is performed.

There are several problems with this implementation:

1. It places a limit on the number of classes. Twenty four classes is the maximum. This has not been a problem in practice.

2. There is a limit on the number of symbols per a class. If there are 3 classes, then each class has 256 symbols available. If there are 24 classes, then there is only 1 symbol for each class.

3. It does not fit the Icon culture to limit `X` to 255. It is also an additional limitation that does not exist in MDC-90.

The limitations could be eased by increasing the number of bits from 32 to a higher number. But this is an imperfect solution. I would prefer to find an algorithm that provides a correct solution.

4 Conclusion

As a way to play with messaging algorithms, I consider Crow a success. I am not sure if it would be worth the effort to port it so that it is truly concurrent. Icon has a lot of overhead and the main reason for concurrency is speed. It might be useful in a situation where speed is not the goal of concurrency (ie. neural nets, quantum computing).

On a related note is how useful MDC is. I found MDC algorithms to be very dense and hard to understand. It took 2 hours to understand a fibanocci algorithm. I am not sure if the effort is worth the result. On the other hand, it could be that there is a paradigm shift that needs to be dealt with. I have heard of successful, moderate size projects, such as compilers, being completed in MDC-90.

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. *AMDC, Active-Message-Driven Computing.* Illinois Institute of Technology, December 1996. `http://www.cs.iit.edu/~amdc/index.html`.

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

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

[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] Clinton L. Jeffery. *Programming in Idol: Version 10*, June 1998. `http://ringer.cs.utsa.edu/research/idol/idol.html`.

[9] Mark Lutz. *Programming Python.* O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472, first edition, October 1996.