The Crow Progamming Language

Taybin Rutkin

March 30, 2000

1 Introduction

Crow is a combination of the Icon programming language and MDC–90, a distributed C language which uses a messaging paradigm. It takes the extensions to C from MDC–90, and adds them to the very high level language Icon. It basically gives Icon concurrency features using the MDC (Message Driven Computing) paradigm.

A good way to think of the MDC paradigm is as a distributed object orientated system[1]. Instead of directly calling `MyClass::MyFunction()`, messages are sent and a behavior is launched when a certain pattern of messages exist. When there are multiple messages of the same type, they are stored in queues and retrieved on a first–come, first–served basis. Any messages that fit a pattern and launch a behavior are then removed from the queue and given to the behavior to deal with.

The use of messaging gives algorithms better parallelism because multiple behaviors can be executed at the same time. The trade–off is that the algorithms become more complicated. MDC also allows the granularity of an algorithm to be adjusted [2]. Messages contain data that is used by the behaviors. It is simple to change the amount of data sent in a message. Sending less data gives the algorithm a finer granularity.

Icon brings to MDC a very powerful generator engine, along with several other features including, string–parsing, very high level data types such as lists and tables, pattern–matching, expression back–tracking, and garbage collection. These features allow very compact programs to be written and make the algorithm more clear. Not as much code is spent on infrastructure tasks.

[1] This is not the only metaphor for MDC. One advantage of MDC is that other concurrent paradigms are easily encoded in it as well.

Example 2.1 Legal symbol use with Icon short–cuts

```
1     s1 := symbol("COMPUTE")
2     s2 := symbol("COMPUTE")
3     send(data, <s1>)
4     send(data, <s2>)
5     send(print, <s1>)
6     send(print, <here.S>)
7     send(value, <here.S,here.X>)
8     send(value, <here>)
9     every 1 to 5 & loci |||:= [symbol("MATH")]
10    every loci |||:= [|symbol("MATH")\5]
11    write(s1)
```

2 Additions

Crow starts with Icon as a subset and adds to it. A basic familiarity with Icon is assumed. It is difficult to give an example which does not rely on the other features. Each section does have an example though. So if it shows a feature that has not been explained yet, it will be explained in the next section.

2.1 Symbols

The purpose of symbols is to make each location unique. It can be thought of as analogous to a pointer in C. It points to a certain location. It takes a class name and returns a unique value. This guarantees that each location has a different name and computes in a separate virtual space[2].

The symbol creation function works as a generator. In Icon, certain expressions can return multiple values. The symbol generator is such a one. It has no limit on the number of times it can be called in an expression. In other words, it will never cause an expression to fail. Line 7 in the example 2.1 shows one such use. The first part of the expression makes the expression fail after 5 passes through. In the second part, the symbol function returns a symbol which is then appended[3] to a list. Line 9 has a identical result, but

[2] This is not the semantics that I had wanted to use. I had wanted to use the X field in a location to determine the uniqueness of each location. After working through several algorithms though, I saw that it is essential for most interesting algorithms.

[3] "`|||:=`" This is an Icon shortcut for `listA := listA ||| listB`. `|||` is the operator for adding lists together.

Example 2.2 Legal uses of locations and here keyword

```
1    s1 := symbol("OUTPUT")
2    locus := <s1,1>
3    send(data, <symbol("MATH"), n>)
4    locus2 := <s2>
```

it uses the repeated alternation[4] operator and the limitation operator[5] [3].

Lines 1 through 4, in the example 2.1, show how two locations, that are of the same class, and differ only in symbols, are distinct. The output of line 9 is implementation specific. In MDC–90, symbols are implemented using long integers. If the same method was used in implementing Crow, the output could look like "`1034398903`". There is no guarantee of how symbols are implemented or whether the integers are sequential. Symbols can be reused though as shown on line 5. This way, multiple messages can be sent to the same location.

One handy feature is the `here` variable as shown on lines 5 and 6. This is a record with two fields, S and X. The S field contains the symbol of the current location. The X is the local node. This lets messages be sent to the current location instead of having to create a new one. Every behavior has access to the `here` keyword. Note that using `<here>` is eqivilant to using `<here.S, here.X>`.

2.2 Locations

Locations are virtual areas where computation can occur. They are created at runtime and so can only be declared inside procedures and behaviors. A location has a name which is defined by two fields, <S,X>[6], where S is a symbol and X is an integer. The X is optional. If it is undefined, it is set to 0. The existence of X is handy for certain algorithms, program A.3, for example.

Each location creates and saves its own execution stack[7]. This is to avoid

[4] "|" This operator, when used as a prefix, will allow a generator to produce all its values. Then it reinitializes the generator and allows it to produces all its values again. This goes on until something else causes the expression to fail.

[5] "\" This operator limits the number of computations an expression can create. It works very well with the repeated alternation operator.

[6] The original syntax in MDC–90 used brackets instead of greater–than and less–than signs. I wanted to do this also, but list literals used the syntax already.

[7] This creates some problems with using traditional Icon tools, such as co–expressions. Co–expressions, basicly, are generators that save their state across calls. If they are local

Example 2.3 Proper uses of messages

```
1     message output(word, number)
2     message first()
3     hello := output("Hello World", 2)
4     write(hello.word || hello.number)
5     send(hello, <s>)
6     leave(hello)
```

racing hazards from the same procedure being executed at the same time. This applies especially to generators which save state during backtracking.

In the example 2.2, Line 1 shows a symbol being created for use in Line 2. Line 3 shows that the X value can be a variable instead of a constant. It also shows how a function that returns a symbol works as well. Line 4 shows the syntax of a location without X.

2.3 Messages

Messages are the data types that are used by classes to communicate with each other. Messages are declared:
message name(field1, field2, . . . , fieldn)
An instance of a message is created with a message constructor that corresponds to the message name. Message fields are accessed with the '`.`' operator.

All data types are allowed as fields within messages. This is a change from MDC–90 which only allowed static types. All Icon primitives, such as lists, tuples, and records are allowed. Messages can also be created without any fields. These types of messages are called signals. These are used to control which behaviors will launch.

Messages are sent to locations with the `send()` function. The syntax is:
send(*message*, *location*)
The semantics should be thought of as adding a node to the end of a queue. A message may be sent back to the same location at which the behavior is executing with the `leave()` function. The major difference between leaving a message and sending it to oneself is that the `leave()` function is guaranteed to be effective immediately. The message will be present before the next behavior is triggered at that location, whereas the `send()` only guarantees

to each location then their functionality is diminished. If they can be globally accessed from any location then there are race hazards.

Example 2.4 Examples of behaviors

```
1    behavior (data d1, d2; precedence 50)
2        write(d1.text)
3    end
4
5    behavior (data d1, d2 ; first = 1 ; precedence 101)
6    ...
7    end
```

the message will be present some time after the behavior terminates[8]

One common use of signals is to have a behavior launch unless a certain message exists. For example, say there is a behavior that should only run once. That behavior, typically, will be the first one launched. It will, during its execution, leave a signal. Then, if that signal exists, the same behavior will not launch.

MDC–90 also has a `delay()` and `undelay()` functions. These would temporarily remove messages from the queues. These have been superseded by message tables. Message tables allow messages to be attached to other messages. Since message tables can be created using Icon's lists, Crow does not have the `delay()` or `undelay()` functions.

2.4 Behaviors

The syntax for behaviors is very similar to that of procedures. They can only be declared within classes. The syntax for their declaration is:
behavior behavior_name(*pattern*)
...
end
The concept of a pattern is the meat and bones of MDC. It is comprised of three parts: 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* messages 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.

[8] These are exactly the same semantics as MDC–90.

Example 2.5 Class definitions

```
1    class OUTPUT
2    behavior (print p)
3    ...
4    end
5    class PRINTER : OUTPUT
```

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 relation operators are `= ~= > < >= <=`.

Precedence The syntax is

precedence *value*

This gives the precedence of the behavior. In the event that more than one pattern matches, the pattern with the highest precedent will be launched. The *number* can be as high as 255.

The component list and the relation can be chained onto each other multiple times. The pattern parts use semi–colons (`;`) as the delimiters. 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.

When a behavior is launched, any messages matched in its pattern are deleted from their respective queues. Only one behavior can be executing at a location at any one time. This is to mimic the semantics of MDC [1].

In the example 2.4, a common technique to have one behavior execute once and then never again is shown. The `first` message in the second behavior on line 5 is a signal. It does not actually contain any data. When the `data` messages are sent for the first time, the executing code also sends a `first` message. The first behavior, on line 1, also matches the pattern of the messages, since all it needs to launch is two `data` messages. This does not happen because the second behavior has a higher precedence.

2.5 Classes

The notation for classes is derived from Clinton Jefferey's syntax for Idol [7]. This way the syntax more closely reflects the OOP style of MDC.

Example 2.6 Proper uses of portals

```
1    message print(text)
2    message filename(filename)
3    locus := portal(print, <here.S>)
4    send(filename, locus)
```

class class_name
...
Behaviors are then defined within the class. The class definition ends when the next class definition begins. As example 2.5 shows, the class definition on line 1 ends on line 5 when the next class definition starts. It also ends at the start of procedure definitions. Classes only contain behaviors. There are not any class variables.

Inheritance is also possible by appending a ':' to the end of the class name and following it with the superclass; like so:
class subclass : superclass
This will give the subclass all the behaviors of the superclass. If a behavior is redefined in the subclass it overrides the behavior of the superclass. Multiple inheritance is also possible by appending multiple superclasses delineated with colons.

Along with the basic syntax, Crow also borrows the semantics of Idol [6]. Instead of taking the superclass and adding the features of subclasses to it, as in most OOP languages, Crow takes the subclass and adds the features of the superclasses to it. This difference is crucial. It solves the problems of multiple inheritance and even allows for circular class trees[9].

Starting with the subclass, Crow looks up the class tree to the first superclass, when it finds a behavior that does not exist in the subclass it adds it to the class. If the behavior has already been defined, then it is not added. This way, no behavior has multiple definitions. If two superclasses define the same behavior differently, then the one that is encountered first is used. Since Crow tranverses the classes in a simple left to right manner, it should be simple to change which class' behaviors are dominant.

2.6 Portals

Portals are a combination of a location and a message type. It can be

[9] I am not sure how useful these would be, but I think that it indicates the correctness of the solution.

thought of as a labeled doorway into a location [4]. A message can be sent to a portal the same way it would be sent to a location. A message that is sent through a portal is converted to the message type of the portal. This requires that that the message types be structurally similar. So a message with two fields could not be sent through a portal with only one field.

One way to think of its semantics is to imagine the portal directing a message to a certain location, and then to a certain message queue, one which it would not have normally gone to. They are created with the following syntax: portal(msgType, location)[10]. This returns a value with a type of PORTAL. The returned value can then be assigned to a variable or can be used in place of a location.

In example 2.6, line 3 shows one important use of portals. They are often used to direct values back to the starting location after they have been computed. The keyword `here` is essential for this use. The Fibonacci program A.4 shows the use of portals.

A Usage

These are some examples and explanations of the techniques used. They grow in complexity. The algorithms are more of a demonstration of messaging techniques than generator and back–tracking techniques. This is because messaging seems to have a greater "paradigm shift" effect.

The algorithms for normally simple tasks are fairly complex. This is because they have been rewritten to use messaging. The more behaviors an algorithm is broken into, the finer the grain size. A smaller grain size lets more parallelism take place. There is a trade–off, as usual. If the grain size is too small, there will be too much overhead to be faster with a larger grain size.

There are a couple of areas of conflict between Icon and MDC–90. For instance, Icon lets the programmer declare certain variables as global. It is very possible to create race hazards if they are used. At the most, Crow would not let more than one behavior write to a variable at the same time through the use of a simple semaphore system. The best advice is to avoid the use of global variables.

[10] I chose this procedure–like syntax for a couple reasons. It is used by several other Icon types such as sets and tables. These types were added after the language was completed. It also lets the type be implemented as a procedure if need be. And finally, the less additions to the grammar, the better.

Program A.1 Hello World example

```
1     message print(text)
2
3     class OUTPUT
4     behavior(print p1)
5         write(p1.text, here.X)
6     end
7
8     procedure main()
9         bottle := print("Hello World ")
10        every i:= 1 to 3 do {
11            s := symbol("OUTPUT")
12            send(bottle, <s, i>)
13        }
14    end
```

A.1 Hello World

This is to show the syntax and semantics in use. The next example will be more in–depth. In program A.1, line 1 creates a message of type `print`. It contains one field, the `text` field. Line 3 begins a class definition of class `OUTPUT`. This class contains one behavior which is launched when there is one `print` message. The behavior then prints out the text field of the message and appends the location number.

Line 8 starts the procedure `main`, which is where execution starts. The procedure then creates a variable of type `print` named `bottle`[11]. Line 10 contains the equivalent of a C–style `for` loop. Line 11 sends the `bottle` message to the `OUTPUT` class at location `i`. Notice that line 12 could be written: `send(bottle,<symbol(OUTPUT),i>)`

The output for this should be:

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

A.2 Factorial

This one is more complicated. The logic is difficult to follow mainly because it is a paradigm shift. While it could be done in a simpler way,

[11] Message in a bottle, get it?

Program A.2 Part 1 of factorial program

```
1     $define MAX 3
2     message fac(dst, i)
3     message fac_contin(dst, i)
4     message print(i)
5     message result(rslt)
6
7     class FAC
8     behavior (fac f)
9         if f.i <= 1 then {
10            r := result(1)
11            send(r, f.dst)
12        } else {
13            s := symbol("FAC")
14            f1 := fac(here.S, f.i-1)
15            send(f1, <s>)
16            leave(fac_contin(f.dst, f.i))
17        }
18    end
19
20    behavior (fac_contin f; result r)
21        r2 := result(r.rslt * f.i)
22        send(r2,f.dst)
23    end
```

Program A.3 Part 2 of factorial program

```
24
25      class IO
26      behavior(print p; result r)
27          write("Factorial ", p.i, " = ", r.rslt)
28      end
29
30      procedure main()
31          every i := 1 to MAX do {
32              s := symbol("FAC")
33              t := symbol("IO")
34              p := print(i)
35              send(p, <t>)
36              f := fac(<t>, i)
37              send(f, <s>)
38          }
39      end
```

by breaking the algorithm into multiple behaviors, a higher grain–density is reached. This allows for better parallelism.

The output for this program is:

```
Factorial 1 = 1
Factorial 2 = 2
Factorial 3 = 6
```

One reason the output is in order is that factorials take longer to compute as n grows larger. So the algorithms for the lower n's finished first. This is different than the countdown example from A.2, where all the behaviors would finish fairly close to each other.

The loop on line 31 splits the program up so that each number is computed on its own node. If the algorithm works for one size number, then it works for all size numbers[12]. We'll look at n=2 because it is the most is the simplest without being too simple.

The behavior on line 8 executes because line 37 sent a `fac` message to a `FAC` location. Since `f.i` is equal to 2, the `if` control skips to line 12. Here is the tricky part. The code creates another `fac` message, but instead of its destination field pointing to an `IO` class, it points to a `FAC` class. Then the original `fac` message is converted to a `fac_contin` message on line 16 and

[12] An inductive leap of faith is happening here.

left at the FAC location.

The new fac message created on line 14 is then picked up again, but this time i is equal to 1. So a result message is created and sent to f.dst. This allows the second behavior to launch since there now exists a fac_contin and a result message. Line 21 multiples r.rslt (which is equal to 1) and f.i (which is equal to 2).

If we looked at n = 3 or higher, we would see that the creation of a new symbol for each new location is necessary. It allows a type of recursion to happen that otherwise would not be possible.

A.3 Fibonacci Sequence

This program is much more complex and, better yet, demonstrates an important technique using portals. This is another recursive, inductive algorithm; similar to the factorial program. We will look at the execution sequence for i being equal to 1 and 2.

On line 51, a portal is created and stored in the fib message on line 52. This makes the default value of fib.dst a print message to be sent to the IO class on the same node. This does change though, on lines 20 and 23.

FibClass's first behavior is always the first to execute. Since i = 1, here.X = 1. This causes the first branch to be taken on line 14. A result message with a value of 1 is added to the local queue. Since line 12 left the first fib message, the second FibClass behavior is launched.

This behavior then sends the value of result r, which is 1, to f.dst. This was set to be a portal to the IO class back on line 51. It is also converted to be a print message. This launches the behavior on line 43, which provides the output. Notice that the behavior on line 30 also leaves the result as well as sending it.

Now we see what happens when i = 2. First, the original fib message, with the portal pointing to the IO class is left, so it is still in existence later on. Since here.X = 2, the execution is diverted to line 17. Two new fib messages are created on lines 19 and 22. These are different from the first one, in that their portals point to the ARITH class instead of IO. The message types are also of lopnd and ropnd, respectively.

Since the nodes below 2 are 1 and 0, they follow the same steps as i = 1. Except that when they reach the second behavior on line 30, instead of going to the IO class, they are sent to the ARITH class. On line 26, an add message is sent to the ARITH class as well, containing a portal to the current location[13]. This launches the behavior for the ARITH class.

[13] A portal is probably not necessary for this part of the algorithm. This portal converts

Program A.4 Part 1 of Fibonacci program

```
1     message fib(dst)
2     message add(dst)
3     message lopnd(value)
4     message ropnd(value)
5     message await_result()
6     message print(val)
7     message result(rslt)
8
9     class FibClass
10    behavior(fib f; await_result = 0)
11        leave(await_result())
12        leave(f)
13
14        if here.X <= 1 then {
15            r := result(1)
16            leave(r)
17        } else {
18            s := symbol("ARITH")
19            f1 := fib(portal(lopnd,<s>))
20            send(f1, <here.S, here.X-1>)
21
22            f2 := fib(portal(ropnd,<s>))
23            send(f2,<here.S,here.X-2>)
24
25            a := add(portal(result,<here.S>))
26            send(a,<s>)
27        }
28    end
```

Program A.5 Part 2 of Fibonacci program

```
29
30      behavior(fib f; result r; precedence 100)
31          send(r, f.dst)
32          leave(r)
33      end
34
35      class ARITH
36      behavior(add a; lopnd left; ropnd right)
37          r := result(left.value + right.value)
38          send(r, a.dst)
39      end
40
41      class IO
42      behavior(print p)
43          write("Fib "|| here.X || " = " || p.val)
44      end
45
46      procedure main()
47          s := symbol("FibClass")
48          t := symbol("IO")
49
50          every i:=1 to 40 do {
51              p := portal(print, <t, i>)
52              f := fib(p)
53              send(f, <s,i>)
54          }
55      end
```

This new behavior adds the two values of `lopnd` and `ropnd`. These values are both 1 so the result is 2. The new value is placed in a result which is sent to the destination in `add.dst`. This destination is, of course, the one set on line 25. So the result is sent back to the original node, number 2.

The old `fib` message that points to `IO` is still in the queue. This message and the new `result` message launch the behavior on line 30. Since `f.dst` points to `IO`, the result is printed. The result is also left behind for other iterations of the algorithm.

B Grammar

This grammar is the additions to the Icon grammar [5]. Since the entire Icon grammar is not included below, some variables are not included below. This is why *expression–list* is not defined.

messages to a `result` message. It might be easier to just send `result` messages instead of converting other messages to them.

location:
    < *identifier* , *expression–list* >

message:
    message *identifier* (*field–list*$_{opt}$)

class:
    class *identifier superclass*$_{opt}$ *behavior–list*

superclass
    : *identifier*
    : *identifier superclass*

behavior–list
    behavior
    behavior behavior–list

behavior:
    behavior–header expression–sequence$_{opt}$ **end**

behavior–header:
    behavior (*pattern*)

pattern:
    msg–id idlist ; *pattern–tail*

pattern–tail:
    msg–id idlist ; *pattern–tail*
    msg–id relop integer ; *pattern–tail*
    precedence *integer*
    empty

idlist:
    msg–id idlist2

idlist2:
    , *msg–id idlist2*
    empty

relop:

```
=
~=
>
<
>=
<=
```

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 and Madge T. Griswold. *The Icon Programming Language*. Peer-To-Peer Communications, 3rd edition, 1997.

[6] Clinton L. Jeffery. Closure-based inheritance and inheritance cycles in Idol. Technical report, The University of Texas at San Antonio, Division of Computer Science, San Antonio, TX 78249, USA, July 1998. `http://www.cs.utsa.edu/research/idol/closure.ps`.

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