Experiences with an Icon-like Expression Evaluation System

Laurence Tratt
Middlesex University, The Burroughs, Hendon, London, NW4 4BT, United Kingdom
laurie@tratt.net

Abstract

The design of the Icon programming language's expression evaluation system, which can perform limited backtracking, was unique amongst imperative programming languages when created. In this paper I explain and critique the original Icon design and show how a similar system can be integrated into a modern dynamically typed language. Finally I detail my experiences of this system and offer suggestions for the lessons to be learned from it.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords Language design, Icon, Converge

1. Introduction

Icon [3] is a programming language designed chiefly by Ralph Griswold, intended to be usable as a general purpose language, but particularly well suited to text processing. Icon's immediate predecessor is the little-known SL5, considered a dead-end by its designer and quickly abandoned [2]; arguably its true predecessor language is the better known SNOBOL4 [5] (indeed, Icon was known as SNOBOL5 during its early development). SNOBOL4 was specifically aimed at text processing; in modern terms, it would be considered a Domain Specific Language (DSL) as it is not well suited to general purpose computing. Icon can be seen as the integration of SNOBOL4-esque text processing capabilities into a dynamically typed general purpose programming language.

Icon's expression evaluation system utilises what it calls *goal-directed evaluation* and was, until recently, the only imperative programming language capable of backtracking. While Icon's goal-directed evaluation strategy is not as powerful or extensive as the backtracking used in some declarative languages (e.g. Prolog), it can express surprisingly complex relationships. In order to allow backtracking, the design of Icon's expression evaluation system is fundamentally different than other imperative languages, yet it manages to ensure that 'vanilla' expression evaluation has the same observable effect as in most other languages (though the means used to achieve this effect are rarely the same). The underpinnings of Icon's expression evaluation system challenge a number of standard assumptions.

To modern eyes, Icon has an old-fashioned feel, reflecting its 1970's heritage. Syntactically it can be described as a dynamically typed Algol variant; semantically it has a number of 'unfortunate' features such as differentiating between values and references that similar languages have long since discarded. This should not be taken as a criticism of Icon – it was ahead of its time in many ways – but is merely an observation that modern programmers expect something slightly different from their programming languages. Today, Icon is, fairly or not, a largely forgotten language. However small elements of its influence live on in other languages: for example, Python's generators were inspired by (although not as powerful as) Icon's equivalent concept [8].

When I was designing the Converge language [11] I integrated an expression evaluation system whose design was heavily inspired by Icon. As far as I am aware, Converge is the first – and, to date, the only – language that is not a direct clone of Icon to integrate such a system. This paper[1] has two aims. First it implicitly shows that the design of Icon's expression evaluation system can be extracted, reused, and altered as needed in other programming languages. Second it aims to capture the changes made to the original Converge design, lessons learnt, and subsequent ideas for the future.

This paper is structured as followed. First I present an overview of Icon and its expression evaluation system. I then describe several unfortunate aspects of this system, followed by an explanation of how Converge tries to address these aspects. I then briefly describe performance implications, and suggest possible optimisations. Finally I present what, from my experience, I consider to be the advantages and disadvantages of such an expression evaluation system.

DLS 2010, October 18, 2010, Reno/Tahoe, Nevada, USA.
Copyright © 2010 ACM 978-1-4503-0405-4/10/10. . . $10.00

[1] A much earlier version of this paper appeared as a blog post on my website.

2. Icon

This section aims to give a basic introduction to the salient features of Icon. It is not a replacement for the full manual [3], but should allow those familiar with mainstream programming languages to gain an understanding of Icon sufficient for the rest of this paper.

2.1 Basics

Icon is a procedural, dynamically typed language using an Algol-esque keyword-based syntax. One syntactic oddity relative to the rest of the language is that groups of expressions – such as the clauses of an `if` construct or body of a `while` loop – are enclosed in curly brackets. Icon does not distinguish between statements (which, in many languages, do not produce values) and expressions (which do produce values): everything in Icon is an expression (though, as we shall see, Icon expressions do not always produce values).

A complete Icon program which counts and then prints the number of lines in a file (the equivalent of the `wc -l` Unix command) is as follows:

```
procedure main(argv)
  f := open(argv[1], "rt")
  i := 0
  while read(f) do {
    i := i + 1
  }
  write(i)
end
```

There is only one small hint in this simple program of something unusual. The `read` function in Icon returns the next line in a given file; when all lines in the file have been read, `read` causes the `while` loops condition to fail and execution to pass to the `write` function. The mechanism by which this happens is explored in the following sub-section.

2.2 Success and failure

In traditional imperative programming languages, expressions produce values. In Icon, however, expressions *succeed* or *fail*; expressions which succeed produce values, while those that fail do not. If an expression fails, it transmits failure to its enclosing expression (which may then fail recursively). This concept is at the heart of Icon's expression evaluation system. While the concept of failure is often confused with exceptions, they are two very different things—exceptions indicate undesired behaviour (such as being unable to open a file), while failure indicates that an expression can produce no more values (e.g. the `read` function from Section 2.1). The two concepts are orthoganal and both can be present in a language, though Icon itself does not have exceptions.

A simple example of an expression which, in different circumstances, can succeed or fail is `x < y`. If `x` is 2 and `y` is 3, then the expression succeeds and the value `3` is produced (in Icon, relational operators produce their right hand value if the operator succeeds); if the values of `x` and `y` are reversed, then the expression fails and no value is produced. As this might suggest, Icon does not have standard boolean logic; indeed there is neither a boolean data type, nor conventional boolean operators. Despite this, in 'normal' use, standard constructs have the same observable effect as in most mainstream programming languages so that code such as the following executes as per standard expectations:

```
if x < y then {
  write(x)
}
```

Nested expressions are valid in Icon, and failure is cascaded recursively (see Section 2.5 for details about the the limits of this cascading). Therefore the expression `z := x < y` only assigns the value of `y` to `z` if the expression `x < y` succeeds; if `x < y` fails, no assignment is made to `z` and it retains its previous value.

2.3 Generators

In Icon, procedures can generate zero or more values. Conventionally those which always generate exactly one value are called simply procedures; those which can generate a variable number of values are called *generators*. The `suspend` keyword is similar to normal `return`, but as well as transmitting a value to the functions caller, it saves the generator's call stack; the generator can later be resumed, with the generator's call stack put back in place, and execution continuing from directly after the `suspend` call, allowing a generator to produce multiple values. If the syntactic end of the generator or `return &fail` is encountered then the generator has finished generating values; failure is then transmitted to its caller.

The following complete program with a generator `ito` prints 1 to 9 inclusive:

```
procedure ito(x)
  i := 0
  while i < x do {
    suspend i
    i := i + 1
  }
end

procedure main()
  every x := ito(10) do { write(x) }
end
```

The `every` construct can be considered to be broadly equivalent to a typical `for` statement, and is the standard way in Icon of *pumping* a generator for all of its values. The above code works by first calling `ito`, which suspends itself with the value `0`; this is then assigned to `x` and the body of the `every` construct executed. Once the body has been executed, `every` then resumes the suspended `ito` generator and repeats the loop; when the generator eventually fails, the `every` construct itself fails. Note that `while` and `every` are very different constructs; `every` evaluates its expression once, and subsequently pumps it for values if it is a generator, whereas `while` evaluates its expression anew on each iteration.

Two other types of generator are noteworthy. The generator `i to j` generates all values from `i` to `j` inclusive (similar to the `ito` example used earlier). More surprisingly, a special type of generator subsumes the boolean notion of 'or'. Icon's *alternation* operator '`|`' is a non-procedure generator which successively – and therefore lazily – generates each of its values. Goal-directed evaluation (see Section 2.4) means that constructs such as `if x | y` work as per typical expectations; however one can also explicitly make use of alternation in expressions such as `every write(1 | 2)` which prints `1` then `2`.

2.4 Goal-directed evaluation

As stated earlier, Icon's goal-directed evaluation strategy allows a limited form of backtracking. This can most easily be seen with the *conjunction* operator '`&`' which joins together two or more sub-expressions and succeeds only if each sub-expression succeeds. If a sub-expression fails, and one of the preceding expressions is a generator, then control backtracks to the generator which is resumed to generate another value; the conjunction then continues execution from after the generator. A simple modification of the program from Section 2.3 prints out the even numbers between 0 and 9 inclusive:

```
procedure main()
  every x := ito(10) & x % 2 == 0 do {
    write(x)
  }
end
```

This works as follows. First the `ito` function is called, which suspends with the value `0` which is then assigned to `x`; since `0 % 2 == 0` succeeds, the conjunction succeeds and the body of the `every` construct is executed. The `every` construct then pumps `ito` for its next value `1` which is assigned to `x`; however since `1 % 2 == 0` fails, the conjunction then causes immediate backtracking and pumps `ito` for its next value, without executing the `every` constructs body.

Icon has several other features related to goal-directed evaluation. If backtracking occurs over a reversible assignment (which uses the syntax `x <- e` to allow Icon to differentiate this from normal un-reversible assignment), then the variable `x` reverts to the value it had before the assignment. Limited generation `e \ i` allows a generator to be pumped a maximum of `i` times.

2.5 Bounded expressions

In languages such as Prolog, backtracking can occur to an arbitrary depth (unless restricted with the 'cut' predicate). Clearly, backtracking in an imperative language can not be as far-reaching either practically (in general an unbounded quantity of memory would be needed to allow backtracking, causing unexpected performance issues) or philosophically (it would subvert widely held expectations about basic imperative programming constructs). Icon's solution to this is the concept of *bounded expressions*; backtracking can only occur within a bounded expression. What this means is that backtracking in Icon is inherently local in nature and is constrained to small units. Icon's bounded expressions are deliberately chosen to preserve typical expectations; for example the conditional of the `if` construct is a bounded expression as are top-level expressions separated by new-lines.

3. Icon's expression evaluation strategy critiqued

In this section I list various issues I have encountered with the design of Icon's expression evaluation strategy. Note that the intention of this section is not to critique Icon in general. Icon reflects the era in which it was designed, and features such as variables with default values and a distinction between variables and references are often considered to be flaws by modern programmers—however such issues are not pertinent to the core of this paper and are thus not considered.

3.1 Procedures fail by default

In Icon, the default 'return value' of a procedure is failure; in other words, there is an implicit `return &fail` at the end of each procedure. This is very useful for generators such as `ito`, whose last action is to signal that they have no more values to generate. However it causes seemingly innocuous non-generator procedures such as the following to behave in ways that are very hard to debug:

```
procedure f(x)
  if x > 0 then {
    return 1
  }
end

procedure main()
  write(f(-1))
end
```

When this program is run, nothing is printed to screen because the failure of `f`'s condition means that the procedure executes its default return action, which is to fail, meaning that `write` is never called. While this particular example is easily debugged, variations on this problem – such as when an assignment `x := f(..)` fails – can cause havoc in large programs.

After falling victim to this problem several times, I briefly analysed several programs. This quickly showed that generators are, by some distance, the least common form of procedure—most procedures always return a single value (and only a single value). In other words, Icon's default behaviour is useful for a small subset of procedures (generators), but dangerous for the majority.

3.2 Lack of a boolean datatype

In general, Icon's lack of a boolean datatype is unproblematic: normal conditional evaluation produces results which match convention (even though the underlying evaluation mechanism is different). Indeed, various other languages (such as K&R C) also lack a boolean datatype, generally relying on the convention that conditionals evaluate to false if an integer value is 0 (used conventionally for 'false') and true for other values (1 is used conventionally for 'true').

However, since any expression which produces a value in Icon succeeds, there is no equivalent way of expressing this notion in Icon. Thus the following code prints `taken` to screen:

```
procedure main()
  c := 0
  if c then {
    write("taken")
  }
end
```

Indeed, because variables have a default value in Icon, even removing the assignment `c := 0` does not alter the behaviour of the above program!

To work around this, individual code needs to use a convention about what a 'true' and a 'false' value are in any given circumstances. Typically one uses `1` and `0` respectively, but this often makes statements such as `if x == 0` seem ambiguous: is this a boolean check that happens to use integers, or simply a normal integer comparison?

3.3 Generators are easily hidden

Generators are an integral part of Icon, and a common idiom is to pair a generator with a normal procedure in an `every` construct to perform a repeated action. For example the following code prints the index of every digit (0-9) in the string `s` using the generator `upto` and the normal procedure `write`:

```
every write(upto('0123456789', s))
```

In general it is not clear from simply looking at the code if either, both, or neither, of the two functions `write` and `upto` is a generator. This can cause confusion when debugging as without this knowledge, there is no way of determining the expected behaviour of the code, discouraging the use of such generator related idioms.

3.4 Backtracking can be unwieldy and difficult to use outside string scanning

One of the major features of goal-directed evaluation is that it allows backtracking. The most common way for this to be achieved is by linking expressions together with the conjunction operator `&`. Combining conjunction with `every` allows compact expressions such as the following to be expressed (where `upto(c, x)` generates each index position where the character `c` appears in the string `s`):

```
x := "cbaabaacvcabcbab"
every write(i := upto('a', x) & i % 2 == 0 & i)
```

This prints out every index position in `x` which is a multiple of two, and which contains the character `a` (4 and 6 in this case). Although this small example is relatively readable, as such code increases in size, it rapidly becomes unreadable. To partly solve this, Icon introduces the concept of string scanning `e1 ? e2` where, in essence, the string `e1` evaluates to is made the global search for the code in `e2`. Procedures such as `upto` follow a protocol where the 'global' search string is used if a specific search string is not specified. The above example can be rewritten using string scanning as follows:

```
every write(x ? { upto('a') & i % 2 == 0 & i })
```

Icon has several string matching procedures which follow the required protocol. While string scanning can slightly reduce the complexity of backtracking, it is far from a complete solution. Furthermore it requires the use of two special pseudo-global variables (`&subject` which records the string being scanned and `&pos` which records the current index of the search in the string) and a suite of functions which operate on these two special variables.

A far more fundamental problem is that, in my experience, the form of backtracking used is too weak to be useful for complex operations on anything other than strings. Icon allows variables in conjunctions to have assignments undone during backtracking, but does not reverse other similar operations such as assignments within lists and so on. Note that I am not arguing that Icon could, or should, do this (see Section 2.5): simply that its absence precludes many scenarios). This means that more complex uses of backtracking (see [12] for an example of highly complex backtracking in an Icon-like system) still require most of the complex backtracking to be manually encoded much as in other imperative languages.

In summary, Icon's backtracking can be useful, but only for certain use cases (mostly related to string scanning).

4. An Icon-like language: Converge

When I was designing the Converge language I integrated an expression evaluation system heavily inspired by Icon. For the purposes of this paper, Converge can be thought of as a cross between Python (in terms of general aesthetics, including its syntax) and Icon (its expression evaluation system). The driving aim behind Converge was to add a macro-esque system to a dynamically typed language; this system is then used to implement syntactically distinct DSLs. As many of the original DSLs implemented in Converge used complex matching over tree and graph structures, it was natural to consider the integration of Icon's expression evaluation system which had sufficiently similar aims and was without precedent in other imperative languages[2]. As far as I am aware, Converge is the only non-Icon clone to integrate such a system. Though it may seem obvious, it is worth explicitly noting that Converge's existence shows that the ideas underlying Icon's expression evaluation system are applicable to other languages.

Most of Converge's aspects – other than its expression evaluation system – are documented elsewhere (see e.g. [10, 13]) and I do not cover most of those here, concentrating on those aspects relevant to Converge's Icon inheritance. Historically speaking, early versions of Converge in-

[2] I became aware of Icon through my (mostly inconsequential) involvement with Python development, specifically many mailing list messages from the sage-like Tim Peters.

herited Icon's expression evaluation system wholesale; over time various changes have been introduced. This section describes and motivates those changes. All example code henceforth is written in Converge and not in Icon.

4.1 Procedures don't fail by default

I considered two solutions to Icon's decision to make procedures return fail by default (see Section 3.1). First, one could syntactically differentiate generators and procedures, with the former failing by default and the latter not. Second, one can make all procedures not fail by default. Given the burden that comes with new syntax, and since generators aren't common, the decision was an easy one. Converge functions are similar to those in many other languages, with their default return action being to return the `null` object. This solves the common problem noted in Section 3.1 at the expense of a less common problem: generators which do not explicitly fail at their end return `null` as their final object. When the lack of an explicit fail is an oversight, it tends to be relatively easy to debug in comparison to the original problem, since it usually results in performing an invalid operation on the `null` object.

4.2 Introducing a boolean datatype by the backdoor

As described in Section 3.2, Icon does not have a boolean datatype; consequently much mundane programming becomes even more of a chore than normal. Simply adding a boolean datatype into an Icon-like language raises several thorny questions about when 'false' should cause failure: the two concepts are clearly related in some cases, but not in others. For example, while one would clearly expect `2 < 1` to cause the `else` branch of an `if` statement to be executed, what should happen to a statement like `x := 2 < 1`? Should `x` be assigned the value 'false' (as in most languages), or the assignment simply not executed (as in Icon)? It seems hard to integrate in a typical boolean datatype and still maintain Icon-like behaviour in such corner cases.

I therefore attempted to find an Icon-esque solution to this problem based on the idea that one only needed a value which caused conditional statements to fail. This would allow one to write idiomatic code such as `if b` and `if not b`. I therefore introduced a `fail` value into the Converge runtime, analogous to `null` (`fail` was in essence a keyword pointing to a singleton object). If statements such as `if` or `return` statement evaluated their child expression and received the `fail` value, then the parent statement would fail.

For most simple uses of booleans, this introduction of a boolean-esque datatype by the backdoor worked very well. However, it introduced some unpleasant corner cases. Consider the following piece of Converge code:

```
l := [1, fail, 2]
Sys::println(l[1])
```

This printed nothing, as `l[1]` is a synonym for `l.get(1)`, which tried to evaluate `return fail`, which then caused `get` to fail. While this might seem a contrived example, a slight variation on it cropped up in real code. Converge modules can enumerate their definitions and associated values; every module contains standard definitions for lexical scoping such as `null`. One such definition was named `fail` which unsurprisingly pointed to the `fail` object; trying to find out the definitions value caused the same problem as `l.get(1)` above, and an innordinate amount of time to debug.

This problem lead to the realisation that the `fail` object is fundamentally dangerous, and that attempts to mitigate this were only likely to move the danger around, without ever actually removing it. A convention (not enforced by the compiler) was developed that the only safe idiom involving `fail` was `return fail`, and this was followed for around 2 years; eventually it was banished from Converge entirely. In current versions of Converge, `fail` is a statement causing a generator to fail. While removing user access to the `fail` object (though it lives on internally in the Converge VM) does require users to resort to the same boolean encoding tactics as in Icon, the result is far less conceptually troubling.

4.3 A convention for generator names

As shown in Section 3.3, it is impossible to statically determine in Icon whether a call to a given function `f` is to a generator or a normal function; this makes code hard to read, particularly in the face of OO polymorphism (as found in Converge). I considered two solutions to this problem. First one could introduce new syntax for generator calls. Second one could introduce a convention for generator function names. Preferring when possible not to add new syntax to Converge, I took the second option. In general, generator names in Converge are prefixed by `iter_` which, whilst unobtrusive, highlights that the function in question is a generator. This gives a surprisingly large boost to code readability.

4.4 Simplification of backtracking features

Converge acknowledges the weakness of Icon's advanced backtracking features (such as reversible assignments) and simply does not include them (though note that the simple backtracking used in Section 3.4 works identically in Icon and Converge). This significantly simplifies the language without losing much practical functionality. Reversible assignments, for example, are used extremely infrequently—even in Icon's core library! A possible solution to the problem would seem to be to increase the power of Icon's backtracking abilities, to better match that of e.g. Prolog. I believe this is impractical: mutable state prevents data-sharing, meaning that huge – and, in general, unbounded – quantities of memory would rapidly be consumed to allow backtracking. As well as quickly exhausting memory, this would also lead to large performance penalties as data was copied and garbage collected on a far larger scale than would be the case with immutable state. More subtly, this would undermine one of the implicit guarantees of that imperative programming languages give—predictable performance. In summary, users would be unlikely to use such advanced backtracking features; they would introduce much complexity into the language and implementation with little likely gain; and, I suggest, they would stray from Icon's design goals.

Most notably, Converge has no equivalent of string scanning, for two reasons. First, modern regular expression libraries subsume most simple cases of string scanning, and formal parsing tools (i.e. parsing using context free grammars) most complex cases. Second, string scanning raises issues about global or semi-global variables and privileged access to them—while there are occasional reasons for using global variables, making them a commonplace seems dangerous at best.

5. Implementation

Both Icon and Converge are implemented as stack-based Virtual Machines (VMs). In both systems, a compiler transforms source code into bytecode which is then executed by a VM. While much of each VM is relatively standard, Icon's expression evaluation system requires an unusual approach. Converge very closely follows Icon's VM in this regard, so most of this section is framed in terms of Converge's implementation (which is smaller and more amenable to experimentation); however the general principles are trivially adaptable to Icon.

The main reason why an Icon-like VM must diverge from standard VM principles is due to failure. This has a number of implications, in particular a heavy reliance on stack operations to cope properly with failure. For example in Converge the following code fragment:

```
x := 1 < 2
rest
```

is translated to the following:

```
ADD_FAILURE_FRAME rest // If failure occurs below,
                // jump to rest
INT 1           // Pushes 1 onto the stack
INT 2           // Pushes 2 onto the stack
LT              // Pops two objects and performs a
                // 'less than' comparison pushing an
                // object to the stack if successful
                // or failing otherwise
ASSIGN_VAR 0 1 // Pops an object from the stack and
                // assigns to a var
REMOVE_FAILURE_FRAME
...             // rest
```

Converge stacks only have a handful of entry types, chiefly: continuation frames (for the purposes of this paper this can be thought of as a 'function call frame'), failure frames (recording what should happen in the case of failure during expression evaluation), generator frames (recording the status of suspended generators), and object references. At any point the stack has to record pointers to the current continuation frame, failure frame, and generator frame (the latter two of which must always be later in the stack than the continuation frame). The `ADD_FAILURE_FRAME` opcode pushes a failure frame onto the stack which records the current failure and generator frame pointers and the 'fail to' point, updates the stacks failure frame to point to the new failure frame and unsets the generator frame pointer. `REMOVE_FAILURE_FRAME` pops the failure frame from the stack and restores the previous failure frame and generator frame pointers. If, after the `ADD_FAILURE_FRAME` opcode, but before the the `REMOVE_FAILURE_FRAME` opcode, failure occurs then the failure frame is summarily popped from the stack (and the previous generator and failure frame pointers restored), and execution jumps to the position *rest*.

The translation of other constructs, in particular the Icon `every` construct, relies on large numbers of similar stack operations. In the interests of brevity, this paper does not look in detail at the translation of each language construct; see [4] for a detailed description.

5.1 Optimisation suggestions

The `ADD_FAILURE_FRAME` and `REMOVE_FAILURE_FRAME` opcodes give an important insight into an inefficiency shared by existing Icon and Icon-like implementations but not by other languages. Since every logical line of code is a bounded expression, and because within logical lines there can be extra bounded expressions, the number of `ADD_FAILURE_FRAME` and `REMOVE_FAILURE_FRAME` operations executed is large: for a typical Converge program execution, they represent around 25-30% of executed opcodes. While relatively cheap opcodes to execute, any execution which touches the stack incurs costs, and their sheer frequency means that a frustrating amount of execution time is devoted to them. Although it is difficult to obtain a precise figure for the cumulative time stack operations specific to the Icon-esque evaluation system occupy in the Converge VM as a whole (since some are intertwined within other code), crude estimates show it is a minimum of 10% of total execution time. Note that in the Converge VM, such stack operations are one of the few reasonably optimised pieces of code; if the rest of the VM was similarly optimised, they would take up a much larger proportion of execution time. Therefore anything which can reduce the number of such opcodes executed, or the time taken to execute them, is likely to have a noticeable effect on performance [7, 9]. It is therefore worth considering how an Icon-esque VM can reduce the number of stack operations.

Considering only failure frame operations, a simple thought experiment shows that a few stack-based operations can be statically removed. For example the expression `x := 2` can never fail (since integer creation is a primitive operation), and need not be surrounded by failure frame operations. However the highly dynamic nature of languages such as Icon and Converge limits the analysis that can be performed statically, and it is likely that only small gains could be realised in this particular aspect.

Fortunately, real execution data suggests that Icon-esque VMs are not as inherently stack-based as previously thought.

The reason why failure frames are pushed onto the stack is that they can be nested to an arbitrary depth, and the maximum depth is not statically calculable. By inserting simple logging statements into the relevant part of the Converge VM, one can see how deeply nested such frames are in a real execution; the following figures are from an execution of the Converge compiler on the `convergec.cv` file (part itself of the compiler). Approximately 44% of continuation frames have a maximum of 1 nested failure frame at any point; 60% have a maximum of 2 failure frames at any point; and 80% have a maximum of 3 failure frames at any point. The maximum number of nested failure frames is 17, which occurs only once during execution (the next highest number of nested failure frames is 13, which again occurs only once). With the exception of the maximum number of nested failure frames (which is unusually high), these data are representative of normal executions. They show that the need to cope with an arbitrary depth of nested failure frames is relatively rarely needed. Indeed, 80% of the time it would be practical to statically reserve 3 'slots' in the global execution state for failure frames; when a new function is called, these can be pushed onto the stack along with other data such as the program counter. In the few cases where the 3 slots are not adequate, a fall-back to the traditional stack-based frame approach can be used. In so doing, expensive stack operations would be significantly reduced. Exactly the same technique can be used for generator frames, which are generally even less deeply nested; the same execution data shows that nearly 99% of continuation frames have a maximum of one nested generator frame.

6. Advantages and disadvantages

This section details the experiences I have derived from the way that I and others have used Converge's Icon-like expression evaluation system. By its very nature, this section is not in any way scientific; however, I hope it provides interesting insights for readers.

6.1 Advantages

Two features from Icon are particularly heavily used in Converge. Most obviously, Icon's generator concept provides a lightweight iteration mechanism which also doubles as a simple means of lazy evaluation.

The second candidate is less obvious: it is the concept of failure in `if` constructs. Frequently used libraries are enhanced by using the 'if this function succeeded, assign the value to this variable and then do xyz' idiom. Converge has evolved to make frequent use of this idiom, which is well demonstrated in Converge's dictionaries (known as hash tables or hash maps in many other languages). As in most languages, container items such as dictionaries have a `get` function which returns the value associated with a certain key; if the key does not exist, `get` raises an exception.

Frequently callers need to see if a key exists and, if it does, to get a value back; if it does not exist, no action is to be taken. There are two main solutions to this problem. The first is that `get` returns `null` if the key is not found; however this means that one can not distinguish between a key mapping to `null` and a key not existing. The second is to have a `contains` function which tests for existence of the key; if it exists, then `get` is called. In most languages this idiom is expressed roughly as follows:

```
d := Dict{"a" : 2, "b" : 8}
if d.contains("a"):
  Sys::println(d.get("a"))
```

Not only is the double lookup of `a` an eyesore, and a maintenance accident waiting to happen, but it can be a significant overhead in tight loops. In Python (and probably other languages), it is not uncommon to see an idiom which, encoded in Converge, looks as follows:

```
d := Dict{"a" : 2, "b" : 8}
try:
  v := d.get("j")
  Sys::println(v)
catch Exceptions::Key_Exception:
  pass
```

This idiom makes use of the fact that it is nearly always quicker to catch an exception if a key isn't found than to perform two lookups. In my opinion this idiom obscures the original intent by using (a rather ungainly) exception for performance reasons rather than the expected 'disaster recovery'.

In Converge the functionality of `contains` and `get` can be subsumed into a single function, which succeeds if the key is found, and fails if it is not. In Converge the `find` function can be used as follows:

```
d := Dict{"a" : 2, "b" : 8}
if v := d.find("a"):
  Sys::println(v)
```

Note that Converge contains both `get` (which raises an exception if the key is not found) and `find` (which fails if the key is not found) but not `contains` (which is equivalent to ignoring the return value of `find` if it succeeds). `get` and `find` are so named because when one goes to get something, one is expected to come back with it, but when one tries to find something, it is possible that it will not be found. The `get / find` idiom is used repeatedly throughout the Converge libraries, and the frequency and consistency of its use has proved a real success.

6.2 Disadvantages

The chief disadvantage of the Icon-esque expression evaluation system in Converge is simply that – apart from generators and failure in `if` constructs – it has not been greatly used. This means that although the language has extra concepts, the compiler is more complex, and the VM slower, users rarely use the features that necessitate these costs. There are several possible reasons for this: perhaps users (including myself) tend to think in 'traditional' ways and forget the new functionality available; and maybe problems do not naturally decompose in ways best suited to such an expression evaluation system. However, I suspect the main reason can be deduced from Icon. String scanning (see Section 3.4) is the main place where sophisticated use of Icon's expression evaluation system is used yet modern regular expression libraries subsume most simple cases of string scanning, and formal parsing tools most complex cases. In short: goal-directed evaluation isn't as useful in a modern language as it might have been in Icon's heyday.

7. Conclusions

In this paper I explained the unusual expression evaluation system of the Icon programming language. I then detailed several minor flaws with this system, and showed how the Converge programming language contains a similar expression evaluation system with some flaws fixed and some unfortunately retained. Finally I outlined what, from my own experiences, I consider to be the advantages and disadvantages of such an expression evaluation system.

Three final questions relating to language design remain.

First, was it worth experimenting with an Icon-esque expression evaluation system in a modern dynamically typed OO language? My answer is an easy 'Yes'. In interviews and writing with Ralph Griswold (e.g. [1, 2]) it is clear that one of the aims of Icon was not to be just another (boring) synthesis of a few existing language design staples, but to try something new. Icon fully succeeded in that, and is a genuinely interesting language, worthy of exploration.

Second, if I were to design another general purpose programming language, would I repeat the experiment of including an Icon-esque expression evaluation system? My answer here is more nuanced. I would definitely include generators. I would try very hard to include a way of allowing failure in `if` constructs. However I would not include most of the rest of Icon's features, including the bulk of features related to goal-directed evaluation: the resulting complexities and inefficiencies are not worth it for the general user.

Third, and finally, do I foresee a place for Icon's expression evaluation system? Icon's goal-directed evaluation shines through in string scanning. However it is unclear to me whether Icon's string scanning retains many uses: regular expressions subsume most simple uses, and formal parsing techniques most complex uses. It is therefore equally unclear whether modern general purpose languages would benefit greatly from Icon's expression evaluation system. However, a fruitful future area of research may be to investigate how domain specific languages (including domain specific embedded languages [6]) can integrate similar functionality. Though I suspect that such systems would require more powerful backtracking features than present in Icon, it may provide a good basis for the eventual design.

For those interested in experimenting with the systems discussed in this paper, open-source, portable versions of Icon (`http://www.cs.arizona.edu/icon/`) and Converge (`http://convergepl.org/`) are freely available.

References

[1] D. S. Cargo. An interview with Ralph and Madge Griswold, July 1990. `http://www.cbi.umn.edu/oh/pdf.phtml?id=135` Last accessed May 31 2010.

[2] R. E. Griswold and M. T. Griswold. History of the Icon programming language. pages 599–624, 1996.

[3] R. E. Griswold and M. T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, third edition, 1996.

[4] R. E. Griswold and M. T. Griswold. *The Implementation of the Icon Programming Language*. Peer-to-Peer Communications, third edition, 1996.

[5] R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language*. Prentice-Hall, second edition, 1971.

[6] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4), Dec. 1996.

[7] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, 2005.

[8] N. Schemenauer, T. Peters, and M. L. Hetland. Simple generators. Python PEP 255, June 2001. `http://www.python.org/dev/peps/pep-0255/` Accessed Sep 15 2008.

[9] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *Transactions on Architecture and Code Optimization*, 4(4):1–36, 2008.

[10] L. Tratt. Compile-time meta-programming in a dynamically typed OO language. In *Proc. Dynamic Languages Symposium*, pages 49–64, Oct. 2005.

[11] L. Tratt. The Converge programming language. Technical Report TR-05-01, Department of Computer Science, King's College London, 2005.

[12] L. Tratt. Model transformations in MT. *Science of Computer Programming*, 68(3):169–186, Oct. 2007.

[13] L. Tratt. Domain specific language implementation via compile-time meta-programming. *TOPLAS*, 30(6):1–40, 2008.