

The Wizard of Oz

Chris Bunch

March 5, 2008

Although it was designed more than a decade and a half ago, the Oz programming language boasts a wide array of programming constructs aimed at satisfying users of all programming paradigms. We have explored the various constructs Oz employs and report on how usable this leaves the language. We also report on how the various design decisions made in Oz affect program design at the hands of the average programmer, notably as far as concurrent and distributed computing are concerned.

1 Introduction

Oz is a multi-paradigm programming language designed to bring programmers together. To do so, it implements features from many other major programming languages. Our primary source of information on Oz is *Concepts, Techniques, and Models of Computer Programming*, a textbook used at MIT for (what?). We have also fallen back onto the official Oz website (<http://www.mozart-oz.org>) for information that we found the book lacking in. Since Oz has too many features to summarize here, we have chosen the

features that a fictional ‘average programmer’ would need to perform their job in a rudimentary fashion, how Oz integrates the various programming paradigms, and a few of Oz’s more unusual features.

2 The Oz environment, Mozart

The most widespread implementation of the Oz programming language is Mozart. It compiles Oz code and runs it through a modified Emacs interface. Mozart acts as both a compiler and interpreter for Oz. In the former, Mozart is fed Oz code in the standard fashion and produces an executable or Oz library file. For the latter, Mozart offers a ‘Feed’ option in Emacs that starts up the runtime and only executes a selected region of code. The system then waits for the user to type in more text and feed it, which it then evaluates.

The interpreted method is the main approach taken by the book, as it makes it easier to describe the various programming language constructs that Oz provides. Unfortunately, the interpreter’s syntax is slightly dif-

ferent than the standalone compiled code, so it is not possible to directly take code ran in the interpreted environment and save it into an Oz source code file for static compilation.

The Oz environment is easy to use, and with the code examples from the book and website, provide an excellent starting point for programming in Oz. However, the fact that different syntax is valid in Mozart versus compiling Oz makes it harder to use Oz the way programmers are used to: by writing source code, compiling it, and running it. This forces the programmer to annotate which regions they feed to the runtime and is unfeasible outside of an academic setting.

3 Classifying Oz as a Language

Most of Oz's features and implementation details are not novel as far as programming languages are concerned. The language is dynamically typed and statements are newline delimited. Variable names must start with an uppercase character, yet the book fails to mention this and the compilation errors do not explicitly name this as an issue. Arrays are done in a Lisp-like fashion: they are merely lists that can be appended to or removed from. There is no syntactic sugar to easily reference multi-dimensional arrays, unfortunately.

Like other functional languages, functions are first-class (they can be passed and treated in the same way as variables in imperative languages). This allows for partial function

evaluation and currying of functions. The 'lazy' keyword can be applied to functions, causing them to be lazily evaluated. The return value of functions can also be denoted as a future through using two exclamation points (!!) on the return value. This causes the function to return immediately and return a stub to the asynchronously executing function. If the value truly is needed before it is done calculating, the current thread blocks until the value is returned.

Finally, we move onto Oz's more novel features. The most notable feature is how undefined variables are dealt with. The Oz philosophy is when an undefined variable's value is needed, the thread getting that value blocks until a value is assigned to it. This naturally lends Oz to the concurrent computing paradigm, but as there is no timeout value associated with threads blocking on variables, it may be difficult for commercial applications to use Oz well.

That being said, threads are very lightweight and Oz encourages using as many of them as possible to get computation done. Threads can share global variables or can use 'ports' (essentially message passing) to share data between them. The same rules for variable blocking still apply in the concurrent case: if a thread passes an undefined variable to another thread, and the second thread tries to use it, that thread will block until some other thread defines it.

The large number of keywords and symbols that can be used in Oz make it very easy to use any of the associated features. The programmer will need some time to get used to all of them, but it is more likely that the pro-

grammer will restrict themselves to a small subset of the keywords for simplicity's sake.

4 Input and Output

Oz deals with input and output differently than other languages. It does not provide any methods we were able to find that perform standard input, and the only method that performs standard output is rarely used by both the book and the web site. The preferred method of input in Oz is to create a graphical user interface using the Tk bindings provided by the QtK library and create some input fields for the user to fill in and process. Unfortunately, this makes scripting based on user input unfeasible compared to nearly every other language we have seen.

Output can also be done via GUI creation, but Oz also provides a special GUI called the Browser conveniently for standard output. The Browser is an acceptable alternative to standard output because it closely resembles it and deals with variable state differently than the language specifies. For example, normal usage of an undefined variable in Oz causes that thread to block until the variable is bound. However, the Browser's implementation appears to bypass this rule and can represent undefined variables with a special character until they are defined. Our suspicion is that the Browser consists of at least two threads per variable to be printed: one to try to access the variable, and another to see if that access blocks (if so, it prints the special character). Thus the Browser tool comes as a useful addition in the concurrent

work we have explored this quarter, and the lack of standard input leaves more to be desired.

5 Concurrent Computing

As we have seen, the ability for Oz to use lightweight threads and variables for synchronization (through undefined variables) give it a strong inclination to be an effective concurrent language. The book follows up on it by implementing primitives for concurrent computing. The most well known primitive is likely the lock, but the book goes further and implements monitors and transactions. The Mozart 'gump' parser generator can then be used to integrate these methods into the language itself and make them keywords.

These three primitives are implemented along the lines of their original definitions in the operating systems contexts they come from. This leads to them being very familiar to programmers and as a result, they are very usable. Combined with the ability to send messages, Oz provides an excellent array of concurrent computing primitives that allow programmers to easily write multi-threaded applications.

6 Distributed Computing

Unfortunately, the one paradigm Oz falls short in is the one that is very likely the future of computing. The book and web site each give a chapter on the distributed computing paradigm, but as Oz is designed

as an ‘academic language’, the mechanisms provided are completely unsuitable for usage outside of a pedagogical setting.

Oz begins by providing a novel way to export classes with the intent of passing them over the network to other programs and performing distributed computation. The ‘Pickle’ library can be used to package up any data type and place it in a file. The main use of this in the book and the web is to place this on the web at a known location and have another process fetch it off the Internet and run it. This process is not particularly difficult, as files can be initialized to the contents of a URL in Oz by default.

The problem is that this packages up classes and defines them via a world-unique ticket. The ticket contains the IP address of the host computer and a series of random characters. Should the programmer want to communicate directly between two computers as opposed to having to use web space as an intermediary, they have to hardcode in this value into the client’s program. However, since this value is dynamically generated by Oz, this is simply not possible to hardcode it in as if it were static. The book and site solve this issue by using the Mozart environment to copy and paste the values between programs. Unfortunately, this also results in distributed computation being stuck on a single computer, defeating the entire point of distributed computation. The only possible remedy we see for this is to run a virtualized environment (e.g., Xen) so that copying the unique tickets across computers is a trivial matter.

Although the book fails to mention it, the

web site does provide an API for socket programming. However, the web site explicitly warns that attempting to connect via sockets blocks the entire Oz system (presumably all threads and computation) until the connection succeeds, and as is the case with undefined variables, there is no timeout option on this method call. Furthermore, Oz claims to take a novel approach on distributed computing, and using sockets in this manner is well-known and far from novel.

7 Pros and Cons

Having given a quick overview of a few of Oz’s capabilities, we take this opportunity to summarize the various perks and drawbacks we have experienced while using Oz.

7.1 Pros

Oz succeeds in its goal of bringing together the various programming paradigms into a single, easy to use language. Any amount of computation can easily be wrapped together into a single thread and run independently from other work. The syntax is also close enough to ML to satisfy functional programmers and allows for sequential blocks of code, which may work well enough to satisfy imperative programmers. Object-oriented features work in a similar fashion as existing languages (C++, Java), and allow for an easy transition to Oz.

Oz also boasts an amazing array of keywords and features from other languages, and provides a tool to add more keywords to the

language as needed. This further allows programmers of all paradigms to use Oz effectively, although the sheer number of keywords make it possible for programmers to stick within their own paradigm and use only a small number of those keywords as necessary.

7.2 Cons

The most apparent drawback to Oz is one we have already mentioned: the lack of a method to get information from standard input. This severely damages its usability, and programmers looking to write command-line interfaces with Oz will quickly find themselves out of luck. The reason for this seems to be that Oz is an instructional language, and as such, the Oz environment will be used primarily. Therefore, the programmer can just bind variables dynamically by feeding assignment statements to the runtime. This acts as another form of input to Oz, but is one that cannot be applied to the compiled source code.

Another notable drawback to Oz is the lack of libraries it offers. Although the web site does provide a repository of libraries, most are aimed at computational science. Given the issues surrounding Oz and distributed computing, a group has released a library that does XML-RPC. Unfortunately, the installer for it is Windows-specific, going against Oz's philosophy that libraries can be ported to any architecture and recompiled. This appears to be because Oz libraries can call external C code, which then can be architecture specific.

8 Conclusion

Oz is truly a novel language, and not because it implements anything really new (asides from dataflow variables blocking threads). It is novel because of how it integrates the various programming paradigms together and because of how learning Oz opens a programmer's mind to programming languages in general. Programmers using Oz can clearly see where Oz has taken its constructs from, and how much programming languages have taken from operating systems.

That being said, Oz presents potential but falls short of being a language a programmer can rely on as their Swiss army knife. Oz's problems are far fewer than its successes, but the few problems it has are certainly the death knell to the language. We are not yet at the point in computing when we can give up on standard input, nor can we be confined to our local computers and the Mozart runtime environment.

Yet Oz succeeds at its goal: to be a programming language that offers all paradigms of computation in a way that is easy for programmers to learn. It is certainly an amazing language to use to teach the concepts of programming languages to students who already know a language or to non-majors. Furthermore, the book progresses throughout the paradigms and concepts in an easy-to-follow manner that makes the book worthwhile even as a reference point on programming languages. As we have said, it is not that Oz does anything better, it just does everything a little bit differently.