Futures and Promises in Alice ML

March 13, 2008

Abstract

Various languages have tried to allow useful tools and solutions to concurrent programming, Alice ML is one of these. Alice ML uses futures and promises to help solve the problem of data synchronisation and concurrency. This report explains how these features work, displays a detailed example, performs comparisons to other language solutions and presents related work in this area.

1 Introduction

Computer technology is changing and increasing all the time. Multi core systems are being produced yet to truly exploit this feature of systems, programs will have to be parallelised. This leads to the development of concurrent programming. Concurrent programming is difficult to do well in practice. The concepts and benefits of a good concurrent program are worthwhile. Increased speed, efficient use of resources, better user response time etc are a few of the advantages.

A dominant solution to concurrent programming seems to be threads. This solution is not perfect and many languages have tried to overcome the many disadvantages and difficulties posed by this solution.

Alice ML is an extension of Standard ML and supports concurrency by the use of futures and promises. This solution to concurrent programming appears to be a valid and reasonable one. This report explains futures and promises in Alice ML in more detail, it provides examples and mentions the advantages and disadvantages to this solution. Other language solutions to the concurrent programming problem are also mentioned and compared to Alice ML

2 Context

Concurrency in systems can be extremely useful. It can aid to the efficient use of resources, increase speed and responsiveness. However concurrency is a hard to do in practice, problems can occur with shared memory, deadlock and livelock etc. A popular approach to concurrent programming is threads. Threads are "sequential processes that share memory" [1]. A concurrent program will have many threads, where each thread represents a different control flow through the program. Although threads allow concurrency in a system they can also create a lot of problems. It is difficult to write code that will be accessed simultaneously by more than one thread, because of this many programs that have threads fail or break in some respect as the programmer did not account for that situation .

3 Proposed Solution

Alice ML is an extension of Standard ML and allows support for concurrency. To perform concurrent programming in Alice *futures* are used. A future is a place holder, it holds the "undetermined result of a (concurrent) computation" [6]. Once the result is computed the future is replaced with the result value, this value may be another future. If a thread requests a future the thread blocks until the future has been replaced with the result of the future computation or a failed future. Once the future result has been received the thread continues.

There are four different types of future.:-

- concurrent futures
- lazy futures
- promised futures
- failed futures

Each of these will now be explained in the following subsections.

3.1 Concurrent Future

A concurrent future is basically a concurrent thread which is initialised using `spawn`.

```
spawn 24*9000
val it : int = _future
```

In the above example $24 * 9000$ is computed in a separate thread. Each time `spawn` is used a concurrent thread goes off and tries to evaluate the result of the computation. From above we can see that the output is a future, because the thread has not evaluated the computation yet a future is used as a place-holder for the result. When the result is computed it will globally replace the future.

3.2 Lazy Futures

The result of a lazy expression will be a lazy future. The result of a lazy future is only calculated and returned if another thread requests it. If a thread requests a lazy future a new thread is spawned which carries out the computation of the future. The lazy future is replaced by a concurrent future and when the result is calculated the result replaces the future.

3.3 Promised Futures

When a promise is created a future is also created for that promise. A promise is a handle for a future, it allows the user to define the future later. A promise allows the user to create and eliminate futures, futures are eliminated by fulfilling them. A promise can only be fulfilled once, once it has been fulfilled the result of the computation replaces the future. A promise allows other computation to carry on until the value of the future is needed. Unlike other future types a promised future can only be eliminated by the promise that created it.

This can be particularly useful if you have a list of infinite numbers, if we only want a list up to a specific point for example 10 then we can just evaluate those 10 numbers when we need them rather than calculating them all.

3.4 Failed Futures

If a future fails for example in a call to itself or the future terminates with an exception it is called a lazy future. If a failed future is requested the exception that describes the failure will be returned.

3.5 Advantages

This approach to concurrency helps remove many of the difficulties that are faced by other programming languages that used threads to achieve concurrency. Alice simplifies the process of creating and terminating threads.

Spawning many threads in Alice is reasonably easy, Alice also takes care of all the concurrency. In other languages to prevent threads from accessing parts of code that are considered to be "critical" (i.e. modifying it or performing a calculation in it could affect the result of something else) concurrently requires extra mechanisms to be put in place. Normally parts of code like this are put inside a critical section with the locks, semaphores or monitors applied to that section. A thread would only be allowed access to that part of code if and only if it had obtained the lock etc. Any thread that does not posses the lock would not be granted access. In some languages to use these techniques the programmer themselves has to specify a lot of this functionality. It is rather tricky to figure out exactly which parts of code should have these mechanisms around them. It is quite difficult for programmers to consider how threads accessing a particular part of code simultaneously will behave.

Alice however, simplifies this. In Alice the user does not have to worry about which section of code will be run concurrently and write it in a manner such that this is possible, instead futures are used. Alice also allows the ability of spawning multiple threads very easily to perform separate computations that may depend on each other. For example if a list of 10 results were to be computed using a specific function and the function specified that the value of one calculation of the list depended on the previous. Alice would spawn 10 threads, once each thread reached the particular part of the computation that depended on the previous result it would block until this was computed. By doing this Alice allows the ability to speed up some operations by allowing the ability to distribute computation of some operation among multiple threads. This is very easy for the programmer to do in Alice compared to other programming languages. As the only thing they need to do code wise is to create futures, Alice takes care of the rest.

Using threads in Alice is a lot simpler than most other languages because in other languages the critical sections mentioned earlier need to be considered, threads need to be created and terminated at the correct places in the code. If the programmer forgets to terminate these threads or does not account for some critical parts of code that cannot be run by two threads simultaneously then the program may exhibit some unusual behaviour.

3.6 Disadvantages

I read many papers on Alice and the manuals, none of these resources gave any indication or highlighted any problems that occur with Alice. The only conclusion I could draw from one paper was that Alice still uses threads to attempt to solve concurrency. Threads are non deterministic and since

Alice still uses these, the programs that are created in this language will still have non determinism which contradicts some of the essential and appealing properties of sequential programming, predictability and determinism [1].

Sequential programs are also considered to be easily understandable compared to concurrent programs. If a large concurrent program was created with Alice using the futures and promises it would still be quite difficult to understand and follow what was actually happening. As programs get larger it is more difficult to see what is going on even with using futures and promises. Using futures and promises does create a slightly clearer understanding of the program than it would by using threads in another language, yet the concept could still be confusing as it scales to larger programs.

Also using threads whether its with Alice or another language it is difficult to prove that the same behaviour will be exhibited each time, also it can be difficult to test the system fully to ensure that the program does not break because some of the threads performed some unexpected behaviour and produced exceptions. Edward A. Lee explained a very good example of this in his paper "The Problem with Threads" [1]. His group were developing a kernel for a project called Ptolemy II, this was "a modelling environment supporting concurrent models of computations" [1]. The goal of the project was the allow the modification of the concurrent programs via a graphical user interface while the concurrent programs were still executing. This was quite challenging as it meant that no thread should ever be able to see an inconsistent view of the program. They used Java threads with monitors and tested the kernel thoroughly. After the system passed all the tests it was released, four years later the system deadlocked. This example shows that programming anything with multiple threads is extremely difficult and even with rigorous testing things can still go undetected, even things that cause serious problems such as deadlock.

4 Example

This example was a modified version of a example from the Alice manual [6]. This example is about buses on a bus route that stop at a particular bus stop.
Itshows buses arriving and leaving the busstop. It uses futures
and promises to add and remove buses from the bus stop queue. This might be useful when monitoring bus tops and the flow of buses through them.

First of all an abstract data type is created called BUSROUTE. This allows any number of buses to arrive at the bus stop. When buses are leaving the bus stop the oldest bus in the queue is the one that leaves, similar to a FIFO (First in First Out) structure. If no buses are in the queue it suspends

until some arrive.

```
signature BUSROUTE =
 sig
       type 'a bus_stop
       val bus_stop : unit -> 'a bus_stop
       val bus_arriving : 'a bus_stop * 'a -> unit
       val bus_leaving : 'a bus_stop -> 'a
 end;
```

Once the data type is set up the bus route structure is actually set up that has the signature that we previously defined. When the type is defined we say the bus stop is a difference list (i.e. the first reference has the tail of the list and the second reference has the head of the list). The end of the list is a promised future as arriving buses will be appended to the end of the list. The value of the queue (or list) of buses depends on the previous queue. When a bus is leaving the bus stop the `bus_leaving` operation will block until the promised future has been evaluated. i.e. the queue of buses at the bus stop is formulated. Once the result is computed the future is replaced by the result. When there is no buses in the queue the bus leaving thread will be suspended until a bus actually arrives.

```
structure BusRoute  : BUSROUTE =
 struct
       open Promise

       type 'a bus_stop = 'a list promise ref * 'a list ref

       fun bus_stop() =

               let
                       val p = promise()
               in
                       (ref p, ref (future p ))
               end
       fun bus_arriving( (newBus, getBus), x) =
                           let
               val p' = promise()
       val p = Ref.exchange (newBus, p')
       in
               fulfill(p, x::future p')
```

```
        end

        fun bus_leaving(newBus, getBus) =
                let val p' = promise()
                val xs = Ref.exchange (getBus, future p')
                in
                        fulfill (p', tl xs); hd xs
                end
        end;
structure BusRoute :
  sig
     type 'a bus_stop = 'a list BusRoute.promise ref * 'a list ref
     val bus_stop : unit -> 'a BusRoute.bus_stop
     val bus_arriving : 'a BusRoute.bus_stop * 'a -> unit
     val bus_leaving : 'a BusRoute.bus_stop -> 'a
  end
```

A bus route must be created using the above structure before we can do anything with it.

```
val bus_route : int BusRoute.bus_stop = BusRoute.bus_stop();
val bus_route : int BusRoute.bus_stop =
   (ref (promise{|_future|}), ref (_future))
```

To add buses arriving and remove buses leaving the bus stop (aka BusRoute) the following calls are made.

```
 - BusRoute.bus_arriving(bus_route, 4); BusRoute.bus_arriving(bus_route, 7); Bus
val it : unit = ()
- BusRoute.bus_leaving(bus_route);
```

The example above displays how futures and promises can be used to help data synchronisation in a list. This is a very trivial example but it can also be applied to things such as asynchronous message queues.

5 Resources

The paper "Alice through the Looking Glass" [4] explains some of the intricate details of Alice. The creation of Alice and some of its features were due to the development and increase of open programming. Characteristics such as modularity, dynamicity, security, distribution and concurrency

have become increasingly popular. Alice has tried to solve these by extending standard ML to allow concepts such as futures, higher-order modules, packages, pickling and proxy functions.

Throughout the paper each of these concepts is described in more detail. The mentioned the various different types of futures and show small examples of how each of these can be used.

"A concurrent lambda-calculus with futures" [2] introduces a new lambda calculus which models the semantics of languages such as Alice ML. It describes how Alice ML contains static type inference and uses a mixture of eager and lazy threads. It also states that many of the Alice features were inspired by Mozart-Oz. Futures also provide a useful mechanism for dealing with network latency, it allows multiple threads to perform computations that do not actually require the result of a future to do computations up to that point concurrently. The rest of the paper proceeds into heavy lambda calculus for futures.

I also discovered a really good paper that mentions the problems with threads, appropriately named " The problem with threads" [1]. The paper presents many arguments for why using threads as a solution for concurrent programming is in fact a backwards solution to the problem. The author shows a few examples of where even with careful programming errors with threads can still occur. It is very difficult to use threads correctly, even if you are an experienced programmer there is bound to be come situation that you did not account for. It is extremely hard to test a program or system that is multi threaded fully to ensure that no errors will occur, this is one of the main arguments of the paper. The paper also proposes some other alternatives to threads for solving the problem of concurrency.

The other main resource that was used for this report was the actual Alice manual [6]. Most of this paper elaborated on many of the concepts and techniques explained in the manual. The manual explained most of the concepts in Alice ML with small examples detailing how such things would be carried out. It explains futures and the various types and also contains information on many other Alice features that were not focused on in this paper. For example packages, pickling, components and distribution.

6 Related Work

This section describes how some other languages have approached the problem of concurrency. It also compares their approach to Alice Ml.

6.1 Oz

Oz is a high-level programming language that incorporates many concepts from object-oriented and functional programming. It is "dynamically typed and has first-class procedures, classes, objects, exceptions and sequential threads synchronising over a constraint store" [5]. The notion of futures and promises in Alice ML was actually taken from Oz. One of the goals of the Alice team was to allow some of the functionality of Oz on top of a typed functional language [4].

In Oz, futures are similar to Alice, the syntax for creating a future is however different. Any thread that requests the value of a future will be blocked until the future result has been calculated. Oz does not have any feature called promises, it does however allow the termination and joining of threads easily. Although the programmer does have to suspend the main thread until the other threads have joined, Alice ML takes care of this for the programmer.

Mozart is a development platform for distributed systems that is based on Oz. It is similar to Alice as it also has support for futures, however it is considered to be a lot more complex.

6.2 Java

Java is an object-oriented language. If the programmer does not want a method of an object to be accessed simultaneously by more than one thread then they can make it `synchronized`. This prevents threads from invoking the same synchronised method for the same object concurrently. If one thread has gained access to the synchronised method, any other threads trying to access the method for the same object will be blocked until the first thread has finished with the object. In this respect Java uses locks, or monitor locks. When the first thread accesses the synchronised method it obtains the lock for that object, the lock is only released once the thread is finished with the object. Once an objects synchronised method has been invoked and finished all changes that have been made to the object are visible to all other threads.

This is very different to Alice as java is object-oriented. Alice also does not use this lock mechanism. Java does have an interface for futures but does not support the concept of promises. Also in Java threads have to be terminated by the programmer. With large scale programs this can soon become problematic and hard to understand. The programmer has to define a lot of the thread behaviour, the general use of threads in Java is harder to do than in Alice. Also if the programmer does not specify some methods to be synchronized when they should be the threads will not behave as expected

and the results will be different. The programmer has to aware of which parts of code need to have the locking mechanism placed upon them.

6.3 MultiLisp

MultiLisp is a version of Scheme which has been extended to allow parallelism. Futures in MultiLisp are also very similar to Alice. However when a process is blocked because it needed the future value and it is waiting for the future value to be computed, this is called "touching a future". Passign of futures as parameters is also possible as the value of the future does not need to be known.

MultiLisp also has the `pcall` function which can be used to evaluate many different variable concurrently. This can be bad however as the number of variable increases [3].

7 Conclusion

Overall Alice ML seems to have a reasonable solution to the concurrency problem. It does however still involve threads, which as mentioned earlier may not be a good solution to the problem. Threads pose many problems as programmers find them very difficult to use and they may make understanding programs very difficult. Alice ML does however simplify and make threads easier to understand compared to some other languages, for example Java. All in all futures and promises are a clever technique, they allow many computations to be performed concurrently and make efficient use of resources.

References

[1] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[2] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. In *5th International Workshop on Frontiers in Combining Systems, Lecture Notes in Computer Science 3717*, pages 248–263., 2005.

[3] Jr. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[4] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. Alice Through the Looking Glass. *Trends in Functional*

Programming, volume 5:79–96, Intellect Books,2006. http://www.ps.uni-sb.de/Papers/abstracts/alice-looking-glass.html.

[5] Christian Schulte. The Oz Programming System. http://www.ps.uni-sb.de/oz2/.

[6] Saarland University. The Alice Manual. http://www.ps.uni-sb.de/alice/manual/.

8 Appendix - Screen shots of Example

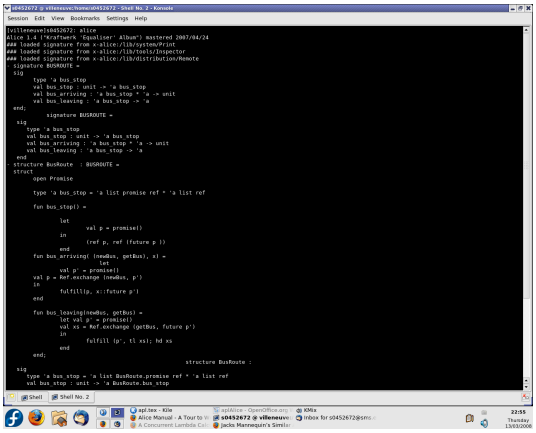

Figure 1: Screen shot of creation of a bus stop queue

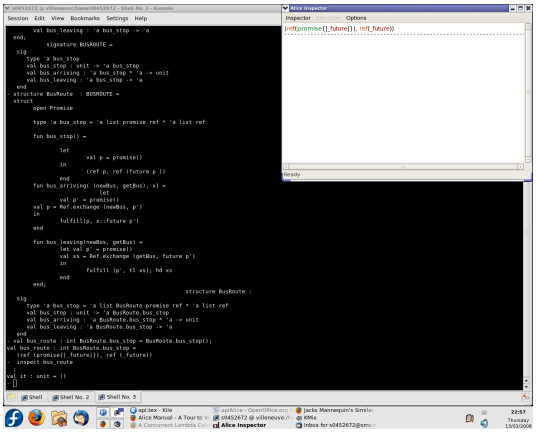

Figure 2: Bus stop queue with inspector

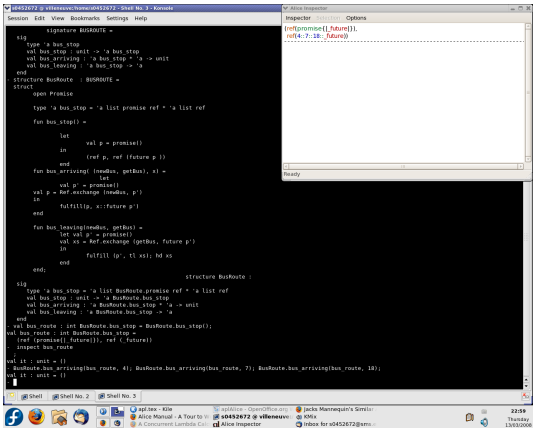

Figure 3: Buses arriving at bus stop, inspector has changed to reflect this

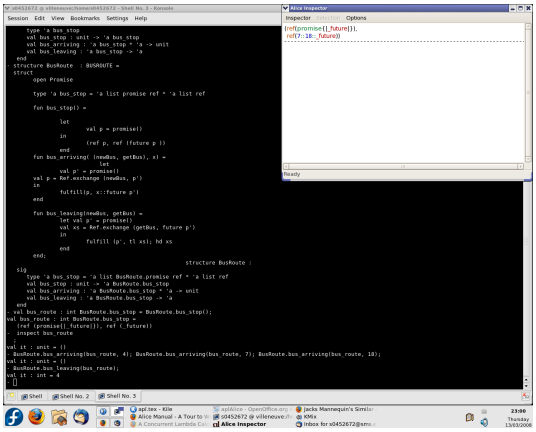

Figure 4: Bus leaving bus stop, inspector has changed to reflect this

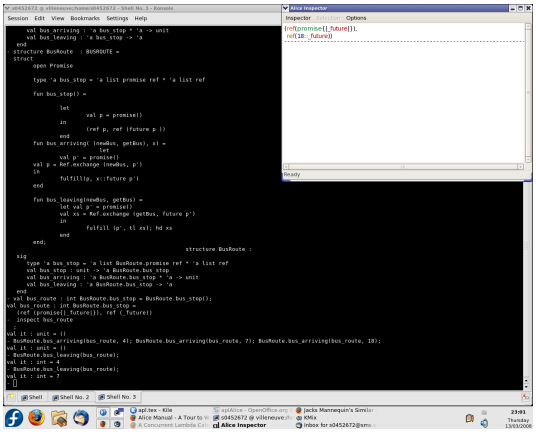

Figure 5: Bus leaving bus stop, inspector has changed to reflect this

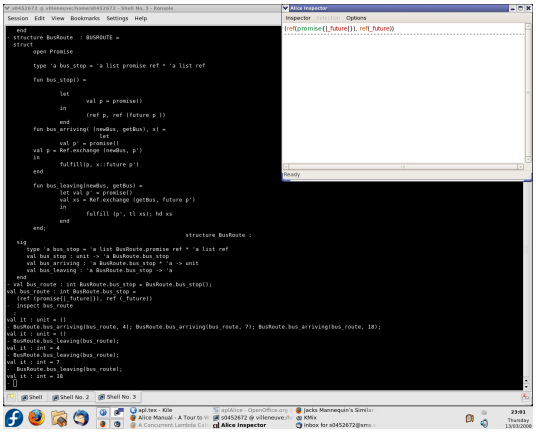

Figure 6: All buses removed from bus stop