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.

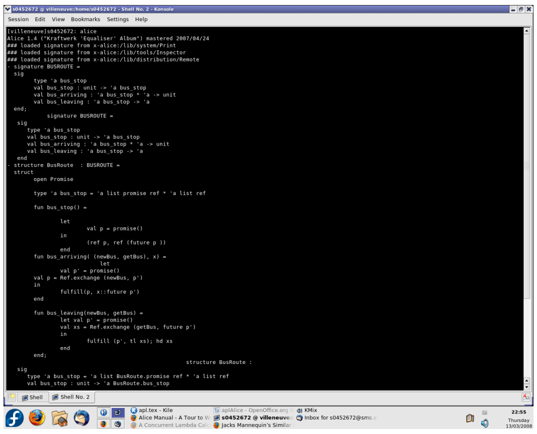
[4] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. Alice Through the Looking Glass. *Trends in Functional*

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/.

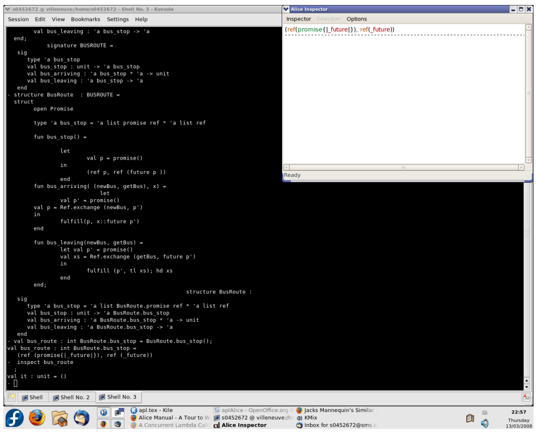
[6] Saarland University. The Alice Manual. http://www.ps.uni-sb.de/alice/manual/.

8 Appendix - Screen shots of Example


s0452672 @ villeneuve:/home/s0452672 - Shell No. 2 - Konsole
Session Edit View Bookmarks Settings Help
Shell
Shell No. 2
22:55
Thursday
13/03/2008

```
[villeneuve]s0452672: alice
Alice 1.4 ("Kraftwerk 'Equaliser' Album") mastered 2007/04/24
### loaded signature from x-alice:/lib/system/Print
### loaded signature from x-alice:/lib/tools/Inspector
### loaded signature from x-alice:/lib/distribution/Remote
- 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;
            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
- 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
```

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

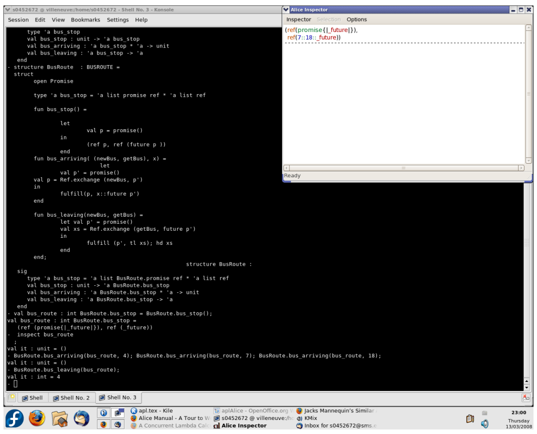

s0452672 @ villeneuve:/home/s0452672 - Shell No. 3 - Konsole
Session Edit View Bookmarks Settings Help
Alice Inspector
Inspector Selection Options
Ready
Shell
Shell No. 2
Shell No. 3
22:57
Thursday
13/03/2008

```
        val bus_leaving : 'a bus_stop -> 'a
  end;
            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
- 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
- val bus_route : int BusRoute.bus_stop = BusRoute.bus_stop();
val bus_route : int BusRoute.bus_stop =
   (ref (promise{|_future|}), ref (_future))
-  inspect bus_route
  ;
val it : unit = ()
- 
```

```
(ref(promise{|_future|}), ref(_future))
```

Figure 2: Bus stop queue with inspector

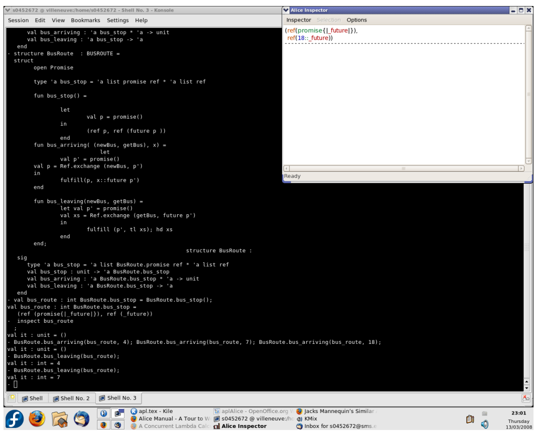
Figure 3: Buses arriving at bus stop, inspector has changed to reflect this


s0452672 @ villeneuve:/home/s0452672 - Shell No. 3 - Konsole
Session Edit View Bookmarks Settings Help
Alice Inspector
Inspector Selection Options
Ready
Shell
Shell No. 2
Shell No. 3
23:00
Thursday
13/03/2008

```
      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
- 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
- val bus_route : int BusRoute.bus_stop = BusRoute.bus_stop();
val bus_route : int BusRoute.bus_stop =
   (ref (promise{|_future|}), ref (_future))
-  inspect bus_route
  ;
val it : unit = ()
- BusRoute.bus_arriving(bus_route, 4); BusRoute.bus_arriving(bus_route, 7); BusRoute.bus_arriving(bus_route, 18);
val it : unit = ()
- BusRoute.bus_leaving(bus_route);
val it : int = 4
- 
```

```
(ref(promise{|_future|}),
 ref(7::18::_future))
```

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

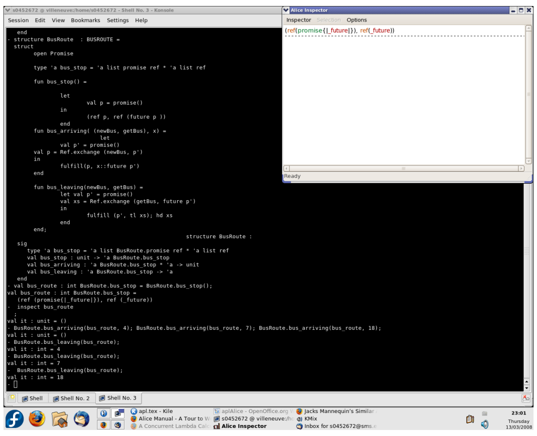


```
    val bus_arriving : 'a bus_stop * 'a -> unit
    val bus_leaving : 'a bus_stop -> 'a
  end
- structure BusRoute  : BUSROUTE =
  struct
        open Promise

        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
- val bus_route : int BusRoute.bus_stop = BusRoute.bus_stop();
val bus_route : int BusRoute.bus_stop =
  (ref (promise{|_future|}), ref (_future))
-  inspect bus_route
  ;
val it : unit = ()
- BusRoute.bus_arriving(bus_route, 4); BusRoute.bus_arriving(bus_route, 7); BusRoute.bus_arriving(bus_route, 18);
val it : unit = ()
- BusRoute.bus_leaving(bus_route);
val it : int = 4
- BusRoute.bus_leaving(bus_route);
val it : int = 7
- 
```

```
(ref(promise{|_future|}),
 ref(18::_future))
```

s0452672 @ villeneuve:/home/s0452672 - Shell No. 3 - Konsole
Session Edit View Bookmarks Settings Help
Alice Inspector
Inspector Selection Options
Ready
Shell
Shell No. 2
Shell No. 3
apl.tex - Kile
Alice Manual - A Tour to W
A Concurrent Lambda Calc
aplAlice - OpenOffice.org
s0452672 @ villeneuve:/h
Alice Inspector
Jacks Mannequin's Similar
KMix
Inbox for s0452672@sms.
23:01
Thursday
13/03/2008

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



```
  end
- 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
- val bus_route : int BusRoute.bus_stop = BusRoute.bus_stop();
val bus_route : int BusRoute.bus_stop =
  (ref (promise{|_future|}), ref (_future))
-  inspect bus_route
  ;
val it : unit = ()
- BusRoute.bus_arriving(bus_route, 4); BusRoute.bus_arriving(bus_route, 7); BusRoute.bus_arriving(bus_route, 18);
val it : unit = ()
- BusRoute.bus_leaving(bus_route);
val it : int = 4
- BusRoute.bus_leaving(bus_route);
val it : int = 7
-  BusRoute.bus_leaving(bus_route);
val it : int = 18
- 
```

```
(ref(promise{|_future|}), ref(_future))
```

s0452672 @ villeneuve:/home/s0452672 - Shell No. 3 - Konsole
Session Edit View Bookmarks Settings Help
Alice Inspector
Inspector Selection Options
Ready
Shell
Shell No. 2
Shell No. 3
apl.tex - Kile
Alice Manual - A Tour to W
A Concurrent Lambda Calc
aplAlice - OpenOffice.org
s0452672 @ villeneuve:/h
Alice Inspector
Jacks Mannequin's Similar
KMix
Inbox for s0452672@sms.
23:01
Thursday
13/03/2008

Figure 6: All buses removed from bus stop