
TL

A Monad for Algebraic Thread Manipulation

Galen Menzel

18 April 2005

The Problem

A general class of problems that are prevalent on the web:

- ▶ Request data
- ▶ Compute with the data
- ▶ Request more data
- ▶ Compute with the data
- ▶ ...

The Problem (continued)

These problems are hard to solve

- ▶ Data comes from multiple sources, with very high latency
 - Serial requests takes too long
 - We need an easy way to perform requests concurrently
- ▶ The network is unreliable
 - We need to be able to time out gracefully
- ▶ We need a robust way to specify computations

Example: Parallel Or

Description:

- ▶ n servers, each with a boolean value
- ▶ We want to calculate the logical or in minimum time

To solve this problem with threads, we need:

- ▶ Explicit creation of a thread for each of the n requests
- ▶ Some shared, synchronized data structure for the values
- ▶ A structure to keep track of all the thread IDs

TL simplifies the solution considerably

Introducing TL

The basic idea: *time lists* (hence *TL*)

- ▶ A time list is a time-ordered list of responses to requests
- ▶ Elements of a time list are ordered by time *received*
- ▶ A time list can be read and manipulated like a normal list
- ▶ Serves as a bridge between concurrent and sequential code

Creating a TL expression

We have a polymorphic data type for time lists:

```
data TL a = ...
```

A `TL a` is a side-effecting action that returns a list of `as`.

We can create the most basic TL functions from IO actions:

```
tlExpr :: IO a -> TL a
```

Given the function `req :: IO Int` that requests a value from a server, we can define the corresponding TL expression like this:

```
tlReq :: TL Int  
tlReq = tlEval req
```

Evaluating a TL expression

TL expressions are evaluated with the `tlEval` function:

```
tlEval :: TL a -> IO (TimeList a)
```

So to execute a request and get its corresponding time list, use

```
do tl <- tlEval tlReq  
  ...
```

TimeList to List

`TimeList a` is an abstract representation of a currently evaluating TL expression. There are two ways to obtain a list from a `TimeList`

- ▶ `tlList :: TimeList a -> IO [a]`
 - Always returns an infinite list
- ▶ `tlKill :: TimeList a -> IO [a]`
 - Kills the computation and returns a finite list

The List Monad

In Haskell, lists are Monads and MonadPlus:

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
```

```
instance MonadPlus
    mzero = []
    mplus = (++)
```

For instance

```
as >>= \x -> bs >>= \y -> return (x,y)
```

produces the cross product of as and bs.

The Basis for the Time List Monad

Modifying the list operations to preserve time-list ordering yields a compact implementation of time lists.

```
return x = ... (new singleton time list)
mzero = ... (new empty time list)
```

Mplus: Parallel Composition

Two TL expressions can be merged into a single TL expression with the `mplus` function:

$$\text{mplus} :: \text{TL } a \rightarrow \text{TL } a \rightarrow \text{TL } a$$

`a 'mplus' b` is the TL expression which, when executed, evaluates `a` and `b` in parallel, and merges the two time lists into a single time list.

`mplus` can be thought of as time-list concatenation. That is, the conjoining of two time lists so that time-order is preserved.

More on Parallel Composition

Given req1 that requests a value from server 1, and req2 that requests a value from server 2, consider

$\text{req1} \text{ 'mplus' } \text{req2}$

Let $v1$ be the value from server 1 and $v2$ the value from server 2. The possible values of the resulting time list are these:

$[], [v1], [v2], [v1, v2], [v2, v1]$

Generalized Parallel Composition

`msum` is a standard monad function:

```
msum :: [m a] -> m a
msum = foldl mplus mzero
```

Given a list of TL expressions $es = e_1, \dots, e_n$ can call them all, and obtain a single time list of all their values:

```
do tl <- tlEval (msum es)
    ...
```

Sequential Composition

Given two TL expressions f and g , we can compose them together sequentially with the ($\gg=$) operator:

$$f \gg= \lambda x \rightarrow g$$

The result is the TL expression that, which evaluated, executes f and *for each value in f 's time list*, binds the value to x and evaluates g .

Just as with the list monad, we can compute the cross-product of two TL expressions f and g :

$$f \gg= \lambda x \rightarrow g \gg= \lambda y \rightarrow \text{return } (x,y)$$

Unlike lists, however, the order of elements in this expression is not deterministic, since it is based on response time.

A Solution to Parallel Or

We can now give a solution to the parallel or problem:

```
por :: [TL Bool] -> IO Bool
por reqs = do tl <- tLEval (msum reqs)
             bs <- tlList tl
             return (or (take (length reqs) bs))
```

Notice how the time list acts as a bridge between `tLEval`'s concurrent evaluation and `or`'s sequential evaluation.

Example: Flow-Rate Calculation

We can calculate the number of values a TL expression returns within t time units:

```
flowRate :: TL a -> Int -> IO Int
flowRate f t = do tl <- tlEval f
                  threadDelay t
                  rs <- tlKill tl
                  return (length rs)
```

Example: Polling

We can poll a service every t time units until its returned value meets a certain criterion:

```
timer :: Int -> TL ()
timer t = tlExpr (threadDelay t)

poll :: TL a -> Int -> TL a
poll r t = r 'mplus' (timer t >> poll r t)

pollUntil :: TL a -> Int -> (a -> Bool) -> IO a
pollUntil r t f = do tl <- tlEval (poll r t)
                    vs <- tlList tl
                    v <- return (head (filter f vs))
                    v 'seq' tlKill tl
                    return v
```