Concurrent Haskell

Many applications are most easily written as a set of cooperating processes/threads executing in an interleaved fashion on a single location.

For example a GUI with one thread for each window, and for each device: keyboard, mouse etc.

The threads must be stateful i.e. interact with the program’s environment, e.g. keyboard thread consumes typed characters.

Stateful threads must be scheduled fairly: cannot ignore them as we can GpH’s stateless threads.

Concurrent Haskell

In Concurrent Haskell stateful threads are created by \texttt{forkIO}, and named with a \texttt{ThreadId}:

\begin{verbatim}
forkIO :: IO() -> IO ThreadId
myThreadId :: IO ThreadId
\end{verbatim}

A concurrent web service example:

\begin{verbatim}
acceptConnections :: Config -> Socket -> IO ()
acceptConnections config socket
  = forever (do { conn <- accept socket ;
                  forkIO (serviceConn config conn)}
\end{verbatim}

In Concurrent Haskell stateful threads are often called I/O threads.

Implicit Synchronisation

As in GpH I/O threads synchronise & communicate implicitly if they share values, e.g.

\begin{verbatim}
import Control.Concurrent

x = fib 30

sillyA = putStrLn ("SillyA "++show (x+3))
sillyB = putStrLn ("SillyB "++show (x-5))

main =
do
  -- Create threads
  forkIO sillyA
  forkIO sillyB
  -- Wait for completion
  threadDelay 2000000
  putStrLn "Done"
\end{verbatim}
Explicit Synchronisation

I/O threads can synchronise & communicate explicitly to lock resources and to describe non-deterministic stateful computations.

Explicit synchronisation occurs in the IO monad using polymorphic semaphores primitives called MVars.

A value of type MVar t is a location that is either

- empty, or
- holds a value of type t

newEmptyMVar :: IO (MVar a)
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO()
isEmptyMVar :: MVar a -> IO Bool

MVar Semantics

<table>
<thead>
<tr>
<th>Operation</th>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>takeMVar</td>
<td>Block</td>
<td>Return contents, leave MVar empty</td>
</tr>
<tr>
<td>putMVar</td>
<td>Fill MVar</td>
<td>Block</td>
</tr>
</tbody>
</table>

MVar example: count the number of client threads running.

acceptConnections :: Config -> Socket -> IO ()
acceptConnections config socket
  = do
    count <- newEmptyMVar
    putMVar count 0
    forever (do
      conn <- accept socket
      forkIO (do
        inc count
        serviceConn config conn
        dec count))

inc,dec :: MVar Int -> IO ()
inc count = do
  v <- takeMVar count
  putMVar count (v+1)
dec count = do
  v <- takeMVar count
  putMVar count (v-1)

Concurrent Haskell Channels

An MVar directly implements:

- a shared data structure, e.g. a write-lock on a file
- a one-place channel

Using MVars it’s easy to build buffered channels

- unbounded FIFO queues
- multireader/multiwriter channels

Channels facilitate the operation of asynchronous systems as the sender doesn’t have to wait for recipient and vice versa.
Channel Interface

newChan :: IO (Chan a)
-- create a channel for values of type a

readChan :: Chan a -> IO a
-- read a value from a channel

writeChan :: Chan a -> a -> IO ()
-- write a value into a channel

Exceptions

Haskell supports exceptions in stateful (IO) code:

```haskell
catch :: IO a -> (IOError -> IO a) -> IO a
userError :: String -> IOError
ioError :: IOError -> IO a
```

Example:

```haskell
catch (do { h <- openFile "foo";
          processFile h })
  (\e -> putStrLn "Oh dear")
```

Concurrent Haskell enables us to manage exceptions in threads, e.g.:

```haskell
raiseInThread :: ThreadId -> Exception -> a
```

Glasgow distributed Haskell (GdH)

GdH is a superset of both GpH and Concurrent Haskell.

It introduces a new abstract data type PEId to represent locations. All locations and a thread’s location can be discovered:

```haskell
allPEId :: IO [PEId]
myPEId :: IO PEId
```

The location of stateful objects like a database or a thread can be discovered:

```haskell
class Immobile a where
  owningPE :: a -> IO PEId
```

```haskell
instance Immobile PEId
instance Immobile (MVar a)
instance Immobile ThreadId
```

Resource discovery, e.g. locating files, environment variables etc, uses standard Haskell interrogation commands.

Remote Evaluation

Like Java RMI `revalIO` job p causes the calling thread to block until the execution of job at the location p completes and returns a result.

```haskell
revalIO :: IO a -> PEId -> IO a
```

It is easy to define a distributed fork using `revalIO`:

```haskell
rforkIO :: IO () -> PEId -> IO ThreadId
```
GdH Channels

GdH has multireader/multiwriter channels with the same interface as Concurrent Haskell channels, and an almost identical implementation.

However, unlike Concurrent Haskell, the readers and writers of GdH channels may be on different PEs.

Distributed Exceptions

The exception libraries must be adapted to work in a distributed setting, i.e. so an exception raised on one PE can be handled on another.

We make ThreadId and instance of the immobile class

```
instance Immobile ThreadId
raiseInThread th ex = revalIO (Concurrent.raiseInThread th ex) th
rforkIO job p = revalIO (forkIO job) p
```

GdH Broadcast Ping

```
main =
do
pes <- allPEId
mapM loop pes
where
loop pe =
do
putStr ("Pinging "++(show p)++" ... ")
(name,ms) <- timeit (revalIO remote pe)
putStrLn ("at "++name++" time="++show ms++")
remote = getEnv "HOST"
```

Typical Output

```
Pinging 262344 ... at ushas time=0ms
Pinging 524389 ... at bartok time=3ms
Pinging 456324 ... at ncc1705 time=2ms
Pinging 786442 ... at brahms time=3ms
```
More Realistic Examples

A **cooperative editor** allows users at remote locations to simultaneously edit the same file(s).

A user **fetches** the file, changes it and **sends** the updated file.

Figure 1: Cooperative Editor screenshot. All X screens re-directed to one host.

The **Distributed Simulation Controller** allows a group of users at remote locations to exchange control of a simulator, view the results of the simulation, and engage in a chatroom style discussion of the results.

Figure 2: Distributed Simulation Control

---

**Comparing GdH and Java/RMI**

Although **ping** and the simulation controller are small examples they illustrates the differences between GdH and Java/RMI.

<table>
<thead>
<tr>
<th>Programming style</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ping</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Lines of code</td>
</tr>
<tr>
<td>Lines for coordination</td>
</tr>
</tbody>
</table>

Table 1: Ping & Sim. Ctrl. Prog. Style.

The GdH programs are both shorter overall and require less coordination code. There are several reasons:

- As GdH systems are closed it’s ping is a single program. In contrast Java/RMI supports open systems and must have both client & server programs.

- Computation and coordination are concisely expressed using high-level constructs, e.g. mapM.

- Java/RMI also forces the inclusion of fault tolerance.
### Performance

<table>
<thead>
<tr>
<th></th>
<th>GdH</th>
<th>Java/RMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>ushas (local)</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>bartok</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>ncc1705</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>brahms</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2: Ping Performance (times in ms).

Where a remote PE is accessed, communication costs dominate and there is no significant difference between the languages.

Where communication is local, the Java/RMI client and server are slower as they must perform inter-process communication and registry lookup.

---

### GdH Classification

GdH is

- **location aware**: locations are named with PEIds, and computations can be sent to named locations.
- **closed**: an arbitrary number of pure & stateful threads can be created dynamically, but all threads must belong to the same program.
- used for **small-scale distribution**: the distributed system is a single large GdH program.