The Design and Implementation of Glasgow distributed Haskell [1]

R.F. Pointon, P.W. Trinder, and H-W. Loidl

Dept. of Computing and Electrical Engineering,
Heriot-Watt University,
Edinburgh, EH14 4AS.
{RPointon,Trinder,HWLoidl}@cee.hw.ac.uk

Abstract. This paper outlines the design and implementation of Glasgow distributed Haskell (GdH), a non-strict distributed functional language. The language is intended for constructing scalable, reliable distributed applications and is a superset of both Concurrent Haskell and Glasgow parallel Haskell.
GdH supports both pure and impure (I/O) threads; Processing Elements (PEs) are made explicit so a program can use resources unique to a PE, and I/O threads can be created on a named PE. Communication and synchronisation may be explicit or implicit. The PE that uniquely owns a resource is identified by a method of a new `immobile` type class. Fault tolerance is provided by distributed exception handling.
GdH is implemented by extending the GUM runtime system underlying the Glasgow Haskell Compiler. To give a flavour of GdH programming, numerous examples are given, together with two demonstrators.

1 Introduction

Distributed languages are used for a number of reasons. Many applications, particularly those with multiple users, are most naturally structured as a collection of processes distributed over a number of machines, e.g. multi-user games, or software development environments. Applications distributed over a network of machines can be made more *reliable* because there is greater hardware and software redundancy: a failed hardware or software component can be replaced by another. Distributed architectures are more *scaleable* than centralised architectures: additional resources can be added as system usage grows.

We distinguish between large-scale and small-scale distribution. Large-scale distributed applications are supported by standard interfaces like CORBA [Sie97] or Microsoft DCOM [Mer96] and may have components written in multiple languages, supplied by several vendors, execute on a heterogeneous collection of platforms and elaborate failure mechanisms. In contrast, small-scale distributed programs entails components written in a single language, is typically constructed by a single vendor, and is often restricted to an homogeneous

[1] Supported by research grant GR/M 55633 from UK's EPSRC and APART fellowship 624 from the Austrian Academy of Sciences.

network of machines, with a simple model of failures. Small-scale distributed applications are typically constructed in a distributed programming language, e.g. Java with Remote Method Invocation (RMI) [DSMS98]. A distributed language allows the system to be developed in a single, homogeneous, framework, and makes the distribution more transparent to the programmer.

Functional languages potentially offer benefits for small-scale distributed programming, and several have been developed, e.g. Kali Scheme [CJK95], Facile [GMP89], OZ [HVS97], Concurrent Clean [PV98], and Pict [PT97]. They allow high level distributed programming, e.g. capturing common patterns of distribution as higher-order functions. Functional languages provide type safety within the constraints of a sophisticated, e.g. higher-order and polymorphic, type system. Several benefits accrue if significant components of the application are *pure* i.e. without side-effects. Such components are easy to reason about, e.g. to optimise, derive or prove properties. Pure components can be evaluated in arbitrary order, e.g. lazily or in parallel. It may be easier to implement fault tolerance for pure computations because a failed computation can be safely restarted [TPL00].

We have designed and implemented a language based on (non-strict) distributed graph reduction, and this confers some specific advantages. Not all communication between processes need be explicit, in particular the shared graph model means that a process has implicit (read-only) access to variables shared with other processes. Moreover, all data transfer between processes is lazy and dynamic. The cost of laziness is an additional message from the recipient requesting the data, but there are several specific benefits. Lazy transfer is useful if part of a large (or infinite) data structure are to be exchanged. Logically the entire data structure is exchanged, but the receiving process will only demand as much of the data structure as is needed. Lazy transfer automatically avoids the problem of a fast producer flooding a slow consumer's memory. Dynamic transfer is useful if the amount of data to be sent is hard to determine *a priori*, or varies between program execution.

Section 4 outlines the design of GdH which incorporates two classes of threads: pure threads and side-effecting I/O threads. Pure threads communicate and synchronise using shared variables, and are introduced and synchronised using parallel and sequential composition. Evaluation strategies abstract over the compositions to allow polymorphic and higher-order description of parallel behaviours [THLP98]. PEs are identified so a program can use resources unique to a PE. A monadic `rforkIO` primitive creates a remote I/O thread on a named PE. Some communication is implicit: I/O and pure threads on one PE can share variables with I/O and pure threads on other PEs. Explicit communication and synchronisation is provided using polymorphic semaphores (`MVars`) within the I/O monad. Higher-level constructs like channels and buffers are constructed by abstracting over distributed `MVars`. Resources like an `MVar` that are unique to a PE are identified by a method of the `immobile` class that associates a PEId with the resource. Fault tolerance is provided by distributed exception handling, i.e. an exception can be raised on one PE and handled on another. In effect GdH is a minimal superset of the GpH and Concurrent Haskell languages, as discussed in section 3.

GdH is implemented by adding four primitive mechanisms to the GUM runtime system [THM+96] underlying the Glasgow Haskell Compiler [PHH+93]. Section 5 outlines the primitives, and shows how GdH constructs are implemented using them. The primitives are a list of PE identifiers, a remote co-routine operation (`revalIO`), a mechanism for retrieving the owning PE of immobile objects, and distributed exception handling.

Section 6 presents and discusses two small GdH demonstration programs. Section 7 discusses future work and section 8 concludes.

2 Related Work

There are many distributed languages, and it is not possible to cover all of them in detail. Instead we outline a framework in which to locate our work. Two key concepts in distributed languages are processes and threads.

- A *process* has a private name (or address) space. It communicates explicitly with other processes using a variety [FPV98] of message passing mechanisms, e.g. channels in Occam [PM87], or RMI in Java [DSMS98].
- A *thread* shares its name space with a group of threads and can thus synchronise through the shared data. Techniques such as semaphores, mutexes and monitors are used. For example Java threads may share a class of objects and communicate using synchronised methods. In discussing functional languages, it is convenient to further partition threads into two classes, depending on how functional they are:
  - *Pure threads* which are non-side-effecting and so perform no I/O.
  - *Impure threads* which may manipulate state, e.g. by performing I/O.

  In general a functional language must have at least one I/O thread to allow it to interact with the outside world. These I/O threads are equivalent to the threads in conventional languages.

Distributed languages can be classified depending on their support for threads and processes as follows.

- *Implicitly-parallel* languages have no explicit processes or threads but execute on multiple processors, e.g. HPF [For97], NESL [Ble96]. These languages are not suitable for distribution, but have parallelism implicit in control or data structures.
- *Shared name space* languages support threads, but not processes, e.g. OZ [HVS97], Oblique [CJK95], GpH [THLP98] and Linda [GC92].
- *Distributed name space* languages support processes but not threads, e.g. Facile [GMP89], PICT [PT97], and languages based on communications libraries like C with MPI [The97].
- *Shared and distributed name space* languages support both threads and processes. Threads may reside in a processes name space, as in Java [DSMS98], or may not, as in Occam [PM87].

Conventional distributed languages like Java or C-Split [DSMS98] provide high-level support for communication and synchronisation, e.g. remote procedure calls (RPCs) and synchronised methods. They typically have explicit, static task and data partitioning, although some recent languages now support dynamic task and data placement, for example Oz [HVS97]. Every value communicated between processes is sent explicitly, and must be fully evaluated prior to transmission. Hence values transmitted must be first order, i.e. functions and computations cannot be transmitted. Programs are non-deterministic and the programmer is responsible for avoiding problems of deadlock and starvation.

Distributed functional languages typically support a more dynamic approach. For example allowing dynamic data and task placement as in Kali Scheme [CJK95], and the dynamic generation of processes and channels, as in Eden [BLOM95]. They also support the transmission of higher-order values. Some languages even have deterministic semantics, like Brisk [HS98].

3 GpH and Concurrent Haskell

The sequential version of the non-strict functional language Haskell runs as a single I/O thread on one PE. The two well developed parallel extensions to sequential Haskell are Glasgow Parallel Haskell (GpH) and Concurrent Haskell. Where GpH is targeting transformational programming [Loo99], and Concurrent Haskell satisfies reactive systems.

3.1 GpH

GpH [THM+96] is a small extension to sequential Haskell that executes on multiple PEs. The program still has one main I/O thread, but may also consist of many pure threads which can be evaluating different sub-parts of the heap in parallel.

Pure Threads are advisory, i.e. they may or may not be created and scheduled depending on the parallel machine state. The `` `par` `` annotation is used to suggest that an expression may be evaluated in parallel with another by a new thread. These threads are then anonymous in that they cannot be manipulated by the programmer once created. This parallelism can be further coordinated by using `` `seq` `` to specific a sequence of evaluation - the first expression is evaluated before the second will be returned:

```
par     :: a -> b -> b
seq     :: a -> b -> b
```

Higher-level coordination of parallel computations is provided by abstracting over `` `par` `` and `` `seq` `` in lazy, higher-order, polymorphic functions, called *evaluation strategies* [THLP98].

3.2 Concurrent Haskell

Concurrent Haskell [PHA+97] adds several extension to sequential Haskell. The program consists of the one main I/O thread, but now the programmer has explicit control over the generation of more I/O threads and the communication between them.

I/O threads are created explicitly by the monadic command, `forkIO`. These new I/O threads are mandatory, i.e. they must be created and scheduled. Once created they can be addressed by *threadId* to further manipulate their operation:

```
forkIO        :: IO () -> IO ThreadId
myThreadId    :: IO ThreadId
```

Synchronisation & Communication. Implicit inter-thread synchronisation occurs within the shared heap, as threads block upon entering shared closures that are under evaluation by other threads. Explicit thread communication occurs within the monadic I/O system by the use of polymorphic semaphores - *MVar*. An MVar, created by `newEmptyMVar`, is a container which has a state of either empty or full. Using `takeMVar` returns and empties the container contents if it was full, otherwise it blocks that thread. A `putMVar` fills the container giving an error if it was already full. Multiple threads may share the MVar, in which case operations upon it may be non-deterministic:

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

These primitives can then be abstracted over to give buffers, FIFO channels, merging, etc.

Exceptions allow the flexible handling of exceptional, error, or unusual situations by changing the flow of control within a thread. Synchronous exceptions occur within a threads execution e.g. divide by zero. Asynchronous exceptions occur outside of the thread, somehow affecting it e.g. an interrupt generated when the user hits <ctrl> −C:

```
raiseInThread :: ThreadId -> Exception -> a
throw         :: Exception -> a
catchAllIO    :: IO a -> (Exception -> IO a) -> IO a
```

4 Design of GdH

GdH is intended to provide a high-level distributed programming model consisting of a hierarchy of threads - the explicitly placed I/O threads of Concurrent Haskell and the implicit pure threads of GpH. This model runs on multiple PEs using as far as possible GpH's automatic lazy and dynamic data sharing between the PEs. The relationship between GpH, Concurrent Haskell, and GdH is illustrated in Figure 1. The new language features for distributed programming primarily concern I/O threads, and are outlined below.

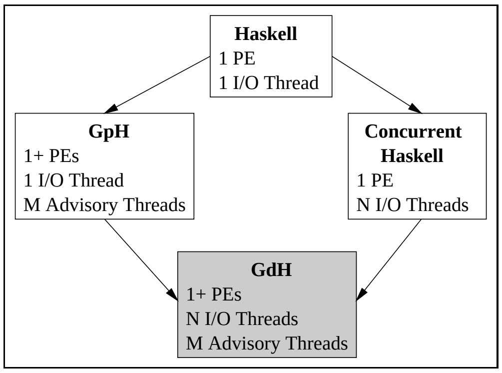

Fig. 1. The Haskell Family of Languages.

Processor Identification. PEs are made explicit in the language, and hence a program can explicitly utilise resources unique to a PE. This is essential for many distributed applications where some resources exist only on some PEs. For example, in a multi-user environment I/O must be attached to a particular machine to reach the right user.

In the same way that Concurrent Haskell allows the user to interact with threads via their ThreadId, GdH provides PEId to identify PEs, two functions are provided in the I/O monad to retrieve the PEId of the current PE, and to obtain a list of all the PEs currently participating in the program:

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

Figure 2 shows usage of `allPEId` to pick the first PE and later use it to specify the location of a thread. The use of `myPEId` is shown in Figure 3 where the PEId is later compared to that of the location of the MVar.

Remote I/O thread Creation. I/O threads are created remotely by `rforkIO`, analogous to Concurrent Haskell's `forkIO`. The `rforkIO` primitive takes an I/O thread, and a PEId and spawns that I/O thread on the specified PE. A blocking form of this function is `revalIO` which cause the calling thread to block until the remote thread finishes - effectively passing the thread temporarily to another PE:

```
rforkIO         :: IO () -> PEId -> IO ThreadId
revalIO         :: IO a -> PEId -> IO a
```

```
main =  do
        p:ps <- allPEId          -- get the 1st PE from the list of all
        m <- newEmptyMVar
        rforkIO (putMVar m 42) p -- create the remote process
        r <- takeMVar m
        putStrLn (show r)

-- Output: 42
```

Fig. 2. MVars and rforkIO

In Figure 2 a remote thread is created to manipulate an MVar and the main thread shows the MVars contents once it is updated.

Communication. Implicit synchronisation exists through the shared distributed heap, as in GpH. Also the `MVars` need to be extended for the distributed context to provide explicit synchronisation and communication between remote I/O threads. As with Concurrent Haskell higher-level constructs like channels and buffers are constructed by abstracting over distributed `MVars`, see section 6.2. The example in Figure 2 shows the use of an MVar where the main thread will block at the `takeMVar` until the remote thread has stored the value 42 within it.

Stickiness. Certain resources, e.g. files, are uniquely located on a particular PE and so only threads local to that PE can manipulate them. Other resources, e.g. MVars, require atomic operations to be performed on them. Stickiness embraces these concepts with the idea of a resource being stuck on a particular PE, sticky objects are thus members of the `Immobile` class so that we can then use `owningPE` to identify where it resides:

class *Immobile a* **where**
    `owningPE ::` $a -> IO\ PEId$

```
main =  do
        ps <- allPEId
        m  <- newEmptyMVar            -- create the MVar on the main PE
        let work = do
                i <- myPEId
                o <- owningPE m                -- where's the MVar?
                return (i,o)
        rs <- mapM (\p ->(revalIO work p)) ps -- map work across all PEs
        putStrLn (show rs)

-- Output: [(262215,262215),(524319,262215),(393218,262215)]
```

Fig. 3. Stickiness and owningPE.

That all PEs identify that a particular MVar exists on one PE is show in Figure 3. The program uses `mapM` to map the monadic operation of `revalIO work` across all the PEs obtained by `allPEId`. The `work` subroutine returns a pair showing which PE it executed on and where it believed the MVar was stuck.

Fault Tolerance. Rudimentary fault tolerance is provided by adapting Concurrent Haskell synchronous and asynchronous exceptions to the distributed context. This requires support for passing exceptions between PEs, and remote I/O threads.

```
main =  do
        ps <- allPEId
        let risky p = revalIO work p
            fails e = return (show e)          -- if we catch an exception
            work = do                          --  then return its name
                name <- getEnv "HOST"
                case name of                   -- the incomplete case
                  "bartok" -> return "Old Friend"
                  "ushas"  -> return "My PC"
        rs <- mapM (\p ->catchAllIO (risky p) fails) ps  -- wrap a handler around risky
        putStrLn (show rs)                               --  and map risky across the PEs

-- Output: ["My PC","Old Friend","Non-exhaustive patterns in case"]
```

Fig. 4. Remote Exceptions.

Figure 4 demonstrates the handling of an exception raised by failing of a pattern match. The primitive `catchAllIO` installs a handler `fail` for exceptions which could be raised remotely from the `revalIO` call. The exception when raised is passed seamlessly from remote to local thread in accordance with the idea that `revalIO` causes a thread to block and begin executing elsewhere as opposed to the reality of it creating a new thread.

5 Implementation

Haskell [PHA+97] is the *de facto* standard non-strict functional language and the Glasgow Haskell Compiler (GHC) is arguably the best non-strict Haskell implementation [PHH+93]. GHC implements Haskell'98 and adds, amongst many other extensions, support for concurrency and exceptions for Concurrent Haskell. Meanwhile the version based on the GUM RTS instead adds support for parallelism and a shared heap across multiple PEs for GpH. The distributed RTS (Sticky GUM) of GdH combines these two RTS varients as can be seen in Figure 5.

The design identified the need for the following extensions and care was taken in the implementation to reuse as much as possible of the existing mechanisms provided within the GUM RTS and to minimise the overall changes to the system.

Processor Identification requires providing calls to read the existing PE tables held in the RTS.

Remote I/O thread Creation. The `revalIO` operation introduced in the design, encapsulates both remote thread creation and the communication of results.

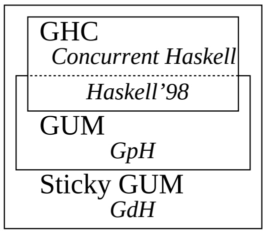

Fig. 5. The Haskell Family of Languages.

This general purpose nature suggests that it should form the primitive operation rather than `forkIO`.

A new message `REVAL` is defined which allows the creation of a mandatory remote thread to immediately evaluate and return a result. The original thread remains blocked until receiving the result message from the remote thread, see Figure 6.

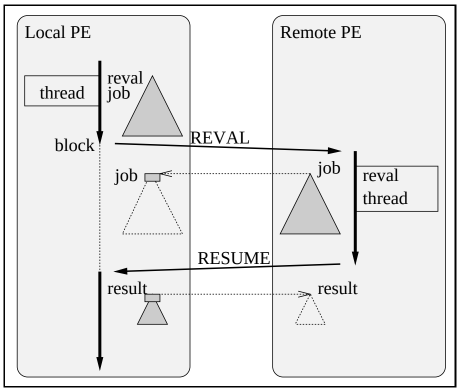

Fig. 6. RevalIO in Action.

This synchronous blocking mechanism used by the thread is the same as the GpH GUM mechanism which causes a thread to block when it enters a closure that is already being evaluated remotely.

Stickiness. Sticky objects are foreign objects, or stateful objects, i.e. closures that are mutated rather than entered and evaluated. For example the RTS automatically makes PEId, ThreadId and MVars sticky, and then creates remote references to them when these objects are shared with other PEs.

Within GpH, the notion of stickiness may also eventually lead to improved data locality for parallel computations, by providing a mechanism for clustering data together.

Communication to be explicit must occur through stateful, i.e. sticky objects. Once it is know where a sticky object exists then a remote thread can then be created at this location to do the required work. So by combining `revalIO` with `owningPE` it is possible to build a general purpose mechanism which performs MVar communication, and general sharing of all the foreign objects in the system. The alternative would have been to implement specific low-level messages for every type of inter-PE communication, which would greatly complicate the RTS.

Figure 7 shows how the low-level concepts of remote I/O thread creation and stickiness are presented to the Haskell programmer. Note that it is the RTS that forces particular primitive data types to be immobile. The language interface allows the programmer to make use of the RTS extensions, rather than altering the immobile behaviour of any new data types within the RTS.

Figure 8 shows how these concepts can easily be used to construct new distributed versions of the Concurrent Haskell primitives and so satisfy all the needs of communication.

```
data PEId
instance Eq PEId

-- choosing PEIds
myPEId          :: IO PEId
allPEId         :: IO [ PEId ]

-- Sticky things
class Immobile a where
  owningPE :: a -> IO PEId
  revalIO :: IO b -> a -> IO b

  revalIO job xx = do
        p <- owningPE xx
        doRevalIO job p
```

Fig. 7. GdH primitives.

```
-- instances of Sticky types
instance Immobile PEId
instance Immobile (MVar a)
instance Immobile ThreadId

-- new functions
rforkIO job p           = revalIO (forkIO job) p

killThread th           = revalIO (Concurrent.killThread th) th

takeMVar mv             = revalIO (Concurrent.takeMVar mv) mv

putMVar mv r            = revalIO (Concurrent.putMVar mv r) mv
```

Fig. 8. Extending Concurrent Haskell.

Fault Tolerance. Care has been taken to ensure that exceptions still behave correctly within `revalIO` code, as an unhandled remote exception needs to be passed back to the original thread for it to try and handle. This means that distributing a program by inserting an `revalIO` preserves its error handling properties.

The GdH initialisation code has also been extended to better handle (unexpected) termination, as well as to allow the user to add more PEs to a running program. These mechanisms have immediate benefit to GpH, which until now had no support for exceptions. The use of asynchronous exceptions may lead to an implementation of speculative evaluation: killing a thread, that has turned out to be redundant, involves the same mechanism of "freezing" computation as in raising an exception.

6 Demonstrators

We present two small demonstrator programs to give a flavour of GdH programming.

6.1 Ping

A very simple distributed program is our UNIX ping-like utility that gives an indication of PE to PE communication cost by timing the use of `revalIO` to evaluate a simple function remotely.

The code in Figure 9 obtains the list of available PEs using `allPEId`. Then `mapM` is used to map `loop` across the list of PEs. Within `loop` we use `timeit` to measure how long the `revalIO remote` operation takes on that PE.

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

-- Output: PEs = [262344,524389,786442,1048586,1310730]
--          Pinging 262344 ... at ushas time=0ms
--          Pinging 524389 ... at bartok time=3ms
--          Pinging 786442 ... at brahms time=3ms
--          Pinging 1048586 ... at selu time=2ms
--          Pinging 1310730 ... at kama time=2ms
```

Fig. 9. A GdH Ping Program.

The results are given for a group of linux x86 PCs on our local network. The first result is approximately zero since the work is being executed locally. GdH currently uses PVM for communication, and the times are comparable to the PVM timings program which returned round trip times of 1.1 - 2.7ms.

6.2 Co-operative Editor

A more sophisticated distributed program is a co-operative text editor that supports multiple text editor windows, a window on each machine, allowing them to communicate and share files. This also allows the sharing of files which are only accessable locally on a particular machine. An interface library in Haskell for the Tcl/Tk [Wel97] libraries, TclHaskell [SD99] is used to create multiple instances of Tcl/Tk running a simple text editor (TED). A new menu within the editors is used to manage the distributed interaction. The menu enables an editor instance to **send** its current buffer contents to all other editors, or for it to **fetch** messages sent to it from other editors.

The communication mechanism is a FIFO channel implemented via multiple MVars as provided in the standard libraries of Concurrent Haskell. The channels are used for two purposes:

- **Termination Control.** There is one global channel, named `fin` in Figure 10, and upon GUI quit or failure each GUI sends a message along this channel, which is then used by the startup thread to detect when the program has actually terminated. The auxilary functions `newWait`, `rforkWait`, and `untilWait` co-ordinate this behaviour, where `rforkWait` encapsulates the `rforkIO` and additional exception handling.
- **Data Exchange.** Each GUI has its own channel, which is a FIFO buffer. By reading or writing to each channel it is possible to co-ordinate the data exchange - however the data is actually transfered lazily, i.e. only when the receiving editor displays it.

Initialisation is shown in Figure 10, where all the channels, the list `ports`, are generated by the first `mapM`. It uses `reval` to ensure that all channels are created seperately on each PE for efficiency. Later the `pick` function chooses the appropriate channels for each editor instance. The second `mapM` is used create seperate instances of the TclHaskell GUI running the editor (`ted`) on each PE. Finally the termination control mechanism of `untilWait` causes the main thread to wait until all the GUIs have finished.

```
main = do
        pes <- allPEId
        putStr("PEs = "++(show pes)++"\n")
        fin <- newWait
        ports <- mapM (\p -> revalIO newChan p) pes
        let remote p = do
                let
                  comms = pick ports p
                  startGUI = do
                        name <- getEnv "HOST"
                        primPutEnv ("DISPLAY="++name++":0.0")
                        start $ (ted comms)
                rforkWait fin startGUI p
        mapM remote pes
        untilWait fin
```

Fig. 10. Initialisation of the Editor.

Buffer Communication is handled within the new menu in the editor. The menu's TclHaskell code is shown in Figure 11. It defines a new menu named `buffer` with three options `Fetch`, `Send`, and `List`, and associates appropriate functions with each option. The function `doFetch` handles the receiving of the data with `isEmptyChan` being used to check if any message exists, and if so then `readChan` extracts the first message. Each message consists of two strings: the name of the buffer and the buffer contents. The second function `doSend` appends the name of the host machine to the file name and then uses `writeChan` mapped across all the other channels to send it's buffer contents to all the other editors. The final function `doList` uses `snapChan` to take a snap-shot of the entire channel contents and then maps a function across it to list all the names of the messages in the buffer.

```
buffer_menu :: Context -> GUI ()
buffer_menu ctx@(Ctx w mp e rf) =
     do m <- menu w [tearoff False]
        cascade mp m [wgt_label "Buffer"]
        mbutton m [wgt_label "Fetch",     command doFetch]
        mbutton m [wgt_label "Send" ,     command doSend ]
        mbutton m [wgt_label "List" ,     command doList ]
        return ()
     where
        doFetch =
             do FES c _ _ _ <- readState rf
                empty <- proc $ (isEmptyChan c)
                if empty
                  then  return ()
                  else  do
                        (m,s) <- proc $ (readChan c)
                        resetEdit e s
                        change_fn ctx m
        doSend =
             do FES _ cs fn _ <- readState rf
                s <- getEdit e
                host <- proc $ (getEnv "HOST")
                let m = fn++"(from@"++host++")"
                proc $ (mapM (\c -> writeChan c (m,s)) cs)
                return ()
        doList =
             do FES c _ _ _ <- readState rf
                vs <- proc $ (snapChan c)
                let s = unlines ( map (\(m,_) -> m) vs)
                resetEdit e s
                change_fn ctx "(message list)"
```

Fig. 11. Buffer communication for the Editor.

In the screenshot Figure 12 the bottom two windows are instances of the editor, redirected via X to the same host, the other windows show PVM running and the console output.

7 Future Work

Some minor work is still required to make the GdH implementation more robust. Once this is complete we plan to consider the following issues:

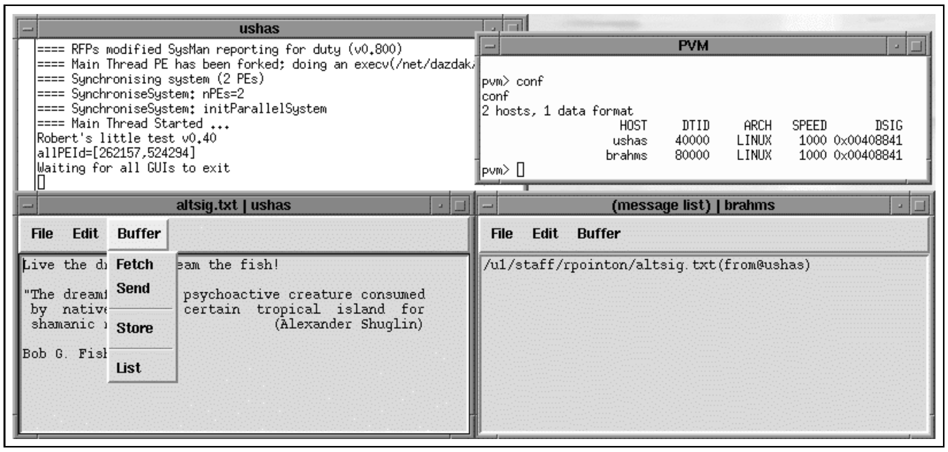

Fig. 12. Editor Screenshot.

- **Use and Evaluation of GdH.** We intend to evaluate GdH in comparison to conventional distributed languages like JAVA. In particular we will construct a GdH version of an existing distributed factory simulation [Kit92].
- **Cheaper Remote Evaluation.** Creating a remote thread for simple operations, e.g. to read an MVar, is rather costly. Instead this could be counteracted by the RTS checking the arguments to the `REVAL` message and then executing more efficient code inline for known simple cases.
- **Unique vs Immobile.** Sticky objects are unique in that only one copy of them exists, however only some of them have to be immobile. e.g. a file is physically immobile, whilst an MVar could freely migrate. The current design of having them immobile makes them easy to locate (via the remote references) however it results in a static resource placement which could interfer and restrict fault tolerance and program mobility.
- **Remote References.** The notion of a remote reference (and in fact stateful mutable objects) is an object-orientated (OO) concept. Such a concept is best implemented by executing the method *of* the object, as opposed to the current scheme of applying functions *to* the object. The current scheme requires the programmer to explicitly check if the object is a remote reference and then take the appropriate action, whereas the OO solution would handle the case automatically within the method. As it stands the RTS will regretfully fail if functions are applied directly to a remote reference.
- **Fault Tolerance.** Many computations are pure, and hence have no side-effects to be reversed during error recovery. The RTS can distinguishes between pure and impure computations: impure computations must be recovered using conventional exception-based techniques, but the RTS could attempt implicit recovery of pure computations. [TPL00]
- **Code Distribution.** Functional languages claim that code (closures) equals data, in the RTS this is true but not all the data is created equal. STG data

can be shared however the RTS lacks the ability to share the static info-tables and compiled code fragments. By making this data sharable using the existing communication routines, would open up the possibilities of mobile and hot-loadable code, similar in nature to what is found in ERLANG [Wik94], or Brisk [Spi99].

8 Discussion

The design objectives and concepts underlying the distributed functional language GdH have been presented. GdH provides explicit threads, with explicit mapping onto PEs. Communication between threads is achieved via virtual shared memory, implemented as a shared heap in our graph reduction machine. Special features of our language are the lazy and dynamic transfer of data between threads, and the dynamic configuration of the network of PEs.

The implementation of GdH is based on the existing extensions of Haskell for concurrency, Concurrent Haskell, and parallelism, GpH. The main modifications necessary to support the requirements of a distributed language affect the thread creation, which has to include placement information, and the communication between explicitly placed threads. By building our system on top of Concurrent Haskell and GpH we also gain a lot of mature functional language technology for free: sophisticated sequential code optimisations, foreign language interface, libraries for graphical user interfaces etc.

Our next steps in the development of GdH will be to test the system on larger applications, to both assess the fitness of the language and to have a more mature implementation of the language available.

References

[Ble96] G.E. Blelloch. Programming Parallel Algorithms. *Communication ACM*, 39(3):85–97, 1996.

[BLOM95] S. Breitinger, R. Loogen, and Y. Ortega-Mallen. Towards a Declarative Language for Parallel and Concurrent Programming. In *IFL'95*, Bastad, Sweden, Sep 1995.

[CJK95] H. Cejtin, S. Jagganathan, and R. Kelsey. Higher-order distributed objects. *ACM Trans. On Programming Languages and Systems (TOPLAS)*, 17(1), September 1995.

[DSMS98] M.C. Daconta, A. Saganich, E. Monk, and M. Snyder. *Java 1.2 and JavaScript for C and C++ Programmers.* John Wiley & Sons, New York, 1998.

[For97] High Performance Fortran Forum. *High Performance Fortran Language Specification*, January 1997. Version 2.0.

[FPV98] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[GC92] D. Gelernter and N. Carriero. Coordination Languages and Their Significance. *Communication ACM*, 32(2):97–107, February 1992.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. Facile: a symmetric integration of concurrent and functional programming. In Springer-Verlag LNCS 352, editor, *Tapsoft89*, pages 181–209, 1989.

[HS98] I. Holyer and E. Spiliopoulou. Concurrent Monadic Interfacing. In *IFL'98*, pages 253–269, UCL, London, UK, Sep 1998.

[HVS97] S. Haridi, P. Van Roy, and G. Smolka. An Overview of the Design of Distributed Oz. In *2nd Intl. Symposium on Parallel Symbolic Computation (PASCO 97)*, New York, USA, 1997.

[Kit92] P. Kiteck. Analysis of Component Interaction in a Distribution Facility using Simulation. In *EUROSIM*, 1992.

[Loo99] R. Loogen. Programming Language Constructs. In K. Hammond and G. Michaelson, editors, *Research Directions in Parallel Functional Programming*, pages 63–91. Springer, 1999.

[Mer96] L. Merrick. DCOM Technical Overview. Technical report, Microsoft White Paper, 1996.

[PHA+97] J. Peterson, K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fasel, A.D. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, E. Meijer, S.L. Peyton Jones, A. Reid, and P. Wadler. *Report on the Programming Language Haskell (Version 1.4)*, April 1997.

[PHH+93] S.L. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P.L. Wadler. The Glasgow Haskell Compiler: a technical overview. In *The UK Joint Framework for Information Technology*, pages 249–257, Keele, 1993.

[PM87] D. Pountain and D. May. *A Tutorial Introduction to Occam Programming*. McGraw-Hill, New York, 1987.

[PT97] B.C. Pierce and D.N. Turner. Pict: A Programming Language Based on the Pi-Calculus. Technical report, Indiana University, 1997.

[PV98] R. Plasmeijer and M. Van Eekelen. *Concurrent Clean - Language Report*. High Level Software Tools B.V. and University of Nijmegen, version 1.3 edition, 1998.

[SD99] M. Sage and C. Dornan. *TclHaskell - users manual*, Aug 1999.

[Sie97] J. Siegel. *CORBA Fundamentals and Programming*. John Wiley and Sons, New York, 1997.

[Spi99] E. Spiliopoulou. *Concurrent and Distributed Functional Systems*. PhD thesis, Department of Computer Science, University of Bristol, 1999.

[The97] The MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, July 1997.

[THLP98] P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[THM+96] P.W. Trinder, K. Hammond, J.S. Mattson, A.S. Partridge, and S.L. Peyton Jones. GUM: A portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation (PLDI'96)*, Philadelphia, USA, May 1996.

[TPL00] P.W. Trinder, R.F. Pointon, and H-W. Loidl. Runtime System Level Fault Tolerance for a Distributed Functional Language. In *SFP'00*, University of St Andrews, Scotland, July 2000.

[Wel97] B.B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, 2nd edition, 1997.

[Wik94] C. Wikstrom. Distributed programming in Erlang. In *PASCO'94 - First International Symposium on Parallel Symbolic Computation*, Linz, September 1994.