The Design and Implementation of GdH DRAFT*

R.F. Pointon‡ P.W. Trinder‡ H-W. Loidl‡

July 10, 2000

Abstract

This paper outlines the design and implementation of Glasgow Distributed Haskell (GdH). GdH combines the multiple processes of Concurrent Haskell with the multiple processing elements (PEs) of Parallel Haskell (GpH). It brings explicit PE identifiers to the language level thereby enabling the distribution of processes.

Data transfer is both lazy and dynamic: data is sent to a PE on request. The language is a minimal super-set of GpH and Concurrent Haskell, and thus maintains backwards compatibility. Implementation extensions of the GUM runtime system (RTS) are outlined. Finally two small demonstration programs are discussed.

1 Introduction

Many applications, particularly those with multiple users are most naturally structured as a collection of processes distributed over a number of machines. Examples include multi-user games and simulations, distance learning tools, and software development environments. The advent of the world wide web has stimulated the development of distributed applications, and the technology to support them [CJK95, DSMS98, GMP89, HRS97, PT97].

Large-scale distributed applications may have components written in multiple languages, supplied by several vendors and executed on a heterogeneous collection of platforms. They also have elaborate mechanisms for managing the failure of components of the system. Large-scale distribution is supported

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

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

by standard interfaces like CORBA [ea97] or Microsoft DCOM [Mer96], and entails considerable machinery. In contrast, small-scale distribution entails components written in a single language, is typically constructed by a single vendor, and is often restricted to an homogeneous 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]. Distributed languages allow the system to be developed in a single, homogeneous, framework, and make the distribution m ore transparent to the programmer. In consequence a distributed language is less cumbersome than a standard interface like CORBA.

Our aim is to extend the power of Haskell with support for small-scale distribution (a-la RMI in JAVA); in particular restricted to an homogeneous network of machines, with a simple model of failures.

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 [DD87], 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.

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 [HPF93], 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 [HRS97], 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 [DD87].

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 [HRS97]. 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 [HE98].

2.1 GpH and Concurrent Haskell

```
-- Forking and suchlike
data ThreadId
forkIO          :: IO () -> IO ThreadId
myThreadId      :: IO ThreadId
killThread      :: ThreadId -> IO ()
raiseInThread   :: ThreadId -> Exception -> IO ()
yield           :: IO ()

-- MVars
data MVar a
instance EQ (MVar a)
newEmptyMVar    :: IO (MVar a)
takeMVar        :: MVar a -> IO a
putMVar         :: MVar a -> a -> IO ()
isEmptyMVar     :: MVar a -> IO Bool
```

Figure 1: Basic Concurrent Haskell primitives.

GpH [THM+96] is a small extension to sequential Haskell. It supports multiple anonymous advisory threads of computation on multiple PEs, but has only a single process. Threads are introduced and synchronised using parallel and sequential composition operators, while the RTS handles the necessary communication and sharing of the heap across the PEs.

Concurrent Haskell [PHA+97] provides named explicit mandatory threads of execution, on a single PE, created by `forkIO` and referenced by *threadId*.

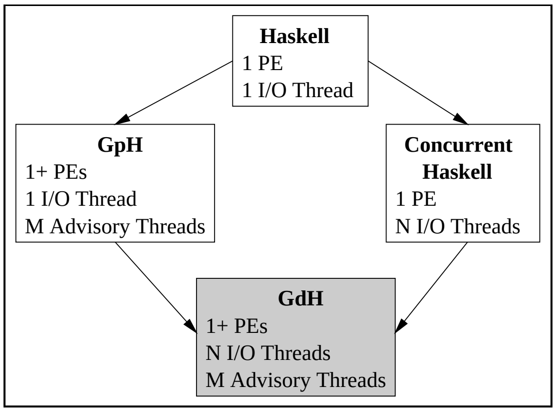

Figure 2: The Haskell Family of Languages.

Inter-thread communication is provided within a shared heap by a named explicit mechanism created by `newEmptyMVar` referenced by *MVar*. The MVar is a container which is either empty or full. A `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. `isEmptyMVar` returns a snap-shot of the full/empty state of the MVar. Multiple threads may share the MVar, in which case operations upon it may be non-deterministic. Figure 1 shows the basic interface to Concurrent Haskell as found in ghc-4.04.

3 Design

GdH combines the two branches of Haskell as shown in Figure 2.

In terms of its usage there is the notion of named PEs, therefore allowing the placement of mandatory threads via `rforkIO` which is an extension of Concurrent Haskell's `forkIO`. Primitives are therefore provided to choose and access the PEIds. Figure 3 details the extensions necessary.

```
data PEId
instance Eq PEId

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

-- remote forking
rforkIO         :: IO () -> PEId -> IO ThreadId
```

Figure 3: Basic GdH extensions.

Stickiness Certain resources, e.g. files, are uniquely located on a particular PE and require a thread on that PE to manipulate them. Other resources, e.g. MVars, require atomic operations to be performed on them. A general solution to this is the notion of making objects *sticky*, i.e. they can never be moved away from the current PE, so that attempts to communicate them create *remote references* which can then later be traced back to the original object. These sticky objects can then only be manipulated on their owning PE.

Remote Evaluation The communication necessary to perform explicit I/O operations is handled by introducing a remote procedure call, or in functional terminology, a remote evaluation `revalIO` operation.

The mechanism for creating a `REVAL` message and blocking until the result returns is handled by the `revalIO` function. This function also ensures that any exceptions raised remotely are passed back to the local thread to handle.

3.1 Features

The features of the design are:

- **Distributed I/O** through the ability to place threads on a specific PE and thus perform remote I/O, allowing the unique resources and attributes of each PE to be utilised. e.g. a music program could use the sound-cards of all the available PEs simultaneously - The Haskell Orchestra?

- **Lazy and Dynamic Transfer**

  In a non-strict distributed language data transfer between processes can be lazy and dynamic. The cost of laziness is an additional message from the recipient requesting the data, 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. In practice, to offset the high communications latencies of distributed systems transfer is at least partly eager. For example a packet-sized chunk of a data structure may be sent when the first element is demanded.
  - Lazy transfer 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 executions.

Each of these issues can of course be explicitly managed in a strict context, but explicit management is cumbersome and error-prone.

- **Dynamic PE Network.** New PEs can be added to the running program. In the longer term PEs will be able to leave gracefully as a consequence of error detection or user intervention.

- **Minimal Super-Set** GpH and Concurrent Haskell are combined with minimal extensions. The obvious extension being to that of Concurrent Haskell with the notion of remote thread creation:
  `rforkIO : IO () -> PEId -> IO ThreadId`

- **Backward Compatibility** with existing Haskell programs will be maintained including the centralised concepts of GpH having a distinguished *main* thread and therefore a main thread PE, likewise, Concurrent Haskell specifies program termination as occuring with the completion of the main thread.

4 Implementation

The GUM runtime system has been extended to:

- Handle distributed MVar communication.
- Provide a mechanism for creating remote threads.
- Provide mechanisms for coping with possible non-local I/O, e.g. if file handles are passed to remote threads.
- Provide an interface to the runtime system to access the local PE.
- Provide an interface to access the table of all PEs.

Sticky Objects 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.

Remote Evaluation A new message `REVAL` is defined which allows the creation of a mandatory remote thread to immediately evaluate and return a result. It is only upon receiving the result message that the original thread can continue. `REVAL(func)` $\rightarrow$ `RESUME(v)`

This blocking mechanism used by the thread is the same as the GUM mechanism which causes a thread to block when it enters a closure that is already being evaluated remotely. `FETCHME(clos)` $\rightarrow$ `RESUME(v)`

The actual decision of when to call `revalIO`, is a little more complex. In the cases when an MVar is being used to protect some PE resource then it makes sense to `revalIO` the whole protected piece of code, however this case is not easily automatically distinguishable at compile time (or runtime) from manipulating a single MVar to communicate a value.

To allow the programmer to decide when and where to call `revalIO`, the functions `owningPE` is provided to identify the original owning PE of a sticky object. Figure 4 shows how the concepts of remote evaluation and stickiness are to be presented to the Haskell programmer, Figure 5 shows how these concepts can easily be used to construct new distributed versions of the Concurrent Haskell primitives.

Note: Currently `_ccall_` is used to implement some extensions, in the future we hope to include the code into the compiler itself.

```
data PEId
instance Eq PEId


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

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

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

Figure 4: GdH primitives.

Exceptions and Errors Care has been taken to ensure that exceptions still behave correctly within `revalIO` code. The GpH initialisation code has been extended to better handle (unexpected) termination, as well as to allow the user to add more PEs to a running program.

```
-- instances of Sticky types
instance Immobile PEId where
  owningPE p = return p

instance Immobile (MVar a) where
  owningPE mv =  _ccall_  cGetCertainOwner mv

instance Immobile ThreadId where
  owningPE th = _ccall_  cGetCertainOwner th

-- 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
```

Figure 5: Extending Concurrent Haskell.

5 Demonstrators

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

5.1 Ping

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

Looking at Figure 6, we first obtain 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
        putStr("PEs = "++(show pes)++"\n")

        let
          remote = getEnv "HOST"

          loop pe = do
                putStr ("Pinging "++show pe++" ... ")

                (name,ms) <- timeit (revalIO remote pe)

                putStr ("at "++name++" time="++(show ms)++"ms\n")

        mapM loop pes
```

Figure 6: Core of the ping code.

Figure 7 gives an indication of the result for a group of PCs on our local network. Note: The first result is approximately zero as you might expect

```
PEs= [262153,524290,786434,1048579]
Pinging 262153 ... at ushas time=0ms
Pinging 524290 ... at kama time=4ms
Pinging 786434 ... at bartok time=6ms
Pinging 1048579 ... at brahms time=7ms
```

Figure 7: Ping Results.

since the work is being executed locally.

5.2 Co-operative Editor

A more sophisticated distributed program is a co-operative text editor that supports multiple text editor windows, possibly on remote machines, allowing them to communicate and share files. This also allows the sharing of files which are only accessable locally on a particular machine. TclHaskell [SD99] is used to create multiple instances of Tcl/Tk [Wel97] running a simple modification of the included TED editor. Each editor has an additional menu that enables the editor to **send** the current buffer contents to all other PEs, or for it to **fetch** messages sent to it from other PEs.

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: (for more detail see the source illustrated in Appendix A)

- **Termination Control** There is one global channel and upon GUI quit or failure each GUIs send a message along this channel, which is then used by the startup thread to detect when the program has actually terminated
- **Data Exchange** Each GUI has its own channel. By reading or writing to each channel it is possible to co-ordinate the data exchange - however the data is actually transfered later lazily when the GUI requests it.

In the screenshot Figure 8 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.

6 Future Work

Currently our implementation is functioning but with a few minor bugs. Once these have been addressed we plan to consider the following issues:

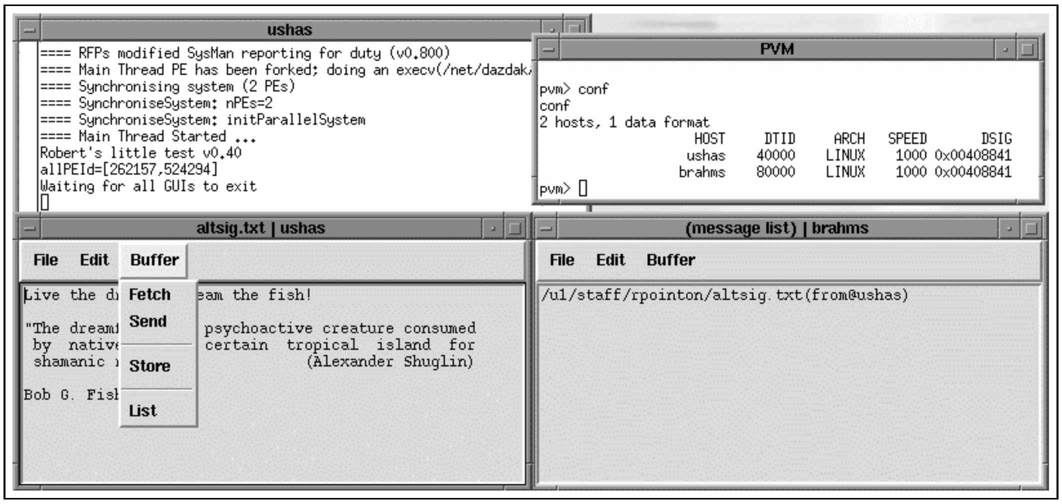

Figure 8: Editor Screenshot.

- **Use and Evaluation of** 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].

- **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 handle must be 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 interfers and restricts fault tolerance and program mobility.

  It may turn out that the distinction is based on foreign objects (+ I/O threads) vs stateful mutable objects.

- **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].

7 Discussion

The design objectives and concepts underlying GdH have been presented where essentially the language is a small extension to, and super-set of, GpH and Concurrent Haskell. The implementation has been outlined and two small programs written in the language have been shown together with their sample output.

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(5), Sept 1995.

[DD87] D.Pountain and D.May. *A Tutorial Introduction to Occam Programming.* McGraw-Hill, New York, 1987.

[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.

[ea97] J. Siegel et al. *CORBA Fundamentals and Programming.* John Wiley and Sons, New York, 1997.

[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.

[HE98] I. Holyer and E.Sipiopoulou. Concurrent Monadic Interfacing. In *IFL'98*, UCL, London, UK, Sep 1998.

[HPF93] High Performance Fortran Language Specification. Technical report, Rice University, Texas, USA, 1993.

[HRS97] 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.

[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.

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

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

Appendix A: Co-operative Editor Source

Initialisation (Figure 9) The functions `newWait`, `rforkWait`, and `untilWait` co-ordinate the termination, where the `rforkWait` does a remote thread fork and the `untilWait` waits until the required number of responses arrive as each `rforkWait` terminates.

The `ports` are the communication channels for the editors, `reval` is used to ensure that they are generated on each PE. Later on the `pick` function determines which ports each editor should use to send and receive from.

The TclHaskell GUI is started by the `start` function, which invokes the editor `ted`.

```
main = do
        pes <- allPEId
        putStr("PEs = "++(show pes)++"\n")

        fin <- newWait

        ports <- mapM (\p -> revalIO newChan p) pes

        let remote i = do
                let
                  pe = (pes!!i)
                  comms = pick i ports
                  startGUI = do
                        name <- getEnv "HOST"
                        primPutEnv ("DISPLAY="++name++":0.0")
                        start $ (ted comms)

                rforkWait i fin startGUI pe

        mapM remote [0..(length pes)-1]

        untilWait fin (length pes)
```

Figure 9: Initialisation of the Editor.

Buffer Communication (Figure 10) A new menu is defined. The option `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. To send the buffer `doSend` uses `writeChan` mapped across all the other channels. Finally to list all the message in our channel `doList` uses `snapChan` to take a snap-shot of the channel and extracts the name of the buffer stored in each message.

```
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,_) -> if null m then "(no name)" else m) vs)
                resetEdit e s
                change_fn ctx "(message list)"
```

Figure 10: Editor code additions necessary for communication.