The Design and Implementation of Glasgow distributed Haskell *

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 presents 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 Haskell'98 compliant, being a superset of both Concurrent Haskell and Glasgow parallel Haskell (GpH).
GdH distributes both pure and impure threads across multiple Processing Elements (PEs), each location is made explicit so a program can use resources unique to PE, and objects including threads can be created on a named PE. The location that uniquely owns a resource is identified by a method of a new `Immobile` type class. Impure threads communicate and synchronise explicitly to co-ordinate actions on the distributed state, but both pure and impure threads synchronise and communicate implicitly to share data. Limited support for fault tolerant programming is provided by distributed exception handling. The language constructs are illustrated by example, and two demonstration programs give a flavour of GdH programming.
Although many distributed functional languages have been designed, relatively few have robust implementations. The GdH implementation fuses and extends two mature implementation technologies: the GUM runtime system (RTS) of GpH and the RTS for Concurrent Haskell. The fused RTS is extended with a small number of primitives from which more sophisticated constructs can be constructed, and libraries are adapted to the distributed context.

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

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

another. Distributed architectures are more *scalable* 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 have elaborate failure mechanisms. In contrast, small-scale distributed programs entails components written in a single language, 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]. 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 Antigua [TLP+93], 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 also 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, in the anticipation of the following benefits that we seek to demonstrate in future work. Not all synchronisation and communication between threads need be explicit, in particular the shared graph model means that a thread has implicit (read-only) access to variables shared with other threads. Moreover, all data transfer between threads 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 is to be exchanged. Logically the entire data structure is exchanged, but the receiving thread 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 2 outlines distributed architecture and language concepts. Section 3 describes GpH and Concurrent Haskell. Section 4 presents the design of GdH, discussing the motivation. Section 5 describes the implementation of GdH. Section 6 presents and discusses two small GdH demonstration programs. Section 7 compares our approach to some other distributed functional languages before section 8 discusses future work and section 9 concludes.

2 Distributed Language Concepts

To provide a framework in which to present our design, we define both distributed architecture and language concepts. Distributed languages execute on multiple processors connected by some network. A *Processing Element* is a processor with associated resources such as memory, disk, and screen. A *Thread* is an independent stream of execution within a program. In functional languages we distinguish between *Pure threads* that have no side-effects, e.g. perform no I/O, and *Impure threads* which may manipulate state, e.g. by performing I/O. A *Process* is a set of threads executing a program and sharing a common address space and resources such as files.

A distributed language may or may not make architectural entities explicit, e.g. threads are explicit in many languages, but implicit in others. More often the language provides abstractions of the architecture level entities, e.g. naming a PE. There are several important concepts for distributed languages.

Locations. A location is a set of resources, e.g. files, memory, etc. A location is an abstraction of a PE and its resources. A process may be considered to be a location with threads. A language is *location independent* if locations are implicit. A language is *location aware* if locations are explicit, enabling the programmer to utilise the resources of a location, e.g. forking a new thread into a named location.

Communication and Synchronisation. Communication involves the exchange of data and synchronisation is the co-ordination of control between threads. The two are closely related as having one allows the implementation of the other. Non-determinism naturally results from communication when messages come from multiple threads. In languages with *implicit communication/synchronisation* threads typically communicate and synchronise using shared data, freeing the programmer from describing the communication/synchronisation. For example Java threads may share a class of objects and communicate using synchronised methods. In languages with *explicit communication/synchronisation* threads within a process typically communicate/synchronise using a shared resources of the location, e.g. a semaphore. If the threads belong to different processes then communication/synchronisation may either address the thread or some other common location like a channel or port.

Centralised/Decentralised. There is no reason why communicating threads must belong to the same program, and often large systems consist of multiple co-operating programs. *Centralised* languages are a single program, and this approach has the advantage that the program and the inter-thread communication/synchronisation can be statically typed. *Decentralised* languages allow multiple programs to interact using a predefined protocol, e.g. a client-server model. This requires some language support to initialise communication. Such languages support dynamic systems that can be extended by adding PEs and new programs. However, the communication between such a dynamic set of programs cannot be statically typed.

Fault Tolerance is the ability of a program to detect, recover and continue after encountering faults. Faults may either be internal to the program, e.g. divide by zero, or external, e.g. disk failure.

Distributed languages support explicit communication/synchronisation between multiple threads on multiple PEs. A distributed language may be centralised or decentralised, and typically provide some support for fault tolerance. Functional languages often attempt to relieve the programmer of the burden of managing distribution, that is, they often provide implicit communication/synchronisation, and a degree of location independence.

3 GpH and Concurrent Haskell

Haskell'98 is sequential and programs execute as a single impure thread, termed an *I/O Thread*, that is executed by one PE. Two well-developed parallel extensions to sequential Haskell are GpH and Concurrent Haskell. GpH targets parallel transformational programming [Loo99], i.e. the program takes some input, performs some parallel calculation and produces an output. In contrast Concurrent Haskell is aimed at reactive systems, i.e. the program constantly interacts with its environment, not necessarily terminating.

3.1 GpH

GpH [THM+96] is a small extension to Haskell'98 that executes on multiple PEs. The program still has one main I/O thread, but may introduce many pure threads to evaluate 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` function is used to suggest that an expression may be evaluated in parallel with another by a new thread. Pure threads are anonymous in that they cannot be manipulated by the programmer once created. Parallelism can be further co-ordinated using `seq` to specify a sequence of evaluation - that one expression is evaluated before another. Higher-level co-ordination 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 extensions to Haskell'98. The program consists of 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`. A new I/O thread is mandatory, i.e. it is created and must be scheduled by the RTS. Once created it can be addressed by its *threadId* to further manipulate their operation, e.g. to terminate it:

$$\begin{aligned}&\texttt{forkIO} &&:: \mathit{IO}\ () \rightarrow \mathit{IO\ ThreadId}\\ &\texttt{myThreadId} &&:: \mathit{IO\ ThreadId}\end{aligned}$$

Synchronisation and 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 synchronisation and communication occurs within the monadic I/O system by the use of polymorphic semaphores - *MVar*. An MVar is created by `newEmptyMVar`, and is a container that has a state of either empty or full. Using `takeMVar` returns and empties the container contents if it was full, otherwise it blocks the thread that is attempting to take. A `putMVar` fills the container giving an error if it was already full. Multiple threads may share an MVar, in which case operations on it may be non-deterministic:

$$\begin{aligned}&\texttt{newEmptyMVar} &&:: \mathit{IO}\ (\mathit{MVar}\ a)\\ &\texttt{takeMVar} &&:: \mathit{MVar}\ a \rightarrow \mathit{IO}\ a\\ &\texttt{putMVar} &&:: \mathit{MVar}\ a \rightarrow a \rightarrow \mathit{IO}\ ()\\ &\texttt{isEmptyMVar} &&:: \mathit{MVar}\ a \rightarrow \mathit{IO\ Bool}\end{aligned}$$

These primitives can then be abstracted over to give buffers, FIFO channels, merging, etc [PHA+97].

Fault Tolerance is supported by exceptions which allow the flexible handling of exceptional, or error 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:

$$\begin{aligned}&\texttt{raiseInThread} &&:: \mathit{ThreadId} \rightarrow \mathit{Exception} \rightarrow a\\ &\texttt{throw} &&:: \mathit{Exception} \rightarrow a\\ &\texttt{catchAllIO} &&:: \mathit{IO}\ a \rightarrow (\mathit{Exception} \rightarrow \mathit{IO}\ a) \rightarrow \mathit{IO}\ a\end{aligned}$$

4 Design of GdH

GdH provides the following facilities to support distributed programming:

- *Location Awareness* - through new language constructs enabling the manipulation of the specific resources at each location. For example to interact with the GUI of a specific user at a machine.
- *Explicit Synchronisation/Communication*- enabling the co-ordination of distributed impure threads. For example to synchronise multiple users who are sharing a resource like a gameboard.
- *Location Independence* - to maintain backward compatibility with Concurrent Haskell and GpH. This allows program behaviour to remain the same even when a program is distributed.
- *Fault Tolerance* - has some support in GdH so that robust programs may be constructed.
- *A Centralised* - approach is adopted, although future versions of GdH may be decentralised. A decentralised distributed Haskell is described in Section 7.

A more complete description of the design of GdH can be found in [Poi01].

4.1 Location Awareness

A GdH program executes at a set of locations. Each location is labelled with a value of a new abstract data type `PEId`. A `PEId` cannot be constructed by the programmer, instead it must be obtained by querying the program state. Locations are part of the global state of the program, therefore GdH primitives to interrogate this state must be monadic commands rather than pure functions.

A GdH program is centralised with a distinguished main location which represent where the program was started from, this location consists of the main I/O thread and provides the environment with `stdin`, `stdout`, etc.

To support location awareness the program must be able to obtain four vital pieces of information: Where is this location, where are other locations, where is an object located, and what is located here? Then to utilise the resources and attributes of a location it is necessary to have some primitive to specify a remote operation.

Where is this location? This is a request for the current location which is provided by the function `myPEId`:

$$\texttt{myPEId ::}\ \mathit{IO\ PEId}$$

Where are other locations? The set of all PEIds must be provided, which is returned conveniently as a list. In recognising that a GdH program will be a centralised program with a distinguished main location, the further decision is made that the head of the list returned by `allPEId` should always be the main location:

$$\texttt{allPEId ::}\ \mathit{IO\ [PEId]}$$

Where is an object located? Objects, i.e. items representing part of the world state such as: MVars, threads, files, etc, are made stationary at one location and only one copy of them ever exists. An object is constructed at a particular location because that is where the current thread is executing and often an object must be associated with a particular location because it is immobilised at that location, e.g. files and foreign objects.

A new Haskell class, `Immobile`, groups stateful objects together, and has a method `owningPE` which returns the location of the specified object:

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

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

What is located here? Each location has access to its local environment and existing Haskell interrogation commands, e.g. on files, environment variables, function appropriately.

A Remote Operation is executed by a remote impure I/O thread to manipulate state. Remote operation also allows the creation of process networks spanning multiple locations.

A new command, `revalIO`, is provided in the I/O monad. Calling `revalIO job p` causes the calling thread to block until the execution of `job` at the location `p` completes and returns a result, i.e. it has the effect of temporarily changing the location of the current thread. Conceptually `revalIO` is very similar to Java RMI:

$$\texttt{revalIO ::}\ \mathit{IO\ a} \rightarrow \mathit{PEId} \rightarrow \mathit{IO\ a}$$

The command `revalIO` represents the current thread temporarily changing location, and so preserves all location independent properties of that thread. This has an advantages for fault tolerance as the error handling capabilities of the thread are location independent, i.e. an exception raised in a thread that is within an `revalIO` will propagate back until an exception handler is found. The exception handler may be within the job at a remote location, or in the original thread.

Object placement can be accomplished by `revalIO`, this allows the creation of an object at a specific location, for example to create a distributed version of the Concurrent Haskell `forkIO` command that places a thread:

$$\texttt{rforkIO ::}\ \mathit{IO\ ()} \rightarrow \mathit{PEId} \rightarrow \mathit{IO\ ThreadId}$$

An example of usage of these location aware functions is shown in Figure 1. The program show that regardless of where a thread executes it consistently determines that a resource is at a particular location. The output of the program is a list of pairs showing thread location and resource location. Firstly the resource, an MVar, is created by `newEmptyMVar` and the list of of available locations is obtained via `allPEId`. A function, `work`, is defined that uses `myPEId` to determine the thread location and then utilises `owningPE` to determine the location of the MVar, the result is returned as a pair of PEIds. The monadic map operation, `mapM`, is used to call this new function, `work`, for each location. The result of the monadic map is a list of pair of PEIds that is shown by the `putStrLn`.

```
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. 1. Using the location aware primitives.

4.2 Location Independence

In a functional language the majority of the data is non-mutable, some data represents suspended computation and thus is mutable once, while the remainder of the data is objects that are mutable many times. When data is distributed over multiple locations then some form of synchronisation is necessary so that the data remains consistent.

Suspended computations synchronise implicitly using the same approach as GpH and Concurrent Haskell, where the first thread that enters the suspension will perform the evaluation and other threads that enter then block until the evaluation completes.

Location independence of objects requires that the language supports some means for manipulating an object regardless of its location. The design decision was made to rewrite the relevant libraries, i.e. for MVars, threads, etc, to encapsulate and hide the location dependent properties. To do this the mechanisms of location awareness can be used by allowing a remote operation via `revalIO` on the object once its location has been determined by `owningPE`.

4.3 Fault Tolerance

Exceptions are a useful construct for supporting fault tolerance. The synchronous and asynchronous exceptions supported by Concurrent Haskell are extended so that they operate in a location independent manner, that is, exceptions can be raised at one location and handled at a remote location.

The only source of synchronous exceptions between locations in GdH occurs within the `revalIO` mechanism when a thread has temporarily migrated to another location as discussed earlier. Asynchronous exceptions between locations are possible when a `raiseInThread` function is applied to a thread which resides at a remote location. The semantics are extended for location independent operation, but no new language constructs are required.

New exceptions may be raised in response to a failure of a PE or connection to a PE. The detection of this class of errors is difficult as often the system cannot distinguish between the loss of a PE, or the loss of a message, from a message being delayed. The handling of these errors is also problematic as the loss of any part of the virtual shared heap can result in dangling heap references and therefore corrupt data and code. Handling these errors is critical for the construction of robust system and we have made an initial study [TPL00] but not yet implemented our design.

5 Implementation

To implement GdH the following steps were necessary. The GpH and Concurrent Haskell runtime systems had to be merged into a new RTS. The new language primitives (`myPEId`, `allPEId`, `owningPE`, and `revalIO`) for location awareness need to be provided in a new Haskell module for the programmer, and require

	Haskell'98	Concurrent Haskell	GpH	GdH
PEs	One	One	Many	Many
Location	N/A	N/A	Independent	Aware
Centralisation	N/A	N/A	Centralised	Centralised
Threads	One	Many impure	Many pure	Many pure & impure
Communication	N/A	Implicit & explicit	Implicit	Implicit & explicit
Fault Tolerance	None	Exceptions	None	Exceptions

Fig. 2. The relationship between Haskell'98, the extensions GpH and Concurrent Haskell, and the new extension GdH.

implementation in the merged RTS. Finally the existing libraries need to be extended to operate safely in a location independent manner. Each of these steps is discussed below, and more detailed information may be found in [Poi01].

5.1 Merge Runtime Systems

GdH fuses and extends the runtime systems used by the Glasgow Haskell Compiler (GHC) [PHH+93]. GHC supports not only Haskell'98 but the extensions for Concurrent Haskell and GpH. The standard GHC RTS has extensions for concurrency and exceptions for use in Concurrent Haskell. GpH is implemented by using a second RTS, GUM, that uses PVM for low-level location communication, adds support for parallelism and a virtual shared heap.

The GdH RTS, Sticky GUM [1], is an extended fusion of GUM and the GHC RTS. The two original runtime systems share significant amounts of code, yet were not primarily designed to coexist. The overlap of the different RTS extensions and the languages they support can be seen in Figure 3. One of the major differences in the GHC RTS and GUM, is what requires synchronisation: the GHC RTS requires additional synchronisation for the implementation of MVars, thread delays, and exceptions; where GUM requires additional synchronisation for the portions of the heap that are shared across multiple locations.

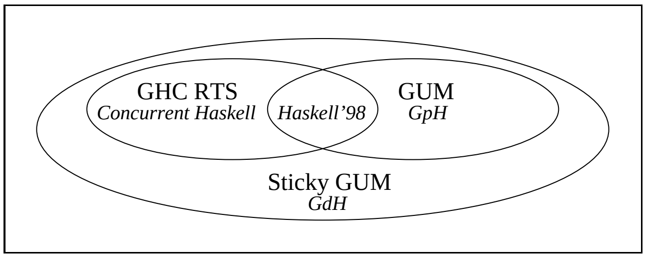

Fig. 3. Relationship of each Haskell extensions and RTS.

[1] The RTS is *sticky* in that immobile objects adhere to a particular location.

5.2 New Language Primitives

The language design identified the following new language primitives: `myPEId`, `allPEId`, `owningPE`, and `revalIO`. These primitives form part of the new Haskell module *Distributed* shown in Figure 4.

```
data PEId                             -- abstract

myPEId     :: IO PEId                 -- the current location
allPEId    :: IO [PEId]               -- list of all locations

class Immobile a where
  owningPE :: a -> IO PEId            -- the location of an object
  revalIO  :: IO b -> a -> IO b       -- remote evaluation to an object
  revalIO job xx = do
        p <- owningPE xx
        doRevalIO job p

instance Immobile PEId                -- a location is immobile
```

Fig. 4. Interface to the new Haskell module *Distributed*.

The current location is returned by `myPEId`. This result corresponds directly to the unique `taskId` from the underlying PVM system. It is straightforward to call a C function, through the Haskell foreign language interface via a `callc`, to fetch this value.

The list of available locations is returned by `allPEId`. GUM already stores a table of all locations for use by the garbage collector and global addressing subsystems. The `allPEId` function accesses the RTS via a `callc` to build a list from this table.

The `Immobile` class contains both `owningPE` and `revalIO` as with practical usage of the language it quickly became apparent that one of the most common uses of the result of `owningPE` is to then immediately call `revalIO`, thus sending a specific piece of work to where the resource is located. Therefore `revalIO` can now operate on any member of the `Immobile` class, automatically calling `owningPE` and then the underlying primitive `doRevalIO` which uses values of type PEId only. If the programmer still wishes to reference the location of an object explicitly then they may do so since PEId is a member of the Immobile class.

Implementing Immobile Objects. Sticky GUM maintains information about which types are immobile and the Haskell programmer cannot alter this. When immobile objects are communicated between locations by the RTS, they are first converted into a new closure type, `REMOTEREF`. Hence there only ever exists one copy of the object and multiple `REMOTEREF` closures that refer back to it. The RTS assigns a unique global address to closures communicated between locations. The `REMOTEREF` has a global address which includes the original PEId that is accessed by `owningPE` via a `callc`.

Implementing Remote Evaluation. A new PVM message `REVAL` is defined that carries the information necessary for the generation of the remote thread. When the remote location receives an `REVAL`, it creates a mandatory thread and immediately begins executing it. Upon termination of this remote thread, the result is sent back to the original thread via the existing GUM RESUME message, as depicted in Figure 5.

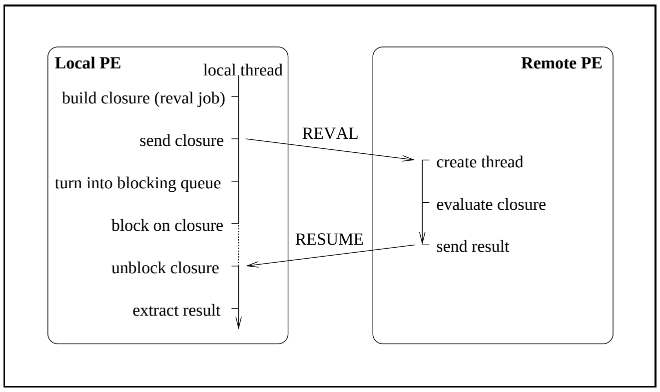

Fig. 5. The remote evaluation process.

The execution of a doRevalIO is outlined in Figure 6, and proceeds as follows. It tests if the destination location is the same as the current location and if so then performs the optimisation of doing the work locally. For the cases involving a remote location, an exception handler is installed around the piece of work, job, which is wrapped up in a constructed type, Status, so that a valid result value is always returned. The primitive unsafePerformIO is used to write the result to a single closure, result, this closure is where the synchronisation of waiting for the remote result takes place.

The synchronisation mechanism used by Sticky GUM to block the original thread while the remote evaluation takes place is almost identical to that used in GUM. In Figure 5 the root of the work to be sent is a closure, so when the doRevalIO function calls a callc to create the REVAL message it also changes the closure into a blocking queue that has an initial state of blocked. A blocking queue is used to synchronise activities such as waiting on an MVar, or for evaluation to complete.

Finally doRevalIO enters the closure in a case statement to check if the result should raise an exception or not. By entering the closure it automatically blocks the local thread until the RESUME message arrives with the result.

5.3 Extend Existing Libraries

Many of the libraries need to be made location aware. The design identified that many useful constructs could be built from the four distribution primitives and much of the implementation to do so required rewriting existing libraries rather than providing totally new constructs. There are two types of extensions needed,

```
data Status a = Okay a | Fail Exception

doRevalIO :: IO a -> PEId -> IO a
doRevalIO job p = do
  i <- myPEId
  if i==p
    then job                       -- do it locally if you can.
    else do
      _ccall_ cRevalIO result p  -- send the work off.
      case result of             -- check the result.
        Okay r  -> return r
        Fail e  -> throw e
  where
    tryjob = do                    -- construct an 'Okay' result.
      r <- job
      return (Okay r)
    caughtjob = catchAllIO tryjob (\e -> return (Fail e))
    result = unsafePerformIO caughtjob
```

Fig. 6. Blocking and error handling of `doRevalIO`.

the first to provide a location independent means for accessing objects, and the second to provide a method for specifying object placement.

To illustrate the process we provide an example for threads. First `threadId` is made an instance of the `Immobile` class so that location independent access routines can be defined, i.e. for `killThread` and `raiseInThread`. Finally new object placement functions can be added, i.e. `rforkIO`. Some functions may require no change, for example `forkIO`, `Eq`, and `myThreadId`:

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

6 Demonstrators

We present two small demonstration programs to illustrate the new constructs provided by GdH; these programs make almost exclusive use of explicit communication. In contrast to these examples we intend GdH to be used for larger applications where the majority of the communication can be implicit, e.g. a game where multiple player can implicitly share a large environment, dynamically fetching the parts of the environment on demand.

6.1 Ping

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

The code in Figure 7 obtains the list of available locations using `allPEId`. Then `mapM` is used to map `loop` across the locations. Within `loop` we use `timeit` to measure how long the `revalIO remote` operation takes on that location.

```
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. 7. 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 use PVM for communication and the times returned 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, on different machines, allowing users to communicate through them and share files. Such an editor allows the sharing of files that are only accessable locally on a particular machine. An interface library in Haskell for the Tcl/Tk 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 to **send** its current buffer contents to all other editors, or for it to **fetch** messages sent to it from any other editor.

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 8, 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 every GUI thread in the program has terminated. The auxiliary 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 between editors. Note, however, the data is transferred lazily, i.e. only when the receiving editor displays it.

Initialisation is shown in Figure 8, where all the channels, the list `ports`, are generated by the first `mapM`. It uses `reval` to ensure that all channels are created separately on each location for efficiency. Later the `pick` function chooses the appropriate channels for each editor instance. The second `mapM` is used create separate instances of the TclHaskell GUI running the editor (`ted`) on each location. 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 startGUI = do
                        name <- getEnv "HOST"
                        primPutEnv ("DISPLAY="++name++":0.0")
                        start $ (ted (pick ports p) )
                rforkWait fin startGUI p
        mapM remote pes
        untilWait fin
```

Fig. 8. Initialisation of the editor.

Buffer Communication is handled within the new menu in the editor. The menu's TclHaskell code is shown in Figure 9. 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.

In the screenshot Figure 10 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 Related Work

Conventional distributed languages like Java or C-Split [DSMS98] provide high-level support for communication and synchronisation, e.g. remote procedure calls 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 the object-orientated, functional, constraint based language Oz [HVS97]. Every value communicated between processes is sent explicitly, and in many languages must be fully evaluated prior to transmission. Hence values transmitted must be first order, i.e. functions and infinite data structures cannot be transmitted. Programs are non-deterministic and the programmer is responsible for avoiding problems of deadlock and starvation.

```
buffer_menu :: Context -> GUI ()
buffer_menu ctx@(Ctx w mp e rf) =
     do m <- menu w [tearoff False]                          -- define the menu layout
        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 ]
     where
        doFetch =
             do FES c _ _ _ <- readState rf                   -- get input channel
                empty <- proc $ (isEmptyChan c)
                if empty
                  then return ()
                  else do (f,s) <- proc $ (readChan c)
                          resetEdit e s                       -- show text in the editor
                          change_fn ctx f                     -- show source as filename
        doSend =
             do FES _ cs fn _ <- readState rf                 -- get output chans&filename
                s <- getEdit e                                -- get text in editor
                host <- proc $ (getEnv "HOST")
                proc $ (mapM (\c -> writeChan c ((fn++"(from@"++host++")"),s)) cs)
        doList =
             do FES c _ _ _ <- readState rf                   -- get input channel
                vs <- proc $ (snapChan c)
                resetEdit e (unlines (map (\(f,_) -> f) vs)) -- show text in the editor
                change_fn ctx "(message list)"                -- show "(m..." as  filename
```

Fig. 9. Buffer communication for the editor.

Like other distributed functional languages, GdH supports a more dynamic approach to distribution than conventional distributed languages. For example in GdH pure and I/O threads can be dynamically created, with pure threads dynamically allocated to PEs. Also data is dynamically communicated between locations on demand and it allows the transmission of higher order values. Like other distributed functional languages, GdH supports more implicit distribution, for example threads communicate and synchronise implicitly on shared data values. Using I/O threads a GdH programmer can construct programs equally as expressive as conventional distributed languages, including programs that are non-deterministic or deadlock.

While many distributed languages opt for the flexibility of a decentralised approach, GdH's centralised approach makes verification easier: all of the interacting threads are part of a single program and both threads and communication can be statically typed. In the former approach, there is no common analysis of the co-operating programs.

There have been many distributed functional language designs and implementations, e.g. Facile Antigua [TLP+93], Goffin Distributed Haskell [CGK98], OZ [HVS97], Kali Scheme [CJK95], Concurrent Clean [PV98], and Pict [PT97]. Some are only paper designs, and others have only been supported by short-lived implementations. There are relatively few robust long-lived implementations, and we discuss some of the more recent and most closely-related to GdH.

ERLANG represents arguably one of the most successful distributed functional languages so far. It was developed specifically in the telecommunications industry for writing concurrent, soft real-time, distributed fault tolerant systems [Wik94] and has had considerable success in this area.In contrast to GdH, Erlang systems are decentralised. As a simple strict impure functional lan-

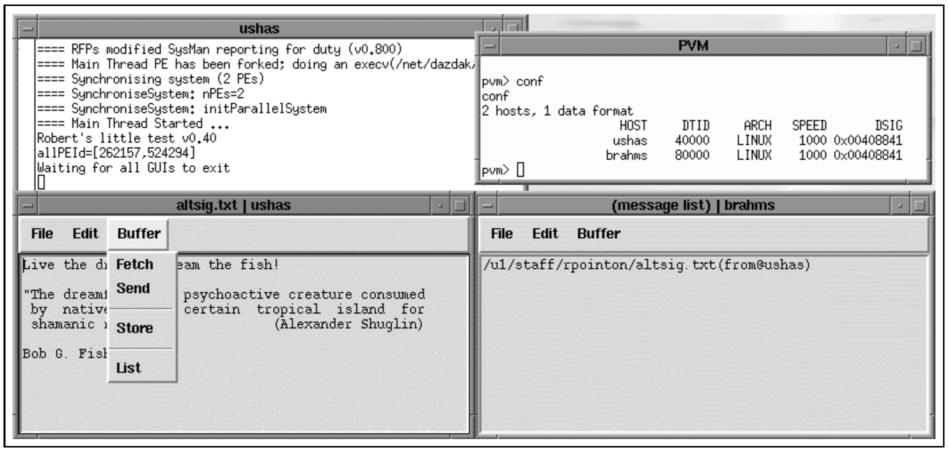

Fig. 10. Editor screenshot.

guage, ERLANG omits many features of modern functional languages. Compared to GdH it omits: currying, higher-order functions and lazy evaluation. More importantly, coming from a logic programming background it is untyped which allows many programmer errors to go unnoticed at compile time. ERLANG supports a number of extremely useful features, especially useful for large complex applications, which are not available in GdH. These include hot loading of new code into running applications, explicit time manipulation to support soft real time systems, and message authentication.

Haskell with Ports [HN00] is a new library for Haskell that adds the benefits of ERLANG style communication using ports, thus allowing inter-process communication and fault tolerance. Use of the library allows decentralised systems to be constructed that communicate with each other over ports using a predefined protocol with dynamic typing of the communication. Unlike GdH it only allows explicit communication and only first order values (including ports) to be transmitted through the ports. Communication of higher-order values is mentioned as possible future work.

Brisk [Spi99] is a derivative of Haskell (currently partially implemented) which makes use of lazy evaluation to give deterministic concurrencyin a multiple demand driven approach. A deterministic form of communications based on merging with hierarchical timestamps is also introduced to extend the expressiveness of the basic deterministic concurrency. Other useful features include the sharing of binary code between machines. Compared to GdH, Brisk uses a more powerful *pure* functional approach without resorting to monadic style I/O, yet is closest in terms of the implicit and lazy communication. Brisk's deterministic concurrency model is much more restrictive and prevents the expressing of many inherently non-deterministic programs, e.g. the dining philosophers problem.

8 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:

Use and Evaluation of GdH - in comparison to conventional distributed languages like Java. In particular we have constructed a GdH version of an existing distributed factory simulation [Kit92], and are constructing a larger application - a multiuser game with map navigation that utilises implicit communication.

Fault Tolerance. - 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].

Decentralised - systems would require GdH to provide connection/disconnection language constructs. Sticky GUM would require further extensions to allow dynamic typing of communication and a robust virtual shared heap that allows PEs and their heap to disconnect.

9 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 implicit communication of, and synchronisation on, shared data, and the lazy dynamic communication of the data between locations.

The implementation of GdH combines two mature runtime systems, and adds a small set of new primitives. The main modifications necessary to support the requirements of a distributed language affect remote thread creation, and the treatment of immobile objects. By basing our system on GHC we utilise mature compiler technology including: sophisticated sequential code optimisations, a foreign language interface, libraries for graphical user interfaces, etc.

We plan to evaluate GdH on larger examples and to make it freely and publicly available as part of the GHC distribution.

References

[CGK98] M. Chakravarty, Y. Guo, and M. Khler. Distributed Haskell: Goffin on the Internet. In *Third Fuji International Symposium on Functional and Logic Programming*, pages 80–97, 1998.

[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.

[HN00] F. Huch and U. Norbisrath. Distributed Programming in Haskell with Ports. In *12th International Workshop on Implementation of Functional Languages (IFL'00)*, Aachen, Germany, September 2000.

[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. Peyton Jones, A. Reid, and P. Wadler. *Report on the Programming Language Haskell (Version 1.4)*, April 1997.

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

[Poi01] R. Pointon. *The Design, Implementation, and Use of a Distributed Functional Language*. PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 2001. In Preparation.

[PT97] B. Pierce and D. 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.

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

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

[TLP+93] B. Thomsen, L. Leth, S. Prasad, T-M. Kuo, A. Krammer, F. Knabe, and A. Giacalone. Facile Antigua Release Programming Guide. Technical report, European Computer-Industry Centre , Germany, 1993.

[TPL00] P. Trinder, R. Pointon, and H-W. Loidl. Runtime System Level Fault Tolerance for a Distributed Functional Language. In *2nd Scotish Function Programming Workshop (SFP'00)*, University of St Andrews, Scotland, July 2000.

[Wik94] C. Wikstrom. Distributed programming in Erlang. In *1st International Symposium on Parallel Symbolic Computation (PASCO'94)*, Linz, September 1994.