Distributed Programming in Haskell: From Ports to Streams

Volker Stolz and Frank Huch*

Lehrstuhl für Informatik II
RWTH Aachen
D-52056 Aachen, Germany

Abstract. We present an extension of the lazy functional programming language Haskell for distributed programming. For the communication between processes we modify the port concept to a stream based approach. Ports are accessible over the Internet and behave like channels in Concurrent Haskell except that only the process which creates a port can read from it. This restriction is not enforceable at compile time and thus may yield undesired effects at runtime. By eliminating the need for explicit `readPort` statements and instead providing a stream of the items received on a port, we can guarantee safety in regard to this matter. The implementation consists of a library written entirely in HASKELL which provides functions for creating new processes, communication between concurrent and distributed processes in an open system using ports or streams, and error handling with exceptions.

1 Distributed Programming

The development of software systems has changed in the last years. Many systems are distributed, because of the following reasons:

- **Parallelization:** Resources (e.g. speed or space) needed for an application are not sufficing on one computer.
- **Inherent distributed character:** The application itself is distributed. Examples are (mobile) telephones and a cash dispenser together with the bank server.
- **Reliability and fault tolerance:** To increase the reliability of a system it is possible to arrange for several computers to cooperate such that the failure of one or more computers does not effect the system behavior as a whole.
- **Access to special resources:** In a heterogeneous network, special resources, e.g. a scanner or printer can only be accessed from one computer.

With the boom of networks and the Internet, the number of distributed applications increases. In particular more and more applications have an inherent distributed character. To provide convenient means for programming, modern languages must support distributed programming. It is not sufficient to provide a library for communication via sockets.

The language has to supply high level concepts for distribution and communication between processes. We want to extend the functional programming language Haskell [8] with features for elegant distributed programming.

* {stolz,huch}@i2.informatik.rwth-aachen.de

2 Distributed Communication

Modern languages need to provide suitable ways for communication in a distributed system. Different methods like *remote procedure call* or *message passing* are employed. Standards like CORBA [10] define methods for interchanging information between different platforms.

Functional languages have been extended for concurrent and distributed programming, too. The most successful one is Erlang [1]. Unlike Haskell, Erlang is an eager functional programming language which is extended with special features for concurrent and distributed programming. Processes can be created dynamically with `spawn` on a local or even a remote computer. Every process has a process identifier (*pid*) which is used as a reference for the communication between processes. Other processes can send messages to this pid.

Arriving messages are stored in a per-process mailbox in order of their arrival (FIFO). The receiving process can conveniently access this mailbox with pattern matching, so it does not need to extract messages in their chronological order. Only relevant messages are fetched with pattern matching while the others reside in the mailbox and can be processed later.

Another important feature of Erlang is that communication between concurrent processes on the same computer is not distinguished from communication to remote processes in a network. Therefore a system developed in a concurrent setting can later be distributed easily. Scalability of the system is supported by the language.

For fault tolerant programming Erlang provides a linking mechanism. Processes of the system can automatically be informed if others die or become unreachable, for example because a computer crashes. Hence these processes can react on the failure and reorganize the system to a consistent state.

Porting this communication scheme from Erlang to Haskell proved infeasible ([6] & [13]) because of limitations in Haskell's type system. Subtyping is necessary because different processes can understand different messages, but have common subsets. Unluckily, this can't be expressed in Haskell where constructors must be unique.

Let's assess the currently available features on which a more comfortable programming environment can be based: Concurrent Haskell [9] is state of the art for concurrent programming in Haskell. It provides functions to start and terminate threads dynamically inside an application and to synchronize them with mutable variables (*MVar*). This requires leaving the purely functional context and entering the IO monad which is used for conventional input/output.

On top of these MVars semaphores and channels provide more comfortable means of message passing. But there is yet no concept for distributed programming. One possible approach is the extension of Concurrent Haskell with communication via channels and MVars for distributed programming. But this leads to implementation problems. In Concurrent Haskell many processes may synchronize on a mutable variable or a channel. Distributing some of these processes in a network leads to synchronization problems, because it is not clear where a MVar is located. Consider the situation in Figure 1. Two processes can write a mutable variable and two other processes want to read it. In a distributed setting these processes could be located on four different computers in a network. But

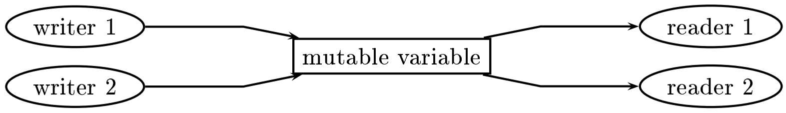

Figure 1: Distribution of Concurrent Haskell Processes

where can the mutable variable be located? It has to be located on one computer in the network, because it needs a state for the storage of a value, if no reader suspends on it and no reader wants to read the value. The possibilities for the location of the MVar are one of the four computers of the example or an independent computer. But all these locations have disadvantages for providing fault tolerance. It is necessary that parts of the system may terminate or even crash without effecting the rest of the system. If the computer the mutable variable is located on crashes, then the whole system cannot work anymore, although there is still a writer and a reader which could communicate with each other from the logical structure of the system. The readers and the other writers hang up and it is difficult to repair the system to a consistent state where the other components can communicate with each other again.

Another problem is garbage collection in a distributed system. We do not know when a mutable variable or channel is garbage. Reference counting is not possible because it would involve a large network overhead. So it will be almost impossible to check if a MVar is still known somewhere in the network and an algorithm would be very expensive and produce much communication in the network. Especially in case that some parts of the system have already crashed, some kind of polling **all** nodes which have a registered reference on the MVar would be necessary.

Systems like Glasgow parallel Haskell [14] or Glasgow distributed Haskell [11] implement distributed garbage collection, but are limited to closed systems. It is also not possible to implement fault tolerance in these systems, which is in our view a major requirement for a programming language for distributed system.

Our solution to the problems arising from distributed garbage collection is to restrict communication to only one reader for each channel. We can thus place the "physical" object onto the machine/processor which is reading it.

3 Distributed Haskell

In the discussion of distributing objects similar to Concurrent Haskell in a network, we have seen which problems appear with multiple readers and writers of a MVar. Our solution to this problem is a restriction to only one reader. With this restriction we can locate the MVar at the same place where the reader is located. If the reader terminates or crashes, the MVar terminates, too. No other readers can suspend on it. Therefore no processes are hanging, although there may still be references to the crashed resource on remote hosts. There can still exist writers that want to write to the MVar. Hence they are also in an inconsistent state. But they can recognize this the next time they send a message to the MVar. A failed write operation can throw an exception, as can the runtime system when we provide means of checking the lifeness of MVars although they are currently unused. The exception can be caught and the writer can initialize a reorganization of the crashed components, for example on another computer, or just fail gracefully.

Although this communication concept is different from Erlang, there are still similarities: Erlang also has a single-reader restriction as each mailbox is associated with a single process which is the only one who can read from it.

In the following, we will allow multiple mailboxes called *ports* (similar to the channels of Concurrent Haskell as programs usually will work an sequences of messages instead of just single values passed on using a MVar), associated to exactly **one** process which may read the data. Of course each port may have a different type. The reader is naturally the process which creates the port, although ownership can be transferred.

However, nobody precludes the programmer from passing the port via `forkIO` to another thread or reading from it on a remote machine. As a remedy we must insert checks at runtime to see if a thread is permitted to read from the port. If we can eliminate the need for explicit reading from the port we could help the programmer in avoiding mistakes. Before we show our approach to this matter, let's give a quick overview of the current API provided for Port-based Distributed Haskell:

3.1 The Distributed Haskell Library

Ports are represented as an abstract polymorphic data type

```
data Port a -- abstract
```

where the type variable `a` represents the type of the values that can be sent to the port. A new port can be created with the function

```
newPort :: IO (Port a)
```

Like in Concurrent Haskell the operations for creating and sending have side-effects because they change the environment the program is running in. Hence they belong to the IO monad which assures sequential execution of IO actions in contrast to Eden [2] and Goffin [3].

A value can be written to and read from a port with the functions

```
writePort :: Port a -> a -> IO ()
readPort  :: Port a -> IO a
```

A port can be used in the same way as a channel in Concurrent Haskell, except that only the process which creates the port can read from it.

For writing to a port it doesn't matter where a port is located. We provide all necessary "contact"-information in the data type and impose no limitations on what thread may write (see Figure 2).

For sending messages over the network we have to encode them in a binary representation as we have no access to the underlying internal closure. In this implementation as a library for the Glasgow Haskell Compiler [4] we send messages as strings. We use the function `show` from the class `Show`. Therefore the type of the messages which can be sent with `writePort` must be an instance of the class `Show`. On the other hand a message must be reconverted from the

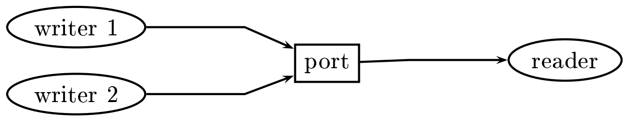

Figure 2: Modelling ports as channels in a distributed setting

string representation into the corresponding data type on reading it from a port. Therefore the type of the messages which can be received with `readPort` must be an instance of the class `Read`, too. This means the messages of a port need both instances. We subsume these properties in a new class named `Serialize`. Note that we will omit this class constraint in the type signatures for brevity.

But this imposes the restriction that no functions, infinite data structures, or mutable structures like MVars can be sent to remote threads.

For the connection of independently started components we provide a global registration mechanism. With

```
registerPort   :: Port a -> Name -> IO ()
unregisterPort :: Port a -> IO ()
```

ports can globally be registered and unregistered on one computer. Other processes can then lookup a registered port with

```
lookupPort :: Host -> Name -> IO (Port a)
```

from anywhere else in the network. In the actual implementation `Port.Host` and `Port.Name` are just type synonyms for `String`.

For suspending on two ports we provide a merge function which even allows a programmer to merge two ports of different types. We use Haskells `Either` type:

```
mergePort :: Port a -> Port b -> IO (Port (Either a b))
```

With this function a process can suspend on messages of different types. This is needed, because a process can suspend on messages from different other processes. The type of these messages should not be the same, because these processes can perform completely different tasks in the system. An example for such a system is a chat where a chat clients may receive messages from a chat server and the keyboard (respectively GUI) [7]. From the software engineering point of view a restriction to merging only ports of the same type like in Eden [2] or Goffin [3] is awful.

It should also be possible that the merged ports can afterwards also be used in their non-merged version. As the client is the only process that can read from these ports there can be no conflict that two readers want to read from the ports and their merged version at the same time.

For the creation, termination and error handling of processes we use the same functions as Concurrent Haskell and the module `Exception`:

```
forkIO         :: IO () -> IO ThreadID
myThreadId     :: IO ThreadId
killThread     :: ThreadID -> IO ()
raiseInThread  :: ThreadId -> Exception -> IO ()
try            :: IO a -> IO (Either Exception a)
```

Finally we provide a linking mechanism. With the function

```
linkAndKill :: Port a -> IO Link
```

a link between the executing process and a port is established. A polling mechanism contacts the host periodically where the linked port is located and checks for its existence. If the port does no longer exist the process is terminated by an exception. The runtime system will trigger matching links, too, if it notices a transmission error. This can be caught and the process can initiate a reorganization of the whole system. A more convenient function for linking ports is the function

```
link :: Port a -> IO () -> IO Link
```

which takes an additional `IO` action as parameter. This action is performed if the linked port does not exist any more (`linkAndKill` is just an application of this function). We can use `link` for example to send a message if a port dies. Once an established link is not needed any more it is possible to remove links with the function `unlink :: Link -> IO ()`. We have also added a fault tolerant version of `writePort`

```
writePortFail :: Port a -> a -> IO () -> IO ()
```

Similar to `link` the action in the third parameter is performed in the case of an erroneous sending.

3.2 Limitations of the current port concept

In a multithreaded environment it is often necessary to allocate resources and pass them on to several threads (we refer to the notion of *threads* opposed to operating system-level *processes*). A common usage would be a simple application consisting of two threads, one writer and one reader:

```
do {p <- newPort; forkIO (reader p); writer p}
```

Although there is no apparent error in this sequence, it surely violates the single-reader criterion as specified above! We require that the only thread allowed to read is the thread which created the port. Of course this could easily be alleviated by forking the writer instead of the reader. However, we feel that the compiler should be able to reject as many faulty programs as possible. Note that distinguishing between ports using different types such as `ReadPort` and `WritePort` doesn't solve this problem, either.

In providing just a stream of messages received on a port we can avoid the necessity for an explicit `readPort` statement. If this stream is passed on to another thread like in the case above, several threads can **share** copies of this stream. Multiple threads reading one port are not competing to grab a message, either. For sending the reference to the port can still be used. We only hide `readPort` from the view of the programmer and provide a new function to create a stream with its corresponding port reference:

```
newStream :: -> IO (Port a, [a])
```

The resulting port can be used for registering or linking as before.

Another new function the API must offer is merging two streams as we can't use `mergePort` any longer. Merging has to interleave access to two streams and return a list of either an item from the first or the other stream, depending on which becomes available first.

```
mergeStreams :: [a] -> [b] -> [Either a b]
```

3.3 A Chat Example

As an example we now give the implementation of a chat server using the new stream based approach. Clients can connect to this server and messages arriving from one client will be relayed to all others. Its communication interface is given by the data type

```
data ServerMsg = Connect String (Port ClientChatMsg) | Send String String
           | Close String (Port ClientChatMsg) deriving (Read, Show, Eq)
```

The server creates a port for external communication on startup and registers this port as `ChatServer`.

```
main = do (serverPort,stream) <- newStream
          registerPort serverPort "ChatServer"
          foldM chatServer [] stream
```

After the initialization it proceeds in a loop. The loop is elegantly implemented with a `foldM` on the stream, which is an infinite list. A list of the ports of all connected clients is kept as the state of the process. This list changes in dependence of connecting or leaving clients. A new chat message is broadcasted to all the other connected clients:

```
chatServer :: [Port ClientMsg] -> ClientMsg -> IO ()
chatServer clientPorts msg = case msg of
     (Connect name clientPort) -> do
         mapM_ (\ p -> writePort p (Login name)) clientPorts
         return (clientPort:clientPorts)
     (Close name clientPort) -> do
         let newClientPorts = filter (/= clientPort) clientPorts
         mapM_ (\ p -> writePort p (Logout name)) newClientPorts
         return newClientPorts
     (Send name str) -> do
         mapM_ (\ p -> writePort p (Chat name str)) clientPorts
         return clientPorts
```

A client process uses the interface

```
data ClientChatMsg = Chat String String | Login String |
                     Logout String deriving (Read, Show, Eq)
```

to receive messages from the chat server. The `Chat` message is used for new chat messages. The first string is the nickname of the user taking part in the chat and the second string is her chat message. The `Logout` message is sent if a client leaves the chat. We don't need an algebraic data type for the interface to the keyboard process, we just use strings.

First a client process initiates the connection to a chat server on `host` by obtaining a reference to the remote stream using `lookupPort`. Then its own streams are created and a process for the input from the keyboard is forked.

```
main = do
  putStrLn "Host of chat server? "
  host <- getLine
  putStrLn "Nickname? "
  name <- getLine
  serverPort <- lookupPort host "ChatServer"
  (chatPort,chatStream) <- newStream
  writePort serverPort (Connect name chatPort)
  (keyboardPort, keyboardStream) <- newStream
  forkIO (readKeyBoard keyboardPort)
  let stream = mergeStreams chatStream keyboardStream
  mapM_ (client name serverPort)
        (takeWhile (/= (Right "")) stream)
  writePort serverPort (Close name chatPort)
```

The two streams are merged into one and the process proceeds in a loop analyzing the messages. Here we use `mapM_` instead of `foldM_` because there is no state involved in the recursion (apart from IO actions). To terminate the recursion we use `takeWhile` on the infinite stream. The empty input from the keyboard represented by the message `(Right "")` in the merged stream terminates the client. The client sends a `Close` message to the server and terminates. All other messages are handled in the function `client`. Messages from the server are displayed (not shown) and messages from the keyboard are forwarded to the server.

```
client :: String -> Port ServerMsg -> Either ServerMsg String -> IO ()
client name serverPort msg = case msg of
     (Left serverMsg) -> putStrLn (display serverMsg)
     (Right str)      -> writePort serverPort (Send name str)
```

The process for reading from the keyboard is not presented here. It just reads strings from the keyboard, sends them to the client process and terminates itself if the user inputs the empty string.

3.4 Fault tolerance

This chat application does not behave fault tolerant. If a client dies and the next chat message is broadcasted to all chat clients an exception is thrown and the chat server crashes because the port of the dead client does not exist anymore. With the linking mechanism and the use of `writePortFail` instead of `writePort` we can easily guarantee fault-tolerance for our server. In the case of a failure we close the port to which writing failed. Therefore we just have to add the following link into the `Connect` case in the server process:

```
(Connect name clientPort) -> do
   link clientPort (writePort serverPort (Close clientPort))
```

and modify the `writePort` instruction in the broadcast of chat messages:

```
writePortFail p ... (writePort serverPort (Close p)) ...
```

The chat server just sends a `Close` message to itself if a port does not exist any more (note that we have to change the loop to pass `serverPort`, too!).

4 Implementation

As Port-based Distributed Haskell is the foundation on which we built streams, we will give only a short overview of the internals of communication using ports, then focusing on the implementation of streams. A more detailed description of the layout of the internal data structures used for ports can be found in [7].

Communication via ports uses two different kinds of techniques for passing data: In case we just want to send a message to a port located in the same program, we can use a simple Concurrent Haskell channel to deliver the message. This channel is typed over the messages the port can receive.

On the other hand, we have to contemplate communication over the network. We use standard TCP/IP system-level functions to connect a socket to a remote host, marshal the message using the class `Serialize` and deliver it by writing to the file descriptor obtained from the socket. All necessary data like host name and TCP/IP port number are encoded in the port's data structure. These are used by remote hosts when writing to a port reference, e.g obtained by receiving a message over the network.

A separate program (called the "external post office") running on remote hosts which wish to offer services based on registered port names translates the symbolic names used in `registerPort` and `lookupPort` statements to their internal representation. It is necessary that this program binds to a well-known TCP/IP-port so it can be contacted without any prior knowledge. The service offered is similar to the mechanism employed in *remote procedure calls* [12].

Internally, data arriving on the socket of a port is transferred to a helper thread using a channel of type `String`. There it is converted from a string to it's equivalent Haskell expression using once again the functions provided by the class `Serialize`. Messages which can't be converted are rejected as bogus, valid ones are finally passed into the typed channel we already used for internal communication, thus forming the final per-thread message queue.

The whole structure of a port and the internal and external communication is summarized in Figure 3.

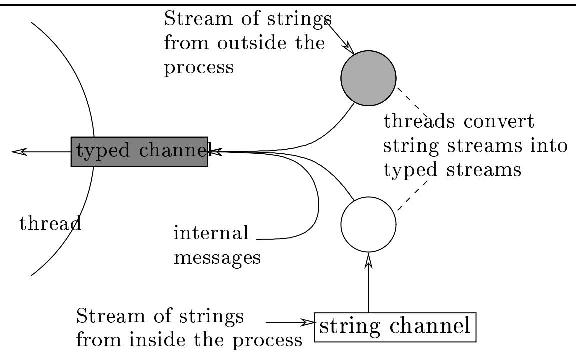

Figure 3: Structure of a port and internal/external communication

A polling mechanism is employed for the linking of ports. All linked ports are stored in a database. A process in the background polls all registered ports in a fixed schedule. If one of these ports does not exist anymore, an exception is thrown (`linkAndKill`) or the specified IO action is performed (`link`, `writePortFail`). Of course the latter could be programmed by hand from the programmer using `catch`. The implementation of links needs the Distributed Haskell runtime system. They cannot be handled outside the Distributed Haskell library.

4.1 Streams

Although streams provide a powerful abstraction they are yet easy to implement. The only major change related to the `Port` module is that we need a version of `readPort` without the safety belt which checks the `ThreadID` of the caller. The helper function `getPortContents` will retrieve all items in a port in a list. This function is modeled in the same fashion as `getChanContents` from the module Concurrent. This includes suspending until a new message arrives in case the port is empty.

```
getPortContents :: Serialize a => Port a -> IO [a]
newStream       :: Serialize a => IO (Port a, [a])
```

Merging two streams is much more difficult. We have to use two different threads, each one waiting for messages on its stream and forwarding them encapsulated in an `Either` type to a common channel.

```
mergeStreams :: [a] -> [b] -> [Either a b]
```

5 Related Work

– **Goffin** [3] extends Haskell with concurrent constraint programming and a special port concept for internal and external communication. The ports are not integrated in the IO monad.

For reacting on multiple ports Goffin proposes a fair `merge :: Port a -> Port a -> Port a` for ports. But both merged ports must have the same type. This restriction is too strong as we have seen in the chat example where the `Either` type was essential.

– **Eden** [2] is an extension of Haskell for concurrent and parallel programming. A process concept is added in which every process has a fixed number of input and output channels for communication with other processes. Communication is not integrated in the IO monad and with a fair `merge`, which is part of Eden, processes can behave nondeterministically. Like in Goffin this `merge` is restricted to channels of the same type.

Furthermore in Eden a process can only read from or write to a fixed number of channels. The connections between the processes cannot be changed dynamically.

Eden is developed for parallel programming where programs have a more hierarchical structure than in distributed programming and it is difficult to implement complex protocols in Eden. It is also not possible to connect two independently started processes in Eden.

– **Curry** [5] is a functional logic programming language which extends Haskell with needed narrowing, residuation and encapsulated search. For distributed programming Curry adds named ports which guarantee that all readers of one port are executed in the same Curry program. This is thought to eliminate the implementation problems with multiple readers. On the other hand a programmer can also send logical variables through ports which is proposed as an easy answering mechanism. But with these logical variables channels which have multiple readers can be programmed. This results in the same problems with multiple readers which should be avoided by the introduction of ports. Logical variables can only be used as comfortable answer variables. This is no restriction from the formal semantics, but from the implementation.

Communication in Curry is like in Goffin a constraint which has to be solved. A programmer must learn a new programming paradigm. Furthermore only the external communication must be integrated in the `IO` monad. Internally concurrent processes communicate via lazy streams. This can result in problems with laziness and strictness annotations have to be added sometimes. Another problem is that a concurrent application cannot easily be distributed to a network because the processes have to be transfered into the IO monad. This can yield problems with the scalability of a system.

– **Glasgow distributed Haskell** [11] is an approach for the integration of Glasgow parallel Haskell [14] and Concurrent Haskell. It provides a closed distributed systems and communication between processes like in Concurrent Haskell. The main idea is the distribution of a shared memory system. Communication between processes is not strict. Hence the programmer does not know when data is exchanged between the components of the network and can not estimate when and how much net traffic is produced.

Fault tolerance is restricted to error handling like in Concurrent Haskell. If one of the computers in the network crashes, then the whole system crashes, too. But the greatest disadvantage of this approach is the restriction to closed systems which makes it impossible to implement many distributed applications like telephony or peer-to-peer applications.

– Finally, we once again want to compare our approach with **Erlang** [1]. The first advantage of Distributed Haskell is that messages are statically typed. The type of the messages which can be sent through a port are the communication interface of the process which reads from the port. In Erlang such an interface does not exist. Furthermore this type system provides safety in program development. For example typos yield a compile-time error in Distributed Haskell, not a deadlock like in Erlang. Another advantage is our linking mechanism which is more powerful than linking in Erlang. We can add arbitrary `IO` actions to the links which are performed if the linked port dies. In Erlang it is only possible to receive a message if another process dies.

6 Conclusion and Future Work

Our modification of the port-based approach introduces the notion of streams. Streams are basically a list of the messages received on a typed port. Both concurrent and distributed programming is covered by this approach. The limitations imposed by the single-reader semantics of ports disappear when we abstract from the explicit `readPort` statement.

All described extensions are implemented using the Glasgow Haskell Compiler [4], based on the libraries `Concurrent` and `Socket`. The module `PStream` can be used together with any Haskell program compiled using the Glasgow Haskell Compiler. The library sources and sample applications can be found at `http://www-i2.informatik.rwth-aachen.de/hutch/DistributedHaskell/`.

A possible future focus will be on analyzing how good distributed programming and lazy evaluation match. External communication forces eager evaluation of large parts of a program. Furthermore, usually other IO actions will require a serial execution, thus reducing the advantage of lazy evaluation, too. As we can't yet send heap fragments to remote hosts, the advantages of lazy functional programming may be limited to the "computational" (in contrast to the "communication") part of programs. On the other hand the higher order functions used for working on streams provide a good abstraction from the otherwise tedious work of explicitely programming the message exchange.

References

1. J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
2. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Pena-Mari. Eden — The paradise of functional concurrent programming. *LNCS*, 1123:710ff., 1996.
3. Manuel M. T. Chakravarty, Yike Guo, and Martin Köhler. Distributed Haskell: Goffin on the Internet. In M. Sato and Y. Toyama, editors, *Proceedings of the Third Fuji International Symposium on Functional and Logic Programming*, pages 80–97. World Scientific Publishers, 1998.
4. The Glasgow Haskell compiler. http://www.haskell.org/ghc/.
5. M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99).* Springer LNCS (to appear), 1999.
6. Frank Huch. Erlang-style Distributed Haskell. In *Draft Proceedings of the 11th International Workshop on Implementation of Functional Languages*, September 7th – 10th 1999.
7. Frank Huch and Ulrich Norbisrath. Distributed programming in Haskell with ports. *LNCS*, 2011:107–121, 2000.
8. Simon Peyton Jones et al. Haskell 98 report. Technical report, http://www.haskell.org, 1998.
9. Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 January 1996.
10. Object Management Group. Common Object Request Broker Architecture (CORBA). http://www.omg.org/gettingstarted/.
11. R. Pointon, P. Trinder, and H-W. Loidl. The design and implementation of glasgow distributed Haskell. *LNCS*, 2011:53–70, 2000.
12. Richard W. Stevens. *UNIX Network Programming.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.
13. Volker Stolz. Robuste verteilte Programmierung in Haskell, 2001. RWTH Aachen, Germany.
14. Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, Philadephia, USA, May 1996.