Distributed Programming in Haskell with Ports

Frank Huch[1] and Ulrich Norbisrath[2]

Lehrstuhl für Informatik II, RWTH Aachen,
52056 Aachen, Germany
[1] huch@i2.informatik.rwth-aachen.de
[2] ulrich@norbisrath.de

Abstract. We present an extension of the lazy functional programming language Haskell for distributed programming. For the communication between processes we add a port concept. Ports behave like channels in Concurrent Haskell except that only the process which creates a port can read from it. Ports can also be sent through other ports. The receiver can then also write messages through the received port. This is independent of the location in a network. The programmer uses the same functions to write to local or remote ports. Communication between concurrent and distributed processes is programmed with the same functions. Concurrent processes can easily be distributed, for example to provide scalability of a system. In many distributed applications it is necessary that two independently started programs can connect at runtime. Therefore we provide the registration of ports. Other processes can look them up from anywhere in a network.
The implementation consists of a library which provides functions for creating new processes, communication between concurrent and distributed processes, 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 co-operate 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 programming, modern programming languages must support distributed programming. It is not sufficient to provide a library for communication via sockets. The language has to be extended with a high level concept for distribution and communication between processes. We want to extend the functional programming language Haskell [J+98] with features for elegant distributed programming.

The discussion of communication in distributed systems in Sect. 2 shows the advantages of a port concept, which is introduced in Sect. 3. We present the implementation in Sect. 4 and discuss related work in Sect. 5. Section 6 concludes and discusses future work.

2 Distributed Communication

There have been made some approaches for extending functional languages for concurrent or distributed programming. The most successful one is Erlang [AWV93], which was developed by Ericsson and has been used for the development of many telecommunication applications. Erlang is an eager functional programming language, which is extended with special features for concurrent and distributed programming. New processes can be created with `spawn` on a local or a remote computer. Every process has a process identifier (*pid*), which is used for the communication between processes. Other processes can send messages to this pid and messages can also contain pids to distribute pids in a network. If a message is sent to a process it is stored in a mailbox. The receiving process can conveniently access this mailbox with pattern matching. It need not extract messages in their chronological order. Only relevant messages can be fetched with pattern matching. The others can be left in the mailbox and processed later.

Another important point of Erlang is that communication between concurrent processes on the same computer does not distinguish from communication to remote processes in a network. The programmer uses the same programming techniques. 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 also provides a linking mechanism. Processes of the system can automatically be informed, if others die, for example because a computer crashes. Hence these processes can react on the failure and reorganize the system to a consistent state.

But for all that Erlang has a great disadvantages. It is untyped. Therefore it is more difficult to find program errors than in Haskell. There have been made two approaches for typing Erlang [MW97,AA98], but they only type sequential programming in Erlang. The communication stays untyped. But also here a type system is necessary. In Erlang already typing mistakes[1], like writing the atom

[1] Here we do not mean type errors. An example for a typing mistake is 'Erlng'.

`lookup` in the pattern of a receive statement and `lookUp` at the corresponding send statement does not yield a compile time or runtime error, but a deadlock. Finding these typing mistakes is very difficult.

We want to extend Haskell with a similar communication mechanism, but in consideration of Haskells type system. This yields more safety in program development. Concurrent Haskell [JGF96] is state of the art for concurrent programming in Haskell. It provides functions to start and terminate threads inside an application and to synchronize them with mutable variables (*MVar*) in the IO monad. On top of these MVars it also provides semaphores and asynchronous channels for message passing. But there is no concept for distributed programming. Therefore we want to extend Haskell with a powerful mechanism for distributed programming.

In our first approach we extended Haskell with an Erlang-style communication mechanism [Huc99]. The main problem in this approach is to type the communication. Different processes can understand different messages but others can also understand the same messages. A type system with subtypes would be needed. But it is difficult to integrate subtyping into Haskells type system. Therefore we have implemented runtime type checking. But this is no good match to Haskells type system.

We now have decided to extend the concepts of Concurrent Haskell with communication via channels and MVars for distributed programming. But this leads into implementation problems. In Concurrent Haskell many processes may synchronize on a mutable variable (*MVar*) or a channel. Distributing some of these processes in a network leads to synchronization problems, because it is not clear where an MVar is located. Consider the following situation:

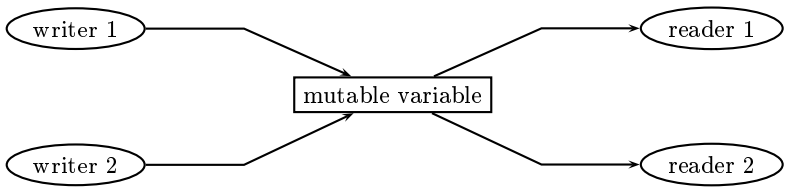

Figure 1: Distribution of Concurrent Haskell Processes

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 where can the mutable variable be located? It has to be located on one computer in the net, 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. We do not know, when a mutable variable or channel is garbage. Other processes in the distributed setting communicate with each other and the MVar (a reference to it) can also be distributed to the whole network. So it will be almost impossible to check, if an MVar is still known somewhere in the net and an algorithm would be very expensive and produce much communication in the network. Especially, if some parts of the system have already crashed. Systems like Glasgow Parallel Haskell [THM+96] or Glasgow Distributed Haskell [PTL00] implement distributed garbage collection, but they do not provide the development of open systems, which is essential for the development of distributed applications. 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. It would be interesting to investigate, if distributed garbage collection can be efficiently implemented with the conditions of programming open and fault tolerant systems. We think that these algorithms are too complex and need too much communication, i.e. traffic in the network. Therefore we restrict communication to only one reader for each channel and can avoid these problems. Our practical experience showed, that this restriction is no disadvantage, as we will discuss later.

3 Distributed Haskell

In the discussion of distributing Concurrent Haskell, we have seen which problems appear with multiple readers and writers of an 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. On the other side, 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, when they send a message to the MVar. A failed write operation can throw an exception. This can be caught and the writer can initialize a reorganization of the crashed components, for example on another computer.

Erlang also has this restriction, because a mailbox is associated with a singleton process. Only this process can read from it. Transferring the Erlang model to Haskell leads to typing problems, because the Mailbox of a process needs a fixed type in Haskell. But consider two clients, which communicate with a database server, but handle different jobs in the rest of the system. Both will receive mes-

sages from the database, for example if they look up the value of a given key. Therefore there mailboxes must have the same type. But in other parts they communicate with different processes and exchange different data there. The result would be that all mailboxes of the system must have the same type. So type checking does not help to find errors. Another problem is that the system is not structured any more, because the messages a process may receive cannot be represented as one data type. But these messages are its communication interface and they have to be visible.

Another solution would be sub-typing for the messages. We have implemented this in [Huc99] by runtime type checking. But this concept is no good match to Haskells type system. Our way out of this problem is to allow multiple mailboxes for every process, like in Concurrent Haskell. We call these mailboxes *ports* and every port of a process may have another type. But we restrict these ports to one reader. This reader is the process which creates the port.

3.1 The Distributed Haskell Library

Ports are represented as a 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. Hence they belong to the IO monad in contrast to Eden [BLOMPM96] and Goffin [CGK98].

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.

To guarantee this by the type system we first wanted to distinguish read-write ports and write ports. Read-write ports can be converted into write ports and only write ports can be sent through other write ports. But sending a port to another process is not the only possibility for distributing ports in a process network. They can also be passed to other processes as parameters, when new processes are created with `forkIO`. Here we cannot avoid the distribution of read-write ports. Therefore we have renounced this distinction. A runtime checking is needed anyway. Hence, reading from a write port yields a runtime error.

A postulation we made above is that it is all the same, where a port is located in the net. Sending to it should stay the same. In the case of a remote computer the messages have to be coded binary. In this first implementation, as a library for the Glasgow Haskell Compiler [GHC], we send them as strings. 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 messages must be reconverted into the corresponding datatype, if a process reads from a port. Therefore the type of the messages, which can be received with `readPort` must be an instance of the class `Read`. This means the messages of a port need both instances. But this is no problem in Haskell, because they can be derived for algebraic data types. But this is a restriction, because no functions, infinite data structures, or mutable structures like MVars can be sent through ports.

On the first sight these functions seem to be enough for programming distributed systems. But consider the development of a chat. Designing a chat system one will program a chat server, which manages the clients taking part in the chat. New clients can join the chat and others can exit. But how can new clients connect to the chat server? For the connection of two independently started components we provide a global registration mechanism. With

```
registerPort :: Port a -> PortName -> 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 :: PortHost -> PortName -> IO (Port a)
```

from anywhere else in the net. In the actual implementation `PortHost` and `PortName` are just type synonyms for `String`. But in later implementations we will also extend PortHost to IP addresses and allow Haskell programs with names, as nodes in Erlang. Then `PortName` will also contain the name of the Haskell program and ports in different Haskell programs can be registered with the same name on one computer. This can for example be useful, because two servers which register themselves globally with the same name can still be executed on the same computer.

With these functions for registration and lookup of ports the chat server can register its in-port and the clients can lookup this port. Their first message to the chat server can contain a port, which the client has created and to which the server sends further messages.

But this example shows another problems. When a client has accessed a chat, it sends new text messages to the server and this server broadcasts them to the other clients. For the client process this means, that it has to read messages coming from the server and messages coming from the keyboard or another process managing the user interface. But there is no fixed order in which the client will receive these messages. Therefore it has to suspend on both. The easiest way would be, that the keyboard process and the chat server use the same port and the client process reads from it. But from the software engineering view this it not nice, because the chat server and the keyboard use messages of the same type to communicate with the client. Hence we need two different ports, the client can suspend on. Therefore Eden proposes a merge function which would in our setting have the type `merge :: Port a -> Port a -> IO (Port`

a). But this function would not solve the problem above, because still both ports must be of the same type. Therefore we provide a merge function, which 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 the client can suspend on messages of different types from the chat server and the keyboard. It should also be possible, that the merged ports can afterwards also be used in there un-merged version, because the client wants to ignore messages from one of them. But the client is the only process that can read from these ports. Hence 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. If the port does not exist anymore, the process is terminated by an exception. 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 anymore. With this function it is for example possible, to send a message if a port dies. If an established link is not needed anymore, it is possible to abolish 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 A Chat Example

As an example we now present the implementation of a chat server. 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)
```

The server creates a port for external communication in its initialization and registers this port as `ChatServer`.

```
main = do serverPort <- newPort
          registerPort serverPort "ChatServer"
          chatServer serverport []
```

After the initialization it proceeds in a loop, where it holds a list of the ports of all connected clients. This list changes in dependence of connecting or closing clients. A new chat message is broadcasted to all the other connected clients:

```
chatServer :: Port ServerMsg -> [Port ClientMsg]  -> IO ()
chatServer serverPort clientPorts = do
   msg <- readPort serverPort
   case msg of
     (Connect name clientPort) -> do
         mapM_ (\p -> writePort p (Login name)) newClientPorts
         chatServer serverPort (clientPort:clientPorts)
     (Close name clientPort) -> do
         let newClientPorts = filter (/= clientPort) clientPorts
         mapM_ (\p -> writePort p (Logout name)) newClientPorts
         chatServer serverPort newClientPorts
     (Send name str) -> do
         mapM_ (\p -> writePort p (Chat name str)) clientPorts
         chatServer serverPort 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. For the interface to the keyboard process we just use strings. First a client process initiates the connection to a chat server on `host`. Then its own read ports 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
        chatserverPort <- lookupPort host "ChatServer"
        chatPort <- newPort
        writePort chatserverPort (Connect name chatPort)
        keyboardPort <- newPort
        forkIO (readKeyBoard keyboardport)
        inPort <- mergePort chatPort keyboardport
        client chatserverPort inPort chatPort
```

The two read ports are merged into one port and the process proceeds in a loop. Here messages from the server are displayed and messages from the keyboard are forwarded to the server.

```
client :: String -> Port ServerMsg -> Port (Either ServerMsg String)
          -> Port ClientChatMsg -> IO ()
client name serverPort inPort chatPort = do
   msg <- readPort inPort
   case msg of
     (Left serverMsg) -> do putStrLn (display serverMsg)
                            client name serverPort inPort chatPort
     (Right "") -> writePort serverPort (Close name chatPort)
     (Right str) -> do writePort serverPort (Send name str)
                       client name serverPort inPort chatPort
    where display :: ClientChatMsg -> String
          display (Chat name str) = name ++ " : " ++ str
          display (Login name)    = name ++ " logged in"
          display (Logout name)   = name ++ " logged out"
```

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.

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 just close the port, which could not be written. Therefore we just add the following linking to the `Connect` case in the server process:

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

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 anymore.

The example shows, that it is easy to implement a client server architecture like a chat in Distributed Haskell. Our practical experience shows, that our

communication model with only one reader per port is no real restriction in the development of arbitrary distributed systems. This is also confirmed by the success of Erlang, which has the same restriction.

As an example multiple readers seam to be useful for architectures like the producer/consumer problem. But this problem can easily be implemented in Distributed Haskell too. We add a process, which simulates the store for the products. The producers insert products by sending messages to the store process. The consumers request the store for new products and obtain them as messages from the store process. The system is divided into two client/server architectures. The opportunity of this implementation is that the processes can be linked to the ports of the store and we can detect if one of the components (e.g. the store) crashes. In this case we can replace the store by a new process. Some products may be lost, but the system still works. Another possibility would be the definition of a second store process, which is (with some delay) identical with the other store and can replace it if it crashes. This implementation of fault tolerance seems to be expensive in this case, but we think this cannot be avoided.

4 Implementation

The implementation is a library, which can be imported in any Haskell program. The whole library is written in Concurrent Haskell.

In our port based extension we have to handle two different kinds of communication through ports. For the internal communication inside a Haskell program, which consist of multiple concurrent processes, we want to use communication through channels from Concurrent Haskell. For the external communication between different Haskell programs, we want to use communication via TCP/IP. Therefore we use the GHC library `Socket`.

We also allow multiple Haskell programs on one computer. Hence distributed systems can also be developed on a single computer. No network of multiple computers is needed. Therefore on every computer an external post office is started. This external post office listens on a fixed socket. All started Distributed Haskell programs register themselves here and obtain a program number and a free socket number, they listen to for external messages. Global registrations of ports in the Distributed Haskell programs are recorded in this external post office and other processes can lookup the corresponding ports here.

The use of ports is supposed to be transparent, independent of the host a writing process is located on. Therefore the internal representation contains the IP-Address of the host, where the port is located and the program number of the Haskell program it is created in. Therefore messages that are written to the port from outside can be directed to the correct Haskell program. We want to guarantee, that only the process which creates a port can read from it. Hence a port also contains the `ThreadID` of its creating process. If another process tries to read from it, then a runtime exception is thrown.

For the internal communication through ports we use channels. This is efficient and even lazy. For external communication we cannot communicate through channels. The values which are sent through a port need a representation as a sequence of bytes. We use a string representation. Values can be converted from and into this representation with `read` and `show`. For algebraic data types, which do not contain function types, these functions can easily be derived in Haskell.

All external communication is sent through the external post office. Therefore all newly created processes are registered in the external post office. They also obtain a socket number, on which all messages from external processes arrive. When a new port is created a process is forked, which listens on this socket and forwards all incoming messages to the typed channel representing the port. When this process is created the type of the port is known. The conversion of incoming strings into typed values with `read` is fixed for the lifetime of the port and this process. So all messages which are sent to a port arrive in the typed channel, independently from their origin. The function `readPort` only tests, if the reading thread has created the port and then reads from this typed channel with `readChan`.

For communication via ports it is also necessary, that ports can be sent through external ports. This means that they also need a string representation and are an instance of the classes `Read` and `Show`. As long, as a port is just sent through the typed channel representing a port (internal communication) its representation stays unchanged. Only if it is sent through the net we lose the typed parts. The string representation only contains the threadID, the program identifier, the IP-address of the host, and the socket number. If a port is sent through the net and returns to the program where it is physically located, it is not possible to access its typed version again, because of the Haskell type system. We would need a data structure, which can contain ports of arbitrary types to get back the typed representation of the port. As an optimization we avoid communication via sockets in this case. For every port we have added a string channel, through which this kind of internal communication is executed. Again a special process forwards the messages from this string channel into the typed channel. This process is created in the function `newPort`. The string channel can be stored in a database. When a port returns to its origin, this string channel can be looked up in the database and added to its representation again. This lookup is a sideffect. It is implemented with `unsafePerformIO`, but it behaves transparent to the application, because every port will only obtain one corresponding representation. The whole structure of a port and the internal and external communication is summarized in Figure 2.

The two conversion processes for each port, which convert the string messages into typed messages with `read` have a disadvantage. They impede garbage collection of a port. They always hold a reference to the string channel and the typed channel of a port. Therefore these port components never become garbage. As a solution we have implemented a function `destroyPort :: Port t -> IO()` which terminates the conversion processes and removes all entries of the port in the internal and external post office. If a new port is created we

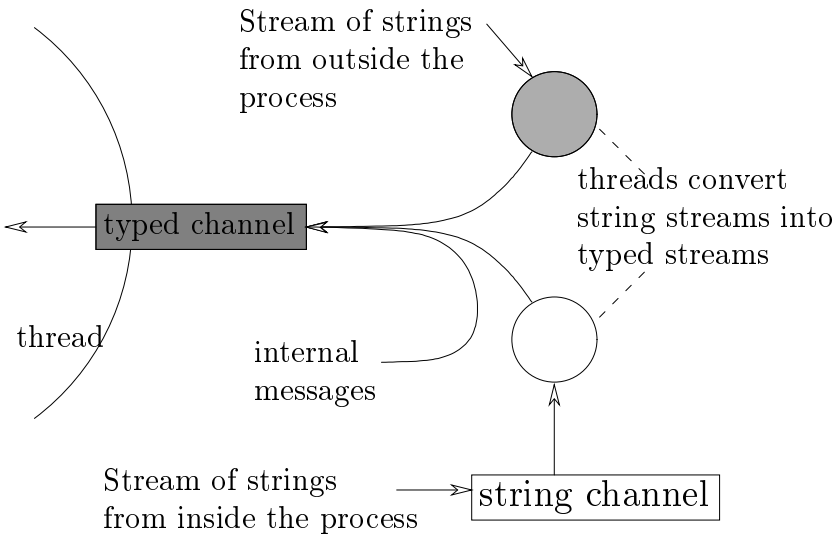

Figure 2: Structure of a port and internal/external communication

bind this function as a finalizer to the port. Hence if no reference to the exists anymore, then the conversion process is killed and all references to its channels are eliminated. So the channels become garbage too.

The last internal representation of a port is the merged port as read only port. Both merged ports are internally represented by channels. If a process wants to read from a merged port, two processes are created, which suspend on these two channels. If one of them receives a message, this message is extended with the constructor `Left` respectively `Right` and transfered to an MVar, representing the merged port. After that both processes are terminated and the original ports can be read again. The only problem here is that both processes can read their channels simultaneously. We have to guarantee mutual exclusion here, but this is impossible without busy waiting. Hence in this case we write one of the received messages back into the corresponding channel using the function `unGetChan`.

The function `register` registers the string channel component of the port at the external post office. With `lookupPort` this string channel can then be accessed from anywhere in the net.

For the process manipulation we just use the functions of Concurrent Haskell.

Finally we provide a polling mechanism 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`). This could be programmed by hand from the programmer, but with the provided functions it is more convenient.

5 Related Work

There are many other approaches for the extension of functional programming languages. We compare the main ones with our approache:

– **Goffin** [CGK98] 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. Nondeterminism and input/output is not encapsulated in the IO monad anymore. Furthermore a user has to learn about concurrent constraint programming, another programming paradigm.

Goffin does not restrict the number of readers and writers of a port. As we described in Sect. 2 multiple readers of a port can yield problems implementing fault tolerant systems. An implementation of this concept seems difficult. This is confirmed by the fact, that an executable implementation of Goffin does not exist yet.

For reacting on multiple ports Goffin proposes a fair `merge :: Port a -> Port a -> Port a` for ports. With this function a process can wait for different messages of different ports and branch in dependence of the incoming message. But both merged ports must have the same type. This restriction is a too strong, as we have seen in the chat example.

– **Eden** [BLOMPM96] 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.

Processes can suspend on their different input channels. Messages in other channels are buffered automatically. To react not only on one input channel, it is possible to merge channels, with the same restrictions as in Goffin. Furthermore in Eden a process can only read from or write to on 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. But this is needed for many distributed applications.

– **Curry** [Han99] 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 can be programmed, which have multiple readers. This results in the same problems with multiple readers, which should be avoided by the introduction of ports. The problems are also reflected in the fact, that no actual Curry implementation provides unrestricted sending of logical variables through a net. 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. Only for the external communication the `IO` monad is used. 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** [PTL00] is an approach for the integration of Glasgow parallel Haskell [THM+96] and Concurrent Haskell. It is provides 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. But this is necessary for the development of distributed applications.

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 (mobile) telephony.

– **Facile** [TLK96] is an extension of Standard ML for distributed programming. It provides channels with multiple readers and writers in a network. It also provides open distributed systems and a registration mechanism similar to ours. But Facile does not provide a linking concept for fault tolerant programming, which is needed for the development of distributed applications. It is only possible to program with timeouts. Finally, Facile is a strict programming language and the communication is implemented as a side effect. We have integrated the communication in the `IO` monad and hence preserve referential transparency.

– Finally, we once again want to compare our approach with **Erlang** [AWV93]. 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 typing mistakes yield a compile-time error in Distributed Haskell, not a deadlock like in Erlang. Another opportunity is our linking mechanism, which is more powerful then 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. The process which receives this messages must interrupt its execution, react on this message, and afterwards resume its work. Performing different actions in case of different exceptions is much more difficult, then in our linking mechanism.

6 Conclusion and Future Work

We have extended Haskell for concurrent and distributed programming. With the example of a chat we have shown, how easy distributed systems can be developed using Distributed Haskell. The main concept of Distributed Haskell is the use of ports. Ports differ from channels in Concurrent Haskell in the fact that only the process, which created a port, can read from it. But this restriction is needed for the implementation, especially when we want to develop robust and fault tolerant systems. Another opportunity of this restriction is that ports, which are not used anymore are detected as garbage and get collected automatically. All described extensions are implemented using the Glasgow Haskell compiler [GHC] with the libraries `Concurrent` and `Socket` and built the library `Port`, which can be used together with the Glasgow Haskell Compiler [GHC].

In future work we want to implement lazy communication through the net. This means sending fragments of the heap instead of sending there string representation. With this extension we would also be able to send infinite data structures and functions. Furthermore we want to investigate if this is suggestive, because calculations can be duplicated. Therefore we want to inspect how good distributed programming and lazy evaluation match.

References

[AA98] Thomas Arts and Joe Armstrong. A practical type system for Erlang. Technical report, Erlang User Conference, September 1998.

[AWV93] J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[BLOMPM96] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Pena-Mari. Eden — the paradise of functional concurrent programming. *LNCS*, 1123:710ff., 1996.

[CGK98] 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.

[GHC] The Glasgow Haskell compiler. http://www.haskell.org/ghc/.

[Han99] 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.

[Huc99] Frank Huch. Erlang-style distributed haskell. In *Draft Proceedings of the 11th International Workshop on Implementation of Functional Languages*, September 7th – 10th 1999.

[J+98] Simon Peyton Jones et al. Haskell 98 report. Technical report, http://www.haskell.org, 1998.

[JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 January 1996.

[MW97] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 136–149, Amsterdam, The Netherlands, 9–11 June 1997.

[PTL00] R. Pointon, P. Trinder, and H-W. Loidl. The Design and Implementation of Glasgow distributed Haskell. In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, number AIB-00-7 in Aachener Informatik Berichte, pages 101–116. RWTH Aachen, 2000.

[THM+96] 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.

[TLK96] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 278–298, Pisa, Italy, 26–29 August 1996. Springer-Verlag.