Erlang-Style Distributed Haskell

Frank Huch

Lehrstuhl für Informatik II, RWTH Aachen,
52056 Aachen, Germany
huch@i2.informatik.rwth-aachen.de
Phone: (+49/0) 241 80 21211

Abstract. We present an extension of the lazy functional programming language Haskell for distributed programming. This extension is motivated by the untyped eager functional language Erlang, which is successfully used in the development of distributed systems. The communication between processes is based on asynchronous message passing and a convenient access to the received messages with pattern matching. Communication between concurrent and distributed processes is programmed with the same functions, which makes it easy to implement distributed systems.
The implementation consists of a library, which extends the IO monad by a state for the mailbox of a process and its process identifier. Furthermore it provides functions for generating new processes and communication between concurrent and distributed processes. The second part of the implementation is a preprocessor, which translates the extended syntax of Distributed Haskell into pure Haskell using the library.

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 distribute character:** The application is distributed. Examples are 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 recourses:** 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 application increases. In particular more and more applications have an inherent distribute 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 [7] with features for elegant distributed programming.

A comparison of different approaches in Section 3 shows the advantages of the concept of Erlang which is introduced in Section 2. We extend Haskell with an Erlang-style process concept, called Distributed Haskell in Section 4 and present the Implementation in Section 5. We conclude and discuss future work in Section 6.

2 Erlang

Erlang [1] is a functional programming language developed by Ericsson. The reduction strategy is call-by-value and Erlang is untyped. Erlang has additional features for concurrent and distributed programming. During the execution of an Erlang program many processes run concurrently. Processes can communicate with each other via asynchronous message passing. Therefore every process is identified by a unique process identifier (*pid*), to which messages can be send: $pid\,!\,v$. Every process has a mailbox, in which all incoming messages are stored. Erlang provides a convenient access for a process to the elements of its mailbox with pattern matching. With the statement

$\texttt{receive}\ pat_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}pat_n\texttt{->}e_n\,\texttt{end}$

the elements of the mailbox of a process are stepwise matched against the patterns $pat_1, \ldots, pat_n$. The first element of the mailbox that matches a pattern pat_i with a substitution ρ is removed from the mailbox. The whole receive statement reduces to $\rho(e_i)$. If no value matches a pattern, the process suspends until a matching value is send to the process.

This access to the elements of the mailbox with pattern matching makes concurrent programming very easy in Erlang. The mailbox is not just a queue. You can pick out the messages of the mailbox your are interested in. All other messages are stored in the mailbox and can be handled later on.

Developing robust distributed applications it is also necessary to program with timeouts. Therefore, Erlang provides an after statement in the receive statement:

$\texttt{receive}\ pat_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}pat_n\texttt{->}e_n\ \texttt{after}\ t\texttt{->}e\ \texttt{end}$

The integer t specifies the time in milliseconds the receive statement tries to match incoming messages against the patterns. After this time it proceeds with e. If t is zero, the patterns are only matched ones against the values in the queue and the receive statement does not suspend.

New processes can be created with the statement

$\texttt{spawn(}m\texttt{,}f\texttt{,[}v_1,\ldots,v_n\texttt{]).}$

The reduction of the newly created process starts with the term $m : f(v_1, \ldots, v_n)$, where f is an exported function of the module m and the arguments $v_1, \ldots, v_n$ are reduced values. The functional result of the `spawn`-statement is the pid of the created process. This can be stored in data-types or send to other processes. Every process can also access his own pid, with the function `self()`.

As an example for a concurrent Erlang program we consider the following database:

```
-module(database).
-export(start/0).

start() -> database([]).

database(L) ->
  receive
    {allocate,Key,P} -> case lookup(Key,L) of
                          fail -> P!free,
                                  receive
                                    {value,V,P} ->
                                      database([{Key,V}|L])
                                  end;
                          {succ,V} -> P!allocated,
                                      database(L)
                        end;
    {lookup,Key,P} -> P!lookup(Key,L),
                      database(L)
  end.
```

Erlang is untyped and data structures can be built as lists (notated as in PROLOG) or tuples (with currely brackets). Furthermore we can use atoms, starting with a lowercase letter, as constructors with arity zero. Variables in Erlang start, as in PROLOG, with an uppercase letter.

Using this module, database processes can be started by calling or spawning the function `start`. The initial interface of a database process are messages consisting of a triple with the flags `allocate` or `lookup`, a key and the pid of the requesting process. When an allocation message is received, the key is looked up in the database. If the key is already allocated the message `allocated` is returned to the requesting process and the database proceeds in its initial state. Otherwise it answers that the requested key is free and waits for the corresponding value, to be stored.

The main point of the implementation of the database is that we guarantee mutual exclusion, when allocating a new key. Otherwise entries could get lost or we would run into other problems, if two clients try to allocate the same key. To guarantee mutual exclusion we use two different mechanisms of Erlang. First, we proceed in another receive statement, which only considers messages with the flag `value`. We do not react on other messages like `allocate`. These messages are stored in the mailbox of the database process. They can be ignored at this point and considered later. Second, we check, if the pid send together with the message is the same as the pid in the allocation message. Therefore, we use the dynamic character of pattern matching and the write once variables of Erlang. In the pattern matching against the pattern `{value,V,P}`, the variable `P` is not newly introduced. `P` is already bound to a value (a pid). The matching only succeeds if a triple is received, which consists of the atom `value`, any value and the same pid, as received before. So values with other pids in the third component are ignored. In a fault tolerant system we would extend this receive statement with an after annotation to prevent a deadlock when a client crashes after allocating a key. And also additional synchronization messages to guarantee the execution of updates. To keep the example small these extensions are not presented here.

For distributed programming Erlang uses the same process concept as for concurrent programming. In a distributed Erlang system multiple nodes exist on maybe multiple computers in the Internet. On an Erlang node multiple processes can be executed concurrently. To distinguish nodes located on the same computer, every node gets a name when it is started. Communication between processes executed on different nodes stays the same as before. Therefore the pid of a process also contains the node on which it is executed.

The only difference in distributed programming are the creation of processes on other nodes and the communication between two independently started Erlang processes. Erlang has an extended `spawn`-statement

$\texttt{spawn(}n\texttt{,}m\texttt{,}f\texttt{,[}v_1,\ldots,v_n\texttt{])}$

with the same behaviour as `spawn`, except that the new process is started on the node n, which is the name of the node followed by the symbol @ and the hostname of the computer the node is located on. With this extension it is very easy to distribute processes, developed in a concurrent environment, e.g to optimize speed of an application or to guarantee fault tolerance to the failure of one node.

Creating processes on different nodes is an expressive mechanism for distributed programming, but some applications are inherent distributed and a hierarchical distribution is not possible. Examples for this are telephony or talk, name-servers or cash dispensers. Here processes are started independently and communication between these processes has to be initiated at runtime. In Erlang this can be realized with global registration of processes on a node. With the statement $\texttt{register(}name\texttt{,}pid\texttt{)}$ the *pid* is globally registered as *name* on the node where it is executed. For the communication between two independently started processes on different nodes we can use the extended send operation $\texttt{\{}name\texttt{,}node\texttt{\}!}v$. The message v is send to the process, which is registered on *node* as *name*. Usually this kind of communication is only chosen for the first contact. Then the pids are exchanged and all further messages are send to pids, as in concurrent programming.

As an example for this kind of communication we modify our database for a distributed version. When a database process is started, it registers itself globally on its node. Additionally, it gets another interface for the connection of remote processes and answers with its own pid.

```
-module(database).
-export(start/0).

start() -> register(database,self()),
           database([]).

database(L) ->
  receive
    {allocate,Key,P} ->
        ...
    {lookup,Key,P} ->
        ...
    {connect,P} -> P!{connect,self()},
                   database(L)
  end.
```

A corresponding client can connect to the database in its initial part and then proceed as before, without any changes of the code.

```
-module(client).
-export(start/1).

start(Node) -> {Node,database}!{connect,self()},
               receive
                 {connect,DB} -> client(DB)
               end.
```

3 Comparison of Erlang and other Approaches

There have been made some other approaches for extending functional languages for concurrent or distributed programming. We compare the main ones for the extension of Haskell with Erlang:

– **Concurrent Haskell**[8] provides methods to start processes inside an application and synchronize them with mutable variables in the IO monad. On top of them it also provides semaphores and asynchronous channels for message passing. Extending this concept for distributing Haskell would lead into implementation problems. In Concurrent Haskell many processes may synchronize on a mutable variable. Distributing processes this synchronization leads to problems, because it is not clear where a mutable variable is located. Furthermore, it is not clear where the processes waiting for the mutable variable to be set are located and how they can be reactivated. Implementing fault tolerant systems, it is also necessary that parts of the system may terminate or even crash down, without effecting the rest of the system. But how can this be guaranteed if the object for synchronization is located on a terminating or crashing node? Hence, it is better to restrict the readers of a synchronizing object, like a mutable variable, to only one reader. Another problem of the implementation is that distributed garbage collection is needed to guarantee that processes run for ever without termination. But how does the system know that there is no one else in the net who wants to read or write a synchronizing object? In Erlang the synchronizing objects are the mailboxes and only one processes can read a mailbox. Hence, if this process terminates, its mailbox gets garbage.

For practical use mutable variables are a much to low concept and programmers choose features built on top of them, like semaphores and asynchronous message channels. In reactive systems it is necessary, that a process can react on different messages, from different partners. Concurrent Haskell only allows a process to wait for a message on one channel. Therefore every message a process may react on has to be send via the same channel. But after receiving a message the state of the process changes and the messages he may react on change. An example therefore is the database process. After a client has allocated a key, the database may not react to new requests anymore to guarantee mutual exclusion. In Concurrent Haskell this can only be implemented using another channel, because otherwise requests from other clients would block the database. Both processes have to know these channels and programming gets more complex, than in Erlang, where uninteresting messages can be ignored and are buffered in the mailbox. So for every group of communication partners own channels are created in Concurrent Haskell and distribution gets hard.

– **Goffin**[4] 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.

Ports in Goffin can be compared with channels in Concurrent Haskell. If a port is made public to the network, every process can send messages to this port and read from this port. This may cause security problems for distributed application and an implementation has the problem that all suspended processes waiting for a message in the port have to be awaken. As described for distributing Concurrent Haskell before, this can also 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. In Erlang these problems do not exist, because the mailboxes are located, where they can only be read.

For reacting on multiple ports Goffin provides a fair `merge :: Port a -> Port a -> Port a` for ports. So a process can wait for different messages of different ports and branch in dependence of the incoming message. But the ports must have the same type. This can lead to typing problems implementing distributed systems. See Section 4.

– **Curry**[6] is an functional-logic programming language, which extends Haskell with needed narrowing and encapsulated search. For distributed programming Curry adds ports with only one reader, but the user can also send logical variables, which results in the same problems described for Concurrent Haskell, because multiple readers are allowed. Communication in Curry is like in Goffin a constraint, which has to be solved. They do not use the `IO` monad and communication via lazy streams. Hence, a programmer can get problems with laziness implementing a distributed system and has to add strictness annotations.

– **Eden**[3] 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.

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, what can lead to the same problems with the type system as `merge` in Goffin. Erlang is untyped. The merge is automatically implemented, because all messages are stored in the same mailbox. Messages on which a process reacts are defined with pattern matching. This has a more dynamic character than suspending on channels in Eden. Pattern matching is not restricted to a finite set of values. An Erlang process can for example suspend until a number greater than an arbitrary value n is received. All other values are buffered in the mailbox. In Eden a process can only suspend on one or a fixed number of channels independently of the values that are received.

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

Hence, Erlang is a very good programming language for implementing distributed systems. Also practice shows this because many application have been implemented using Erlang.

The main disadvantage of Erlang is that it is untyped. As an example, a typing error in an Erlang message usually does not yield a runtime error, but a deadlock. Two approaches have been made for typed Erlang [9, 2]. But they were not very successfully and untyped Erlang is still standard. Especially the communication between processes and the mailboxes stay untyped.

As an advantage the other approaches are based on Haskell and its type system. Especially in the development of the sequential parts of an application this yields more safety in program development. Extending Haskell with the process concept of Erlang we get two advantages. Distributed systems can be developed as easy as in Erlang and the type system of Haskell makes program development safer than in Erlang.

4 Distributed Haskell

We want to extend Haskell by a process concept in the Erlang style. Every process has a mailbox and is identified by a pid, with which messages can be send to the process. Convenient access to the messages in the mailbox should be provided by patter matching. For distributed communication between independent processes on different nodes a global registration and message passing to them should be provided. On the other hand, we want to use a typed language for more safety in program development.

4.1 Process Concept

The main problem of extending Haskell with the Erlang communication model is that communication in Erlang is untyped and has to be typed in Haskell. In a first approach you would try to define an algebraic data-type for all messages a process exchanges with others. The different constructors can be compared with the atoms used as flags in Erlang. But consider two different client processes communicating with the database. Using algebraic data-types for the communication, both clients have to use the same data-type, because both send messages to the database and receive messages from it. But in other parts of their behaviour they could need different messages for communication. So why should one client know all the messages another client uses. Even the database would have to know all these messages. What we need is sub-typing. In our context this means, that the messages two processes use to communicate with each other have to be part of all there messages. For the communication with other processes other messages are allowed.

The data-type definition for the messages the process can communicate with are defined with

$$\texttt{messages}\ C_1 t_{11} \ldots t_{1n_1}\ \ |\ \ \ldots\ \ |\ \ C_m t_{m1} \ldots t_{mn_m}$$

analogously to the definition of an algebraic data-type. C_i are constructor functions of type t_{i1} `->` ... `->` t_{in_i} `-> Msg` and `Msg` is the type of the message of a process.

For the definition of processes we introduce the following notation, similar to the definition of modules in Haskell:

`process` *procName* (*functions to start the process*) `where`

The functions a process can be started with have to be of type `IO()`.

New processes can be created with the function

```
spawn :: IO () -> IO Pid ,
```

where `Pid` is the predefined type of process identifiers. Messages in Distributed Haskell are send to these process identifiers, like in Erlang. Therefore, the user can use the function

```
(<!>) :: Pid -> a -> IO a .
```

The send value is also returned and can be used for further calculations. For practical reasons, especially considering distribution, communication in Distributed Haskell is strict.

For receiving values we use pattern matching and the same semantics as in Erlang:

```
receive                          receiveafter t e
   p1 | g1 -> e1                    p1 | g1 -> e1
        ⋮                                ⋮
   pn | gn -> en                    pn | gn -> en
```

The syntax and typing conventions are the same as for the `case` statement. The `receive` statement blocks a process until a message is received, which matches one of the patterns. The first combination of pattern and message is chosen: the message is removed from the mailbox and the corresponding right-hand side is executed. For timeouts Distributed Haskell also provides the `receiveafter` statement which behaves like `receive` except that after t milliseconds the action e is performed. Like in Erlang, the patterns are matched against the values in the mailbox at least once, even if t is zero.

In Erlang the pid of a process itself can be obtained with the function `self`. This is not possible in Haskell, because the result depends on the process in which it is called. Instead we provide a function `self :: IO Pid` with which the own pid can be read.

Now we can define a Distributed Haskell process, which behaves like the database defined in Erlang. A corresponding client is presented in Appendix A.

```
process Database (start) where

messages = Alloc String Pid | Value String Pid |
           Lookup String Pid | Allocated | Free |
           IsValue (Maybe String)

start :: IO ()
start = database []

database :: [(String,String)] -> IO ()
database l = receive
                Alloc key pid ->
                  case lookup key l of
                    Nothing -> do
                      pid <!> Free
                      receive
                         Value v pid' | pid'==pid ->
                           database ((key,v):l)

                    Just _ -> do pid <!> Allocated
                                 database l

                Lookup key pid -> do
                     pid <!> (IsValue (lookup key l))
                     database l
```

4.2 Distributed Communication

For distributed communication we want to use the same primitives as for concurrent programming. The node on which a process is located is encoded in the pid and the function `<!>` can be used in the same way. But how can processes connect to each other, that are executed on different nodes? In Erlang processes can be globally registered on a node and messages can be send to such a registered process. We provide this mechanism too. The function

```
register :: String -> IO ()
```

registers a process on a node. Its name is given by the string. Old registrations with the same string are overwritten.

Values can be send to registered processes with the function

```
(<!!>) :: (Hostname,String) -> a -> IO a
```

which behaves analogously to `<!>`, except that a hostname and a registered name are used instead of a pid. In this first implementation we do not allow multiple nodes on the same computer and can omit the node-name. In later implementations this will be added.

In our example a client process could connect to a remote database in the following way:

```
process Client (start) where

messages = ... | Connect Pid

start :: IO ()
start = do
  me <- self
  ("server.rwth-aachen.de","database") <!!> (Connect me)
  receive
    Connect db -> client db
```

The database has to register itself with the name "database" when it is started and needs an additional interface in the main receive statement for the connection of a remote process:

```
Connect pid -> do
     me <- self
     pid <!> (Connect me)
     database l
```

Every other communication can be left unchanged.

In Erlang it is also possible to start processes on other nodes. We would like to have a function `remoteSpawn :: String -> IO () -> IO Pid` , for starting processes at a remote node, but this is not possible in Distributed Haskell yet, because of problems in the implementation. If this is needed in an application it is possible to start a process, which creates processes for a fixed set of functions, when corresponding messages are send to it.

5 Implementation

5.1 The Module Distributed

We want to model an Erlang like programming language. First, we have to model the concurrent processes. This can be most easily done on the top of Concurrent Haskell. Every process running in Distributed Haskell is a process in Concurrent Haskell. In Distributed Haskell, processes additionally have a mailbox and the constant self, to access the own pid of a process. Therefore, we define a variation of the `IO` monad, that is extended with a state, in which information can be stored.

```
newtype Dist a b = Dist (a -> IO (a,b))

instance Monad (Dist a) where

  -- return :: b -> Dist a b
  return x = Dist (\v -> return (v,x))

  -- >>= :: Dist a b -> (c -> Dist a c) -> Dist a c)
  (Dist g) >>= f = Dist (\v1 -> do (v2,x) <- (g v1)
                                   let Dist h = f x
                                   h v2)
```

The main function of a Haskell program has to be of Type `IO ()`. So we need a function to start our monad in the `IO` monad, functions for reading and writing the state and a function for casting the functions for the `IO` monad into our monad.

```
runDist :: a -> Dist a b -> IO b
runDist self (Dist f) = do (_,v) <- f self
                           return v
getValue :: Dist a a
getValue = Dist (\v -> return (v,v))

putValue :: a -> Dist a a
putValue v = Dist (\v1 -> return (v,v))

proc :: IO b -> Dist a b
proc f = Dist (\v -> do x <- f
                        return (v,x))
```

Every process in Distributed Haskell will run as an instance of this monad, where the mailbox and the pid of the process are stored in the state. The type of the messages it can receive is fixed for every process. Therefore, we can model the mailbox as a list of messages in the state of our monad. The communication between two processes on one node can be implemented most easily with asynchronous communication over channels in Concurrent Haskell. The state of every process also contains a channel for incoming messages.

```
data DIOState msgType = DIOST {selfPid :: Pid,
                                mailbox :: [msgType],
                                inCh :: StringChan}

type DIO msgType a = Dist (DIOState msgType) a
```

One of the main ideas of Distributed Haskell is, that messages can be send to pids, independently of the node a process is executed on. Hence, we encode the IP-address in the pids. We also want to allow Pids as parts of messages. To simulate sub-typing, we communicate via strings and every message must be an instance of the classes `Read` and `Show`. Channels cannot be converted into strings. Hence, we use another representation for pids, natural numbers. Whenever a process is spawned a new number is generated. Pids can be defined as follows:

```
newType Pid = Pid IPAddr Int    deriving (Show, Read, Eq)
```

The instance of the class `Eq` is also needed, because it can be useful to check, if two pids are identical.

For sending messages to a process, every spawned process gets a channel in its state, when it is created. The conjunction between the pids with the channels is a global information and cannot be stored in the state of our monad `DIO`. Instead we create a server process, when starting Distributed Haskell, as a database for all these conjunctions. The server can be accessed over fixed message channels. These channels have to be known in every process and we add them to our state record.

When a message is send to a pid on the same node, the function `<!>` looks up the corresponding channel in the database and the string representation of the value is send to this channel. If it is send to a remote node, its string representation is send to a socket at the corresponding IP-address.

Receiving a value is a bit more complicated. We want to use pattern matching for the access to the messages in the mailbox. In Haskell pattern matching can only be used in function definitions or `case`-statements. To model pattern matching for receiving values, `receive` takes a function that estimates the values of the mailbox. This function can be defined with pattern matching in the lambda abstraction of a case statement. To express, that a value of the mailbox matches or not, we use the type `Maybe`:

```
receive :: (Read a, Show a) =>
           (a -> Maybe (DIO a b)) -> DIO a b
```

The values of the mailbox are sequently estimated with the function. If this yields `Just` e the corresponding message is removed from the mailbox and e is the result of receive. This works, even if e does not terminate, because of laziness. If the result is `Nothing` the next value is estimated. When there are no other values in the mailbox, `receive` waits for new messages on its channel and is suspended in Concurrent Haskell until a message is send to it. How the `receive` statement of Distributed Haskell can be translated into an application of the `receive` function will be explained in the next section.

The library also provides a function

```
receiveafter :: (Read a, Show a) => Int -> DIO a b ->
                (a -> Maybe (DIO a b)) -> DIO a b
```

for programming with timeouts. The application `receiveafter` $t\ e\ f$ is analogously defined to `receive` f, except that after the time t the action e is performed. Like in Erlang, the patterns are matched against the values in the mailbox at least once, even if t is zero.

For the communication between distributed processes that do not know each other, we have also added the mechanism of registered processes. This is implemented accordingly to the conjunction between pids and the underlying channels. A database with all registered processes and the corresponding pids operates concurrently and can be accessed via channels which are part of the internal state (`DIOState`) of our monad `DIO`.

For distributed programming a message is send to the socket of a remote computer. To receive these messages too, we always have to listen to this socket. This is implemented by a concurrent processes, that forwards the messages received on the socket to their target process. This process must not be known in every process because it only sends messages to them and not vice versa. It only needs access to the database process yielding the message channel corresponding to a pid and the database process for all registered processes.

5.2 Translation into Haskell

In this section we describe how a Distributed Haskell program can be translated into a pure Haskell program, which uses the functions of the module Distributed, as described above.

First, we have to think about how we can model the messages of a process in Haskell. Every process communicates with a subset of all messages and different processes use the same messages. One way of implementing this would be to collect all messages and put them together in one data-type. Every process would work on this data-type. But then it would not be possible to detect communication errors anymore, because every process can receive any message and testing, if the messages can be accepted would be difficult. Instead we want to restrict the messages a process receives to the defined messages and yield a runtime error, when another message is received. A way to model this is using the module system of Haskell. Data-types, that are defined in a module, but not exported, are only known inside the module. Hence, we can model a process as a module and the allowed messages as an algebraic data-type in the module. This data-type will also be the type of the messages in the mailbox. But two different processes have different message types. At runtime these processes have to communicate with each other and the type of the messages has to be changed. We implement this runtime type system by converting messages to strings with the method `show`. On the other side the message is converted into the message type of the receiving process with the function `read`. So, if the receiving process does not accept this message, we get a runtime error showing the type-error.

This approach restricts communication to first order. But for many practical application this is sufficient and considering distributed communication, an implementation of sending functions will be very difficult.

The translation of the head of the process definition for the database is:

```
module Main (main) where

import Distributed

data Msg = Alloc String Pid | Value String Pid | ...
  deriving (Show, Read)
```

Now we can extend the type `DIO` which is used in the Distributed Haskell program with the type `Msg`. This parameter is needed, because the mailbox, which is part of the state of every process, is typed.

The receive statement is implemented as a higher order function, which takes an estimation function for the elements of the mailbox as its parameter. The receive statement can be translated into this receive function as follows:

$$
\begin{array}{lcl}
\texttt{receive} & & \texttt{receive (\textbackslash v -> case v of} \\
\quad p_1 \mid g_1 \texttt{ -> } e_1 & & \quad p_1 \mid g_1 \texttt{ -> Just (}e_1\texttt{)} \\
\quad\quad \vdots & \longmapsto & \quad\quad \vdots \\
\quad p_n \mid g_n \texttt{ -> } e_n & & \quad p_n \mid g_n \texttt{ -> Just (}e_n\texttt{)} \\
 & & \quad \_ \quad\;\; \texttt{-> Nothing)}
\end{array}
$$

where `v` does not occur free in p_i, g_i and e_i for all $i \in \{1, \ldots, n\}$. The same translation is used for `receiveafter`.

Finally the functions of type `IO` have to be casted in our monad `DIO msg`. Therefore, we add the function `proc`. The complete Translation of the database process programmed in Distributed Haskell into pure Haskell is:

```
module Main (main) where

import Distributed

data Msg = Alloc String Pid | Value String Pid |
           Lookup String Pid | Allocated | Free |
           IsValue (Maybe String) | Connect Pid
  deriving (Show, Read)

main :: IO ()
main = start (do register "database"
                 database [])
```

```
database :: [(String,String)] -> DIO Msg ()
database l =
  receive (\v -> case v of
    Alloc key pid -> Just $
      case lookup key l of
        Nothing -> do
          pid <!> Free
          receive (\v' -> case v' of
            Value v pid' | pid'==pid -> Just $
              do proc (putStrLn ("New entry : "++key))
                 database ((key,v):l)
            _ -> Nothing)
        Just _ -> do pid <!> Allocated
                     database l
    Lookup key pid -> Just $
      do proc (putStrLn ("Lookup : "++key))
         pid <!> (IsValue (lookup key l))
         database l
    Connect pid -> Just $
      do me <- self
         pid <!> (Connect me)
         database l
    _ -> Nothing)
```

6 Conclusion and Future Work

We have extended Haskell for concurrent and distributed programming. The process concept is motivated by Erlang. How easy distributed systems can be developed using Distribute Haskell is demonstrated with the example of a database process. The implementation of Distributed Haskell is separated in a library for functions for distributed programming and a preprocessor, which translates Distributed Haskell programs into pure Haskell programs using this library.

The presented work is still in progress. All described extensions are implemented using the Glasgow Haskell compiler [5]. At the moment we are extending the implementation with nodes. This is a usefull feature because then distributed systems can by implemented on one computer with different shells. This is needed because all input/output of one node is performed in one shell. Hence, for example the development of talk on one computer would be impossible only having maximally one node per computer. In future work we want to add mechanisms for robust programming, like linking of processes in Erlang and the creation of processes on remote nodes (`remoteSpawn`).

Messages in Distributed Haskell are not statically typed. We have a runtime type system. This is better, than having no type system like in Erlang, but we want to investigate what can be checked at compile time using a type system with subtypes.

References

1. J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
2. Thomas Arts and Joe Armstrong. A practical type system for Erlang. Technical report, Erlang User Conference, September 1998.
3. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Pena-Mari. Eden — the paradise of functional concurrent programming. *LNCS*, 1123:710ff., 1996.
4. 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.
5. The Glasgow Haskell compiler. http://www.haskell.org/ghc/.
6. 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.
7. Simon Peyton Jones et al. Haskell 98 report. Technical report, `http://www.haskell.org`, 1998.
8. 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.
9. 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.

Appendix

A Client process in Distributed Haskell

```
process Client (start) where

messages = Alloc String Pid | Value String Pid |
           Lookup String Pid | Allocated | Free |
           IsValue (Maybe String) | Connect Pid

start :: IO ()
start = do
  me <- self
  ("server.rwth-aachen.de","database") <!!> (Connect me)
  receive
    Connect db -> client db
```

```haskell
client :: Pid -> IO ()
client pid = do putStr "(i)nsert / (l)ookup > "
                str <- getLine
                me <- self
                case str of
                  "i" -> do
                    putStr "Key > "
                    key <- getLine
                    pid <!> (Alloc key me)
                    receive
                      Free -> do
                         putStr "Value > "
                         v <- getLine
                         pid <!> (Value v me)
                         client pid

                      Allocated -> do
                         putStrLn "Key is allocated > "
                         client pid

                  "l" -> do
                    putStr "Key >"
                    key <- getLine
                    pid <!> (Lookup key me)
                    receive
                      IsValue mStr -> do
                        putStr "Value > "
                        print mStr
                        client pid

                  _ -> client pid
```