A Debugger for Concurrent Haskell

Thomas Böttcher[1] and Frank Huch[2]

[1] RWTH Aachen, 52056 Aachen, Germany
[2] Christian-Albrechts-University of Kiel, 24118 Kiel, Germany

Abstract. We present a debugger for Concurrent Haskell. It is defined as a library `ConcurrentDebug` with the same interface as the library `Concurrent`. Calling a function of `ConcurrentDebug`, the corresponding function of Concurrent Haskell is executed and additional output for debugging is produced. For convenient debugging the output is visualized in a graphical user interface. Here the states of actual threads and communication abstractions are displayed. For finding deadlocks, the user can also determine the scheduling of the threads. Hence, it is also possible to test a program with respect to other schedules which could for example be possible on other computers.

1 Introduction

Debugging is a common technique for finding errors of programs. Beside errors of sequential systems, in concurrent and distributed systems additional kinds of errors can occur. The most known problems are deadlocks, lifelocks, and mutual exclusion. For the analyzation of these problems, standard debuggers are not very useful. They are designed to observe variable bindings during a single execution. However, for errors in concurrent systems the programmer is more interested in the suspension, execution, and communication behavior of threads. Additionally, concurrent systems can behave non-deterministically. For example, only some possible schedule may result in a deadlock. Therefore, influencing the scheduler should also be possible in a debugger for concurrent systems.

Usually, the implementation of a debugger requires much work in extending existing compilers, interpreters or abstract machines. We developed an alternative approach for Concurrent Haskell, a concurrent extension of the purely functional programming language Haskell. In Concurrent Haskell all functions and data types for concurrency and communication are defined in the module `Concurrent`. Our debugger simply redefines these functions with additional input/output for debugging. In combination with a graphical user interface, it is even possible to influence the scheduling of executed threads.

The paper gives an overview of the design of the debugger.

2 Concurrent Haskell

Concurrency is a useful feature for many real world applications. Examples are graphical user interfaces or web servers. Such reactive systmes (have to) interact with multiple interfaces at the same time. Therefore, a programming language should support concurrency for "simultanous" serving of requests.

The lazy functional programming language Haskell 98 [6] does not support concurrency. Therefore, the extension Concurrent Haskell [7] was proposed. Concurrent Haskell extends monadic I/O primitives of Haskell 98 with a thread concept and inter-thread communication using shared variables.

Concurrent Haskell is implemented within the Glasgow Haskell Compiler [2]. This section will give a short introduction to the possibilities and usage of Concurrent Haskell.

2.1 Threads

The first extension is a thread concept. Multiple threads can be executed concurrently, where one thread can be seen as the execution of one Haskell program. With the function

```
forkIO :: IO () -> IO ThreadId
```

new threads are started. The newly created thread starts with the execution of the argument of `forkIO`. On the creating side, `forkIO` is an `IO` operation, which yields a unique thread identifier of the created thread. This can be used to kill the thread with

```
killThread :: ThreadId -> IO ()
```

All threads are executed concurrently. Their I/O actions are interleaved in an unspecified way. A thread can finish itself by performing no more actions.

With the primitive

```
threadDelay :: Int -> IO ()
```

it is possible to suspend a thread for a specified time in microseconds.

2.2 Communication

For communication between threads Concurrent Haskell provides mutable variables (`MVar`). An `MVar` can either contain a value of a specified type or be empty.

```
data MVar a  -- abstract
newEmptyMVar :: IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
```

The action `newEmptyMVar` creates a new mutual variable which is actually empty. `putMVar` fills an empty `MVar` with a value. If the `MVar` is full, then the thread performing `putMVar` is supended until the `MVar` is empty again[1]. `takeMVar` reads the contents of a full `MVar` leaving it empty. Similarly to `putMVar`, the thread

[1] This semantics has changes. Originally, `putMVar` threw an exception in this case.

executing `takeMVar` is suspended until another thread writes a value to it, if the `MVar` is already empty. In the case, that multiple threads suspend on the same `MVar`, one thread will be chosen to continue at random, when the `MVar` is empty (`putMVar`) respectively full (`takeMVar`).

The library Concurrent implements more actions on `MVar`s such as testing for emptiness, swapping the contents, or just reading without emtying the `MVar`.

`MVar`s are the base for other, more complex communication abstractions: different extensions of `MVar`s, two kinds of semaphores, and channels. The most important ones are channels. A channel can store several messages of a specified type and can be accessed by multiple readers and writers:

```
data Chan a
newChan   :: IO (Chan a)          -- abstract
readChan  :: Chan a -> IO a
writeChan :: Chan a -> a -> IO ()
```

The action `newChan` generates a new empty channel. `readChan` reads the oldest value from a channel and eliminates it. If the channel is empty, `readChan` suspends until another thread writes a new value to the channel with `writeChan`. Again in the case of multiple readers, one thread is selected at random. The main difference to `MVar`s is that the writing thread can continue directly, even if the channel is not empty. Furthermore, the elements in the channel are read in the same order as they are put in the channel while threads which want to put a value in an `MVar` may overtake threads which tried to perform the `putMVar` action earlier. Internally, channels are efficiently implemented as a stream of `MVar`s, with pointers to the beginning and the end of the stream for reading and writing a channel in constant time, independent of the number of its elements.

The library Concurrent contains further actions to test a channel for emptiness or to write a value at the front of a channel. For more details see [5, 7].

3 Motivation

Writing a program is not simple. If the programmer has found and corrected all the errors the compiler reported, then everything seems to work fine. But then the program (suddenly) does something unexpected. It may give a wrong result, crash with a runtime error, or loop. In most cases the programmer is glad to have a debugger which helps him finding errors.

This section describes the possibilities of debugging Haskell. It will show why there is a requirement for a debugger especially designed for Concurrent Haskell.

3.1 Debugging Haskell

An overview of debuggers for Haskell is give by [1]. The authors present three systems: *Freja* [8], *Hat* [9][2], and *HOOD* [3]. These help programmers finding a bug

[2] At first Hat was called *Redex Trail System*

within their Haskell programs. Unfortunately, Freja and Hat do not support full Haskell 98 standard. In particular, monadic I/O is not (completely) supported. Therefore, these systems cannot be used for debugging Concurrent Haskell programs. Only Hood [3] provides the possibility to observe data-structures within a program with monadic I/O. In Hood the programmer may annotate expressions and obtains (with respect to there evaluation) the represented values. Hence, it could be possible to show different contents of an `MVar` during the execution.

However, for analyzing e.g. a deadlock this does not help. The value in an `MVar` is not the reason for the deadlock. The reason is the missing of a value in an `MVar` and this cannot be visualized with Hood. Furthermore, Hood is only designed to display normal-forms of values, but not values of `MVar`s with respect to time. The order in which the contents of an `MVar` are displayed depends on the time when its value reduced to head normal form. However, this may be independent of the sequence defined in the `IO` monad. Hence, the debug information of Hood does not help finding errors resulting from the concurrent execution.

Furthermore, it is not possible to influence the scheduler in Hood which is needed to find bugs only related to some special schedules. On the other hand, the additional code generating the debugging output of Hood influences the scheduler. A program with a deadlock might execute deadlock free while debugging with Hood.

3.2 Example: Dining Philosophers

An example for a possible deadlock situation is the dining philosophers problem, a classic synchronization problem. Five philosophers are sitting around a table. Each philosopher is doing his daily work: thinking until he is hungry. Then he eats and afterwards he starts thinking again. For eating, a philosopher needs two chopsticks. There are only five sticks distributed among the philosophers. Each philosopher shares his left stick with his left neighbor and his right stick with his right neighbor. Getting hungry, a philosopher picks up his sticks sequentially. After picking up the sticks, he starts eating. Finishing the dish, the philosopher sequentially puts both sticks back to the table and continues thinking. If a philosopher wants to pick up an occupied stick, then he waits until the neighbor returns this stick. So the philosophers can run into a deadlock, if all philosophers pick up their left stick at the same time.

In Concurrent Haskell the sticks can be implemented by means of `MVar`s. An `MVar` is filled, if the represented stick is laying on the table. It is empty if the philosopher has picked it up. We do not need to store different values in the `MVar`. Therefore, we use `MVar`s of type `()` which only contains the value `()`. Then, a simple philosopher can be implemented as follows:

```
philosopher :: MVar () -> MVar () -> IO ()
philosopher left right = do
    --thinkSomeTime
    takeMVar left
```

```
      takeMVar right
      --eatSomeTime
      putMVar left ()
      putMVar right ()
      philosopher left right

startPhils :: Int -> IO ()
startPhils n = do
   sticks@(firstStick:_) <- mapM (\_ -> newMVar ()) [1..n]
   let distSticks = pair sticks
   mapM_ (\(n,(l,r)) -> forkIO (philosopher n l r)) distSticks
   philosopher n (head sticks) (last sticks)

     where pair :: [a] -> [(a,a)]
           pair [x] = []
           pair (x:xs@(y:_)) = (x,y):pair xs
```

Executing the program in ghc, may crash (maybe after several hours) with the error message `Fail: thread blocked indefinitely`. A deadlock occurred, but the programmer gets no hint where all threads of the program suspend. So far, the only possibility to analyze a deadlock is to add debug output to the program. Then the programmer can try to understand which actions the threads performed, before running into the deadlock. However, in larger applications adding this debug output (and later removing it again) can be costly. A debugger for analyzing the concurrent behavior would be helpful.

This example also shows, that the use of Hood does not help to find the deadlock. The value written to the `MVar`s is always `()` and it is not even desired. Therefore, observing this value will not help to find the deadlock. The critical aspect is the sequence of writing and taking the `MVar`s.

What we need is a tool which provides a visualization of the threads, the used communication-structures, and the actions performed by the threads. Furthermore, the tool should help the programmer to find possible deadlocks, but also lifelocks or other problems of concurrent systems, like the access of non-initialized communication abstractions. The programmer should also be able to directly influence the thread-scheduling to provoke possible bugs.

4 Basic Idea

As described in the previous section, there is need for a debugger for Concurrent Haskell. The basic idea of this work is the extension of the concurrent actions.

4.1 The Simplest Debugger

In a first step we extend the concurrency functions with text messages written to the standard output device. Therefore, we extend all functions with debugging messages in the following way:

```
module ConcurrentDebug { ... } where

import qualified Concurrent as C

forkIO :: IO () -> IO ThreadID
forkIO a = do id <- C.forkIO a
              putStr "New Thread forked\n"
              return id

takeMVar :: MVar a -> IO a
takeMVar m = do putStr "Thread wants to take MVar\n"
                C.takeMVar m
                putStr "Thread took MVar\n"
```

We define a module `ConcurrentDebug` with exactly the same interface as the library `Concurrent`. Every function of `Concurrent` is extended with debugging information. To avoid name conflicts we import `Concurrent` qualified and call it `C`. Beside the debug output, the original functions of the library `Concurrent` are executed.

Using this simplest Debugger in a concurrent application produces much output of performed actions, but the information content for the programmer is scanty. Threads and communication abstractions cannot be distinguished. For more precise debugging information we need additional information for threads, `MVar`s, and channels. In the next section we refine the debugger for a distinction of communication abstractions. For the distinction of threads another technique is needed which we present in Section 4.3.

4.2 Distinguishable Communication Abstractions

To provide unique identifiers for communication objects we extend the consisting types of communication abstractions in the following way:

```
data MVar a = MVar MVarNo (C.MVar a)
newtype MVarNo = MVarNo Integer
```

Values of the new type `MVar` consist of a type-constructor, a number as a unique identifier, and an original `MVar` from the library `Concurrent`.

Whenever a new `MVar` is created, we generate a new unique number for this new `MVar`. But how can we get such a number? We cannot add an additional parameter to the functions `newMVar` and `newEmtyMVar` for a new unique identifier. All functions must have the same type as in the module `Concurrent`. Therefore, we use the technique of global constants for the implementation of the `MVar` counter. The use of such a global state is ugly in the development of applications in purely functional languages. However, this cannot be avoided in the development of a debugger, which is a kind of meta-program working on arbitrary other programs. The same holds for the implementation of the debuggers Hat and Hood [9, 3]. For the creation of debug information side-effects are needed.

The global state is implemented by the definition of a reference to a mutable variable as a global constant. Whenever a new `MVar` is created, then the counter in the global reference is incremented. The actual value is taken as the `MVarNo` of the new `MVar`. In the context of concurrent executions, the use of a global `MVar` instead of a global reference cell is necessary, because we must impede that multiple threads read and write the global counter in an interleaved order. This guarantees that no other thread reads the global state, before another thread reads and increments it. In other words reading and incrementing the global `MVar`-counter build an atomic action.

```
mVarCounter :: C.MVar MVarNo
mVarCounter = unsafePerfomIO (C.newMVar (MVarNo 0))

newEmptyMVar :: IO (MVar a)
newEmptyMVar = do MVarNo c <- C.takeMVar mVarCounter
                  C.putMVar mVarCounter (MVarNo (c+1))
                  mVar <- C.newEmptyMVar
                  putStr ("newMVar :"++(show c)++"\n")
                  return (MVar (MVarNo c) mVar)
```

The use of `unsafePerformIO` in this context seems superfluous because the access to `mVarCounter` is always located in the `IO`-Monad. However, without using `unsafePerformIO` the constant `mVarCounter` would represent the action of creating an `MVar`, which would be performed in every call of `mVarCounter`. We would not obtain a global constant.

With the use of this global `mVarCounter` and the extended datatype `MVar` we can give more expressive debug information:

```
takeMVar :: MVar a -> IO a
takeMVar (MVar no m) = do
    putStr ("Thread wants to take MVar "++(show no)++"\n")
    C.takeMVar m
    putStr ("Thread took MVar "++(show no)++"\n")
```

Communication abstractions can be distinguished and the programmer can track the use of the same communication abstractions. However, she cannot distinguish the threads performing actions which would also be a helpful debugging information. Therefore, we introduce a debugger thread in the next section.

Although in the library `Concurrent` all communication abstractions are implemented on top of MVars, we want to distinguish them in our debugger. A programmer using the debugger does not want to see the implementation details of e.g. a channel. She is only interested in the behavior of the communication abstraction `Chan`. This is reading from, suspending on, and writing to the channel. Additionally, she may be interested in the number of elements in a channel. To provide these information we extend the other communication abstractions in the same manner. We use a separate numbering for every communication abstraction.

4.3 The Debugger Thread

At first sight, Concurrent Haskell already provides unique thread identifiers (`ThreadId`), which we can use for our debugger. Unfortunately, these identifiers cannot be displayed, because no instance of `Show` is defined for them.

The next idea is to use the same extension as for unique communication abstractions for `ThreadId`s. But this approach fails too. During the execution of a Concurrent Haskell thread the own `ThreadId` is not present as a value, like an `MVar`. A thread can always access its own id with the function `myThreadId`. `ThreadId`s are not passed around like `MVar`s. Hence, we cannot define an extended data structure as for communication abstractions. We must find another solution.

Fortunately, we can use concurrent threads in our debugger. We define a data base thread which stores a mapping from `ThreadId`s to thread numbers. Again we use a global constant as a reference (`debugMsgChan`) to this thread. This reference is a channel to which threads send insert and lookup messages of `ThreadId`s and receive the corresponding `ThreadNo` in an MVar.

```
data ThreadIdMsg = Insert ThreadId (MVar Integer)
                 | Lookup ThreadId (MVar Integer)

debugMsgChan :: IO (Chan ThreadIdMsg)
debugMsgChan = unsafePerformIO
                     (do ch <- C.newChan
                         C.forkIO (chd (ThreadNo 0) [])
                         return ch)

chd :: Integer -> [(ThreadId,ThreadNo)] -> IO ()
chd n db = do input <- C.readChan debugMsgChan
              case input of
                (Insert id m) -> do C.putMVar m n
                                    chd (n+1) ((id,n):db)
                (Lookup id m) -> do
                      C.putMVar m (fromJust (lookup id db))
                      chd n db
```

As an interface to this thread we use a channel, because multiple requests to this database thread are possible. In the newest implementation of Concurrent Haskell a `putMVar` action on a full MVar does not crash as in older implementations. The thread performing this action just suspends until the `MVar` is empty. Therefore, it would also be possible to use an `MVar`. However, there is no comment on the fairness of the new implementation of `putMVar`. With the use of a channel this fairness is guaranteed.

To keep the presented code simple, we used a list to implement the database of `ThreadId`s and corresponding thread numbers. In the real implementation we use a balanced tree (`FiniteMap`), which is much more efficient.

With the global database thread the debugger version of forkIO can be defined as:

```
forkIO :: IO() -> IO ThreadId
forkIO t = forkIO (do id <- C.myThreadId
                      answerMVar <- C.newEmptyMVar
                      C.writeChan debugMsgChan
                                  (Insert id answerMVar)
                      C.takeMVar answerMVar
                      t)
```

Note, that we perform the registration of the new thread in the forked thread. This is necessary, because otherwise the forked thread could accesses its own `ThreadNo` before this number is inserted in the database.

With this global thread database we can create more precise debug information:

```
myThreadNo :: IO ThreadNo
myThreadNo = do id <- C.myThreadId
                answerMVar <- C.newEmptyMVar
                C.writeChan debugMsgChan (Lookup id answerMVar)
                C.takeMVar answerMVar

takeMVar :: MVar a -> IO a
takeMVar (MVar no m) = do
    threadNo <- myThreadNo
    putStr ("Thread "++(show threadNo)++
            "wants to take MVar "++(show no)++"\n")
    C.takeMVar m
    putStr ("Thread"++(show threadNo)++
            " took MVar "++(show no)++"\n")
```

5 From Text to GUI

The last section presented the basic idea of our debugger. In this debugger, much output is produced which contains valuable information about the concurrent behavior of a run of the program. However, we produce too much information and the important aspects get lost. As Hat, we could store this information in a file and develop a viewer which allows pretty printing and surfing this information. Unfortunately, influencing the scheduling is not possible in this approach. Therefore, we integrated a comfortable visualization at runtime.

The main idea is the replacement of the output by communication to a thread controlling a graphical user interface (*gui*). In the gui we do not present all actions performed during the execution. Instead we present a snapshot of the actual system: For `MVar`s this is empty or full and for `Chan`s this is the number of elements in the channel. For threads this snapshot is running or suspended on a communication abstraction. Furthermore, we display when a thread writes or reads a communication abstraction. Figure 1 presents a screen shot of the debugger gui. The threads are displayed on the left and the communication

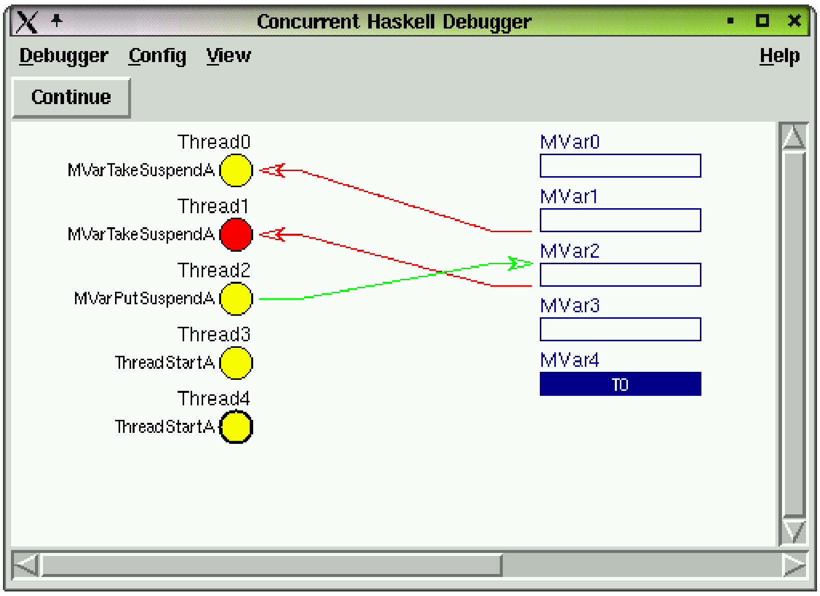

Figure 1: Screenshot of the Debugger

abstraction (here only `MVars`) on the right.

To provide this global view of the system we implemented another thread for the administration work. This debugger thread holds, reads, and changes the global status in a similar way to the `chd` thread discussed in the Section 4.3. It receives status messages for the individual threads/ communication abstractions from the global channel constant `debugMsgChan`. For instance, with the use of this thread instead of printing debug messages the action `takeMVar` is defined as:

```
takeMVar :: MVar a -> IO a
takeMVar (MVar mvarNo mvar) = do
    threadNo <- myThreadId
    C.writeChan debugMsgChan (threadNo,Suspend mvarNo)
    value <- C.takeMVar mvar
    C.writeChan debugMsgChan (threadNo,Read mvarNo)
    return value
```

The messages sent to the `debugMsgChan` represent modifications of the system state. Hence, the debugger thread displays these modifications in the gui.

5.1 Influencing the scheduling

So far, our debugger only displays the behavior of the system. For real debugging of the concurrent system it is necessary to be able to interrupt the execution and influence its scheduling.

Fortunately, this is no problem with introducing a debugger thread in the last section. Whenever a thread performs a concurrent action it sends a message via the `debugMsgChan` to the debugger thread. Here we can easily integrate a synchronization: the thread performing the action does not only inform the gui of the action. Additionally, it expects an acknowledgment. Only after receiving this acknowledgment from the debugger thread the thread continues. Hence, the sending of the acknowledgment can be delayed until the user clicks the visualization of a thread in the gui.

There is a large number of possible break points during the actions. To keep influencing the scheduler manageable, the user can also globally define on which actions the debugger should break the whole system. For the other actions it directly gives acknowledgments to the requesting threads.

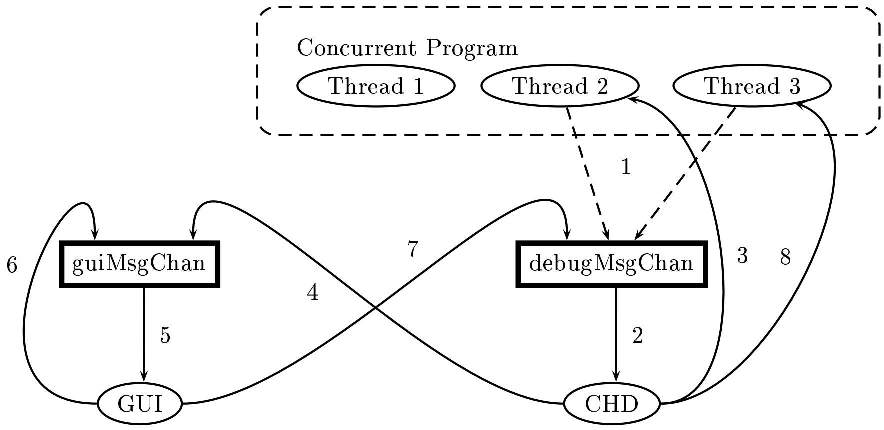

Figure 2: Structure of the Debugger

Figure 2 presentes the global structure of the debugger. The labeled arcs represent a possible execution sequence of exchanged messages:

1. concurrent actions are perfomed
2. chd reads thread message
3. automatic acknowledgment / other actions are blocked
4. modification of system state
5. eventually visualization of modification
6. modification of global configuration/appearance
7. selected thread may continue
8. wake up of blocked actions

In Concurrent Haskell programs communication abstractions are often not used as statically as in the philosophers example. For instance, in many cases an `MVar` is created only for one answer. After that, the `MVar` is not referenced any more and it is collected by the garbage collector. In our debugger we add finalizers to all communication abstractions, which remove them from the gui. Hence, the visualized communication abstractions is restricted to the used ones. Similarly, we remove terminated threads from the gui. This cannot be implemented by a finalizer. Instead a special termination message termination message is sent to the `debugMsgChan`, when a thread terminates or gets killed by another thread. The code for this termination message is appended to the code of each thread when the thread is created by `ConcurrentDebug.forkIO`.

5.2 Library Extensions

For some cases it can also be useful to add breakpoints to threads or to name threads, communication abstractions and values written to communication abstractions. So far, they are only named with numbers. Therefore, we have added the following functions to the library `ConcurrentDebug`:

```
chdBreakThread :: IO ()

labelThread :: String -> IO ()
labelMVar :: MVar a -> String -> IO ()
forkIOLabel :: IO () -> String -> IO ThreadID
putMVarLabel :: String -> MVar a -> a -> IO ()
```

Internally, we extend the data structures for threads, communication abstractions and the values in the communication abstraction by a possible name (`Maybe String`). This string is displayed if it is set. Otherwise, a standard name is generated from the number.

As a consequence of these library extensions, it is not simply possible to get back from the debug-able program to the original program. As a solution we provide a module `ConcurrentNoDebug`, which extends the module `Concurrent` with dummy functions (`return ()`) for these functions.

5.3 Further Features

Another useful information provided by our debugger, is the thread number of the thread which wrote a value into a communication abstraction. The programmer often has the view that messages are passed from one thread to another thread (or maybe a set of threads). If the message stays in an `MVar` or somewhere in a channel for a longer time, then it can be interesting to know, which thread created the message. This provides the understanding of the message-flow in the concurrent program.

Although our debugger visualizes the behavior of a Concurrent Haskell program, the connection between the visualization and the source program can

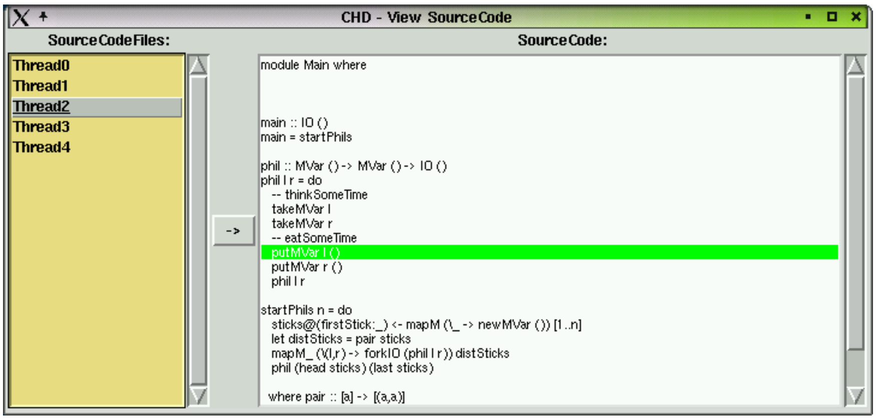

Figure 3: Source Code Presentation corresponding to Figure 1

sometimes be too weak. By means of labeled concurrency functions the programmer can enhance this connection. However, this is an additional effort modifying the code. As a solution, we also provide versions of each concurrency function which also takes a module name/line number as first, additional argument. Furthermore, we implemented a preprocessor which adds module/line information to each concurrency function. Debugging the converted program, it is then possible to locate the executed concurrency functions of each thread in the source code. Hence, the connection between the visualization and the source code in most case becomes comprehensible. Figure 3 presents a screen shot of the source code presentation.

6 Conclusion and Future Work

We developed a debugger for Concurrent Haskell which in contrast to other debuggers is simply a wrapper for the library `Concurrent`. All wrapped functions are extended with additional debugging output visualized by means of a graphical user interface. Adding synchronization to this debugging output it is also possible to influence the scheduling which is necessary to analyze a system in uncommon schedules.

The only larger experience of using our debugger is the implementation of Distributed Haskell [4]. Beside finding deadlocks during the development, it was also possible to detect bad styles of programming: For instance, in some schedules data structures could be accessed before they were initialized by another thread or messages could be fetched by the wrong process. For more practical experience, the debugger libraries can be found and downloaded at `http://www.informatik.uni-kiel.de/~fhu/chd/`.

For future work, we plan to make the visualization more comfortable. For example, tracking messages from one thread through a `Chan` to another thread could be an interesting information. Also other views like message sequence charts could be useful for debugging Concurrent Haskell programs.

In the longer term it would also be interesting, to allow backwards steps in the debugging. This could be possible with the use of configuration files which can be used for restarting the system up to a specified point. Then the user could search bugs in different schedules with backtracking. Furthermore, we want to investigate if our approach could be combined with other Haskell debuggers, like Hat. The concepts are quite different, but for the practical acceptance of Haskell such a debugger is needed. In a concurrent system bugs resulting from the calculation my as well occur as bugs from a badly designed communication protocol. It is not always possible to distinguish these bugs.

References

1. O. Chitil, C. Runciman, and M. Wallace. Freja, hat and hood — A comparative evaluation of three systems for tracing and debugging lazy functional programs. *Lecture Notes in Computer Science*, 2011:176–??, 2001.
2. The Glasgow Haskell compiler. http://www.haskell.org/ghc/.
3. A. Gill. Debugging haskell by observing intermediate data structures. In *Haskell Workshop*, Sept. 2000.
4. F. Huch and U. Norbisrath. Distributed programming in Haskell with ports. *Lecture Notes in Computer Science*, 2011:107–121, 2000.
5. S. P. Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. `http://research.microsoft.com/~simonpj/#marktoberdorf`, Jan. 2001.
6. S. P. Jones et al. Haskell 98 report. Technical report, http://www.haskell.org, 1998.
7. S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 Jan. 1996.
8. H. Nilsson and J. Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4(2):121–150, Apr. 1997.
9. J. Sparud and C. Runciman. Complete and partial redex trails of functional computations. *Lecture Notes in Computer Science*, 1467:160–174, 1998.