

Searching for Deadlocks while Debugging Concurrent Haskell Programs

Jan Christiansen

Frank Huch

Christian-Albrechts-University of Kiel
Institute of Computer Science
Olshausenstr. 40, 24118 Kiel, Germany
{jac,fhu}@informatik.uni-kiel.de

ABSTRACT

This paper presents an approach to searching for deadlocks in Concurrent Haskell programs. The search is based on a redefinition of the IO monad which allows the reversal of Concurrent Haskell's concurrency primitives. Hence, it is possible to implement this search by a backtracking algorithm checking all possible schedules of the system. It is integrated in the Concurrent Haskell Debugger (CHD), and automatically searches for deadlocks in the background while debugging. The tool is easy to use and the small modifications of the source program are done by a preprocessor. In the tool we use iterative deepening as search strategy which quickly detects deadlocks close to the actual system configuration and utilizes idle time during debugging at the best.

Categories and Subject Descriptors

Software [SOFTWARE ENGINEERING]: Testing and Debugging—*Debugging aids*

General Terms

Languages

Keywords

Concurrent Haskell, debugging, deadlock, detecting deadlocks

1. INTRODUCTION

Developing concurrent applications is a difficult task. Beside the bugs a programmer can make in the sequential parts of his program, he can also produce bugs related to concurrency and thread synchronization, like deadlocks, livelocks, or not guaranteed mutual exclusion. One approach to prevent programmers from such bugs is formal verification, like model checking or theorem proving, combined with a formal

system specification and refinement to obtain the executable program. However, in practice, this approach does not scale well for larger concurrent applications. The communication (and synchronization) part of the application may interfere with the computation. Often, it is even the case that concurrency is a means to express algorithms. In these cases, formal verification is often not applicable, e.g. because of infinite state spaces or state space explosion problems. As a consequence, validating an implementation is in many cases restricted to testing.

In this paper, we mainly focus on the detection of deadlocks. During the development of a concurrent application, it is often the case that the execution runs into a deadlock and there is no hint, in which configuration of the program this deadlock occurs. To visualize the deadlock and the schedule causing it, debuggers can be helpful. To also detect deadlocks related to other schedules, a debugger for a concurrent language should also give the possibility to influence the scheduling to detect possible, hidden deadlocks not reached in normal system executions. These hidden deadlocks can become real deadlocks when for instance, the system is executed with a different scheduler or the scheduling is effected by adding further threads to the application. However, debugging is not a good technique for detecting these hidden deadlocks, since it only performs a single schedule (although selectable by the user) like the non-debugged execution. To detect deadlocks in other schedules it would be nice to search in all possible schedules during debugging. Since the state space of the application may be infinite, this search should be restricted in depth and should consider the input/output behavior of the program in a sensible way.

We have implemented this idea of searching for deadlocks for Concurrent Haskell [19], an extension of the pure, lazy functional programming language Haskell 98 [18]. The search is based on a redefinition of the IO monad which allows the reversal of Concurrent Haskell's concurrency primitives. The approach is conveniently integrated into our graphical Concurrent Haskell Debugger (CHD) [2]. The debugger performs the search for deadlocks in the background, without any debugging restrictions for the user. When a deadlock is found the user is guided into the deadlock state. The implementation uses iterative deepening as search strategy. Hence, deadlocks close to the actual configuration are quickly detected and time until the user selects a thread to continue is utilized at the best. To keep the users effort for debugging as small as possible, the tool also provides a preprocessor which performs the necessary modifications of the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'04, September 19–21, 2004, Snowbird, Utah, USA.
Copyright 2004 ACM 1-58113-905-5/04/0009 ...\$5.00.

source program modules.

The paper is organized as follows: in Section 2 we give a short introduction to Concurrent Haskell and introduce previous work on the Concurrent Haskell Debugger in Section 3. Section 4 motivates the search for deadlocks during debugging. In Section 5 we discuss the underlying idea of our approach with interpreting the IO monad as a data structure. Our algorithm for search and an optimization based on partial order reduction is described in Section 6. The integration in CHD is shortly described in Section 7 and the practical applicability of the tool is discussed in Section 8. Finally, we discuss related work in Section 9 and conclude in Section 10.

2. CONCURRENT HASKELL

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

The lazy functional programming language Haskell 98 [18] does not support concurrency. Therefore, the extension Concurrent Haskell [19] was proposed. Concurrent Haskell extends the monadic IO primitives of Haskell 98 with a thread concept and inter-thread communication by means of shared variables. It is implemented within the Glasgow Haskell Compiler (ghc) [7]. 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 each thread can be seen as the execution of one Haskell program. The function

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

creates a new thread which starts with the execution of the argument of `forkIO`. The function `forkIO` itself is an IO action, which yields the unique thread identifier of the created thread. This `ThreadId` can be used to kill the thread by

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

All threads are executed concurrently. Their IO actions are interleaved in a non-deterministic way. A thread terminates by performing no more actions.

2.2 Communication

For inter-thread communication Concurrent Haskell provides different kinds of communication abstractions. The simplest communication abstraction are mutable variables (MVars). 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 initially empty. `putMVar` fills an empty MVar with a

value. If the MVar is already full, then the thread performing `putMVar` is suspended until the MVar is empty again¹. `takeMVar` reads the contents of a full MVar leaving it empty. Similarly to `putMVar`, the thread executing `takeMVar` is suspended, if the MVar is already empty, until another thread writes a value to it. In the case that multiple threads suspend on the same MVar, non-deterministically one thread is chosen to continue, when the MVar is emptied (`putMVar`) respectively filled (`takeMVar`). The implementation guarantees that each of these functions (and all the other functions on communication abstractions) are executed atomically. They cannot be interrupted by the scheduler.

The library `Concurrent` implements more actions on MVars such as testing for emptiness, swapping the contents, or just reading without emptying the MVar. Furthermore, MVars are the base for other, more complex communication abstractions: a slight modification of MVars (`SampleVar`), two kinds of quantity semaphores (`QSem` and `QSemN`), and unbound channels (`Chan`).

A Concurrent Haskell program is stuck in a *deadlock* if all threads are suspended on communication abstractions. If at least one thread is waiting for input from the outside world (e.g. like in a web server), then this is not a deadlock, since the thread may awake the other threads after receiving input. Note, that systems may contain partial deadlocks in which only some threads are waiting for each other. In the remaining paper we do not consider partial deadlocks, since they are very difficult to identify and many situations (e.g. an idle web server) can be seen as a partial deadlock.

2.3 An Example

As a small example for Concurrent Haskell we present an implementation of the well known dining philosophers problem. The sticks the philosophers have to share for eating, are represented by MVars which are filled if the corresponding stick is laying on the table. Hence, taking a stick is implemented by a `takeMVar` action. Since philosophers usually do not communicate, but only synchronize for eating, it is sufficient to use MVars of type `()`. A philosopher can simply be defined by a non-terminating function:

```
phil :: MVar () -> MVar () -> IO ()
phil left right = do -- get hungry
  takeMVar left
  takeMVar right
  -- eat
  putMVar left ()
  putMVar right ()
  -- think
  phil left right
```

3. CONCURRENT HASKELL DEBUGGER

In previous work [2], we developed a debugger specialized for bugs related to concurrency in Concurrent Haskell, called the *Concurrent Haskell Debugger (CHD)*. The basic idea of this debugger, is the extension of all concurrency primitives provided by Concurrent Haskell with debugging output. The debugger consists of a module `ConcurrentDebug` with the same interface as `Concurrent`. Calling a function of `ConcurrentDebug` first performs some debug output. Then the original function of the module `Concurrent` is executed

¹This semantics has changed. Originally, in [19], `putMVar` threw an exception in this case.

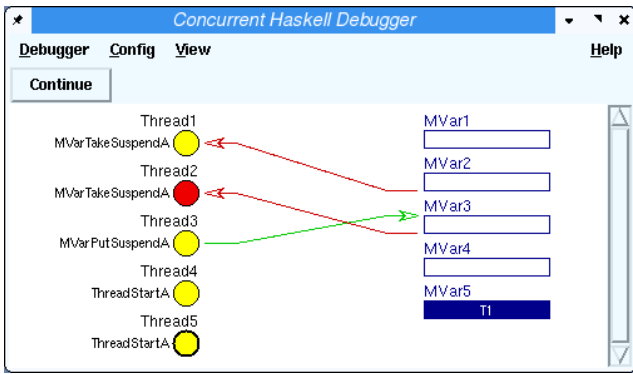


Figure 1: Five dining philosophers

and finally, again, some debug output is performed. The debug output is sent to a graphical user interface, which visualizes all threads, communication abstractions and performed actions. For example, if a thread performs a `putMVar` action, then an arrow from the graphical representation of this thread to the corresponding `MVar` is drawn and the state of the `MVar` switches from empty to filled. The different concurrency actions can easily be distinguished by the orientation and color of the drawn arrows. To observe the flow of messages, CHD also indicates which thread has stored a value in a communication abstraction by a thread identifier. A screenshot of CHD used in the dining philosophers program is presented in Figure 1. In the presented state, only `MVar5` is filled and this value was written by Thread1 (T1). Thread2 is suspended on `MVar2` and Thread1 wants to perform a `takeMVar` action on `MVar2` and will, since `MVar2` is empty, suspend in the next step. Thread3 will put a value into `MVar3` in the next step, which will wake up Thread2.

For debugging a concurrent system, it is not sufficient to just execute it step by step. It is also important to influence the scheduling, since not every schedule may contain a bug (e.g. deadlocks). In CHD the scheduling of the system can easily be influenced by clicking on the thread that is supposed to continue its execution. This is implemented by additional synchronization messages between the extended debug version of the concurrent action and a control thread which also communicates with the graphical user interface. The CHD provides further possibilities for convenient debugging, like naming threads or communication abstractions, configuring the interaction behavior for the visualized objects as well as for special concurrent actions, and showing source code positions of threads. For more details see <http://www.informatik.uni-kiel.de/~fhu/chd>.

4. MOTIVATION

The Concurrent Haskell Debugger is a useful tool if you want to check whether a schedule causes a deadlock. You can check this by just executing your program with exactly that schedule. However, if you do not find a deadlock you nevertheless do not know if there is any. You can also interactively modify the scheduling and check other schedules that way, but you will never check every possible schedule. Perhaps if you just exchange the execution order of two threads you would run into a deadlock. To avoid this undesirable situation it would be necessary to check every possible schedule of a program.

This paper shows how we combine a search for deadlocks with the useful features of the Concurrent Haskell Debugger. The new debugger should allow the user to execute one step of a particular thread just as CHD does. In the background it should also check the program for a deadlock by executing the program with every possible scheduling. This causes the problem of restoring an old state after one computation has been checked. One approach for solving this problem would be a restart of the whole system with serving input actions with recorded input and ignoring output actions. However, this would be very inefficient in combination with searching all possible schedules since the system has to be restarted for every possible schedule. Furthermore, the effort for restarting grows with the progress of the computation. Hence, search will dramatically slow down after some debugging steps.

An alternative would be to retract performed concurrency primitives. Inspecting the concurrency primitives, we see, that this is always possible, as the following examples show:

action	can be retracted by
<code>newEmptyMVar</code>	doing nothing
<code>putMVar m v</code>	<code>takeMVar m</code>
<code>readMVar m</code>	doing nothing
<code>v <- takeMVar m</code>	<code>putMVar m v</code>

To retract `newEmptyMVar` we have to do nothing because the garbage collector will eliminate the created `MVar` for us. Writing to an `MVar` by `putMVar` can be retracted by `takeMVar` on the same `MVar` and discarding the taken value. Reading an `MVar` does not effect the state of the concurrent system. Finally, to retract `takeMVar` we have to put the value, that has been taken out of the `MVar`, back into it. Hence, it is not sufficient to consider only a restore action, but a *restore function* (of type `a -> IO ()`) which takes the result of the *retractable action* (of type `IO a`) and retracts it by means of the *restore action*. Formally, the restore function that corresponds to a retractable action has to fulfill the following property:

$$(\text{retractable action} \gg= \text{restore function}) \simeq \text{return } ()$$

For the discussed concurrency actions we get:

retractable action	restore function
<code>newEmptyMVar</code>	<code>_ -> return()</code>
<code>putMVar m v</code>	<code>_ -> takeMVar m >> return ()</code>
<code>readMVar m</code>	<code>_ -> return()</code>
<code>takeMVar m</code>	<code>\v -> putMVar m v</code>

The creation of a thread could be retracted by killing this thread again. However, we want to control the scheduling during search. Hence, our implementation uses a list of threads and the functions `forkIO` and `killThread` are implemented just in form of adding and removing elements of this list. We will discuss this in more detail in the next section.

With this insight we are able to undo every action that is performed by a concurrent program. So, the checker is able to execute the program with a particular schedule and restore the state before continuing the execution. However, it is not sufficient to be able to retract actions. The performed actions are dynamically combined during the execution of the system. Hence, we have to collect the restore functions while searching for deadlocks. For this purpose we use a tree structure that represents the actions of the program and their order. Every leaf in this tree contains a retractable action and the corresponding restore function. To

generate this representation we redefine the `IO` datatype and all functions yielding it. Values of the new `IO` datatype are leafs in this tree. Inner nodes are created by interpreting the bind operator of the monad instance as a constructor. The function `checkThreads` interprets this structure and checks whether there is a schedule that causes a deadlock.

Considering all possible schedules seems to result in a very large search space. Fortunately, we do not have to consider all possible schedules: in Concurrent Haskell, the matter of communication and synchronization between threads are the (retractable) actions for reading and writing communication abstractions defined in the module `Concurrent`. Hence, it is sufficient to consider only schedulings which perform these actions in all possible orders.

There is another possibility for threads to communicate: by other `IO` actions. For example, one thread can write a file and another thread can later read this file. Fortunately, this is very uncommon and we do not perform search when such communication is involved. Sequential `IO` operations produce special nodes in the `IO` tree. Thus we can handle such actions separately, as we discuss in Section 5.1.2.

5. GENERATING THE IO TREE

This section attends on the generation of a tree representation of the actions used in a concurrent program, similar to Hinzes Backtracking Monad transformers [11]. Before we present how various schedules can be checked by means of this representation, we shortly discuss how Haskell’s module system provides the redefinition of the `IO` datatype. For convenient use of our debugger, we want to keep the name `IO` for the our newly defined datatype. Hence, the original `IO` type must be hidden explicitly which has to be done explicitly since the prelude is always imported.

```
import Prelude hiding ( IO )
```

Furthermore, we also need the original `IO` datatype from the prelude and import it qualified, which means use it qualified as `P.IO`.

```
import qualified Prelude as P
```

To create the `IO` tree we have to redefine the Concurrent Haskell functions. Since these redefinitions will use the original ones, we import the module `Concurrent` qualified as well.

```
import qualified Concurrent as C
```

Hence, each function imported from the original Concurrent Haskell library is preceded with “C.”.

5.1 IO Redefinition

The tree representation of a Concurrent Haskell program is generated by redefining the `P.IO` datatype and the bind operator. The representation consists of three categories each represented by a constructor. They classify all `IO` functions of (Concurrent) Haskell. Each constructor has its action and if required its restore function as argument. The bind operator of the `IO` monad is defined as a constructor, generating a tree structure. Additionally, we add a constructor for `return`.

5.1.1 ConcAction

The constructor `ConcAction` represents all functions of Concurrent Haskell, except for `forkIO`. Naturally, we do

not want our checker to suspend when for example, executing `takeMVar` on an empty `MVar`. Therefore, the first argument of `ConcAction`, the retractable action, is of type `P.IO (Maybe a)`. It yields `Nothing` if the retractable action would suspend. Otherwise, it yields `Just v` where `v` is the result after performing the retractable action. The second argument of `ConcAction` is the corresponding restore function.

```
data IO a = ...
  | ConcAction (P.IO (Maybe a)) (a -> P.IO ())
  ...
```

5.1.2 SeqAction

A concurrent program does not only consist of functions defined in the Concurrent Haskell library. The `SeqAction` constructor is used for all the rest of the `P.IO` functions.

```
data IO a = ...
  | SeqAction (ActionType a) (P.IO a)
```

It has an argument of type `ActionType a` that is defined as follows.

```
data ActionType a = NonBackTrackable
  | Ignorable (P.IO a)
```

This argument distinguishes between two types of sequential actions. `NonBackTrackable` are such actions that request an input from the user. If such an action is used in a concurrent program the behavior can depend on that input. Thus, the debugger cannot check every possible schedule. Nevertheless, we could check for a deadlock, but every time we execute a function of type `NonBackTrackable` we would have to ask the user for input, which does not make sense during search. Thus, if a program uses a `SeqActions` of type `NonBackTrackable`, we stop the check until this action is executed.

The `SeqActions` of type `Ignorable` are such actions that can be ignored by the search because they are of no relevance for the deadlock checking. For instance, performing output will not effect the behavior of the concurrent system. On the other hand, such an action may not be performed during search, because the system would produce too much output. Therefore, `Ignorable` has an argument which defines the action that is executed during search instead of the output. Mostly, this will be `return ()` because we just want to do nothing, but in principle other actions are possible.

5.1.3 ForkAction

We use the constructor `ForkAction` to represent the result of `forkIO`. The first argument is the thread that is forked by `forkIO`, represented by a value of type `IO ()`. The second argument is the retractable action of type `P.IO a`. Because the result type of `forkIO` is `IO C.ThreadId` the polymorphic datatype `a` will be unified with `C.ThreadId`. Thus, the retractable action is of type `IO C.ThreadId` and it yields the `ThreadId` of the forked thread. As mentioned before a thread is killed by removing it from the list of all threads. Thus `ForkAction` takes no restore function as parameter. A detailed discussion of the thread list follows in Section 6.

```
data IO a = ...
  | ForkAction (IO ()) (P.IO a)
```

5.1.4 A Monad: Bind and Return

The `Bind` constructor glues a value of `IO b` and one of `b -> IO a` together to a tree structure, like `(>>=)` from the class `Monad`. Because the result type is `IO a` we have to existentially quantify the polymorphic datatype `b`. This is not possible in the type system of Haskell 98, but fortunately, `ghc` supports existential quantification in data type declarations, described in [16]. Finally, we need a constructor representing `return`.

```
data IO a = ...
  | forall b. Bind (IO b) (b -> IO a)
  | Return a
```

This results in the following datatype for `IO`:

```
data IO a =
  forall b. Bind (IO b) (b -> IO a)
  | ConcAction (P.IO (Maybe a)) (a -> P.IO ())
  | ForkAction (IO ()) (P.IO a)
  | SeqAction (ActionType a) (P.IO a)
  | Return a
```

By defining our `IO` type as an instance of class `Monad`, `IO` functions in the program can be interpreted as functions generating an `IO` tree.

```
instance Monad IO where
  (>>=) = Bind
  return = Return
```

5.2 Concurrent Redefinition

As a next step, every `IO` function has to be redefined, so that it yields a value of the new `IO` type. In this section, we discuss the redefinition of `IO` functions with respect to the different kinds of actions, as defined in the new `IO` datatype.

5.2.1 ConcAction

Retracting `newEmptyMVar` is simple. We just do nothing because the garbage collector will remove the created `MVar`. Thus, the restore function is `_ -> return ()`. Executing `newEmptyMVar` no suspension is possible. The retractable action always yields a new `MVar` packed into the constructor `Just`.

```
newEmptyMVar :: IO (MVar a)
newEmptyMVar =
  ConcAction
    (do searchMVar <- C.newEmptyMVar
       return (Just searchMVar))
    (\_ -> return ())
```

In the case of `putMVar` the retractable action works as follows. First, it tests if the `MVar` is empty. If this is the case, then it performs the `C.putMVar` and yields `Just ()`. Otherwise, it yields `Nothing` because a `putMVar` on a full `MVar` would suspend. In the context of search there will be no interleaving, so that the `MVar` cannot be filled in between `isEmptyMVar searchMVar` and `putMVar searchMVar` (see Section 6). The restore function empties the `MVar` by calling `C.takeMVar`.

```
putMVar :: MVar a -> a -> IO ()
putMVar searchMVar x =
  ConcAction
    (do b <- C.isEmptyMVar searchMVar
```

```
    if b
    then do C.putMVar searchMVar x
           return (Just ())
    else return Nothing)
    (\_ -> do C.takeMVar searchMVar
           return ())
```

The retractable action of `takeMVar` is similar to the one of `putMVar`. If the `MVar` is empty it yields `Nothing` because `takeMVar` would suspend on an empty `MVar`. Otherwise, it takes the value out of the `MVar` and yields it. When we check for a deadlock we execute the retractable action and if the `MVar` can be taken we pass the result to the restore function. The restore function puts this value back into the `MVar`.

```
takeMVar :: MVar a -> IO a
takeMVar searchMVar =
  ConcAction
    (C.takeMVar searchMVar)
    (do b <- C.isEmptyMVar searchMVar
       if b
       then return Nothing
       else do v <- C.takeMVar searchMVar
              return (Just v))
    (\v -> C.putMVar searchMVar v)
```

All the other concurrency primitives are redefined in the same way.

5.2.2 ForkAction

There is only one function yielding a `ForkAction`, namely `forkIO`. The first argument of `ForkAction` is the argument of `forkIO`. This is the thread of type `IO` that should be forked. Again, we refer to Section 6 for the explanation why we do not need a restore action in `ForkAction`. That section will also explain why the retractable action forks a thread that immediately terminates.

```
forkIO :: IO () -> IO C.ThreadId
forkIO newThread =
  ForkAction
    newThread
    (do threadId <- C.forkIO (return ())
       return threadId)
```

As we have already seen, the first argument of `SeqAction` is the `ActionType`: `NonBackTrackable` or `Ignorable`. The second argument is the original action which is needed for performing the action in the debugger. All functions that request an input from the user, like `getChar`, are of type `NonBackTrackable`. The constructor does not take a retractable action nor a restore action because we stop the search at this point.

```
getChar :: IO Char
getChar = SeqAction NonBackTrackable (P.getChar)
```

The search processes an `Ignorable SeqAction` by executing the action defined by the argument of `Ignorable`. In the example of `putChar` we do not want to perform any output. Instead of calling the original `P.putChar` function we do nothing. Thus, the action is `return ()`.

```
putChar :: Char -> IO ()
putChar chr = SeqAction (Ignorable (return ()))
                    (P.putChar chr)
```

Our debugging library provides reimplementations of all IO functions from the standard libraries, returning the new IO datatype. Thus for every SeqAction we define whether it is Ignorable or NonBackTrackable.

6. DEADLOCK CHECKING

This section attends to searching for deadlocks given an IO tree as defined above. We cannot decide whether the concurrent program terminates or not. Thus, we do not know whether the checker would terminate. So we have to limit the execution depth that we check.

To generate various schedules the checker has to coordinate the execution of the forked threads. It has to execute the next action of one of the forked threads, then the next action of the same or another thread and so forth. Thus, we need a mechanism that allows to execute only one action of one specific thread.

Because we do not execute suspending actions, it is possible to map every schedule of various threads to the execution of only one new thread. For every schedule this thread just executes the same actions in the same order as the original threads would. To check for a deadlock we must know whether all threads are dead, that is suspended or terminated. Thus, to check whether a schedule causes a deadlock we just execute a thread that represents this schedule and check if a state exists in which all threads are dead. Therefore, we need the information which thread executes which action. We manage the IO trees of the threads in a list of Threads where a Thread is defined as follows.

```
type Thread = (C.ThreadId, IO ())
```

We assign its unique ThreadId to every thread. Thus the checker can yield a list of these ThreadIds as a deadlock path of threads to be executed. They are later required for the definition of killThread too. This idea of using a list of threads is similar to the cooperative scheduling [4] in Hugs [14].

Now the definition of forkIO makes sense. When we fork a thread we just generate a ThreadId and add the corresponding Thread to the list. The only way to generate a new ThreadId is to fork a new thread. Because we only want to generate a ThreadId the forked thread immediately terminates.

```
forkIO :: IO () -> IO C.ThreadId
forkIO newThread =
  ForkAction
    newThread
    (do threadId <- C.forkIO (return ())
       return threadId)
```

6.1 General Checking

So far, our new IO datatype and new variants of all IO functions are defined. As a next step, we develop the interpreter for searching through all possible schedules of the concurrent system. We start the interpretation with the definition of the function

```
checkThread :: Thread -> P.IO Result
```

where the Result datatype is defined as

```
data Result = Suspended
            | Stepped (Thread,P.IO ())
```

```
| Stop
| Terminate
| Fork Thread (Thread,P.IO ())
```

checkThread executes the next node in the IO tree, i.e. it executes the next retractable action. If the next node is a NonBackTrackable SeqAction, then the result is Stop. The result Terminate is used, when a thread terminates, i.e. the node is Return (). If it is a ConcAction and the retractable action yields Nothing, i.e. executing the action would suspend, then the result is Suspended. Otherwise, the result is Stepped with the performing thread and restore action as parameter. Similarly to Stepped, we yield Fork for ForkActions.

On top of checkThread, the function checkThreads searches for a deadlock in a given list of Threads up to a given depth. The function checkThreads yields a value of type P.IO (Maybe [C.ThreadId]). The list of ThreadIds is a schedule that causes a deadlock. It is Nothing if no deadlock is found. You can think of the function checkThreads as an interpreter for the IO tree. It executes the associated program with every possible scheduling up to a given execution depth. The function uses a depth first strategy, that is it first executes one scheduling up to the maximum depth. Then it undoes the last action and executes another thread instead and so on.

The signature of the function checkThreads is defined as follows.

```
checkThreads :: [Thread] -> Int -> Bool -> [Int]
              -> P.IO (Maybe [C.ThreadId])
```

The first argument ([Thread]) is a list of actually forked threads. When checkThreads is called for the first time the list contains only one Thread, namely the main thread. The second argument (Int) is the depth to be checked. The third argument (Bool) indicates if all checked threads in this depth are dead, that is suspended or terminated. If all Threads in a depth are dead, then there is a deadlock. The last argument ([Int]) says which Threads still have to be checked in this depth. The numbers are positions in the Thread list. We will call this list position list.

The first rule of checkThreads processes an empty position list. In this case all Threads in this depth have been checked.

```
checkThreads ts@(_:_ ) _ dead [] = do
  if dead
    then return (Just [])
    else return Nothing
```

If dead is True all threads are dead and a deadlock is found. Therefore, it yields Just []. The list is empty because we are already in a deadlock state. If dead is False we yield Nothing because after checking all Threads no deadlock was found and so there is no schedule to a deadlock.

If the position list is non-empty checkThreads calls the function checkThread with the first Thread to be checked. Note, that by means of the as pattern n is bound to the value n1-1.

```
checkThreads ts@(_:_ ) n1@(n+1) dead (m:ms) =
  do let thread = ts!!(m-1)
       threadId = fst thread
       resultT <- checkThread thread
       case resultT of
         ...
```

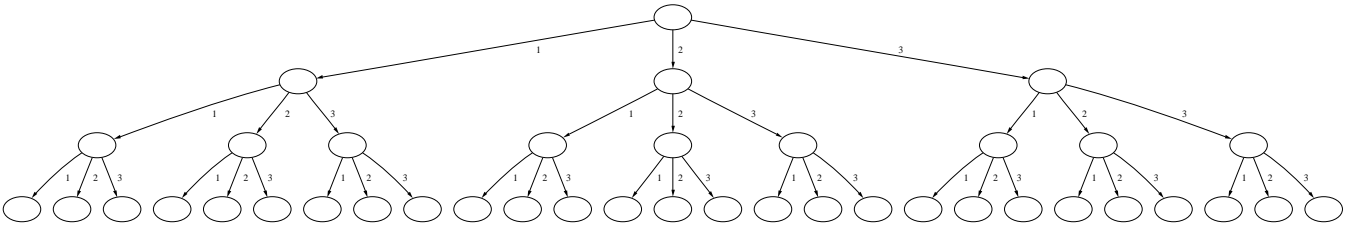


Figure 2: Full Computation Tree

Depending on the result of this function there are five cases to continue, which we discuss in the following:

Suspended

This indicates that the thread would suspend after the next action. In this case we just check the other Threads in this depth. This is done by calling `checkThreads` with the remaining position list:

```
Suspended -> checkThreads ts n1 dead ms
```

Stop

This result indicates that the thread would next perform a `SeqAction` of type `NonBackTrackable`. As already mentioned, in this case we stop the check and yield `Nothing`.

```
Stop -> return Nothing
```

```
Fork newT (t',restoreAction) -> do
  let ts' = replaceWithPos m t' ts ++ [newT]
      checkRes <- checkThreads ts' n True
          [1..length ts']
  restoreAction
  case checkRes of
    Just path -> return (Just (threadId:path))
    Nothing   -> checkThreads ts n1 False ms
```

Terminate

In this case we know that the thread we checked terminated. First this `Thread` is deleted from our `Thread` list. Then the search is continued with the new `Thread` list, a decreased depth and a newly initialized position list because we have deleted one `Thread`. If this call yields a deadlock path, then we add the `ThreadId` of the `Thread` that terminated to this path. Otherwise, we look for a deadlock in the original `Thread` list, with the original depth and indicate by `False` that no deadlock was found.

```
Terminate -> do
  let ts' = deleteWithPos m ts
      checkRes <- checkThreads ts' n True
          [1..length ts']
  case checkRes of
    Just path -> return (Just (threadId:path))
    Nothing   -> checkThreads ts n1 False ms
```

Fork

If the thread has performed a `forkIO`, then `checkThread` yields `Fork`. The first argument of `Fork` is the forked `Thread`. The second is a tuple consisting of the modified `Thread` that forked the new one and a restore function. First the old `Thread` is replaced with the modified one and the new `Thread`

is added to the `Thread` list. Then we look for a path in the new `Thread` list with a decreased depth. The old state is restored by executing `restoreAction`. If a path is found the `ThreadId` of the executed `Thread` is added to the path. Otherwise, the search is proceeded with the old `Thread` list and depth.

Stepped

If the thread has executed an action that doesn't fit in one of the cases discussed so far, then `checkThread` yields `Stepped`. All the concurrent functions like `newEmptyMVar`, `putMVar` and `takeMVar` and the `SeqActions` of type `Ignorable` fall in this category. The argument of `Stepped` is a tuple consisting of the modified `Thread` and a restore function.

In this case the old `Thread` is replaced by the modified one. Then the search is continued with the new `Thread` list and a decreased depth. the retractable action performed by `checkThread` is undone by calling `restoreAction`. If a deadlock path is found the `ThreadId` of the executed `Thread` is added to the path and the search ends with this result. If the call yields `Nothing` the search is proceeded with the old `Thread` list and depth.

```
Stepped (t',restoreAction) -> do
  let ts' = replaceWithPos m t' ts
      checkRes <- checkThreads ts' n True
          [1..length ts']
  restoreAction
  case checkRes of
    Just path -> return (Just (threadId:path))
    Nothing   -> checkThreads ts n1 False ms
```

This search for deadlocks already finds nearby deadlocks during debugging. Indeed, we first integrated this search into CHD and used it successfully for debugging. However, there is the possibility to optimize the approach with ideas from partial order reduction in model checking [5]. Hence, we will first discuss this optimization, before we describe the integration into CHD in Section 7.

6.2 Search Space Reduction

To motivate the reduction of the search space, we consider a concurrent program with three threads, where the actions of all threads are independent from each other, e.g. all threads work on disjoint sets of `MVars`. We search for deadlocks in that program with a maximum depth of 3. Figure 2 shows the structure of a tree representing the computation of `checkThreads`. Each node represents a state of the checked program and each edge the execution of one action. The numbers at the edges indicate which `Thread` is executed in this step. Because the three threads are independent from each other, some paths are equivalent with

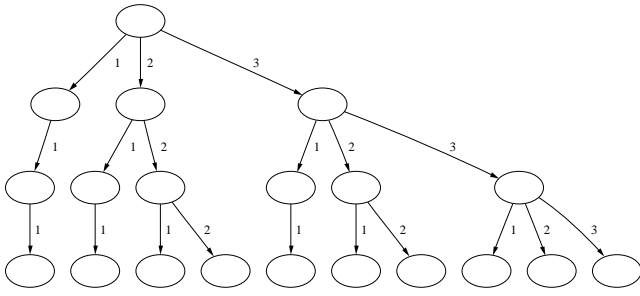


Figure 3: Minimized Computation Tree

respect to searching for a deadlock. All paths that execute the same multi-set of threads are equivalent. For instance, the path 2-1-1 is equivalent to the path 1-2-1. To find a deadlock in this case it would be sufficient to check each multi-set of executions just once. Because the number of states to check is exponential in the depth we use this insight to minimize the number of states to be checked.

We are searching for a minimized tree that is equivalent with respect to detecting deadlocks to the one shown in Figure 2. Every path of that tree must represent a unique multi-set of executed threads and every multi-set of executed threads may only be represented by one path. Figure 3 shows a tree that fulfills this task. For example, the multi-set of executed threads $\{2,1,1\}$ is represented only once in the tree by the path 1-1-2.

In every state we only check the positions in the thread list that are less or equal to the current position. We just have to change the calls of `checkThreads` that go down the tree. Instead of checking the whole position list from 1 to the length of the `Thread` list we have to check only the positions from 1 to the position of the current `Thread`.

Because only in this very artificial example the actions of all threads are independent from each other there is some work to do. For example, if the next action of two threads, say 1 and 2, are dependent on each other (e.g. because they perform `takeMVar` on the same `MVar`) we have to check both execution orders, 1-2 and 2-1.

We define that two actions are *dependent on each other* if they work on the same communication abstraction, that is the same `MVar`, `Chan` and so forth. Otherwise, they are *independent*. Therefore, we add an argument to the `ConcAction` constructor of the `IO` datatype that gives us the communication abstraction on which the action operates.

```
data IO a = ...
  | ConcAction ComAbstr (P.IO (Maybe a))
    (a -> P.IO ())
  | ...
```

In the type `ComAbstrs` all `MVars`, `Chans`, `QSems`, `QSemNs` and `SampleVars` are subsumed and every `ComAbstr` obtains a unique identifier. Now it is possible to write a function that yields all threads that operate on the same `ComAbstr` as the current thread. We add the positions of all `Threads` that operate on the same `ComAbstr` as the current one to the position list.

The function `dependentOn` takes a `ComAbstr` and a `Thread` list and yields all positions in that list that operate on this `ComAbstr` in the next step. Because we do not want to check a `Thread` twice, we delete all positions that are already in

the position list. This is done by filtering only the positions that are greater than `m`:

```
checkThreads ts@(_:_) n1@(n+1) dead (m:ms) =
  do let thread = ts!!(m-1)
      threadId = fst thread
      nextComAbs = nextComAbstr (snd thread)
      dependList = dependentOn nextComAbs ts
      list = filter (>m) dependList
      resultT <- checkThread thread
      case resultT of
        ...
```

Again we discuss the different cases for the result of the call to `checkThread`:

Suspended/Stop

In both cases there are no changes because the search stops or just the rest of the position list has to be checked.

Terminate

In the case of `Terminate` we just check all threads that have a number less than the one of the current thread. We do not check the `m`-th `Thread` because it just terminated and we have deleted it from the list. We additionally check all `Threads` that are dependent on the current one.

```
Terminate -> do
  let ts' = deleteWithPos m ts
      checkRes <- checkThreads ts' n True
        ([1..m-1]++list)
  case checkRes of
    Just path -> return (Just (threadId:path))
    Nothing -> checkThreads ts n1 False ms
```

Fork

When we fork a thread we do not know which actions it will perform. Thus, we do not know which other threads are dependent on the actions of the new thread. We have to assume that the `ForkAction` is dependent on all other threads and we still check all `Threads` in this case. Thus there is no change in this case.

Stepped

The case of `Stepped` completes the changes in the program code. It is similar to the `Terminate` case except that we do not delete the thread but replace it with the modified one. We only check the threads with positions that are less or equal to the current position and add the list of the dependent thread positions.

```
Stepped (t',restoreAction) -> do
  let ts' = replaceWithPos m t' ts
      checkRes <- checkThreads ts' n True
        ([1..m]++list)
  restoreAction
  case checkRes of
    Just path -> return (Just (threadId:path))
    Nothing -> checkThreads ts n1 False ms
```

With this reduction the search space is decreased noticeably, as shown in Figure 3. Unfortunately, we can not find deadlocks anymore: As mentioned above, if the list of thread positions to be checked is empty, then we test for a deadlock as follows:

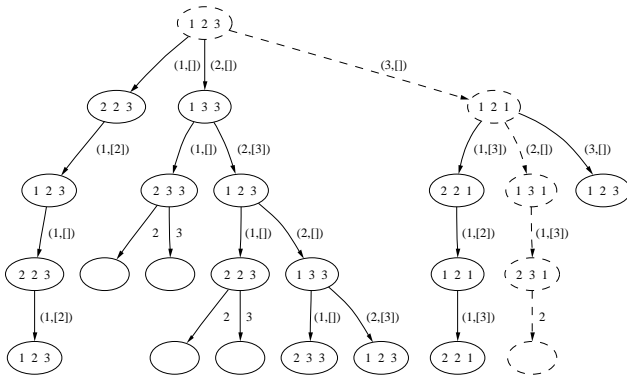


Figure 4: Three Dining Philosophers

```
checkThreads ts@(:_) _ dead [] = do
  if dead
    then return (Just [])
    else return Nothing
```

Because we now do not check all threads in a depth we do not know if all threads are dead. Therefore, we have to keep in mind which threads we have checked. In the case that the list of positions is empty we have to check all threads that we have not checked so far. We do this by calling the function `checkSuspended`:

```
checkSuspended :: [Thread] -> [Int] -> P.IO Bool
checkSuspended _ [] = return True

checkSuspended ts (m:ms) = do
  let thread = ts!!m
      resultT' <- checkThread thread
  case resultT' of
    Terminate -> checkSuspended ts ms
    Suspended -> checkSuspended ts ms
    Stepped (_,restoreAction) -> do
      restoreAction
      return False
    Fork _ (_,restoreAction) -> do
      restoreAction
      return False
    Kill _ (_,restoreAction) -> do
      restoreAction
      return False
    Stop -> return False
```

`checkSuspended` takes a `Thread` list and a position list pointing to the threads to be checked for detecting the deadlock. It starts with the first `Thread` position and checks whether the thread is dead or not. If the next action is a `Stepped`, `Fork`, `Kill` or `Stop` then we just yield `False` (non deadlock) after performing the restore action if necessary. In the case that the next action is `Terminate` or `Suspended`, we check the remaining `Threads`. A check of the empty list yields `True`.

Figure 4 shows a tree representing the computation of the `checkThreads` function when checking a system of three dining philosophers, as defined in Section 2.3. The root node of the tree belongs to a system state in which all philosopher threads are already forked. The numbers within the nodes represent the states of the three `Threads`. For example, if

the first number in a node is 2, then this means that `Thread 1` will next operate on `MVar 2`. The empty nodes represent states checked by `checkSuspended`. The edges are labeled with tuples. The first component indicates which `Thread` is executed in this path. The second component is the list of positions that is additionally added in that specific call of `checkThreads`. The dashed edges are part of the deadlock path.

7. INTEGRATION IN CHD

This section will give some information about our implementation of the presented approach on top of CHD.

Additionally to the retractable action and the restore function which are executed during the check, we add a third argument for actions to `ConcAction` and `ForkAction`. These actions are similar to the retractable actions but do not check whether the execution of this action would suspend. Thus, in the case of `ConcAction` their type is not `IO (Maybe a)` but `IO a`. Furthermore, these actions use the functions provided by CHD instead of the original Concurrent Haskell functions. A function interprets the `IO` tree by executing these actions like the normal program. Because we use the functions defined in CHD we get the control of the scheduling of the program. The CHD blocks every function and allows the user to decide which thread to unblock. Every time CHD blocks the function `checkThreads` is called with the current `Thread` list. Thus, the resulting program works like CHD but searches for deadlocks in the background.

The screenshot in Figure 1 shows the debugger applied to the dining philosophers problem, introduced in Section 2.3. The black circle marks the thread the execution should be continued with (by clicking this thread by the user) to reach the found deadlock (here `Thread5`). If you allow this thread to continue the next thread of the deadlock path will be marked. If you select another `Thread` the program uses iterative deepening to find a new deadlock path based on the new `Thread` list. The program increases the depth until a deadlock is found or the search tree contains more states than a maximum selectable by the user, initially 50,000 states. If the user unblocks a thread before `checkThreads` has terminated, `checkThreads` is signaled to stop. It restores the state before the check and yields `Nothing`. The selected thread is unblocked, performs a step and then a new check is started with the new `Thread` list.

Now we explain which modifications in the Concurrent Haskell program have to be made, to apply our debugger. First, we have to make some changes in our program. Instead of importing the original module `Concurrent`

```
import Concurrent
```

we import our new debugger

```
import ConcurrentSearch
```

Because the prelude is always imported we have to hide its `IO` datatype and all functions using it.

```
import Prelude hiding (IO,putStr,getLine,...)
```

Furthermore, we import the prelude qualified to access the original `IO` monad.

```
import Prelude qualified as P
```

The type of the `main` function of our program is `IO` now instead of `P.IO`. Thus, we rename the original `main` function of the program to `stepperMain`. Then we add a new `main` function which just calls the function `startCheck` that takes a value of type `IO` and starts the check with this value.

```
main :: P.IO ()
main = startCheck stepperMain
```

Now the `main` function has the proper type `P.IO ()`. We have also implemented a preprocessor which takes a program and makes all these changes automatically for the program and all imported modules. Furthermore, it adds position informations to every Concurrent Haskell function. This information is later used by CHD to present the user actual source code positions of the threads.

Additionally to the functions presented in this paper we have redefined the remaining (Concurrent) Haskell `IO` functions. Thus, the presented `IO` type contains an extra constructor namely `KillAction` for the redefinition of the function `killThread` of the Concurrent Haskell library. Furthermore, we have redefined all functions on `QSemS`, `QSemNs`, `SampleVars`, `Chans` and the `IO` functions defined in the `System.IO` modules of the `ghc`. Because of the modular design of the debugger it is easy to extend it with further concurrency primitives or to hide the concurrency primitives behind your own complex communication abstractions.

8. PRACTICAL APPLICATION

We have not evaluated our tool in a large case study yet but to show the practical usability we applied it to different versions of the dining philosopher problem. First, we used it for a system of ten dining philosophers. Already in the initial configuration, before the philosophers are forked and the sticks are created, our debugger is able to find the deadlock and guide the user through more than 40 steps into the deadlock configuration. This was surprising to us, since initially, we were only interested in finding deadlocks very close to the actual configuration.

As a simulation of a more complex application, we redefined the application such that the sticks are administrated by a server. In this system, the initial configuration is too far away from the deadlock position and the search space too large to find a deadlock. The same holds even for some later states of the system. However, testing showed that it is very probable to reach a configuration in which the debugger is able to detect a deadlock path with about 22 steps.

For larger applications, we expect our approach to scale well for the following reasons:

- If the path to a deadlock is too long, then the user can still debug the program by using CHD features like stepwise execution. Search stops if the state space contains more than a specified number of states.
- The search space is independent of the number of used communication abstractions. Only the number of active (not suspended) threads is relevant.
- The degree of dependent concurrent actions is usually low. Communication is usually only performed between a small amount of threads. Hence, it is sufficient to consider all possible interleavings for only these threads and partial order reduction should reduce the state space efficiently, as known from using it

for improving model checking. Furthermore, in larger applications many threads are waiting for the results of other applications, e.g. client threads wait for results from a server thread. These threads do not effect the search space until the server produces the result. Then the server is usually suspended until another client submits a request. Hence, parts of the system are sequentialized and the state space is not blown up.

- If the system waits for external input, e.g. from the keyboard or via a TCP socket, then the existence of a deadlock may depend on the concrete input. Hence, search does not make sense in this case and as long as one thread waits for input, it is not possible to detect a deadlock in the system. The same problem holds for formal verification. Here search is done with respect to all possible inputs which is either an abstraction of the real program behavior (and may contain mistakes) or results in a state space explosion making verification impossible.

However, it would make sense that the user can specify subsets of threads which should not suspend altogether in one system state. This could give a hint to partial deadlocks, although it is very difficult to identify partial deadlocks, since other threads may later communicate with one of these threads and reactivate these suspended threads.

- A slowdown during debugging is usually no problem for concurrent systems, since these systems in most cases do not perform complex computations. Instead, concurrency is used as a means for guaranteeing reactivity of a system. Larger initializations or computations can be skipped during debugging by introducing break points in the source code. No search is performed until the break point is reached.

However, using our debugger consumes memory, especially stack space since we perform backtracking, while debugging. Using `ghc` this is no problem, since stack space can be increased when starting the application. For example, for debugging the server based implementation of the dining philosophers, 5 MB stack space is sufficient.

9. RELATED WORK

To the best of our knowledge, there is no similar approach to improving debugging of concurrent systems. The only work related to ours, is Java MultiPathExplorer (JMPaX) [1, 20], which also extracts properties of other schedules during the execution of concurrent Java programs. However, they only consider one execution of the system, and verify properties for all schedules obtained by switching independent actions. These are exactly the schedules, we are not interested in, because they contain a deadlock if and only if the execution yields into a deadlock. However, they are not interested in detecting deadlocks but in other unexpected configurations which (in most cases only) might yield into e.g. a deadlock. For instance, they consider lock reversals [15] which means two locks (in our context usually `MVars`) are once taken in one order and later they are taken in the other order. We think, that our approach would also improve their Tool. However, applying our approach to Java

is difficult, since Javas sequence operator (`;`) is a language feature which cannot be redefined like the function (`>>=`) in Haskell. A modification of Javas virtual machine would be necessary, implementing the presented search on the level of Java byte code.

Our work is also related to the area of software model checking where formal verification of system implementations is proposed by applying model checkers, like SPIN [12], to concurrent programming languages. For instance, Java Pathfinder [9] and the Bandera Tool [6] translate concurrent Java programs into Promela, the specification language of SPIN. System properties can be defined in a temporal logic and then be checked using SPINs model checker which uses a similar partial order reduction as presented in this paper. To avoid state space explosion and to make the approach applicable for infinite state systems, the user can also define data abstractions. However, finding good data abstractions can be difficult and error-prone. Although these works are considered for software verification, they are only applied to toy examples in the case studies. The verification of real concurrent systems still seems to fail because of the state space explosion problem. As a consequence, the developers of Java Pathfinder started the development of Java PathExplorer [10], which performs runtime model checking for concurrent Java programs. The goal is not formal verification any more but applying the formal techniques of specifying system properties in temporal logics and model checking for more powerful testing. The further development of Java PathExplorer resulted in Java MultiPathExplorer, discussed in the last paragraph.

Other work related to this approach to debugging are debuggers for sequential Haskell. First, we should mention the Haskell Object Observation Debugger HOOD [8] which is also integrated in `ghc`. In this system the user is able to observe the evaluation of intermediate data structures without influencing lazy evaluation. However, this approach does not help searching for deadlocks, since usually, deadlocks do not depend on the values but the order of taking and putting values to resp. from a communication abstraction. For instance, in the philosophers examples, the value (`()`) will not be evaluated at all, because there is no pattern matching on it. Hence, debugging by observation cannot help. Other approaches to debugging Haskell programs are the algorithmic debugger Freja [17] and the Redex Trail System [21] which was later extended to Hat [3] covering the ideas of HOOD and Freja as well. However, Freja and Hat are not even applicable to programs written in Concurrent Haskell. Furthermore, these approaches do not fully support the `IO` monad and reactive, non-terminating functions.

There are also many debuggers for non-functional languages supporting concurrency. However, these debuggers usually work with a modified compilation/abstract machine and are hence, not comparable with our lightweight approach as a library. Furthermore, there is no approach providing search for deadlocks available.

10. CONCLUSIONS AND FUTURE WORK

We have presented a new approach for improving debugging of Concurrent Haskell programs. The system is conveniently integrated in the Concurrent Haskell Debugger. The basic idea is searching for deadlocks, whenever the computation is idle, while the user steps through the system states in the debugger. If a deadlock is found, then the

user is guided into this deadlock. We have presented the basic ideas of the implementation of the search and an optimization based on partial order reduction. Furthermore, we sketched the integration of the search into CHD. We have mostly restricted the presentation to `MVars`, but all concurrency primitive of Concurrent Haskell are provided as well as all common `IO` functions. The system is available at <http://www.informatik.uni-kiel.de/~fhu/chd>.

We implemented the search for deadlocks for Concurrent Haskell. However, a similar approach should also be useful for debuggers for other programming languages providing concurrency, although in most cases such a lightweight, library based implementation will not be possible. The main opportunity of Concurrent Haskell is the possibility to redefine the sequence operator (`>>=`), which is not possible for the imperative sequence operator (`;`). Here different solutions on the level of code generation or more complex debugging engines have to be found. Further opportunities of (Concurrent) Haskell which would also be useful for transferring the approach to other languages are

- communication abstractions, that make an identification of inter-thread communication possible so that all schedules with respect to these communication actions can be searched,
- the possibility of redefining the communication functions,
- being able to retract concurrent actions, and
- having an expressive type system which allows the definition of `IO` actions as data structures, like existential quantification in data type declarations [16].

The idea of the unoptimized search works independently of the synchronization and communication concepts of the language since all possible schedulings are performed and retracted during backtracking. However, it is important to reduce the possible thread switches to selected positions during the computation by identifying inter-thread communication. For example, this could be a problem, when threads communicate in an unsynchronized way via shared memory, like it is possible in Java. Fortunately, the same problems have to be solved in the verification tools for Java. Hence, these results should help transferring our approach to languages with a less adequate communication concept. The application of partial order reduction should be applicable to other synchronization and communication concepts as well, but this optimization depends on the identification of independent actions, which can be more complicated for other concepts.

For future work, we have different directions in mind. At the moment the implemented search is only considering deadlocks. However, programmers are also interested in other bugs like partial deadlocks, lifelocks and not guaranteed mutual exclusion. For finding other bugs, we developed an environment for checking Linear Time Logic (LTL) formulas while executing a Concurrent Haskell program [13]. We want to integrate both approaches to improve the search. However, using search with integrated LTL run-time checking is not as simple as searching for deadlocks. Additional effort of the programmer is needed to specify properties of the system in LTL and annotate the program by state propositions. On the other hand, we showed that some fixed prop-

erties (in [13] we presented lock reversal) can easily be integrated in this approach. We want to find more common bugs and integrate this into this extended search.

Another direction for future work, is a larger case study. This should give hints for improving our debugger, especially adding different views. We also want to check whether a more precise distinction of independent actions will improve the search. At the moment we distinguish them only by the communication abstraction they operate on. As an improvement, we could also distinguish the different operations. For example, two `isEmptyMVar` actions performed on the same `MVar` are independent as well. However, being more precise yields additional costs and it is not clear that this will really allow deeper search in practice. Finally, we will investigate, if a combination of our debugger with Hood could be useful. Although the values in the communication abstractions are not directly relevant for deadlocks, it can be interesting to view the values inside a concurrency abstraction (and perhaps also how much they are evaluated). At the moment the programmer can only label these values by hand e.g. using `putMVarLabel` instead of `putMVar`, but an automatic labeling by means of Hood could be more convenient.

11. REFERENCES

- [1] G. Agha, G. Rosu, and K. Sen. Runtime safety analysis of multithreaded programs. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 337–346. ACM Press, 2003.
- [2] T. Böttcher and F. Huch. A Debugger for Concurrent Haskell. In *Proceedings of the 14th International Workshop on the Implementation of Functional Languages*, Madrid, Spain, Sept. 2002.
- [3] 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*, 2011:176–193, 2001.
- [4] K. Claessen. A poor man’s concurrency monad. *Journal of Functional Programming*, 9(3):313–323, 1999.
- [5] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *Software Tools for Technology Transfer*, 2, 1998.
- [6] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. IEEE Computer Society.
- [7] The Glasgow Haskell compiler. <http://www.haskell.org/ghc/>.
- [8] A. Gill. Debugging Haskell by observing intermediate data structures. In *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41-1 of *ENTCS*. Elsevier, Sept. 2000.
- [9] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), Apr. 1998.
- [10] K. Havelund and G. Rosu. Java PathExplorer — A runtime verification tool. In *In Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space, ISAIRAS’01, Montreal, Canada, June 18–22, 2001.*, 2001.
- [11] R. Hinze. Deriving backtracking monad transformers. *ACM SIGPLAN Notices*, 35(9):186–197, 2000.
- [12] G. J. Holzmann. Proving properties of concurrent systems with SPIN. In *Proceedings of the Sixth International Conference on Concurrency Theory*, volume 962, pages 453–455, 1995.
- [13] F. Huch and V. Stolz. Runtime verification of Concurrent Haskell programs. In *Proceedings of the Fourth Workshop on Runtime Verification*, to appear in *ENTCS*. Elsevier Science Publishers, 2004.
- [14] The Haskell interpreter Hugs. <http://www.haskell.org/hugs/>.
- [15] K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In W. Visser, K. Havelund, G. Brat, and S. Park, editors, *SPIN Model Checking and Software Verification (7th International SPIN Workshop)*, volume 1885 of *Lecture Notes in Computer Science*, Stanford, CA, USA, August/September 2000. Springer.
- [16] K. Läufer. Type classes with existential types. *J. of Functional Programming*, 6(3):485–517, May 1996.
- [17] 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.
- [18] S. Peyton Jones et al. Haskell 98 report. Technical report, <http://www.haskell.org>, 1998.
- [19] S. Peyton Jones, A. Gordon, and S. 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 Jan. 1996.
- [20] G. Rosu and K. Sen. An instrumentation technique for online analysis of multithreaded programs. In *to appear as invited Paper at Parallel and Distributed Systems: Testing and Debugging (PADTAD-2)*, 2004.
- [21] J. Sparud and C. Runciman. Complete and partial redex trails of functional computations. *Lecture Notes in Computer Science*, 1467:160–174, 1998.