

Concurrency abstractions for Concurrent Haskell

Volker Stolz¹ and Frank Huch²

¹ RWTH Aachen, 52056 Aachen, Germany

² Christian-Albrecht-University of Kiel, 24118 Kiel, Germany

Abstract. In this article we propose a basic set of concurrency abstractions for Concurrent Haskell based on our experiences with various applications. These abstractions serve as templates for patterns recurring in concurrent program development. Especially for threads depending on each other, high-level constructs allow monitoring of a set of child threads and taking actions in case a vital component terminates. The implementation utilises advanced features of the Glasgow Haskell compiler.

1 Introduction

Concurrent and parallel programming are of growing interest to the functional programming community. Functional languages have been used for some time now for parallel scientific calculations (e.g. [3, 12]). Opposed to parallelism, where a computation is divided into sub-computations and distributed among the available processing elements, concurrency can also be used to support reactivity of a program. For instance, this can be useful in the development of graphical user interfaces or programs communicating with other applications or with other computers via a network. Finally, for some algorithms very intuitive concurrent specifications can be given. For their implementation a concurrent language can be useful.

In general, every time the system which is to be implemented focuses on the semantic abstraction into separate processes instead of just serial computation, it is more adequate to use threads to model the actual system. This has several advantages: It saves developer time, is less prone to bugs and does not require the tedious or even infeasible task of trying to express the behaviour of the system in a sequential way.

For concurrent programming in the lazy functional programming language Haskell [9], Jones, Gordon and Finne proposed Concurrent Haskell [10] which integrates concurrent lightweight threads in Haskell's IO monad. The current implementation uses a preemptive scheduler inside the Glasgow Haskell Compiler (*GHC*) [17]. For the improvement of efficiency, current efforts in the implementation of Concurrent Haskell in the *GHC* try to map threads to operating system threads, which in turn can be scheduled on different CPUs. Recent developments even cross the boundary between concurrent and distributed computing: Glasgow Distributed Haskell [14], Erlang-style distributed Haskell [15] and Port-based Distributed Haskell [6] have their own paradigm regarding interaction between threads on different physical machines.

For communication between different threads, Concurrent Haskell offers a variety of concepts, all based on mutable variables (`MVar`). Mutable variables are embedded in the IO monad [8]. This is necessary because of the nondeterminism of the underlying interleaving semantics. Different schedules may lead to different communication taking place and therefore to different results. In the IO monad, threads can create `MVars`, read values from `MVars` and write values to `MVars`. If a thread tries to read from an empty `MVar` or write to a full `MVar`, then it suspends until the `MVar` is filled respectively emptied by another thread. `MVars` can be used as simple semaphores, e.g. to assure mutual exclusion or for simple inter-thread communication (notice that we distinguish between threads and operating system processes). Based on `MVars` a couple of higher-level objects are built, most notably:

- Semaphores, to indicate that a certain quantity of resources is available/needed
- Channels, which provide FIFO-queues

Especially the latter proves useful for asynchronous communication between several threads as the sender does not have to wait for a `writeChan` to succeed while a `putMVar` on a full `MVar` will block. On the receiving end of the channel (which can have several readers), messages will be retrieved sequentially in the order they have been written into it, blocking until another item arrives if the channel is empty.

Unfortunately, the abstractions provided by Concurrent Haskell are not expressive enough for high-level concurrent programming. Already in the early seventies, the disadvantages of semaphores were pointed out and other approaches like monitors were proposed [16]. Nevertheless, Concurrent Haskell still proposes this low-level concept of `MVars` and semaphores. The only exception are channels, which can be used for communication by means of message passing instead of communication by shared variables. However, this is still a very low-level concept compared to high-level functional programming in Haskell. Although these data types for communication are called concurrency abstractions, they are only communication abstractions. In a high-level concurrent functional programming language threads should be first class citizens. This means, real concurrency abstractions combine threads to new, more complex concurrency objects. In this paper, we present some concurrency abstractions, which we suppose to be useful for many concurrent applications. All these abstractions are collected in a library available from <http://www-i2.informatik.rwth-aachen.de/~stolz/Haskell/>.

2 Concurrency Abstractions

In this section we first give a relatively simple abstraction to show how to use Concurrent Haskell's features. The more complex abstractions provide a powerful means of expression while maintaining a concise interface. Although the general idea of their implementations is straight forward, Concurrent Haskell has some pitfalls which have to be accounted for.

The main elements we will be using are:

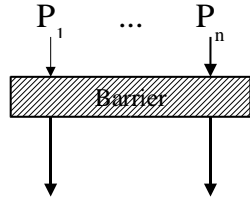


Fig. 1. Simple barrier

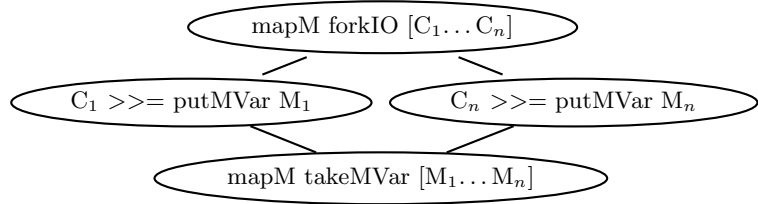


Fig. 2. Specialised barrier

- `newEmptyMVar :: IO (MVar a)` creates an empty mutable variable;
- `putMVar :: MVar a -> a -> IO ()` which writes a value to an MVar;
- `takeMVar :: MVar a -> IO a` which retrieves a value, blocking if the MVar is still empty;
- `forkIO :: IO () -> IO ()` starts an IO computation in a separate thread.

The first concurrency abstraction we define is a parent-child relationship between threads which is not provided by Concurrent Haskell. Its definition will be a replacement for `forkIO` which ensures that children are either terminated ‘gently’ or by force, depending on the requirements. This section will also serve as a quick introduction to the perilous waters of asynchronous exception [13] which we need for the other abstractions.

For advanced usage, we show how to adapt two important abstractions employed by Erlang [1]: Generic Servers and the Supervision Trees. These abstractions will present a slight paradigm shift from simple synchronisation to message passing, including control messages for sub-threads.

2.1 Barrier

The barrier is the most simple application we present. It is designed not to hand out work to a number of threads, but simply as a means of synchronisation. Barriers are mostly found in applications divided into several threads where at one point it must be assured that all threads reach a certain state and do not proceed until indeed all threads have arrived at their designated locations. Only then each thread continues its operation. Some APIs such as OpenMP [2] embed a `barrier` command into their language. In Concurrent Haskell, we can get this effect using a list of MVars. In contrast to a build-in directive, we have to initialise the barrier with the number of threads that will later synchronise on it.

While for the general case of barriers in arbitrary locations of threads there is no need to provide an abstraction because the naïve usage of MVars is sufficient (see Figure 1), it makes sense to provide a specialised version: One thread splits into various threads, each one doing his job and then terminates, returning a result to the parent. Computation only continues after all sub-threads have finished their work (Figure 2).

Listing 2.1. A simple barrier implementation

```
barrier :: [IO a] -> IO [a]
barrier ps = do
  mvars <- replicateM (length ps) newEmptyMVar
  mapM_ (\ (mv,p) -> forkIO (p >>= putMVar mv)) (zip mvars ps)
  mapM takeMVar mvars
```

The implementation shown in Listing 2.1 creates a set of MVars and uses standard list functions to set up the necessary data structures. For each job in the list of IO actions a thread is started using `forkIO`. Each thread places its result in its associated MVar where it can be collected by the parent. Note that the expression `mapM takeMVar mvars` does not necessarily imply that the threads terminate in the order they were started. Just their results are retrieved in the required order.

3 Parent/Child relationship

An important quality in threaded, concurrent programming is the ability to make threads dependant of each other, e.g. a failure in one thread should sometimes propagate to another thread. Of course this can always be achieved by explicit programming. However, a better approach is to encapsulate this behaviour.

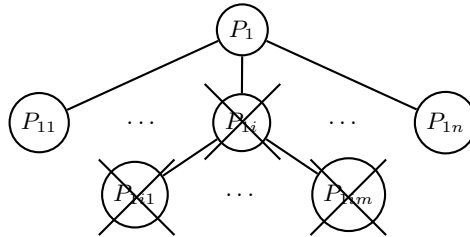


Fig. 3. Notification of children

A common conception is that a thread should be forcibly terminated if its parent, that is the thread which created it, terminates. This is illustrated in Figure 3: P_{1i} terminates and its children are terminated as well. We wrap the parent's computation into `Exception.catch` to be notified if it exits abnormally. An MVar-protected finite map, the family tree, keeps a mapping from parent threads to lists of all their children. This map has to be passed to the functions involved as an additional parameter because Haskell as a functional programming language does not have global, mutable variables. By means of the `ThreadKilled`-exception all children of the terminated thread are looked up in the finite map and terminated as well. To distinguish our new function from the regular `forkIO`, we call it `spawn`.

Listing 3.1. spawn provides a parent-child relationship

```

type Family = FiniteMap ThreadId [ThreadId]

spawn :: MVar Family -> IO () -> IO ThreadId
spawn familyMV io = do
  sync <- newEmptyMVar
  parent <- myThreadId
  block $ do
    modifyMVar familyMV $ \ familyFM -> do
      child <- forkIO $ do
        me <- myThreadId
        finally
          (do putMVar sync ()
             unblock io
             removeFromTree familyMV me)
          (reapChildren familyMV me)
      takeMVar sync
    let newFamFM' = addToFM_C (++) familyFM parent [child]
    return (newFamFM', child)

```

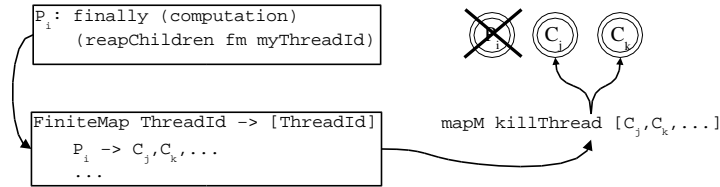


Fig. 4. Table of requested links

Using the MVar sync, our implementation (see Listing 3.1) ensures that the parent waits at least until its child has entered the `finally`-block. Otherwise, it would be possible for the parent to terminate and kill its child before this child has registered its exception handler which will take care of terminating its children in turn. For the modification of the MVar containing the family tree map we use the function `modifyMVar` which guarantees mutual exclusion for MVar modifications. Furthermore, this function returns the second component of the result pair of its argument function (the `ThreadId` of the created child). The surrounding `block`-statement is required so that the current thread does not get interrupted by other threads while doing his housekeeping. Notice how the actual computation of the child executes inside of an `unblock`, so that asynchronous exceptions like `ThreadKilled` can be received.

If a spawned thread exits, it will enter into the second argument of `finally` where it triggers the removal of its children via `reapChildren` (see Figure 4, it simply delivers a `ThreadKilled` exception to all children registered in the map and prunes the family tree). We will later pick up a similar design in the supervision tree where the parent monitors the state of its children.

3.1 Linking

In the same direction as the parent/child abstraction heads the more general linking: Here, the programmer is interested in getting notified if some thread terminates. While in the previous case such notification only exists for the parent, we would now like to attach to an arbitrary thread at a later time. This thread should not need to be a direct child of the current thread and we would like to be able to install any IO action instead of hard-coding a notification function. It should also be possible to remove this event handler.

It is not possible to attach this kind of handler to a thread already spawned using `forkIO` because we can only attach exit handlers to threads which have been started using our `spawn` function. Otherwise, there is now way to wrap the thread's already running computation in the `finally` required for our mechanism to work. To indicate this we use the separate type constructor `Spawned` for `ThreadId`s returned by `spawn`.

It might be possible to use `GHC.Weak.addFinalizer`, but finalisers currently have some issues: There is no guarantee that finalisers are run at all (this should not happen with our implementation unless you leave the Haskell runtime system 'by force', e.g. a C call) nor when they are run (recent examples include running finalisers with `print` statements after the runtime system closed the standard IO handles).

Instead, we establish a mechanism similar to the one used previously where we used `finally` in case a thread terminates either expectedly or unexpectedly. The solution in fact turns out very similar: Using the `link :: Spawned -> IO () -> IO Link` directive, a thread can add an exit handler to another thread. To do this, we keep a map from threads to associated threads interested in receiving notification. `reapChildren` from Listing 3 turns into `notify` which triggers execution of the associated IO action.

More sophisticated versions using Dynamic Exceptions [11] instead of the indiscriminate use of `ThreadKilled` should provide a better way of informing other threads of such events. Otherwise, it is not possible to distinguish these events from any other occurrences of `ThreadKilled` exceptions.

Removal of such a link is just a matter of deleting the corresponding entry from the values of the thread tree with `unlink :: Link -> IO ()`.

As the data structures and the implementation for linking and the parent/child-abstraction are similar, the actual library subsumes both approaches into a single `spawn` function.

4 Event based programming

This section offers a different view on concurrent computations: We want to hide the explicit computation going on. Our goal is to develop an event listener which, after being initially created, can be triggered from our applications. It will undergo various modifications, each making it more generic.

The first goal is to execute an arbitrary IO action asynchronously when an event is triggered. The IO action shall be determined at creation time of the

event listener. Naturally, we would like to be able to pass parameters to the event, so the argument for the initialisation in fact turns out to be a function. As another requirement, the initialisation should return the function we use to trigger the event. Thus we obtain as a first type signature (we will derive the implementation from the more general case below):

```
createEventHandler :: (a -> IO ()) -> IO (a -> IO ())
```

After creating an event handler, the event can be triggered by calling the function resulting from `createEventHandler`. For example, a simple event handler which logs strings by printing them to the terminal can be defined as:

```
log = do
  logger <- createEventHandler (\ m -> putStrLn $ "Log: " ++ m)
  logger "Successfully created logging service"
```

If the event handler should maintain a state, an implementation using the above signature would imply first creating a global structure inside an `MVar` and supplying it as an additional argument to `createEventHandler`. The event handler would then need to use Concurrent Haskell expressions to retrieve the current state and protect itself against race conditions before finally updating the global state with the result of its computation. Only in this case another instance of the event handler will be able to proceed.

As an improvement, the state can be moved into the construct: With a minor modification, we require the handler to take an initial global state and yield the new state. By now, the similarity to the `fold` functions becomes obvious (the type variable `s` denotes the state):

```
createStateEventHandler :: (s -> a -> IO s) -> s -> IO (a -> IO ())
createStateEventHandler io e = do
  ch <- newChan
  forkIO $ do xs <- getChanContents ch
              foldM_ io e xs
  return $ writeChan ch
```

Assume that we want to keep track of the number of messages logged so far, starting at 1:

```
logger <- createStateEventHandler 1
  (\ i m -> do putStrLn $
              "Log entry" ++ (show i) ++ ": " ++ m
              return (i+1))
```

Using a channel in this implementation, we ensure that all events are executed in the order they are received (without precluding the user from using even more concurrency inside `io`, e.g. executing all events concurrently if the state is read-only). The event handler without a state can easily be expressed by means of this function:

```
createEventHandler' :: (a -> IO ()) -> IO (a -> IO ())
createEventHandler' io = createStateEventHandler (const io) ()
```

By encapsulating both the iteration and the state transformation it is once again possible to capture behaviour in a pattern which allows re-usage and adap-

tion to maintain a smaller code base and reduces the number of functions implementing similar behaviour in different ways.

5 Client/Server constructs

In this section we define abstractions from the realm of client/server systems. A client sends a request to the server which triggers some computation inside the server. Then, the result of the computation is sent back to the client.

In the first client/server abstraction, the server (a work-pool) exists only virtually. It is just a queue of jobs. Idle clients from a set of (idle) workers request tasks from a server and execute them concurrently. The other abstractions are the Generic Server and the Supervision Tree which are derived from client/server libraries available in Erlang.

5.1 Work-pool

A common problem is how to distribute tasks between workers. To prevent bottle necks (e.g. in processing capacity), it should be possible to perform tasks concurrently. However, executing all possible tasks concurrently can also be problematic. System restrictions (e.g. bandwidth restricts to the maximal speed of concurrent network communications) must be considered. Therefore, we restrict the number of workers to a fixed limit. In a first approach, we start a thread for each worker in the initialisation part of the work-pool. All tasks are stored in a channel which idle workers can query for a new task (instead of distributing the different tasks – usually of type `IO a` – beforehand). As the execution of one task by a worker may produce new tasks, those new tasks are then added to the channel representing the work-pool. So, we provide a way of injecting more tasks into the queue of things still to do.

This makes it a bit harder to establish whether our workers have finished all jobs: If the list of jobs is empty, but there are still workers solving their job, more work might be generated at a later time. This complicates formulating the termination criterion and is a form of the problem of Distributed Termination as detailed in [4]. We will give two approaches to solve this problem.

Unluckily, the first solution which relies on GHC's ability to detect deadlocks, needs to work around a deficiency and is not applicable in the general case. The second solution solves the problem but is not as easy to follow because of some overlapping conditions which rely on the order of evaluation.

The type signature of our function `concurrentEval` implementing the work-pool resolves as follows: The first argument is the number of workers to use. As the computation might depend on a global state, the next parameter serves as the initial state. If this state is indeed to be shared with write access by all workers, this value must be protected by an `MVar`. Next comes the actual function which takes a copy of the current state, the result of one evaluated job from the list of things to do and calculates the new state. Its return-type

Listing 5.1. Workerpool with deadlock detection

```
concurrentEval :: Int -> s -> (s -> b -> IO (s,[IO b]))
               -> [IO b] -> IO s
concurrentEval n initialState f work = do
  ch <- newChan
  stateMV <- newMVar initialState
  mapM (writeChan ch) work
  barrier $ replicate (n-1) (runWorker stateMV f ch)
  runWorker stateMV f ch
  takeMVar stateMV

runWorker :: MVar s -> (s -> b -> IO (s,[IO b])) -> Chan (IO b) -> IO ()
runWorker stateMV f ch = do
  Exception.catch (do
    job <- readChan ch
    result <- job
    moreWork <- modifyMVar stateMV $ \state -> f state result
    mapM (writeChan ch) moreWork
    runWorker stateMV f ch)
  (\ _ -> return ())
```

indicates that more work may be created as the second component of the tuple. Finally, the last argument is the list of IO computations to execute.

In the first approach in Listing 5.1 we use a channel which we fill with the list of initial tasks. More work can simply be added by `writeChan :: Chan a -> a -> IO ()`. We start the desired number of worker threads by using the `barrier` function as defined above. Each worker takes an item out of the channel and starts to work. This is done in an endless loop. Thus, we will reach a situation in which all workers have terminated and are suspended on the `readChan :: Chan a -> IO a` instruction. The Haskell runtime system will detect this situation under certain conditions and send an asynchronous exception to one of the blocked threads which terminates the thread if not caught. These steps are repeated until the deadlock is relieved or no threads are left.

This asynchronous exception can be caught in the usual way using the function `Exception.catch :: IO a -> (Exception -> IO a) -> IO a`. If we spawn $n-1$ worker threads and use the main thread to initialise the computation and as the last worker, we can observe the following behaviour:

- If there is at least one worker active, the system is not deadlocked and proceeds as required by our specification.
- If the system runs out of work, effectively deadlocking and a worker thread receives the exception it simply terminates. The system remains deadlocked, leading after several steps to the interesting case:
- The main thread receives the exception: We know for sure that all computations are finished and can proceed with the remainder of the program! For

Listing 5.2. Work-pool with one control thread

```
concurrentEval :: Int -> s -> (s -> b -> IO (s,[IO b]))
               -> [IO b] -> IO s
concurrentEval n state f jobs = do
  m <- newEmptyMVar
  concurrentEval' n n state f jobs m

concurrentEval' :: Int -> Int -> s ->
                 (s -> b -> IO (s,[IO b])) ->
                 [IO b] -> MVar b -> IO s
concurrentEval' nMax n state f jobs m
  | (n==nMax) && (null jobs) = return state
  | (n==0) || (null jobs) = do
    r <- takeMVar m
    (state',newJobs) <- f state r
    concurrentEval' nMax (n+1) state' f (newJobs++jobs) m
  | otherwise = do
    forkIO ((head jobs) >>= putMVar m)
    concurrentEval' nMax (n-1) state f (tail jobs) m
```

cleanup it might be necessary to terminate the remaining (now idle) worker threads which have not been killed by the runtime system yet.

Unfortunately, the deadlock is only detected if all threads in the runtime system stuck. I.e., if there is at least one other unrelated thread running (which is to be expected in a concurrent system), the deadlock will remain undetected and the work-pool does not terminate. Therefore, we need a different way of tracking the amount of work without resorting to deadlock detection:

In the solution in Listing 5.2 we do not start any workers initially. We create a worker for every task if the maximum number of workers is not reached. Once a worker has finished his task, it writes the result to an `MVar` and terminates. The function `concurrentEval'` loops until all workers have returned (`n == nMax`) and the list of work to do is empty (`null jobs`). If there is still work we have to distinguish two cases:

- No idle worker available (`n == 0`) or no work to do (see above);
- At least one worker is available and we have still got work to do.

In this case, we wait on the `MVar` for a worker to return its result, apply `f` to it and then reenter the loop, indicating that a worker is available via the counter and additionally adding new jobs to the end of the queue. So, in the `otherwise` case, we simply start the worker and decrement the current worker count for the next iteration.

The idea to this concurrency abstraction came up with an analysis tool for web pages. Although the whole system is implemented purely sequential, concurrency enables optimised access to the network as an HTTP client. By applying

the work-pool abstraction, it is not necessary to directly use or understand any concurrency features in the implementation of the analysis tool.

5.2 Generic server

A very common abstraction found in the imperative world is programming by means of remote procedure calls (RPC) or remote method invocation (RMI) in an object oriented context. In [6] and [7] we showed that especially in designing large systems it is best to use a communication paradigm which does not only offer concurrency. The same API should also be used for distributed computing. Using this programming technique, it is much easier to move subtasks to different machines for reasons of load balancing without having to rewrite the entire application. Usually, the API offers access to different services by host name and service name, so that instead of specifying a local service the only change is indeed requesting a remote host to execute the task via the name service API.

Another important factor is to remain consistent in programming large systems or even in separate projects and use the same style to make later understanding easier. As we saw before, we can almost arbitrarily use MVars or Channels for communication. To make maintenance easier, we adapt Erlang's 'Generic server' concept. Since the communication mechanism of Erlang is different than the one in Concurrent Haskell, some modifications have to be made. Especially, the fact that atoms (constructors with no arguments) are interpreted as names of defined functions in the Erlang implementation of generic servers requires a completely different implementation.

In our implementation, clients communicate with a generic server via a channel. The channel can either contain control messages (e.g. to suspend, restart or terminate the server) or arbitrary client requests which also contain MVars for the answer of the server. We use this channel as a reference to the generic server as well. It is also needed in the standardised call by the client. The functionality of the server is passed as a functional argument, when the generic server is initialised by a call to `genServer`:

```
type GenServer a b = Chan (Either Control (a, MVar b))
genServer :: IO a -> (a -> b -> IO (a,c)) -> IO (GenServer b c)
```

The first argument of `genServer` specifies the initialisation and the result is the reference to server. By means of a standardised interface, we pass arguments to the server functionality using the function `call :: GenServer a b -> a -> IO b`. The server function is then invoked inside the server thread and the result is returned in a temporary MVar which is not visible to the caller.

For different functionalities inside the server one will often use an algebraic datatype specifying the different possible branches in the server. In the following example this is the type `DbMsg`. The program defines a small database server which can store and retrieve key/value-pairs. The functionality of the server is defined by means of the function `dbAction`.

```
-- Message definition
data DbMsg = Lookup String | Insert String String
```

```

-- Initialise server with empty database and enter main loop
database :: IO (GenServer DbMsg (Maybe String))
database = genServer (return []) dbAction

type Database = [(String,String)]

dbAction :: Database -> DbMsg -> IO (Database,Maybe String)
dbAction s (Lookup key) = return (s,lookup key s)
dbAction s (Insert key value) =
  case (lookup key s) of
    (Just _) -> return (s,Nothing)
    Nothing -> return (((key,value):s),Just value)

-- A simple client
client = do ...
  db <- database
  ...
  answer <- call db (Insert key value)
  case answer of
    Nothing -> putStr "Key already allocated"
    Just _ -> return ()

```

5.3 Supervision tree

Another concurrency abstraction adapted from Erlang is the so called supervision tree. Here, once again a distinct parent-child relationship is required. A supervisor thread controls a set of child threads. Each child thread can in turn be either yet another supervisor or a worker threads. If a supervised child dies, the following supervision behaviours can be configured for the supervisor:

- **One-for-all supervision:** If one child dies (unexpectedly), the supervisor terminates all other children as well and restarts the entire set afterwards.
- **One-for-one supervision:** Only the terminated child is restarted.

The gist of the different cases is clear: If the children are dependant on each other, for example because one manages a central database which the others can query, it makes sense to use the one-for-all supervision to bring the system back into a well-defined state. This can be more appropriate instead of providing the correct handling techniques for every possible failure in each thread separately.

On the other hand, if a one-for-one supervised child terminates, this indicates only a minor error situation (or maybe none at all) and allows the system to recover gracefully without resorting to restarting all other threads.

How can we adequately model this in Concurrent Haskell? Specifically, can we find a better way than throwing `ThreadKilled`-exceptions to the children? If we take a closer look to the Erlang version, we note that here the use of message-passing is prominent. Each child can receive a message indicating that it should restart. If the child does not react on this message, then there is still the

possibility of forcibly terminating the child. An implementation should ensure that this ‘termination message’ is indeed processed so that the child has cleanly exited before the parent decides to take the second step.

This has some implications for the structure of child processes: Implementing this scheme for a supervisor, e.g. by means of a Channel, is easy as the supervisor does not do any actual computation apart from monitoring its children. But the worker children have to be specifically crafted so that the incoming control message can be treated in time. If these children use Concurrent Haskell operations which can block, e.g. Channels and MVars, on their own, they might easily come into a situation where an administrative signal remains pending for too long. This can be avoided if the children adhere to a programming style based on message passing, i.e. they wait at a central location for either a peer or their supervisor to contact them.

This clearly requires a paradigm shift similar to the one we already observed with the Generic Server approach. Instead of using any tool available in any conceivable way, the programmer is required to limit himself to a subset thereof. As stated above, we consider this not necessarily a limitation but instead an advantage because it makes maintaining and understanding a program easier.

So what is the proposed API to look like? Again, let us consider the required features:

- The supervisor receives on startup a list of children to start, including some parameters on how to treat them;
- each child must have a way of accepting control requests from the supervisor apart from its actual work.

Which feature do we request for treating the death of children? Again, Erlang/OTP gives three different options: a *permanent* child always implies a restart, a *transient* child is only restarted if it terminated abnormally and a child marked *temporary* is never restarted at all. We can model these flags adequately in an algebraic datatype in Haskell. Additionally to the type, each child has to be a function which takes the control channel as argument.

We can thus define all necessary objects for the `supervisor` function as follows:

```
data Control -- abstract
data ChildType = Permanent | Transient | Temporary
data Child = Child ((Control -> IO ()), ChildType)

data SupervisorType = OneForOne | OneForAll
supervisor :: SupervisorType -> [Child] -> IO ()
```

The implementation of the main parts, that is the supervisor function proper and one of the two functions for starting children either in `OneForOne` or `OneForAll` mode, can be given as follows:

```
supervisor :: SupervisorType -> [Child] -> IO ()
supervisor flag children = do
  -- the following MVar indicates if a child request a global restart
  mv_restart <- newEmptyMVar
```

```

-- this MVar signals that children should not restart themselves
mv_exitAll <- newEmptyMVar
-- start all children using a small helper function which sets up and
-- passes some additional data structures (not shown)
handles <- mapM (wrapfork $ runChild mv_restart mv_exitAll flag)
             children
-- wait until a child signals a restart
takeMVar mv_restart
-- signal to other children
putMVar mv_exitAll ()
-- signal all children that they should terminate
mapM shutdown (zip handles children)
supervisor flag children

runChild :: MVar () -> MVar () -> SupervisorType -> MVar () -> Child
          -> Chan Control -> IO ()
runChild mv_restart mv_exitAll OneForOne mv_childExit child ch = do
  -- disable unwanted exceptions
  block $ do
    result <- catch (unblock ((io child) ch) >> return True)
                (\ e -> return False)
    -- did parent tell us to exit or may we loop?
    exitAll <- isEmptyMVar mv_exitAll
    let restart = (not exitAll) ||
                  ((ctype child) == Permanent) ||
                  (((ctype child) == Transient) && (not result))
    if restart
      -- okay, nothing serious, restart child
      then runChild mv_restart mv_exitAll OneForOne mv_childExit child ch
      -- we have to leave
      else putMVar mv_childExit ()

```

On top of these supervision principles, another important abstraction can be modelled: The system should be able to limit the rate with which the different services are restarted. Frequent failures of the same subsystem or child usually indicate that there is something amiss and an intervention might be required. The usual procedure apart from notifying the operator is to delay the restart of the problematic child for some time. Such behaviour is frequently found in network services

Here we could apply some higher-order techniques from Haskell: If we assume that we receive notifications of required restarts with timestamps in a list, we can use a simple `foldr` to decide the current state our system is in.

The supervisor would need a function which takes the current state of the fold, the service requesting a restart and decides based on those data and the current time whether a child can safely be restarted:

```
allowRestart :: ServiceState -> Service -> IO Bool
```

6 Future work

Some fields of investigation lend themselves to further investigation:

We already mentioned in Section 3 that using the `ThreadKilled` exception for ‘notification’ purposes is rather crude. Haskell’s Dynamic Exceptions would be appropriate to obtain a finer degree of control. As a detailed description of Dynamic Exceptions is beyond the scope of this paper we think that this approach should separately be covered in more detail on a single abstraction like the parent/child abstraction.

The natural extension of those abstractions stretches beyond the Haskell runtime system level into the operating system level. In the previous paragraphs a kind of monitoring mechanism helps to build dependent systems and decrease the amount of programming necessary for some forms of synchronisation. Especially the parent/child relationship is important in real-world applications where the child is delegated to perform a specific task and report a result to the parent. Even if the result is not important to the parent (because the desired results are side-effects, e.g. when using the Haskell function `System.system` which executes an arbitrary command as a new process), it is usually vital to be informed if an error occurs and the child terminates abnormally.

In the realm of (Posix [5]) processes, the default behaviour of a forked child is to notify its parent that it terminated. Furthermore, the parent may query the exit code of the child which contains an integer value to convey information. Clearly, this is an inferior way of passing information between processes.

But that is not the only problem in the face of heavy-weight processes: Not only passing a return value is severely limit, but the means of communication between parent and child is extremely complicated. The only ways of passing data hence and forth are either via file descriptor (where the data must be converted to binary form) or Unix or Internet domain sockets. As there is no way of sharing memory (e.g. for a heap), it is impossible to transmit Haskell-specific things like functions. It is only possible to transmit values which are ‘serialisable’ similar to the way the class `Show` works in Haskell (cf. Implementation of Port-based Distributed Haskell [7]).

7 Conclusion

In this article we present a library of concurrency abstractions for the non-strict functional language Haskell. Various patterns of concurrent and distributed programming can be subsumed by a few succinct functions. Although this sometimes limits the programmer to only a subset of features found in the base language (here Concurrent Haskell), we feel that these abstractions aid in encapsulating communication, make programs easier to understand and increase the possibility of reusing components. The library is available from <http://www-i2.informatik.rwth-aachen.de/~stolz/Haskell/>.

References

1. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall Europe, Hemel Hempstead, Hertfordshire, 2nd edition, 1996.
2. OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface, Version 2.0. <http://www.openmp.org/specs/mp-documents/cspec20.pdf>, 2002.
3. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Pena-Mari. Eden — the paradise of functional concurrent programming. In *Proceedings of Second International EuroPar Conference*, volume 1123 of *Lecture Notes in Computer Science*, page 710ff., 1996.
4. E.W. Dijkstra, W.H. Feijen, and A.J. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letter*, 16, 1983.
5. The Open Group. Single Unix Specification v3. <http://www.opengroup.org>, 2001.
6. F. Huch and U. Norbistrath. Distributed Programming in Haskell with Ports. In *Proceedings of the 13th International Workshop on Implementation of Functional Languages (IFL 2000)*, volume 2011 of *Lecture Notes in Computer Science*. Springer, 2000.
7. F. Huch and V. Stolz. Implementation of Port-based Distributed Haskell. In T. Arts and M. Mohnen, editors, *Draft proceedings of the 13th International Workshop on Implementation of Functional Languages*, September 2001.
8. S. Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. <http://research.microsoft.com/~simonpj/#marktoberdorf>, January 2001.
9. S. Peyton Jones et al. Haskell 98 report. Technical report, <http://www.haskell.org/>, 1998.
10. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Conference Record of POPL '96: The 23rd ACM Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.
11. libraries@haskell.org. Haskell Core Library `Control.Exception`. <http://www.haskell.org/ghc/docs/latest/html/base/>, 2002.
12. H.-W. Loidl and P.W. Trinder. Engineering large parallel functional programs. *Lecture Notes in Computer Science*, 1467:178–197, 1998.
13. S. Marlow, S. Peyton Jones, and A. Moran. Asynchronous exceptions in Haskell. In *4th International Workshop on High-Level Concurrent Languages (HLCL'00)*, Montreal, Kanada, September 2000.
14. R.F. Pointon, P.W. Trinder, and H.-W. Loidl. The Design and Implementation of Glasgow distributed Haskell. In M. Mohnen and P. Koopman, editors, *Proceedings of the 13th International Workshop on Implementation of Functional Languages (IFL 2000)*, volume 2011 of *Lecture Notes in Computer Science*. Springer, 2000.
15. V. Stolz. Robuste verteilte Programmierung in Haskell. Master's thesis, RWTH Aachen, 2001.
16. A. S. Tannenbaum. *Modern Operating Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.
17. The GHC Team. The Glasgow Haskell Compiler. <http://www.haskell.org/ghc/>, 2002.