Asynchronous Exceptions in Haskell

Simon Marlow and Simon Peyton Jones
Microsoft Research, Cambridge

Andrew Moran
Oregon Graduate Institute

Abstract

Asynchronous exceptions, such as timeouts, are important for robust, modular programs, but are extremely difficult to program with — so much so that most programming languages either heavily restrict them or ban them altogether. We extend our earlier work, in which we added synchronous exceptions to Haskell, to support asynchronous exceptions too. Our design introduces scoped combinators for blocking and unblocking asynchronous interrupts, along with a somewhat surprising semantics for operations that can suspend. We give a formal semantics for our system, along with the first steps towards a theory that enables us to prove program properties. Using our primitives we can define combinators that support (for example) robust access to shared data; and using the theory we can prove that they have no race conditions, regardless of the behaviour of the program's environment. Shazam!

1 Introduction

Exceptions have come to be recognized as an important programming language feature for building robust systems. When we say "exception", we normally mean "synchronous exception" in the sense that an exception can only be raised as a direct consequence of executing the program itself.

In contrast, *asynchronous exceptions*, such as a time-out or Control-C interrupt, occur as the result of some external stimulus. This presents a number of problems for the language designer and the programmer. If asynchronous exceptions are allowed at all, then by definition they can occur at any point during execution, so the programmer has to worry about data structures being left in a partially-modified state after receiving an asynchronous exception, and what happens if another asynchronous exception is received before processing of the first one is complete. For these reasons, amongst others, many languages do not support asynchronous exceptions at all, and of those that do, most tend to support a semi-asynchronous model. Semi-asynchronous exceptions can only be delivered at certain programmer-specified safe points in the code (see Section 13).

However, true asynchronous exceptions have some attractive properties, particularly in a multi-threaded language. If we have true asynchronous exceptions, we can terminate threads at any time, we can assign time limits to arbitrary computations, and we can notify the program of changes in its environment that may affect it, or resource limits that it is about to exceed. In this paper, we present an extension to Concurrent Haskell [12] to support true asynchronous exceptions. The contributions of the paper can be summarised as follows:

- We propose an extension to Concurrent Haskell and our existing exceptions mechanism that supports asynchronous delivery of exceptions between threads (Section 5).
- We show how, with asynchronous exceptions, one thread can terminate another and a generic, composable time-out combinator can be defined (Section 6).
- Motivated by some subtle race conditions, we introduce a control mechanism for postponing the delivery of asynchronous exceptions, based around two scoped combinators, `block` and `unblock` (Section 7). We also give a surprising semantics for operations that may block indefinitely, and show how together these techniques enable us to acquire and release locks safely in the presence of asynchronous exceptions (Section 8).
- We give the definitions of some useful combinators built on top of the low-level exception primitives (Section 9).
- We give an operational semantics for Concurrent Haskell extended with asynchronous exceptions (Section 10), and propose a theoretical framework for proving robustness properties of code fragments (Section 11).
- We outline an implementation of asynchronous exceptions and the associated primitive operations (Section 12).

It is unusual to support fully-asynchronous exceptions; it is even more unusual to have a formal specification of the resulting system, and one that is sufficiently tractable to prove useful properties of programs. There is much left to do, but we believe that our proposal represents a sound starting point.

2 Asynchronous exceptions

An exception is described as synchronous if it is one that arises as a result of evaluating the expression in which it is raised. Examples include: divide by zero, pattern-match failure, and explicitly raising a user exception by calling `raise`. Synchronous exceptions are relatively tractable:

- The denotation, or meaning, of an expression says whether evaluating the expression will raise a synchronous exception and, if so, specifies the set of exceptions that may be raised. In other words, the synchronous exceptions that an expression may raise is properly part of the semantics of that expression.
- It follows that a compiler can reasonably infer (an approximation to) the set of synchronous exceptions that any given expression could possibly raise [11]. For example, Java requires the programmer to list the possible exceptions raised by each method in a `throws` clause; if any are omitted, the compiler will flag an error.

Asynchronous exceptions, on the other hand, are raised as the result of an "external event", such as the user typing Control-C, and can occur at any point during execution. So in principle, the evaluation of *any* expression could yield an asynchronous exception; we cannot sensibly consider asynchronous exceptions as part of the semantics of the expression. This property makes asynchronous exceptions much less tractable, both semantically and from an implementation standpoint than synchronous exceptions.

If asynchronous exceptions are so awkward, why don't we ban them? After all, several programming languages have explicitly disallowed asynchronous exceptions because of their dangerous nature. For example, Java [2] introduced asynchronous exceptions in the form of the `java.lang.Thread.stop()` method, then later deprecated the feature on the grounds that it was too dangerous [19].

Nevertheless, there are several compelling reasons to support asynchronous exceptions:

Resource exhaustion. Most Haskell implementations use a stack and heap, both of which are essentially finite resources, so it seems reasonable to inform the program when memory is running out, in order that it can take remedial action. The ideal way to do this appears to be to raise an exception.

Time-outs. A time-out is another example of resource exhaustion; if some computation does not complete within a specified time budget, it should be aborted. For example, a server application might close a connection to the client if it fails to respond within a certain time.

User interrupt. Interactive systems often need to cancel a computation that has already been started, for example when the user clicks on the "stop" button in a web browser.

Killing the children. There is no way in Concurrent Haskell, as it stands, to kill a running thread. A parent thread might start a child thread, and then, in response to some event (some new input, or an exception), it may want to kill that thread.

Without true asynchronous exceptions, we can only do these things with co-operation from the code that is to receive the exception. For example: to terminate a thread we might set a global flag, and rely on the thread to periodically check the flag. For example, Modula-3 supports `Alerts`; the the procedure `TestAlert()` polls the thread's alert-pending bit. This is no accident: as Modula-3's designers say "if `t` could be alerted asynchronously, it would be essentially impossible for it to maintain the invariants of its critical sections" [4]. We discuss our solution to this problem in Section 7.

All of the above motivations concern the premature abortion of a computation. We do not deal with *resumption*, in which the interrupted computation can be resumed.

3 Catching and throwing exceptions

First, we briefly review our existing proposal for adding synchronous exception facilities to Haskell [13][1]:

```
catch :: IO a -> (Exception -> IO a) -> IO a
raise :: Exception -> a
throw :: Exception -> IO a
```

Exceptions can be caught with `catch`, where the first argument is the computation to run (*i.e.* the scope of the `catch`), and the second argument is the exception handler. In non-`IO` code, exceptions are raised with `raise`, whereas in `IO` code, exceptions can be raised with `throw`[2]:

The `Exception` type is a fixed datatype with a constructor for each built-in kind of exception:

```
data Exception =
    ...
  | StackOverflow       Integer
  | HeapOverflow        Integer
  | TimeOut             ThreadId
  | Interrupted
  | ThreadKilled
    ...
```

4 Concurrent Haskell

Asynchronous exceptions are particularly desirable in a concurrent programming language, because it is often natural for one thread to kill another, usually when the former has earlier forked the latter. Conversely, if one wishes to propagate a synchronous exception that reaches the "top" of a thread, it will become an asynchronous exception to that thread's parent. Accordingly, we briefly divert to review Concurrent Haskell, an extension of Haskell that supports concurrency [12].

Concurrent Haskell extends standard Haskell with a small set of primitives for creating new threads and performing simple inter-thread communication:

```
forkIO   :: IO a -> IO ()
sleep    :: Int -> IO ()
```

[1] Note that we've changed the names of some of the functions for the purposes of this paper.

[2] It may seem that `throw` is an instance of `raise` and is therefore superfluous, but in fact they have subtly different semantics: `seq (raise e) (...)` raises exception `e`, but `seq (throw e) (...)` does not. The distinction is also important for the semantics as we will see later.

```
data MVar a  -- abstract, instance of Eq
newEmptyMVar :: IO (MVar a)
putMVar      :: MVar a -> a -> IO ()
takeMVar     :: MVar a -> IO a
```

A new thread can be "forked" using `forkIO`, with the informal understanding that the `IO` computation passed to `forkIO` may be arbitrarily interleaved with the current computation. Both cooperative and preemptive multitasking implementations of Concurrent Haskell exist. Threads can sleep for a specified period of time (in microseconds) by calling `sleep`.

`MVar`s are a generic synchronisation mechanism, similar to the M-structures of Id [3]. A value of type `MVar t` can be thought of as a box that can be in two possible states: empty or containing a value of type `t`.

The `takeMVar` operation waits if it finds the box empty, or removes and returns its contents otherwise. The `putMVar` operation puts a new value in the box, waking up any threads that were waiting for the `MVar` to become full. Applying `putMVar` to a full `MVar` is an error, and causes the `PutFullMVar` exception to be raised.

Using only `MVar`s, many complex datatypes for concurrent communication can be built, including typed channels, semaphores and so on. One common use for `MVar`s is to synchronise access to a shared mutable data structure, such as a mutable array. We store the array in an `MVar`; any thread wishing to access the array must first take it from the `MVar`, and put it back afterwards. Hence only a single thread has access to the array at any one time.

5 Raising an exception asynchronously

We are now ready to work on asynchronous exceptions. We'll start by adding some primitives to Concurrent Haskell to enable one thread to asynchronously raise an exception in another. We need a way to refer to threads, so we introduce the abstract `ThreadId` type, and change `forkIO` to return the `ThreadId` of the newly created thread.

```
data ThreadId -- abstract
forkIO     :: IO a -> IO ThreadId
```

We also need a way to get the `ThreadId` of the current process:

```
myThreadId :: IO ThreadId
```

Next, we provide a way to raise an exception in another thread:

```
throwTo :: Exception -> ThreadId -> IO ()
```

Informally, the meaning of `throwTo e t` is that the exception `e` is raised in thread `t` as soon as possible, and `throwTo` only returns when the exception has been successfully raised. If the thread `t` has already died or completed, then `throwTo` trivially succeeds.

Note that implementing `throwTo` isn't as simple as it might seem: the target thread may be blocked, perhaps on an `MVar`, and will therefore have to be woken up before the exception can be raised. The implementation of `throwTo` is covered in more detail in Section 12.

A call to `throwTo` is synchronous, in the sense that it waits for the exception to be delivered before returning. This is important because it guarantees that two threads simultaneously attempting to raise exceptions in each other can't both succeed. It also means that the implementation doesn't need to maintain a queue of "pending exceptions" in the case where the target thread isn't in a state where it can receive an exception (perhaps because it is currently running on another processor, for example).

We now have a way to kill a thread:

```
killThread :: ThreadId -> IO ()
killThread t = throwTo ThreadKilled t
```

6 Time-outs

Let's examine the use of asynchronous exceptions in Concurrent Haskell by taking the example of time-outs: the problem of how to assign an arbitrary time-out to a computation. In other words, we want to define a function `timeout`, with the following type:

```
timeout :: Int     -- time limit
        -> IO a    -- computation to run
        -> IO (Maybe a)
```

More precisely, here is the behaviour we want:

- the semantics of `timeout t action` are equivalent to `(action >>= return . Just)`, provided `action` can be completed in approximately `t` microseconds or less.
- in the event that `action` cannot be completed in time `t`, the running computation `action` will receive an exception `TimeOut _`, which it may use to clean up. The result of the whole computation will then be replaced by `return Nothing`.

One consequence of these requirements is that if `action` raises an exception (before the time limit has expired), then `timeout t action` must raise the same exception.

It is also important that we can nest calls to `timeout` without them affecting each other.

6.1 (Incorrect) implementation of `timeout`

The basic idea is to fork a new thread (called the *timeout thread*) which will sleep for the specified time period, while the parent executes the computation. The timeout thread raises an exception in the parent when the timer expires.

The code for `timeout` is given in Figure 1. Take a moment or two to examine the implementation; it is quite subtle. After finding our own `ThreadId` with `myThreadId`, we fork off the timeout thread to do the waiting. The timeout thread simply blocks for `secs` seconds and then raises the `TimeOut` exception in the parent thread. The `TimeOut` exception is tagged with the `ThreadId` of the timeout thread, so it can be uniquely distinguished from `TimeOut` exceptions generated by other instances of `timeout`.

Meanwhile, the parent thread starts a new exception handler with `catch`, and begins to execute `action`. On receiving a result, it kills the timeout thread and returns the result to the caller.

If the timer expires before `action` has completed, the `TimeOut` exception will be received by the parent thread, and we end up in the exception handler. The exception handler checks the exception, and if it is indeed our `TimeOut`, we return `Nothing`. If we caught an exception, but it wasn't the `TimeOut`, we have to be sure to kill the timeout thread

```
timeout secs action = do {
    parent  <- myThreadId;
    timeout <- forkIO (timeout_thread parent);
    catch
      ( do { result <- action;
             killThread timeout;
             return (Just result);
        })
      ( \exception ->
          case exception of
            TimeOut id | id == timeout ->
              return Nothing
            other -> do { killThread timeout;
                          throw exception
                     })
 }

timeout_thread parent = do {
   sleep (secs * 1e6);
   id <- myThreadId;
   throwTo parent (TimeOut id)
 }
```

Figure 1: `timeout`, version 1.

```
timeout secs action = do {
    parent  <- myThreadId;
    ready   <- newEmptyMVar;
    timeout <- forkIO (timeout_thread parent ready);
    catch
      ( do { putMVar ready ();
             result <- action;
             killThread timeout;
             return (Just result);
        })
      ( \exception ->
          case exception of
            TimeOut id | id == timeout ->
               return Nothing
            other -> do { killThread timeout;
                          throw exception
                     })
 }

timeout_thread parent ready = do {
   sleep (secs * 1e6);
   id <- myThreadId;
   takeMVar ready;
   throwTo parent (TimeOut i)
 }
```

Figure 2: `timeout`, version 2.

to ensure we can't receive any `TimeOut` exceptions from it in the future. In other words, we are ensuring that the `TimeOut` exception can't "leak" outside `timeout`.

6.2 What is wrong with `timeout`

This implementation of `timeout` works fine, except for an annoying race condition: if the timeout is unusually short, or some scheduling anomaly means that the parent thread receives hardly any CPU time before the timer expires, then the parent thread can receive the `TimeOut` exception before installing the exception handler (*i.e.* before the `catch`).

We can close this hole by forcing the timeout thread to wait until the exception handler has been installed before raising its `TimeOut` exception. This is a straightforward synchronisation task, and we can use an `MVar` to achieve the desired effect — Figure 2 contains the new version of `timeout`.

Now we have stopped the `TimeOut` exception from being delivered too early, but what happens if it is delivered too late? There's another race condition in our code:

- If `action` raises a exception (synchronous or asynchronous), we end up in the exception handler. There is a small window between entering the exception handler and killing the timeout thread where we could potentially receive another exception. Since there is no active exception handler at this point, either:
  - it is our `TimeOut` exception, which will leak, or
  - it is a different asynchronous exception, in which case the timeout thread will continue running and raise its `TimeOut` exception at some later time.

How can we close this race hole? There are several possibilities:

- Ensure that there is always an exception handler in force, right up until the point when `timeout` returns. Unfortunately, this isn't possible because there will always be a small window between starting to execute the existing exception handler and installing a new exception handler, during which an asynchronous exception may strike.
- Try to ensure that the timeout thread can't send its exception at an awkward time, by using an `MVar` for synchronisation again. This seems plausible — perhaps if we take the `ready` lock again at the right moment, we might be able to prevent the timeout thread from acquiring the lock and sending the exception. However, no matter where we try to put another `takeMVar`, we still can't close the window. If `action` raises an exception, we always end up in the exception handler and we can't kill the timeout thread or acquire the `ready` lock fast enough.

This leads us to the conclusion that some more primitive control over the delivery of asynchronous exceptions is needed.

7 Blocking Asynchronous Exceptions

The problem of robust programming in the presence of asynchronous exceptions is by no means a new one. Operating system code has always had to cope with asynchronous interrupts accessing global mutable data, for example. The way this is normally dealt with is to disable interrupt servicing during critical sections. We propose an similar approach for Concurrent Haskell, using two combinators:

```
timeout secs action = do {
    parent  <- myThreadId;
    block ( do {
       timeout <- forkIO (timeout_thread parent);
       catch
         ( unblock (
             do { result <- action;
                  killThread timeout;
                  return (Just result);
             }))
         ( \exception ->
             case exception of
               TimeOut id | id == timeout ->
  return Nothing
               other -> do { killThread timeout;
                             throw exception
                        })
       })
 }

timeout_thread parent = do {
   sleep (secs * 1e6);
   id <- myThreadId;
   throwTo parent (TimeOut i)
 }
```

Figure 3: `timeout`, version 3.

```
   block   :: IO a -> IO a
   unblock :: IO a -> IO a
```

Applying `block` to a computation will execute the computation with asynchronous exceptions *blocked*; that is, any thread attempting to raise an exception in the current thread will block until asynchronous exceptions are enabled again. System-generated asynchronous exceptions are likewise disabled in the scope of the `block`. The programmer doesn't need to worry about re-enabling interrupts; this is done automatically on exiting the scope of `block`.

Applying `unblock` to a computation re-enables the delivery of asynchronous exceptions (if they are currently blocked) while its argument is being executed.

The two combinators can be arbitrarily nested. If a `block` is executed in a context where exceptions are already blocked, it is treated as a no-op[3].

7.1 Fixing `timeout`

How do these new combinators help solve our problems with `timeout`? Figure 3 contains the new code for `timeout`, using blocking of asynchronous exceptions to close the race holes.

The changes made relative to the previous version are:

- we enclosed most of the body of the function in an application of `block`;
- we enclosed the first argument of `catch` in `unblock`;
- we removed the `MVar` used to synchronise the timeout thread with the installation of the exception handler in the parent thread.

[3] An alternative approach would be to use a reference count: two nested `block` would require two nested `unblock` to re-enable exception delivery. We considered this alternative, but didn't find any compelling reason to adopt the more complicated approach.

The `block` ensures that the parent thread cannot receive any asynchronous exceptions until we enter the scope of the `unblock`; and by this time we have installed an exception handler ready to catch the exception[4].

But the scope of the `block` also covers the exception handler itself. If an exception is received during `action`, whether synchronous or asynchronous, we end up in the exception handler *with asynchronous exceptions blocked*, even though the `unblock` enabled exceptions during `action`. So there is now no window between entering the exception handler and killing the timeout thread during which we could receive an unexpected exception.

If it wasn't a `TimeOut` exception, and the timeout thread is in fact waiting to send the `TimeOut` exception to the parent, then the parent can still kill the timeout thread and the exception will never be raised.

Using the scoping semantics of `block` and `unblock` we have managed to block exceptions during the critical parts of the code only, and leave the intended semantics of the operation as a whole unaffected.

8 Safe Locking

There are some other races in Concurrent Haskell with asynchronous exceptions that are impossible to avoid without the ability to block asynchronous exceptions. For example, if we use an `MVar` as a lock (with type `MVar ()`), then it is common practice when acquiring the lock to install an exception handler to release the lock in case of an error. The code to do this might look like:

```
   takeMVar m ;
   catch action
         (\e -> do { putMVar m (); throw e }) ;
   putMVar m ()
```

What happens if an asynchronous exception is received just after the `takeMVar`? The exception is propagated, and the lock is held indefinitely. If we rewrite the code so that the exception handler encompasses the `takeMVar`, then we have a similar problem if the asynchronous exception is received just before the `takeMVar`.

Let's try to apply `block` and `unblock` from the previous section:

```
   block ( do {
      takeMVar m ;
      catch (unblock action)
            (\e -> do { putMVar m (); throw e }) ;
      putMVar m ()
   })
```

This appears to solve our original problem: there are now no windows of vulnerability during which the thread could terminate with the lock still in hand. But there's a new problem, namely that if the thread has to wait for the lock, it now waits in a state where asynchronous exceptions cannot be delivered to it. This is in violation of one of the cardinal

[4] Note that the scope of the `block` does not extend to threads forked by `forkIO`: new threads always start with asynchronous exceptions unblocked.

rules of concurrent programming: "don't block while holding a lock", the reason being that it increases the potential for deadlock. More practically, it means we can't time-out the thread waiting for the lock until it acquires the lock. We ought to be able to do much better than this.

How do we take the `MVar`, and atomically install an exception handler as soon as we have the `MVar`? We initially thought that the solution to this problem was to introduce a new primitive operation, essentially combining `takeMVar` and `block` into a single atomic operation. While this solution worked, it was somewhat kludgey, and furthermore there are several other situations that require atomically waiting for a resource and installing an exception handler to release it later (*e.g.* accepting connections from the network in a server application). Rather than add new primitives for each of these situations, we realised that a subtle change to the semantics provides the required functionality:

> Any operation which may block indefinitely (*e.g.* `takeMVar`) may receive asynchronous exceptions while blocked, even within an enclosing `block`. Such operations are termed *interruptible operations.*

Some justification for this decision:

- Without this change, `takeMVar` is unsafe to use inside `block` at all.
- We don't need to introduce a new primitive to solve the locking problem above.
- Although it seems strange that operations inside a `block` may raise asynchronous exceptions, the exceptions are somewhat synchronous in nature since we can specify exactly which operations are interruptible.
- `takeMVar` already raises at least one kind of exception: if the `MVar` being taken is empty and this thread has the only reference to it, then the `takeMVar` will block for ever. The garbage collector can detect this condition and cause `takeMVar` to raise an `InfiniteBlock` exception.

The code to acquire the lock as given above works fine with this addition to the semantics, with the difference that now the `takeMVar` operation is interruptible.

In Concurrent Haskell, `MVar`s are commonly used to hold an immutable structure, but they can equally well be used as a semaphore to protect a mutable structure (standard Haskell doesn't have mutable structures, but many compilers support them as extensions). If an `MVar` is being used to protect a shared mutable data structure, such as a mutable array, then the chances are that we don't want to be disturbed at all while we operate on it, because an exception received during the operation may leave the mutable data structure in an inconsistent state. In this case, it makes sense to omit the call to `unblock` in the above code fragment. But what if `action` is going to take a long time? Then we have to explicitly program checkpoints into the code such that `action` will receive any pending asynchronous exceptions at designated safe points during execution. The easiest way to implement a safe point is to `unblock` for a short period of time:

```
safePoint :: IO ()
safePoint = unblock (return ())
```

Note that the semantics (Section 10) doesn't specify that `unblock` automatically unblocks all threads currently waiting to raise asynchronous exceptions in the current thread, because it has no concept of blocked threads. However, in an implementation it is straightforward to ensure that as soon as the current thread enters an `unblock`, we unblock and schedule any threads waiting to raise exceptions.

9 Coding for asynchronous exceptions

The features introduced in Section 7 and Section 8 are expressive but rather low-level. We do not advocate using programming with them directly; instead, we hope to build a library of robust abstractions, layered on top of the primitives, that express common programming patterns.

The `timeout` combinator we discussed at length earlier is an example of such a combinator. Another is `finally`, which embodies the concept of "do A, then whatever happens do B":

```
finally :: IO a -> IO b -> IO a
```

A possible implementation of `finally` is:

```
finally a b =
   block (do {
     r <- catch (unblock a)
                (\e -> do { b; throw e });
     b;
     return r;
   })
```

In fact, `finally` is an instance of a more general combinator, `bracket`:

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO ()
```

`bracket` is useful for the class of tasks of the form "acquire a resource, operate on it, free the resource". We want the resource to be freed if either the operation succeeds or raises an exception. For example, consider opening a file:

```
bracket (openFile "file.tmp")
        (\h -> hClose h)
        (\h -> workOnFile h)
```

Using `bracket` here makes sure that the file will always be closed, regardless of what exceptions are flying around. Furthermore, it makes sure that the `openFile` operation behaves atomically: it either succeeds, in which case we have acquired the resource, or raises an exception, in which case we haven't. Note that `openFile` is an interruptible operation.

The `forkIO` primitive is very asymmetrical: it forks a child, while the parent continues in parallel. Here are two more symmetrical forms of forking:

```
either :: IO a -> IO b -> IO (Either a b)
both   :: IO a -> IO b -> IO (a,b)
```

`either` executes both of its arguments concurrently, and returns the result from the first one to finish; the other thread is sent a `KillThread` exception. `either` generalises timeout:

```
timeout secs action
  = do { a <- either (sleep secs) action;
         case a of
              Left _  -> return Nothing
              Right r -> return (Just r) }
```

$$
\begin{array}{rcl}
x, y & \in & \textit{Variable} \\
k & \in & \textit{Constant} \\
c & \in & \textit{Constructor} \\
ch & \in & \textit{Char} \\
d & \in & \textit{Integer} \\
e & \in & \texttt{Exception} \\
m & \in & \texttt{MVar} \\
t, u & \in & \texttt{ThreadId} \\
V & ::= & x \mid \texttt{\\}x\;\texttt{->}\;M \mid k \mid c\,M_1 \cdots M_n \mid ch \mid d \mid e \mid m \mid t \mid \\
 & & \texttt{return}\ M \mid M \texttt{>>=} N \mid \\
 & & \texttt{putChar}\ M \mid \texttt{getChar} \mid \\
 & & \texttt{forkIO}\ M \mid \texttt{myThreadId} \mid \texttt{sleep}\ M \mid \\
 & & \texttt{putMVar}\ M\,N \mid \texttt{takeMVar}\ M \mid \texttt{newEmptyMVar} \mid \\
 & & \texttt{throw}\ M \mid \texttt{throwTo}\ M\,N \mid \\
 & & \texttt{catch}\ M\,H \mid \\
 & & \texttt{block}\ M \mid \texttt{unblock}\ M \\
M, N, H & ::= & x \mid V \mid M\,N \mid \cdots
\end{array}
$$

Figure 4: The syntax of values and terms.

`both` also evaluates its arguments concurrently, but waits for them both to terminate before returning the results in a pair. Structured combinators like these should satisfy useful algebraic laws – but we ought to prove such laws, and we turn to that next.

10 Operational Semantics

The preceding sections should have convinced you that it is hard to be absolutely certain that one has closed all the holes in a concurrent program. A first step towards such certainty is to provide a precise, formal specification for all the primitives, and that is what we do in this section. Then we would like to develop a theory based on this specification, so that we can reason about the safety properties of our programs; we discuss this in Section 11.

Our semantics is presented in an operational style, in two levels: an *inner operational semantics* that describes the behaviour of pure terms (treating members of the `IO` monad as uninterpreted constants), and *an outer monadic transition semantics* that describes the behaviour of `IO` computations (including `MVar`s, processes, and the exceptions mechanism). The inner semantics was described in detail in [10]; here, we focus on the outer monadic transition semantics for the exceptions mechanism and concurrency.

We consider a simplified version of Haskell: our language has the usual features of a lazy functional language (lambda abstraction, application, data structures, case expressions, *etc.*), augmented with constants corresponding to `IO` operations and the elements of the `Exception` and `ThreadId` types. We will only present those elements of the syntax that are relevant to the semantics; other aspects (such as how we represent lists, or how to write a case expression) wouldn't aid comprehension of the semantics, and are not presented.

In this section, M, N and H range over terms in our language, and V ranges over all terms that the inner semantics treats as values, such as lambda abstractions, constants, and `IO` operations (see Figure 4). By treating `IO` operations as lazy constructors in the inner semantics, we reinforce the distinction between an `IO` computation and its effect on the environment when performed[5]. The inner semantics describes the evaluation (to WHNF) of an `IO` computation; the outer semantics describes its effect when performed.

$$
\begin{array}{rcll}
a & ::= & \langle\rangle_m & \text{empty } \texttt{MVar} \text{ named } n \\
 & \mid & \langle M\rangle_m & \text{full } \texttt{MVar} \text{ named } n\text{, holding } M \\
P, Q, R & ::= & (\!| M |\!)_t & \text{thread of computation named } t \\
 & \mid & \mathbb{0}_t & \text{finished thread named } t \\
 & \mid & a & \text{full or empty } \texttt{MVar} \\
 & \mid & \nu x.P & \text{restriction} \\
 & \mid & P \mid Q & \text{parallel composition}
\end{array}
$$

Figure 5: The syntax of processes.

In the following sections, we will confuse m `::` `MVar` with the name of that `MVar`, and t `::` `ThreadId` with the name of the thread. We will also treat `MVar` and thread names as normal variables (*i.e.* they may be bound and α-converted).

10.1 Process-level Transitions

Since this is *Concurrent* Haskell, in general we will have many `IO` computations, executing concurrently, communicating via `MVar`s. We will define a transition relation, $\rightarrow$, which describes how a system of `IO` computation processes evolves. Processes are of the form presented in Figure 5. Atomic processes are threads of `IO` computation M, labelled with a thread identifier t, written $(\!| M |\!)_t$, or, if the thread has died or is finished, written $\mathbb{0}_t$. There is one distinguished thread, the main thread, denoted by $(\!| M |\!)_{\texttt{main}}$. `MVar`s, empty or full, are also processes. The scope of thread and `MVar` names is limited by restriction in the usual manner. Finally, we may combine processes in parallel. We write $\mathit{fn}\,(P)$ to mean the the set of free names in a process P.

We will define labelled transitions between processes, which will include labelled transitions between program terms. We will write

$$P \xrightarrow{\alpha} Q$$

where $\alpha \in \{\lightning e, \$d, \texttt{!}\,ch, \texttt{?}ch, \tau\}$, to mean that P becomes Q, under the influence of asynchronous signal α. $\lightning e$ indicates that an asynchronous exception e has been generated by the environment; $\$d$ is signal indicating that d milliseconds have passed; $\texttt{!}\,ch$ and $\texttt{?}ch$ reflect how `putChar` and `getChar` interact with the external world, and τ is the silent or opaque transition (usually indicating internal computation). We will usually elide τ labels from silent transitions, in the interests of reducing clutter.

Following [12], we define a structural equivalence over processes, formalising the idea of a "solution" of processes *a la* the chemical abstract machine [8]. Let $\equiv$ be the least congruence (*i.e.* equivalence relation preserved by all process contexts) that also satisfies the (standard) rules in Figure 6. Also presented in that figure are two structural transition rules that allow transitions within parallel compositions and

[5] From [14]: " ... type `IO a` denoting actions that, *when performed*, may do some I/O and then return a value of type `a`".

$$
\begin{aligned}
P \mid Q &\equiv Q \mid P \\
P \mid (Q \mid R) &\equiv (P \mid Q) \mid R \\
\nu x.\nu y.P &\equiv \nu y.\nu x.P \\
\nu x.P \mid Q &\equiv (\nu x.P) \mid Q, \qquad x \notin fn(Q)
\end{aligned}
$$

$$
\frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R} \qquad \frac{P \xrightarrow{\alpha} Q}{\nu x.P \xrightarrow{\alpha} \nu x.Q}
$$

$$
\frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha} Q}
$$

Figure 6: Structural congruence, and structural transitions.

$$
\begin{array}{lcl}
I & ::= & \texttt{getChar} \mid \texttt{putChar}\ ch \mid \texttt{sleep}\ d \mid \\
 & & \texttt{takeMVar}\ m \mid \texttt{throwTo}\ e\, t \\
\mathbb{P} & ::= & [\cdot] \mid \texttt{putChar}\ [\cdot] \mid \texttt{sleep}\ [\cdot] \mid \\
 & & \texttt{putMVar}\ [\cdot]\, M \mid \texttt{takeMVar}\ [\cdot] \mid \\
 & & \texttt{throw}\ [\cdot] \mid \texttt{throwTo}\ [\cdot]\, M \mid \texttt{throwTo}\ e\, [\cdot] \\
\mathbb{M} & ::= & \mathbb{P} \mid \mathbb{M} \texttt{>>=} M \mid \texttt{catch}\ \mathbb{M}\, H \\
\mathbb{U} & ::= & \mathbb{M} \mid \mathbb{E}[\texttt{unblock}\ \mathbb{M}] \mid \mathbb{E}[\texttt{block}\ \mathbb{M}[I]] \\
\mathbb{E} & ::= & \mathbb{M} \mid \mathbb{M}[\texttt{block}\ \mathbb{E}] \mid \mathbb{M}[\texttt{unblock}\ \mathbb{E}]
\end{array}
$$

Figure 7: The evaluation contexts.

inside restrictions, and a rule allowing structural equivalence to bring "reagents" together.

Transitions take place within *evaluation contexts*, and these are also crucial to our explanation of the blocking and unblocking of asynchronous exceptions. An evaluation context is a program (not process) context that contains a single hole (denoted by $[\cdot]$) which indicates where the next transition will take place. The four kinds of evaluation contexts are presented in Figure 7, and will be discussed as they are used in the semantics. We will write $\mathbb{C}[M]$ to mean that term M has been "plugged into" context $\mathbb{C}$. We assume that the resultant term $\mathbb{C}[M]$ is well-typed. As usual, free variables in M may be captured by $\mathbb{C}$.

10.2 Internal Transitions

Many transitions are internal to a particular process and reflect the sequencing of `IO` computations within that process. In this section, we will describe all such transitions, writing $M \rightarrow N$ to mean that M silently becomes N. We lift these transitions to the process level with the help of evaluation contexts thus:

$$
\frac{M \rightarrow N}{(\!|\mathbb{E}[M]|\!)_t \xrightarrow{\tau} (\!|\mathbb{E}[N]|\!)_t}
$$

$\mathbb{E}$-contexts are defined in Figure 7, and encompass all possible evaluation contexts, including $\mathbb{P}$-, $\mathbb{M}$-, and $\mathbb{U}$-contexts. $\mathbb{P}$-contexts describe how certain `IO` operations evaluate their arguments. $\mathbb{M}$-contexts include $\mathbb{P}$-contexts and describe how `>>=` and `catch` evaluate their arguments. Discussion of $\mathbb{U}$-contexts (and the real motivation for $\mathbb{M}$-contexts) is deferred until Section 10.6.

10.2.1 Promoting the Inner Semantics.

The inner operational semantics of [10] defined two relations over terms:

- *Convergence* of terms is written $M \Downarrow V$, meaning that closed term M evaluates to value V.
- *Exceptional convergence*, written $M \uparrow e$, means that closed term M *may raise* exception e.

Apart from describing call-by-name evaluation of our language, the inner semantics also describes how exceptions propagate and gives meaning to `raise`. For example, $\texttt{raise}\ e \uparrow e$, if e is an exception.

In a type-correct program, the outer monadic transitions semantics must refer to the inner semantics when the term in question has the type `IO a`, for some `a`. Two rules link the inner and outer semantics in this case:

$$
\frac{M :: \texttt{IO a} \quad M \not\equiv V \quad M \Downarrow V}{M \rightarrow V}
$$

$$
\frac{M :: \texttt{IO a} \quad M \uparrow e}{M \rightarrow \texttt{throw}\ e}
$$

In the first rule, we stipulate that $M \not\equiv V$ to prevent infinite sequences of the form $V \rightarrow V \rightarrow V \rightarrow \cdots$.

10.2.2 Simple Transitions.

Sequencing of `IO` operations is handled by `>>=`, which also allows exceptions to propagate:

$$
\begin{aligned}
(\texttt{return}\ N) \texttt{>>=} M &\rightarrow M\, N \\
(\texttt{throw}\ e) \texttt{>>=} M &\rightarrow \texttt{throw}\ e
\end{aligned}
$$

After the first rule has been applied, we will need to evaluate $M\, N$; this is where the two rules that link the inner and outer semantics come into play.

`catch` passes return values on untouched, and hands `IO` exceptions off to the given exception handler, with asynchronous exceptions blocked:

$$
\begin{aligned}
\texttt{catch}\ (\texttt{return}\ M)\, H &\rightarrow \texttt{return}\ M \\
\texttt{catch}\ (\texttt{throw}\ e)\, H &\rightarrow \texttt{block}\ (H\, e)
\end{aligned}
$$

When an `IO` computation is finished (either with a return value or an exception), `block` and `unblock` disappear:

$$
\begin{aligned}
\texttt{block}\ (\texttt{return}\ M) &\rightarrow \texttt{return}\ M \\
\texttt{unblock}\ (\texttt{return}\ M) &\rightarrow \texttt{return}\ M \\
\texttt{block}\ (\texttt{throw}\ e) &\rightarrow \texttt{throw}\ e \\
\texttt{unblock}\ (\texttt{throw}\ e) &\rightarrow \texttt{throw}\ e
\end{aligned}
$$

Note that we don't need to specify any change to the global state (*e.g.* signalling that asynchronous exceptions are no longer blocked), since this is entirely dependent upon scope.

Some of the simpler `IO` operations are strict in their arguments. How evaluation proceeds in those cases is defined by the shape of the primitive evaluation contexts $\mathbb{P}$, given in Figure 7. For example, `throwTo` evaluates its arguments from left to right. Since these IO operations have arguments that are general terms (*i.e.* not in the IO monad), we need to refer to the inner semantics:

$$\frac{M \not\equiv V \quad M \Downarrow V}{\mathbb{P}[M] \ \rightarrow\ \mathbb{P}[V]}$$

$$\frac{M \uparrow e}{\mathbb{P}[M] \ \rightarrow\ \texttt{throw}\ e}$$

If an exception is raised when evaluating an argument, then that exception is raised as an IO exception. The behaviour of these operations when their arguments are fully evaluated is deferred until subsequent sections.

10.3 Basic I/O

The canonical IO operations are putChar and getChar:

$$(\!|\mathbb{E}[\texttt{putChar}\ ch]|\!)_t \xrightarrow{!\,ch} (\!|\mathbb{E}[\texttt{return ()}]|\!)_t$$
$$(\!|\mathbb{E}[\texttt{getChar}]|\!)_t \xrightarrow{?\,ch} (\!|\mathbb{E}[\texttt{return}\ ch]|\!)_t$$

Both are dependent upon interaction with the outside world, and may therefore block indefinitely, motivating their inclusion in the set of interruptible operations. Other operations like openFile have analogous semantics.

10.4 Basic Concurrency

Forking a new thread is straightforward:

$$(\!|\mathbb{E}[\texttt{forkIO}\ M]|\!)_t \ \rightarrow\ \nu u.(\!|\mathbb{E}[\texttt{return}\ u]|\!)_t \mid (\!|M|\!)_u,\ \ u \notin \mathit{fn}\,(\mathbb{E}, M)$$

and any thread can easily gain access to its own ThreadId:

$$(\!|\mathbb{E}[\texttt{myThreadId}]|\!)_t \ \rightarrow\ (\!|\mathbb{E}[\texttt{return}\ t]|\!)_t$$

We deliberately underspecify sleep:

$$(\!|\mathbb{E}[\texttt{sleep}\ d]|\!)_t \xrightarrow{\$d} (\!|\mathbb{E}[\texttt{return ()}]|\!)_t$$

Here, the $\$d$ label represents an external clock interrupt, indicating that d milliseconds have passed. A correct implementation must guarantee that at least d milliseconds have passed before a process executing sleep d is woken; further delay is acceptable.

10.5 Synchronisation Primitives

MVars give rise to essentially the same transitions as in [12]:

$$\langle\rangle_m \mid (\!|\mathbb{E}[\texttt{putMVar}\ m\,M]|\!)_t \ \rightarrow\ \langle M\rangle_m \mid (\!|\mathbb{E}[\texttt{return ()}]|\!)_t$$
$$\langle M\rangle_m \mid (\!|\mathbb{E}[\texttt{takeMVar}\ m]|\!)_t \ \rightarrow\ \langle\rangle_m \mid (\!|\mathbb{E}[\texttt{return}\ M]|\!)_t$$
$$(\!|\mathbb{E}[\texttt{newEmptyMVar}]|\!)_t \ \rightarrow\ \nu m.\langle\rangle_m \mid (\!|\mathbb{E}[\texttt{return}\ m]|\!)_t, \quad m \notin \mathit{fn}\,(\mathbb{E})$$

Note that takeMVar will block if the the mutable variable in question is empty (since it cannot make a transition in that case). While blocked it may of course be interrupted by an asynchronous exception. Attempting to write to a full MVar results in an exception being raised:

$$\langle M\rangle_m \mid (\!|\mathbb{E}[\texttt{putMVar}\ m\,N]|\!)_t \ \rightarrow\ \langle M\rangle_m \mid (\!|\mathbb{E}[\texttt{throw PutFullMVar}]|\!)_t$$

It would be possible to have a blocking putMVar also, simply by removing this last transition.

10.6 Asynchronous Exceptions

In order to explain the behaviour of asynchronous exceptions in Concurrent Haskell, we need one last evaluation context type: $\mathbb{U}$-contexts.

There are three kinds of $\mathbb{U}$-contexts. If the hole is enclosed by unblock, then asynchronous exceptions are not blocked at the hole. Secondly, any $\mathbb{M}$-context is considered unblocked (since this is the default state of the system). While no transition uses $\mathbb{M}$-contexts explicitly, they are crucial to the definition of the first kind of $\mathbb{U}$-contexts: we need to know that there is no block in between the innermost unblock and the hole. Lastly, if the object of evaluation is an *interruptible* operation, like takeMVar and sleep, then even if it occurs in the context of a block, it may still be interrupted by an asynchronous exception.

Any thread that is executing in a context in which asynchronous exceptions are not blocked may be interrupted at any time:

$$(\!|\mathbb{U}[M]|\!)_t \xrightarrow{\not{\ } e} (\!|\mathbb{U}[\texttt{throw}\ e]|\!)_t, \qquad \mathbb{U}\ \text{maximal}$$

Note that this can only happen inside a $\mathbb{U}$-context; if the hole is protected by block then this rule cannot fire[6], preventing the asynchronous exception from interrupting the computation.

A crucial requirement is that $\mathbb{U}$ be maximal. Without this side condition, transitions like the following are possible:

$$\texttt{unblock (catch (return ())}\,H) \xrightarrow{\not{\ } e} \texttt{unblock (throw}\ e)$$

Disaster: the catch is ignored! This is because the context used to apply the rule, unblock $[\cdot]$, is not maximal. The maximal context in this case is unblock (catch $[\cdot]\,H$), which leads to the correct behaviour. Maximality of $\mathbb{U}$ boils down to the stipulation that the M in question cannot have a top-level >>= or catch.

Any thread may be interrupted by any other thread that knows its ThreadId, provided the first thread is executing with asynchronous exceptions unblocked:

$$(\!|\mathbb{U}[M]|\!)_t \mid (\!|\mathbb{E}[\texttt{throwTo}\ e\,t]|\!)_u \ \rightarrow\ (\!|\mathbb{U}[\texttt{throw}\ e]|\!)_t \mid (\!|\mathbb{E}[\texttt{return ()}]|\!)_u, \quad \mathbb{U}\ \text{maximal}$$

If the thread in question is already dead, then the throwTo succeeds immediately:

$$\mathbb{O}_t \mid (\!|\mathbb{E}[\texttt{throwTo}\ e\,t]|\!)_u \ \rightarrow\ \mathbb{O}_t \mid (\!|\mathbb{E}[\texttt{return ()}]|\!)_u$$

Recall that the notation $\mathbb{O}_t$ indicates that thread named t has finished computation.

Can thread throw an exception to itself? Not in the semantics given thus far, since no thread can ever be parallel to itself. We need to add the following rule:

$$(\!|\mathbb{U}[\texttt{throwTo}\ e\,t]|\!)_t \ \rightarrow\ (\!|\mathbb{U}[\texttt{throw}\ e]|\!)_t$$

If may seem that even with this rule, if a thread tries to throw to itself when asynchronous exceptions are blocked, then it will deadlock. This is not the case, since throwTo is an interruptible operation.

[6] Barring the case when an interruptible operation is the subject of evaluation, of course.

10.7 At the End of It All

Final thread values or any uncaught exceptions are lost:

$$\begin{aligned} (\!|\texttt{return}\ M|\!)_t &\rightarrow \mathbb{O}_t \\ (\!|\texttt{throw}\ e|\!)_t &\rightarrow \mathbb{O}_t \end{aligned}$$

If the main thread finishes, for whatever reason, then all other threads must also die, and all MVars are garbage collected:

$$\mathbb{O}_{\texttt{main}} \mid P \ \rightarrow\ \mathbb{O}_{\texttt{main}}$$

Since complex termination behaviour, such as a thread killing any threads that it forked, or throwing an uncaught exception to the thread that started it, can be readily implemented with the primitives at hand, the semantics is free to specify this simple termination behaviour.

11 (Toward) A Theory of Asynchronous Exceptions

Although it came into being in the later stages, the operational semantics has been a very useful tool in the development of the asynchronous exceptions mechanism, both in pointing out design alternatives, and highlighting problem areas. However, the main benefit of the semantics will be that we can use it to prove properties of programs and program fragments. Though a full blown theory based upon this semantics is beyond the scope of this paper, in this section we will briefly describe the beginnings of a theory, and sketch how it may be used to prove the safety and robustness of the locking mechanism presented in Section 8.

As process calculi go, ours is fairly parsimonious: the only forms of process construction are parallel combination and restriction, and synchronisation and communication are implemented at a pretty low level by MVars. However, matters are complicated by the fact that a thread is subject to interruption at any time (either by an external interrupt, or some other thread).

Write $P\downarrow$ to mean that closed process P *must terminate*, defined by this inductive definition:

$$\frac{\forall Q.P \xrightarrow{\alpha} Q \implies Q\downarrow}{P\downarrow} \qquad \mathbb{O}_{\texttt{main}}\downarrow$$

In other words, a process must terminate if and only if all possible transitions lead to $\mathbb{O}_{\texttt{main}}$.

We will define an approximation relation over processes via process contexts, and over terms via program contexts:

$$\begin{aligned} \mathbb{A}, \mathbb{B} &::= [\cdot] \mid \langle\rangle_m \mid \langle M\rangle_m \mid (\!|M|\!)_t \mid \nu x.\mathbb{A} \mid \mathbb{A} \mid \mathbb{B}. \\ \mathbb{C} &::= \text{program term with occurrences of } [\cdot] \end{aligned}$$

We are now in a position to define observational approximation, over both processes and terms.

Definition 1 *We say that process P* observationally approximates *process Q, written $P \lesssim Q$, whenever, for all process contexts $\mathbb{A}$ such that both $\mathbb{A}[P]$ and $\mathbb{A}[Q]$ are closed,*

$$\mathbb{A}[P]\downarrow \implies \mathbb{A}[Q]\downarrow.$$

Furthermore, we say that term M observationally approximates *N, written $M \lesssim N$, whenever, for all program contexts $\mathbb{C}$ such that $\mathbb{C}[M]$ and $\mathbb{C}[N]$ have type* IO a,

$$(\!|\mathbb{C}[M]|\!)_t \lesssim (\!|\mathbb{C}[N]|\!)_t.$$

Observational equivalence, *denoted by $\sim$ for both processes and terms, is defined as mutual observational approximation.*

This notion of observational approximation is not the only preorder that one might be interested in, but it does allow us to talk about *guaranteed* behaviour. For example, we can show that:

$$\texttt{block}\ \textit{after} \lesssim (\texttt{throw}\ e)\ \texttt{`finally`}\ \textit{after}$$

This says that, even in the worst case (when M immediately throws an exception, or equivalently, is immediately interrupted), M `finally` N is guaranteed to terminate at least as often as block N. In other words, M `finally` N *is guaranteed* to perform N, which is exactly the desired behaviour of finally. There is a similar inequation for bracket.

12 Implementation

Implementing synchronous exceptions is done in the standard way:

- catch pushes a *catch frame* on the stack which contains a pointer to the handler, before beginning to execute its argument.
- when an exception is raised, the stack is truncated up to (and including) the nearest enclosing catch frame, and control is passed to the handler with the exception given as an argument.

There is one additional issue in Haskell: what to do with "computations in progress", or *thunks*. The program may later attempt to demand the value of a thunk that was under evaluation when the exception was triggered. Since the exception is synchronous, we know that re-evaluating this thunk would yield the same exception, so it is safe to overwrite the thunk with a closure which will immediately raise the same exception if demanded. More details are given in [15].

The implementation of asynchronous exceptions differs only in the treatment of thunks; since we can't be sure that re-evaluating the thunk would raise the same asynchronous exception, we must either revert the thunk to its initial state, or "freeze" it at the point where the exception was received. The difference between the two techniques is operational only, the effect is not observable by the programmer. We use the technique for freezing thunks given in [16].

12.1 Implementation of block and unblock

To extend our implementation of exceptions with the block and unblock operations, we had to do the following:

- Extend the per-thread data block to include the current state of asynchronous exceptions, which is either blocked or unblocked. In the blocked state, the per-thread data block also maintains a queue of threads waiting to raise exceptions in this thread (recall that throwTo is synchronous, in the sense that the raising thread must wait until the exception has been delivered before it can continue).

- Extend the catch frame to include the state (blocked or unblocked) of asynchronous exceptions at the time when the frame was placed on the stack. This is necessary in order to be able to restore the correct state after handling an exception.
- Add two new types of stack frame: the *block frame* and the *unblock frame*. When execution returns to an unblock frame, asynchronous exceptions are unblocked (waking up any threads on the blocking queue), and the frame is removed from the stack. Block frames are identical, except that exceptions are blocked when execution returns to the frame.

The implementation of `block` is fairly straightforward:

1. If exceptions are already blocked, go to step 4.
2. Set the asynchronous exception state in the current thread to "blocked".
3. If there is a block frame on the top of the stack, remove it. Otherwise, push an unblock frame on the stack.
4. Continue by executing the argument of `block`.

The implementation of `unblock` is obtained by reversing "block" and "unblock" in the above sequence, and adding an extra step: before continuing, wake up any threads on the current thread's exception blocking queue.

Step 3 appears confusing, but it is designed to avoid unnecessary stack growth. Consider the following example:

```
f = do { ...; block (do { ...; unblock f }) }
```

The first `block` will push an unblock frame on the stack, which will still be on the top of the stack when we reach `unblock`. If we simply pushed a block frame before calling `f`, the stack would look like:

```
f's caller
unblock frame
block frame
```

and the stack would continue to grow by two frames for each recursive call to `f`. The adjacent block/unblock frames are superfluous: on return, we will simply block asynchronous exceptions and then immediately unblock them again for each pair of block/unblock frames. So step 3 in the above implementation of `block` is designed to remove the extra frames so that functions like `f` can run in constant stack space.

12.2 Implementation of `throwTo`

The `throwTo` operation is quite straightforward:

- If the target thread is currently running (perhaps on another processor), arrange for it to stop at a safe point as soon as possible. A single processor implementation with preemptive multithreading usually won't need this step, as all non-running threads will be stopped at safe points.
- If the target thread is currently in the scope of a `block`, and not currently waiting in an interruptible operation, then add the current thread to the target thread's exception blocking queue and reschedule.
- Otherwise, raise the exception in the target thread and continue.

13 Related Work

To our knowledge, no other language provides supports for asynchronous exceptions in such a way that they can be used safely and without resorting to gratuitous use of exception handlers to recover from untimely exceptions.

Erlang [1] has asynchronous exceptions of a kind: processes can be *linked* together, such that each process will receive an asynchronous exception if the other dies for some reason. The exception can be caught in the normal way. Erlang also has a way to control delivery of these asynchronous exceptions, providing the opportunity to have them delivered using asynchronous message passing instead of as exceptions. However, the control mechanism is stateful rather than scoped as in our approach, and hence doesn't allow safe exception handlers to be defined (asynchronous exceptions will always be enabled on entry to the exception handler, so there is a race window before they can be disabled again).

Standard ML originally had a weak form of asynchronous interrupt, whereby an external Control-C would asynchronously raise the `Interrupt` exception. Perhaps because of its semantic difficulties, this feature was removed in the Revised Definition [9]. However, Reppy describes an extension to SML that maps asynchronous Unix signals onto New Jersey SML [17]. The story is quite different to ours. A signal does not raise an exception; instead, it captures the running thread as a first-class continuation, and passes that to an exception handler. This has the advantage that execution can be resumed following a signal (which is not possible in our scheme), but it makes it difficult to interrupt the normal execution of the thread, except by throwing it away altogether. In particular, it is hard to restore invariants using handlers as we do.

Concurrent ML [18], Caml [6], and OCaml [7], all provide concurrency and exceptions, but none of them provide asynchronous exception passing between threads.

Java supports an `java.lang.Thread.interrupt()` method, which unblocks a waiting or sleeping thread with an `InterruptedException` [2]. When the thread is not waiting or sleeping, however, the `interrupt()` method merely sets the threads interrupt flag, which can be polled with `interrupted()`. This design is very similar to that in Modula-3, except that interrupts are called "alerts" in the latter, and only one call, `AlertWait`, can be unblocked by `Alert(t)` [4]. The big difference from our proposal is that we require no polling, and therefore in addition provide a way to mask interrupts.

Java originally offered a more asynchronous feature, namely `java.lang.Thread.stop()`, but deprecated the feature in version 1.2 [19] because it was too dangerous to program with. The reason given is that a method may receive an asynchronous exception while making changes to the object's mutable state, leaving the object in an internally inconsistent state. To recover from this situation would require a large number of exception handlers, and would be extremely difficult to get right.

C on Unix (without threads) has some support for asynchronous exceptions, in the form of signals combined with `setjmp/longjmp`. This use of signals as asynchronous exceptions is dangerous at best: blocking signals around critical sections can be expensive, and most library functions aren't written to be safe in the presence of asynchronous interrupts. For example, if the program is in the middle of a call to `malloc` when the signal strikes, the global heap may be left in an inconsistent state.

POSIX threads [5] provide an asynchronous method for killing threads, called thread cancellation, together with a state-based mechanism for disabling cancellation during critical sections. The synchronicity of cancellation is under programmer control, and can be either fully asynchronous, or semi-asynchronous (in which cancellation can happen only at programmer-defined "cancellation points"). A limited exception-handling mechanism is available so that the canceled thread can clean up after itself and release any locks it holds, but there is no way to recover from a cancellation.

14 Conclusion

We have shown how asynchronous exceptions can be incorporated into Concurrent Haskell in such a way that they can be used safely and robustly. The changes required to existing code to make it safe to use in the presence of asynchronous exceptions are kept to a minimum. In the case of pure non-I/O code, no changes at all are required to be able to use it in an asynchronous exception-enabled system. This means that in Haskell, not only are a large proportion of libraries automatically thread-safe, they are also automatically exception-safe too!

Our asynchronous exception model has several advantages over existing methods: for example, the compositional nature of our `timeout` function relies on true asynchronous exceptions. Synchronous exceptions just won't do since we don't want to have to modify the code that we're timing (which might even be unavailable) to include checkpoints.

The scoped nature of our `block` and `unblock` combinators leads to a clean and elegant operational semantics for Concurrent Haskell with exceptions. We hope to be able to formulate proofs, using this semantics and the theory outlined in Section 11, that simple combinators built using these primitives have the properties that we expect.

Experience with using our asynchronous exception model is still limited, although we have used it to construct a prototype fault-tolerant HTTP server which makes heavy use of time-outs, multithreading and exceptions.

15 Acknowledgements

Many thanks to Tony Hoare for his valuable comments on an earlier draft of this paper.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang.* Prentice Hall Europe, second edition, 1996.

[2] K. Arnold and J. Gosling. *The Java Programming Language.* The Java Series. Addison-Wesley, second edition, 1998.

[3] P. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a parallel, non-strict functional language with state. In R. J. M. Hughes, editor, *Proc. FPCA'91*, volume 523 of *LNCS*, pages 538–568. Springer-Verlag, 1991.

[4] A. D. Birrell. An introduction to programming with threads. In G. Nelson, editor, *Systems Programming with Modula-3*, chapter 4, pages 88–118. Prentice Hall, Englewood Cliffs, NJ, 1991.

[5] D. R. Butenhof. *Programming with POSIX Threads.* Professional Computing. Addison-Wesley, 1997.

[6] X. Leroy. The Caml Light system documentation and user's guide (release 0.74). Technical report, INRIA, Dec. 1997. At http://caml.inria.fr/man-caml/.

[7] X. Leroy, D. Rémy, J. Vouillon, and D. Doligez. The Objective Caml system documentation and user's manual (release 2.04). Technical report, INRIA, 1999. At http://caml.inria.fr/ocaml/htmlman/.

[8] R. Milner. The polyadic π-calculus: A tutorial. In F. L. Hamer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification.* Springer-Verlag, 1993.

[9] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, Cambridge, Massachusetts, 1997.

[10] A. K. Moran, S. B. Lassen, and S. L. Peyton Jones. Imprecise exceptions, co-inductively. In *Proc. HOOTS'99*, volume 26 of *Electronic Notes in Theoretical Computer Science*, Paris, Sept. 1999.

[11] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *Proc. POPL'99*, pages 276–290, San Antonio, Texas, 1999. ACM Press.

[12] S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. POPL'96*, pages 295–308, St. Petersburg, Florida, Jan. 1996. ACM Press.

[13] S. L. Peyton Jones, A. Reid, T. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. In *Proc. PLDI'99*, volume 34(5) of *ACM SIGPLAN Notices*, pages 25–36. ACM Press, May 1999.

[14] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. POPL'93*, pages 71–84, Charleston, South Carolina, Jan. 1993. ACM Press.

[15] A. Reid. Handling exceptions in Haskell. Research Report YALEU/DCS/RR–1175, Yale University, Aug. 1998.

[16] A. Reid. Putting the spine back in the Spineless Tagless G-Machine: An implementation of resumable blackholes. In *Proc. IFL'98 (selected papers)*, volume 1595 of *LNCS*, pages 186–199. Springer-Verlag, 1999.

[17] J. H. Reppy. Asynchronous signals in standard ML. Technical Report TR90-1144, Cornell University, Computer Science Department, Aug. 1990.

[18] J. H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, Oct. 1999.

[19] Why are Thread.stop, Thread.suspend, Thread.resume, and Runtime.runFinalizersOnExit deprecated? In the Java 2 SDK Standard Edition Documentation. At http://java.sun.com/products/jdk/1.3/docs/guide/misc/threadPrimitiveDeprecation.html.