Asynchronous Exceptions in Haskell

Simon Marlow and Simon Peyton Jones
Microsoft Research, Cambridge

Andrew Moran
Oregon Graduate Institute

John Reppy
Bell Labs, Lucent Technologies

December 12, 2006

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. Uniquely, we also give a formal semantics for our system.

1 Introduction

An important goal of language design is to support modularity. For concurrent languages, this goal means language support for localizing synchronization issues and support for composing components without interference. One concurrent language feature that appears to be the antithesis of modularity is the asynchronous signaling (or killing) of one thread by another. Since, by definition, such signaling can occur at any point in the target thread's execution, locks held by the target may not be properly released and invariants may not be maintained. For these reasons, few concurrent languages or thread libraries support truly asynchronous signalling, and those that do have discouraged its use.

There are situations, however, where allowing a thread to asynchronously signal another thread is extremely useful. For example, we might wish to provide a *timeout* operator that limits the execution time of a computation or we might wish to run two different computations in parallel taking the first result and terminating the other. In this paper, we present an extension to Concurrent Haskell [11] that supports true asynchronous signalling in a robust, modular way. The principal contributions of the paper are as follows:

- We explain why a fully-asynchronous signalling is both *useful* (as opposed to semi-asynchronous signalling) and *feasible* (Section 2). In fact, while imperative languages often use polling to implement a semi-asynchronous signalling mechanism, we explain that signalling must *necessarily* be truly asynchronous in purely-functional languages like Concurrent Haskell.
- We propose an extension to Concurrent Haskell that supports asynchronous delivery of exceptions between threads (Section 5). This mechanism allows one thread to terminate another.
- 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 5.1). It is also necessary to allow indefinitely blocking operations to be interrupted, we show how these mechanisms enable us to acquire and release locks safely in the presence of asynchronous exceptions (Section 5.3).
- We give an operational semantics for Concurrent Haskell (Section 6) and extend it with asynchronous exceptions (Section 6.3). A precise specification of exactly what asynchronous exceptions *do* is crucial for both programmers and implementors: asynchronous exceptions are subtle beasts. *We believe that this is the first formalization of an asynchronous exception or interrupt mechanism.*

In addition, we give the definitions of some useful combinators built on top of the low-level exception primitives, including a composable timeout combinator (Sections 7.4 and 7); and we outline an implementation of asynchronous exceptions and the associated primitive operations (Section 8).

2 Asynchronous exceptions

Many high-level languages provide *exceptions* as a way to support robust handling of error conditions. Errors are signaled by *throwing* an exception and are handled by *catching* the exception. 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. Examples include: divide by zero, pattern-match failure, and explicitly raising a user exception. 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.

In Proc ACM Conference on Programming Languages Design and Implementation, 2001.

- It follows that a compiler can reasonably infer (an approximation to) the set of synchronous exceptions that any given expression could possibly raise [18].

Since exceptions already provide a control-flow mechanism for signalling and handling exceptional conditions, it is natural to consider extending the exception handling mechanism to include *asynchronous* exceptions.[1] Such asynchronous exceptions are raised as the result of an "external event," such as a signal from another thread, and can occur at any point during execution. Since 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 a programmer's standpoint, than synchronous exceptions.

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

Speculative computation. A parent thread might start a child thread to compute some value speculatively; later the parent thread might decide that it does not need the value so it may want to kill the child thread.

Timeouts. If some computation does not complete within a specified time budget, it should be aborted.

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.

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. Since such exceptions can occur at almost any program point, it is natural to treat them as asynchronous.

A naïve approach to these problems is to provide a mechanism for one thread to *kill* another thread. While such a mechanism gets the job done, it creates serious problems. If a thread is killed while it holds a lock, how does the lock get released? If the thread is in the process of mutating a shared data structure, how do we reestablish the data structure's invariants? For these reasons, a simple kill mechanism is unacceptable.

In practice, few concurrent languages provide any mechanism for one thread to asynchronously signal another thread. More common are semi-asynchronous mechanisms based on *polling*, where the target occasionally checks for signals; or *safe points*, where the target accepts signals at certain designated points. For example,to terminate a thread we might set a global flag, and rely on the thread to periodically check the flag, as is done in POSIX threads, Modula-3, and Java (Section 10 elaborates).

While the semi-asynchronous approach avoids breaking synchronization abstractions, it is non-modular in that the target code must be written to use the signalling mechanism. Worse still (for us), *the semi-asynchronous approach is simply incompatible with a purely-functional language, such as Concurrent Haskell.* The problem is that polling a global flag is not a functional operation, yet in a Concurrent Haskell program, most of the time is spent in purely-functional code. On the other hand, since *there is absolutely no problem with abandoning a purely-functional computation at any point*, asynchronous exceptions are safe in a functional setting. In short, in a functional setting, fully-asynchronous exceptions are both necessary and safe — whereas in an imperative context fully-asynchronous exceptions are not the only solution and are unsafe. For these reasons, the semi-asynchronous approach is almost universal in imperative languages.

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. We also do not deal with killing *rogue* threads, since such threads can exploit our mechanisms to ignore asynchronous exceptions.

[1] There are reasons why one might keep these notions distinct, but they are orthogonal to the main points of the paper. See Section 9 for more discussion.

3 Input/output in Haskell

Haskell is a purely functional language with lazy semantics. Input/output in Haskell is done using a monad; in Haskell a value of type `IO a` is an "action" that, when performed, may do some input/output before delivering a value of type `a`. For example, here are two basic I/O functions[2]:

```
getChar :: IO Char
putChar :: Char -> IO ()
```

`getChar` is an I/O action that, when performed, reads a character from the standard input, and returns it to the program as the result of the action. `putChar` is a function that takes a character and returns an action that, when performed, prints the character on the standard output, and returns the trivial value `()`.

I/O actions can be combined using the `>>=` operator:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

So, for example, we can make a compound I/O action that reads a character from standard input and writes it to standard output (`\x -> e` is Haskell's notation for $\lambda x.e$):

```
getChar >>= \c -> putChar c
```

The action as a whole has type `IO ()`. Haskell also provides syntactic sugar, the `do`-notation, for expressing monadic combinations. The above expression could also be written

```
do { c <- getChar; putChar c }
```

Most of the code examples in this paper will use this `do`-notation.

For a thorough introduction to I/O in Haskell, see [**?**].

4 Concurrent Haskell

Concurrent Haskell [11] extends standard Haskell with a small set of primitives for creating new threads and performing simple inter-thread communication:

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

data MVar a           -- abstract
newEmptyMVar :: IO (MVar a)
```

[2] The notation "`f :: t`" means "`f` has type `t`"

```
putMVar      :: MVar a -> a -> IO ()
takeMVar     :: MVar a -> IO a

throw :: Exception -> IO ()
catch :: IO a -> (Exception -> IO 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 implementations of Concurrent Haskell exist. The `forkIO` function returns the `ThreadId` of the forked thread. A thread can also obtain its own `ThreadId` by calling `myThreadId`. `ThreadId`s support equality. Threads can sleep for a specified period of time (in microseconds) by calling `sleep`.

`MVar`s are a generic synchronization 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, or waits if the `MVar` is already full[3]. Using only `MVar`s, many complex datatypes for concurrent communication can be built, including typed channels, semaphores and so on [11].

A synchronous exception can be raised by `throw`, and caught by `catch`. The computation `catch` M H runs M. If M succeeds, then its result is the result of the `catch`. If instead it raises an exception, the handler H is run, passing the exception raised by M. The types of `throw` and `catch` are identical to the `ioError` and `catch` operations in the Haskell 98 standard [**?**], except that we have enlarged the `IOError` type to `Exception`, to take account of non-IO exceptions.

5 Asynchronous Exceptions in Haskell

We are now ready to work on asynchronous exceptions. We start by adding a new primitive to Concurrent Haskell to enable one thread to asynchronously raise an exception in another, `throwTo`:

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

Informally, the meaning of `throwTo t e` is that the exception `e` is raised in thread `t` as soon as possible, and the call returns immediately. In practice, the exception may not be delivered to the target thread until some time later, perhaps because it is running on another processor, or even another machine. If the thread `t` has already died or completed, then `throwTo` trivially succeeds.

Note that implementing `throwTo` is not 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 8.

Note that `throwTo` is the only source of asynchronous exceptions in the system, but asynchronous interrupts from the environment may also be converted into asynchronous exceptions by the programmer.

[3]This semantics for `putMVar` is slightly different from that given in [11], where `putMVar` on a full `MVar` was an error.

5.1 Safe Locking

We now consider the issues raised by writing code in the presence of asynchronous exceptions. An example which illustrates a number of these issues is the use of locking to provide concurrency control for shared mutable state. It is important that asynchronous exceptions be handled without leaving the shared mutable state in an internally inconsistent state.

In Concurrent Haskell, shared mutable state is normally represented by an `MVar`, which holds the value of the current state. Any thread wishing to access the state must first take the value from the `MVar`, leaving it temporarily empty, and put the new state back in the `MVar` afterwards. Hence only a single thread has access to the state at any one time.

The problem with this approach is that if an exception is raised while the thread holds the `MVar`, it will be left in an empty state, and deadlock may ensue.

To make this process safe in the presence of synchronous exceptions is straightforward: we simply arrange that should an exception be raised while we are building the new value of the state, the old value is replaced in the `MVar` and the exception propagated to the caller. If `m` is the `MVar` in question, and `compute` is an `IO` operation that takes the old state and returns the new state, then the code would look like this:

```
do { a <- takeMVar m;
     b <- catch (compute a)
           (\e -> do { putMVar m a; throw e });
     putMVar m b }
```

This is fine for synchronous exceptions, but in the presence of asynchronous exceptions there is a race condition: an exception can occur just after the `takeMVar` but before `catch`, when there is no exception handler in place to restore the state.

We could try to fix the hole by moving the `catch` around the `takeMVar`, but this opens another race window: the exception could occur *before* the `takeMVar`, causing the `putMVar` which replaces the old state to block forever on the still-full `MVar` (this is in addition to the fact that expanding the scope of the `catch` around the `takeMVar` makes it difficult to propagate the value of the old state to the exception handler).

5.2 Blocking exceptions

Clearly, some way to postpone the delivery of asynchronous exceptions during critical sections is needed. The standard method is to disable interrupts, using two operations placed around the critical section:

```
block   :: IO ()
unblock :: IO ()
```

The idea is that executing `block` puts the thread into a state in which asynchronous exceptions are *blocked*, and `unblock` does the reverse. These combinators are still somewhat clumsy for our purposes though, as we can see if we try to use them to fix up the locking example:

```
do { block;
     a <- takeMVar m;
     b <- catch (do { unblock; compute a; block })
                (\e -> do { putMVar m a; throw e });
     putMVar m b;
     unblock;
   }
```

Notice that we try to unblock asynchronous exceptions for the duration of the call to `compute` only. However, if an exception is raised during `compute`, whether synchronous or asynchronous, then control is passed to the exception handler *with asynchronous exceptions unblocked*, which opens up another race window as there is the possibility that we can receive another exception before replacing the contents of the `MVar`. Also somewhat unsatisfactory is the fragility of the programming model: it is all too easy to forget to re-enable exceptions after a critical section, especially if the control flow is complicated.

A much better approach is to use scoped combinators:

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

block (do {
   a <- takeMVar m;
   b <- catch (unblock (compute a))
          (\e -> do { putMVar m a; throw e });
   putMVar m b
})
```

where the meaning of `(block a)` is "execute `a` in a state where asynchronous exception delivery is blocked," while `unblock` does the reverse. The `block` and `unblock` combinators may be arbitrarily nested. There is no counting of scopes, *i.e.*, two nested `block`s behave the same as a single `block`. This is an important property from a modularity perspective: it means that `unblock` always unblocks asynchronous exceptions, regardless of the context.

Notice how the scoping of `block` closes the race condition in the exception handler: if an exception is received during `compute a`, then we exit the scope of `unblock` and enter the exception handler which is inside the `block`ed scope.

In practical terms, the implementation of exception handling must respect scopes. That is, it must save the current state of the thread (blocked or unblocked) when entering the scope of a `block` or `unblock` combinator, and restore it again when leaving that scope, whether normally or by a synchronous or asynchronous exception being raised.

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 is a new problem, namely that if the thread has to wait for the lock, it now waits in a `block`ed state. This is in violation of one of the cardinal rules of concurrent programming: "do not block while holding a lock," the reason being that it increases the potential for deadlock. More practically, it means we cannot time-out the thread waiting for the lock until it acquires the lock.

5.3 Interruptible Operations

How do we take the `MVar`, and atomically install an exception handler as soon as we have the `MVar`? Our solution is a subtle change to the semantics of blocking operations:

> Any operation which may need to wait indefinitely for a resource (*e.g.*, `takeMVar`) may receive asynchronous exceptions even within an enclosing `block`, but only while the resource is unavailable. Such operations are termed *interruptible operations.*

Wait a minute! Have we not just shot ourselves in the foot? Previously `block` was sure protection from asynchronous interrupts, now it is not. Nevertheless, there are several good reasons for adopting this approach:

- Having made this modification to the semantics, `takeMVar` behaves atomically when enclosed in a `block`. The `takeMVar` may receive asynchronous exceptions right up until the point when it acquires the `MVar`, but not after. Without this change, `takeMVar` is unsafe to use inside `block` at all.
- Although it seems strange that operations inside a `block` may raise asynchronous exceptions, the exceptions are synchronous in nature since we specify exactly which operations are interruptible.

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

Also note the careful wording of the definition above: by implication it states that an interruptible operation cannot be interrupted if the resource it is attempting to acquire *is always available*. Looking back at our locking example from the previous section, even though we used `putMVar`, an interruptible operation, in the exception handler, in this case the `putMVar` is non-interruptible because we can be sure the `MVar` is always empty.

6 Operational Semantics

In this section we give an operational semantics for Concurrent Haskell with exceptions,[4] and then proceed to add in our new features for asynchronous exceptions. We have so far introduced the new constructs using informal definitions; the semantics in this section precisely specifies the intended meanings. Our semantics is stratified in two levels: an *inner denotational semantics* describes the behaviour of pure terms, while an *outer monadic transition semantics* describes the behaviour of `IO` computations.

M and N range over *terms* in our language, and V ranges over values (Figure 1). A *value* is a term that is considered by the inner, purely-functional semantics to be evaluated. The values in Figure 1 include constants and lambda abstractions, as usual, but are unusual in two ways:

- *We treat the primitive monadic* `IO` *operations as values.* For example, `putChar 'c'` is a value. No further work can be done on this term in the purely-functional world; it is time to hand it over to the outer, behavioural semantics. In the same way, M `>>=` N, `sleep` 3, and `return` M are all values.
- Many of these monadic `IO` values have arguments that are not arbitrary terms (M, N, *etc.*), but are themselves values (m, c, d, *etc.*). For example, `putChar (chr 65)` is not a value, but `putChar 'A'` is. It is as if `putChar` is a *strict* data constructor. The reason for this choice is that evaluating `putChar`'s argument is indeed something that can be done in the purely-functional world; indeed, it must be done before the output operation can take place.

[4] There exists a published operational semantics for Concurrent Haskell [11], a denotational semantics for exceptions in Haskell [12], and an operational semantics for exceptions in Haskell [10], but so far no published semantics links these concepts or describes exceptions in the `IO` monad.

$$
\begin{array}{rrcl}
 & x, y & \in & \textit{Variable} \\
 & k & \in & \textit{Constant} \\
 & c & \in & \textit{Constructor} \\
 & ch & \in & \textit{Char} \\
 & d & \in & \textit{Integer} \\
 & e & \in & \texttt{Exception} \\
 & m & \in & \texttt{MVar} \\
 & t, u & \in & \texttt{ThreadId} \\
\textit{Values} & V & ::= & x \mid \texttt{\textbackslash}x \texttt{ -> } M \mid k \mid c\, M_1 \cdots M_n \mid \\
 & & & ch \mid d \mid e \mid m \mid t \mid \\
 & & & \texttt{return } M \mid M \texttt{ >>= } N \mid \\
 & & & \texttt{putChar } ch \mid \texttt{getChar} \mid \\
 & & & \texttt{putMVar } m\, N \mid \texttt{takeMVar } m \mid \\
 & & & \texttt{newEmptyMVar} \mid \texttt{sleep } d \mid \\
 & & & \texttt{throw } e \mid \texttt{catch } M\, H \mid \\
\textit{Terms} & M, N, H & ::= & V \mid M\, N \mid \texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2 \mid \cdots
\end{array}
$$

Figure 1: The syntax of values and terms.

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

Figure 2: The syntax of program states.

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

6.1 Program Transitions

We give the semantics by describing how one *program state* evolves into a new program state by making a *transition*. A program state consists of a collection of threads and `MVar`s in parallel, see Figure 2.

The transition from one program state to the next may or may not be *labelled* by an *event*, α. We write a transition like this:

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

The events α represent communication with the external environment; that is, input and output. We will use just three events:

- $P \xrightarrow{!\,ch} Q$ means "program state P can move to Q, by writing the character ch to the standard output".
- $P \xrightarrow{?\,ch} Q$ means "program state P can move to Q, by reading the character ch from the standard input".
- $P \xrightarrow{\$t} Q$ means "program state P can move to Q, when an amount of time t has elapsed".

$$
\begin{array}{rcll}
P \mid Q & \equiv & Q \mid P & (\textit{Comm}) \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & (\textit{Assoc}) \\
\nu x.\nu y.P & \equiv & \nu y.\nu x.P & (\textit{Swap}) \\
(\nu x.P) \mid Q & \equiv & \nu x.(P \mid Q), \quad x \notin \mathit{fn}(Q) & (\textit{Extrude}) \\
\nu x.P & \equiv & \nu y.P[y/x], \quad y \notin \mathit{fn}(P) & (\textit{Alpha})
\end{array}
$$

$$\frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R}\ (\textit{Par}) \qquad \frac{P \xrightarrow{\alpha} Q}{\nu x.P \xrightarrow{\alpha} \nu x.Q}\ (\textit{Nu})$$

$$\frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha} Q}\ (\textit{Equiv})$$

Figure 3: Structural congruence and structural transitions.

In the standard way [11], we define a structural equivalence over processes, formalizing 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 3 and contains alpha equivalence. Rules (*Par*) and (*Nu*) allow transitions within parallel compositions and inside restrictions respectively. The equivalence rules, (*Comm*), (*Assoc*) *etc.*, say that `|` is associative and commutative and that the scope of ν can be restricted or expanded as long as it does not interfere with any existing scopes, while (*Equiv*) says that we are free to use these equivalence rules to bring parts of the program state together.

6.2 Transition Rules

Transition rules for the standard `IO` and concurrency operations are given in Figure 4. Transitions take place within *evaluation contexts*, where an evaluation context is defined as

$$\mathbb{E} ::= [\cdot] \mid \mathbb{E} \texttt{ >>= } M \mid \texttt{catch } \mathbb{E}\, M$$

That is, to find the evaluation site, repeatedly look inside the first argument of `>>=` and `catch`. The rules in Figure 4 are standard in form, so we describe them only briefly:

Sequencing of `IO` operations is handled by `>>=`. When its left operand becomes a `return`, rule (*Bind*) passes the returned value on to `>>=`'s right operand.

Input/output. The canonical `IO` operations are `putChar` and `getChar`, described in (*PutChar*) and (*GetChar*). Other basic I/O operations, like `openFile`, have analogous semantics. Rule (*Sleep*) deliberately underspecifies `sleep`. Here, the $\$d$ label represents an external clock interrupt, indicating that d microseconds have passed since `sleep` d first became blocked. A correct implementation must guarantee that at least d microseconds have passed before a thread executing `sleep` d is woken; further delay is acceptable.

MVar operations are described by rules (*PutMVar*), (*TakeMVar*) and (*NewMVar*). (Recall from Figure 2 that $\langle\rangle_m$ represents an empty MVar, while $\langle M\rangle_m$ represents a full MVar containing M.) Note that if `takeMVar`

$$(\!|\mathbb{E}[\texttt{return}\ N\ \texttt{>>=}\ M]|\!)_t \ \rightarrow\ (\!|\mathbb{E}[M\ N]|\!)_t \qquad (\mathit{Bind})$$

$$(\!|\mathbb{E}[\texttt{putChar}\ ch]|\!)_t \xrightarrow{!ch} (\!|\mathbb{E}[\texttt{return}\ ()]|\!)_t \qquad (\mathit{PutChar})$$

$$(\!|\mathbb{E}[\texttt{getChar}]|\!)_t \xrightarrow{?ch} (\!|\mathbb{E}[\texttt{return}\ ch]|\!)_t \qquad (\mathit{GetChar})$$

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

$$\langle\rangle_m \mid (\!|\mathbb{E}[\texttt{putMVar}\ m\ M]|\!)_t \ \rightarrow\ \langle M\rangle_m \mid (\!|\mathbb{E}[\texttt{return}\ ()]|\!)_t \qquad (\mathit{PutMVar})$$

$$\langle M\rangle_m \mid (\!|\mathbb{E}[\texttt{takeMVar}\ m]|\!)_t \ \rightarrow\ \langle\rangle_m \mid (\!|\mathbb{E}[\texttt{return}\ M]|\!)_t \qquad (\mathit{TakeMVar})$$

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

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

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

$$(\!|\mathbb{E}[\texttt{throw}\ e\ \texttt{>>=}\ M]|\!)_t \ \rightarrow\ (\!|\mathbb{E}[\texttt{throw}\ e]|\!)_t \qquad (\mathit{Propagate})$$

$$(\!|\mathbb{E}[\texttt{catch}\ (\texttt{return}\ M)\ H]|\!)_t \ \rightarrow\ (\!|\mathbb{E}[\texttt{return}\ M]|\!)_t \qquad (\mathit{Catch})$$

$$(\!|\mathbb{E}[\texttt{catch}\ (\texttt{throw}\ e)\ H]|\!)_t \ \rightarrow\ (\!|\mathbb{E}[H\ e]|\!)_t \qquad (\mathit{Handle})$$

$$(\!|\texttt{return}\ M|\!)_t \ \rightarrow\ \mathbb{0}_t \qquad (\mathit{Return\ GC})$$

$$(\!|\texttt{throw}\ e|\!)_t \ \rightarrow\ \mathbb{0}_t \qquad (\mathit{Throw\ GC})$$

$$\mathbb{0}_{\texttt{main}} \mid P \ \rightarrow\ \mathbb{0}_{\texttt{main}} \qquad (\mathit{Proc\ GC})$$

$$\frac{M :: \texttt{IO a} \quad M \not\equiv V \quad M \Downarrow V}{(\!|\mathbb{E}[M]|\!)_t \ \rightarrow\ (\!|\mathbb{E}[V]|\!)_t}\ (\mathit{Eval}) \qquad \frac{M :: \texttt{IO a} \quad M \uparrow e}{(\!|\mathbb{E}[M]|\!)_t \ \rightarrow\ (\!|\mathbb{E}[\texttt{throw}\ e]|\!)_t}\ (\mathit{Raise})$$

Figure 4: Transition Rules for Concurrent Haskell (without asynchronous exceptions).

finds an empty `MVar`, no transition can take place; this is how a stuck thread is modeled in the semantics. Similarly for `putMVar`: when the `MVar` is full, the thread cannot make any further transitions.

Forking a new thread is described by rule (*Fork*), while (*ThreadId*) allows a thread access to its own `ThreadId`.

Synchronous exceptions raised by `throw` are propagated by `>>=` (*Propagate*), and caught by `catch` (*Catch*). Rule (*Handle*) explains how `catch` behaves when the computation it protects succeeds.

Termination. Rules (*Return GC*) and (*Throw GC*) state that final return values and uncaught exceptions are lost, while (*Proc GC*) says that once the main thread is finished, all other threads will eventually die.

These rules are enough if there is a value at the evaluation site. But sometimes there is not — for example, after a use of rule (*Bind*) the evaluation site is an application, which will not match any of the rules described so far. Of course, we must *evaluate* the application $M\ N$, using the "inner" semantics, and that is what rules (*Eval*) and (*Raise*) are about.

The inner operational semantics, which we do not present here, is described in [10]. It defines 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 allows one to raise (but not catch) an exception in purely-functional code, using the function

```
raise :: Exception -> a
```

A crucial characteristic of the inner semantics is that convergence and exceptional convergence are mutually exclusive: no term both evaluates to some value *and* raises an exception. Moreover, while convergence is deterministic, the exceptional convergence is not. In other words, a term may raise many different exceptions; which it does raise when evaluated is decided upon at run-time. This is the essence of *imprecise* exceptions [12].

Given this inner semantics, rule (*Eval*) "lifts" evaluation in the inner semantics to a transition in the outer system. (We stipulate that $M \not\equiv V$ to prevent infinite sequences of the form $V \rightarrow V \rightarrow V \rightarrow \cdots$.) Similarly, if the evaluation yields an exception, rule (*Raise*) replaces the failing evaluation by a `throw` of the exception.

6.3 Operational Semantics for Asynchronous Exceptions

We now extend the semantics to support the asynchronous exceptions we introduced in Section 5. Firstly, we need to

$$(\!|\mathbb{E}[\texttt{putChar}\ ch]|\!)^b_t \xrightarrow{!\,ch} (\!|\mathbb{E}[\texttt{return}\ ()]|\!)^\circ_t \qquad (\textit{PutChar})$$

$$(\!|\mathbb{E}[\texttt{getChar}]|\!)^b_t \xrightarrow{?\,ch} (\!|\mathbb{E}[\texttt{return}\ ch]|\!)^\circ_t \qquad (\textit{GetChar})$$

$$(\!|\mathbb{E}[\texttt{sleep}\ d]|\!)^b_t \xrightarrow{\$\,d} (\!|\mathbb{E}[\texttt{return}\ ()]|\!)^\circ_t \qquad (\textit{Sleep})$$

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

$$(\!|\mathbb{E}[\texttt{block}\ (\texttt{return}\ M)]|\!)_t \rightarrow (\!|\mathbb{E}[\texttt{return}\ M]|\!)_t \qquad (\textit{Block Return})$$

$$(\!|\mathbb{E}[\texttt{unblock}\ (\texttt{return}\ M)]|\!)_t \rightarrow (\!|\mathbb{E}[\texttt{return}\ M]|\!)_t \qquad (\textit{Unblock Return})$$

$$(\!|\mathbb{E}[\texttt{block}\ (\texttt{throw}\ e)]|\!)_t \rightarrow (\!|\mathbb{E}[\texttt{throw}\ e]|\!)_t \qquad (\textit{Block Throw})$$

$$(\!|\mathbb{E}[\texttt{unblock}\ (\texttt{throw}\ e)]|\!)_t \rightarrow (\!|\mathbb{E}[\texttt{throw}\ e]|\!)_t \qquad (\textit{Unblock Throw})$$

$$(\!|\mathbb{E}[\texttt{throwTo}\ t\,e]|\!)_u \rightarrow (\!|\mathbb{E}[\texttt{return}\ ()]|\!)_u \mid [\![t \not\downarrow e]\!] \qquad (\textit{ThrowTo})$$

$$(\!|\mathbb{E}[\texttt{unblock}\ \mathbb{F}[M]]|\!)_t \mid [\![t \not\downarrow e]\!] \rightarrow (\!|\mathbb{E}[\texttt{unblock}\ \mathbb{F}[\texttt{throw}\ e]]|\!)_t, \qquad M \not\equiv \texttt{block}\ N \qquad (\textit{Receive})$$

$$(\!|\mathbb{E}[M]|\!)^\bullet_t \mid [\![t \not\downarrow e]\!] \rightarrow (\!|\mathbb{E}[\texttt{throw}\ e]|\!)^\circ_t \qquad (\textit{Interrupt})$$

$$(\!|\mathbb{E}[\texttt{putChar}\ ch]|\!)^\circ_t \rightarrow (\!|\mathbb{E}[\texttt{putChar}\ ch]|\!)^\bullet_t \qquad (\textit{Stuck PutChar})$$

$$(\!|\mathbb{E}[\texttt{getChar}]|\!)^\circ_t \rightarrow (\!|\mathbb{E}[\texttt{getChar}]|\!)^\bullet_t \qquad (\textit{Stuck GetChar})$$

$$(\!|\mathbb{E}[\texttt{sleep}\ d]|\!)^\circ_t \rightarrow (\!|\mathbb{E}[\texttt{sleep}\ d]|\!)^\bullet_t \qquad (\textit{Stuck Sleep})$$

$$\langle M\rangle_m \mid (\!|\mathbb{E}[\texttt{putMVar}\ m\,N]|\!)^\circ_t \rightarrow \langle M\rangle_m \mid (\!|\mathbb{E}[\texttt{putMVar}\ m\,N]|\!)^\bullet_t \qquad (\textit{Stuck PutMVar})$$

$$\langle\rangle_m \mid (\!|\mathbb{E}[\texttt{takeMVar}\ m]|\!)^\circ_t \rightarrow \langle\rangle_m \mid (\!|\mathbb{E}[\texttt{takeMVar}\ m]|\!)^\bullet_t \qquad (\textit{Stuck TakeMVar})$$

Figure 5: Transition Rules for Asynchronous Exceptions.

add new values for `throwTo`, `block`, and `unblock`:

$$V \quad ::= \quad \ldots \mid \texttt{throwTo}\ t\,e \mid \texttt{block}\ M \mid \texttt{unblock}\ M.$$

Secondly, we need to add a new form of process that represents an "exception in flight":

$$P \quad ::= \quad \ldots \mid [\![t \not\downarrow e]\!].$$

Here, $[\![t \not\downarrow e]\!]$ represents an exception e which has been thrown to thread t, but not yet received.

Thirdly, we need to extend our notion of evaluation context to distinguish blocked and unblocked contexts:

$$\begin{array}{lcl} \mathbb{F} & ::= & [\cdot] \mid \mathbb{F}\ \texttt{>>=}\ M \mid \texttt{catch}\ \mathbb{F}\ H \\ \mathbb{E} & ::= & \mathbb{F} \mid \mathbb{F}[\texttt{block}\ \mathbb{E}] \mid \mathbb{F}[\texttt{unblock}\ \mathbb{E}] \end{array}$$

The split-level evaluation context allows us to specify whether the innermost context in a thread is `block` or `unblock`. Thus an unblocked context is of the form

$$\mathbb{E}[\texttt{unblock}\ \mathbb{F}].$$

We will follow the convention that when parsing a term with a view to matching evaluation context rules, contexts must be maximal.

We also need to distinguish between threads that are *runnable*, and those that are *stuck* (*e.g.*, trying to do a `putMVar` to a full `MVar`, or trying to take an empty `MVar`). We denote runnable threads by a superscript $\circ$, thus: $(\!|M|\!)^\circ_t$, and stuck threads by a superscript $\bullet$: $(\!|M|\!)^\bullet_t$. We will also write $(\!|M|\!)^b_t$ to mean that a thread is either runnable or stuck, but we do not know (or care) which.

Since most of the rules concern runnable threads, we normally elide the $\circ$ in the interests of reducing clutter. Most of the rules from Section 6.1 are still valid with this new definition of $\mathbb{E}$-contexts, and apply only to runnable threads. The rules that change are discussed below.

Our new transition rules are given in Figure 5. The first four rules are revised versions of rules from Figure 4. The next four rules are concerned with propagating return values and exceptions through `block` and `unblock`, and are unsurprising.

Rule (*ThrowTo*) describes how invoking `throwTo` causes the exception to be spawned as a separate entity, with the caller of `throwTo` continuing immediately.

Rule (*Receive*) says that any runnable thread may be interrupted by an exception targeted at its `ThreadId`, provided the thread is executing in an unblocked context. Any thread that is stuck may be interrupted, (*Interrupt*), except that the interruption is allowed in *any* context. As a side-effect, the interrupted thread becomes runnable.

To express the fact that they each wait for some impetus from the outside world, `putChar`, `getChar`, and `sleep` may all immediately become stuck. (We say *may* since we allow a signal from the environment will take precedence.) `putMVar` will become stuck when putting to a full `MVar`, and `takeMVar` will become stuck when trying to take from an empty `MVar`.

7 Building more powerful combinators

The features introduced in Section 5.1 are expressive but rather low-level. We do not advocate 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.

7.1 Bracketing abstractions

A useful combinator 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; })
```

Notice that the second argument to `finally` is executed inside a `block`. This is necessary in order to guarantee that the second argument is always executed, and using `block` in this case ensures that. The behaviour is similar to that of interrupts or Unix signals: in a signal handler, signals of the same type are normally disabled, so that the application has a chance to deal with the signal it has already received.

Reversing the arguments to `finally` yields `later`, which is sometimes useful:

```
later b a = finally a b
```

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

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

`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 -> workOnFile h)
        (\h -> hClose 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 have not. The implementation of `bracket` is a straightforward generalization of `finally`, above.

7.2 Symmetric process abstractions

The `forkIO` primitive is asymmetric: 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. `both` also evaluates its arguments concurrently, but waits for them both to terminate before returning the results in a pair.

These informal descriptions seem simple, but in the presence of asynchronous exceptions we have to be more precise about the behaviour. For instance, what happens when an asynchronous exception is sent to a thread executing `(either a b)`? Does it get propagated to the child threads? What happens if one of the child threads raises an exception?

Here is a more precise specification of the desired behaviour of `(either a b)`:

- `a` and `b` run concurrently
- Result is `(Left r)` if `a` finishes first and returns `r`, `(Right r)` if `b` finishes first and returns `r`, or `(throw e)` if either `a` or `b` raises an exception `e` before one of them returns a result.
- If the thread executing `either` receives an asynchronous exception, it is propagated to both children.
- The behaviour is undefined if either computation throws an exception to the main thread.

One possible implementation uses two child threads, and an `MVar` to hold the result:

```
data EitherRet a b = A a | B b | X Exception

either a b = do {
  m <- newEmptyMVar;
  block (do {
     a_id <- forkIO (catch (do { r <- unblock a;
                                 putMVar m (A r) })
                           (\e -> putMVar m (X e)));
     b_id <- forkIO (catch (do { r <- unblock b;
                                 putMVar m (B r) })
                           (\e -> putMVar m (X e)));
     let loop = catch (takeMVar m)
                      (\e -> do { throwTo a_id e;
                                  throwTo b_id e;
                                  loop });
     r <- loop;
     throwTo a_id KillThread; throwTo b_id KillThread;
     case r of {
         A r -> return (Left r);
         B r -> return (Right r);
         X e -> throw e
     } }) }
```

Note how we propagate all received exceptions to the children until one of them has returned a result or raised an exception.

It is important here that the `throwTo` calls in the main thread are non-interruptible: we have to be sure that all exceptions are properly propagated to the children, and also that both children are sent the `KillThread` exception before we return. If `throwTo` was interruptible, these properties would be hard to guarantee (see Section 9 for a discussion of an alternative design in which `throwTo` is interruptible).

7.3 Time-outs

Having `either` allows us to define a composable `timeout` combinator:

```
timeout :: Int -> IO a -> IO (Maybe a)
timeout t a = do r <- either (sleep t) a
                 case r of
                      Left _  -> Nothing
                      Right a -> Just a
```

`timeouts` may be arbitrarily nested, and the semantics of `either` ensure that they cannot interfere with each other.

7.4 Safe points

Sometimes we cannot use an immutable value to represent the data structure we are interested in; perhaps it has been passed to us across a foreign language interface, or the structure we are dealing with is large enough that creating a new one for each operation would be too expensive (standard Haskell does not have mutable structures, but many compilers support them as extensions). In Concurrent Haskell, `MVar`s are commonly used to hold an immutable value, but they can equally well be used in a more conventional way, as a semaphore to protect a directly mutable structure.

If an `MVar` is being used to protect a shared mutable data structure, such as a mutable array, then the chances are that we do not 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 locking example in the previous section. But what if `compute` is going to take a long time? Then we have to explicitly program checkpoints into the code such that `compute` 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 ())
```

8 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 `raise`d, 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 [13].

The implementation of asynchronous exceptions differs only in the treatment of thunks; since we cannot 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 [14].

8.1 Implementation of `block` and `unblock`

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

- Extend the per-thread data block to include the current state of asynchronous exceptions, which is either blocked or unblocked, and a queue of pending asynchronous exceptions waiting to be delivered to the thread.
- As soon as a thread exits the scope of a `block`, and at regular intervals during execution inside `unblock`, its pending exceptions queue must be checked. If there are pending exceptions, the first one is removed from the queue and delivered to the thread.
- 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 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.

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.

8.2 Implementation of `throwTo`

The `throwTo` operation is quite straightforward:

- Place the exception on the target thread's queue of pending exceptions. This may involve sending a "message" to the target thread in a distributed or multiprocessor implementation.

9 Design Alternatives

An alternative design, and one which we experimented with for some time, is to have `throwTo` be a *synchronous* operation in that it waits for the exception to be delivered before returning. In this design, `throwTo` also becomes an interruptible operation, because it can block indefinitely. The choice between these two designs is a hard one, there are arguments in favour of both approaches:

- The synchronous version of `throwTo` is sometimes easier to program with, because it provides a guarantee that the target thread has received the exception. On the other hand, the synchronous `throwTo` being an interruptible operation can cause headaches.
- The asynchronous version of `throwTo` can easily be implemented in terms of the synchronous one simply by forking a new thread to perform the `throwTo`. The reverse is somewhat harder, but can usually be achieved using an `MVar`.
- An asynchronous `throwTo` is likely to be easier to implement and more efficient in a multi-processor or distributed environment, because it doesn't require synchronization with the target thread. In a single-processor environment, both designs are equally straightforward to implement.
- The presentation of the semantics for the asynchronous version of `throwTo` is simpler than the synchronous version (the synchronous version needs a special case for a thread throwing an exception to itself, and extra cases to deal with the interruptibility of `throwTo`).

Our proposal uses a single datatype for both synchronous and asynchronous exceptions. We choose this design for this paper, because it simplifies the presentation and semantics, but there are arguments in favor of distinguishing between them in the type system. Since synchronous exceptions are dependent to the local execution of a thread, it is possible to use analysis to check for uncaught exceptions [18] and for a compiler to optimize the control-flow of statically matching `throw`/`catch` pairs. Adding asynchronous exceptions to the mix means that any expression can be the source of an exception, which renders these techniques useless. Another problem is that sequential code that was written without thought of asynchronous exceptions may break assumptions of our combinators. For example, if we put the expression

```
e `catch` \ _ -> e
```

in the context of the `timeout` combinator, it can intercept the `TimeOut` exception, which breaks the combinator. While one might argue that universal handlers like this one are bad programming practice, such code is quite reasonable in a sequential setting, where one understands exactly which exceptions the expression `e` might raise. A solution is to define two datatypes, `exceptions` and `alerts`, with a distinct `catch` operator for each type. Using Haskell's typeclasses, we can overload the `catch` operator to provide some syntactic unification. Java addresses a similar problem by distinguishing in the type system between *checkable* and *uncheckable* exceptions, where methods must declare the *checkable* exceptions they may raise.

10 Related Work

To our knowledge, no other language supports fully-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. Furthermore, we believe that our semantics is the first formal accounting of truly asynchronous signalling.

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. Because it was not possible to write robust handlers for the `Interrupt` exception [15], it was removed from the 1997 revision of the language [9]. The SML of New Jersey system uses a more general mechanism of asynchronous signal handlers as a replacement [15]. In this mechanism, an asynchronous exception causes the current thread of control to be reified as a first-class continuation, which is then passed to a signal handler. The signal handler runs with signals masked, so additional signals are deferred until the handler is done. The signal handler may either resume the interrupted thread or transfer control to a different thread. This mechanism is used to implement preemption in Concurrent ML (CML) [16], but CML does not support asynchronous signalling between threads. It should be possible to add asynchronous signalling (including the `block` and `unblock` combinators) to CML using first-class continuations and the signal handler mechanism, but we do not know how to implement the `block` combinator using these mechanisms in a way that preserves tail recursion (as described in Section 8.1). OCaml [7] also provides support for concurrency, but does not support asynchronous signaling.

Some concurrent languages provide support for semi-asynchronous exceptions. For example, Modula-3 defines a mechanism for one thread to *alert* another, which causes the `Alert` exception to be raised in the target. The raising of the exception is deferred until either the target calls either `TestAlert` or `WaitAlert`. The alert mechanism has been formalized as part of a Larch specification of Modula-3's thread synchronization [4]. Java supports a similar mechanism for unblocking a waiting or sleeping thread with an `InterruptedException` [2]. When the thread is not waiting or sleeping, however, the `interrupt()` method merely sets the thread's interrupt flag, which can be polled with `interrupted()`. The big difference between these mechanisms and our design is that ours is fully asynchronous.

Java originally offered a fully asynchronous exception method (the `stop` method of the `Thread` class), but deprecated the feature in Version 1.2 [17]. The reason given is the one discussed in Section 2, namely that since a method may receive an asynchronous exception while making changes to the object's mutable state, the feature was too dangerous to program with.

There are several parallel Lisp and Scheme dialects that support speculative computation using some form of parallel-or operator (like our `either` combinator). In the language QLisp, a child thread can throw an exception that is caught by its parent [5]; *i.e.*, the scope of a `CATCH` in QLisp includes any threads spawned below it. Furthermore, other computations below the `CATCH` are also terminated (*e.g.*, the siblings of the throwing thread). QLisp also provides the `UNWIND-PROTECT` form to support cleanup when an exception is thrown; the cleanup handler runs in an *unkillable* state, so that multiple throws are not a problem. The main difference between the asynchronous signalling mechanisms of QLisp and our mechanism is that QLisp is motivated by controlling speculative computation, and so asynchronous signalling is a heavy-weight mechanism that affects a whole tree of threads. It should be possible to build similar mechanisms using our more primitive construct. Another difference is that QLisp does not have a formal semantics. In some respects, the language PaiLisp may have the closest mechanism to ours [6]. In PaiLisp, a thread can invoke a *first-class continuation* in another thread, which has the effect of forcing control in the target thread to the `call/cc` that bound the continuation. PaiLisp uses this primitive mechanism to define higher-level combinators, such as parallel-or. From the published description, it does not appear that PaiLisp has any signal masking/unmasking mechanism like our `block`/`unblock` combinators.

While existing *languages* have not provided support for asynchronous signaling, many *operating systems* have such mechanisms. The best known example of these is the POSIX signal mechanism (which is the model for signals in SML/NJ). While POSIX signals are sufficient to implement asynchronous signaling, they are expensive (all operations involve user/kernel transitions) and most POSIX library code is not asynchronous-signal safe.

Extending POSIX signals to multithreaded programs written using the POSIX Threads API (PThreads) has proven problematic and the recommended practice is for multithreaded programs to designate a single thread to handle all asynchronous signals. The PThreads API does provide an asynchronous method for killing threads, called *thread cancellation*. A thread can define the type of cancellation it accepts (deferred or asynchronous) and can enable or disable cancellation. Deferred cancellation, what we have called semi-asynchronous, is the default behavior. In this mode, cancellation messages are deferred until the target thread executes a library function that is defined to be a *cancellation point* (similar to our notion of interruptible operations). A mechanism for maintaining a stack of cleanup routines is also provided, which allows threads to restore invariants. The use of asynchronous cancellation is discouraged, since it can only be safely used for code that does not hold resources or modify global state. While the basic function of PThread cancellation is similar to our design, our language-based approach offers many advantages to the programmer. Our `block` and `unblock` combinators are easier to use correctly than cancellation-state changing operations of PThreads. Furthermore, our combinators support robust cleanup of asynchronous exceptions, whereas the PThread cleanup routines are not robust in the asynchronous cancellation mode (because of the possibility of multiple cancellation requests). Our design also has the advantage of allowing the signalled thread to continue executing, whereas a canceled thread must terminate after cleanup.

11 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, and this is a property guaranteed by the *type* of the expression; no further analysis is required. 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 will not do since we do not want to have to modify the code that we are 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, that simple combinators built using these primitives have the properties that we expect. We believe that there two useful theories that arise from the semantics: a simple equational theory, and a more subtle theory based on a commitment ordering, where a process will approximate another if the latter is *committed* to performing at least the same operations as the former. The commitment theory is novel, and would allow us to prove, for example, that `finally a b` is committed to performing the same operations as `block b`. Work on these theories is at a very early stage.

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 [**?**].

12 Acknowledgements

Many thanks to Tony Hoare for his valuable comments on an earlier draft of this paper. We are also grateful to Claus Reinke for his very detailed comments on an earlier draft of this paper, which, apart from improving the paper generally, led to the correction of a bug in the semantics.

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, J. V. Guttag, J. J. Horning, and R. Levin. Thread synchronization: A formal specification. In

G. Nelson, editor, *Systems Programming with Modula-3*, chapter 5, pages 119–129. Prentice Hall, Englewood Cliffs, NJ, 1991.

[5] R. P. Gabriel and J. McCarthy. Queue-based multi-processing Lisp. In *Proc. LFP'84*, pages 25–44, Aug. 1984.

[6] T. Ito and M. Matsui. A parallel lisp language PaiLisp and its kernel specification. In *Parallel Lisp: Languages and Systems*, volume 441 of *LNCS*, pages 58–100, June 1989.

[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] 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.

[12] 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.

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

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

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

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

[17] 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.

[18] K. Yi. Compile-time detection of uncaught exceptions in Standard ML programs. In *sas94*, volume 864 of *LNCS*, pages 238–254, Sept. 1994.