# Asynchronous communication between threads

Marcin Kowalczyk `<qrczak@mimuw.edu.pl>`

Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warszawa, Poland

September 2, 2005

### Abstract

In programming there is sometimes a need to make computation interruptible by external events. This paper argues that a dedicated language mechanism solves this need better than a pure library convention. The main issue in designing such a mechanism is specifying points of computation where it's safe to interrupt a thread in order to execute a signal handler. The paper proposes rules for identifying such points, rules which are especially suited for impure functional languages. The proposed mechanism of asynchronous signals happens to be useful for specifying how threads should interact with Unix `fork()`.

## 1 INTRODUCTION

In programming there is sometimes a need to make computation cancelable by external events. Examples:

- The user initiated some lengthy action, but later pressed `^C` or clicked a GUI widget to cancel it.

- The program is using too much heap or stack. Current action should better be abandoned, or we risk slowing down the whole system or even killing our program before it has a chance to save any successfully completed work.

- A computation has been started in the background (e.g. preloading articles likely to be visited next in a newsreader, or counting lines in a large document), yet another action determined that its results will not be needed (e.g. the user changed the group or closed the document).

There are also other kinds of events which do not cancel a computation, but may happen at any time too and the program should promptly react to them:

- The configuration file of a server has been changed. Handlers of currently connected clients should begin using the new configuration.

- The user manipulates a UI widget, e.g. moves a slider to scroll the contents of a window, which causes it to be redrawn.

There are two basic styles of handling such events:

1. The computation periodically checks whether someone wishes it to be canceled or altered.

2. The language provides some mechanism of asynchronous signals. The program specifies handlers for signals which may occur at any point during a complex computation.

The first style doesn't need special language features, but becomes hard to manage as programs get large and consist of separately developed components. Without programmers agreeing on a single protocol for event notification, calling a lengthy computation delays processing of events and such call becomes uninterruptible.

An agreed convention about synchronous checking of a thread-specific flag would be a workable but partial solution. Such flag would have to be manually checked during all loops which might take a long time. Mistakes in identifying all necessary places would be relatively harmless but hard to detect, as the symptom is only poor responsivity.

This calls for a language mechanism which would allow to register handlers of particular signals, separately from code which can be interrupted by these signals. But even though an external event can occur at any time, a thread is not necessarily ready to handle it at an arbitrary point of its execution. Data structures might have their invariants temporarily violated. The thread might have taken a mutex; if the signal handler tries to take the same mutex, depending on the semantics of recursive locking of the mutex the outcome is an error, a deadlock, or data corruption. The natural thing do to is to delay processing of events until invariants of data structures are restored.

There should be a mechanism for identifying regions of code which can't be interrupted. This paper proposes such mechanism which aims at minimizing explicit manipulation of the signal blocking state. Some operations like mutex locking block signals automatically. Obtaining a useful behavior wrt. signals from scoped mutex unlocking and from waiting for a resource acquisition require to make the signal blocking state more elaborated than a two-state *blocked/unblocked* switch, as described below.

The author implemented this design in his language Kogut[1]. It has been exercised to program reactions to keyboard interrupts, memory exhaustion, terminal resizing, and timeouts in a game of Tetris and in an interactive Scheme interpreter.

## 2  SIGNALS AND EXCEPTIONS

Note that it's not enough to map asynchronous events to exceptions, even though some languages have done or used to do this [2, 4]. There are two reasons for this: they would not deal with events which should not abort a computation, and selective blocking is necessary for robust handlers of exceptions which may occur at any time, because some sequences of actions may not be executed only partially.

Asynchronous exceptions are subsumed by asynchronous signals. Delivery of a signal to a thread causes it to run a signal handler, which can just throw an exception thus emulating an asynchronous exception, but can also do something else and then resume the interrupted code.

Signals differ from exceptions in that they can be temporarily blocked or ignored. It makes no sense to block synchronous exceptions: an exception is thrown when the execution *cannot* continue normally.

So the two mechanisms complement each other. Exceptions provide a mechanism to abort a computation up to the nearest enclosing handler. Signals themselves don't unwind the stack, but a signal handler can throw an exception. This way exceptions deal only with escaping, and signals only with asynchronicity.

The term *signal* reminds Unix signals, but this is a separate mechanism to which Unix signals can be mapped.

## 3 BLOCKING SIGNALS TEMPORARILY

Thread cancellation can be implemented as a special case of a signal (whose handler just throws an exception). Learning from existing practice in blocking thread cancellation helps in designing signals.

In POSIX reaction to thread cancellation has three possible states: blocked, unblocked synchronously, and unblocked asynchronously.

When cancellation is unblocked synchronously, the thread can be canceled only during execution of selected functions. They are basically those functions which let a thread block indefinitely, except `pthread_mutex_lock()`. If a well-behaved thread performs a long computation without doing I/O nor synchronizing with other threads, it should periodically invoke `pthread_testcancel()` in order to be cancelable promptly.

Fully asynchronous cancellation is rarely used in C because it's very hard to gurantee integrity of data structures which are being modified, and that all resources are released. Even a `malloc()` call is not safe to be interrupted, and thus may not be used in the asynchronous cancellation mode. Asynchronous cancellation makes sense only during a long pure computation which doesn't allocate memory.

People developing Haskell made an interesting observation [3]: for a purely functional computation asynchronous cancellation is both safe and necessary. It is safe because the computation has no side effects other than turning some unevaluated thunks into computed values (they should be reverted if the computation is aborted), so it can't leave data in inconsistent state, and because the only resources it can obtain is memory, which is garbage collected. On the other hand it's essential that code may be interrupted asynchronously, because a purely functional code has no way to invoke an equivalent of `pthread_testcancel()`, which is an imperative action. This observation partially extends to code written in a mostly functional style. While it's not automatically safe to handle an asynchronous signal at an arbitrary point, the regions of computation which are safe to be interrupted are large enough to make asynchronous signals viable. Signals should only be blocked during selected code fragments. The language should assist in making signals safe by blocking them automatically in various language constructs, so that libraries don't have to be unnecessarily peppered with explicit blocking where it's easily seen that they should better be blocked.

Making signals unblocked in a sufficiently large portions of a program allows to map Unix signals to this concept too. The goal of blocking is to find the right balance between promptness and safety, such that it is safe to do interesting things in a signal handler, yet the program doesn't wait too long until the main computation thread reaches a safe point. When it's made safer, it can be sensibly used for more things than Unix signals.

### 3.1 The blocking state

*Scoped* combinators which change the blocking state during a computation and restore it afterwards are easier to use properly than stateful actions which just get or set the current state. This is also the approach of [3] and [4] which provide two primitives applied to a block of code: `block` and `unblock`.

In both of these designs the blocking state can be described as a simple Boolean flag: either asynchronous exceptions are blocked or unblocked. It could be done better, and here are two reasons.

The following Haskell fragment from [3] describes a common pattern of working with shared 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."

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

Asynchronous exceptions are unblocked during the main processing of the value, but they are blocked during the short critical fragments just after the `MVar` has been emptied or just before it's filled back, so this code fragment can't exit with the `MVar` left empty even if an asynchronous exception arrives at a bad moment.

The problem appears when such code is hidden inside some complex action and the whole action is executed with signals blocked, with the intent that they remain blocked all the time. This code unconditionally unblocks them for the main processing, and it's impossible to execute it avoiding the unblocking.

While this could be fixed by manually querying the blocking state and making the inner `unblock` conditional, the language should make restoring the previous state as easy as unconditional unblocking, without having to pass around the saved state of blocking.

This goal can be archieved by altering the semantics of these modifiers: count the number of nested `blocks`, subtract the count of `unblocks`, and allow signals only when the count reaches 0 again. But this scheme is still insufficient, as the following scenario explains.

When waiting for an event like a key to be pressed, where an item is "consumed" from some queue by one thread which asks for the item, it is useful to be able to allow signals until the very moment when the resource is obtained. This behavior must be supported by particular waiting primitives, otherwise it's unimplementable (without race conditions and busy waiting).

Haskell chooses to make these waiting operations interruptible even when they are called with signals nominally blocked, and Python proposed to use this idea too. But this makes impossible to use these operations while keeping signals blocked all the time. While well-behaved threads should not wait too long with signals blocked, this is sometimes necessary, especially if we are not aware that an operation we intend to use with signals blocked can sometimes briefly wait while synchronizing with some other thread.

As a principle, signals may be implicitly blocked by various syntactic constructs and functions, but they should be unblocked only explicitly. The Haskell behavior violates it.

The state in which signals should be allowed only while the thread is waiting for some resource is distinct from having signals completely unblocked and from having them completely blocked. Combined with the need of counting scopes, this leads to the following proposal.

The blocking state is encoded as a pair $(all, async)$, where $all$ is the number of reasons why all signals should be blocked (a natural number or $\infty$), and $async$ is 0 when fully asynchronous signals are unblocked and 1 when signals are blocked in all places other than points where the thread would be suspended (this is called *synchronous mode*). This infinite space of states yields only three different direct effects on the reaction to signals:

| | |
|---|---|
| $(0,0)$ | signals are handled immediately (the default) |
| $(0,1)$ | signals are handled when the thread would be suspended |
| anything else | signals are blocked |

All interesting transitions of the blocking state can be obtained from the following primitives (they are scoped, i.e. they restore the state after executing some a given piece code):

| | |
|---|---|
| `BlockSignals` | $all := \infty;\ async := 1$ |
| `UnblockSignals` | $all := 0;\ async := 0$ |
| `BlockSignalsMore` | $all := all + 1$ |
| `BlockSignalsLess` | $all := \max(all - 1, 0)$ |
| `BlockAsyncSignals` | $async := 1$ |
| `UnblockSyncSignals` | $all := 0$ |

In addition signals should be implicitly blocked by certain language constructs and functions, so that in many cases code becomes asynchronous signal safe without explicit manipulations of the signal blocking state. This is detailed in the following sections.

## 3.2 Obtaining and releasing a resource

Several languages provide a particular construct for guaranteed release of a resource, sometimes called `try...finally`. It delimits two pieces of code: a main body and a closing action. The closing action is executed after the main body finishes, even if it fails with an exception.

Few languages however distinguish an opening action to be executed before the body. Special treatment of the opening action becomes neccessary in the presence of asynchronous signals.

It's easy to see that the closing part should run with signals blocked. If it was interrupted and the signal handler threw an exception, the resource would not have been fully released and would be lost. While releasing a resource rarely needs to suspend the thread, if it does (e.g. to access shared data used by another thread), signals should not interrupt it, as if `BlockSignals` was used.

But there is also a time window between the resource is obtained and an exception handler for the body is installed, and during that time the computation should not be interrupted either. Even if obtaining the resource is moved inside the exception handler, there is a window between the resource is obtained and it is assigned to a variable which is checked in the closing handler. Permitting an asynchronous signal at that point would lead to the resource being lost.

For this reason [4] proposes to extend the `try...finally` syntax with an opening clause, `initially`, which is also run with signals blocked.

But what if we must wait for the resource for a long time? There is no problem with interrupting the thread **before** the resource is obtained. Here the notion of interruptible operations and synchronous mode comes handy: this time suspending the

thread should be interruptible, as if `BlockAsyncSignals` was used. The function which obtains a resource is responsible for not leaking the resource if it's called in synchronous mode.

The main body should run with the same state of signals as code outside the whole construct.

## 3.3 Locks

Locking a mutex or a read-write lock is usually scoped over a fragment of code. Waiting for the mutex to be free can be handled like waiting for a resource: in synchronous mode. Unlocking a mutex never suspends the thread, so it doesn't matter how it is done as long as a signal doesn't interrupt it in the middle. For the consistency of presentation we can pretend that unlocking is done with signals blocked.

There is a difference however between mutexes and other resources. There are two reasons why blocking signals during the computation protected by the mutex is a good idea:

- A mutex is often used around a few operations which should be performed atomically. Interrupting them in the middle might leave the protected object in an inconsistent state.

- If the signal handler would take the same mutex, it should not be run until the mutex is released by the main code.

A well behaved program will not hold a mutex for a long time, so having signals blocked while a mutex is held should not impact responsiveness too much.

Locking a read-write lock for writing brings the same issues as for a mutex. Only the second reason is valid for locking it for reading (the data can't be left in inconsistent state since it's not being modified), but the first reason is enough to treat it the same way because the signal handler might want to lock it for writing.

Note that `pthread_mutex_lock` is not a cancellation point in POSIX, even though it suspends the thread. This is because in C synchronous mode is the default and adding a mutex to protect some internal data should not increase the amount of cancellation points. Nevertheless the author decided to make locking a mutex interruptible, like obtaining other resources. In my case synchronous mode is used only in certain cases where we wait for a resource. Most of the time signals are either completely unblocked (and interrupting the wait for a mutex is harmless) or completely blocked (and it will not be interrupted anyway).

A natural extension of scoped locking is scoped unlocking: the mutex is unlocked, a piece of code is executed, and the mutex is locked again no matter how the code finished before we proceed further. This piece of code should run with the same blocking state as the code surrounding the corresponding locking, assuming there are no other intervening reasons for blocking signals. But there may be other reasons; perhaps another mutex has been locked in the meantime. The easiest solution is to count the number of scopes which block signals minus the number of scopes which unblock them again. This is what `BlockSignalsMore` and `BlockSignalsLess` do.

The beginning of the temporary unlocking is easy because it can be executed atomically. There is a problem however with locking back at the end: this is a rare case where restoring the previous state after a scoped operation may suspend the thread. The semantics of a scoped operation requires to leave the mutex locked no matter

what, so this time we will wait for the mutex with signals blocked. While in theory it would be possible to process signals until some signal throws an exception and block further signals at that point, The author thinks it's not worth the effort to introduce such strange blocking state. In a well designed program a mutex should not be locked for too long anyway.

Waiting for a condition variable (like `pthread_cond_wait` in POSIX) behaves like temporary unlocking, including the guaranteed locking at the end. During the wait the mutex is unlocked and signals are processed. Since during processing a signal we are not hooked to the condition variable and we might miss a notification from another thread that the condition has become true (or even self-notification from our own signal handler), a signal can cause a spurious wakeup. As all spurious wakeups, it relocks the mutex, reevaluates the predicate associated with the condition, and continues waiting if the predicate is still false.

This adds a reason for legitimacy of spurious wakeups from condition waits. Some people argue that condition wait sould not have spurious wakeups. But even without spurious wakeups a well written program should recheck the predicate, because another thread might come before we relock the mutex and render the predicate false again. The author treats the loop which checks for the predicate as a part of the semantics of a condition wait, and actually embed it in in my `Wait` function.

## 3.4 Signal handlers and exception handlers

While a signal is handled, further signals should be blocked by default. The programmer can explicitly unblock them if he wishes.

The case of an exception handler is more delicate. Should it run with the state of blocking signals taken from the point of `try`? Often this would make sense: the state of locks, resources etc. has been generally restored during propagation of the exception to the state belonging to this point of code. On the other hand if the exception was caused by a signal and it should be processed before further signals, they should be kept blocked while the exception is propagated.

Instead of trying to design a semantics of combining two blocking states into one, it is just declared that signals are blocked in exception handlers. Exceptions are exceptional anyway.

This may cause a problem if the exception handler includes a long computation, performed not for recovering from an error but for trying an alternative path. E.g. if we search for a file in several directories, the exception handler for trying the first directory might include traversing of the rest of the directory list.

There is another reason however for avoiding this design and searching for the rest of directories outside the exception handler. If the runtime prints a stack trace when a program dies with unhandled exception, it is useful to not consider an exception handled when its handler throws another exception (or rethrows the same), and in this case the stack trace of the previous exception is still included in the output. This means that the stack is not unwound until the exception handler finishes, and thus it's unwise to make the recursive call inside it.

## 3.5 Lazy variables

Lazy variables can be used even in languages with strict evaluation policy. When a lazy variable is created, there is specified an expression denoting its value. The

expression is evaluated when the variable is accessed for the first time. Later accesses return the previously computed value.

Lazy variables are useful to ensure that a given initialization is performed as late as possible, yet only once. They are also used for making possibly infinite lists whose elements are computed on demand, only up to the point of the last value ever requested.

What should happen when trying to evaluate the value of a lazy variable throws an exception? The obvious answer is that the exception is propagated to the code which tries to access the variable, and the next time the variable is evaluated the same exception is immediately thrown again. This is for example how lazy variables work in OCaml. Not storing the exception and later evaluating the value of the variable from the beginning might cause a surprising duplication of side effects.

Unfortunately asynchronously generated exceptions cause a problem because they can interrupt code which is not **responsible** for the exception. An asynchronous exception, generated while the value of a particular lazy variable was computed, should not spoil forever a lazy variable whose value could be computed another time without problems.

Glasgow Haskell provides both laziness and asynchronous exceptions. It avoids this problem altogether because it uses laziness only in purely functional parts of the code, which can be safely restarted from the beginning, and which can't catch and process exceptions. And this is what Glasgow Haskell does: it reverts any lazy variables being under evaluation, by scanning the stack. This is an advantage of distinguishing purity of certain sections of code.

Note that Haskell's purity is fake in the case of certain lazy I/O functions, e.g. `getContents`. And here an asynchronous exception does spoil the lazy list of characters being constructed while waiting for further input: trying to evaluate it again from a different thread rethrows the asynchronous exception instead of attempting to read more input from the point where it was interrupted.

An impure language must deal with the possibility of catching exceptions inside evaluation of a lazy variable, and possible transformation of that exception to a different exceptions while rethrowing. Since exceptions which were indirectly caused by asynchronous signals are indistinguishable from exceptions resulting from inability to meaningfully compute the value, such language doesn't know which lazy variables should remember the exception and immediately throw it again when evaluated, and which ones should be reverted and evaluated from the beginning.

There is a different approach than reverting lazy variables. Signals could be simply blocked while a lazy variable is being evaluated, so instead of restarting from the beginning the evaluation would proceed to the end and only then will the asynchronous signal be processed.

Unfortunately this would cause unpleasant side effects. Let's consider a lazy list of lines being read from the terminal. While a thread is waiting for the user to enter the line, it would be uninterruptible, because the line is being read inside initialization of a lazy variable which is a component of a lazy list. In this case restarting the computation from the beginning would work better than waiting for it to complete: reading a line **may** be safely interrupted before the thread receives it, and later restarted, as long as any characters buffered while trying to read the line are not lost but are returned when the line is read again.

The author proposes to make possible to split the specification of the value of a lazy variable into two parts. The first part obtains some resource, possibly waiting for it

indefinitely, and it should be run in synchronous mode. After it completes, signals are blocked and the value of the lazy variable is computed basing on the resource. The first part must be safe to be started again from the beginning if it was interrupted in synchronous mode.

For example in a lazily read list of lines the resource of each lazy variable is a line being read. And in a lazy `Map` (a function which applies a function to each element of a lazy list and collects a lazy list of results), the resource is the corresponding element of the source list. The thread which computes the value of the lazy variable can be interrupted while it is still waiting for the resource, and exceptions resulting from this phase are not stored in the lazy variable; the computation is repeated from the beginning the next time the variable is accessed.

Something more elaborated is needed in some cases. In a function which transforms a lazy list by selecting only values which satisfy a certain predicate, during evaluation of a single lazy variable the computation state may alternate between taking the next element of the source and computing the predicate. This happens when the value of the predicate is `False` several times in a row, and thus the function proceeds along the input without producing new elements of the output. Taking each next element should be done in synchronous mode, and computing the predicate should be done with signals blocked.

Moreover, after a given application of the predicate was determined to be `False` and the next element is examined, the lazy variable should be committed to restart from the next source element instead of from the very beginning, in order for the predicate to be used exactly once for each element.

To solve this, a special operation `CommitLazy` is provided, which can be used during the second phase of the evaluation of a lazy variable. It has two arguments: a function which obtains a resource again, and a function which transforms it into the value of the variable again. It causes the lazy variable to go back to the first phase, with the two functions replacing the original definition of the lazy variable.

The major problem with this extension of lazy variables is that it's quite complicated to understand and use.

## 4 FORKING A PROCESS AND THREADS

### 4.1 Design

Unix provides the `fork()` function which duplicates the current process. At the moment of forking the state of the process is cloned and continues in two separate processes concurrently. Open files are shared, including the file pointer. The desirable semantics of `fork()` in the presence of multiple threads is very unclear.

The most common purpose of forking is to set up the environment for another program (set environment variables, change current directory, redirect I/O etc.) and call a function from the `execv()` family, which loads and executes a different program. Forking can also be used to parallelize computation, as an alternative to threads.

It's easy to specify that all threads are cloned on fork, but this choice is rarely useful. If one thread is writing to a file and at the same time a different thread does `fork()` and `execv()`, then we don't want a clone of the first thread to continue writing to the file between `fork()` and `execv()`, because the file pointer is shared between threads and writing would interfere with the original process.

This suggests that all threads other than the one doing the `fork()` should disappear;

this is how POSIX threads behave. Unfortunately this may be bad too. If setting up the environment for the other program accesses a variable protected by a mutex, and at the time of the fork the mutex was held by a different thread, then in the child process there is nobody to unlock the mutex and we have a deadlock.

POSIX provides `pthread_atfork()` function for registering handlers to be run around forking, and they can be used to temporarily lock global mutexes. This mechanism sometimes helps but it's not enough. For example when the garbage collector has spawned a background thread for running finalizers of objects which are about to die, this thread should not be evaporated by fork, otherwise some objects in the child process are left half-finalized or not finalized at all.

Generally we want to wait until all other threads reach some safe points (e.g. when they are not holding any mutexes), then do the fork, and cancel the threads in the child process, so they don't proceed further but only release resources which were supposedly cloned by fork. A thread should be suspended at its safe point while we are waiting for others.

If we know that a particular usage of `fork()` is soon followed by `execv()`, the POSIX version of fork which evaporates all threads is better, because it doesn't have to wait for threads to reach safe points. For other uses a safer version is desirable.

How should safe points be defined? Introducing special functions for marking them would imply that functions which weren't explicitly adapted for forking get in the way of a different thread wanting to fork. Fortunately there is an existing criterion which works well in practice and is easy to implement once the framework of asynchronous signals is in place.

## 4.2 Implementation

Let's assume that a thread is safe to be forked when it is ready to process signals sent to it.

The design of blocking of signals aims to distinguish regions where the thread is in the middle of a sensitive sequence of operations which should not be interrupted, i.e. when they should always run into completion without causing the thread to process asynchronous events in the middle, nor to abort the computation, nor to clone the current continuation.

The thread-aware `ForkProcess` proceeds as follows. Each thread other than the thread doing the fork is sent a signal which asks it to perform a certain action. The action sends back a reply, which informs the forking thread that this thread has reached a safe point and is now ready for forking, and suspends the thread on a semaphore. The thread doing the fork waits until all threads send a reply.

In addition it monitors threads which are finishing (it no longer expects a reply from such a thread) and threads which are being created (they are sent the signal too).

When all expected replies have been received, we perform a variant of raw fork which clones all threads. Then in the parent process we wake up the threads and let them continue, while in the child process they are canceled.

The possibility of implementing a fork which clones all threads depends on how our threads are implemented and what primitives the OS provides:

- If threads are implemented entirely in userspace (*green threads* or *fibres*), there is no problem because plain `fork()` will clone the whole process memory, including out structures which describe running threads.

- If threads are mapped 1-1 to OS threads, then some systems (e.g. Solaris) provide a function `forkall()` which let all threads continue running in both processes. Unfortunately this function is not specified by POSIX, even though it has been proposed (the reasoning of the rejection was that it's too hard to ensure that other threads are at places which are safe to clone).

  Even if it's not possible to continue other threads in the child process, the semantics is still acceptable. Threads reach safe points and disappear in the child process instead of being canceled gently. Some resources might not be released, but otherwise data should not be corrupted and there should be no deadlocks.

- Kogut uses a mixture of green and OS threads, according to an interesting design described in [5]. The effect is that threads canceled by fork usually have a chance for cleanup, but if a thread is inside a callback from C at the time of the fork, then it evaporates when the callback returns, because the active C functions it wants to return to no longer exist.

This shows that a variant of `fork()` which clones all threads is useful for implementing the safe variant. Kogut provides all three variants: `ForkProcess`, `ForkProcessCloneThreads` and `ForkProcessKillThreads`.

## 5   USING SIGNALS SYNCHRONOUSLY

Some languages, in particular Common Lisp, extend the notion of exceptions to conditions where computation can resume the computation instead of aborting, depending on what the handler chooses.

Instead of making some exceptions resumable, the mechanism of signals can be used for this. After all, signal handlers already establish a relation between signal values, dynamic regions of code, and handlers, such that a handler can either do something locally and continue or abort the current computation.

So the Kogut language allows signals to be sent to the self thread, using a different operation than signalling another thread. Such signal is completely synchronous and is thus processed regardless of the blocking state. The return value of such a signal handler (if it returns) is not ignored but can be used by the signalling code.

In order to ease the decision whether some condition should better be reported by a signal or by an exception, any unhandled signal is automatically thrown as an exception from the same point. This means that non-fatal conditions can be reported using signals, and handled either as signals or as exceptions.

## 6   CONCLUSION

The need to express interruptible computations is often overlooked by language designers. This is understandable in low-level imperative languages, where interruption which doesn't terminate the whole program is unsafe most of the time, so polling for an interrupt must be done explicitly anyway. The other end of the spectrum, purely functional code, has the luxury that interruption at an arbitrary point is always safe. But even Haskell has an imperative subsystem, which from the point of view of asynchronous signals yields an impurely functional language. Such languages benefit from a dedicated mechanism for delivering signals to threads and selective blocking of signal processing.

The paper proposes a semantics for such selective blocking, which is a refinement of the mechanism used in Glasgow Haskell and proposed for Python. The proposal includes effects of some common language constructs on signal blocking (deterministic resource releasing, locking, exception handling).

This notion of signals is more general than asynchronous exceptions. The paper proposes a semantics for interaction of threads with Unix `fork()`, which relies on such signals.

## REFERENCES

[1] The Kogut Programming Language, http://kokogut.sourceforge.net/kogut.html

[2] John H. Reppy, *Asynchronous Signals in Standard ML*, August 1990

[3] Simon Marlow, Simon Peyton Jones, Andrew Moran, John Reppy, *Asynchronous Exceptions in Haskell*, November 2000

[4] Stephen N. Freund, Mark P. Mitchell *Safe Asynchronous Exceptions For Python*, December 2002

[5] Simon Marlow, Simon Peyton Jones, Wolfgang Thaller *Extending the Haskell Foreign Function Interface with Concurrency*, Proceedings of the ACM SIGPLAN workshop on Haskell, September 2004