The Concurrent Haskell Foreign Function Interface 1.0
An Addendum to the Haskell 98 FFI Report

Wolfgang Thaller

The master version of the Concurrent Haskell FFI Report is at `haskell.org`. Any corrections or changes in the report are found there.

Contents

1 Introduction

This report intends to define the interaction of two extensions to Haskell 98, namely the Foreign Function Interface and Concurrent Haskell. It is therefore an addendum to both the Haskell 98 FFI report and the (yet unwritten) Concurrent Haskell report. Familiarity with Haskell 98 and both extensions is assumed.

Concurrent Haskell itself does not require operating system thread primitives[1] to be used. Today's Concurrent Haskell implementations do in fact use their own scheduler loop and run all Concurrent Haskell threads in just one OS thread.

This is significantly more efficient and sometimes also easier to implement than solutions based on the OS primitives. However, there are problems with interfacing such an implementation with libraries written in other languages.

The functionality described in this addendum facilitates interoperation with foreign languages without sacrificing performance.

1.1 Definitions

Throughout this document, the term *Haskell thread* will be used to refer to the entity that is visible to Haskell programs. Every Haskell IO action runs in a Haskell thread, and you can create new Haskell threads using `forkIO`.

The term *OS thread* will be used to refer to threads managed by the operating system. All code that is run (both Haskell and foreign code) runs in an OS thread. New OS threads are created using OS-specific primitives, like `pthread\_create` on POSIX systems and `CreateThread` on Win32.

A Haskell run-time system is responsible for managing the relationship between OS threads and Haskell thread. Every Haskell thread has to be run by an OS thread to do anything at all.

2 Problems

This section outlines the problems with Haskell implementations that use a single OS thread for executing all Haskell threads.

2.1 Blocking foreign calls

If all Haskell scheduling is done in one OS thread, then there can be only one call to a foreign imported function in progress at any one time. While a foreign call is in progress, the Haskell run-time system is not in control, and therefore all other Haskell threads are blocked.

This severely limits the usefulness of Concurrent Haskell when used together with the FFI.

For some time now, there has been an optional extension to the Glasgow Haskell Compiler, the so-called "Threaded RTS", that allows non-blocking foreign calls to be made. However, this solution makes the problem described in the next section even worse.

2.2 Thread-local state

OS threads can be uniquely identified by their thread id and by their thread-local state. To libraries that make use of this, it does matter from which OS thread they are called from.

Thread-local state is used mostly to allow libraries that use global state variables as part of their interface to be used from multiple (OS) threads concurrently. One important example of this is OpenGL. OpenGL manages a lot of state variables for each "rendering context". The "current rendering context" is not passed as a parameter to OpenGL functions; rather, it is stored in a thread-local state variable. It is therefore possible to use OpenGL from two separate OS threads to render into two separate contexts (two separate windows).

[1] for example `pthread_create` on POSIX systems or `CreateThread` on Win32

When a Haskell implementation uses only one OS thread to schedule several Haskell threads, only one of these may access a library that uses thread-local state at any given time, as all Haskell threads will share the same OS-thread-local state.

GHC's threaded RTS made the problem worse: it doesn't execute a (foreign exported) callback in the same OS thread as the (foreign) function that calls it, and it may move all Haskell threads to a different OS thread at any time. While this behaviour sounds far-fetched, it is a good way to preserve GHC's good multithreading performance.

Foreign libraries that set a thread-local state variable to a particular value will not find the same value there when they are called from a different OS thread. For example, programs that use OpenGL segfault because OpenGL functions are called from an OS thread that does not have a current OpenGL context set. Similar problems arise with Microsoft's Win32, and Apple's Carbon and Cocoa libraries.

3 Requirements

The following requirements were used as guidelines for developing the solution to the above problems:

- Safe Foreign calls (i.e. calls not marked as unsafe) should not cause other threads to block.
- Libraries that rely on thread-local state should be usable from Haskell.
- The specification should be implementable in a way that allows a lot of "unsafe" foreign calls to be made with no additional overhead. Unsafe calls to libraries that rely on thread-local state must be possible.

  Using a library like OpenGL from Haskell would not be practical otherwise.
- The excellent performance of "lightweight" threads, that is, of using one OS thread to execute all Haskell threads, should not be sacrificed. Performance should still OK when using the new features with only a few threads (i.e. not more than commonly used from multithreaded C programs).

  This requirement is what makes this whole document necessary in the first place. Using exactly one OS thread for every Haskell thread solves the problems by sacrificing some performance.
- The specification shouldn't explicitly require lightweight threads to exist. The specification should be implementable in a simple and obvious way in Haskell systems that always use a 1:1 correspondence between Haskell threads and OS threads.
- The specification shouldn't specify which particular OS thread should be used to execute Haskell code. It should be possible to implement it with e.g. a Haskell interpreter running in one OS thread that just uses other OS threads for foreign calls.

4 Informal semantics

Here's the basic idea:

Haskell threads and OS threads.

- Every Haskell thread is *either* unbound, *or* bound to a exactly one OS thread.
- At most one Haskell thread may be bound to one OS thread. In particular, `forkIO` forks a new unbound Haskell thread.
- A Haskell thread, bound to a new OS thread, can be created with `forkOS`.

Foreign interface.

- No `safe` vs `threadsafe` distinction[2]. But we retain the `safe/unsafe` distinction.
- A foreign call made by a Haskell thread is (guaranteed to be) made by its bound OS thread, if any.
- If a `safe` foreign call blocks, then no Haskell threads block. (Remember, every OS thread has at most one Haskell thread bound to it.)
- A foreign call *into Haskell* (via `foreign export` or `foreign import wrapper`) is run by a Haskell thread bound to the OS thread that made the call.

Open questions and notes.

- Notice that, there *can* be a 1-1 mapping between Haskell threads and OS threads. Furthermore, we can run efficiently on an SMP.

5 Formal semantics

The syntax of a native thread is this:

$$
\begin{array}{llcll}
\text{Native thread} & t & ::= & N[S] & \\
 & & & & \\
\text{Native thread stack} & S & ::= & \epsilon & \text{Empty} \\
 & & | & H : S & \text{Executing Haskell} \\
 & & | & F^{si}\ a_{bt} : S & \text{Executing foreign code} \\
 & & | & \bullet & \text{Unknown} \\
 & & & & \\
\text{Safety indicator} & si & ::= & u & \text{Unsafe} \\
 & & | & s & \text{Safe}
\end{array}
$$

A native thread of form $N[S]$ has thread-id N, while S is an abstraction of its call stack. If H is on top of the stack, the thread is willing to execute a Haskell thread. If $F^{si}\ h$ is on top of the stack, the thread is in the process of dealing with a call to a foreign function, which will return its result to the Haskell thread h. The safety-indicator si is from the FFI spec.

A native thread of form $N[H]$ has a stack that exists only to serve Haskell threads, and so can safely block inside a foreign call without mucking anything else up. We might call them "worker threads".

[2] "`threadsafe`" has already been removed from the current Release Candidate of the FFI addendum

The syntax of a Haskell thread is this:

Haskell thread	h	$::=$	$(a)_{bt}$	
Bound thread id	bt	$::=$	ϵ	Not bound
		$\mid$	N	Bound to native thread N
Haskell action	a	$::=$	p `>>` a	Sequence
		$\mid$	RET	Return from a call into Haskell
Primitive action	p	$::=$	τ	Internal action
		$\mid$	`forkIO` a	Fork a thread
		$\mid$	`forkOS` a	Fork a native thread
		$\mid$	$F^{si}\ f$	Foreign call

A Haskell thread h of form $(a)_N$ has action a. The indicator N identifies the native thread N to which the Haskell thread is *bound*.

An action a is a sequence of primitive actions, finishing with a return of some kind. A primitive action is either some internal Haskell thing (such as performing a bit of evaluation, or operating on an `MVar`), or else it is a call to a foreign function f.

We do not model the data passed to, or returned from, a foreign call, nor any details of what "internal Haskell" operations are.

5.1 Evolution

We describe how the system evolves in a very standard way, using transition rules, of form

$$\mathcal{N};\mathcal{H} \ \Rightarrow\ \mathcal{N}';\mathcal{H}'$$

The structural rules are these:

$$\frac{\mathcal{N};\mathcal{H} \ \Rightarrow\ \mathcal{N}';\mathcal{H}'}{\mathcal{N}\cup\{t\};\mathcal{H} \ \Rightarrow\ \mathcal{N}'\cup\{t\};\mathcal{H}'} \qquad \frac{\mathcal{N};\mathcal{H} \ \Rightarrow\ \mathcal{N}';\mathcal{H}'}{\mathcal{N};\mathcal{H}\cup\{h\} \ \Rightarrow\ \mathcal{N}';\mathcal{H}'\cup\{h\}}$$

These standard rules allow us to write the interesting transitions with less clutter.

$$
\begin{array}{rcll}
N[H:S];(\tau \texttt{ >> } a)_{bt} & \Rightarrow & N[H:S];(a)_{bt} & (INT) \\[2ex]
N[H:S];(\texttt{forkIO}\ b \texttt{ >> } a)_{bt} & \Rightarrow & N[H:S];(a)_{bt},(b)_{\epsilon} & (FORKIO) \\[2ex]
N[H:S];(F^{si}\ f \texttt{ >> } a)_{N} & \Rightarrow & N[F^{si}\ a_N : H:S]; & (FCALL1) \\
N[H];(F^{si}\ f \texttt{ >> } a)_{\epsilon} & \Rightarrow & N[F^{si}\ a_{\epsilon} : H:S]; & (FCALL2) \\[2ex]
N[F^{si}\ a_{bt} : S]; & \Rightarrow & N[S];a_{bt} & (FRET) \\[2ex]
N[\bullet]; & \Rightarrow & N[H:\bullet];(f \texttt{ >> } RET\ )_{N} & (HCALL1) \\[2ex]
N[F^{s}\ a : S]; & \Rightarrow & N[H:F^{s}\ a:S];\ (f \texttt{ >> } RET\ )_{N} & (HCALL2) \\[2ex]
N[H:S];(RET\ )_{N} & \Rightarrow & N[S]; & (HRET) \\[2ex]
;(RET\ )_{\epsilon} & \Rightarrow & ; & (HEND) \\[2ex]
(nothing) & \Rightarrow & N[H]; & (WKR) \\
 & & \text{where } N \text{ is fresh} & \\[2ex]
(nothing) & \Rightarrow & N[\bullet]; & (EXT) \\
 & & \text{where } N \text{ is fresh} & \\[2ex]
N[S]; & \Rightarrow & (nothing) & (NEND)
\end{array}
$$

WKR. This rule models the birth of new worker OS threads, in case they should all be blocked in a foreign call.

6 Primitives

The following primitives are exported from the module `Control.Concurrent`, except for the `forkProcess` function, which is only available on POSIX systems and exported from `System.Posix.Process`.

6.1 rtsSupportsBoundThreads

```
rtsSupportsBoundThreads :: Bool
```

Defined to be `True` if multiple OS threads are supported as described in this document. When `rtsSupportsBoundThreads` is `False`, the function `isCurrentThreadBound` below will always return `False`, and `forkOS` will fail.

Note that an implementation which uses a simple 1:1 correspondence between Haskell threads and OS threads will define `rtsSupportsBoundThreads` to be `True`.

6.2 forkIO and forkOS

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

As described in the formal semantics above.

This document does not specify the meaning of the `ThreadId` return value. The definition of what constitutes one Haskell thread used for the return value need not agree with the definition used for describing the formal semantics. Questions like "are thread ids preserved across foreign calls and call-backs" are outside the scope of this document.

6.3 isCurrentThreadBound

```
isCurrentThreadBound :: IO Bool
```

... should return `True` if and only if it is safe to use foreign calls that rely on thread-local state. That means it will return True when executed from a bound Haskell thread. It may also return `True` for threads that are not bound according to the above semantics if the run time system is implemented in such a way that thread-local-state can be used from all threads, e.g. using a 1-1 relationship between Haskell threads and OS threads.

This primitive is intended to make use of forkOS unnecessary when a bound thread is already available; take a look at `runInBoundThread` below.

6.4 forkProcess

```
forkProcess :: IO () -> IO ProcessID
```

The primitive `forkProcess` is available in the module `System.Posix.Process` on Posix platforms only. While it is based on the POSIX fork system call, it's semantics are slightly different: It only returns to the parent; it doesn't return to the child process, rather, the IO action passed as a parameter will be run as a bound thread in the child process. No other threads will be copied to the child process. When the IO action finishes, the child process will terminate.

7 Utility Functions

The following utility functions are exported from `Control.Concurrent`. They can be implemented in terms of the primitives above; simple reference implementations that ignore exception handling issues are provided below.

7.1 runInBoundThread

```
runInBoundThread :: IO a -> IO a
```

Run the IO computation passed as the first argument. If the calling thread is not bound, a bound thread is created temporarily. `runInBoundThread` doesn't finish until the IO computation finishes.

```
runInBoundThread action = do
    bound <- isCurrentThreadBound
    if bound
        then action
        else do
            mv <- newEmptyMVar
            forkOS (action >>= putMVar mv)
            takeMVar mv
```

7.2 runInUnboundThread

```
runInUnboundThread :: IO a -> IO a
```

Run the IO computation passed as the first argument. If the calling thread is bound, an unbound thread is created temporarily using `forkIO`. `runInBoundThread` doesn't finish until the IO computation finishes.

```
runInUnboundThread action = do
    bound <- isCurrentThreadBound
    if bound
        then do
            mv <- newEmptyMVar
            forkIO (action >>= putMVar mv)
            takeMVar mv
        else action
```