# A semantics for foreign threads

The Haskell FFI team

May 1, 2003

## 1 Introduction

Threads created using OS primitives (for example pthreads on POSIX systems) can have thread-local state. Several libraries make use of this thread-local state. Most Haskell systems use just one OS thread to execute all foreign code. With GHC's threaded RTS, this is not even necessarily the same OS thread all the time. Therefore it is not possible to use foreign libraries that depend on thread-local state reliably.

The following foreign libraries are known to be affected:

- OpenGL
- Carbon and Cocoa (Apple's GUI toolkits)

## 2 Requirements

- It should be possible for Haskell code to arrange that a sequence of calls to a given library are performed by the same native thread and that if an external library calls into Haskell, then any outgoing calls from Haskell are performed by the same native thread.
- The specification should be implementable in a way that allows a lot of foreign calls to be made with no additional overhead with respect to GHC's current "unsafe" foreign calls.
- The good performance of the existing lightweight "green" threads in GHC 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).
- The specification shouldn't explicitly require lightweight "green" 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.
- There should be no unexpected blocking. Especially, threadsafe calls should never cause other threads to block.

# Part I

# First Proposal

## 3 Proposed design

The key extension is the ability to create a "bound" foreign export, that can be called from a foreign language. The intuitive idea is that "`bound`" means that the native OS thread that makes the call into Haskell is also guaranteed to run any `foreign` calls that the Haskell thread makes.

### 3.1 Extension to the FFI Syntax

A bound foreign export is declared by adding the specialid `bound` after the calling convention (before the exported entity string).

The bound attribute can also apply to thunks created by foreign imported "wrapper" functions. These stub factories are the only situation where the `bound` attribute applies.

Example:

```
foreign export ccall bound "CFoo" haskellFoo
    :: IO ()
foreign import ccall bound "wrapper" wrapHaskellFoo
    :: IO () -> IO (FunPtr (IO ()))
```

The `bound` attribute should only be valid for exported entities that are in the IO monad.

### 3.2 Library Addition

A function named `forkBoundThread` should be added to the library (somewhere in `Foreign` or `Concurrent`). A possible implementation for POSIX systems is given below.

```
forkBoundThread :: IO () -> IO ()

-- pthread_create(thread, attr, start_routine, arg)
foreign import ccall pthread_create
    :: Ptr (Ptr ())
    -> Ptr ()
    -> FunPtr (Ptr () -> IO (Ptr ()))
```

```
    -> Ptr ()
    -> IO CInt

foreign export ccall bound "wrapper" wrapHaskell
    :: (Ptr () -> IO (Ptr ()))
    -> IO (FunPtr (Ptr () -> IO (Ptr ())))

forkBoundThread action =
    wrapHaskell (\ptr -> do
            freeHaskellFunPtr thunk
            action
            return nullPtr
        )
    >>= \thunk -> alloca
    >>= \thread -> pthread_create thread nullPtr thunk nullPtr
    >> return ()
```

# 4 Operational semantics

While the intuition is clear, it is quite difficult to say *precisely* what a "`bound`" export means. The purpose of this section is to give a precise description, using a simple operational semantics.

## 4.1 State

The state of the system is represented by a pair

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

where

- $\mathcal{N} = \{t_1, \ldots, t_n\}$ is a set of *native threads.* A native thread is directly supported by the underlying operating system, and may have some thread-local state. It may therefore be important which native thread executes foreign (non-Haskell) code.
- $\mathcal{H} = \{h_1, \ldots, h_n\}$ is a set of *Haskell threads.* Haskell threads are supposed to be extremely lightweight, and can be created very cheaply. A Haskell thread can only make progress if a native thread acts as its execution engine.

The syntax of a native thread is this:

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

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. *(What about referring to the $H$ state as "inside the Haskell runtime system"? It includes things like GC and other "administrative tasks". A native thread might be in the $H$ state without ever really executing Haskell code.)* 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.

The syntax of a Haskell thread is this:

$$
\begin{array}{llcll}
\text{Haskell thread} & h & ::= & (a)_{bt} & \\
\\
\text{Haskell action} & a & ::= & p \gg a & \text{Sequence} \\
 & & | & R\ N & \text{Return to native thread } N \\
\\
\text{Primitive action} & p & ::= & \tau & \text{Internal action} \\
 & & | & F^{si}\ f & \text{Foreign call} \\
\\
\text{Bound thread id} & bt & ::= & \epsilon & \text{Not bound} \\
 & & | & N & \text{Bound to native thread N}
\end{array}
$$

A Haskell thread $h$ of form $(a)_{bt}$ has action $a$. The indicator $bt$ tells whether the Haskell thread is *bound to* a native thread. If $bt$ is $\epsilon$, the Haskell thread is not bound to a native thread; if $bt$ is $N$, the Haskell thread is bound to native thread $N$. Specifing exactly what it means for a Haskell thread to be bound to a native thread is the main purpose of this document.

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.

## 4.2 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.

$$N[H:S];(\tau \mathbin{>\!>} a)_{bt} \;\Rightarrow\; N[H:S];(a)_{bt} \qquad (INT)$$

$$N[H];(F^{si}\ f \mathbin{>\!>} a)_{\epsilon} \;\Rightarrow\; N[F^{si}\ a_{\epsilon}:H]; \qquad (FCALL1)$$

$$N[H:S];(F^{si}\ f\mathbin{>\!>} a)_{N} \;\Rightarrow\; N[F^{si}\ a_{N}:H:S]; \qquad (FCALL2)$$

$$N[F^{si}\ a_{bt}:S]; \;\Rightarrow\; N[S];a_{bt} \qquad (FRET)$$

$$N[\bullet]; \;\Rightarrow\; N[H:\bullet];(f \mathbin{>\!>} R\ N)_{bt} \qquad (HCALL1)$$
$$\text{where } bt = \epsilon \quad \text{if } f \text{ is a non-bound foreign export}$$
$$= N \quad \text{if } f \text{ is a bound foreign export}$$

$$N[F^{si}\ a_{bt}:S]; \;\Rightarrow\; N[H:F^{si}\ a_{bt}:S];\ (f \mathbin{>\!>} R\ N)_{bt} \qquad (HCALL2)$$
$$\text{where } si \in \{s,t\}$$
$$\text{and } bt = \epsilon \quad \text{if } f \text{ is a non-bound foreign export}$$
$$= N \quad \text{if } f \text{ is a bound foreign export}$$

$$N[H:S];R\ N \;\Rightarrow\; N[S]; \qquad (HRET)$$

$$(nothing) \;\Rightarrow\; N[H]; \qquad (WKR)$$
$$\text{where } N \text{ is fresh}$$

$$(nothing) \;\Rightarrow\; N[\bullet]; \qquad (EXT)$$
$$\text{where } N \text{ is fresh}$$

Here is what the rules mean:

- (INT) says that an internal action of a Haskell thread can be executed by any native thread that is in the Haskell state; that is, has $H$ on top of its stack. This is true even if the native thread has called a bound foreign export, and that Haskell function is still running.
- On the other hand, (FCALL1) says that only a vanilla worker thread $N[H]$ (i.e. with nothing on its stack except $H$) can execute an unbound foreign

call. Similarly (FCALL2) says that only native thread $N$ can execute a foreign call from a Haskell thread bound to $N$. (This is under the proposal that foreign imports are not labelled bound/unbound.)

- In either (FCALL) case, the Haskell thread is removed from the pool, and captured in the native thread's stack. The rule (FRET) reinstates the Haskell thread (bound or otherwise) when the native thread decides to return from the call.
- Rules (HCALL1) and (HCALL2) deal with foreign code calling Haskell. (HCALL1) starts with a native thread that comes "out of the blue" with an unknown stack $\bullet$. It just puts a new Haskell thread into the pool, bound or otherwise depending on whether the foreign export being called is labelled as bound.

  (HCALL2) is similar, except that we aren't allowed to make such a call if the native thread is in the middle of an unsafe foreign call. The difference between (HCALL1) and (HCALL2) looks very unsavoury.
- (HRET) deals with returning from a completed Haskell thread to the appropriate calling native thread. Notice that even an unbound Haskell thread must return its result to the right calling native thread, hence the "return action" $R\ t$ tacked onto the end of the Haskell thread.
- (WKR) and (EXT) allow new native threads to enter the system.

# 5 Issues

- Somebody suggested labelling foreign imports as bound or unbound, and allowing the RTS to choose any thread it wants for executing unbound foreign imports.

  However, we need top performance for *all* foreign imports, not just unbound ones. HOpenGL applications make hundreds and thousands of unsafe calls, all of which have to be bound to a specific native thread. Now if bound foreign imports need to be as fast as unbound foreign imports [1], we don't need the additional complexity.
- The meaning of "`bound`" in the FFI declarations is probably rather unobvious to people who haven't been following this discussion. Can somebody think of something that is more meaningful?
- Some very broken libraries (Apple's GUI libraries) seem to have initialization routines (that run prior to main) which initialize thread local state for the "main thread" (the OS thread that the C language `main()` function runs in). Some routines can then only be run in that main thread. Some

[1] This will be the case in GHC

way is needed to allow the programmer to specify that the haskell thread that the `main` action runs in should be bound to the main OS thread.

- In which module in the libraries should the `forkBoundThread` function go?

# Part II

# Second Proposal

This proposal avoids introducing yet another language extension, and it allows foreign calls to be bound to the main thread (which is required to use Apple's libraries).

The main idea is that Haskell threads are at first only "associated" with native threads. There is a combinator `threadbound :: IO() -> IO()` (somebody *please* think of a better name) that "binds" a Haskell thread to its associated native thread, or to a new native thread if there is no associated native thread.

Foreign calls from inside the `threadbound` combinator are guaranteed to be all executed in the same thread, which is guaranteed not to be used for other foreign calls from other Haskell threads.

A Haskell thread created by calling-in from Foreign Land is automatically associated with the native thread that called the foreign exported function. There is no performance penalty if the `threadbound` combinator is never used.

## 6 Operational semantics

### 6.1 State

The state of the system is represented by a pair

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

where

- $\mathcal{N} = \{t_1, \ldots, t_n\}$ is a set of *native threads.* A native thread is directly supported by the underlying operating system, and may have some thread-local state. It may therefore be important which native thread executes foreign (non-Haskell) code.
- $\mathcal{H} = \{h_1, \ldots, h_n\}$ is a set of *Haskell threads.* Haskell threads are supposed to be extremely lightweight, and can be created very cheaply. A Haskell thread can only make progress if a native thread acts as its execution engine.

The syntax of a native thread is this:

| | | | | |
|---|---|---|---|---|
| Native thread | $t$ | $::=$ | $N[S]$ | |
| Native thread stack | $S$ | $::=$ | $\epsilon$ | Empty |
| | | $\mid$ | $H : S$ | Executing Haskell |
| | | $\mid$ | $F^{si}\ h : S$ | Executing foreign code |
| | | $\mid$ | $\bullet$ | Unknown |
| Safety indicator | $si$ | $::=$ | $u$ | Unsafe |
| | | $\mid$ | $s$ | Safe |
| | | $\mid$ | $t$ | Thread-safe |

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 "inside the Haskell runtime system" (that includes, but is not limited to, executing Haskell code). 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.

The syntax of a Haskell thread is this:

| | | | | |
|---|---|---|---|---|
| Haskell thread | $h$ | $::=$ | $(a)_{at}$ | |
| Haskell action | $a$ | $::=$ | $p >> a$ | Sequence |
| | | $\mid$ | $\epsilon$ | End of thread |
| Primitive action | $p$ | $::=$ | $\tau$ | Internal action |
| | | $\mid$ | $F^{si}\ f$ | Foreign call |
| | | $\mid$ | $B(a)$ | "`threadbound`" combinator |
| | | $\mid$ | $B(a)_{bt}$ | the same with specified native thread |
| | | $\mid$ | $ForkIO(a)$ | `forkIO`: fork a new thread |
| Associated thread id | $at$ | $::=$ | $\epsilon$ | Not associated |
| | | $\mid$ | $N$ | Associated with native thread N |
| Bound thread id | $bt$ | $::=$ | $N$ | Bound to native thread N |

Note that $B(a)_{bt}$, the "`threadbound`" combinator with an explicitly specified native thread, is only an "intermediate result" of the rules below. There will be no way to directly specify this action in a Haskell program (or if there will be, it will be a separate extension to the FFI).

In the other proposal (and in a previous version of this one), there was a $R\ N$ action for returning to foreign land after a call-in. This is not necessary here, as Haskell threads always return to their associated native threads. Threads forked using forkIO do not return anywhere, they just terminate.

## 6.2 Initial State

A program starts with a single Haskell thread that is associated with the native thread that executes the C language `main()` function. This Haskell thread executes the Haskell `main` action.[2]

A Haskell thread spawned using `forkIO` is not associated with any native thread.

[2]This resolves the issue that some broken libraries initialize thread-local state for the main thread from places like C++ static initializers and therefore insist on other functions to be called from the main thread and not from any other thread.

### 6.3 Evolution

$$
\begin{array}{rcll}
N[H:S];(B(a) >> b)_N & \Rightarrow & N[H:S];(B(a)_N >> b)_N & (BIND1) \\
;(B(a) >> b)_\epsilon & \Rightarrow & N[H];(B(a)_N >> b)_\epsilon & (BIND2) \\
N[H:S];(B(\epsilon)_N >> b)_N & \Rightarrow & N[H:S];(b)_N & (UNBIND1) \\
N[H];(B(\epsilon)_N >> b)_\epsilon & \Rightarrow & ;(b)_\epsilon & (UNBIND2) \\
N[H:S];(B(B(a) >> b)_{bt} >> c)_{at} & \Rightarrow & N[H:S];(B(a >> b)_{bt} >> c)_{at} & (NEST) \\
;(\tau >> a)_{at} & \Rightarrow & ;(a)_{at} & (INT1) \\
;(B(\tau >> a)_{bt} >> b)_{at} & \Rightarrow & ;(B(a)_{bt} >> b)_{at} & (INT2) \\
N[H];(F^{si}\ f >> a)_\epsilon & \Rightarrow & N[F^{si}\ a_\epsilon : H]; & (FCALL1) \\
N[H:S];(F^{si}\ f >> a)_N & \Rightarrow & N[F^{si}\ a_N : H : S]; & (FCALL2) \\
M[H];(F^{si}\ f >> a)_N & \Rightarrow & M[F^{si}\ a_N : H]; & (FCALL3) \\
N[H:S];(B(F^{si}\ f >> a)_N >> b)_a t & \Rightarrow & N[F^{si}\ (B(a)_N >> b)_{at} : H : S]; & (FCALL4) \\
N[F^{si}\ a_{at} : S]; & \Rightarrow & N[S]; a_{at} & (FRET) \\
N[\bullet]; & \Rightarrow & N[H:\bullet];(f)_N & (HCALL1) \\
 & & \text{where } f \text{ is a foreign export} & \\
N[F^{si}\ a_{at} : S]; & \Rightarrow & N[H : F^{si}\ a_{at} : S];\ (f)_N & (HCALL2) \\
 & & \text{where } f \text{ is a foreign export} & \\
 & & \text{and } si \in \{s,t\} & \\
N[H:S];(\epsilon)_N & \Rightarrow & N[S]; & (HRET) \\
;(ForkIO(a) >> b)_{at} & \Rightarrow & ;(b)_{at},(a)_\epsilon & (FORK1) \\
;(B(ForkIO(a) >> b)_{bt} >> c)_{at} & \Rightarrow & ;(B(b)_{bt} >> c)_{at},(a)_\epsilon & (FORK2) \\
;(\epsilon)_\epsilon & \Rightarrow & (nothing) & (END) \\
(nothing) & \Rightarrow & N[H]; & (WKR) \\
 & & \text{where } N \text{ is fresh} & \\
(nothing) & \Rightarrow & N[\bullet]; & (EXT) \\
 & & \text{where } N \text{ is fresh} &
\end{array}
$$

Here is what the rules mean:

- (BIND1) states that a Haskell thread associated with a native thread $N$ becomes bound to the thread $N$ when the `threadbound` combinator is used (The `threadbound` combinator is annotated with the thread id it binds to).
- (BIND2) states that an unassociated Haskell thread becomes bound to a new native thread when the `threadbound` combinator is used.
- A Haskell thread becomes unbound again when the action inside the `threadbound` combinator is finished (UNBIND1). If the Haskell thread is not associated with the native thread, the native thread, which was spawned by (BIND2), terminates (UNBIND2).
- (NEST) says that nested invocations of the `threadbound` combinator are legal, but have no effect.
- Internal actions are simply executed. For the purposes of this proposal, it doesn't matter which native thread is used to execute them, so (INT1) doesn't mention any native thread. Internal actions inside a `threadbound` combinator are no different from other actions (INT2).
- (FCALL1) says that only a vanilla worker thread $N[H]$ (i.e. with nothing on its stack except $H$) can execute a foreign call from an unassociated, unbound thread.
- Foreign calls from a Haskell thread associated with a native thread $N$ can either be executed by the native thread $N$ (FCALL2) or by any vanilla worker thread $M[H]$ (FCALL3). This is left to the discretion of the run-time system - the fact that a Haskell thread is associated with a native thread has no effect by itself.
- (FCALL4) says that a foreign call from a bound Haskell thread (i.e. from inside the `threadbound` combinator) must be executed by the native thread that the Haskell thread is bound to.
- In all (FCALL) cases, the Haskell thread is removed from the pool, and captured in the native thread's stack. The rule (FRET) reinstates the Haskell thread (bound or otherwise) when the native thread decides to return from the call.
- Rules (HCALL1) and (HCALL2) deal with foreign code calling Haskell. (HCALL1) starts with a native thread that comes "out of the blue" with an unknown stack $\bullet$. It just puts a new Haskell thread into the pool and associates it with the native thread.

(HCALL2) is similar, except that we aren't allowed to make such a call if the native thread is in the middle of an unsafe foreign call. The difference between (HCALL1) and (HCALL2) looks very unsavoury[3].

- (HRET) deals with returning from a completed Haskell thread to the appropriate calling native thread, which happens to be the associated native thread.
- (FORK1) and (FORK2) say that threads forked using `forkIO` are not associated with any native thread.
- A thread that is not associated to a native thread (one spawned using `forkIO`) just vanishes when it's finished (END).
- (WKR) and (EXT) allow new native threads to enter the system.

From these rules follows an invariant that is very useful from an implementor's point of view:

> At any one time, there is at most one Haskell thread associated with, or bound to, any given native thread.

# 7 Issues

- `threadbound` is a terrible name.
- In which library module should `threadbound` go?
- Shound the threadbound combinator, when used from an unassociated Haskell thread, always create a fresh native thread, or may the native thread be reused?

  Rule (BIND2) implies that it is a new native thread, and rule (UNBIND2) says that the native thread dies after being used. An alternative might be to allow pooling of the threads by replacing rules (BIND2), (UNBIND1) and (UNBIND2) by:

  $$N[H]; (B(a) \texttt{>>} b)_\epsilon \quad \Rightarrow \quad N[H]; (B(a)_N \texttt{>>} b)_\epsilon \qquad (BIND2')$$

  $$; (B(\epsilon)_N \texttt{>>} b)_{at} \quad \Rightarrow \quad ; (b)_{at} \qquad (UNBIND')$$

  The obvious advantage of pooling is, of course, a possible performance gain; However, a "fresh" thread has well-defined thread-local state, but for a reused (pooled) thread, thread-local state is unknown. There shouldn't be any libraries that rely on threads being "fresh"[4], but having old thread-local state lying around could make bugs harder to reproduce.

[3] Maybe we shouldn't be modeling the safety levels here at all — after all, they are independent of the issue at hand.

[4] but of course, you can never know what strange ideas people can come up with

# Part III

# Third Proposal

This proposal, from Simon Marlow and Simon Peyton Jones, is another attempt to simplify the bound-threads design. We think that it's more or less what Daan proposed, only he never wrote it up.

We have a growing feeling that the First Proposal including 'bound' foreign exports and so forth is too complicated - especially when OS designers are already trying to solve similar problems. c.f. the arguments about 1:1 vs. M:N threading in Linux and other OSs. If both the OS and the Haskell RTS are doing dynamic M:N style thread allocation, it seems like overkill, and we don't have a lot of confidence that we would be able to tune this kind of system effectively.

## 8 Informal semantics

Here's the basic idea:

**Haskell threads and OS threads.**

- Every Haskell thread is bound to a exactly one OS thread.
- Many Haskell threads may be bound to the same OS thread. In particular, `forkIO` forks a new Haskell thread bound to the same OS thread as the call of `forkIO`.
- A Haskell thread, bound to a new OS thread, can be created with `forkOS`.

**Foreign interface.**

- No `safe` vs `threadsafe` distinction. 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 that foreign call blocks, then so do all the Haskell threads bound to that OS thread.
- 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.

**Non-blocking calls.**

- To make a foreign call from Haskell which does not block other Haskell threads bound to the same OS thread, it's easy to hand off to a new OS thread:

```
nonBlocking :: IO a -> IO a
nonBlocking a = do { r <- newEmptyMVar ;
                     forkOS (do { v <- a; putMVar r v }) ;
                     takeMVar r }

foreign import foo :: Int -> IO Int

....do { x <- .... ;
         r <- nonBlocking (foo x) ;
         ... }...
```

- The `nonBlocking` function looks expensive, because it starts a new OS thread, but it's easy to make a more efficient version that keeps a pool of Haskell threads, each with a dedicated OS thread, ready to execute any IO actions passed to `nonBlocking`. This allows us to write in Haskell what we would otherwise have to implement in the runtime system
- If the programmer knows that the Haskell thread will have its own dedicated OS thread (e.g. it's a thread born from a call into Haskell) then he can avoid the `nonBlocking` wrapper and the consequent OS-thread-switch costs. Let the programmer decide!

**Open questions and notes.**

- We lose out by not being able to take advantage of SMP automatically for existing Concurrent Haskell programs. But: (a) running multithreaded Haskell programs on SMP is an open research problem anyway, and

  (b) Assuming we solve (a), and your OS has a pretty good threading implementation, then you can just use forkOS everywhere (i.e. get a 1:1 Haskell to OS thread mapping).
- It'd be possible to allow any OS thread to run a Haskell thread (not just its bound OS thread), provided it didn't make any foreign calls.

# 9 Formal semantics

The syntax of a native thread is this:

| | | | | |
|---|---|---|---|---|
| Native thread | $t$ | $::=$ | $N[S]$ | |
| Native thread stack | $S$ | $::=$ | $\epsilon$ | Empty |
| | | $\mid$ | $H : S$ | Executing Haskell |
| | | $\mid$ | $F^{si}\ a : S$ | Executing foreign code |
| | | $\mid$ | $\bullet$ | Unknown |
| Safety indicator | $si$ | $::=$ | $u$ | Unsafe |
| | | $\mid$ | $s$ | Safe |

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.

The syntax of a Haskell thread is this:

| | | | | |
|---|---|---|---|---|
| Haskell thread | $h$ | $::=$ | $(a)_N$ | Bound to native thread N |
| Haskell action | $a$ | $::=$ | $p$ >> $a$ | Sequence |
| | | $\mid$ | $R$ | Return from a call into Haskell |
| Primitive action | $p$ | $::=$ | $\tau$ | 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.

## 9.1 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.

$$\begin{array}{rcll}
N[H:S];(\tau \mathrel{\texttt{>>}} a)_M & \Rightarrow & N[H:S];(a)_M & (INT) \\[1ex]
N[H:S];(F^{si}\;\; f\texttt{>>}\; a)_N & \Rightarrow & N[F^{si}\;\; a_N : H : S]; & (FCALL) \\[1ex]
N[H:S];(\texttt{forkIO}\; b \mathrel{\texttt{>>}} a)_N & \Rightarrow & N[H:S];(a)_N,(b)_N & (FORKIO) \\
N[H:S];(\texttt{forkOS}\; b \mathrel{\texttt{>>}} a)_N & \Rightarrow & N[H:S],M[H];(a)_N,(b)_M & (FORKOS) \\[1ex]
N[F^{si}\;\; a:S]; & \Rightarrow & N[S];a_N & (FRET) \\[1ex]
N[\bullet]; & \Rightarrow & N[H:\bullet];(f \mathrel{\texttt{>>}} R\;)_N & (HCALL1) \\[1ex]
N[F^{s}\;\; a:S]; & \Rightarrow & N[H:F^{s}\;\; a:S];\;\; (f \mathrel{\texttt{>>}} R\;)_N & (HCALL2) \\[1ex]
N[H:S];(R\;)_N & \Rightarrow & N[S]; & (HRET) \\[1ex]
(nothing) & \Rightarrow & N[\bullet]; & (EXT) \\
& & \text{where } N \text{ is fresh} &
\end{array}$$

The pattern of these rules is exactly as for the First Proposal, but they are noticeably simpler. They are very much simpler than the Second Proposal.