Lightweight concurrency: working notes

July 17, 2006

Olin Shivers

Simon Peyton Jones

Abstract

This is a set of brain-dump working notes only. Don't take it too seriously.

1. What we want do to

GHC has quite sophisticated support for concurrency. It supports:

- The notion of a *thread*. Threads are supposed to be extremely lightweight; it's fine to have millions of threads.
- A scheduler that decides what threads to run. The ocncurrency is pre-emptive.
- `MVars`, which allow threads to communicate, including the mechanism for blocking when rreading an empty `MVar` [5].
- Transactional memory [1].
- Support for multi-processors. In particular, GHC has a per-(virtual-)processor allocation area and run-queue. Threads stick to one processor by default (to get good locality), but idle processors can steal threads from busyones.

The trouble with all this is that most of it is implemented in the run-time system (RTS), which itself is written in C. This has three undesirable consequences. First, the RTS is rather large, and easy to get wrong. Second, only GHC HQ (and Simon Marlow in particular) can modify it. Third, many policy decisions are "baked into" this substrate.

We'd like to move a good chunk of GHC's concurrency support into a Haskell library, built on lower-level primitives. That would mean that Haskell programmers could swap out the GHC-supplied library, and use their own instead, if they wanted to experiment with different policies. Furthermore, the concurrency support is more likely to be correct if it's written in Haskell rather than C.

2. How we propose to do it

The idea of writing a concurrency library in the language itself is as old as the hills — well, as old as Mitch Wand anyway [**?**]. The basic idea is to suspend a thread by capturing its *continuation*, and resume it by invoking the continuation.

Other relevant papers:

- Scheme "parameters" see `http://download.plt-scheme.org/doc/301/html/mzscheme/mzscheme-Z-H-7.html#node_sec_7.9`
- Engines [2].
- Featherweight concurrency in C– [**?**].
- what else?

Nevertheless, it's not obvious exactly how to do the job.

3. Vocabulary and context

Vocabulary:

- The *substrate* is everything that is not written in Haskell; it is the foundation on which the concurrency library is written.
- The *concurrency library* implements the concurrency system presented to the programmer, using the substrate primitives.

The name of the game is to *specify the types, and the operations over those types, provided by the substrate*.

The concurrency library must support at least the following interfaces:

```
forkIO :: IO a -> IO ThreadId  -- Fork a threads

-- MVars
newMVar :: a -> IO (MVar a)
takeMVar :: MVar a -> IO a       -- Blocks if empty
putMVar  :: MVar a -> a -> IO () -- Blocks if full

-- Transactional memory
atomic :: STM a -> IO a
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
retry     :: STM ()
orELse    :: STM a -> STM a -> STM a
```

For the moment we are neglecting (a) asynchronous exceptions [3], (b) bound threads [4]

4. Things we agree on

- The substrate has a data type for continuations, and some operations over continuations (such as callcc).
- A *thread* has a continuation (which says how it will proceed when next run. But it may have lots of other stuff too, such as a thread-id, thread-local state, priority, parent thread (if threads have a parent-child relationship), and maybe more besides.
- Therefore the substrate has no business knowing about threads at all. Threads are built by the concurrency library.
- Similarly, the substrate knows nothing about scheduling threads; that too must be done by the library.

- Since *blocking* involves threads, and the substrate does not know about threads, operations that involve blocking or re-awakening threads must be under the control of the concurrency library. Similarly transactional memory.
- We would like to be able to program hierarchical schedulers. That is, a scheduler is a Haskell function that divides cycles among multiple threads (whatever its notion of a "thread" might be) – but any of those threads might in turn run a scheduler, and so on.

5. A first stab

Simon's first thoughts.

5.1 The IO Monad

One of the things provided by the substrate is the IO monad. The IO monad supports mutable state.

```
instance Monad IO

newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

But see Section 5.8.

5.2 Continuations

The substrate provides a type of *quasi-continuations*, with the following operations:

```
data QCont a    -- Opaque
cutTo   :: QCont a -> a -> IO b
newCont :: IO a -> QCont ()
callCC  :: (QCont a -> IO b) -> IO a
```

A value of type `QCont a` is a continuation expecting to be given a value of type `a`. We call these things "quasi-continuations", where the "quasi" means "like continuations but not entirely, so watch out". We will often say "continuation" rather than "quasi-continuation" from now on.

The function `cutTo` transfers control to a continuation, giving it the value it is awaiting. The function `newCont` makes a new continuation that, when awoken by `cutTo`, will run the specified I/O computation. Lastly, `callCC` does the usual thing. However, it captures a *delimited* continuation: it captures the continuation that represents the "rest of the current thread", and *not* the "rest of the entire program". In the vocabulary of delimited continuations, `newCont` pushes a prompt, and `callCC` captures the continuation down to that prompt.

The design of quasi-continuations deliberately exposes their expected implementation: *a continuation is represented by a stack*:

- `newCont` allocates a new stack object, and initialises it to run the given I/O computation when someone cuts to it.
- `cutTo` transfers control from one stack to another or possilby to a continuation lower down in the same stack.
- `callCC` is dirt cheap: it captures a pointer into the stack.

So `callCC` is like `setjmp`, and `cutTo` is like `longjump`.

Because of this intended implementation, a quasi-continuation is a value, but it is not fully first class:

1. A quasi-continuation is *linear*, or *one-shot*. That is, you can only `cutTo` a continuation once, and then it is used up.
2. When you capture a continuation with `callCC f`, the continuation passed to `f` dies if `f` returns.

These properties are not statically checked, and violating them might lead to unspecified behaviour. It would be nice to provide static guarantees that the properties are met, but that would almost certainly require something new from the type system. But we might be able to have dynamic (run-time) checks: see Section 5.6.

5.3 Threads

The details of how a thread is implemented are the business of the concurrency library, but to make it possible to write examples we'll suppose that the concurrency library (not the substrate) defines:

```
data Thread a
getThreadCont :: Thread a -> IO (QCont a)
setThreadCont :: Thread a -> IO (QCont a)
threadId :: Thread a -> ThreadId
```

Here a `Thread` value contains some mutable objects, and hence one can perform side-effecting operation like `setThreadCont` on a `Thread`. There are likely to be other operations on threads, though: creation, changing priority, and so forth. But these are enough for now.

A `Thread` contains a continuation, which itself is parameterised on the type of value it's expecting, and hence so is `Thread`. That is, a `Thread` can be blocked awaiting a return value.

5.4 Running a continuation

The concurrency library makes a thread by allocating a mutable structure containing (at least) a continuation, of type `QCont ()`, representing the rest of that computation of that thread.

If a scheduler wants to run a thread for a bit, it grabs the thread's continuation and runs it using the following substrate primitive:

```
run :: Int -> QCont () -> IO RunResult
data RunResult = More (QCont ()) | Done
```

If the continuation finishes, `run` returns `Done`. If the continuation yields voluntarily, or its time slice expires, `run` returns `More k`, where `k` is the continuation to run next.

Incidentally, `Done` does not necessarily mean that the thread is finished, only that the scheduler is no longer responsible for it. See Section 5.7, for example.

The `Int` is the number of cycles (ticks, microseconds, whatever) that the caller of `run` is prepared to allocate to the called continuation. The caller itself may be inside *his* caller's `run`. So `run 3 c` doesn't really mean "run `c` for 3 ticks"; rather, it means "run `c` for the next 3 ticks that are given to me by my caller".

Dual to `run` is `stop`:

```
stop :: RunResult -> IO Void
```

The idea is that `stop` returns the specified `runResult` to the enclosing `run`. So a voluntary yield could be written like this:

```
yield :: IO ()
yield = callCC (\k -> stop (More k))
```

In this way, a thread can *voluntarily* yield to its scheduler. But it can also yield involuntarily, something that must be supported directly by the substrate:

- The "time slice expires". Then the substrate packages up the continuation and returns a `More k` result to `run`. Just which `run` is re-activated depends on which time-slice has expired. For example:

```
do { c <- newCont (do { More d' <- run 6 d
                      ; run 6 d' }
   ; More c' <- run 10 c
   ; ..more... }
```

The outer `run` devotes 10 ticks to `d`. It devotes 6 ticks to `d`, and (assuming `d` does indeed return `More d'`, `c` tries to devote a futher 6 ticks to `d'`. However, after 4 of those ticks, the outer `run` expires, and `c` is arrested, returning control to `..more...` The continuation `c'` can be scheduled with another `run`, which will resume execution with the last 2 ticks of `d'`.

- The thread evaluates a thunk that is locked, becuase it is being evaluated by another thread. In that case, again the substrate packages up the continuation and returns it to the enclosing `run`. But see Section 5.5.

The substrate must be able to *find* the enclosing `run`(s) somehow.

5.5 Thunks

In old GHC, when a thread blocks on a locked thunk, the suspended thread was actually queued on the thunk itself, so that it could be un-blocked when the thunk was updated. Since thunk evaluation is definitely part of the substrate, this would be an awkward interaction of the different levels.

However, GHC *no longer* attaches the blocked thread to the thunk. Reason: that makes the act of updating a thunk much more epensive, becuase it must check whether there is a queue of blocked threads; and must do so using atomic instructions. Instead, the blocked thread polls the thunk; the thunk update plays no part in freeing the blocked thread.

This new mechanism is much easier for us. When a computation evaluates a locked thunk, the substrate can just package up the continuation and return it to the enclosing `run`. The polling process could amount to just running the thread again; if the thunk is still locked, the same thing will happen all over again (though perhaps this is not fantastically efficient).

Perhaps `RunResult` should indicate what took place, because the scheduler might want to schedule such threads differently.

5.6 Guaranteeing quasi-continuation usage

If a quasi-continuation is mis-used, you might get a seg-fault. You might think it is terrible for this to happen in allegedly-statically-typed language. However, *it's still better do to this stuff in Haskell rather than C*, for the reasons given above. It's a bit like having `unsafePerformIO`.

Idea. Maybe we can guarantee condition (2) statically after all. What if the argument to `callCC` was guaranteed never to return? Then we can be sure that k never dies, except perhaps by being `cutTo`, anbd condition (1) says you can only do that once. We can (nearly) statically guarantee that `f` doesn't return. Suppose there is no value of type `Void` *[***Simon***: what about bottom?]*:

```
callCC :: (QCont a -> IO Void) -> IO a
cutTo :: QCont a -> a -> IO Void
```

If the only way we can manufacture an `IO Void` is using `cutTo` and `stop` (or perhaps simply diverging) then we can be sure that the function passed to `callCC` does not return. Cool.

In any case, it's very easy to dynamically check condition (2): if the function called by `callCC` returns, report a runtime error.

5.7 MVars

When a thread blocks on an empty `MVar`, we want to attach the sleeping thread to the `MVar` itself. So the concurrency library might implement MVars like this:

```
newtype MVar a = MV (IORef (MVState a))
data MVState a = Full a [(a,Thread ())]
               | Empty [Thread a]
```

If an MVar is full, it can have zero or more threads waiting to put its value; if it is empty, there may be zero or more threads waiting to take its vaulue.

Here's the code for `take`. We're assuming that the operations on mutable state take place atomically.

```
takeMVar :: MVar a -> IO a
takeMVar (MV mv)
  = do { cts <- readIORef mv
       ; take mv cts }

take mv (Full x [])
  = do { writeIORef mv (Empty [])
       ; return x }
take mv (Full x ((x',t):ts))
  = do { writeIORef mv (Full x' ts)
       ; makeRunnable t
       ; return x }
take mv (Empty ts)
  = callCC $ \k ->
    do { t <- currentThread
       ; setThreadCont t k
       ; writeIORef mv (Empty (t:ts))
       ; stop Done }
```

Notice that the last block of code returns `Done` to the scheduler even though the thread is far from finished. Nevertheless, the scheduler can now ignore it, and in that sense it's `Done`.

To do this we've used a couple of new functions:

```
makeRunnable :: Thread -> IO ()
currentThread :: IO Thread
```

Since both involve threads, they can't be substrate primitives. Both require access to a structure created and owned by the enclosing `run`. This strucure might contain mutable variables (e.g. the run queue for that scheduler). This is a kind of thread-local state and we Must Have It: see Section 6.

While I think about it, though, notice that we could get away without the `RunResult` (I think) result of `run`. The `yield` function could put the current thread into the run queue. Aha, but the *implicit* yield from pre-emption could not do so.

5.8 Multiprocessor issues

- I have been writing `readIORef`, `writeIORef` etc, but in a multiprocessor the substrate must provide something akin to CAS. I'm not sure what the exact signature should. Probably at least a multi-word CAS, so as to support STM commit.
- Something about how to get N schedulers started when we have N processors. Perhaps the Right Thing is just to have the substrate expose the OS facilities:

  ```
  forkOS :: IO a -> IO OsThreadId
  ```

  The idea is that we spawn a real OS thread here. The business of multiplexing lightweight Haskell threads onto OS threads is then 100% done by the concurrency library.

5.9 Atomicity

We need some primitive notion of atomicity ("primitive" meaning "implemented by the substrate"). Here are two possibilities:

- Add compare-and-swap to `IORefs`, and make that the primitive. That's what real machines have.
- Implement transactional memory in the substrate. Possibly without `retry` and `orElse`?

5.10 Par

GHC supports the `par` combinator:

```
par :: a -> b -> b
```

`pat` puts a *spark* (thunk) into a *spark pool*. Idle processors look for work in the spark pool. We might have per-processor spark pools; along with stealing to grab sparks from the spark pool of another processor.

There are lots of different spark-pool strategies, and we'd like that to be programmable too. But here the task is a bit harder, because `par` is pure, but side-effecting the spark pool is, well, a side effect. We'll need to use something `unsafePerformIO`-like here... and the `unsafePerformIO` must be able to get at the per-procesor spark pool.

6. Thread-local state

See Adrian Hey's page `http://www.haskell.org/hawiki/GlobalMutableState`, and Ian Stark's ACIO message `http://www.haskell.org/pipermail/haskell-cafe/2004-November/007664.html`.

As Brian Hulley put it, at the moment, there is a strange unnatural discrepancy between the fixed set of built-in privileged operations such as Data.Unique.newUnique which are "allowed" to make use of global state, and user defined operations which have to rely on a shaky hack in order to preserve natural abstraction barriers between components such as a user-defined Unique, Atom, and anything involving memoisation or device management etc.

The kind of applications we have in mind (please add more) are:

- A source of random numbers, or of unique numbers. This should be on a per-thread basis.
- The value of 'stdin' or 'stdout'. We don't want to mutate this (although note that the handle itself ¡em¿contains¡/em¿ mutable state), but we might want to set the value for sub-computations, including any spawned threads.

6.1 A straw man

The basic idea is this:

- Each thread has access (via the IO or STM monad) to a "dictionary" that maps a (typed) key to a value of that type.
- The dictionary is not mutable, but the values might themselves be mutable cells (think of 'stdin'). More concretely:
  - A new built-in data type, `Key a`, an instance of `Eq`, but not `Ord`. It's a "thing with identity" (TWI), and called a "fluid" in Scheme [**?**].
  - A new built-in data type of dictionaries, ¡code¿Dict¡/code¿, which maps a typed key to a typed value:

    ```
    lookup :: Dict -> Key a -> Maybe a
    insert :: Dict -> Key a -> a -> Dict
    union  :: Dict -> Dict -> Dict
    etc
    ```

    Yes, it'll need a massive interface, like Data.Map.
- You can allocate a fresh (globally unique) key in the IO monad

  ```
  newKey :: IO (Key a)
  ```

- However, we add a new top-level declaration to allocate new top-level key:

  ```
  key rng :: Key StdGen
  ```

$$
\begin{array}{rrcll}
 & x, y & \in & \mathit{Variable} & \\
 & r, t & \in & \mathit{Name} & \\
 & c & \in & \mathit{Char} & \\
\\
\text{Value} & V & ::= & r \mid c \mid \backslash x \texttt{->} M & \\
 & & \mid & \texttt{return}\ M \mid M \texttt{>>=} N & \\
 & & \mid & \texttt{putChar}\ c \mid \texttt{getChar} & \\
 & & \mid & \texttt{throw}\ M \mid \texttt{catch}\ M\ N & \\
 & & \mid & \texttt{run}\ V\ M \mid \texttt{cutTo}\ V\ M & \\
\\
\text{Term} & M, N & ::= & x \mid V \mid M\ N \mid \cdots & \\
\\
\text{Heap} & \Theta & ::= & r \hookrightarrow H & \\
\text{Heap values} & H & ::= & M & \text{Mutable locations} \\
 & & \mid & \mathbb{E} & \text{Continuations} \\
\\
\text{Evaluation contexts} & \mathbb{E} & ::= & [\cdot] \mid \mathbb{E} \texttt{>>=} M \mid \texttt{catch}\ \mathbb{E}\ M & \\
\text{Action} & a & ::= & \texttt{!}c \mid \texttt{?}c \mid \epsilon &
\end{array}
$$

Figure 1. Syntax

This is useful independent of mutability. For example, GHC has lots of code going

```
thenIdKey = mkPreludeIdKey 3
returnIdName = mkPreludeIdKey 4
```

etc, where we hand out unique identifiers by hand. Easy to screw up.

- Each thread has an implicit, thread-specific dictionary:

  ```
  getDict :: IO Dict
  letDict :: Dict  -> IO a -> IO a
  ```

  (Think of myThreadId.) Notice that `letDict` does not mutate the dictionary; it just sets the implicit dictionary for a nested sub-computation.

  The thread-specific `Dict` can map a key to an `IORef` or `MVar`. That makes it possible for one thread to mutate the implicit state of another thread; or, in the case of thread-private mutable state, for a thread to mutate its own private state.
- A forked thread inherits its parent's dictionary.

6.2 Initialisation

Initialisation is the big question here. A library may want to allocate a key, and an initialiser to be run the first time the key is accessed, without the client of the library needing to know about it at all. A second issue that we may sometimes want to implicitly (?) re-initialise (or clear) all or part of the dictionary when forking a thread.

7. Semantics

Here's a start at an operational semantics.

References

[1] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, June 2005.

[2] C. Haynes and D. Friedman. Engines build process abstractions. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 1984.

[3] S. Marlow, S. Peyton Jones, A. Moran, and J. Reppy. Asynchronous exceptions in Haskell. In *ACM Conference on Programming*

I/O transitions $M; \Theta \xrightarrow{a} N; \Theta'$

$$\begin{array}{rcll}
\mathbb{E}[\texttt{putChar}\ c];\ \Theta & \xrightarrow{!c} & \mathbb{E}[\texttt{return}\ ()];\ \Theta & (PUTC) \\
\mathbb{E}[\texttt{getChar}];\ \Theta & \xrightarrow{?c} & \mathbb{E}[\texttt{return}\ c];\ \Theta & (GETC)
\end{array}$$

$$\begin{array}{rcll}
\mathbb{E}_1[\texttt{run}\ n\ t];\ \Theta[t \mapsto \mathbb{E}_2] & \rightarrow & \mathbb{E}_1[\texttt{running}\ n\ \mathbb{E}_2[()]];\ \Theta & (RUN) \\
\mathbb{E}[\texttt{stop}\ M];\ \Theta & \rightarrow & \texttt{return}\ M;\ \Theta & (STOP) \\
\mathbb{E}[\texttt{callCC}\ M];\ \Theta & \rightarrow & (M\ t\ \texttt{>>}\ \texttt{throw BadCallCC});\ \Theta[t \mapsto \mathbb{E}] \quad t \notin dom(\Theta) & (CALLCC) \\
\mathbb{E}_1[\texttt{cutTo}\ t\ M];\ \Theta[t \mapsto \mathbb{E}_2] & \rightarrow & \mathbb{E}_2[M];\ \Theta & (CUTTO) \\
\mathbb{E}[\texttt{running}\ 0\ M];\ \Theta & \rightarrow & \mathbb{E}[\texttt{return}\ (\texttt{More}\ t)];\ \Theta[t \mapsto (\texttt{return}\ [\cdot]\ \texttt{>>}\ M)] \quad t \notin dom(\Theta) & (PPEEMPT)
\end{array}$$

$$\frac{M \ \rightarrow \ N}{\mathbb{E}[M];\ \Theta \ \rightarrow \ \mathbb{E}[N];\ \Theta}\ (ADMIN)$$

$$\frac{n > 0 \quad M;\ \Theta \ \rightarrow \ N;\ \Theta'}{\mathbb{E}[\texttt{running}\ n\ M];\ \Theta \ \rightarrow \ \mathbb{E}[\texttt{running}\ (n-1)\ N];\ \Theta'}\ (RUNNING)$$

Administrative transitions $M \ \rightarrow \ N$

$$\begin{array}{rcll}
M & \rightarrow & V & \text{if } \mathcal{V}[\![M]\!] = V \text{ and } M \not\equiv V \quad (EVAL) \\
\texttt{return}\ N\ \texttt{>>=}\ M & \rightarrow & M\ N & (BIND) \\
\texttt{throw}\ N\ \texttt{>>=}\ M & \rightarrow & \texttt{throw}\ N & (THROW) \\
\texttt{catch}\ (\texttt{throw}\ M)\ N & \rightarrow & N\ M & (CATCH1) \\
\texttt{catch}\ (\texttt{return}\ M)\ N & \rightarrow & \texttt{return}\ M & (CATCH2)
\end{array}$$

Figure 2. Operational semantics of STM Haskell

Languages Design and Implementation (PLDI'01), pages 274–285, Snowbird, Utah, June 2001. ACM Press.

[4] S. Marlow, S. Peyton Jones, and W. Thaller. Extending the Haskell Foreign Function Interface with concurrency. In *Proc Haskell workshop, Snowbird, Utah*, pages 57–68, 2004.

[5] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308, St Petersburg Beach, Florida, Jan. 1996. ACM Press.