Concurrent and Multicore Haskell

Concurrent Haskell

- For responsive programs that multitask

- Plain old threads, with a few twists

- Popular programming model

A simple example

```
backgroundWrite path contents =
    done <- newEmptyMVar
    forkIO $ do
      writeFile path contents
      putMVar done ()
    return done
```


In spite of the possibly unfamiliar notational style, this is quite normal imperative code. Here it is in pseudo-Python:

```
def backgroundWrite(path, contents):
    done = newEmptyMVar()
    def mythread():
        writeFile(path, contents)
        putMVar(done, ())
    forkIO(mythread)
    return done
```

Imperative code!?

- Threads, assignment, "return"... *huh*?

- Haskell is a *multi-paradigm* language
  - Pure by default
  - Imperative when you need it

What's an MVar?

- An *atomic* variable
  - Either empty or full
    - takeMVar blocks if empty
    - putMVar blocks if full

- Nice building block for mutual exclusion

See Control.Concurrent.MVar for the type.

Coding with MVars

- Higher-order programming
  - modifyMVar: atomic modification
  - Safe critical sections

- Combine MVars into a list
  - FIFO message channels

The modifyMVar function extracts a value from an MVar, passes it to a block of code that modifies it (or completely replaces it), then puts the modified value back in.

If you like, you can use MVars to construct more traditional-looking synchronisation primitives like mutexes and semaphores. I don't think anyone does this in practice.

FIFO channels (Chan)

- Writer does not block
- Reader blocks if channel is empty

- Duplicate a channel
  - Broadcast to multiple threads

See Control.Concurrent.Chan for the type. A Chan is just a linked list of MVars.

Smokin' performance

From the "Computer Language Benchmark Game"

- Create 503 threads
- Circulate token in a ring
- Iterate 10 million times

Language	Seconds
GHC	6.70
Erlang	7.49
Scala	53.35
C / NPTL	56.74
Ruby	1890.92

Runtime

- GHC threads are incredibly cheap
  - Run millions at a time
- File and network APIs are blocking
  - Simple mental model
  - Async I/O underneath

Time for a change

- That didn't rewire my brain at all!

- Where's the crazy stuff?

Purity and parallelism

Concurrent vs parallel

- Concurrency
  - Do many unrelated things “at once”
    - Goals are *responsiveness* and *multitasking*

- Parallelism
  - Get a faster answer with multiple CPUs

Pure laziness

- Haskell is not just *functional* (aka *pure*)
- It's *non-strict*: work is deferred until needed
  - Implemented via *lazy evaluation*
- Can laziness and parallelism mix?

If we're deferring all of our work until the last possible moment, how can we specify that any of this evaluation should occur in parallel?

Laziness is the *default*

- What if something must happen *right now*?
  - Use a special combinator
    - *seq* – adds strictness
  - Evaluates its 1st argument, returns its 2nd

A simple use of *seq*

```
daxpy k xs ys = zipWith f xs ys
  where f x y = k * x + y
```

```
daxpy' k xs ys = zipWith f xs ys
  where f x y = let a = k * x + y
                in  a `seq` a
```


The daxpy routine is taken from the venerable Linpack suite of linear algebra routines. Jack Dongarra wrote the Fortran version of this function in 1978. Needless to say, it's a bit longer.

The routine scales one vector by a constant, and adds it to a second. In this case, we're using lists to represent the vectors (purely for convenience).

The first version of the function returns a list of thunks. A thunk is an unevaluated expression, and for simple numeric computations it's fairly expensive and pointless: each element of the list contains an unevaluated "k * x + y" for some x and y.

The second version returns a list of fully evaluated numbers.

par

- "Sparks" its first argument
  - Sparked evaluation occurs *in parallel*
- Returns its second

The par combinator does not promise to evaluate its first argument in parallel, but in practice this is what occurs.

Why not bake this behaviour into its contract? Because that would remove freedom from the implementor. A compiler or runtime might notice that in fact a particular use of par would be better represented as seq.

Our favourite whipping boy

```
pfib n | n <= 1 = 1
```

```
pfib n =
  a `par` (b `pseq` (a + b + 1))
 where a = pfib (n-1)
       b = pfib (n-2)
```


The pseq combinator behaves almost identically to seq.

Parallel strategies

- *par* might be cute, but it's *fiddly*
  - Manual annotations are a pain

- Time for a Haskell hacker's favourite hobby:
  - Abstraction!

Algorithm + evaluation

- What's a *strategy*?
- How to evaluate an expression
- Result is in a *normal form*

Head normal form

- "What is my value?"

- Completely evaluates an expression

- Similar to traditional languages

Weak head normal form

- "What is my *constructor*?"

```
data Maybe a = Nothing
             | Just a
```

- Does not give us a complete value
  - *Only* what constructor it was built with

The elements that I've marked in green are the constructors (properly, the "value constructors") for the Maybe type.

When we evaluate a Maybe expression to WHNF, we can tell that it was constructed using Nothing or Just. If it was constructed with Just, the value inside is not necessarily in a normal form: WHNF only reduces ("evaluates") until the outermost constructor is known.

Combining strategies

- A strategy is a normal Haskell function
- Want to apply some strategy in parallel across an entire list?

```
parList strat []     = ()
parList strat (x:xs) =
  strat x `par` parList strat xs
```


We process the spine of the list in parallel, and use the strat parameter to determine how we'll evaluate each element in the list.

Strategies at work

- Map a function over a list in parallel
  - Pluggable evaluation strategy per element

```
using x strat = strat x `seq` x

parMap strat f xs =
  map f xs `using` parList strat
```


Notice the separation in the body of parMap: we have normal Haskell code on the left of the using combinator, and the evaluation strategy for it on the right. The code on the left knows nothing about parallelism, par, or seq.

Meanwhile, the evaluation strategy is pluggable: we can provide whatever one suits our current needs, even at runtime.

True or false?

- Inherent parallelism will save us!

- Functional programs have *oodles*!

- All we need to do is exploit it!

Limit studies

- Gives a maximum *theoretical* benefit
  - Model a resource, predict effect of changing it
- Years of use in CPU & compiler design
- Early days for functional languages

So ... true or false?

- Is there lots of "free" parallelism?
  - Very doubtful
- Why? A familiar plague
  - Data dependencies
- Code not *written* to be parallel *isn't*

Two useful early-but-also-recent papers:

"Feedback directed implicit parallelism", by Harris and Singh

"Limits to implicit parallelism in functional application", by DeTreville

Current research

- Feedback-directed implicit parallelism
  - Automated *par* annotations
  - Tuned via profiled execution

- Results to date are fair
  - Up to 2x speedups in some cases

This is the work described in the Harris and Singh paper.

Parallelism is hard

- Embarrassingly parallel: not so bad
  - Hadoop, image convolution
- Regular, but squirrelly: pretty tough
  - Marching cube isosurface interpolation, FFT
- Irregular or nested: really nasty
  - FEM crack propagation, coupled climate models

Current state of the art

- Most parallelism added by hand
  - Manual coordination & data layout
    - MPI is akin to assembly language

- Difficult to use, even harder to tune
- Irregular data is especially problematic

Nested data parallelism

- Parallel functions invoke other parallel code
- One SIMD "thread of control"
- Friendly programming model

This project is known as "Data Parallel Haskell", but is sometimes acronymised as "NDP" (Nested Data Parallelism) or "NPH" (Nested Parallel Haskell). Confusing, eh?

NPH automation

- Compiler transforms code and data
- Irregular, nested data becomes flat, regular
- Complexity hidden from the programmer

Current status

- Work in progress
- Exciting work, lots of potential
  - Attack *both* performance and usability
- Haskell's purity is a critical factor

Fixing threaded programming

Concurrency is hard

- Race conditions
- Data corruption
- Deadlock

Transactional memory

- Fairly new as a practical programming tool
- Implemented for several languages
  - Typically comes with weird quirks
- Haskell's implementation is *beautiful*

Atomic execution

- Either an entire block succeeds, or it all fails
- Failed transactions retry automatically
- Type system forbids non-atomic actions
  - No file or network access

How does retry occur?

- When to wake a thread and retry a transaction?

- No programmer input needed

- Runtime tracks variables read by a failed transaction, retries *automatically*

Composability

- All transactions are *flat*

- Calling transactional code from the current transaction is normal

- This simply extends the current transaction

Early abort

- The *retry* action manually aborts a transaction early
- It will still automatically retry
- Handy if we know the transaction must fail

Choosing an alternative

- The *orElse* action combines two transactions
  - If the first succeeds, both succeed
    - Otherwise, it tries the second
    - If the second succeeds, both succeed
  - If both fail, the first will be retried

STM and IPC

- TVar – simple shared variable
- TMVar – atomic variable (like an MVar)
- TChan – FIFO channel
- If the enclosing transaction retries...

  ...then so does any modification

A useful analogy

- Concurrency
  - Mutexes, semaphores, condition variables
  - Software transactional memory
- Memory management
  - malloc, free, manual refcounting
  - Garbage collection

The analogy between garbage collection and STM is, as far as I know, due to Dan Grossman. He was at least the first to publish it in academic circles.

Manual / auto tradeoffs

- Memory management
  - Performance, footprint
  - Safety against memory leaks, corruption
- Concurrency
  - Fine tuning for high contention
  - Safety against deadlocks, corruption

Brief recap

- Concurrency
  - Fast, cheap threads
  - Blocking I/O and STM are *friendly to your brain*
- Multicore parallelism
  - Explicit control or a strategic approach
  - NPH offers an exciting future