Standard Chartered

Haskell in the Large

Don Stewart

Google Tech Talk / 2015-01-27

sc.com | Here for good

Haskell in the Large

Themes and Context

Standard Chartered

Themes

- How language tools can help you make better software
- Type systems for:
  - Maintainable software
  - Composable/modular /reusable software
  - And, ultimately, cheaper software.
- Structures and approaches that are useful in “enterprisey” software to:
  - Reduce latency
  - Make code faster
  - Make code safer /more robust

Haskell at Standard Chartered

- Total Haskell code base: >2M lines, >150 contributors (~20 core), >110k commits.

- Two Haskell compilers: off-the-shelf GHC and our in-house “Mu” compiler.

- Haskell code everywhere: real time pricing, risk management, data analysis, regulatory systems, desktop and web GUIs, web services, algo pricing, end-user scripting, and as plugins to C++, Java, C#, Excel.
- Windows, Linux, clusters and grids. On servers and desktops in >30 countries.

- All due to the work of the Standard Chartered MAG and Strats teams: Marten Agren, Neville Dwyer, Lennart Augustsson, Stuart Croy, Raphael Montelatici, Neil Mitchell, Huu-Hai Nguyen, Ravi Nanavati, Malcolm Wallace, Roman Leshchinskiy, Gergo Erdi, Maciej Wos, Chris Kuklewicz, George Giorgidze, Matthias Gorgens, Andy Adams-Moran and Peter Divianszky and more.

Haskell at Standard Chartered

- This talk is about the day to day experience of using Haskell to write larger systems

  - Building bigger systems from smaller pieces
  - Reusable approaches for designing systems
  - Techniques for making code maintainable
  - And ways to improve code that you come across

Using examples where we have

  - Integrated Haskell into existing systems
  - Used Haskell to wrap untrusted or external systems

Theme of the talk: using the language to reduce and manage software complexity.

Strats @ Standard Chartered

One team among several using Haskell, within a larger organization using Java, C#, C++

Strats are a small team of software engineers in a very large organization.

But with an unusual mandate:

1. Fast delivery
   - Move fast: deliver in hours/days/weeks what used to take days/weeks/years.
   - Equivalent functionality at 5 to >10 times sooner and/or cheaper.
   - Ship early, polish continuously. (Perfection never delivers).
2. Robust systems
   - Libraries, apps, services should just work. Even with poor latency and bandwidth; and little to no user training; when underlying systems fail, and when data formats change or break.
   - Build systems the users trust.
3. Automate everything
   - Few resources for manual tasks, so automate everything. Integration; releases; monitoring.
   - Turn your company into an API.

Strats @ Standard Chartered

4. Invent new things.
   - Move fast so we can get to build things that weren’t even seen as feasible
   - What can we build if we have good typed APIs to the world?

5. Light weight specifications.
   - Often working from verbal, ad hoc evolving spec. And at best from informal functional specs.
   - Spec often refined/derived based on the Haskell implementation.
     - “No plan survives contact with implementation”

6. Low maintenance requirement
   - Few support /devops around, but a lot of code; so make sure the systems support themselves.
   - Rely on types, types, types to make future changes easier

7. Need high success rate
   - Have to demonstrably out-perform. Be clearly more efficient. Reproducible project success.
   - Don’t think like/be the incumbent.
     - “Work as if you live in the early days of a better nation.”

Strats Toolkit – Haskell + Cortex

- Writing *lots* of apps; services; libraries in Haskell.
- Built on top of the large “Cortex” library (where most of the C++ and Haskell lives)
  - Kind of an “enterprise Prelude and Base library”
  - Plus all the quant models and analytics
  - And APIs to the bank’s systems
- In-house Haskell compiler: \`Mu’. One of Lennart’s many Haskell compilers.
- Has a few language and runtime customizations:
  - Whole world optimizing compiler.
  - Largely strict(-ish) evaluation strategy (n.b. devs have very mixed view on this).
  - Compiles to portable bytecode – same blob runs on Linux or Windows.
  - Values, closures, functions can be serialized.
  - Primitive String type is a unicode text blob. No [Char].
  - \`Any\` type. For the dynamically typed boundary interfaces.

Things we use – Haskell + Cortex

- Usual things:
  - QuickCheck
  - fclabels data structure lenses
  - Attoparsec; blaze; bytestring; cairo; vector; HaXml; geniplate; etc etc.

- New things:
  - GUIs: Generate native Windows WPF GUI or JavaScript app from single Haskell impl.
  - Data mining: In-language relational algebra data structures and API.
  - Memoization: In-language data flow graph library with memoization support.
  - Lens compiler. Derive data structure editor APIs for free.
  - Bindings: Tons of bindings to external services. Haskell as API glue.
  - Time and date: very good calendar; timezone; datetime handling

    Common joke: “all bugs are calendar bugs”.

Things we use – Haskell + Cortex

- Useful things:
  - In-house haddock docs and type-based API search for corporate APIs
  - Ubiquitous lightweight, asynchronous logging.
  - Continuous, automated merge/test/integration/deploy infrastructure
  - Cross platform, distributed job scheduling. Like a dependency graph multi-host `cron`.

- Surprising things:
  - Heavy use of process isolation. Baby Erlang in Haskell. Assume the quant library; external service, whatever will fail.
    - Wrap it up in a process.
    - Monitor that process (pipes or zeromq)
    - Log, restart, kill as necessary
  - Process for parallelism. (Data) parallelism via process `map`.
    - Same code scales from local threads to local processes, to remote, 1000+ node grid jobs
    - Parallel "strategies" and monadic workflows for parallelism.
  - Native relational algebra data type.
    - An awful lot of data mining and analysis is best done with relational algebra.
    - Not just in the database.

Things we use – Haskell + Cortex

- By 2015, Haskell+Cortex has become a significant platform for lowering the cost of implementing new systems in-house
- More and more of the enterprise available through common API
- Team members now all with multiple project success under their belt
- Old code doesn't fall apart – large, (very) strongly typed code bases don't rot.

Each new project getting easier and cheaper to deliver.

So... How did we get here?

Haskell in the Large

Why Haskell?

Standard Chartered

Why Haskell? Modularity and compositionality

Large systems are much, much harder to build than small ones

- Complexity of the system begins to dominate engineering costs
- Maintenance costs more over time as knowledge of the system is forgotten
- No individual remembers how all the things fit together
- Developers come and go. Sometimes people take holidays!
- Specification changes are inevitable, frequent

Control software complexity and you can move faster, and do more things.

It’s all about building software quickly **and** correctly. And planning for change.

Modularity

Any steps to restrict or reduce the ways components can interact will help you

- Plan for future developers who won’t know the code at all
- Won’t have time to do whole-program transformations to fix things

Unnoticed interactions between components will bite. They will slow you down.

So constantly look for tools to mechanically support modularity of interactions between components.

Why Haskell? Reducing accidental complexity

The language is one of the best tools to manage software complexity.

Use the machine to:

- Catch mistakes earlier
- Enforce design assumptions / invariants
- Document the design for later developers
- Limit unintended interactions between components
- Make wrong code impossible to write
  - “Make illegal states unrepresentable” (in your logic) – Jane Street
- Make it easier to change pieces in isolation.

Programming languages have many tools to support these goals.

Better code through types

Strong, expressive types give us machine-checkable, modular software

- APIs become self-describing. Lego blocks that fit together.
- Not just documentation!
- Minimize secret/magic behaviour that isn’t captured in the type

Result:

- Any dev can build systems without worrying about implementation internals.
    - Just fit types together.
- Bad devs have a harder time breaking things. The compiler won’t let them cheat!

Types pay off the most on large systems. Architectural requirements captured formally.

Capture the design in machine-checkable form, in the code.

One big step on the path to formal methods.

Not your usual type system: newtype

Large systems are built from lots of smaller systems in conversation.
How does the language help us specify valid communication channels?

- Obvious first step: don't use String and Double for all the things

```
f :: Double -> Double -> String -> Double
```

The type tells me basically nothing about what values are valid and what are illegal.
"Stringly-typed" programming. Erasure of information that would help the programmer.

Do I pass percent values? Rates? Volatilities? Who knows?
So name the things:

```
f :: Rate Libor -> Spot SGD -> Date -> Rate SIBOR
```

Standard Chartered

Not your usual type system: newtype

Use `newtype` and `data` to distinguish unique entities in the system.

Layer the semantics of the system over the underlying representation.

Representation is only one thing we might say about a value's meaning.

```
newtype Tenor = Tenor String
  → Tenor "1M"

newtype NonNegative a = NonNegative Double
  → 3.14159 :: NonNegative D5

newtype Spot
newtype Forward
newtype Bid
newtype Ask
newtype Today; newtype PrevDay
```

Make it impossible to mix up or combine values in nonsense ways.

Not your usual type system: phantom types

Same concept (e.g. Rate, Spot, Forward, Bid, Ask) can appear in many circumstances.

Need a generic way to tag values with type-level information: phantom types

- Augments types with origin/security/other metadata.
- Makes it possible to prove security properties, information flow properties
- Very lightweight, but high power/expressiveness.
- First steps down the road to GADTs and type level programs.

Examples:

```
data Ratio n = Ratio Double
1.234 :: Ratio D3

data Ask ccy = Ask Double
Ask 1.5123 :: Ask GBP
```

Massively reduces the number of intended paths values may take in the system.

Not your usual type system: Boolean blindness

As a dual to how String and Double types have too many valid values for most use cases, `Bool` often has too little information.

This has been coined “Boolean blindness” by Harper.

A Boolean value on its own tells me nothing about what it is that is true or false.

What proposition was proved? Makes it too easy to dispatch in the wrong direction, or pass the truthiness of one proposition to some code that needed to know something else.

Phantom types can help eliminate this. Compare:

```
authenticate :: String -> String -> IO Bool
```

With

```
authenticate :: Priv p => User -> Passwd -> IO (AuthUser p)
data Editor
data Owner
instance Priv Editor ; instance Priv Owner
```

Type tags to improve code

A common technique when taking over someone's code is to start introducing newtypes for each distinct value you come across. Name the concepts as you go.

- Test your assumptions about what values are distinct semantic entities
- Teach the compiler at the same time.
- Type errors reveal where the data flows in the program
- Domain/specification-level concepts become embedded in the code
- Invariants in the specification are now machine checked
- Fewer incorrect programs can be written.
- Makes refactoring much, much safer.

Once the objects of the system are tagged and bagged, you can start separating distinct instances/sessions/capabilities with phantom types.

- Distinct user sessions, tainted data, data that is generic in some way.
- Tag values that are produced by one class of user from another.

Standard Chartered

Haskell in the Large

Polymorphism

Standard Chartered

Polymorphic code and data

Code reuse increases our leverage as a team. The less any code depends on any particular data representation, the more reusable it is.

Compiler can enforce this with parametric polymorphism.

Classic example: sequency things:

```
map   :: (a -> b) -> [a] -> [b]
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Or generalize to traversals over many different structures - a bounded set of types:

```
fmap  :: Functor f => (a -> b) -> f a -> f b
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

Name a concept once. Implement it once. Reuse the code forever.

More abstract == more reuse

Evaluate statements in any “language-like” structure:

```
    sequence :: Monad m => [m a] -> m [a]
```

Combine two things, generally:

```
    mplus :: MonadPlus => m a -> m a -> m a
```

Choose between two alternatives, generally:

```
    (<|>) :: Alternative => f a -> f a -> f a
```

And you end up with libraries of the fundamental patterns for combining and composing code – truly generically. Type prevents any particular implementation details leaking.

Learn the patterns once, reuse them again and again. Helps minimize app-specific logic.

More abstract / more symbols == more obscure?

Abstraction can have downsides though:

```
    const (fmap (fmap (fmap (fmap (toAny t)))) . f a . fromAny t)
```

Traversing at least 4 nested structures at different types. Hard to refactor later.

But at least it is safe to refactor. Keep adding type information until you understand the code!

Needs some judgement about when to generalize, and when to be specific.

Become can go overboard on operators too. (Arguably a domain-expert may like it though):

```
    <-> (visibility_ warning (\s -> ctrl (s @@ color)))
             <-> (grid -# details)
            <|||> visibility def fac refillBox
              <-> visibility def selectedLeg periodsBox
            <---> editor (used -# legs) bls
              <|> deleteAct @@ icon ListRemove @@ name "Delete"
    ) @@@ backColor sty @@ padding 5
```

Haskell in the Large

Algebras, interpreters and compilers

Standard Chartered

Algebraic data types

Newtype and Phantom types are cheap, easy ways to teach the compiler new facts about a value. To reduce the places a value can migrate to unintentionally.

But we also want to capture deeper rules about the system. Capture the API as an *algebra.*

Specify the set of valid entities in your system, and the operations that may be performed.

No more, no less. No extra, unintended semantic baggage. And maximum modularity and compositionality.

From simple to complex data types:

- As simple record types – statements of schemas.
- To capture state transitions
- To capture the APIs of an external systems
- ADTs for embedded languages

Classic ADTs for “mini DSLs”

We want to permit only well-formed JSON to be constructed.

So we model the JSON grammar as a (recursive) data type.

```
data JSValue
    = JSNull
    | JSBool Bool
    | JSDouble Double
    | JSString String
    | JSRecord [(String, JSValue)]
    | JSArray [JSValue]
```

Now the JSON type system is embedded in the Haskell type system. It will be a type error to construct ill-formed JSON.

Very common technique to capture semantics separately to syntax/API.

Standard Chartered

Classic ADTs for “mini DSLs”

An external system may have a very rich API. Rather than open up the whole thing, start by adding each primitive as a new variant in ADT.

e.g. permit only well-formed an external key/value store:

```
data Query
    = MetadataKey Is Key
    | MetadataValue Is Value
    | Size Is Integer [Integer]
    | SizeRange Is Integer Integer
    | IngestTime Is Datetime
    | Custom Is String

 data Is = Is | IsNot
```

And then write an interpreter to evaluate the query, or a compiler to translate the query into e.g. SQL or JavaScript.

Compiling a query

Interfaces between Haskell and the external system are mediated through interpreters and compilers.

Translating the precise representation of the system as a Haskell data type into the foreign/external representation.

The point at which we erase our controlled, precise representation:

```
compile :: [Query] -> QueryString
compile [] = "*:*"
compile qs = String.intercalate " " (map compileQuery qs)

 compileQuery :: Query -> String
 compileQuery (MetadataKey is k) =
   implode [compileIs is, "(", "customMetadataContent:" , "\"k.", escape k , "\"", ")"]
compileQuery (MetadataValue is v) =
   implode [compileIs is, "(", "customMetadataContent:" , "\"v.", escape v , "\"", ")"]C
compileQuery (Size is s ss) =
   implode [compileIs is, "(", "size:" , "("
       , String.intercalate " " (map compileInteger (s:ss)) , ")" , ")"
       ]
...
```

Total Functional Programming

Pattern matching guides implementation. Functions are built via induction on the structure of the ADT.

Needs proper pattern matching for best results.

```
jsonToAny :: JSValue -> Any
jsonToAny JSNull          = toAny ()
jsonToAny (JSBool b)      = toAny b
jsonToAny (JSDouble d)    = toAny d
jsonToAny (JSString s)    = toAny s
jsonToAny (JSRecord ljs) = toAny ...
```

Enable exhaustive pattern match warnings and you are on the road to pervasive total functions. Avoid exceptions for control flow.

Lift errors into types (Maybe, Either) to make functions more self-describing and modular.

Property-based testing of ADT-based APIs

APIs and libraries written in this way become easier to test.

- Use QuickCheck to generate and interpret random expressions in the algebra
- Test for predicates that hold over the results
- Identify laws that holds, develop new API invariants that should hold.

You can *discover* better APIs this way.

Guided by the ease of stating laws that hold over the API.

- Cheap verification of external systems.
- Bug-report generator for your colleague's services ☺

Reusable design approach

Big systems are built from smaller systems. Small systems are captured by ADTs and functions (domain specific languages).

Interfaces are mediated by “compilers and interpreters” -- functions that map high level types to different representations.

You could be building systems this way. Large systems are libraries of DSLs. E.g:

- Embedded SQL and other query languages
- Plotting and graphics visualization APIs
- External data representations (JSON, XML)
- Domain-specific data (FpML etc)
- GUI and graphics APIs (WPF, X11)
- Job control and orchestration (workflow APIs)
- CRUD apps (state transitions between domain-specific data types).

The “Category Theory for Software Architects” view of software design.

Haskell in the Large

Complexity and Side Effects

Standard Chartered

Of IO and complexity

Modularity through types means capturing semantics at the type level.

Side-effects, like IO and network actions, are fundamentally anti-modular.

- You can’t re-order these actions at will
- Can’t (easily) evaluate them at compile time...
- It is hard to refactor/optimize them to equivalent expressions
- Hard to apply typical parallel strategies to improve performance.
- Assumptions about ordering/resource usage may be invisible

So we try to tag effects in the type to keep them separate.

```
    IO          -- arbitrary external effects. Few rules.
    SafeIO      -- read-only effects. Safer.
    ST          -- memory-only.
```

etc

Of IO and complexity

Effects are vital, but so powerful they need to be managed. Otherwise side channels proliferate

Typical patterns:

- IO at initialization time. Gather resources into an environment
- Core logic is pure data transformations using read-only environment
- Evaluate result by interpreting an ADT for its effects on the world

Huge benefits from capturing IO at the type level:

- More aggressive compiler optimizations
- Actions become first class (makes exotic control structures easier)
- Programmer knows more about what unknown code can and cannot do.

But ultimately its about APIs with no surprises. Reduces system complexity.

Haskell in the Large

Making thing fast

Standard Chartered

Techniques for scaling

Many issues arise at scale.

- Network latency
- Network bandwidth
- Memory hierarchy and tiers of service. From:
  - Data/compute source to local servers
  - To local users
  - To “nearby” users
  - To remote users networks
  - To remote user disks
  - To remote memory
  - To remote threads
- Need libraries of control flow and resource management to tier and distribute data and compute power.

Some small steps we take

Memoization:

- In-memory, local-machine and local node transparent query caching of dictionary key/value queries
- A lot of data changes rarely (not even daily) so fan out and distribute to the leaves

Migrating compute tasks

- Users generate closures to run. Functions applied to data.
- Work out where best to run them:
  - Local thread (s)
  - Local processes?
  - Nearby grid
  - Central/large data centers?
- Needs plenty of application-specific knowledge.
- Mu compiler customizations for mobile code: serialize any closure

Process isolation

When integrating many different components (in many languages) resource bugs (out of mem) and control flow bugs (loops, segfaults) are inevitable

Restrict their impact with process isolation.

- Needs some metrics for what size of task should be isolated
- And forking should be fairly cheap.
- Needs fast closure serialization

But opens up a world of workflow management APIs.

- Retry tasks on failure
- Speculatively execute (pure) tasks
- Automatically migrate tasks to better systems ...

Erlang has valuable lessons to borrow/steal.

Haskell in the Large

Caveats and limitations

Standard Chartered

Limitations

You need infrastructure in place to get many of these benefits

- ADTs are painful to describe without recursive data types, sums and products.
  - Declaring new data structures should be very cheap!
  - Declaring new functions should be very, very cheap!
- Concurrency and speculative evaluation can be infeasible if threads are too expensive
- Migrating work by sending closures requires runtime support
- Type-level tagging and control needs at least a 1980s-era type system
- Critical mass of experienced FP developers

And, even if the language has the tools, **developers need taste and guidance** to use them.

You can write Java in any language:

```
data T = T {
        cvxM    :: IORef Double
      , cvxMR   :: IORef Double
      , stream1 :: Dict String [(String,IORef Double)]
     ...
   }
```

Standard Chartered

Conclusions

- Look for machine-supported ways to minimize complexity
  - To improve task time estimations and cost
  - Reduce delivery times
  - Make maintenance far simpler
  - And lead to a platform of reusable, modular components

- Benefits become obvious at scale and under time pressure

- Your choice of programming language can have a real effect on these results over time.

Standard Chartered