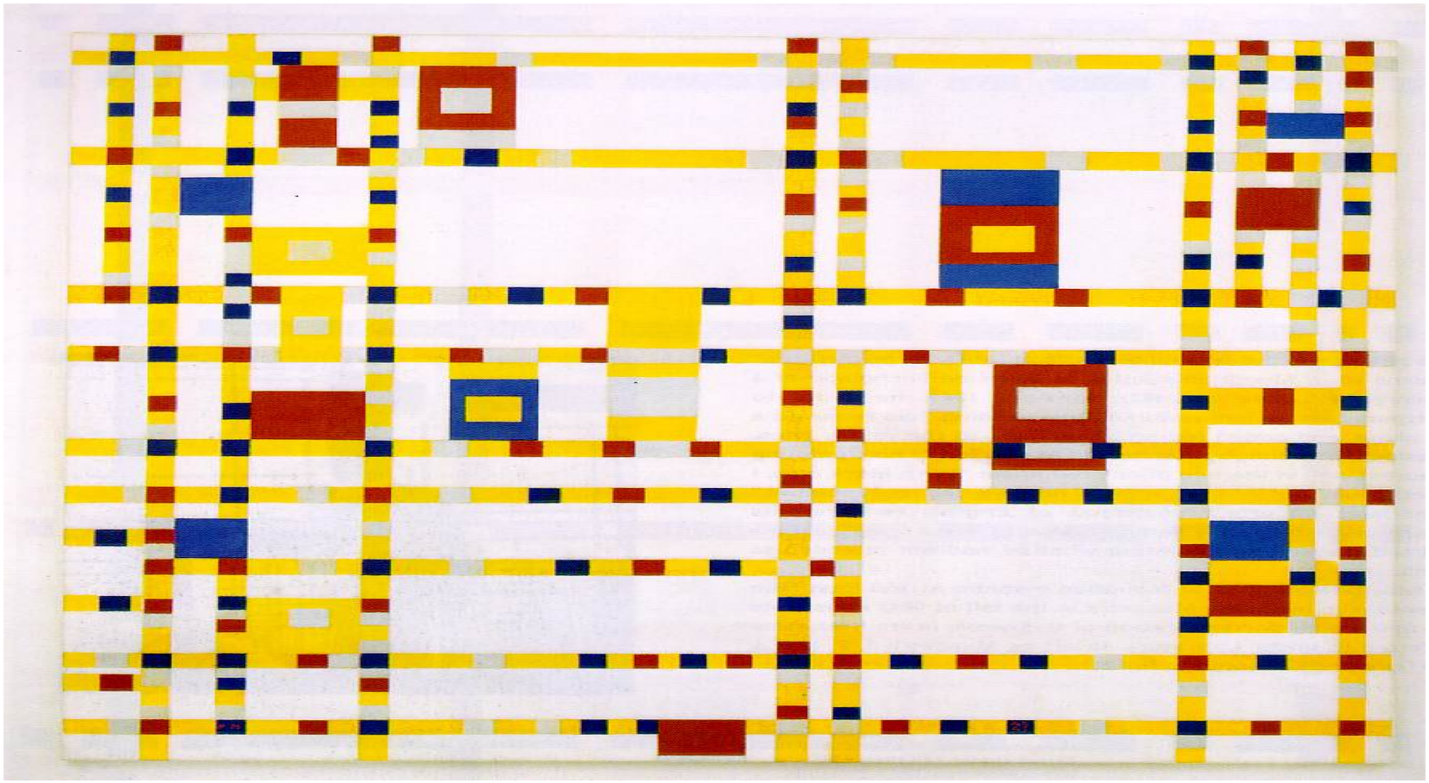
Make Things Now!

Pragmatic FP with Haskell

Don Stewart
Standard Chartered Bank



PADL - 2012 01 24 - Philadelphia, USA

What's this talk about?

Notes on trying to build better software using

- smart teams,
- good engineering processes, and
- typed functional programming

Context for this talk

Open Source Haskell

- xmonad – a cult window manager written in Haskell
- bytestring – core high perf. Haskell byte arrays library

Closed Source Haskell

- Financial pricing and risk tools for wide range of asset classes and financial products at SCB
- Used by traders across the globe, acround the clock
- Significant time pressure – clear value to being more efficient

Really big picture: why software?

- Increase economic productivity by automating everything
- Make the complexity of the world tractable by making it hackable
- Impose structure through typed data
- Transform work/jobs/tasks into resuable/composable functions
- Find new efficiencies. Repeat

(Demonstrate you can do this and you'll never be out of work)

Good software projects

Are faster

- Get new software to your users sooner than the competition

Are leaner

- Make software of equivalent functionality, but with less effort/cost

Are more reliable

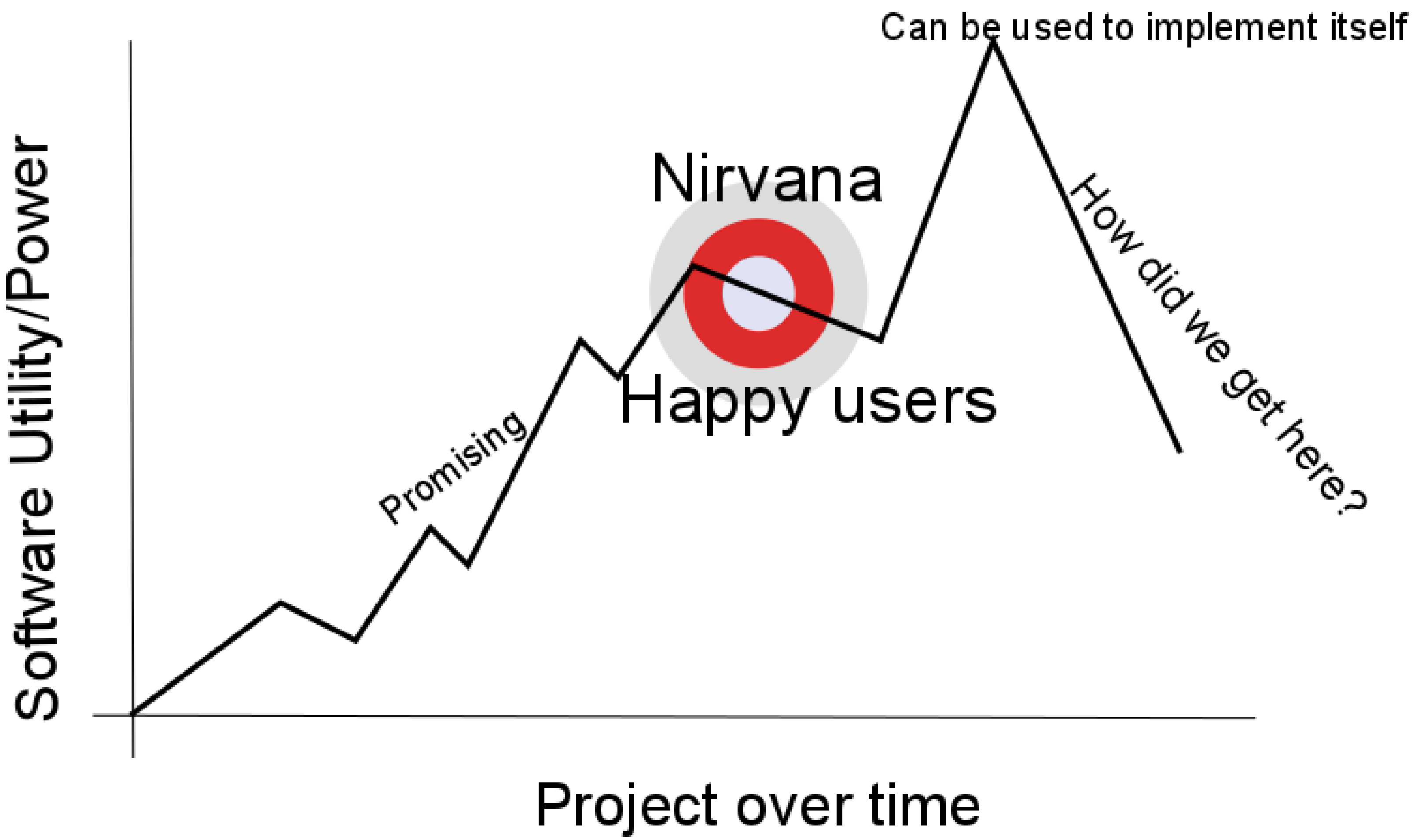
- Software that is more robust and easier to maintain
- That users actually trust to work

Are fun

- Software that your users love and recommend to others

Project goals

Software that is just the right amount of useful



Doing it right

- Get to usefulness Nirvana faster
- Try to stay there longer
- Know when to stop

How?

- Use the computer to minimize the amount of work developers need to do
- By minimizing the complexity of the system

And

- Bring the users along with you on the project development

3 Themes

Three sides to the triangle

- Technology – monads, types, code generation
- Engineering – release structure, testing, delivery
- People – users, marketing, passion

Making useful things means working in all three areas

Theme 1: technology
Better code through typed functional programming

Language tools – types, purity, abstraction, composability, semantics – exist to help us manage complexity.

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

So use the machine to

- Catch mistakes earlier
- Limit unintended interactions between components
- Reduce the ways code can go wrong
- Minimize the possibility of human error

This is what programming languages are for!

Theme 2: engineering
Better software through processes

- Ship early, ship often – maximise feedback, and “revise, revise, revise”
- Defend against all the errors you and your team will make – test, prove, derive
- Automate releases, automate building, automate installing

Theme 3: people and communities
Teams and structures to produce better code

- Get others involved, communities of passionate, advocate/users
- Expertise takes 10,000 hours [Outliers]. You have to do the work
- Care deeply about quality – do you actually prefer to use the apps you write?

Theme 3: people
Teams and structures to produce better code

- Be passionate – do you wake up thinking about the project? Are you deeply uncomfortable with known bugs in your software? Do you show attention to detail – are you often disatisified with your code, and feel compelled to improve it?
- Communication – work in the open to build trust, and keep the team functioning
- **Will** – you still have to actually execute on an idea, and see it through to user acceptance. To finish things – which will also mean doing marketing and documentation.

Haskell in 2012

Large, broad ecosystem

- 1000s of libraries and tools, in every domain
- Projects with 100k, 500k, 1M lines of code

Big teams

- Projects with 10, 50, 100+ developers over their lifetime
- Experienced devs: many now with 5, 10, 15 years with the language

Reliable

- Used to build critical tools used by 100s or 1000s of users every day
- Software they rely on to do their jobs

Haskell as a technology

An unsurpassed technology for building software:

- Quickly
- Safely

And with the performance to match

- Rich toolchain
- Libraries for everything

Technology, Engineering and People

Technology : Types

Large, complex systems are built from thousands of smaller pieces.

Those pieces need to communicate.

- Algebraic data types are the exact tool for precisely specifying the possible data exchanged between components.
- No ambiguity, full machine checking, flexible.
- Obvious: Don't encode things in strings, or dynamic types, or XML, when you can define an ADT to do the job exactly.

Example : Lenses for GUIs

Lenses: introduced by Pierce et al. Bi-directional, composable data access and modification

- Allow for easy editing of (nested!) data structures
- Keep views/edits on data structures in sync
- Can be generated from types

Many routine/boring tasks done by humans boil down to editing structured data

The perfect, minimal, typed bridge connecting the view (GUI) and the model data structure

Example : Typed GUIs

Beyond lenses, we can automate much of the work of constructing graphical interfaces for data structure editors

- Type-driven GUI component derivation
- Dispatch to formatting/editing/component based on type class instances
- \`newtype\` to pick instances – Money, Currency, Ratio, types for decimal precision

Example : Configuration

Complex software has a complex configuration space

- Many variables, parameters

How best to represent the desired user configuration?

- Ad hoc configuration files?
- Text files?
- Machine checked, structured data types!

Type checked, maintainable config files.

Technology : Polymorphism

Parametric polymorphism perfectly captures a notion of reuse of software

- The type clearly specifies how an algorithm is abstracted from the data it operates on
- Machine-checkable abstraction contract that can't be violated
- Reduces unintended complexity

Haskell code bases tend to build up significant amounts of reusable, generic code – many have a 'Utils' folder that travels with them

Bounded Polymorphism

Components fit together like jigsaw pieces

Bounded polymorphism (e.g. typeclasses) captures sets of components with similar behavior

- Ensures pieces can only be composed in exactly the ways that are sensible
- Statically prevents unexpected/unhandled interactions between systems
- Allows for interactive programming by-the-types (aka “Tab/Return programming”).

Example : window system layout

Many different algorithms exist for laying out windows on the screen

- Assigning rectangle geometries to client windows
- Hard to test – need to run a windowing system?

Make the algorithms polymorphic in the window type. Test geometries separately to any live system.

Allows for machine-checked separation of layout algorithms, and the underlying toolkits they drive

Example : plotting

Plot and data visualization is very common

- Language communities should think about supporting data visualization better
- What types of data can be visualized?
- Different viz/plot methods have different parameters
  - Bar charts might have solid fill
  - Area spots might have opacity and radius ratio

Use type classes to parameterize plots by their X and Y types, and the attributes each plot can accept – add some statically checking to data visualization.

Technology : parallelism

Multiple cores everywhere

- 16/24-core desktops relatively common
- Computationally expensive routines
- Large data sets

Parallel strategies + array/structure folds/maps remarkably effective.

Local code transformation for sometimes large payoff

Needs only a few good combinators – parmap

Technology : concurrency

What's so hard about concurrency?

- forkIO -based thread launching
- MVar / Chan for (typed) synchronization

Useful for

- Hiding latency
- Shuffling heavy work to the background

Event-driven UIs relatively easy to build – but can lead to non-local/spaghetti interactions. Be wary

Technology : persistance

Libraries for serialization all data structures

- Makes it easy to reproduce the state of any app (when app is written in a purely functional style)
- Critical for debugging issues produced in other timezones

Mobile code for free

- Save and restore
- Ship code to a data grid
- Build apps out of multiple processes

Technology : modularity

Anything you can do to break down interactions between components will help you eventually

- Critical to plan for future local modifications to code
- Won't have time to do whole program transformations to fix things
- Unnoticed interactions between components will significantly slow you down
- Use types, purity, existential hiding to statically enforce modularity – and often gain compositionality

Engineering : modularity

Use dynamic loading of modules to increase component modularity:

- Quick fix and replace of code
- Enforces strong separation of components
- Leads to useful meta-programming abilities – embedded scripting
- Faster start times, lazy/on-demand loading as needed

Technology : total FP

Partial functions will eventually bite you

- If the type doesn't constrain the data properly, fix the type
- If you can't, make sure you have detailed intro / elimination forms to validate the data

Technology : DSLs

A good way to increase productivity is to use custom languages

- Use an embedded domain-specific language to generate code for repetitive, or tricky parts
- Use types to derive smarter code generation

Massive power to weight ratio --- because you're using language technology

Example: generating GUI apps for declarative specifications of the layout of fields in data types

Engineering : testing

Don't let bugs make it into code that is visible to users

- You make more mistakes than you think
- Mistrust is essential!
- **Assume** any complex change will break code
- Test ruthlessly – “things get weird at scale”

Continuous **property-based testing** running on spare machine

Stress testing to find corner cases – be nasty!

People : testing

Having better testing than your colleagues, when part of a large team can be interesting:

- You start to find bugs in other people's code
- They start having to fix things
- You find things sooner, while they're still fresh
- Your job gets easier

People end up working on your problems for you

Engineering : Daily releases

Roll releases daily

- Helps build a community of beta users
- Keeps expectations reasonable – incremental improvements on a continuous basis
- Becomes easier to estimate delivery dates

Daily delivery schedules

- Increases trust and goodwill by your users
- Allows for faster convergence on good solutions
- Easier to stay focused on user value

Engineering : validation

Catch as many errors as early as possible

Type checking not enough …

- Test on every commit
- Run every sanity check that's feasible
- Auto-reject patches that fail

Double, or triple “distill” the code before shipping through repeated checking and culling

Engineering : accessibility

Make sure it is as easy as possible to get and run your software

- Under-appreciated, but has a big impact!
- Put links in visible places
- Simplify installation
- Minimize time to install

Play well with competition tools – allow data in and out of your app, in formats immediately usable by the competition

Your tool will become the de facto standard

Engineering : debugging

Use tools that support reproducible program states.

- Stack traces – hard to live without them
- Minimize application GC roots
- Snapshot application state on errors

Significantly easier in a pure FP setting – initialization == function application

People : debugging

Make it as easy as possible to report errors by your users

- Users tend to do weird things
- If your app goes wrong, make sure it is only 1 click to send a detailed bug report

Engineering : profiling

High quality software is efficient with resources

- Essential to have good profiling tools for time
- Space also useful

Identify hot spots, improve their complexity (constant factors often ok) , repeat

Need to be able to reason through abstractions

People : Don't over engineer

- Typed FP folks can be overly tempted by category-theoretic patterns

```
fmap (fmap (fmap (fmap …
```

- With type inference, hides subtleties.
- Can bite when refactoring – code will continue to type check, but is silently picking an unexpected instance
- Take a lesson from startup land : build the minimally viable thing, before trying to build a "fully general" solution

People : write documentation

Most developers don't like writing documentation

- I don't know why

Write stories to yourself in code – sort of like speaking aloud what you're trying to do

- Documentation helps others.
- It is good to help others.

People : teams

With such levels of automation and user buy-in to the process, “people skills” become very important

- Empathise with user's concerns (and pain)
- Feel personally responsible for the user's success with the software
- High attention to detail – small oversights can hurt
- Harder: know when to apply theory, and when more ad hoc solutions are appropriate
- When the problem is your fault, or another team – and how to get them to help you.

People : gather data

Gather as much data as you can about your users

- What code are they running, how, where, when?
- Machine specs, system info, machine resources
- What are they doing with the tools?

Use this to increase adoption through targetted marketing (aka email)

Optimize for behaviours most users will notice

Use the data as evidence when getting assistance from other groups

People : hiring

Lots of people out there with some typed FP experience now

- More than you can hire

However, hard to evaluate skills

- I have a strong bias towards demonstrated success in open source
- Succeeding in open source (i.e. can write an accepted library on Hackage or Github) strongly correlates with broad tech skills
- Knowing how to market open source is much rarer...

Conclusion: *Managing complexity*

Keeping complexity down is the key to success

- Leads to accurate estimations of task time and cost
- Software delivered on schedule
- Simpler maintainance
- More composable and reusable code
- Benefits become clear at scale and under pressure
- You get to Nirvana faster, and stay there longer

Don't rely on humans – use your language and machine to control complexity