Teaching Software Architecture Using Haskell

Alejandro Serrano Mena
Universiteit Utrecht
A.SerranoMena@uu.nl

Trends in Functional Programming in Education
Soesterberg, May 25, 2014

From Programming Languages to Software Architecture

Usual topics on Computer Science curriculum:

- Programming and Algorithms

From Programming Languages to Software Architecture

Usual topics on Computer Science curriculum:

- Programming and Algorithms
- Software Engineering

From Programming Languages to Software Architecture

Usual topics on Computer Science curriculum:

- Programming and Algorithms
- Software Engineering
- Software Architecture

From Programming Languages to Software Architecture

Usual topics on Computer Science curriculum:

- Programming and Algorithms → low-level
- Software Engineering → middle-level
- Software Architecture → high-level

From Programming Languages to Software Architecture

Usual topics on Computer Science curriculum:

- Programming and Algorithms → low-level
- Software Engineering → middle-level
- Software Architecture → high-level

Software Architecture will become more important, as larger systems are built and maintained

Current Software Architecture Courses

Software Architecture is taught in a descriptive way:

- Software lifecycle
- Quality attributes
- Requirements
- Architectural patterns and styles

Current Software Architecture Courses

Software Architecture is taught in a descriptive way:

- Software lifecycle
- Quality attributes
- Requirements
- Architectural patterns and styles

But it usually misses:

- Exercising the patterns
- Formal treatment

Current Software Architecture Courses

Software Architecture is taught in a descriptive way:

- Software lifecycle
- Quality attributes
- Requirements
- Architectural patterns and styles

But it usually misses:

- Exercising the patterns
- Formal treatment

→ Haskell!

Haskell and Hackage Features

- Many ways to abstract
- Ease of formal reasoning
- Enforcement of invariants by strong typing
  - Think of the `ST` monad

Haskell and Hackage Features

- Many ways to abstract
- Ease of formal reasoning
- Enforcement of invariants by strong typing
  - Think of the `ST` monad
- Tons of libraries ready to be used
- Libraries are made of small pieces
  - Well structured as primitives + combinators
  - And their developers care about laws

Benefits

Exercising Software Architecture

- Hackage provides libraries which encode most of the usual architectural patterns and styles
- Students can fiddle with building an actual system
  - Pinpoint problems when implementing the pattern
- Types keeps students honest

Benefits

Exercising Software Architecture

- Hackage provides libraries which encode most of the usual architectural patterns and styles
- Students can fiddle with building an actual system
  - Pinpoint problems when implementing the pattern
- Types keeps students honest

Reasoning about patterns

- Look at primitives, their types and laws
  - And reason formally about what it's possible
- Ask questions about design choices
- Simulate other patterns, while keeping invariants

Example: Shared Database / Blackboard Architectural Pattern

A series of components communicate with each other by reading and updating a common repository for data

`stm` library shows this pattern

- Common repository = a set of `TVars`
- Invariants are enforced by types
  - `STM` monad with restricted set of operations
  - `atomically` to execute the transaction in `IO` → why?
- Documentation provides laws

```
M1 `orElse` (M2 `orElse` M3) = (M1 `orElse` M2) `orElse` M3
           retry `orElse` M = M
           M `orElse` retry = M
```

- Channels can be simulated

Side Effect: Learning More FP!

- Programming and Algorithms
- Software Engineering
- Software Architecture

Side Effect: Learning More FP!

- Programming and Algorithms → Imperative Functional
- Software Engineering → Object Oriented
- Software Architecture → Functional?

These libraries bridge the gap between functional programming and its use in an actual entire system

- Students loose the feeling that FP is just for academia

Other Architectural Patterns

More mappings are found in the full paper

- Patterns from *Software Architecture in Practice* [Bass, Clemens and Kazman, 2012], used at UU

Relaxed layers	$\longleftrightarrow$	Monad transformers
Broker	$\longleftrightarrow$	Effects
Pipe-and-filter	$\longleftrightarrow$	Streams and pipes
Dataflow and MapReduce	$\longleftrightarrow$	`IVars`
Shared database	$\longleftrightarrow$	STM
Peer-to-peer	$\longleftrightarrow$	Actors in Cloud Haskell
Publish-subscribe	$\longleftrightarrow$	Reactive

Program Flow with Monad Transformers

Monad transformers express relaxed layers

- Each layer is allowed to call all below it

Program Flow with Monad Transformers

Monad transformers express relaxed layers

- Each layer is allowed to call all below it

Transformers are well-known to be non-commutative

- The assumption that the way you combine independent parts do not affect the final result is false
- This is clear by looking at the types!

Program Flow with Monad Transformers

Monad transformers express relaxed layers

- Each layer is allowed to call all below it

Transformers are well-known to be non-commutative

- The assumption that the way you combine independent parts do not affect the final result is false
- This is clear by looking at the types!

Point at the differences between data and control flow

- Monad layers do not just save and interchange data, they can also affect to the entire control flow
- Composition becomes much more difficult $\rightarrow$ `monad-control`
- This is an interesting architectural problem

Resource Management

Functional components are side-effect-free

- We can model independent systems which cooperate

One global issue is resource management

- How are shared connections, sockets, . . . handled?
- Instrumental for getting good performance

Resource Management

Functional components are side-effect-free

- We can model independent systems which cooperate

One global issue is resource management

- How are shared connections, sockets, . . . handled?
- Instrumental for getting good performance

A solution from Hackage: `resourcet` and `pipes-safe`

```
allocate :: MonadResource m
         => IO a -> (a -> IO ()) -> m (ReleaseKey, a)
release :: MonadIO m => ReleaseKey -> m ()
```

Needs a central repository of release handlers

Streams and Leftovers

Libraries such as `pipes` and `conduit` model pipe-and-filter

- Sequential components which inspect and manipulate streams
- `pipes` even gives categorical laws

Streams and Leftovers

Libraries such as `pipes` and `conduit` model pipe-and-filter

- Sequential components which inspect and manipulate streams
- `pipes` even gives categorical laws

Problem from architectural point of view: leftovers

- Components can return information into input stream
- Used in parsing with backtracking
- How to handle this problem shows choices
- `pipes` and `conduit` take different paths

Summary

Use Haskell to explore architectural patterns

- Put patterns in practice using Hackage
- Look at primitives and laws
- Span discussion based on libraries design choices
- Strong typing encodes invariants and keeps code honest
- Students keep learning FP