Context Switching

- Projects are composed of different contexts
- For example, when working with a RESTful API, we have to switch between GET operations and processing the data
- The expense in time and effort of switching contexts is called *context switching overhead*

Question

How do you manage the overhead of switching contexts?

Debugging example

- UI crashes when a number beginning with 9 is input
- Look at text processing
- Look at decimal parser
- Finally look at *decimalDigits* function:
- It should be ≤ 9, not < 9, and it only gives the wrong result when the number begins with 9.

Psychology of Context Switching

- Studies, etc.
- One theory calls this “Task-set inertia”
- Your task gains inertia as you continue and it takes less effort to continue, just like taking your foot off the gas doesn’t stop the car.
- Switching from one context to a radically different one (an 180 deg turn) requires that you make a full stop, reorient yourself, and fully accelerate. You lost the inertia that was carrying you forward, you expended effort to stop, and you expended effort to restart. Doing this often is exhausting and inefficient.

Sources of Overhead

- We have to store the state of what we're doing in short term memory
- We may have to adapt our environment (Think switching IDEs or opening/closing tabs)
- If we're multitasking *at all*, switching contexts has all the *stop the world* problems of a poorly threaded stop-the-world garbage collector
- We have to either fetch or recreate the state of the other context

Car analogy

- If you're not going very fast, it's a lot easier to turn
  - Try tapping slower
- If we're going fast, we can still turn slowly
  - Try changing the tapping height more slowly
- When going fast, the destination is reached more quickly but mistakes are amplified

Back to code

How do these things translate to working with codebases?

- If we work through things slowly, it's much easier to change contexts, but we get less done
- If we dig into a part of the code, it's hard to extract ourselves, but it may be easier to drift to nearby sections
- Keeping high and low level code in mind is hard

How to manage this?

1. Just don't switch contexts
   - Only work within one model
   - Don't dig into details, or distribute tasks so each person stays within one general context
2. Minimize the overhead required to switch contexts
   - Keep coding style consistent
   - Have an architecture that's easy to navigate
   - Increase rapport between sections

Minimizing overhead

We can minimize context switching overhead through good abstractions.
This allows us to

1. Magnify small actions in controlled ways
2. Utilize similar patterns across a diverse codebase

What's the point of an abstraction?

To factor out repeated patterns and operations.
The best abstractions: 1) make the code easier to understand, 2) make it shorter, 3) make it faster.
Anything that does the opposite is not a good abstraction.

What makes a good abstraction?

What, in your opinion, makes a good abstraction?
“All problems in computer science can be solved by another level of indirection, except for the problem of too many layers of indirection.” âĂŞ David J. Wheeler

What makes a good abstraction? Cont.

Abstractions deal with the problem of specificity: The more specific, the more useful in one case and useless in the general case. The less specific, the more useless in any specific case.
HAMMER SCREWDRIVER COMBO
“The key to making programs fast is to make them do practically nothing.” – Mike Haertel, original author of GNU grep

What makes a good abstraction? Cont.

In my opinion, a good abstraction takes a common yet restrictive pattern that is not bound to a single domain and provides a common interface.
A lens example: '_head' provides access to a large number of container types that have a first value.

Types of abstractions

I break abstractions into three types:

1. Generics: Methods that “do the same thing” in differing contexts
2. Transformers: Methods for transferring a method between contexts
3. Combinators: Methods of combining functions that behave a certain way

Generic classes

A function that does exactly the same thing in a number of contexts

Class	Use when you know:
Use it to:	
Show	How to convert it to a String
Show things	
Semigroup	How to combine it associatively
Simplify your style	
Alternative	When it fails
Try, try again	

Limits:

- Instances must be defined for many types manually
- You're trusting that it's implemented properly

Transformer classes

A method of transferring a method from one context to another

Class	Use when you know:	Use it to:
Functor	How to apply a function	Map generically
folds	How to reduce it with a	Extend binary operations to more arguments
unfolds	How to extend it	Build recursively-defined data structures

Limits:

- Builders may look radically different for different types
  - For example, vectors vs. cons lists
- Nesting transformers can create difficulties
  - For example, nested monad transformer problem in Haskell

Combinator classes

A methods of combining other methods within a given context

Class	Use when you know:	Use it to:
Applicative	How to apply functions inside i	Generalize ‘fmap’ to n-ary functions
Bifunctor	How to apply a function to two	Map more complex types generically
Monads	How to compose functions tha	Sequence operations you can’t unwrap

Limits:

- Nesting different types of composition can reduce benefits (for example, StateT (MaybeT IO))

General limits

- It's easy to use abstractions to avoid solving anything
  - "Lenses solve many problems, including ones you didn't have until you started using lenses"
- If rules are not established, followed, and tested, you get things like an overloaded , operator that prints things in C++
- "No free lunch" means that more advanced abstractions are harder to use
- Abstractions that make one part of a program easier to reason about often make another part much harder to understand
  - This is seen all over the place in Haskell with predicting performance
- Overabstracting can reduce clarity when applied to simple problems

Example of overabstraction

Junior Haskell Programmer:

Applying abstractions to parsing

Here, I will use making a monadic parser in Haskell to show example problems and solutions through abstraction.

Parsing Features

What are some things we want to cover with our parser?

- We want to be able to parse small pieces with descriptions
- We want to be able to combine small parsers into larger parsers
- If a parser fails, we should be able to try something else

Parser type

newtype Parse object = Parser { runParser :: String -¿Either String (String, object) }

Combinator classes: Functor

class Functor f where fmap :: (a -¿ b) -¿ f a -¿ f b
Given:
fmap id == id fmap (f . g) == fmap f . fmap g

Functor example

Given a function *infuriateCoder* :: *Coder* $\rightarrow$ *AngryCoder* then fmap infuriateCoder :: Either NonCoder Coder -¿ Either NonCoder AngryCoder – Only infuriate coders, we'd need another function for NonCoder's
fmap infuriateCoder :: RoomFullOf Coder -¿ RoomFullOf AngryCoder – Infuriate every coder in the room, wouldn't work if there were any NonCoder's in the room
Given a $RoomFullOf(EitherNonCoderCoder)$
fmap (fmap infuriateCoder) :: RoomFullOf (Either NonCoder Coder) -¿ RoomFullOf (Either NonCoder AngryCoder)

Functor requirements

fmap id == id – means that fmaping the identity to all the objects in f̈ objects̈hould do nothing
fmap (f . g) = fmap f . fmap g – means that fmaping one function then another is the same as fmaping b̈oth at once

Another Functor example

toJSON :: Account -¿ JSON Account parseAccount :: Parse Account parseAccountJSON :: Parse (JSON Account) parseAccountJSON = fmap toJSON parseAccount

Functor instance for Parse

instance Functor Parse where fmap :: (a -¿ b) -¿ Parse a -¿ Parse b fmap f (Parser p) = Parser (fmap (fmap f) . p)

Applicative class

pure :: Applicative f =¿ a -¿ f a (¡*¿) Applicative f =¿ f (a -¿ b) -¿ f a -¿ f b
f x1 -¿ f ¡$¿ x1 :: Applicative f =¿ (a1 -¿ b) f a1 -¿ f b
f x1 x2 -¿ f ¡$¿ x1 ¡*¿ x2 :: Applicative f =¿ (a2 -¿ a1 -¿ b) -¿ f a2 -¿ f a1 -¿ f b
f x1 x2 x3 -¿ f ¡$¿ x1 ¡*¿ x2 ¡*¿ x3 :: Applicative f =¿ (a3 -¿ a2 -¿ a1 -¿ b) -¿ f a3 -¿ f a2 -¿ f a1 -¿ f b
f x1 x2 x3 x4 -¿ f ¡$¿ x1 ¡*¿ x2 ¡*¿ x3 ¡*¿ x4 :: Applicative f =¿ (a4 -¿ a3 -¿ a2 -¿ a1 -¿ b) -¿ f a4 -¿ f a3 -¿ f a2 -¿ f a1 -¿ f b

Applicative examples

(+) ¡$¿ parseInt ¡*¿ parseInt – parses two integers and returns their sum

(,) ¡$¿ parseA ¡*¿ parseB = parseAB – parses A and B, and returns them in a tuple

Applicative instance for Parse

```
instance Applicative Parse where pure :: a -¿ Parse a pure =
Parser . (Right .) . flip (,)
(¡*¿) :: Parse (a -¿ b) -¿ Parse a -¿ Parse b Parser f ¡*¿ Parser x =
Parser $ combine . f where combine (Left s1 ) = Left s1 combine
(Right (s2, f')) = fmap (fmap f') (x s2)
```

Monad class

The core function for Monads is the bind function @(»=)@. Here's one way to understand it:
(»=) :: t a -¿ (a -¿ t b) -¿ t b passedTo :: a -¿ (a -¿ b) -¿ b x f = f x

Monad example

(»=) :: Either Bird Dog -¿ (Dog -¿ Either Bird Dog) -¿ Either Bird Dog

Monad instance for Parse

instance Monad Parse where return :: a -¿ Parse a return = pure
(»=) :: Parse a -¿ (a -¿ Parse b) -¿ Parse b Parser x »= f = Parser
$
s -¿ x s »=
(s', y) -¿ either (const $ Left s) Right (runParser (f y) s')

Monad instance for Parse Cont.

For example:
Parser Password -¿ (Password -¿ Parser Secret) -¿ Parser Secret

Alternative class

empty :: Alternative f =¿ f a (¡—¿) :: Alternative f =¿ f a -¿ f a -¿ f a

Alternative instance for Parse

instance Alternative Parse where empty :: Parse a empty = Parser Left – A parser that consumes no input and always fails (¡—¿) :: Parse a -¿ Parse a -¿ Parse a Parser x ¡—¿ Parser y = Parser $ either y Right . x – If x returns (Right x') then return (Right x') else return y