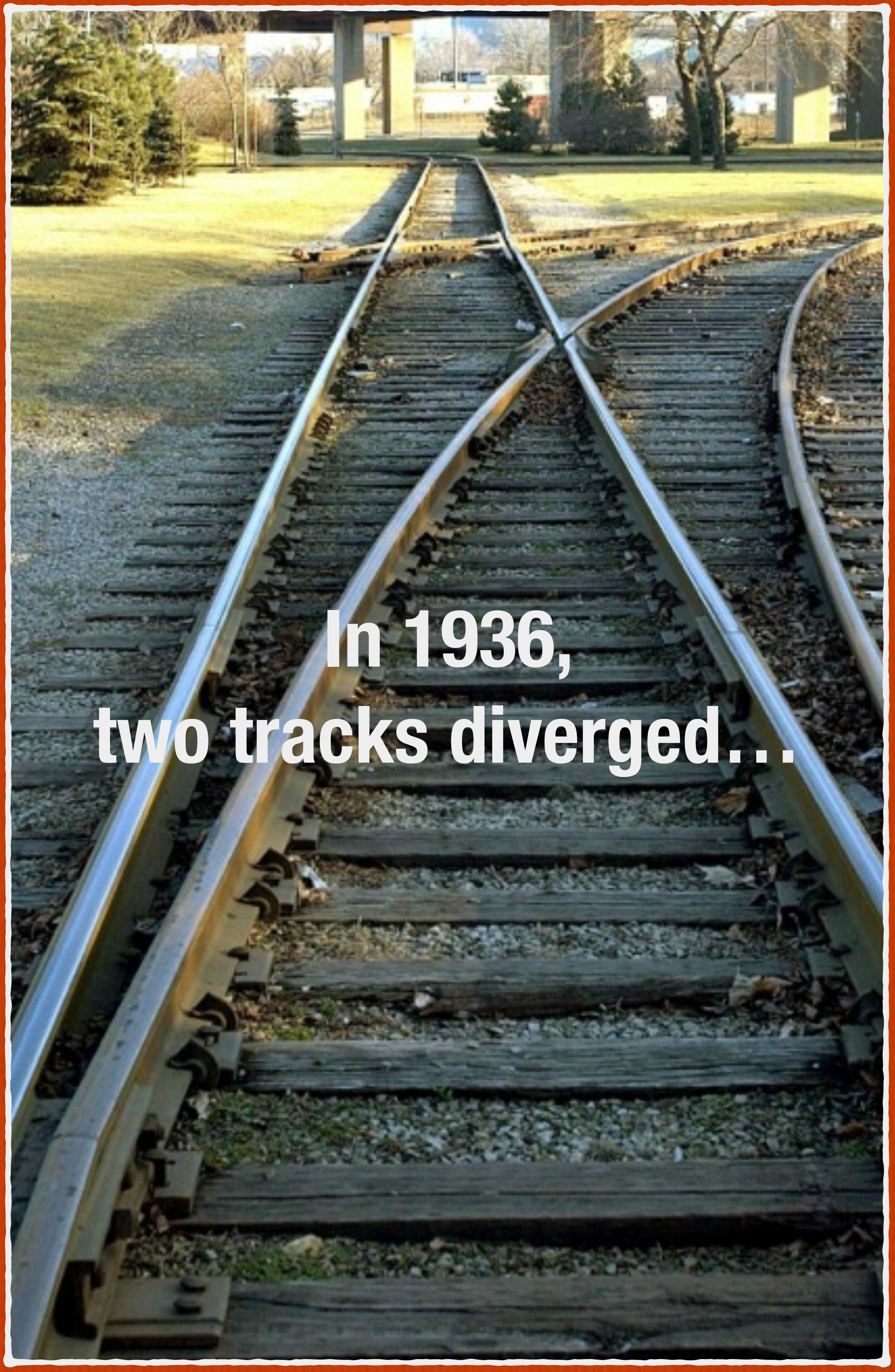
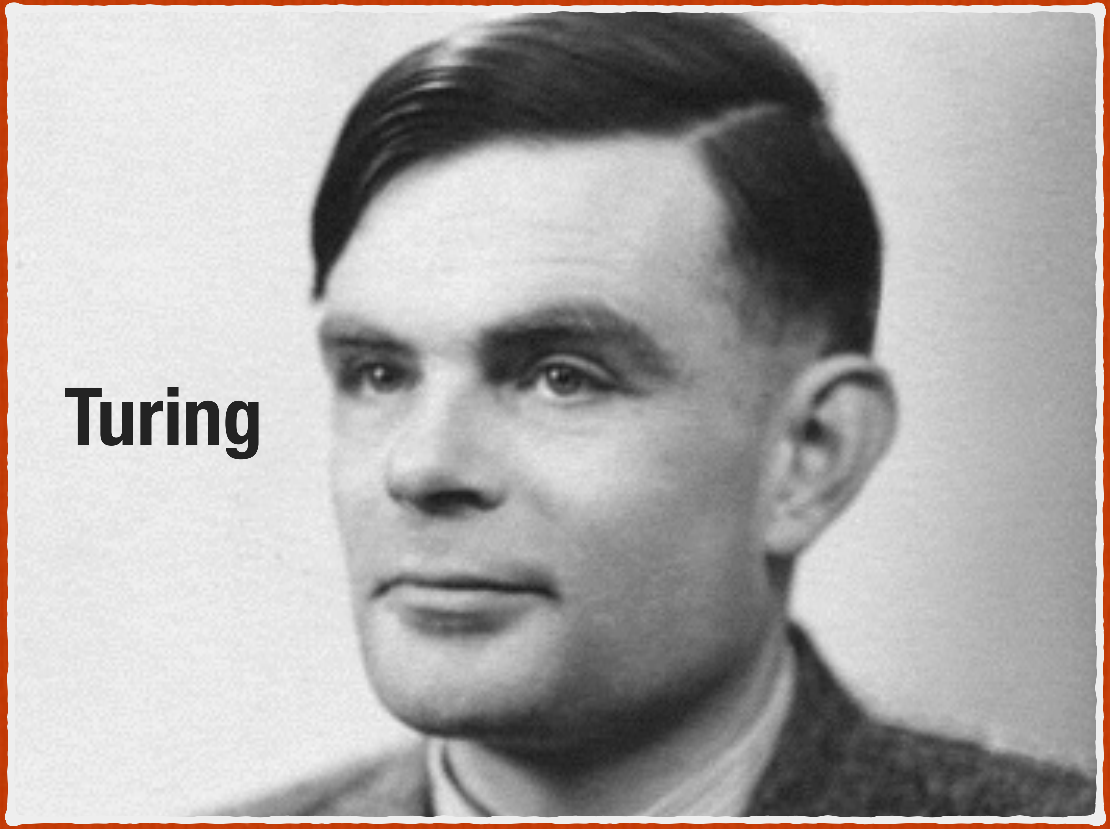
Functional Programming

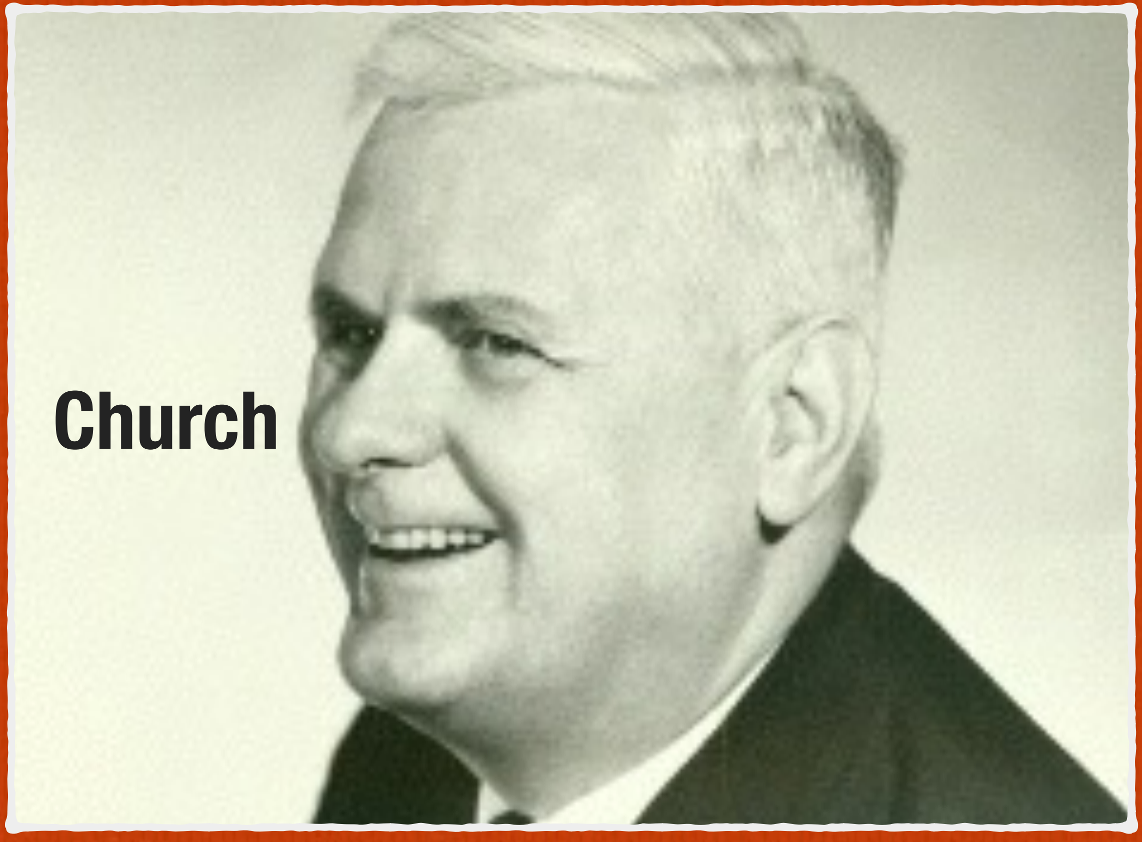
In five easy parts

Part 1

Background


In 1936,
two tracks diverged…


Turing


Church

Aside: Understanding

- Before I can understand some answer
- I want to know what the question is
- and that usually depends on history

David Hilbert

- Towering mathematical figure in the 20th century
- Proposes, <u>among other things</u>, what becomes known as Entscheidungsproblem

Entscheidungsproblem

- German for "decision problem"
- Asks: "Here's a statement in first-order logic, can you give me an algorithm to decide if it is universally true?"
- In solving this problem, both Turing and Church define what computation is
- BTW: the answer to the D.P. turns out to be "no" in general, but that's a whole other talk!

Aside: First-order logic

- $\forall x$ hacks_ruby(x) $\Rightarrow$ is_a_programmer(x)

  "It is true for everyone, that if you program ruby then you are also a programmer"

- $\exists x$ hacks_ruby(x) $\wedge$ hacks_haskell(x)

  "There's someone who uses both ruby and haskell"

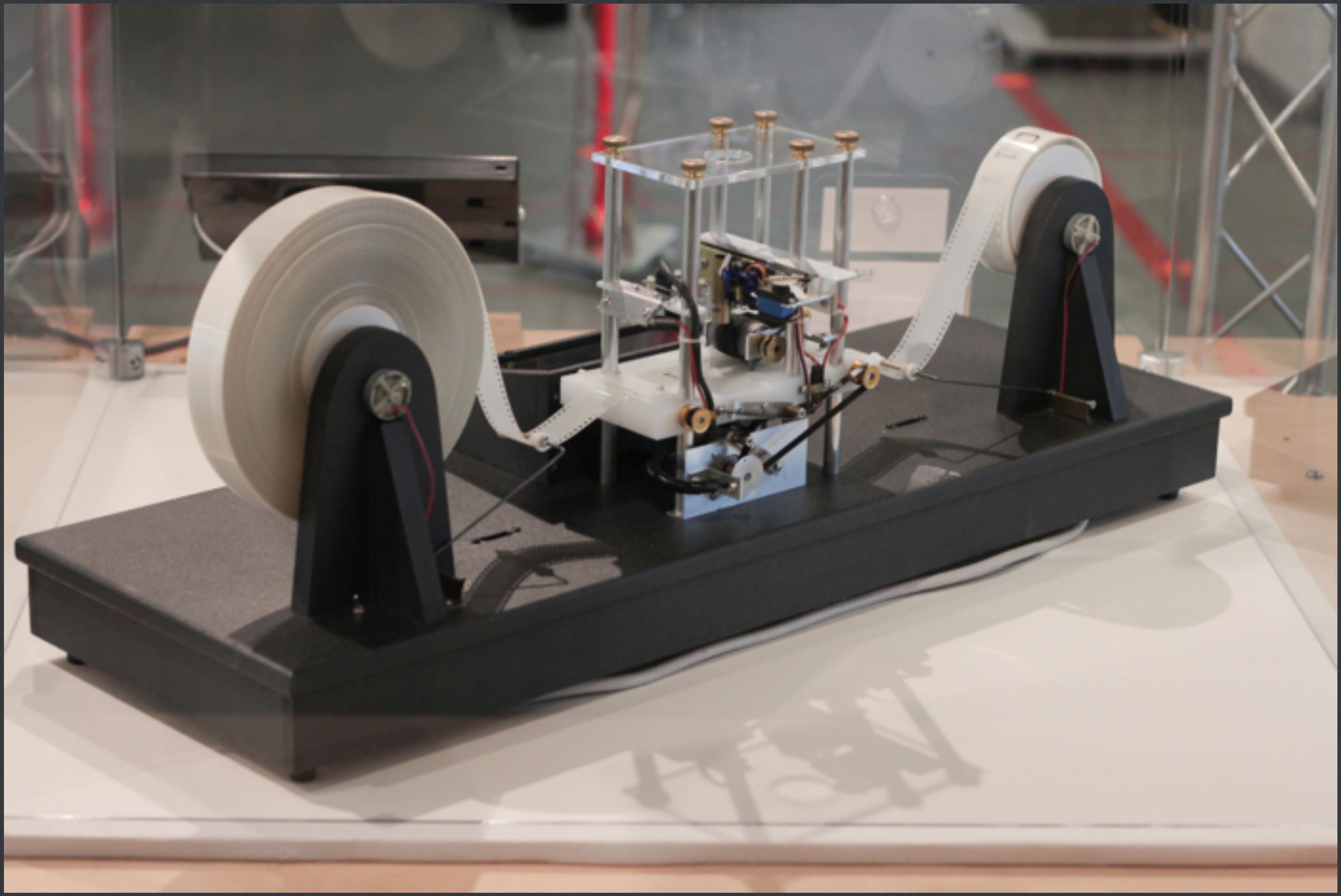
Question: Entscheidungsproblem

Turing's Answer

Turing

- Perhaps better-known of the two
- You can compute with a machine that has an infinite paper tape…
- …also did a bunch of other things like crack WWII German codes, helped to design early computers, and described a test for artificial intelligence…
  - just a few things…

Turing Machine



Church's Answer

Church

- Published "An Unsolvable Problem of Elementary Number Theory" slightly before Turing, though Turing didn't know about it
- You can compute using the λ-calculus…

Aside: λ-Calculus

- α-conversion (rename): $(\lambda\ x\ .\ x) \rightarrow (\lambda\ y\ .\ y)$
- β-reduction (apply): $(\lambda\ x\ .\ x)\ y \rightarrow y$
- η-conversion ("cancel" args.): $(\lambda\ x\ .\ f(x)) \rightarrow f$

Aside: λ-Calculus

- Church encoding of numerals:
- 0 := λf.λx.x
  1 := λf.λx.f x
  2 := λf.λx.f (f x)
  3 := λf.λx.f (f (f x))
- INC := λn.λf.λx.(f ((n f) x))
  (IYI, show that: INC 1 = 2)

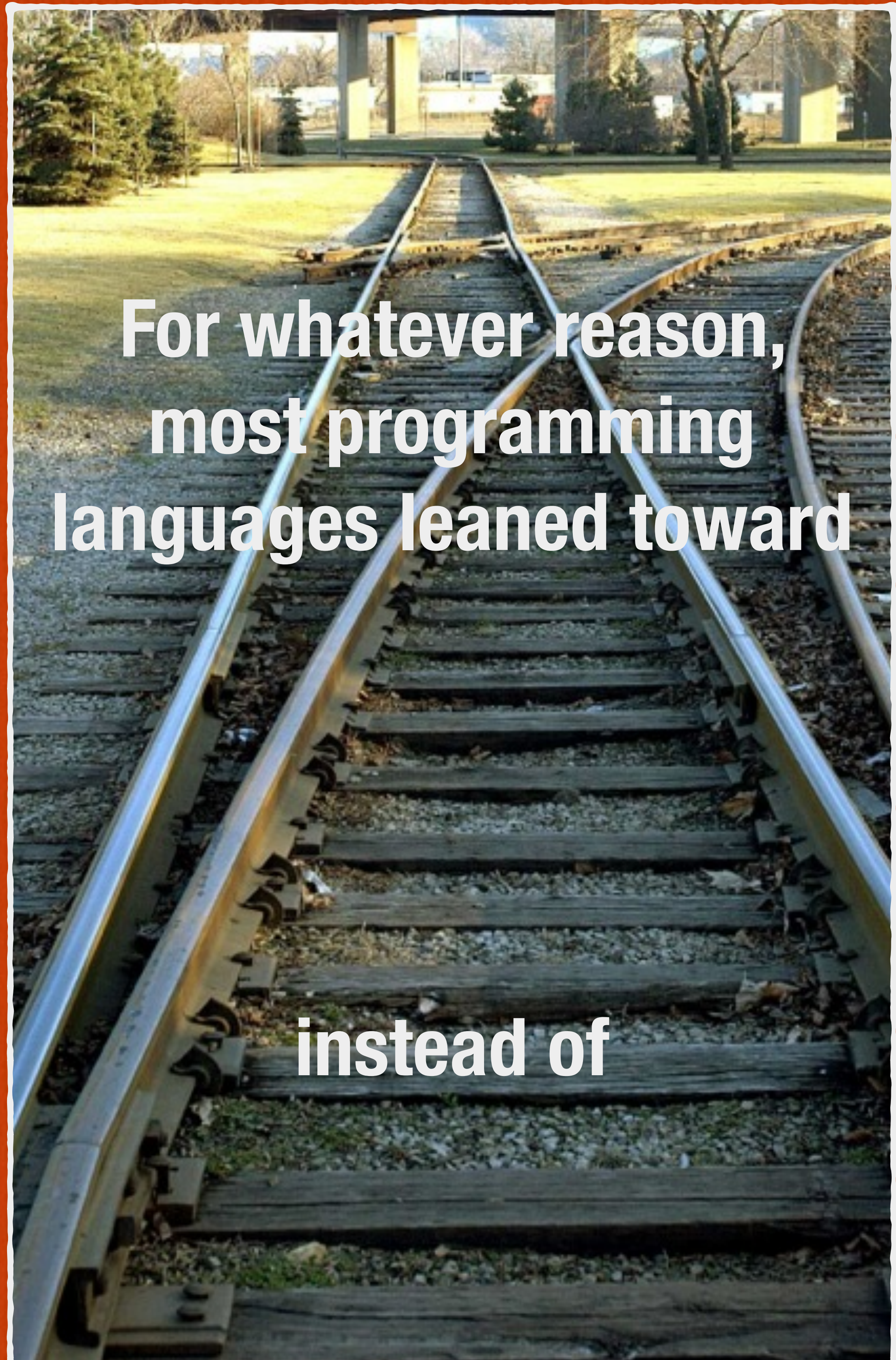
Church-Turing

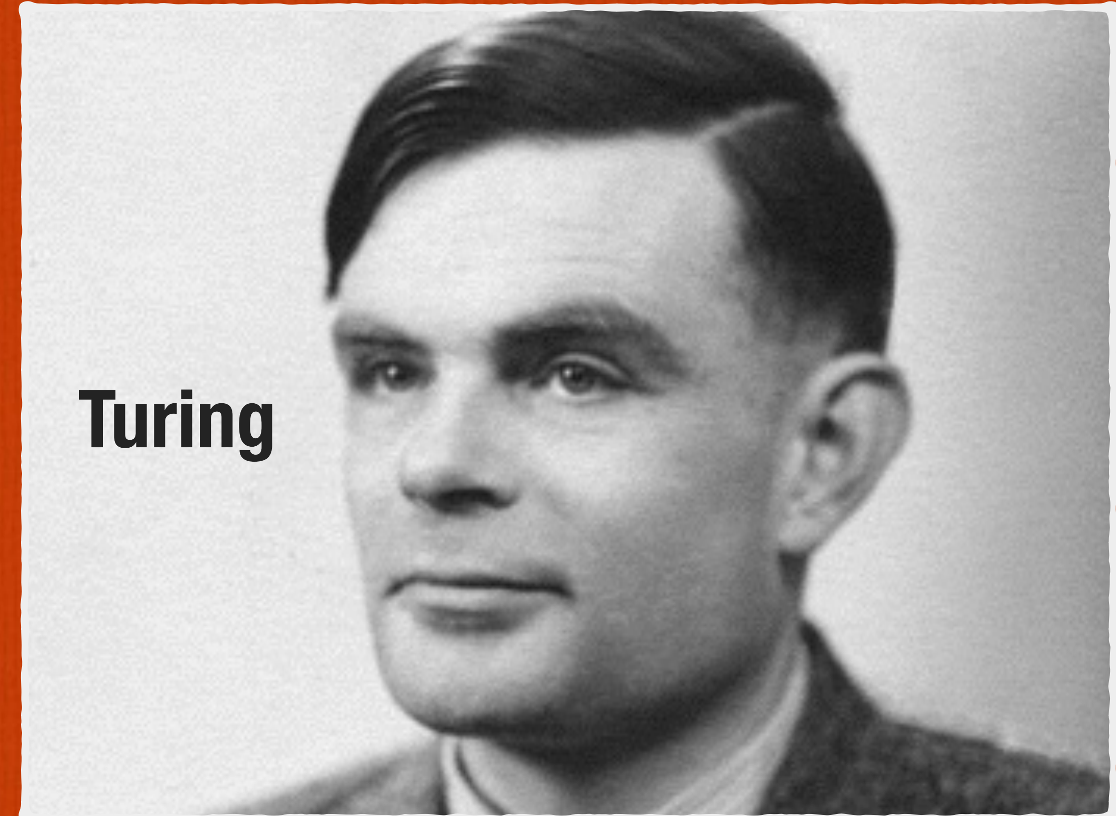
- So, you can compute with either Turing machines or the λ-calculus…
- λ-calculus and Turing machines are equivalent!
  - Anything that can be computed can be computed by the λ-calculus and a Turing machine

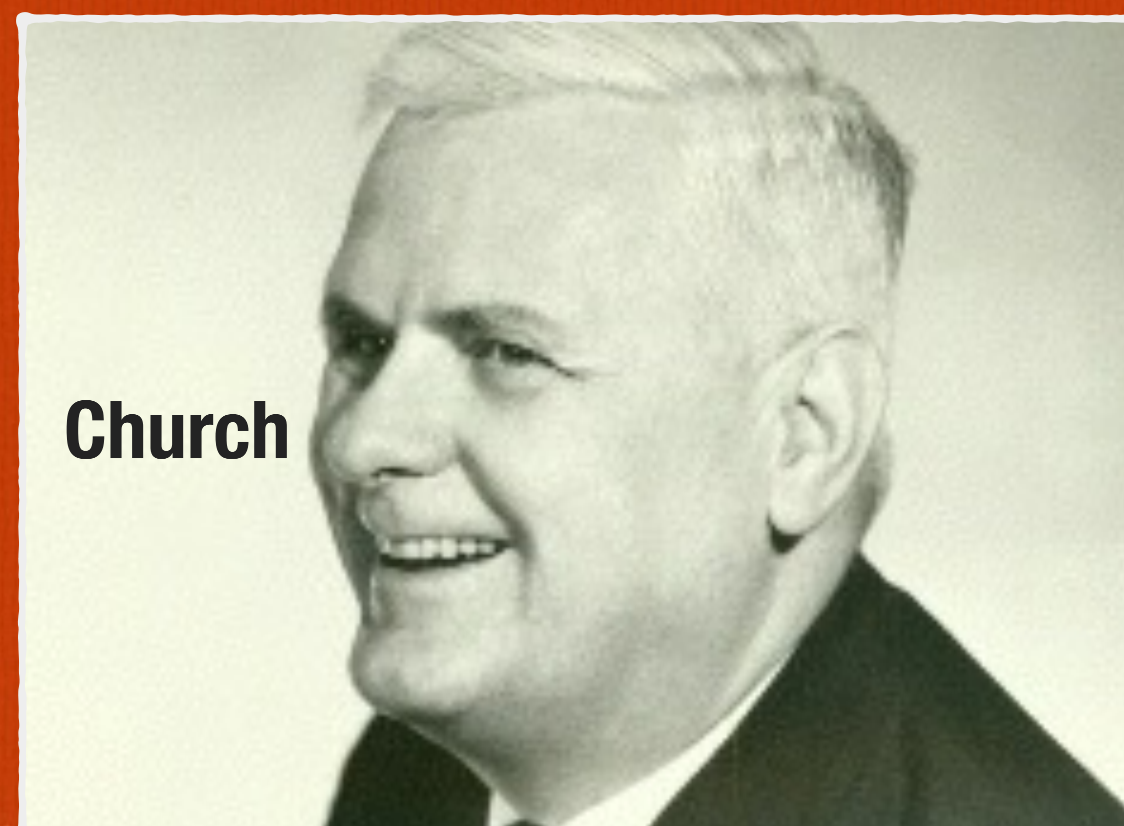
SO!? before functional programming

- Surprisingly then, or maybe not at all, there is no before functional programming
- Functional programming was one of the answers to the question that prompted “computation”

Here we are ~80 years later


For whatever reason,
most programming
languages leaned toward
instead of


Turing


Church

Part 2

Programming with Functions

Functions

- An object that has just one method, “call”
- A correspondence between inputs and outputs
    - each input is related to just one output

What is it about this…

```
Person = Struct.new(:first, :last) do
  def school_name
    "#{last}, #{first}"
  end
end

me = Person.new("Chris", "Wilson")
me.school_name
# => “Wilson, Chris”
```

…that looks the same as this?

```
Person = lambda do |first, last|
  {
   school_name: lambda { "#{last}, #{first}" }
  }
end

me = Person["Chris", "Wilson"]
me[:school_name][]
# => “Wilson, Chris”
```

…or even?

```
Person = {
  # ...
  ["Chris", "Wilson"] => "Wilson, Chris"
  # ...
}

Person[["Chris", "Wilson"]]
# => “Wilson, Chris”
```

What is an object…

- …but a context in which to call a function?
- Why do we distinguish between .new() and any other method call?
- Functions can be called with args and return a closure holding any needed state

Building programs

- Objects structure code
  - little bit of state
  - little bit of behavior
- Functions structure code
  - no state
  - all behavior

Part 3

Abstraction

Composition

```
compose = lambda do |f, g|
  lambda do |x|
    f[g[x]]
  end
end

add5   = lambda{|x| x+5}
double = lambda{|x| x+x}

puts compose[add5, double][3]
# => 11
```

But then…

- …it's only useful for functions that take exactly one argument!?
- That's okay, because that's all that there are.

Currying

- You can always rewrite:
  - some_func(x, y) → some_func(x)(y)
- Built into Ruby:
  - ```
    f = lambda{|x,y| x + y}.curry
    f[2][3]
    # => 5
    ```
    ```

Currying and Compositon

```
compose_all = lambda do |args|
  args.reduce do |memo, f|
    compose[memo, f]
  end
end

add = lambda{|x, y| x + y}.curry
announce = lambda{|x| “Answer: (#{x})”}
funcs = [announce, add[5], double]

compose_all[funcs][3]
# => “Answer: (11)”
```

Change your perspective

- You've all seen map?
- [1, 2, 3].map{|x| x*2} # => [2, 4, 6]
- Used to thinking:
  - map :: (Int → Int) → [Int] → [Int]
- With currying in hand, think of it like:
  - map :: (Int → Int) → ([Int] → [Int])

Change your perspective

- map lifts a function over values to a function over arrays
- fmap lifts a function over values to a function over values in a context
- ```
  class Proc
    def fmap(obj); obj.fmap(self); end
  end

  class Array
    def fmap(f); self.map(&f); end
  end

  lambda {|x|x*2}.fmap([1, 2, 3]) # => [2, 4, 6]
  ```
  ```

It’s more general!

```
class User
  attr_accessor :name
  def fmap(f); f[name]; end
end

u = User.new
u.name = “Chris Wilson”
lambda{|x| x.split}.fmap(u) # => [“Chris”, “Wilson”]
```

Other possibilities for fmap

- Empty-or-not values
- Trees
- Hashes
- Other functions!

Three variations on fmap

- Yeah, let's talk about map even more!
- Watch for similarities

Variation 1: Array

- We know this one: [1, 2, 3, 4].map { |n| n + 1 }
  (or lambda{ |n| n + 1}.fmap([1, 2, 3, 4]))

- But, imagine no “bare” values allowed

- def foo(item)
    item.map { |n| n + 1 }
  end
  foo([1]) # => [2]

Variation 1: Array

- We'd need some "plumbing"
- ```
  def fmap(f, x)
    x.map(&f)
  end

  fmap(->x{x+1}, [1, 2]) # => [2, 3]
  ```
  ```

Variation 2: Hash

- More (but familiar) plumbing:
- ```
  def fmap(f, x)
    x.inject({}) do |memo, (k, v)|
      memo[k] = f[v]; memo
    end
  end

  fmap(->x{x+1}, {a: 1, b: 2}) # => {:a=>2, :b=>3}
  ```
  ```

Variation 3: Proc

- This may be a bit weirder, but think about it…
- Yet more plumbing:
- ```
  def fmap(f, x)
    lambda { |y| f.call(x.call(y)) }
  end

  fmap(->x{x+1}, ->y{y*2})[2] # => 5
  ```
  ```

Variation 3: Proc

- Did you catch that fmap for Procs was just compose?
- plus1 = lambda{|x| x+1}; times2 = lambda{|x| x*2}
  fmap(plus1, times2)[2]
  # => 5
- Think of a Proc as a kind of box “holding” its eventual return value…
  - fmap lets us swap out that value!

Fmap's similarities?

- Is fmap, in some sense, the "same" in all these cases?
- There's a property of mapping independent of Array, Hash, or Function
- Because fmap works for so many different things, it must behave like:

  fmap(g, fmap(f, x)) == fmap(compose(g, f), x)
  fmap(id, x) == id x

Parametric Polymorphism

- or, Zen-like: “more general is more specific”
- Reason about things regardless of specific type
- Notice how we could talk about mapping yet never mention Array?
- Speak at a higher level, “all things that do this can also do that” etc.
- Best: “we don’t know what this is, so we can’t treat it specially”

Part 4

Evaluation

Laziness

```
compute = lambda do |x, y|
  return x if true
  y
end

def expensive
  puts "GREAT EXPENSE!"
  1
end

puts compute[2, expensive]
GREAT EXPENSE!
# => 2
```

Laziness

- Why did we need to evaluate expensive?
- It wasn't ever used!
- Eager evaluation mixes concerns (cf. SoC)
  - Concern 1: computation embodied in the method
  - Concern 2: computation embodied in method's arguments

Laziness

- We often want to decouple code from its evaluation:
  - Scopes, method definitions, lambda/proc, FactoryGirl, let blocks in RSpec…
- Leads to general, modular, and pluggable code (good things!)
- Strict-by-default → often need laziness
- Lazy-by-default → sometimes need strictness

Example: sorting

- Q: what's the time, as in O(N), for:
  - `range.map{rand(1000)}.first`
  - `O(N)`
- How about:
  - `range.lazy.map{rand(1000)}.first`
  - `O(1)`
- `Times (N= 1e7): 3.6s vs 0.000029s`

Aside: Bonus

- Mind-blowing threat level: Elevated
- take 1 (sort random_nums)
- runs in O(N) time!

Part 5

Potpourri

Property testing

- If you take nothing else away from this talk, try this out!
- If we know the domain (math sense) of a function, shouldn't the computer automatically test it?
- What properties hold? Rather than what test cases can I think of?
- Imagine that I wrote "sort" and wanted to test it…

Property testing

```
require 'rushcheck'

# sorting preserves length
RushCheck::Assertion.new(IntegerRandomArray) {|arr|
  arr.sort.length == arr.length
}.check

# first element is min
RushCheck::Assertion.new(IntegerRandomArray) {|arr|
 arr.sort.first == arr.min
}.check

# last element is max
RushCheck::Assertion.new(IntegerRandomArray) {|arr|
  arr.sort.last == arr.max
}.check
```

Property testing

- Run this:
  OK, passed 100 tests.
  OK, passed 100 tests.
  OK, passed 100 tests.
- I just wrote 300 tests

Property testing

- Complements imperative-style tests really well
- Encourages functional design
  - where input and output completely characterize the function
- Great for finding obscure edge cases
  - good libs also find a simpler thing that still fails

rant_mode do

Stuff I wouldn't even try…

- What does FP do better?
    - wrong question
- what do I attempt that I wouldn't even try without functional programming?

Static types

- Most popular static languages have, essentially, types like Algol/Pascal
  - C, C++, Objective C, Java, C#
- Or are dynamic (no static type checking at all)
  - Lisp, JavaScript, Python, Ruby, Perl

Static types

- A lot has happened with types in the last 40 years!
  - e.g. OCaml, F#, Haskell, Scala, Rust
- They can really improve expressiveness:
- map :: (a → b) → [a] → [b]
  find :: (a → Bool) → [a] → Maybe a
  sort :: Ord a => [a] → [a]
- Act as machine-checked comments that can't lie

Dependent types

- Adding two vectors pairwise:
- total

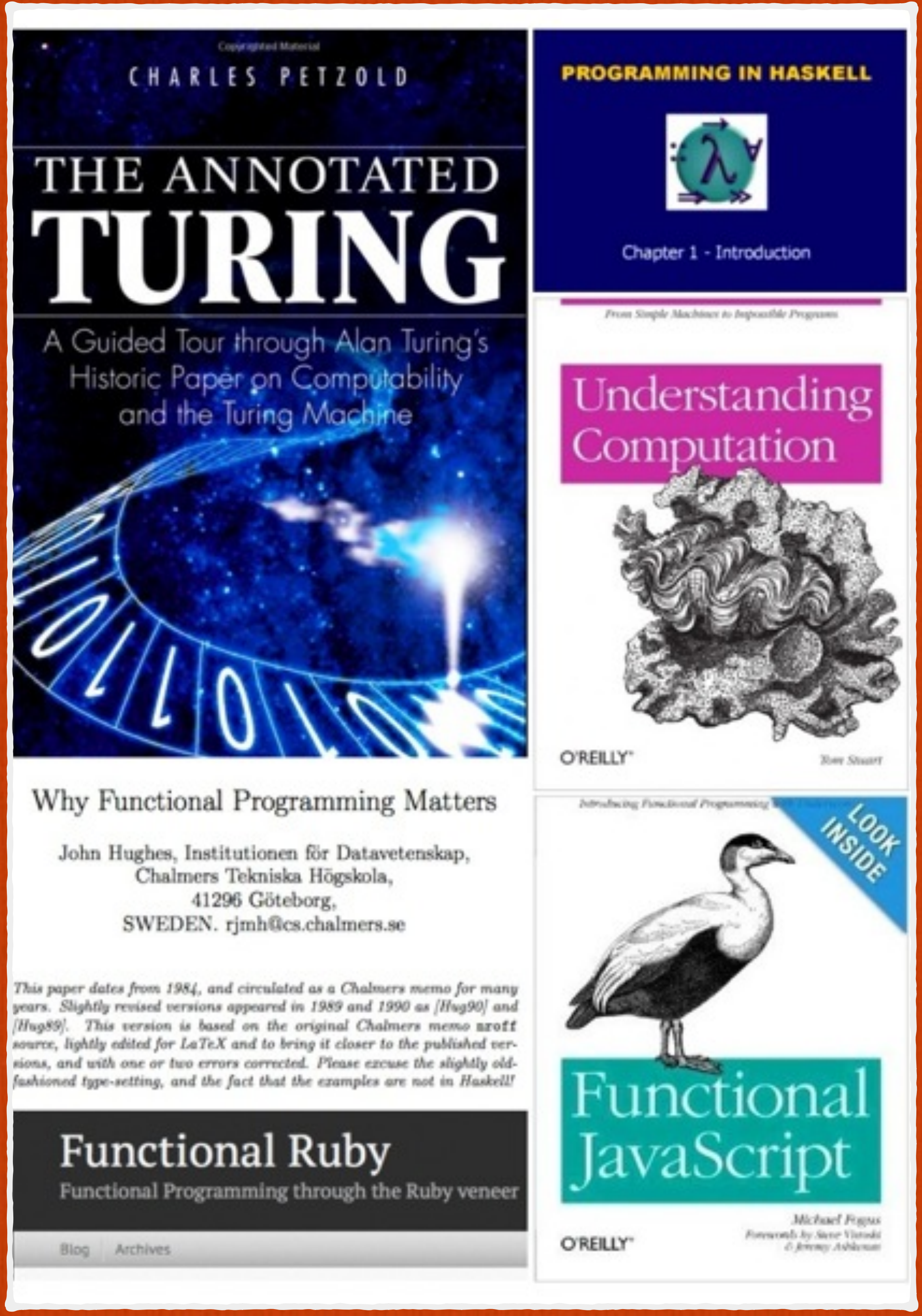
```
pairAdd : Num a => Vect n a -> Vect n a -> Vect n a
pairAdd Nil     Nil     = Nil
pairAdd (x::xs) (y::ys) = (x+y) :: pairAdd xs ys
```

- Type system ensures they are the same length end

Thanks!

Resources

1. C9 Lectures: Functional Programming Fundamentals
2. Functional JavaScript
3. Why Functional Programming Matters
4. Functional Ruby
5. Understanding Computation
6. The Annotated Turing
7. Can Programming Be Liberated from the von Neumann Style? (PDF)



And lots more…
(but you'll have to ask)

Thanks

Chris Wilson

chris@bendyworks.com

@twopoint718

http://sencjw.com