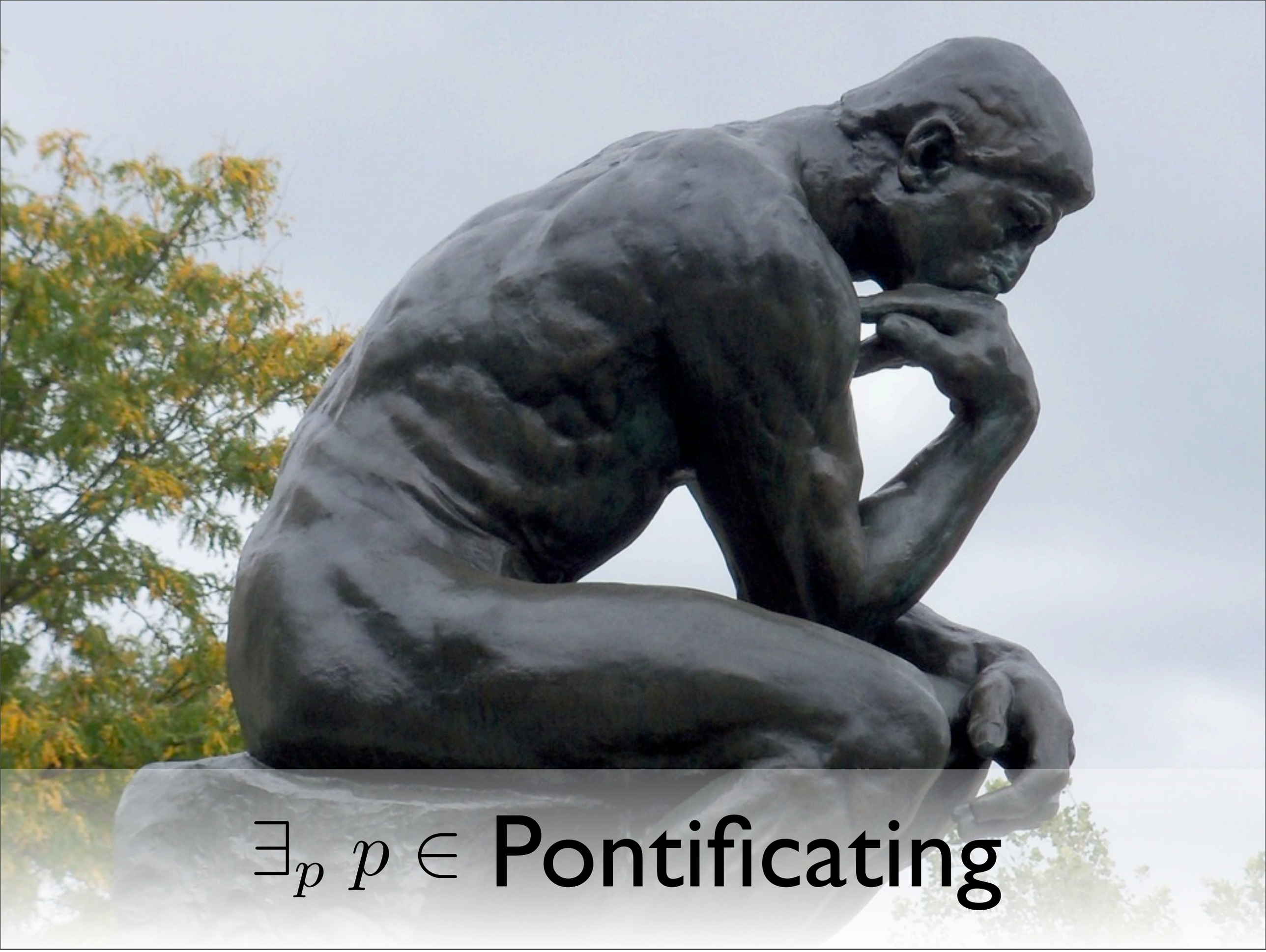
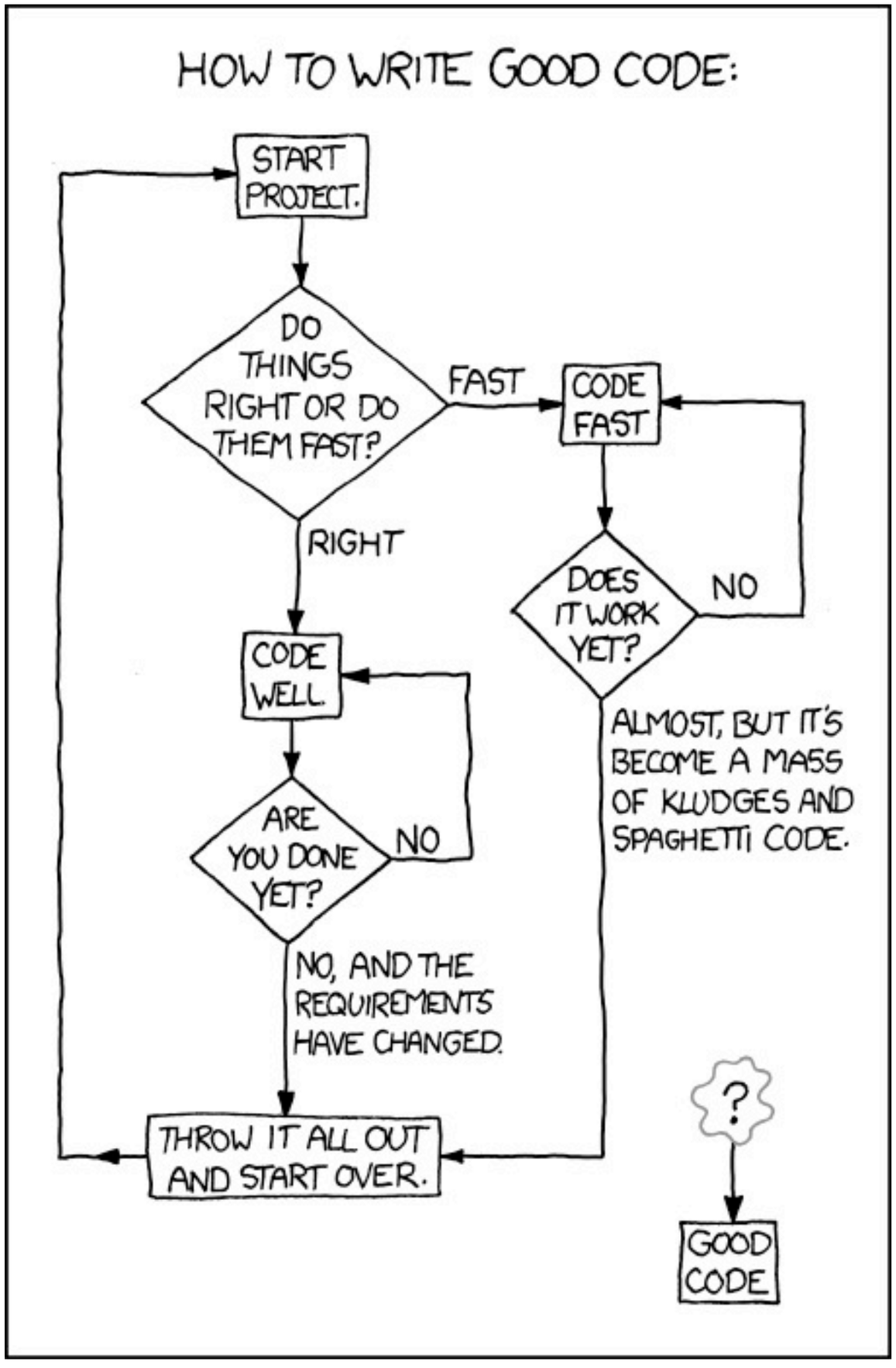


$\exists_p\ p \in$ Pontificating

Programming is Hard


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
RIGHT
DOES IT WORK YET?
NO
CODE WELL.
ARE YOU DONE YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
NO, AND THE REQUIREMENTS HAVE CHANGED.
?
THROW IT ALL OUT AND START OVER.
GOOD CODE

Who watches the watchers?

Let's talk about verification

XUnit

Standard Practice

- We all use it
- Most of us rely on it day to day
- We seem to trust it implicitly

```
def test_example
  # create something to test
  obj = MyExampleObject.new(arguments)
  # program statement
  result = obj.example_method(arguments)
  # verification
  assert_equal("fact", result)
  #implicit teardown
end
```

```
it "must be an example" do
  obj = MyExampleObject.new(arguments)
  result = obj.example_method(arguments)
  result.should == fact
end
```

XUnit is Complected

- It's doing too much

XUnit is Complected

- It's doing too much
- BDD is just XUnit with lots of bad English

XUnit is Complected

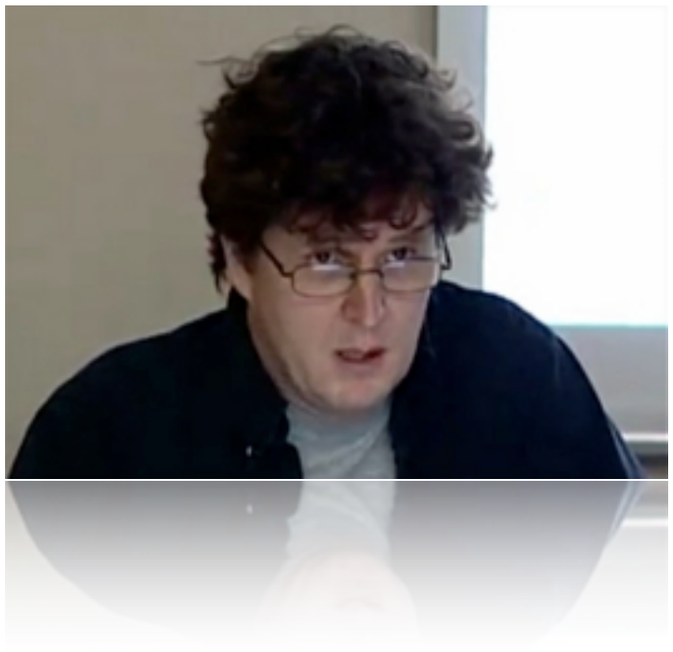
- It's doing too much
- BDD is just XUnit with lots of bad English
- Poor separation of the tester and testee

XUnit is Complected

- It's doing too much
- BDD is just XUnit with lots of bad English
- Poor separation of the tester and testee
- What are you testing?

XUnit is Complected

- It's doing too much
- BDD is just XUnit with lots of bad English
- Poor separation of the tester and testee
- What are you testing?
  - Are you *sure?!*



Your mock object is a joke; that object is mocking you. For needing it. – Rich Hickey

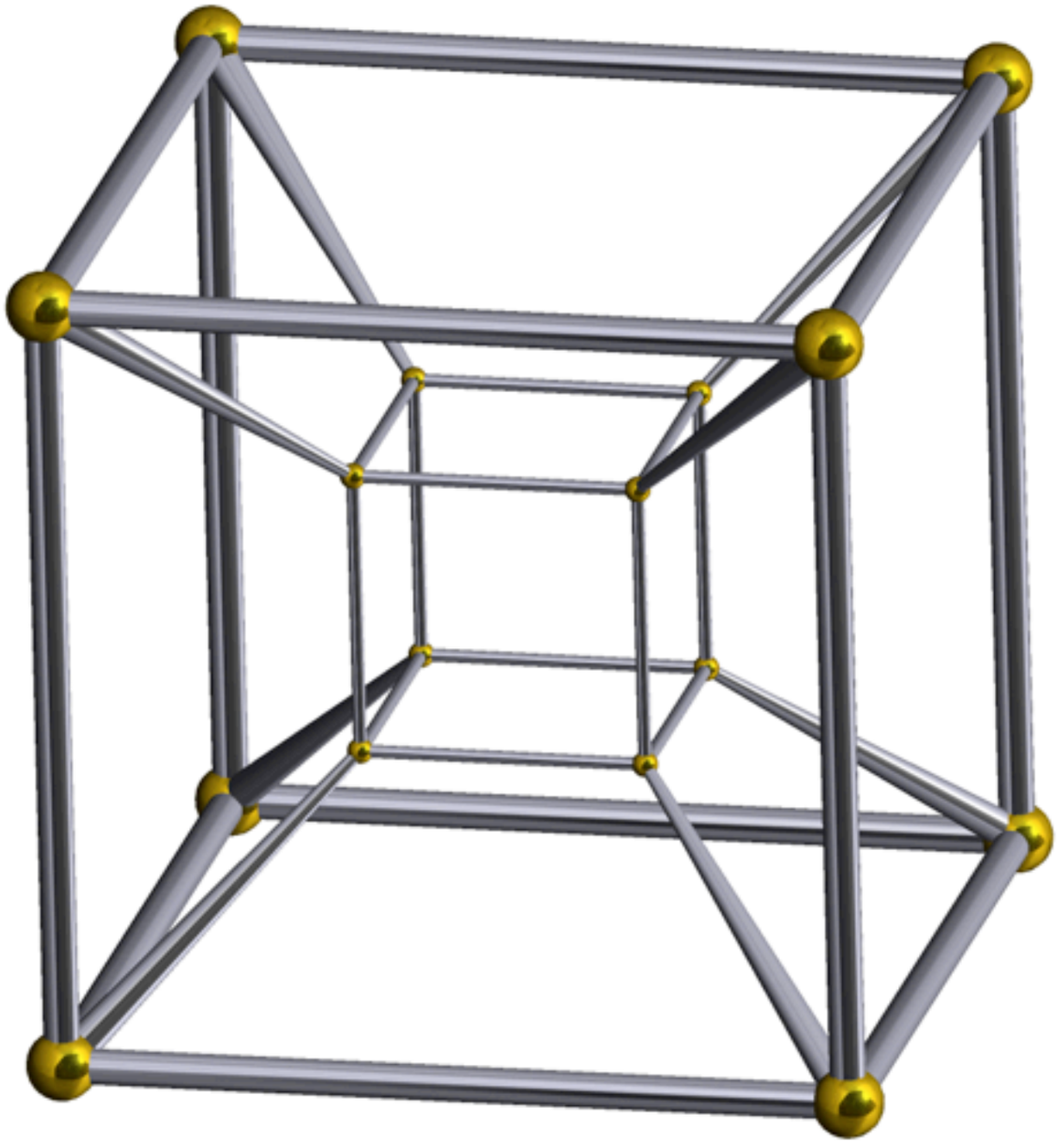
Coverage is a Lie

- Too many false indicators of comprehensive verification
- We work really hard at the wrong things to achieve good coverage
- “The Emperor's new suit”

Always in Manual Drive

- Devise, write, and maintain all cases
- Edge cases are a (big) problem
  - Especially with simple functions
- Example: Integer commutative law

Generative Testing



Logic #fail

$$\exists_x \; f(x) \neq expect(x) \Rightarrow \text{bugs}$$

Logic #fail

$$\exists_x \; f(x) \neq expect(x) \Rightarrow \text{bugs}$$

$$\exists_x \; f(x) = expect(x) \Rightarrow \neg\text{bugs}$$

Logic #fail

$$\exists_x \; f(x) \neq expect(x) \Rightarrow \text{bugs}$$

$$\exists_x \; f(x) \equiv expect(x) \Rightarrow \neg\text{bugs}$$

Generative Testing

- Developing test cases is hard
- We have computers!
- Define your domain, auto-generate inputs
- Write less, test more
- Find edge cases automatically!

Generative Testing

- Developing test cases is hard
- We have computers!
- Define your domain, auto-generate inputs
- Write less, test more
- Find edge cases automatically!
  - (money back guarantee)

```clojure
(defspec integer-commutative-laws
  (partial map identity)
  [^{:tag `integer} a ^{:tag `integer} b]
  (if (longable? (+' a b))
    (assert (= (+ a b) (+ b a)
               (+' a b) (+' b a)
               (unchecked-add a b)
               (unchecked-add b a)))
    (assert (= (+' a b) (+' b a))))
  (if (longable? (*' a b))
    (assert (= (* a b) (* b a)
               (*' a b) (*' b a)
               (unchecked-multiply a b)
               (unchecked-multiply b a)))
    (assert (= (*' a b) (*' b a)))))
```

```
"integer commutative law" in {
  check { (x: Int, y: Int) =>
    (x + y) mustEqual (y + x)
    (x * y) mustEqual (y * x)
  }
}
```

Generative Testing

- Real world cases?

Generative Testing

- Real world cases?
  - Sparse trees

Generative Testing

- Real world cases?
  - Sparse trees
  - Complex pre-conditions

Generative Testing

- Real world cases?
  - Sparse trees
  - Complex pre-conditions
- How many iterations?

Rice's Theorem

[…] there exists no automatic method that decides with generality non-trivial questions on the black-box behavior of computer programs. – Wikipedia

Rice's Theorem

- You can *never* have enough assertions

Rice's Theorem

- You can *never* have enough assertions
- All you can do is improve "confidence"
- (for some naïve definition thereof)

Rice's Theorem

- You can *never* have enough assertions
- All you can do is improve "confidence"
  - (for some naïve definition thereof)
- Black box testing is a farce

Rice's Theorem

- You can *never* have enough assertions
- All you can do is improve "confidence"
  - (for some naïve definition thereof)
- Black box testing is a farce
- Partial function indistinguishable from total

```scala
def addGood(x: Int, y: Int) = x + y


// egad!
def addBad(x: Int, y: Int) = (x, y) match {
  case (0, 0) => 0
  case (0, 1) => 1
  case (1, 0) => 1
  case (2, 3) => 5
  case (100, 500) => 600
}
```

Black Box Testing

- Do we just…enumerate cases?

Black Box Testing

- Do we just…enumerate cases?
- When do we stop? (hint: never!)

Black Box Testing

- Do we just…enumerate cases?
- When do we stop? (hint: never!)
- Can we assume edge cases are predictable?

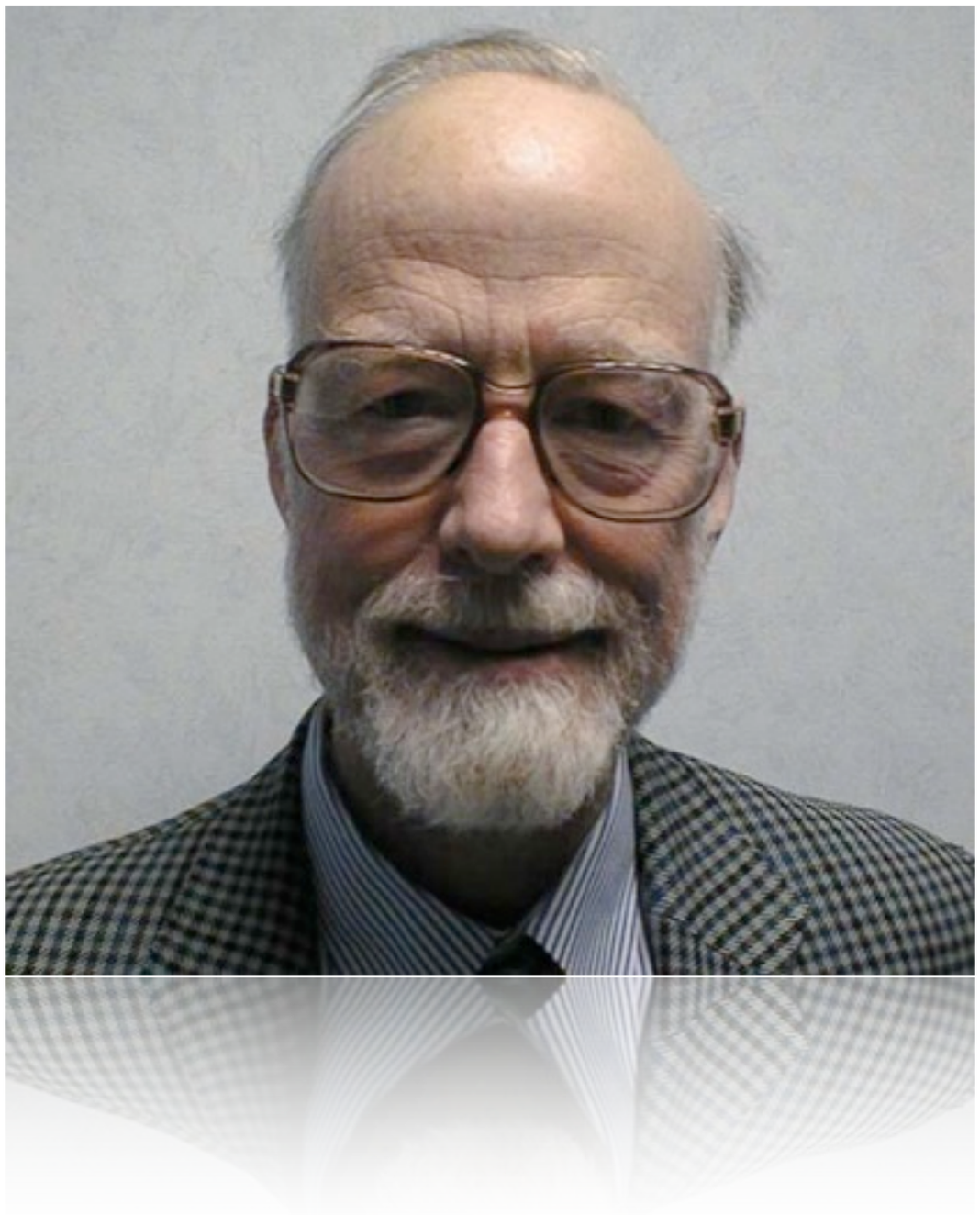
Black Box Testing

- Do we just…enumerate cases?
- When do we stop? (hint: never!)
- Can we assume edge cases are predictable?
- What *can* we assume?

Black Box Testing

- Do we just…enumerate cases?
- When do we stop? (hint: never!)
- Can we assume edge cases are predictable?
- What *can* we assume?
- It's all undecidable, let's go shopping

In the beginning…



An Axiomatic Basis for Computer Programming

Partial Correctness

$$P\,\{Q\}\,R$$

Built on two ideas

- Axioms
  - The foundation of proofs in our system (assignment, precendence, identity, etc)
- Inference
  - Higher order ideas that use combinations of axioms to draw conclusions

It drove us to think about consequence in program execution

Be Careful!

$$\neg\neg\exists_x \forall_y \ (y \leq x)$$

Computers are finite!

$$\forall_x \ (x \leq \max)$$

Real Example

Commutative law

$$x + y = y + x$$

$$x \times y = y \times x$$

Rules of Consequence

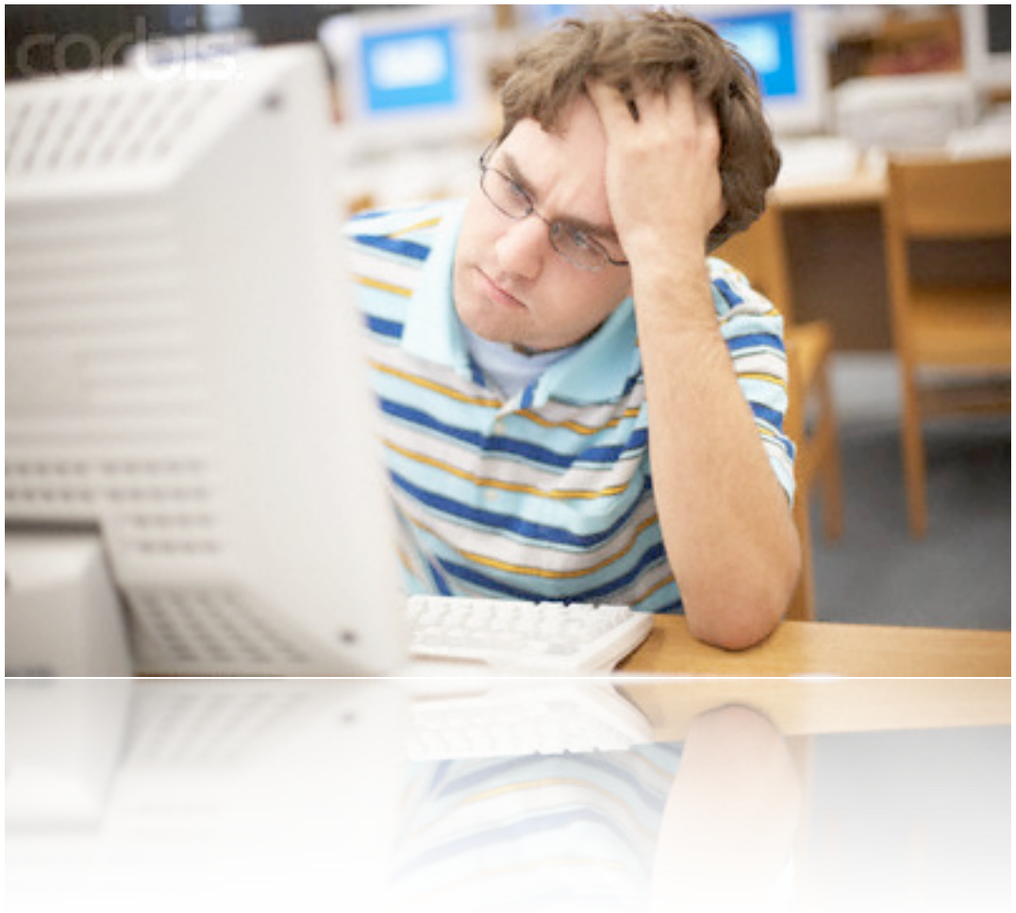
$$(\vdash P\,\{Q\}\,R) \wedge (\vdash R \supset S) \Rightarrow (\vdash P\,\{Q\}\,S)$$

$$(\vdash P\,\{Q\}\,R) \wedge (\vdash S \supset P) \Rightarrow (\vdash S\,\{Q\}\,R)$$

Our problems are solved!

Our problems are solved!

Type Systems



Type Systems

- Mildly controversial…

Type Systems

- Mildly controversial…
- Formal verification…that you can't disable!

Type Systems

- Mildly controversial…
- Formal verification…that you can't disable!
- Part of the language (inextricably)
  - (yes, that is a definitional requirement)

Type Systems

- Mildly controversial…
- Formal verification…that you can't disable!
- Part of the language (inextricably)
  - (yes, that is a definitional requirement)
- Model correctness as consistency

$$\frac{\Gamma \vdash t : T}{t \Rightarrow t' \ \vee \ t \text{ is value}} \textsc{Progress}$$

$$\frac{\Gamma \vdash t : T \qquad t \Rightarrow t'}{\Gamma \vdash t' : T} \textsc{Preservation}$$

```scala
case class Miles(value: Int)
case class Kilometers(value: Int)


def launchMarsMission(distance: Miles) = {
  val kph = distance.value / (6 * 30 * 24)
  increaseSpeedToKPH(kph)
}
```

```scala
trait Forall[CC[_]] {
  def apply[A]: CC[A]
}

trait ST[S, A] {
  def flatMap[B](f: A => ST[S, B]): ST[S, B]
}

object ST {
  def run[A](
    f: Forall[({ type λ[S] = ST[S, A] => A })#λ]): A
}
```

Correct by Construction

- Incorrect algorithms fail to compile

Call me Ishmael…

Correct by Construction

- Incorrect algorithms fail to compile
- Bugs are a thing of the past

Call me Ishmael…

Correct by Construction

- Incorrect algorithms fail to compile
- Bugs are a thing of the past
- Programmers herald new era

Call me Ishmael…

Correct by Construction

- Incorrect algorithms fail to compile
- Bugs are a thing of the past
- Programmers herald new era
- Profit?

Call me Ishmael…

Curry-Howard

- Types are logical propositions
- Values are proofs of propositions
- Type checking is testing logical validity
- A system's *consistency* is what is checked

```haskell
-- Thm. Integer implies Walrus

theorem :: Integer -> Walrus
theorem a = theorem a
```

```haskell
-- Thm. everything implies Walrus

theorem :: forall a . a -> Walrus
theorem a = theorem a
```

```
-- Thm. everything is true!

theorem :: forall a . a
theorem = theorem
```

Gödel

- Self-referentiality is a problem

Gödel

- Self-referentiality is a problem
  - Hello, Strange Loop!

Gödel

- Self-referentiality is a problem
  - Hello, Strange Loop!
- Incomplete *or* inconsistent

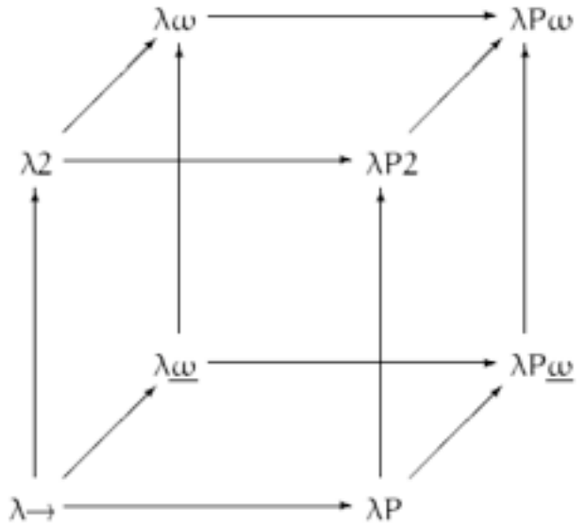
Gödel

- Self-referentiality is a problem
  - Hello, Strange Loop!
- Incomplete *or* inconsistent
- Some valid programs cannot be typed

Gödel

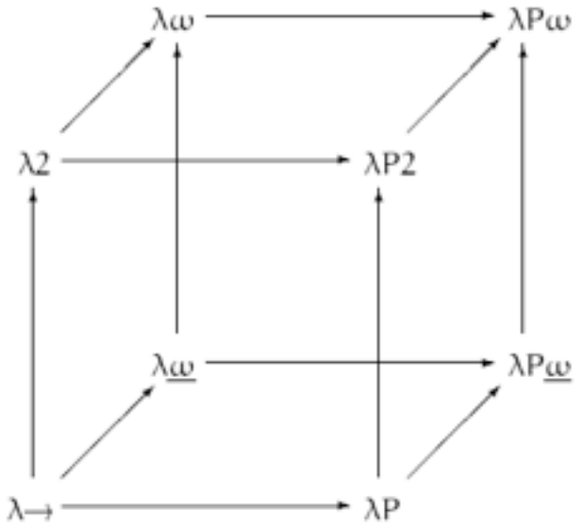
- Self-referentiality is a problem
  - Hello, Strange Loop!
- Incomplete *or* inconsistent
- Some valid programs cannot be typed
- Strong normalization helps
  - (avoids the need for recursive types!)

System F



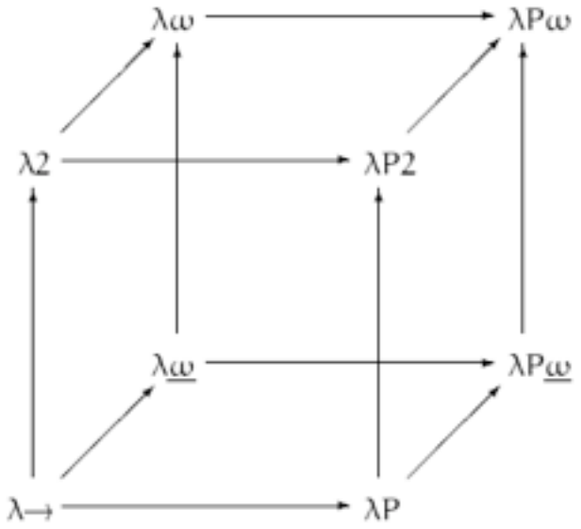
- Strongly normalizing λ calculus

System F



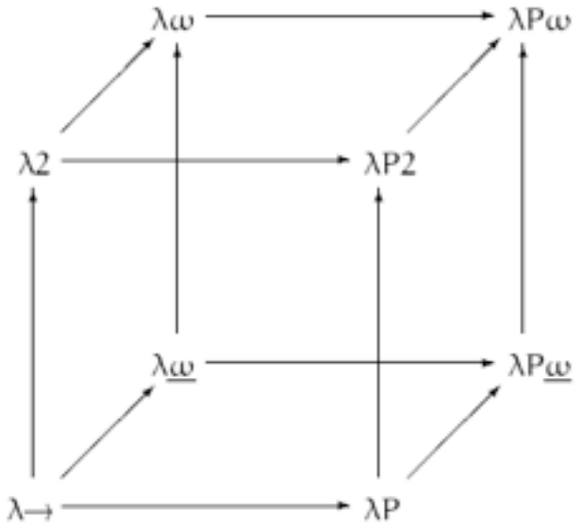
- Strongly normalizing λ calculus
- Statically typed, and complete!

System F



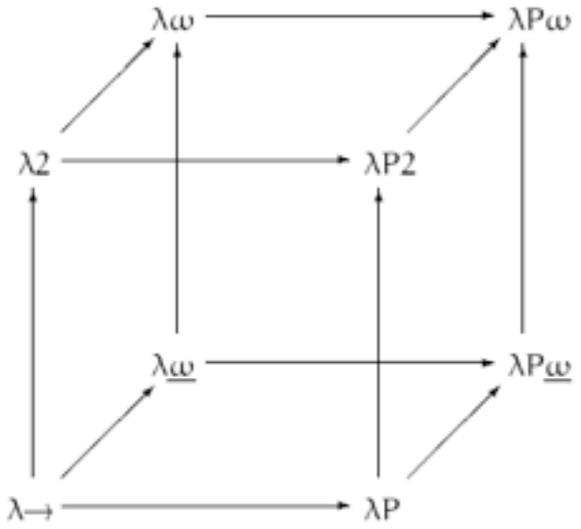
- Strongly normalizing λ calculus
- Statically typed, and complete!
- All valid programs can be typed

System F



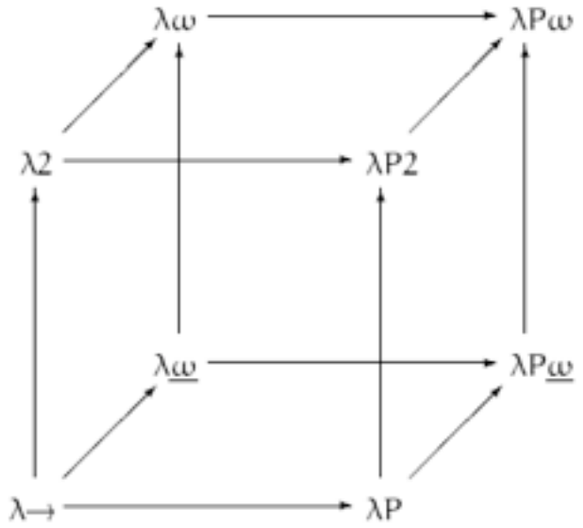
- Strongly normalizing λ calculus
- Statically typed, and complete!
  - All valid programs can be typed
  - No invalid programs can be typed

System F

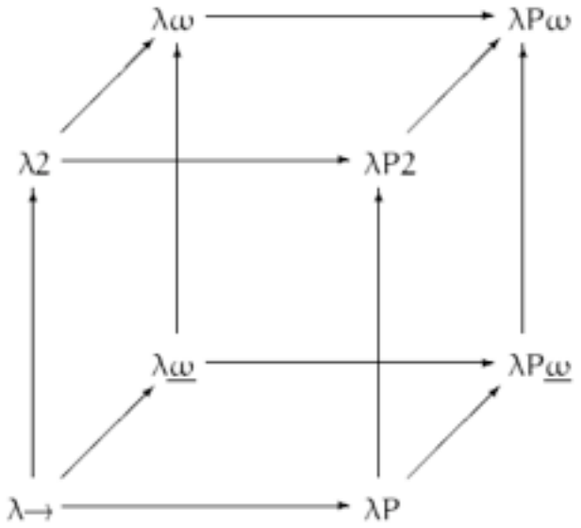


- Strongly normalizing λ calculus
- Statically typed, and complete!
  - All valid programs can be typed
  - No invalid programs can be typed
- Immensely expressive (lists, numbers, etc)

System F

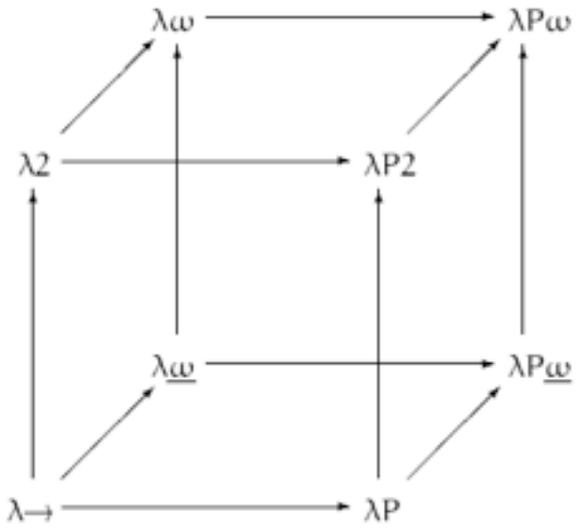


System F



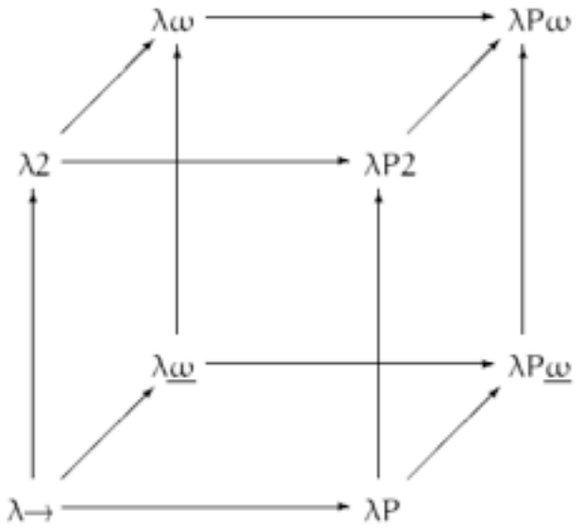
- Not expressive enough

System F



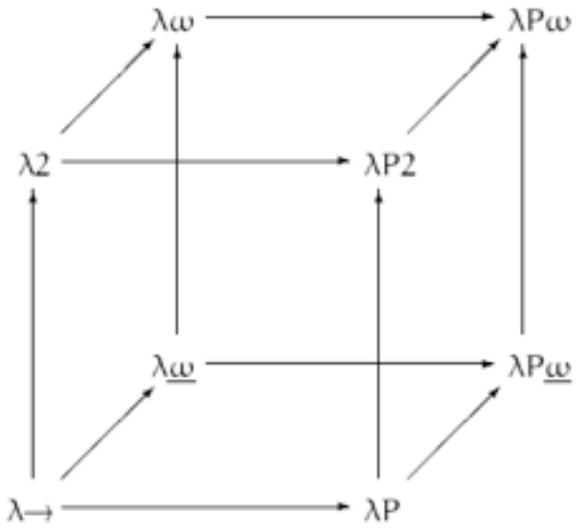
- Not expressive enough
- Cannot bootstrap its own compiler

System F



- Not expressive enough
- Cannot bootstrap its own compiler
- Many valid programs have no encoding

System F



- Not expressive enough
- Cannot bootstrap its own compiler
- Many valid programs have no encoding
- Not Turing-complete! (trivially so)

Static Typing

- Can't even answer some basic questions

Static Typing

- Can't even answer some basic questions
- Fails to definitively answer *any* questions
- …for Turing-complete calculi

Static Typing

- Can't even answer some basic questions
- Fails to definitively answer *any* questions
  - …for Turing-complete calculi
- Arms race of inconvenience

Static Typing

- Can't even answer some basic questions
- Fails to definitively answer *any* questions
  - …for Turing-complete calculi
- Arms race of inconvenience
- Guardrails on a sidewalk

Contract

I will: finish my lunch on time and not dawdle.

My teacher will: give me a fish sticker.

Then, I will get: to pick out a goldfish for the class when I have 10 stickers.

Signed: Raquld
me
Ms Lambertino
my teacher
May 8
today

```haskell
add :: Integer -> Integer -> Integer
add x y = x + y
```

```haskell
add :: Integer -> Integer -> Integer
add x y = x + y
```

```haskell
add :: Prime -> Prime -> Integer
add x y = x + y
```

```
(define (add x y)
  (+ x y))
```

```
(define (add x y)
  (+ x y))


(module strangeloop racket
  (provide
   (contract-out
    [add (->* (prime? prime?)
              ()
              integer?)])))
```

```
(define (divisible? i m)
  (cond
   [(= i 1) true]
   [else (cond
           [(= (remainder m i) 0) false]
           [else (divisible? (sub1 i) m)])]))

(define (prime? n)
  (divisible? (sub1 n) n))
```

```
(require 'strangeloop)
(add 7 7)
;; > 14
```

```
(add 4 7)
;; > add: contract violation
;;     expected: prime?, given: 4
;;     in: the 1st argument of
;;           (-> prime? prime?
;;               integer?)
;;     contract from: strangeloop
;;     blaming: top-level
;;     at: stdin::79-82
;;     context...:
```

But like everything else, it suffers from deficiencies...

Before Contract Application

```
(time (add 492876847 492876847))
;; > cpu time: 0 real time: 0 gc time: 0
;; 985753694
```

Before Contract Application

```
(time (add 492876847 492876847))
;; > cpu time: 0 real time: 0 gc time: 0
;; 985753694
```

After Contract Application

```
(time (add 492876847 492876847))
;; > cpu time: 15963 real time: 15995 gc time: 0
;; 985753694
```

Runtime issues can make contracts a non-starter for production code

It's All Bunk

- Black-Box verification is insufficient

It's All Bunk

- Black-Box verification is insufficient
- White-Box verification is undecidable

It's All Bunk

- Black-Box verification is insufficient
- White-Box verification is undecidable
  - …in general!

It's All Bunk

- Black-Box verification is insufficient
- White-Box verification is undecidable
  - …in general!
- Constrain the domain

Functional Style

- Write your code in a functional style
- It is constraining…that's the point!
- FP is easier to reason about
  - (both formally and informally)
- Easier to test, *and* easier to verify

Programming:
The Good Parts

References

- From System F to Typed Assembly Language
- An Axiomatic Basis for Computer Programming
- Hoare Logic and Auxiliary Variables
- The Underlying Logic of Hoare Logic
- Applying Design by Contract
- Proof Carrying Code
- Uniform Proofs as a Foundation for Logic Programming
- Safe Kernel Extensions Without Run-Time Checking