```
#if THRIFTY
  save! cake
#elif HUNGRY
  eat! cake
#endif
```

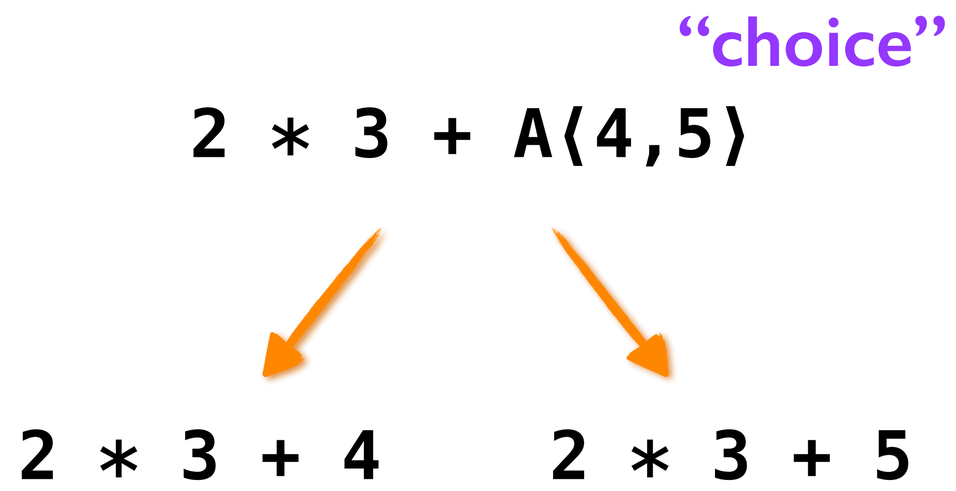
Toward a variational programming language

Eric Walkingshaw
Oregon State University

What is variational programming?

Computing with *explicit* variation in code and data

```
2 * 3 +
#if A
  4
#else
  5
#endif
```



2 * 3 + A⟨4,5⟩
↦ 6 + A⟨4,5⟩
↦ A⟨6+4,6+5⟩
↦ A⟨10,6+5⟩
↦ A⟨10,11⟩

Variational programming by example

```
A⟨2,3⟩ + A⟨10,20⟩
↦ A⟨12,23⟩

A⟨True,3⟩
: A⟨Bool,Int⟩
```

“choice type”

```
A⟨succ,even⟩
: A⟨Int -> Int,Int -> Bool⟩
≡ Int -> A⟨Int,Bool⟩
```

Variational programming by example

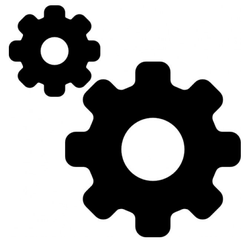
```
A⟨2,3⟩ + B⟨10,20⟩
↦ A⟨B⟨12,22⟩,B⟨13,23⟩⟩

vsum (A⟨2,3⟩ + B⟨10,20⟩)
↦ 70

vmax (A⟨2,3⟩ + B⟨10,20⟩)
↦ 23 @ [A.R,B.R]
```

Application: validating highly configurable systems

(static variation)

```
class Buffer {
  int buff = 0;
#ifdef UndoOne
  int back = 0;
#endif
#ifdef UndoMany
  Stack stack =
    new Stack();
#endif
  int get() {
    return buff;
  }
  void set(int x) {
#ifdef Logging
    log(buff+"->"+x);
#endif
#ifdef UndoOne
    back = buff;
#endif
#ifdef UndoMany
    stack.push(buff);
#endif
    buff = x;
  }
#ifdef UndoOne
  void undo() {
#ifdef Logging
    log(back+"<-"+bug);
#endif
    buff = back;
  }
#endif
#ifdef UndoMany
  void undo() {
#ifdef Logging
    log(stack.peek()
        +"<-"+buff);
#endif
    buff =
      stack.pop();
  }
#endif
}
```

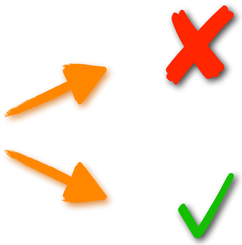
1. configure



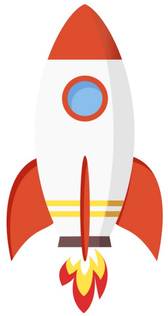
```
class Buffer {
  int buff = 0;
  int get() {
    return buff;
  }
  void set(int x) {
    buff = x;
  }
}
```

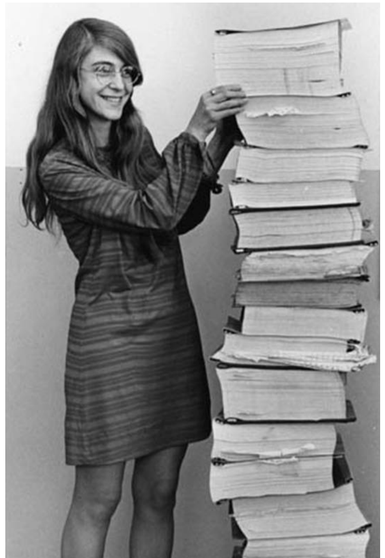
2. validate

program analysis



3. deploy

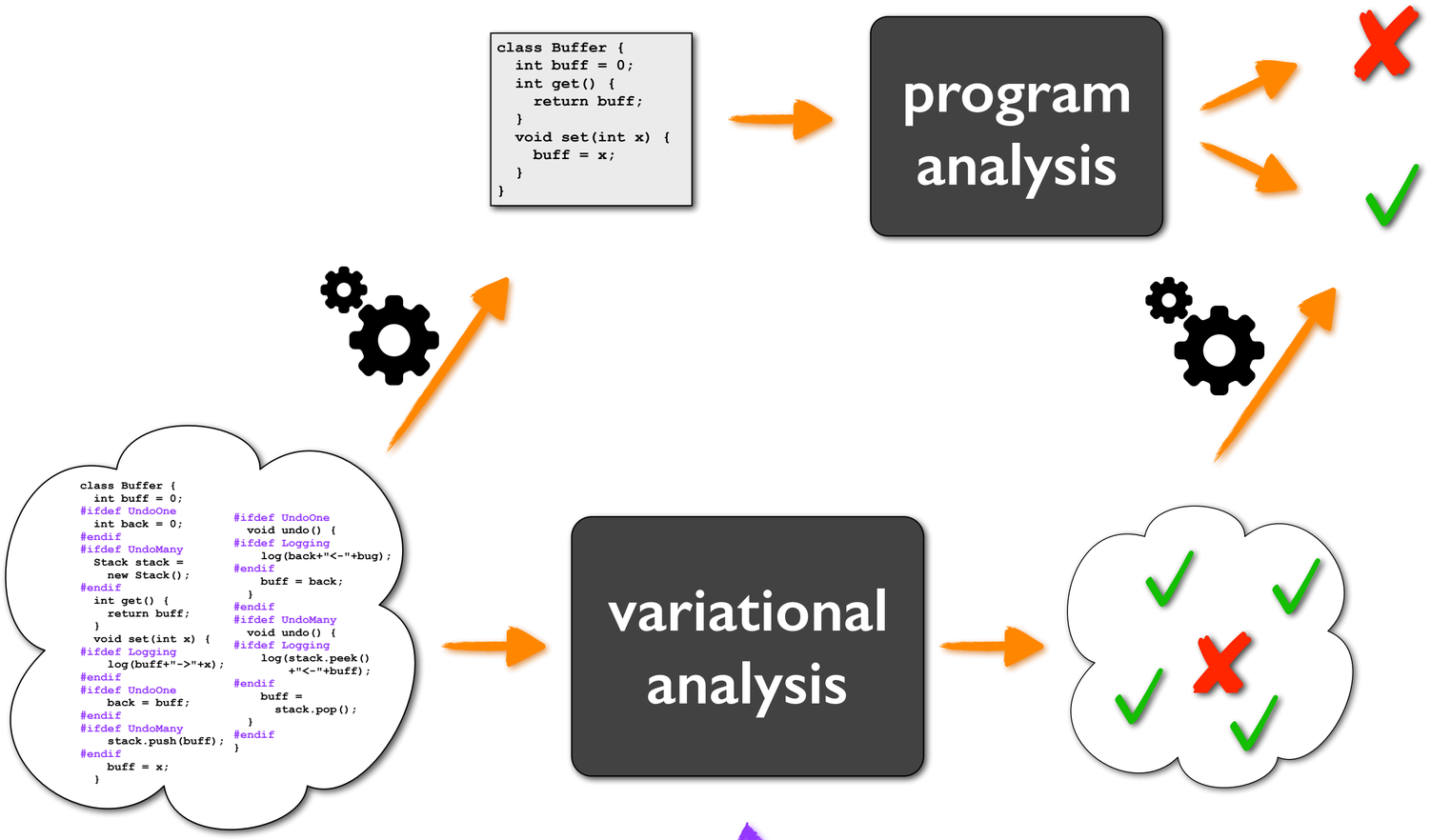




This process finds errors too late!

Too many configs to check them all

Application: validating highly configurable systems



Variation in input is **preserved** in output

variational programming inside!

Application: information-flow security "faceted execution"

protected by policy **P**

```
def test(secret):
  result = 5
  ok = check(secret)
  if ok then:
    result = result * 2
  return result
```

```
secret = P⟨"sesame",⊥⟩
result = 5
ok = P⟨true,⊥⟩

result = P⟨10,5⟩
```

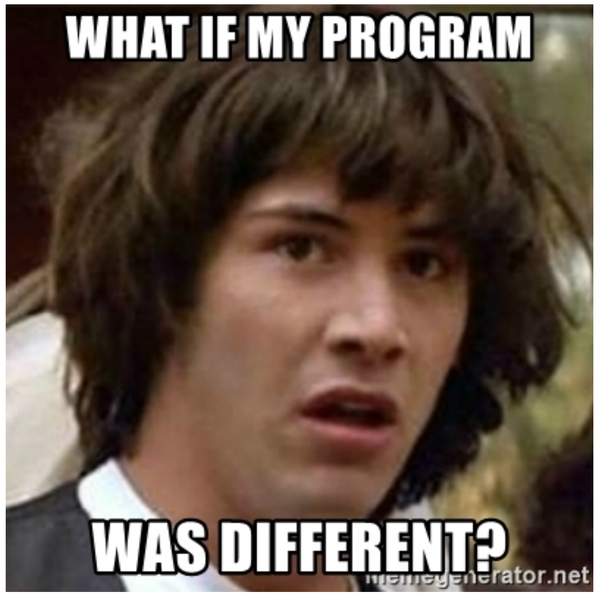
```
secret = P⟨"wrong",⊥⟩
result = 5
ok = P⟨false,⊥⟩

result = 5
```

if you can't see the secret you better not learn anything here!

non-interference
≈
variation preservation

Application: speculative program analyses

Idea: use variation to explore *hypothetical* scenarios



Speculative analysis for error location

```
fold f z []     = [z]
fold f z (h:t) = fold f (f z h) t

flip f x y = f y x

reverse = fold (flip (:)) []

palindrome xs = reverse xs == xs
```

error is in base case of fold
… but fold type checks!

use of fold also type checks!

error finally detected

- **Occurs check: cannot construct the infinite type: a ~ [a]**
  Expected type: [[a]]
  Actual type: [a]
- **In the second argument of ‘(==)’, namely ‘xs’**
  In the expression: reverse xs == xs

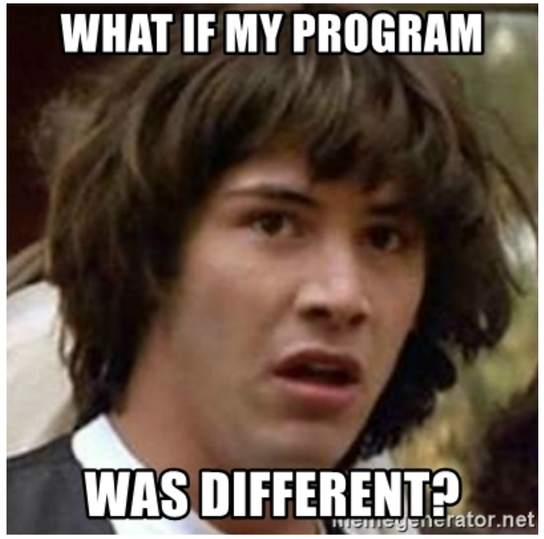
Speculative analysis for error location

"counterfactual typing"

Problem: locating the cause of a type error is hard

- type inference *commits* too early
- a successfully inferred type could be wrong!

Solution:



1. Error-tolerant variational type inference where for *every* subexpression

   `e : T` → `e : d⟨T,a⟩` `d & a` *are fresh*

2. Search output variational type for
   - non-error type
   - as few right selections as possible

Chen and Erwig
POPL 2014, JFP 2018

Speculative analysis for migrating gradual types

```
def f(mode:bool, x):
  if mode:
    return even(x)
  else:
    return not(x)
```

Gradual typing
mix static and dynamic types
in the same program

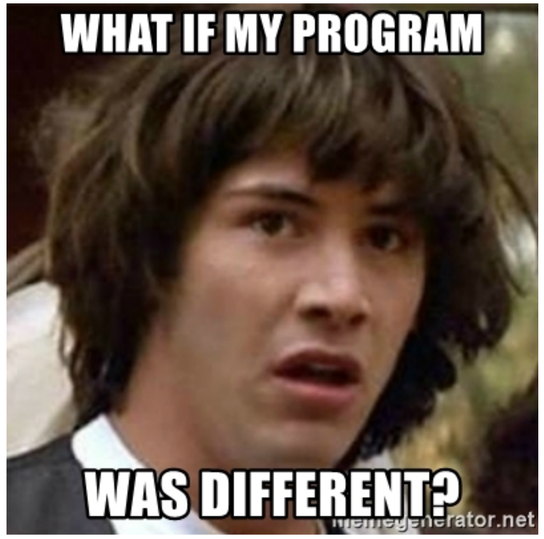
Migration challenges: (adding/removing annotations)

- mutually exclusive annotations
- local type-safety maxima
- potential for extreme performance degradation

Speculative analysis for migrating gradual types

Problem: migrating gradual types is perilous
and exploration by trial-and-error is infeasible

Solution:

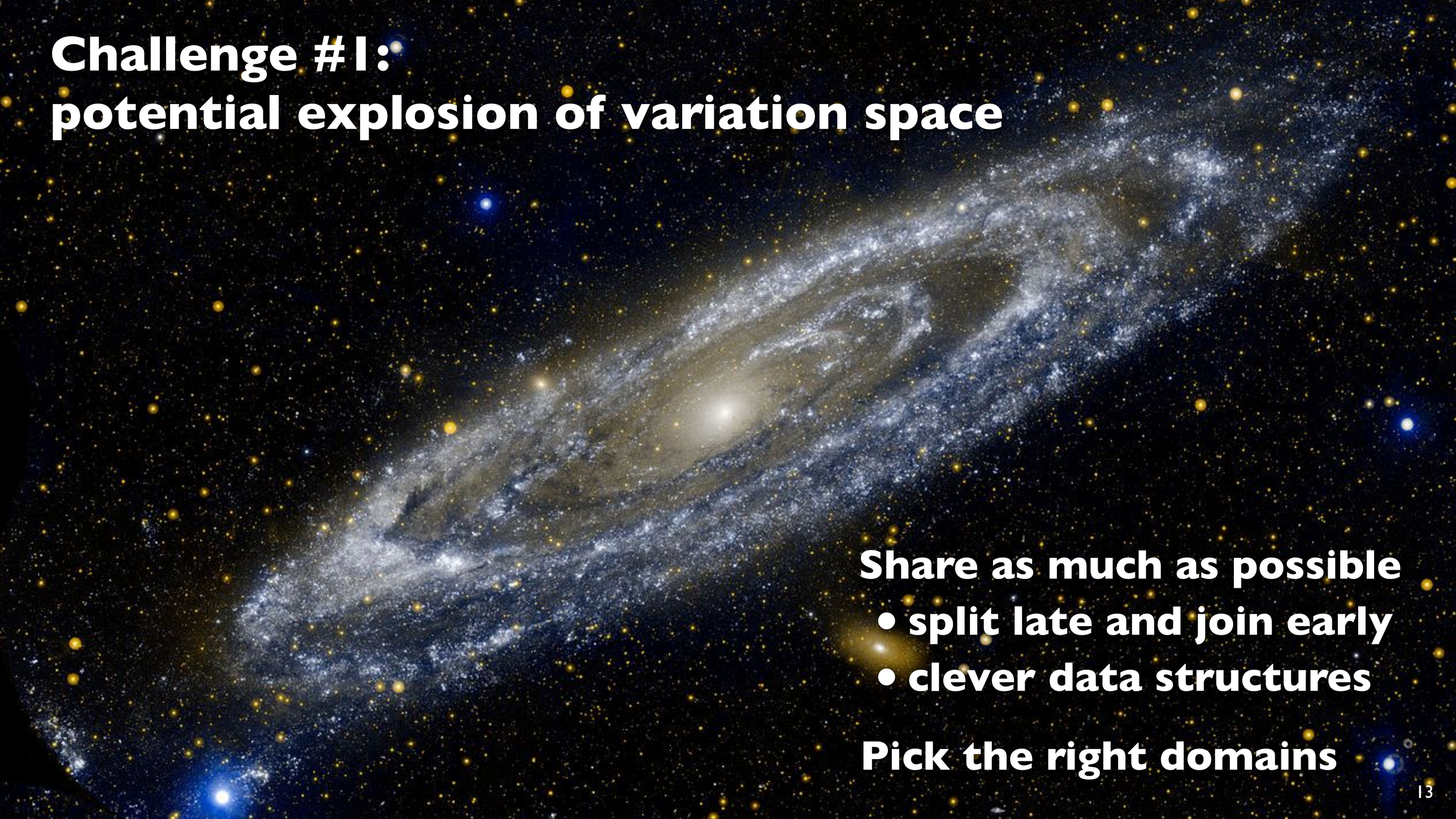


1. Every (unannotated) parameter is initially `d⟨a,?⟩`
   `d` & `a` *are fresh* `?` *is the dynamic type*

2. Variational gradual type inference + cost analysis
   output = summary of *all possible* migrations

3. Filter/search variational output
   most static = fewest right selections
   cheapest = lowest cost

prototype in Reticulated Python!



Challenge #1: potential explosion of variation space

Share as much as possible

- **split late and join early**
- **clever data structures**

Pick the right domains

Challenge #2: variation is highly cross-cutting

Application level and language level

Variation in structured data

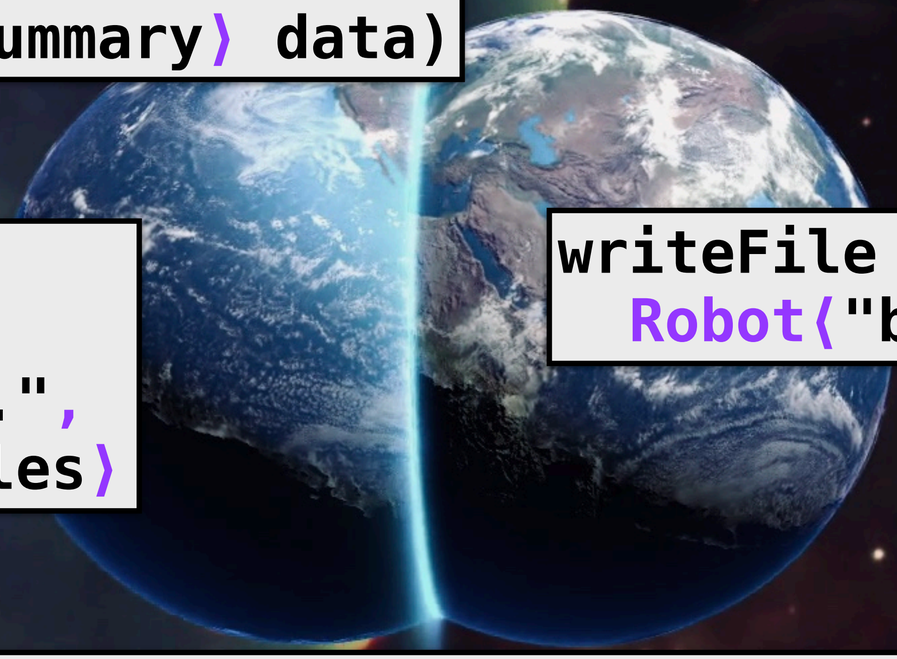
```
A⟨[1,2,3],[1,2,4,5]⟩
≡ [1,2,A⟨3,4⟩,A⟨5,_⟩] ??
```

... need *variational data structures*

Challenge #3: how to handle external effects?

```
print (Verbose⟨show,summary⟩ data)
```

```
countdown >>
  Simulator⟨
    print "Phew.",
    launchMissiles⟩
```



```
writeFile "shopping.txt"
  Robot⟨"batteries","bananas"⟩
```

```
runQuery $
  select `from` dosing `restrict`
    Fluarix⟨volume > 500,
      mass > Albendazole⟨1000,2000⟩⟩
```

Why a variational programming language?

Goals:

- hard to do efficiently at library level
- advance and promote variational programming as a broadly applicable strategy
- make what we know practically usable by others
- tackle the unsolved challenges (e.g. effects)

Effects + variation

Can create generic "variation aware" interfaces to some effects

- lots of work, not always possible
- some configurations will need something special anyway

Idea: give the *programmer* the tools to deal with this
via algebraic effects and handlers

prototype in Eff!

Generic variational file handler

```
vFileHandler = handler
  read! file k ->
    k (fromIfDef (get! vctx) file)
  write! file str k ->
    toIfDef (get! vctx) file str; k ()
```

```
write! "shopping.txt"
  Robot⟨"batteries", "bananas"⟩

Robot⟨write! "todo.txt" "revolution", ()⟩


> "buy " ++ read! "shopping.txt"
Robot⟨"buy batteries", "buy bananas"⟩

> Robot⟨"robowax", read! "shopping.txt"⟩
Robot⟨"robowax", "bananas"⟩
```

```
#if Robot
batteries
#else
bananas
#endif
```

shopping.txt

```
#if Robot
revolution
#endif
```

todo.txt

Specialized variational file handler

```
vEncryptedFileHandler = handler
  read! f k ->
    Encrypt⟨k (decrypt (get! key) (read! f)),
      read! f k⟩
  write! f s k ->
    Encrypt⟨k (encrypt (get! key) (write! f l)),
      write! f l k⟩
```

Toward a variational programming language

Planned features:

- choices + variation-preserving typing and execution
- variational algebraic data types
- variant querying and aggregation
- reify / reflect on variational data
- extensible effect system