Calculating Programs

Lecture 1: Fold/Unfold Calculation

MGS Spring School 2019

What is Program Calculation?

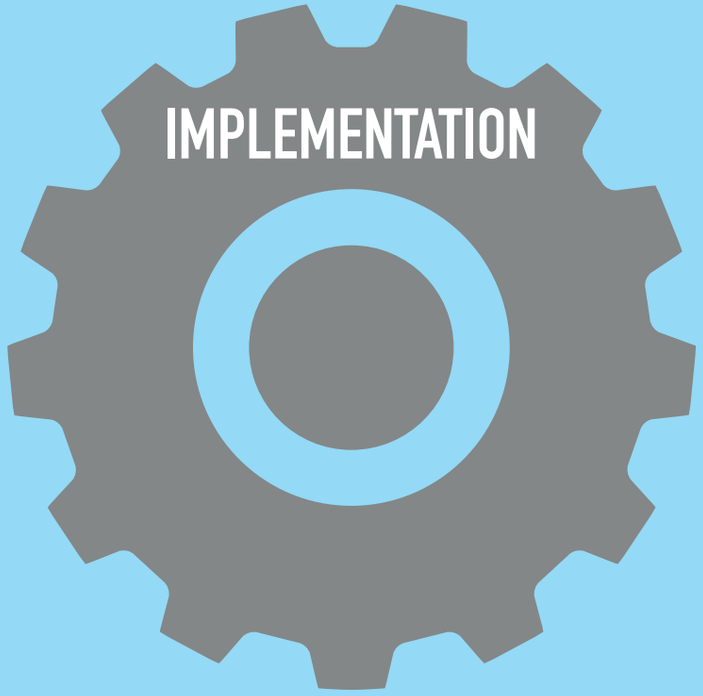
This is what you were taught to do:

What is Program Calculation?

This is what you were taught to do:

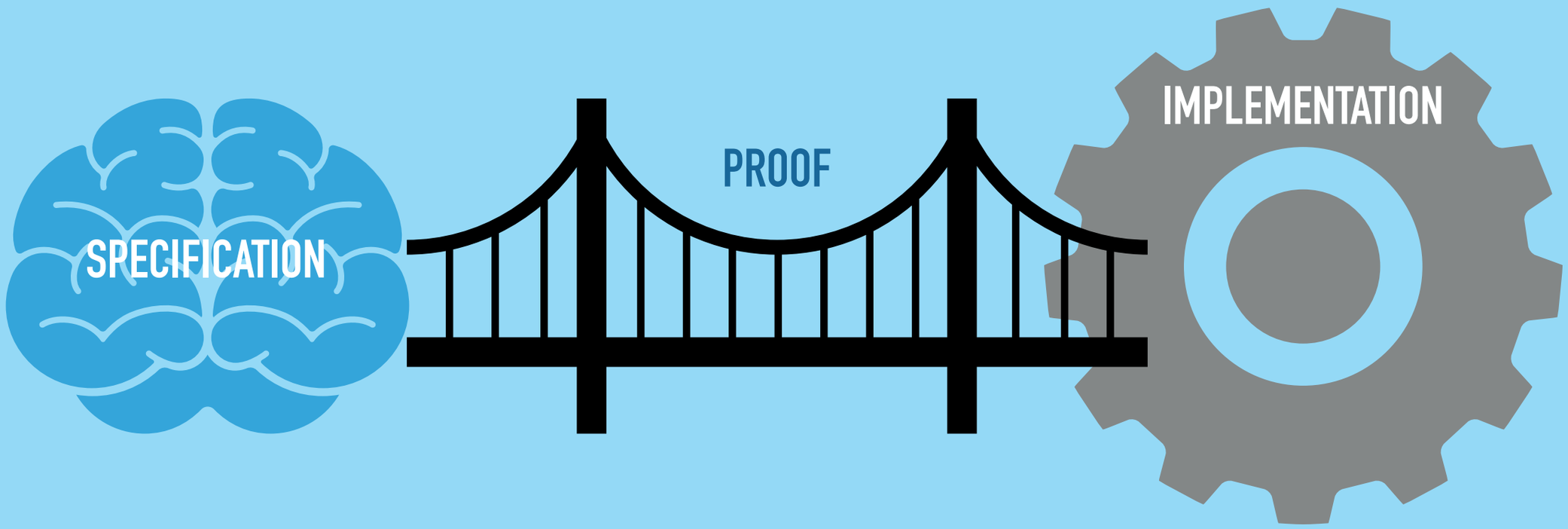
What is Program Calculation?

This is what you were taught to do:



What is Program Calculation?

This is what you were taught to do:

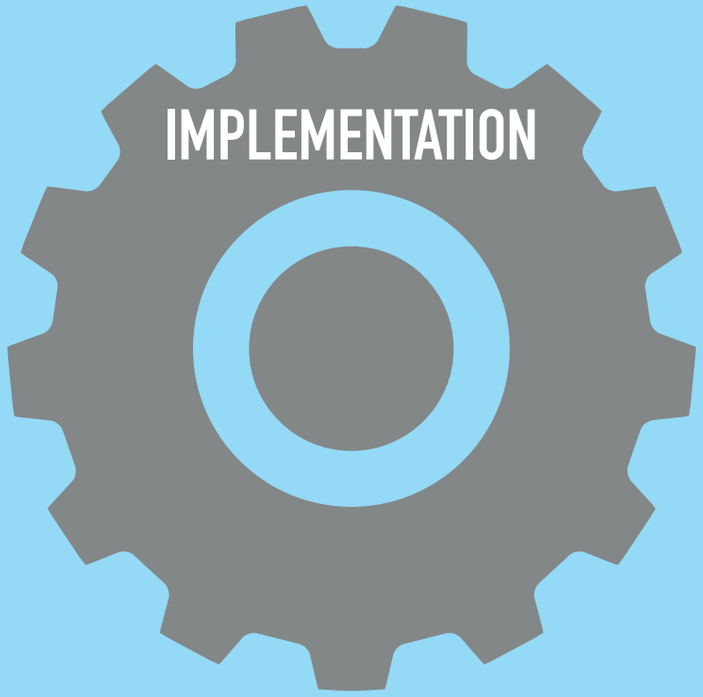


What is Program Calculation?

This is what you probably *actually* do:

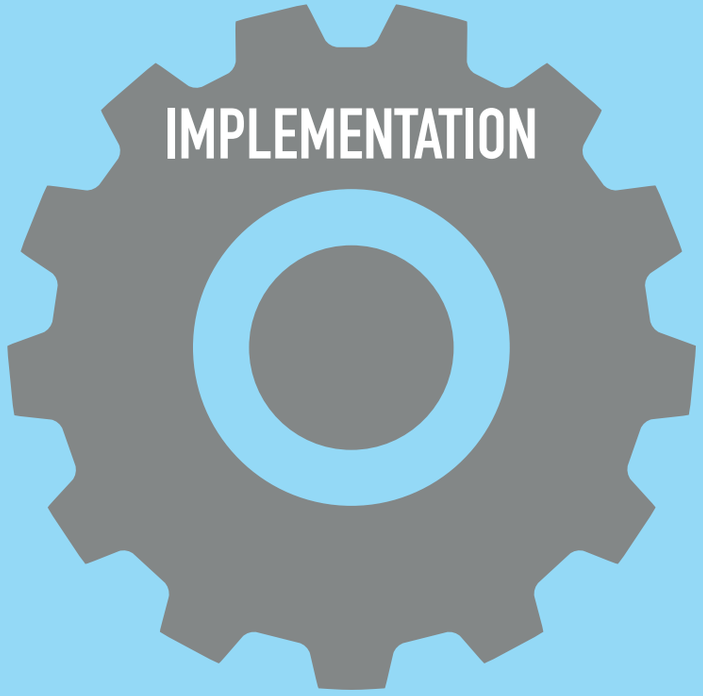
What is Program Calculation?

This is what you probably *actually* do:



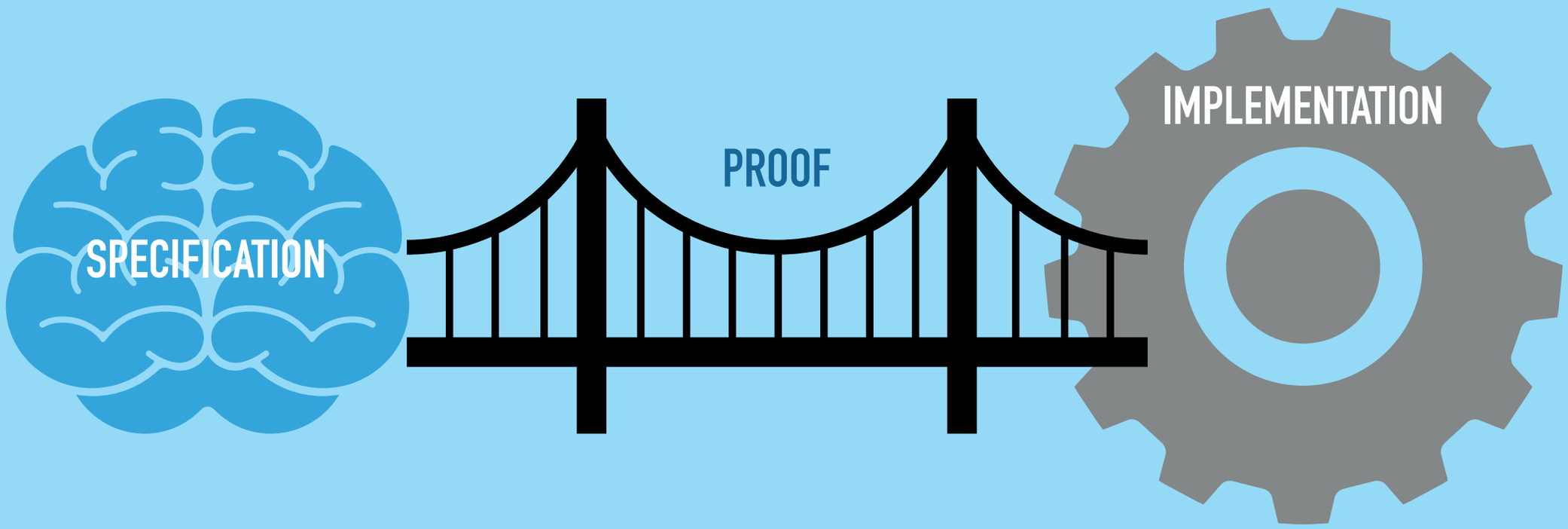
What is Program Calculation?

This is what you probably *actually* do:



What is Program Calculation?

This is what you probably *actually* do:



What is Program Calculation?

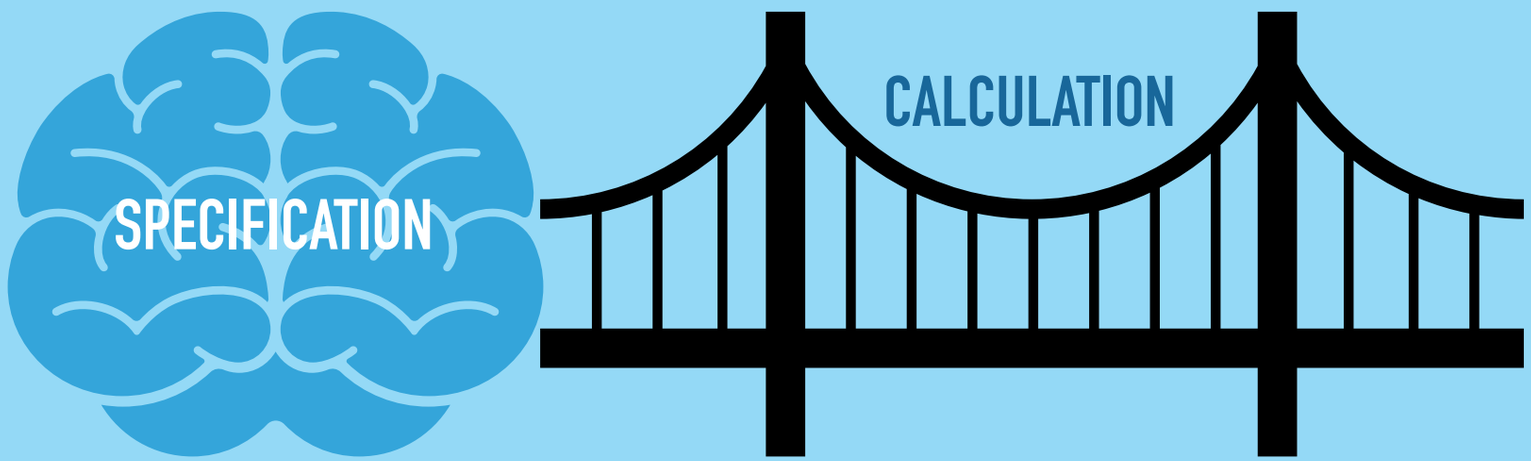
…and *this* is program calculation:

What is Program Calculation?

…and *this* is program calculation:

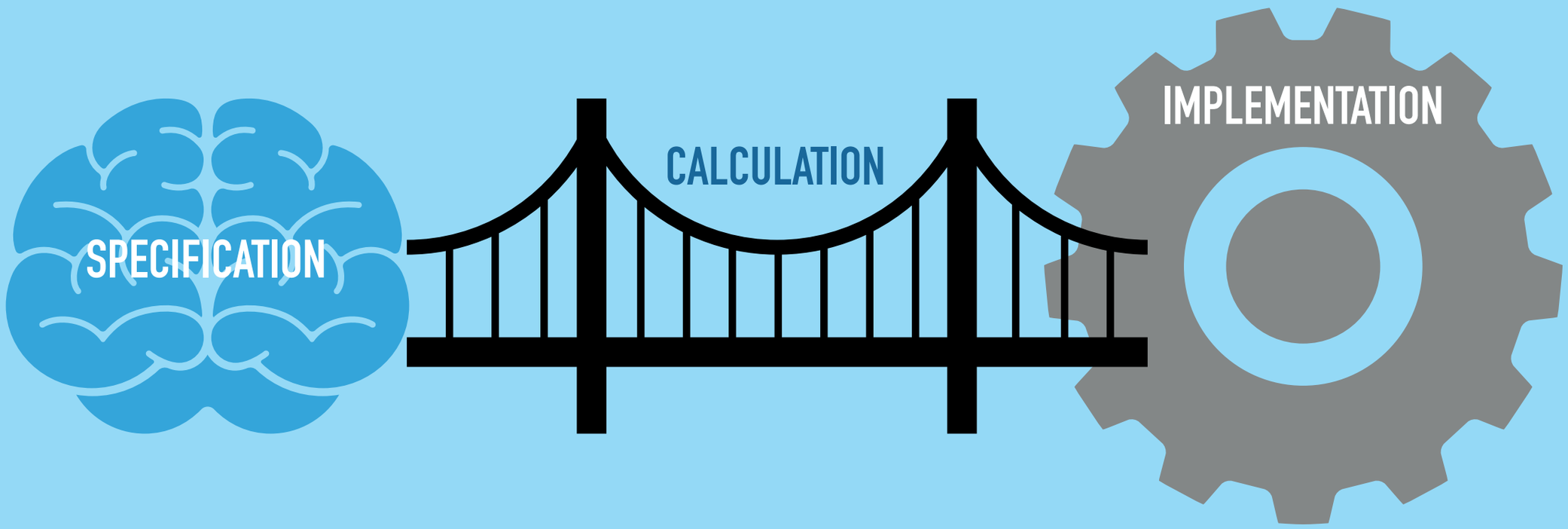
What is Program Calculation?

…and *this* is program calculation:



What is Program Calculation?

…and *this* is program calculation:



What is Program Calculation?

What is Program Calculation?

- A collection of approaches to deriving a program from its specification

What is Program Calculation?

- A collection of approaches to deriving a program from its specification
- These approaches typically provide *correctness-by-construction*, meaning that our program will not have bugs…

What is Program Calculation?

- A collection of approaches to deriving a program from its specification
- These approaches typically provide *correctness-by-construction*, meaning that our program will not have bugs…
  - …so long as we didn't make any mistakes in the calculation

What is Program Calculation?

- A collection of approaches to deriving a program from its specification
- These approaches typically provide *correctness-by-construction*, meaning that our program will not have bugs…
  - …so long as we didn't make any mistakes in the calculation
- In this course, I will focus on three of these approaches:

What is Program Calculation?

- A collection of approaches to deriving a program from its specification
- These approaches typically provide *correctness-by-construction*, meaning that our program will not have bugs…
  - …so long as we didn't make any mistakes in the calculation
- In this course, I will focus on three of these approaches:
  1. Fold/unfold calculations

What is Program Calculation?

- A collection of approaches to deriving a program from its specification
- These approaches typically provide *correctness-by-construction*, meaning that our program will not have bugs…
  - …so long as we didn't make any mistakes in the calculation
- In this course, I will focus on three of these approaches:
  1. Fold/unfold calculations
  2. The Bird-Meertens formalism, a.k.a. *Squiggol*

What is Program Calculation?

- A collection of approaches to deriving a program from its specification
- These approaches typically provide *correctness-by-construction*, meaning that our program will not have bugs…
  - …so long as we didn't make any mistakes in the calculation
- In this course, I will focus on three of these approaches:
  1. Fold/unfold calculations
  2. The Bird-Meertens formalism, a.k.a. *Squiggol*
  3. Fixed-point calculus

What is Program Calculation?

- A collection of approaches to deriving a program from its specification
- These approaches typically provide *correctness-by-construction*, meaning that our program will not have bugs…
  - …so long as we didn't make any mistakes in the calculation
- In this course, I will focus on three of these approaches:
  1. Fold/unfold calculations
  2. The Bird-Meertens formalism, a.k.a. *Squiggol*
  3. Fixed-point calculus
- The final lecture will go through a long worked example

A Note on Notation...

A Note on Notation...

- In this course, I will write programs in a Haskell-like notation:

```
fac 0     = 1
fac (n+1) = (n+1) * fac n

[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

A Note on Notation…

- In this course, I will write programs in a Haskell-like notation:

```
fac 0     = 1
fac (n+1) = (n+1) * fac n

[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

- Function application requires no brackets and associates to the left, functions defined by pattern-matching on the input, operators can be infix…

Fold/Unfold Calculation

Fold/Unfold Calculation

- Nothing to do with cata-/anamorphisms

Fold/Unfold Calculation

- Nothing to do with cata-/anamorphisms
- Treats a program as a set of definitions, *folding* and *unfolding* those definitions step-by-step

Fold/Unfold Calculation

- Nothing to do with cata-/anamorphisms
- Treats a program as a set of definitions, *folding* and *unfolding* those definitions step-by-step
- Preserves *partial* correctness

Fold/Unfold Calculation

- Nothing to do with cata-/anamorphisms
- Treats a program as a set of definitions, *folding* and *unfolding* those definitions step-by-step
- Preserves *partial* correctness
  - "If both the initial and final program terminate, they will have the same result"

Fold/Unfold Calculation

- Nothing to do with cata-/anamorphisms
- Treats a program as a set of definitions, *folding* and *unfolding* those definitions step-by-step
- Preserves *partial* correctness
  - "If both the initial and final program terminate, they will have the same result"
  - Might lose *total* correctness, which requires same termination behaviour

Fold/Unfold Calculation

- Nothing to do with cata-/anamorphisms
- Treats a program as a set of definitions, *folding* and *unfolding* those definitions step-by-step
- Preserves *partial* correctness
  - "If both the initial and final program terminate, they will have the same result"
  - Might lose *total* correctness, which requires same termination behaviour
- See *A Transformation System for Developing Recursive Programs* — Burstall & Darlington, 1977

Example: Factorial

Example: Factorial

- Recall the factorial function:

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
```

Example: Factorial

- Recall the factorial function:

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
```

- Suppose we want a new program that does the same thing, but using an accumulating parameter and tail-recursion…

Example: Factorial

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
```

Example: Factorial

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
fac' a n = a * fac n
```

Example: Factorial

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
fac' a n = a * fac n
fac' a 0 = a * fac 0
```

Example: Factorial

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
fac' a n = a * fac n
fac' a 0 = a * fac 0
         = a * 1
```

Example: Factorial

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
fac' a n = a * fac n
fac' a 0 = a * fac 0
         = a * 1
         = a
```

Example: Factorial

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
fac' a n = a * fac n
fac' a 0 = a * fac 0
         = a * 1
         = a
fac' a (n+1) = a * fac (n+1)
```

Example: Factorial

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
fac' a n = a * fac n
fac' a 0 = a * fac 0
         = a * 1
         = a
fac' a (n+1) = a * fac (n+1)
             = a * ((n+1) * fac n)
```

Example: Factorial

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
fac' a n = a * fac n
fac' a 0 = a * fac 0
         = a * 1
         = a
fac' a (n+1) = a * fac (n+1)
             = a * ((n+1) * fac n)
             = (a * (n+1)) * fac n
```

Example: Factorial

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
fac' a n = a * fac n
fac' a 0 = a * fac 0
         = a * 1
         = a
fac' a (n+1) = a * fac (n+1)
             = a * ((n+1) * fac n)
             = (a * (n+1)) * fac n
             = fac' (a * (n+1)) n
```

Example: Factorial

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
fac' a n = a * fac n
fac' a 0 = a * fac 0
         = a * 1
         = a
fac' a (n+1) = a * fac (n+1)
             = a * ((n+1) * fac n)
             = (a * (n+1)) * fac n
             = fac' (a * (n+1)) n
```

New definition

Example: Factorial

```
fac 0       = 1
fac (n+1) = (n+1) * fac n
fac' a n = a * fac n
fac' a 0 = a * fac 0
         = a * 1
         = a
fac' a (n+1) = a * fac (n+1)
             = a * ((n+1) * fac n)
             = (a * (n+1)) * fac n
             = fac' (a * (n+1)) n
```

New definition

Instantiation

Instantiation

Example: Factorial

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
fac' a n = a * fac n
fac' a 0 = a * fac 0
         = a * 1
         = a
fac' a (n+1) = a * fac (n+1)
             = a * ((n+1) * fac n)
             = (a * (n+1)) * fac n
             = fac' (a * (n+1)) n
```

New definition

Instantiation

Unfolding

Instantiation

Unfolding

Example: Factorial

```
fac 0       = 1
fac (n+1) = (n+1) * fac n
fac' a n = a * fac n
fac' a 0 = a * fac 0
                = a * 1
                = a
fac' a (n+1) = a * fac (n+1)
                = a * ((n+1) * fac n)
                = (a * (n+1)) * fac n
                = fac' (a * (n+1)) n
```

New definition

Instantiation

Unfolding

Instantiation

Unfolding

Folding

Example: Factorial

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
fac' a n = a * fac n
fac' a 0 = a * fac 0
         = a * 1
         = a
fac' a (n+1) = a * fac (n+1)
             = a * ((n+1) * fac n)
             = (a * (n+1)) * fac n
             = fac' (a * (n+1)) n
```

New definition

Instantiation

Unfolding

Laws

Instantiation

Unfolding

Laws

Folding

Questions

Questions

- Is the new `fac'` program totally correct? Why/why not?

Questions

- Is the new `fac'` program totally correct? Why/why not?
  - Answer: yes, because it and the original program are both total

Questions

- Is the new `fac'` program totally correct? Why/why not?
  - Answer: yes, because it and the original program are both total
- Can you give an example of a calculation that does *not* preserve total correctness?

Questions

- Is the new `fac'` program totally correct? Why/why not?
  - Answer: yes, because it and the original program are both total
- Can you give an example of a calculation that does *not* preserve total correctness?
  - Answer: here's a trivial example:

Questions

- Is the new `fac'` program totally correct? Why/why not?
  - Answer: yes, because it and the original program are both total
- Can you give an example of a calculation that does *not* preserve total correctness?
  - Answer: here's a trivial example:

```
f x = 0
```

Questions

- Is the new `fac'` program totally correct? Why/why not?
  - Answer: yes, because it and the original program are both total
- Can you give an example of a calculation that does *not* preserve total correctness?
  - Answer: here's a trivial example:

```
f x = 0
    = { folding f }
```

Questions

- Is the new `fac'` program totally correct? Why/why not?
  - Answer: yes, because it and the original program are both total
- Can you give an example of a calculation that does *not* preserve total correctness?
  - Answer: here's a trivial example:

```
f x = 0
    = { folding f }
      f x
```

Fold/Unfold, In Summary

Fold/Unfold, In Summary

- We start from a potentially inefficient program given as a set of recursive equations; this is our *specification*

Fold/Unfold, In Summary

- We start from a potentially inefficient program given as a set of recursive equations; this is our *specification*
- We can make five types of step in our calculations:

Fold/Unfold, In Summary

- We start from a potentially inefficient program given as a set of recursive equations; this is our *specification*
- We can make five types of step in our calculations:
  1. *Introduce* a new definition

Fold/Unfold, In Summary

- We start from a potentially inefficient program given as a set of recursive equations; this is our *specification*
- We can make five types of step in our calculations:
  1. *Introduce* a new definition
  2. *Instantiate* a definition with specific arguments

Fold/Unfold, In Summary

- We start from a potentially inefficient program given as a set of recursive equations; this is our *specification*
- We can make five types of step in our calculations:
  1. *Introduce* a new definition
  2. *Instantiate* a definition with specific arguments
  3. *Unfold*: rewrite something by replacing the lhs of a definition with its rhs

Fold/Unfold, In Summary

- We start from a potentially inefficient program given as a set of recursive equations; this is our *specification*
- We can make five types of step in our calculations:
  1. *Introduce* a new definition
  2. *Instantiate* a definition with specific arguments
  3. *Unfold*: rewrite something by replacing the lhs of a definition with its rhs
  4. *Fold*: rewrite something by replacing the rhs of a definition with its lhs

Fold/Unfold, In Summary

- We start from a potentially inefficient program given as a set of recursive equations; this is our *specification*
- We can make five types of step in our calculations:
  1. *Introduce* a new definition
  2. *Instantiate* a definition with specific arguments
  3. *Unfold*: rewrite something by replacing the lhs of a definition with its rhs
  4. *Fold*: rewrite something by replacing the rhs of a definition with its lhs
  5. Apply some *laws*

Fold/Unfold, In Summary

- We start from a potentially inefficient program given as a set of recursive equations; this is our *specification*
- We can make five types of step in our calculations:
  1. *Introduce* a new definition
  2. *Instantiate* a definition with specific arguments
  3. *Unfold*: rewrite something by replacing the lhs of a definition with its rhs
  4. *Fold*: rewrite something by replacing the rhs of a definition with its lhs
  5. Apply some *laws*
- The resulting program may not be totally correct, but it will be partially correct

Fold/Unfold, In Summary

- We start from a potentially inefficient program given as a set of recursive equations; this is our *specification*
- We can make five types of step in our calculations:
  1. *Introduce* a new definition
  2. *Instantiate* a definition with specific arguments
  3. *Unfold*: rewrite something by replacing the lhs of a definition with its rhs
  4. *Fold*: rewrite something by replacing the rhs of a definition with its lhs
  5. Apply some *laws*
- The resulting program may not be totally correct, but it will be partially correct
- Exercise: Explain, informally, why these five steps will preserve partial correctness.

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
twofib 0
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
= { arithmetic }
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
= { arithmetic }
  (fib 0, fib 1)
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
= { arithmetic }
  (fib 0, fib 1)
= { unfolding fib }
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
= { arithmetic }
  (fib 0, fib 1)
= { unfolding fib }
  (1,1)
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

```
twofib (n+1)
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
= { arithmetic }
  (fib 0, fib 1)
= { unfolding fib }
  (1,1)
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
= { arithmetic }
  (fib 0, fib 1)
= { unfolding fib }
  (1,1)
```

```
  twofib (n+1)
= { unfolding twofib }
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
= { arithmetic }
  (fib 0, fib 1)
= { unfolding fib }
  (1,1)
```

```
  twofib (n+1)
= { unfolding twofib }
  (fib (n+1), fib ((n+1)+1))
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
= { arithmetic }
  (fib 0, fib 1)
= { unfolding fib }
  (1,1)
```

```
  twofib (n+1)
= { unfolding twofib }
  (fib (n+1), fib ((n+1)+1))
= { arithmetic }
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
= { arithmetic }
  (fib 0, fib 1)
= { unfolding fib }
  (1,1)
```

```
  twofib (n+1)
= { unfolding twofib }
  (fib (n+1), fib ((n+1)+1))
= { arithmetic }
  (fib (n+1), fib (n+2))
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
= { arithmetic }
  (fib 0, fib 1)
= { unfolding fib }
  (1,1)
```

```
  twofib (n+1)
= { unfolding twofib }
  (fib (n+1), fib ((n+1)+1))
= { arithmetic }
  (fib (n+1), fib (n+2))
= { unfolding fib }
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
= { arithmetic }
  (fib 0, fib 1)
= { unfolding fib }
  (1,1)
```

```
  twofib (n+1)
= { unfolding twofib }
  (fib (n+1), fib ((n+1)+1))
= { arithmetic }
  (fib (n+1), fib (n+2))
= { unfolding fib }
  (fib (n+1), fib (n+1) + fib n)
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
= { arithmetic }
  (fib 0, fib 1)
= { unfolding fib }
  (1,1)
```

```
  twofib (n+1)
= { unfolding twofib }
  (fib (n+1), fib ((n+1)+1))
= { arithmetic }
  (fib (n+1), fib (n+2))
= { unfolding fib }
  (fib (n+1), fib (n+1) + fib n)
= { introducing a let-binding }
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
= { arithmetic }
  (fib 0, fib 1)
= { unfolding fib }
  (1,1)
```

```
  twofib (n+1)
= { unfolding twofib }
  (fib (n+1), fib ((n+1)+1))
= { arithmetic }
  (fib (n+1), fib (n+2))
= { unfolding fib }
  (fib (n+1), fib (n+1) + fib n)
= { introducing a let-binding }
  let (f0,f1) = (fib n, fib (n+1))
        in (f1, f0+f1)
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
= { arithmetic }
  (fib 0, fib 1)
= { unfolding fib }
  (1,1)
```

```
  twofib (n+1)
= { unfolding twofib }
  (fib (n+1), fib ((n+1)+1))
= { arithmetic }
  (fib (n+1), fib (n+2))
= { unfolding fib }
  (fib (n+1), fib (n+1) + fib n)
= { introducing a let-binding }
  let (f0,f1) = (fib n, fib (n+1))
        in (f1, f0+f1)
= { folding twofib }
```

Example: Fibonacci

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Spec: `twofib n = (fib n, fib (n+1))`

```
  twofib 0
= { unfolding twofib }
  (fib 0, fib (0+1))
= { arithmetic }
  (fib 0, fib 1)
= { unfolding fib }
  (1,1)
```

```
  twofib (n+1)
= { unfolding twofib }
  (fib (n+1), fib ((n+1)+1))
= { arithmetic }
  (fib (n+1), fib (n+2))
= { unfolding fib }
  (fib (n+1), fib (n+1) + fib n)
= { introducing a let-binding }
  let (f0,f1) = (fib n, fib (n+1))
        in (f1, f0+f1)
= { folding twofib }
  let (f0,f1) = twofib n
        in (f1, f0+f1)
```

Example: Fibonacci

Example: Fibonacci

- So, starting from this exponential-time program…

Example: Fibonacci

- So, starting from this exponential-time program…

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Example: Fibonacci

- So, starting from this exponential-time program…

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

- …and this spec…

Example: Fibonacci

- So, starting from this exponential-time program…

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

- …and this spec…

```
twofib n = (fib n, fib (n+1))
```

Example: Fibonacci

- So, starting from this exponential-time program…

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

- …and this spec…

```
twofib n = (fib n, fib (n+1))
```

- …we have calculated this program…

Example: Fibonacci

- So, starting from this exponential-time program…

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

- …and this spec…

```
twofib n = (fib n, fib (n+1))
```

- …we have calculated this program…

```
twofib 0 = (1,1)
twofib (n+1) = let (f0,f1) = twofib n in (f1, f0+f1)
```

Example: Fibonacci

- So, starting from this exponential-time program…

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

- …and this spec…

```
twofib n = (fib n, fib (n+1))
```

- …we have calculated this program…

```
twofib 0 = (1,1)
twofib (n+1) = let (f0,f1) = twofib n in (f1, f0+f1)
```

- …which will run in *linear* time!

Fold/Unfold Calculations: Pros and Cons

Fold/Unfold Calculations: Pros and Cons

+ Simple: based on a small set of rules

Fold/Unfold Calculations: Pros and Cons

+ Simple: based on a small set of rules
+ Could be computer-checked or even part-automated

Fold/Unfold Calculations: Pros and Cons

- Simple: based on a small set of rules
  - Could be computer-checked or even part-automated
- Widely-applicable

Fold/Unfold Calculations: Pros and Cons

+ Simple: based on a small set of rules
  + Could be computer-checked or even part-automated
+ Widely-applicable

– Only provides partial correctness

Fold/Unfold Calculations: Pros and Cons

+ Simple: based on a small set of rules
  + Could be computer-checked or even part-automated
+ Widely-applicable

– Only provides partial correctness
  – Sometimes needs a "eureka" moment

Exercises!

Exercises!

1. Explain, informally, why the five steps allowed in fold/unfold calculations preserve partial correctness.

Exercises!

1. Explain, informally, why the five steps allowed in fold/unfold calculations preserve partial correctness.

2. Consider the following function that reverses a list:

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Calculate a new version of this function that uses an accumulating parameter.

Why will this function be more efficient? Remember the definition of ++:

```
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

Exercises! (cont.)

Exercises! (cont.)

3. Consider the following simple expression evaluator:

```
data Expr = Num Int | Add Expr Expr
eval (Num n)       = n
eval (Add e1 e2)   = eval e1 + eval e2
```

Calculate a new version of the program that takes as an additional argument a function that tells it what to do with the final result. The specification should be as follows:

```
evalCont e k = k (eval e)
```

(Programs written in this form are said to be in *continuation-passing style*.)