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…
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
Calculating Programs: Lecture 1

**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:

- \( \text{fac } 0 = 1 \)
- \( \text{fac } (n+1) = (n+1) \times \text{fac } n \)
- \( [] + ys = ys \)
- \( (x:xs) + ys = x:(xs + ys) \)
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:
  
  \[
  \text{fac } 0 = 1 \\
  \text{fac } (n+1) = (n+1) \times \text{fac } n
  \]

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

\[
\begin{align*}
\text{fac } 0 &= 1 \\
\text{fac } (n+1) &= (n+1) \times \text{fac } n
\end{align*}
\]
EXAMPLE: FACTORIAL

\[
\begin{align*}
\text{fac } 0 & = 1 \\
\text{fac } (n+1) & = (n+1) \times \text{fac } n \\
\text{fac'} a \ n & = a \times \text{fac } n
\end{align*}
\]
EXAMPLE: FACTORIAL

\[
\begin{align*}
\text{fac } 0 &= 1 \\
\text{fac } (n+1) &= (n+1) \times \text{fac } n \\
\text{fac’ } a \ n &= a \times \text{fac } n \\
\text{fac’ } a \ 0 &= a \times \text{fac } 0
\end{align*}
\]
EXAMPLE: FACTORIAL

\[
\begin{align*}
\text{fac } 0 & = 1 \\
\text{fac } (n+1) & = (n+1) \times \text{fac } n \\
\text{fac'} a \ n & = a \times \text{fac } n \\
\text{fac'} a \ 0 & = a \times \text{fac } 0 \\
& = a \times 1
\end{align*}
\]
EXAMPLE: FACTORIAL

\[
\begin{align*}
\text{fac } 0 &= 1 \\
\text{fac } (n+1) &= (n+1) \times \text{fac } n \\
\text{fac'} \ a \ n &= a \times \text{fac } n \\
\text{fac'} \ a \ 0 &= a \times \text{fac } 0 \\
&= a \times 1 \\
&= a
\end{align*}
\]
EXAMPLE: FACTORIAL

\[
\begin{align*}
\text{fac } 0 & = 1 \\
\text{fac } (n+1) & = (n+1) \times \text{fac } n \\
\text{fac’ } a \ n & = a \times \text{fac } n \\
\text{fac’ } a \ 0 & = a \times \text{fac } 0 \\
& = a \times 1 \\
& = a \\
\text{fac’ } a \ (n+1) & = a \times \text{fac } (n+1)
\end{align*}
\]
EXAMPLE: FACTORIAL

\[
\begin{align*}
\text{fac } 0 & = 1 \\
\text{fac } (n+1) & = (n+1) \times \text{fac } n \\
\text{fac’ } a \ n & = a \times \text{fac } n \\
\text{fac’ } a \ 0 & = a \times \text{fac } 0 \\
& = a \times 1 \\
& = a \\
\text{fac’ } a \ (n+1) & = a \times \text{fac } (n+1) \\
& = a \times ((n+1) \times \text{fac } n)
\end{align*}
\]
EXAMPLE: FACTORIAL

\[
\begin{align*}
\text{fac } 0 & = 1 \\
\text{fac } (n+1) & = (n+1) \times \text{fac } n \\
\text{fac’ } a \ n & = a \times \text{fac } n \\
\text{fac’ } a \ 0 & = a \times \text{fac } 0 \\
& = a \times 1 \\
& = a \\
\text{fac’ } a \ (n+1) & = a \times \text{fac } (n+1) \\
& = a \times ((n+1) \times \text{fac } n) \\
& = (a \times (n+1)) \times \text{fac } n
\end{align*}
\]
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

\[
\begin{align*}
\text{fac } 0 & = 1 \\
\text{fac } (n+1) & = (n+1) \times \text{fac } n \\
\text{fac'} \ a \ n & = a \times \text{fac } n \\
\text{fac'} \ a \ 0 & = a \times \text{fac } 0 \\
& = a \times 1 \\
& = a \\
\text{fac'} \ a \ (n+1) & = a \times \text{fac } (n+1) \\
& = a \times ((n+1) \times \text{fac } n) \\
& = (a \times (n+1)) \times \text{fac } n \\
& = \text{fac'} \ (a \times (n+1)) \ n
\end{align*}
\]
**EXAMPLE: FACTORIAL**

\[
\begin{align*}
\text{fac } 0 &= 1 \\
\text{fac } (n+1) &= (n+1) \times \text{fac } n \\
\text{fac'} a \ n &= a \times \text{fac } n \\
\text{fac'} a \ 0 &= a \times \text{fac } 0 \\
&= a \times 1 \\
&= a \\
\text{fac'} a \ (n+1) &= a \times \text{fac } (n+1) \\
&= a \times ((n+1) \times \text{fac } n) \\
&= (a \times (n+1)) \times \text{fac } n \\
&= \text{fac'} (a \times (n+1)) \ n
\end{align*}
\]
**EXAMPLE: FACTORIAL**

\[
\begin{align*}
\text{fac } 0 &= 1 \\
\text{fac } (n+1) &= (n+1) \times \text{fac } n \\
\text{fac'} a \ n &= a \times \text{fac } n \\
\text{fac'} a \ 0 &= a \times \text{fac } 0 \\
&= a \times 1 \\
&= a \\
\text{fac'} a \ (n+1) &= a \times \text{fac } (n+1) \\
&= a \times ((n+1) \times \text{fac } n) \\
\text{Unfolding} \\
&= (a \times (n+1)) \times \text{fac } n \\
&= \text{fac'} (a \times (n+1)) \ n
\end{align*}
\]
**EXAMPLE: FACTORIAL**

\[
\begin{align*}
\text{fac } 0 & \quad = \quad 1 \\
\text{fac } (n+1) & \quad = \quad (n+1) \times \text{fac } n \\
\text{fac'} a \ n & \quad = \quad a \times \text{fac } n \\
\text{fac'} a \ 0 & \quad = \quad a \times \text{fac } 0 \\
& \quad = \quad a \times 1 \\
& \quad = \quad a
\end{align*}
\]

- **New definition**
- **Instantiation**
- **Unfolding**
- **Instantiation**
- **Unfolding**
- **Folding**
EXAMPLE: FACTORIAL

\[
\begin{align*}
\text{fac } 0 & = 1 \\
\text{fac } (n+1) & = (n+1) \times \text{fac } n \\
\text{fac'} \ a \ n & = a \times \text{fac } n \\
\text{fac'} \ a \ 0 & = a \times \text{fac } 0 \\
& = a \times 1 \\
& = a \\
\text{fac'} \ a \ (n+1) & = a \times \text{fac } (n+1) \\
& = a \times ((n+1) \times \text{fac } n) \\
\text{fac'} \ a \ (n+1) & = (a \times (n+1)) \times \text{fac } n \\
\text{fac'} \ (a \times (n+1)) \ n & \\
\end{align*}
\]
QUESTIONS
QUESTIONS

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

▸ Is the new \texttt{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 \texttt{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 \textit{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:

  \[
  f \times = 0 \\
  = \{ \text{folding } f \} 
  \]
Questions

▸ Is the new \texttt{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 \textit{not} preserve total correctness?

▸ Answer: here’s a trivial example:

\begin{verbatim}
f x = 0
  = \{ folding f \}
  f x
\end{verbatim}
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.
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:
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
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

\[
\begin{align*}
\text{fib } 0 &= 1 \\
\text{fib } 1 &= 1 \\
\text{fib } (n+2) &= \text{fib } (n+1) + \text{fib } n \\
\text{Spec: } \text{twofib } n &= (\text{fib } n, \text{fib } (n+1))
\end{align*}
\]
Example: Fibonacci

\[
\begin{align*}
\text{fib 0} & = 1 \\
\text{fib 1} & = 1 \\
\text{fib (n+2)} & = \text{fib (n+1)} + \text{fib n} \\
\text{Spec: twofib n} & = (\text{fib n}, \text{fib (n+1)})
\end{align*}
\]

twofib 0
EXAMPLE: FIBONACCI

\[
\begin{align*}
\text{fib} \ 0 & \quad = \ 1 \\
\text{fib} \ 1 & \quad = \ 1 \\
\text{fib} \ (n+2) & \quad = \ \text{fib} \ (n+1) + \ \text{fib} \ n \\
\text{Spec:} \ \text{twofib} \ n & \quad = \ (\text{fib} \ n, \ \text{fib} \ (n+1)) \\
\text{twofib} \ 0 & \\
& \quad = \ \{ \ unfolding \ \text{twofib} \ \} 
\end{align*}
\]
EXAMPLE: FIBONACCI

\[
\begin{align*}
\text{fib } 0 & \ = \ 1 \\
\text{fib } 1 & \ = \ 1 \\
\text{fib } (n+2) & \ = \ \text{fib } (n+1) + \text{fib } n \\
\text{Spec: } \text{twofib } n & \ = \ (\text{fib } n, \ \text{fib } (n+1)) \\
\text{twofib } 0 & \\
= & \ {\text{unfolding } \text{twofib}} \\
= & \ (\text{fib } 0, \ \text{fib } (0+1))
\end{align*}
\]
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**

\[
\begin{align*}
\text{fib } 0 & \quad = \quad 1 \\
\text{fib } 1 & \quad = \quad 1 \\
\text{fib } (n+2) & \quad = \quad \text{fib } (n+1) + \text{fib } n
\end{align*}
\]

**Spec:** \( \text{twofib } n = (\text{fib } n, \text{fib } (n+1)) \)

\[
\begin{align*}
\text{twofib } 0 & \\
= \ & \{ \text{unfolding } \text{twofib} \} \\
& = \quad (\text{fib } 0, \text{fib } (0+1)) \\
= \ & \{ \text{arithmetic} \} \\
& = \quad (\text{fib } 0, \text{fib } 1)
\end{align*}
\]
**EXAMPLE: FIBONACCI**

\[
\begin{align*}
\text{fib 0} &= 1 \\
\text{fib 1} &= 1 \\
\text{fib (n+2)} &= \text{fib (n+1)} + \text{fib n} \\
\text{Spec: twofib n} &= (\text{fib n}, \text{fib (n+1)}) \\
\text{twofib 0} &= \\
&= \{ \text{unfolding twofib} \} \\
&= (\text{fib 0}, \text{fib (0+1)}) \\
&= \{ \text{arithmetic} \} \\
&= (\text{fib 0}, \text{fib 1}) \\
&= \{ \text{unfolding fib} \}
\end{align*}
\]
EXAMPLE: FIBONACCI

\[
\begin{align*}
\text{fib } 0 & = 1 \\
\text{fib } 1 & = 1 \\
\text{fib } (n+2) & = \text{fib } (n+1) + \text{fib } n \\
\text{Spec: twofib } n & = (\text{fib } n, \text{fib } (n+1)) \\
\text{twofib } 0 & \\
& = \{ \text{unfolding twofib} \} \\
& \quad (\text{fib } 0, \text{fib } (0+1)) \\
& = \{ \text{arithmetic} \} \\
& \quad (\text{fib } 0, \text{fib } 1) \\
& = \{ \text{unfolding fib} \} \\
& \quad (1,1)
\end{align*}
\]
EXAMPLE: FIBONACCI

\[
\begin{align*}
  \text{fib} \ 0 & \quad = \quad 1 \\
  \text{fib} \ 1 & \quad = \quad 1 \\
  \text{fib} \ (n+2) & \quad = \quad \text{fib} \ (n+1) \ + \ \text{fib} \ n \\
  \text{Spec: twofib} \ n & \quad = \quad (\text{fib} \ n, \ \text{fib} \ (n+1)) \\
  \text{twofib} \ 0 & \quad = \quad \{\text{unfolding twofib}\} \\
  & \quad = \quad (\text{fib} \ 0, \ \text{fib} \ (0+1)) \\
  & \quad = \quad \{\text{arithmetic}\} \\
  & \quad = \quad (\text{fib} \ 0, \ \text{fib} \ 1) \\
  & \quad = \quad \{\text{unfolding fib}\} \\
  & \quad = \quad (1, 1)
\end{align*}
\]
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

\[
\begin{align*}
\text{fib 0} & = 1 \\
\text{fib 1} & = 1 \\
\text{fib (n+2)} & = \text{fib (n+1)} + \text{fib n} \\
\text{Spec: twofib n} & = (\text{fib n}, \text{fib (n+1)}) \\
\text{twofib 0} & = \{ \text{unfolding twofib} \} \\
& = (\text{fib 0}, \text{fib (0+1)}) \\
& = (1,1)
\end{align*}
\]

\[
\text{twofib (n+1)} = \{ \text{unfolding twofib} \} \\
& = (\text{fib (n+1)}, \text{fib ((n+1)+1)})
\]
Example: Fibonacci

\[
\begin{align*}
\text{fib } 0 & = 1 \\
\text{fib } 1 & = 1 \\
\text{fib } (n+2) & = \text{fib } (n+1) + \text{fib } n
\end{align*}
\]

Spec: \( \text{twofib } n = (\text{fib } n, \text{fib } (n+1)) \)

\[
\begin{align*}
\text{twofib } 0 & \\
& = \{ \text{unfolding twofib} \} \\
& = (\text{fib } 0, \text{fib } (0+1)) \\
& = \{ \text{arithmetic} \} \\
& = (1,1)
\end{align*}
\]

\[
\begin{align*}
\text{twofib } (n+1) & \\
& = \{ \text{unfolding twofib} \} \\
& = (\text{fib } (n+1), \text{fib } ((n+1)+1)) \\
& = \{ \text{arithmetic} \}
\end{align*}
\]
**EXAMPLE: FIBONACCI**

\[
\begin{align*}
\text{fib 0} &= 1 \\
\text{fib 1} &= 1 \\
\text{fib (n+2)} &= \text{fib (n+1)} + \text{fib n} \\
\text{Spec: twofib n} &= (\text{fib n}, \text{fib (n+1)}) \\
\text{twofib 0} &= \text{unfolding twofib} \ \\
&= (\text{fib 0}, \text{fib (0+1)}) \\
&= \text{arithmetic} \ \\
&= (1,1)
\end{align*}
\]
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

\[
\begin{align*}
\text{fib} \ 0 & \ = \ 1 \\
\text{fib} \ 1 & \ = \ 1 \\
\text{fib} \ (n+2) & \ = \ \text{fib} \ (n+1) + \ \text{fib} \ n \\
\text{Spec:} \ \text{twofib} \ n & \ = \ (\text{fib} \ n, \ \text{fib} \ (n+1)) \\
\text{twofib} \ 0 & \ = \ \{ \text{unfolding} \ \text{twofib} \} \\
& \ = \ (\text{fib} \ 0, \ \text{fib} \ (0+1)) \\
& \ = \ \{ \text{arithmetic} \} \\
& \ = \ (1,1) \\
\text{twofib} \ (n+1) & \ = \ \{ \text{unfolding} \ \text{twofib} \} \\
& \ = \ (\text{fib} \ (n+1), \ \text{fib} \ ((n+1)+1)) \\
& \ = \ \{ \text{arithmetic} \} \\
& \ = \ (\text{fib} \ (n+1), \ \text{fib} \ (n+2)) \\
& \ = \ \{ \text{unfolding} \ \text{fib} \} \\
& \ = \ (\text{fib} \ (n+1), \ \text{fib} \ (n+1) + \ \text{fib} \ n) \\
& \ = \ \{ \text{introducing \ a \ let-binding} \} \\
\end{align*}
\]
EXAMPLE: FIBONACCI

\[
\begin{align*}
fib \ 0 & \quad = \ 1 \\
fib \ 1 & \quad = \ 1 \\
fib \ (n+2) & \quad = \ fib \ (n+1) \ + \ fib \ n
\end{align*}
\]

**Spec:** \( \text{twofib} \ n = (fib \ n, \ fib \ (n+1)) \)

\[
\begin{align*}
\text{twofib} \ 0 & \\
& \quad = \ \{ \text{unfolding} \ \text{twofib} \} \\
& \quad (fib \ 0, \ fib \ (0+1)) \\
& \quad = \ \{ \text{arithmetic} \} \\
& \quad (fib \ 0, \ fib \ 1) \\
& \quad = \ \{ \text{unfolding} \ fib \} \\
& \quad (1,1)
\end{align*}
\]

\[
\begin{align*}
\text{twofib} \ (n+1) & \\
& \quad = \ \{ \text{unfolding} \ \text{twofib} \} \\
& \quad (fib \ (n+1), \ fib \ ((n+1)+1)) \\
& \quad = \ \{ \text{arithmetic} \} \\
& \quad (fib \ (n+1), \ fib \ (n+2)) \\
& \quad = \ \{ \text{unfolding} \ fib \} \\
& \quad (fib \ (n+1), \ fib \ (n+1) \ + \ fib \ n) \\
& \quad = \ \{ \text{introducing} \ \text{a} \ \text{let-binding} \} \\
& \quad \text{let} \ (f0,f1) = (fib \ n, \ fib \ (n+1)) \\
& \quad \quad \text{in} \ (f1, \ f0+f1)
\end{align*}
\]
EXAMPLE: FIBONACCI

\[
\begin{align*}
\text{fib 0} &= 1 \\
\text{fib 1} &= 1 \\
\text{fib (n+2)} &= \text{fib (n+1)} + \text{fib n} \\
\text{Spec: twofib n} &= (\text{fib n}, \text{fib (n+1)}) \\
\text{twofib 0} &= \{\text{unfolding twofib}\} \\
&= (\text{fib 0}, \text{fib (0+1)}) \\
&= \{\text{arithmetic}\} \\
&= (1, 1)
\end{align*}
\]

\[
\begin{align*}
\text{twofib (n+1)} &= \{\text{unfolding twofib}\} \\
&= (\text{fib (n+1)}, \text{fib ((n+1)+1)}) \\
&= \{\text{arithmetic}\} \\
&= (\text{fib (n+1)}, \text{fib (n+2)}) \\
&= \{\text{unfolding fib}\} \\
&= (\text{fib (n+1)}, \text{fib (n+1) + fib n}) \\
&= \{\text{introducing a let-binding}\} \\
&\quad \text{let } (f0, f1) = (\text{fib n}, \text{fib (n+1)}) \\
&\quad \text{in } (f1, f0+f1) \\
&= \{\text{folding twofib}\}
\end{align*}
\]
**EXAMPLE: FIBONACCI**

\[
\begin{align*}
\text{fib } 0 & = 1 \\
\text{fib } 1 & = 1 \\
\text{fib } (n+2) & = \text{fib } (n+1) + \text{fib } n \\
\text{Spec: twofib } n & = (\text{fib } n, \text{fib } (n+1)) \\
\text{twofib } 0 & = \{ \text{unfolding twofib} \} \\
& \quad (\text{fib } 0, \text{fib } (0+1)) \\
& = \{ \text{arithmetic} \} \\
& \quad (1,1)
\end{align*}
\]

\[
\begin{align*}
twofib & (n+1) \\
& = \{ \text{unfolding twofib} \} \\
& \quad (\text{fib } (n+1), \text{fib } ((n+1)+1)) \\
& = \{ \text{arithmetic} \} \\
& \quad (\text{fib } (n+1), \text{fib } (n+2)) \\
& = \{ \text{unfolding fib} \} \\
& \quad (\text{fib } (n+1), \text{fib } (n+1) + \text{fib } n) \\
& = \{ \text{introducing a let-binding} \} \\
& \quad \text{let } (f0,f1) = (\text{fib } n, \text{fib } (n+1)) \\
& \quad \quad \text{in } (f1, f0+f1) \\
& = \{ \text{folding twofib} \} \\
& \quad \text{let } (f0,f1) = \text{twofib } n \\
& \quad \quad \text{in } (f1, f0+f1)
\end{align*}
\]
EXAMPLE: FIBONACCI
EXAMPLE: FIBONACCI

- So, starting from this exponential-time program...
EXAMPLE: FIBONACCI

- So, starting from this exponential-time program...

\[
\begin{align*}
\text{fib } 0 & = 1 \\
\text{fib } 1 & = 1 \\
\text{fib } (n+2) & = \text{fib } (n+1) + \text{fib } n
\end{align*}
\]
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...

  \[
  \begin{align*}
  \text{fib } 0 &= 1 \\
  \text{fib } 1 &= 1 \\
  \text{fib } (n+2) &= \text{fib } (n+1) + \text{fib } n
  \end{align*}
  \]

- ...and this spec...

  \[
  \text{twofib } n = (\text{fib } n, \text{fib } (n+1))
  \]
EXAMPLE: FIBONACCI

- So, starting from this exponential-time program...
  
  \[
  \begin{align*}
  \text{fib } 0 & = 1 \\
  \text{fib } 1 & = 1 \\
  \text{fib } (n+2) & = \text{fib } (n+1) + \text{fib } n
  \end{align*}
  \]

- ...and this spec...
  
  \[
  \text{twofib } n = (\text{fib } n, \text{fib } (n+1))
  \]

- ...we have calculated this program...
**EXAMPLE: FIBONACCI**

- **So, starting from this exponential-time program...**

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

- **...and this spec...**

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

- **...we have calculated this program...**

  ```plaintext
  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...
  
  \[
  \begin{align*}
  fib \ 0 & \ = \ 1 \\
  fib \ 1 & \ = \ 1 \\
  fib \ (n+2) & = fib \ (n+1) + fib \ n
  \end{align*}
  \]

- ...and this spec...
  
  \[
  twofib \ n = (fib \ n, fib \ (n+1))
  \]

- ...we have calculated this program...
  
  \[
  \begin{align*}
  twofib \ 0 & = (1,1) \\
  twofib \ (n+1) & = let \ (f0,f1) = twofib \ n \ in \ (f1, f0+f1)
  \end{align*}
  \]

- ...which will run in \textit{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.
1. Explain, informally, why the five steps allowed in fold/unfold calculations preserve partial correctness.

2. Consider the following function that reverses a list:

   \[
   \text{reverse } \text{[]} = \text{[]}
   \]

   \[
   \text{reverse } (x:xs) = \text{reverse } xs ++ [x]
   \]

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

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

   \[
   \text{[]} ++ ys = ys
   \]

   \[
   (x:xs) ++ ys = x:(xs ++ ys)
   \]
EXERCISES! (CONT.)
3. Consider the following simple expression evaluator:

```haskell
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:

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

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