Control flow, continuations, monads
Week 8

• Structured programming
• Procedural abstraction
• Exceptions
• Continuations
• Monads
Turning the clock back

To understand what structured programming is, let’s look at what programs looked like before it:

Do you know what this Fortran program does?

➤ A: yes, B: no
Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

CR Categories: 4.22, 5.23, 5.24

Editor:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all “higher level” programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer’s activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the “making” of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the
1. Primary way to understand programs: processing code in sequence

2. goto programs: can jump anywhere => spaghetti

3. Eliminate gotos!

4. Need higher level control flow constructs!
Structured Programming with go to Statements

DONALD E. KNUTH
Stanford University, Stanford, California 94305

A consideration of several different examples sheds new light on the problem of writing reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved methods for debugging and error control, making it possible to write a large class of programs exactly and efficiently without go to statements; (b) a methodology of program design, beginning with modular and support, but possibly multilevel programs that are systematically transformed into efficient and correct, but possibly less readable code. The discussion brings out opposing views about whether or not go to statements should be abolished; some merit is found on both sides of this question. Finally, an attempt is made to define the true nature of structured programming, and to recommend fruitful directions for further study.

Keywords and phrases: structured programming, go to statements, language design, error localization, recursion, Boolean variables, iteration, optimization of programs, program transformations, program manipulation systems, abstracta, Quickstart, efficiency

CR categories: 4.2, 4.30, 4.20, 5.5, 6.1 (5.22), 6.54, 6.25, 5.25
1. Eliminate gotos in some cases
2. Introduce gotos in other cases

Where?
Structured programming

• What is structured programming?
  ➤ Programming paradigm consisting of: control structures, procedures, and blocks

• Why?
  ➤ Largely because gotos make it extremely hard to reason about and prove things about programs
Programming with gotos

• Largely unstructured control flow graphs

• But: extremely flexible!

Q: What constructs do we need to express any computable function?

A: Sequence, selection, iteration
CFG of sequencing
CFG of if-else statement
CFG of while loop
CFG of for loop
What is this a CFG of?
What is this a CFG of?

switch statement!
Week 8

• Structured programming
• Procedural abstraction
• Exceptions
• Continuations
• Monads
Procedural abstraction

• We can chain blocks using the previous constructs to program any computable function
  ➤ Downside:

• Procedural abstraction
  ➤ Organize code into subroutines that can be called multiple times from different blocks
  ➤ Upside: code reuse and better organization
  ➤ Downside: complicates CFGs
Procedural abstraction

• We can chain blocks using the previous constructs to program any computable function
  ➤ Downside: programs are giant blobs of code

• Procedural abstraction
  ➤ Organize code into subroutines that can be called multiple times from different blocks
  ➤ Upside: code reuse and better organization
  ➤ Downside: complicates CFGs
function f(x) {
    return h(x) + 1;
}

function g(x) {
    return h(x) - 1;
}

function h(x) {
    return x * 2;
}
By the way, this problem of determining the interprocedural control flow graph of a program is a quite important one, because it serves as the basis for approaches to control-flow integrity, one of the primary defenses against code execution attacks (e.g. buffer overflows and return-oriented programming.)

But what happens when we RETURN from the function call? An execution of \( g \) could return to \( f \) or \( g \) or maybe some arbitrary code which called into \( h \). There might be a lot of places we could return to. At runtime, how do we know where to go?

Don't know where \( h \) was called from until runtime!
Don’t know where h was called from until runtime!
What about return?

• There may be a lot of places where we could return to from a function call

• In general: determining the interprocedural CFG of a program is hard and super important

  ➤ Modern attacks hijack control flow to execute arbitrary code (control-flow integrity)

• At runtime: how do we know where to go?

  ➤ We keep track of return pointer on the stack!
Dynamic control flow

• The return pointer on the stack dictates where control goes

• Do we need to keep track of where control goes for if-statements, while loops, etc. on the stack as well?

➢ No! We know exactly where execution will go if condition is true or false!
Dynamic control flow

• The return pointer on the stack dictates where control goes

• Do we need to keep track of where control goes for if-statements, while loops, etc. on the stack as well?

➤ A: yes, B: no
Dynamic control flow

• The return pointer on the stack dictates where control goes

• Do we need to keep track of where control goes for if-statements, while loops, etc. on the stack as well?
  ➤ A: yes, **B: no**
Week 8

• Structured programming

• Procedural abstraction

• Exceptions

• Continuations

• Monads
Exceptions

• Two main language constructs
  ➢ raise/throw
  ➢ handler/catch

• Used to terminate part of a computation
  ➢ Jump out a construct
  ➢ Pass data as part of the jmp
  ➢ Return to the most recent site set up to handle ex
• Can we determine statically where to return to when we throw?
  ➤ A: yes, B: no
Exceptions

• Can we determine statically where to return to when we throw?

  ➤ A: yes, B: no
Exceptions

• How do we know where to return?
  ➤ keep track of handler information on the stack
  ➤ throw returns to the handler frame that is found on the stack

• How is exception handling scoped?
  ➤ User knows how to handle error
  ➤ Author of library function does not
function f(y) {
  throw "w00t";
}
try {
  f(1);
} catch (e) {
  console.log(e);
}
function f(y) {
    throw "w00t";
}
try {
    f(1);
} catch (e) {
    console.log(e);
}
function f(y) {
    throw "w00t";
}
try {
    f(1);
} catch (e) {
    console.log(e);
}
function f(y) {
  throw "w00t";
}
try {
  f(1);
} catch (e) {
  console.log(e);
}
A more complicated example

```javascript
try {
    function f(y) {
        throw "w00t";
    }
    function g(h) {
        try {
            h(1);
        } catch (e) { [3] }
    }
}

try {
    g(f);
} catch (e) { [1] }
} catch (e) { [2] }
```
A more complicated example

```
try {
    function f(y) {
        throw "w00t";
    }
    function g(h) {
        try {
            h(1);
        } catch (e) { [3] }
    }
}

try {
    g(f);
} catch (e) { [1] }
} catch (e) { [2] }
```
try {
  function f(y) {
    throw "w00t";
  }
  function g(h) {
    try {
      h(1);
    } catch (e) { [3] }
  }
}

try {
  g(f);
} catch (e) { [1] }
} catch (e) { [2] }
try {
  function f(y) {
    throw "w00t";
  }

  function g(h) {
    try {
      h(1);
    } catch (e) {
      [3]
    }
  }

  try {
    g(f);
  } catch (e) {
    [1]
  }
} catch (e) {
  [2]
}
A more complicated example

```javascript
try {
    function f(y) {
        throw "w00t";
    }
    function g(h) {
        try {
            h(1);
        } catch (e) { [3] }
    }

    try {
        g(f);
    } catch (e) { [1] }
} catch (e) { [2] }
```
A more complicated example

```
try {
    function f(y) {
        throw "w00t";
    }
    function g(h) {
        try {
            h(1);
        } catch (e) { [3] }
    }
    try {
        g(f);
    } catch (e) { [1] }
} catch (e) { [2] }
```
try {
  function f(y) {
    throw "w00t";
  }
  function g(h) {
    try {
      h(1);
    } catch (e) { [3] }
  }
}

try {
  g(f);
} catch (e) { [1] }
} catch (e) { [2] }
try {
    function f(y) {
        throw "w00t";
    }
    function g(h) {
        try {
            h(1);
        } catch (e) { [3] }
    }
    try {
        g(f);
    } catch (e) { [1] }
} catch (e) { [2] }
Dynamic vs. static scoping

- Again: exceptions follow dynamic scoping rules!
- Which handler would have been called if we had used static/lexical scoping rules?

➤ A: [1]

➤ B: [2]

➤ C: [3]
Dynamic vs. static scoping

• Again: exceptions follow dynamic scoping rules!

• Which handler would have been called if we had used static/lexical scoping rules?

➤ A: [1]

➤ B: [2]  

➤ C: [3]
Week 8

• Structured programming
• Procedural abstraction
• Exceptions
• Continuations
• Monads
Continuations

• Historical accident: to perform “asynchronous” computations many languages forced users to write their code in continuation passing style
  ➤ Algol 60, Landin’s SECD machine, Scheme
  ➤ See Reynolds’ *The Discoveries of Continuations*

• History repeats itself: JavaScript!
Example: async programming

```js
fs.readFile('myfile.txt', (err, data) => {
  console.log(data);
  processData(data);
});
```

- When you write an explicit callback:
  - you are implementing cooperative multithreading!
  - callback is a way for thread of execution to “save its current state” and let other code to run while it waits
Example: debugger

• Debugger is tool that builds on continuations
  ➤ execution pauses at set breakpoint:
    ➤ can inspect memory
    ➤ can continue running the program (have continuation to rest of the program!)
Plan

- What is a continuation?
- Continuation-passing style
- Short summary of how to use continuations to implement control flow
Continuations are implicit in your code

- Code you write implicitly manages the future (continuation) of its computation

- Consider: $(2 \times x + \frac{1}{y}) \times 2$

  A. Multiply 2 and x

  B. Divide 1 by y

  C. Add A and B

  D. Multiply C and 2
Continuations are implicit in your code

• Code you write implicitly manages the future (continuation) of its computation

• Consider: \((2 \times x + \frac{1}{y}) \times 2\)

  A. Multiply 2 and \(x\)

  B. Divide 1 by \(y\)

  C. Add A and B

  D. Multiply C and 2

  Current computation: rest of the program, current continuation
Continuations are implicit in your code

- Code you write implicitly manages the future (continuation) of its computation

- Consider: \((2 \times x + \frac{1}{y}) \times 2\)
  
  A. Multiply 2 and x
  
  B. Divide 1 by y
  
  C. Add A and B
  
  D. Multiply C and 2

```javascript
let before = 2*x;
let cont = curResult =>
    (before + curResult) * 2;
cont(1/y)
```
Node.js example

• Implicit continuation:

```javascript
const data = fs.readFileSync('myfile.txt');
console.log(data);
processData(data);
```

• Explicit continuation

```javascript
fs.readFile('myfile.txt', callback);
function callback (err, data) {
    console.log(data);
    processData(data);
};;
Continuation passing style (CPS)

- Some languages let you get your hands on the current continuation
  - call/cc (call with current continuation) is used to call a function and give it the current continuation
    - Why is this powerful?

- Most languages don’t let you get your hands on the current continuation: transform your code to CPS!
Continuation passing style (CPS)

• Some languages let you get your hands on the current continuation
  ➤ call/cc (call with current continuation) is used to call a function and give it the current continuation
  ➤ Why is this powerful?
    A: let’s some inner function bail out and continue program by calling continuation

• Most languages don’t let you get your hands on the current continuation: transform your code to CPS!
Continuation passing style

• Why do we want to do this?
  ➤ Makes control flow explicit: no return!

• So? Why should you care about this?
Continuation passing style

• Why do we want to do this?
  ➤ Makes control flow explicit: no return!
  ➤ Makes evaluation order explicit

• So? Why should you care about this?
Continuation passing style

• Why do we want to do this?
  ➤ Makes control flow explicit: no return!
  ➤ Makes evaluation order explicit

• So? Why should you care about this?
  ➤ IR of a number of (research) languages
  ➤ Turns function returns, exceptions, etc.: single jmp instruction! Can get rid of runtime stack!
To CPS, by example

function zero() {
    return 0;
}

function zero(cc) {
    cc(0);
}

cc is a function (the current continuation)
continue execution by calling cc
To CPS, by example

```javascript
function zero() {
    return 0;
}

function zero(cc) {
    cc(0);
}
```

**cc** is a function (the current continuation)

continue execution by calling **cc**
function fact(n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}

To CPS, by example

function fact(n, 
            cc) {
    if (n == 0) {
        cc(1);
    } else {
        fact(n - 1, r => 
            cc(n * r));
    }
}
To CPS, by example

```
function fact(n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}
```

```
function fact(n, cc) {
    if (n == 0) {
        cc(1);
    } else {
        fact(n - 1, r => cc(n * r));
    }
}
```
function fact(n, cc) {
    if (n == 0) {
        cc(1);
    } else {
        fact(n-1, r => cc(n*r));
    }
}

fact(3, id) ->
fact(2, rA => id(3*rA)) ->
fact(1, rb => (rA => id(3*rA))(2*rb)) ->
fact(0, rc => (rb => (rA => id(3*rA))(2*rb))(1*rc)) ->
    (rc => (rb => (rA => id(3*rA))(2*rb))(1*rc))(0) ->
    (rb => (rA => id(3*rA))(2*rb))(1*0) ->
    (rA => id(3*rA))(2*1*0) ->
    id(3*2*1*0)
function twice(f, x) {
    return f(f(x));
}

function cmp(f, g, x) {
    return f(g(x));
}
function twice(f, x) {
    return f(f(x));
}

function cmp(f, g, x) {
    return f(g(x));
}

to CPS, by example

function twice(f, x, cc) {
    f(x, r => f(r, cc));
}

function cmp(f, g, x, cc) {
    g(x, r => f(r, cc));
}
To CPS, by example

```javascript
function twice(f, x) {
    let r = f(x);
    return f(r);
}
```
To CPS, by example

```javascript
function twice(f, x) {
    let r = f(x);
    return f(r);
}
```

```javascript
function twice(f, x, cc) {
    f(x, r => f(r, cc));
}
```
To CPS, the rules

- Function decls take extra argument: the continuation
  
  \[ \text{function (x) \{ } \rightarrow \text{function (x, cc) \{ } \]

- There are no more returns! Call continuation instead
  
  \[ \text{return x; } \rightarrow \text{cc(x); } \]

- Lift nested function calls out of subexpressions
  
  \[ \text{let r = g(x); } \rightarrow \text{g(x, r => \{} \]
  \[ \text{stmt}_1 \rightarrow \text{stmt}_1 \]
  \[ \text{stmt}_2 \rightarrow \text{stmt}_1 ; \text{stmt}_2 \]
  \[ \text{\}) } \]
Why is this useful?

• Makes control flow explicit
  ➤ Compilers like this form since they can optimize code
• Multithreaded programming
• Event based programming such as GUIs
Continuations are extremely powerful

• Generalization of goto!

• Can implement control flow constructs using continuations

• How do we do if statements?

• How do we do exceptions?
Exceptions w/ continuations

1. function f() { throw "w00t"; } 
2. 
3. try {
4.   f();
5.   console.log("no way!");
6. } catch (e) {
7.   console.log(e);
8. }
9. console.log("cse130 is lit");
Exceptions w/ continuations

1. function f() { throw "w00t"; }
2.
3. try {
4.   f();
5.   console.log(“no way!”);
6. } catch (e) {
7.   console.log(e);
8. }
9. console.log(“cse130 is lit”);
success cont = line 5; previous cc = lines 5; 9

fail cont = lines 6-8; previous cc = lines 6-9

1. function f() { throw "w00t"; }
2.
3. try {
4.   f();
5.   console.log("no way!");
6. } catch (e) {
7.   console.log(e);
8. }
9. console.log("cse130 is lit");
Putting continuations in context

- Structured programming
- Procedural abstraction
- Exceptions
- Continuations
Week 8

• Structured programming
• Procedural abstraction
• Exceptions
• Continuations
• Monads
Can we do IO as usual?

\[
\text{ls :: } [(\cdot), (\cdot)] \\
\text{ls } = [\text{putChar 'x', putChar 'y']}
\]

Is this okay? A: yes, B: no
Laziness gets in the way?

• Depending on evaluation order, order of effects may vary or may not even be observed
  ➤ E.g., length `ls` vs. head `ls`

• Without laziness, are we okay? A: yes, B: no
Laziness gets in the way?

- Depending on evaluation order, order of effects may vary or may not even be observed.
  - E.g., `length ls` vs. `head ls`.

- Without laziness, are we okay? A: yes, B: no.
  - Laziness forces us to take a more principled approach!
Monad IO

• Extend category of values with actions

• A value of type (IO a) is an action

• When performed, the action of type IO a may perform some I/O before it delivers a result of type a

• How to think about actions:
Monad IO

• Extend category of values with actions

• A value of type \((\text{IO} \ a)\) is an action

• When performed, the action of type \(\text{IO} \ a\) may perform some I/O before it delivers a result of type \(a\)

• How to think about actions:
  
  ➤ type \(\text{IO} \ a = \text{World} \to (a, \text{World})\)
getChar :: IO Char
IO actions are first-class

• What does this mean? (Recall: first-class functions)
  ➢
  ➢
  ➢
IO actions are first-class

- What does this mean? (Recall: first-class functions)
  - Can return actions from function
  - Can pass actions as arguments
  - Can create actions in functions
putChar :: Char -> IO ()
How do we create actions?

• The return function:
  ➤ Worst name ever: has nothing to do with terminating early
  ➤ Given value produce IO action that doesn’t perform any IO and only delivers the value
  ➤ Type:
How do we create actions?

• The return function:
  ➤ Worst name ever: has nothing to do with terminating early
  ➤ Given value produce IO action that doesn’t perform any IO and only delivers the value
  ➤ Type: return :: a -> IO a
Example: return

• return 42

• $f(x) = \begin{cases} 
\text{return "what"} \\
\text{else return "no way!"}
\end{cases}$
How do we create actions?

• The compose function (>>>)
  ➤ Given an IO action $\text{act}_1$ and action $\text{act}_2$ produce a bigger action, which when executed:
    ➤ executes $\text{act}_1$
    ➤ execute $\text{act}_2$ and deliver the value produced by $\text{act}_2$
    ➤ Type:
How do we create actions?

• The compose function (>>>)

  ➤ Given an IO action \( \text{act}_1 \) and action \( \text{act}_2 \) produce a bigger action, which when executed:

  ➤ executes \( \text{act}_1 \)

  ➤ execute \( \text{act}_2 \) and deliver the value produced by \( \text{act}_2 \)

  ➤ Type: \((\gg\gg) : \text{IO} \ a \rightarrow \text{IO} \ b \rightarrow \text{IO} \ b\)
Example: >>

• return 42 >> putStrLn ‘A’ >> putStrLn ‘B’

• f x = putStrLnLn “hello world” >>
  if x == “hello”
    then return x
  else return “bye bye!”
How do we create actions?

• The bind function (>>=)
  ▶ Like (>>), but doesn’t drop the result of first action: it chains the result to the next action (which may use it)
  ▶ Type:

• Can we define (>>) in terms of (>>=)? A: yes, B: no
How do we create actions?

• The bind function (>>=)
  ➤ Like (>>), but doesn’t drop the result of first action: it chains the result to the next action (which may use it)
  ➤ Type: (>>=) :: IO a -> (a -> IO b) -> IO b

• Can we define (>>) in terms of (>>=)? A: yes, B: no
Recall:

➤ \((\gg\gg=) :: \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b\)

gg (\gg\gg) :: \text{IO } a \rightarrow \text{IO } b \rightarrow \text{IO } b

From this:

➤ \((\gg\gg) \text{ act1 act2} = \)
(\triangleright\triangleright\triangleright) \text{ via } (\triangleright\triangleright\triangleright\triangleright)\]

• Recall:

\begin{itemize}
  \item \( (\triangleright\triangleright\triangleright) :: \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b \)
  \item \( (\triangleright\triangleright) :: \text{IO } a \rightarrow \text{IO } b \rightarrow \text{IO } b \)
\end{itemize}

• From this:

\begin{itemize}
  \item \( (\triangleright\triangleright) \text{ act1 } \text{ act2} = \text{ act1 } \triangleright\triangleright\triangleright\triangleright \_ \rightarrow \text{ act2} \)
\end{itemize}
Example: >>=

- `return 42 >>= (\i -> putChar (chr i))`
- `echo :: IO ()
  echo =`
Example: >>=

- return 42 >>= (\i -> putChar (chr i))

- echo :: IO ()
  echo = getChar >>= (\c -> putChar c)
Example: >>=

- echoTwice :: IO ()
  echoTwice =

- getTwoChars :: IO (Char, Char)
  getTwoChars =
Example: >>==

- `echoTwice :: IO ()`
  ```haskell```
  echoTwice = getChar >>= \c ->
              putChar c >>= \_ ->
              putChar c
  ```

- `getTwoChars :: IO (Char, Char)`
  ```haskell```
  getTwoChars =
  ```
Example: $>>=\$

- **echoTwice :: IO ()**
  
  ```haskell```
  ```
  echoTwice = getChar >>= \c ->
    putChar c >>= \_ ->
    putChar c
  ```
  ```haskell```

- **getTwoChars :: IO (Char, Char)**
  
  ```haskell```
  ```
  getTwoChars = getChar >>= \c1 ->
    getChar >>= \c2 ->
    return (c1, c2)
  ```
  ```haskell```
Do notation

- Syntactic sugar to make it easier create big actions from small actions

```haskell
getTwoChars :: IO (Char, Char)
getTwoChars = do
    c1 <- getChar
    c2 <- getChar
    return (c1, c2)
```
Do notation: de-sugaring

- `do x <- e
  s` ➡

- `do e` ➡

- `do e
  s` ➡

- `do e` ➡
Do notation: de-sugaring

- $\text{do } x <- e$
  - $s$
  $\Rightarrow e >>= \lambda x \to \text{do } s$

- $\text{do } e$
  $\Rightarrow e >> \text{do } s$

- $\text{do } e$
  $\Rightarrow e$

How do we execute actions?

- Haskell program has to define main function
  ```haskell
  main :: IO ()
  ```
- To execute an action it has to be bound!
wc -l in Haskell
Mutable references in IO monad!

- `data IORef a = ....`
  - `readIORef ::`
  - `writeIORef ::`
  - `atomicModifyIORef ::`
Can we escape IO monad?

• Is it okay to define a function of type: IO a -> a

No! unsafePerformIO can be used to violate type safety
Monads are cool!

• Principled way to expose imperative programming in FP languages

• Evaluation order is explicit

• Idea goes beyond IO: you can define your own monad
  ➤ Monad is a type class (with return and >>=)
  ➤ E.g., LIO monad does security checks before performing, say, a readFile to prevent data leaks
Monads are a type class?

class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
hasmap.hs
Functor type class

class Functor f where
  fmap :: (a -> b) -> f a -> f b

• Laws

• What does this mean?
Functor type class

class Functor f where
    fmap :: (a -> b) -> f a -> f b

• Laws

  ▶ fmap id = id

  ▶ fmap (f . g) = fmap f . fmap g

• What does this mean?
Monad type class

class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b

• Laws

➤

➤

➤

• What does this mean?
Monad type class

class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

• Laws

  ▶ return a >>= k = k a

  ▶ m >>= return = m

  ▶ m >>= (\x -> k x >>= h) = (m >>= k) >>= h

• What does this mean?
Why do these matter?

- **Theorem:** \(\text{putStr } r \gg \text{putStr } s = \text{putStr } (r ++ s)\)

- Proof (base case):
Why do these matter?

• **Theorem:** putStr r >> putStr s = putStr (r ++ s)

• Proof (inductive case):
Example instance that’s not IO?

```haskell
instance Monad Maybe where
    return :: a -> Maybe a

    (>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
```