(sfp)

Strong Functional Programming

Without ⊥

`Without ⊥`

Turing Complete

`Without ⊥`

Turing Complete

Codata

Without $\perp$

Turing Complete

Codata

Comonad

Life without $\perp$

```
loop :: Int -> Int
loop n = 1 + loop n
```

Life without ⊥

```
loop :: Int -> Int
loop n = 1 + loop n
```

```
loop 0 = 1 + loop 0
```

Life without ⊥

```haskell
loop :: Int -> Int
loop n = 1 + loop n
```

```haskell
loop 0 = 1 + loop 0
```

```haskell
0 = 1
```

Life without ⊥

```
loop :: Int -> Int
loop n = 1 + loop n
```

```
loop 0 = 1 + loop 0
```

```
0 = 1
```

```
Int(⊥)
```

Simpler language design

Strict vs lazy

Simpler language design

Strict vs lazy

```
-- a function returning the first argument
first a b = a

-- with strict evaluation
first 1 ⊥ = ⊥

-- with lazy evaluation
first 1 ⊥ = 1
```

Simpler language design

Pattern matching

Simpler language design

Pattern matching

```
-- will not match if (a, b) is ⊥
first (a, b) = a

-- a bottom value can be "lifted" to a pair
   of bottom values
(⊥a, ⊥b) = ⊥
```

Simpler language design

& Operator

```
True  & True  = True
True  & False = False
False & True  = False
False & False = False
```

Simpler language design

& Operator

```
True  & True  = True
True  & False = False
False & True  = False
False & False = False
```

```
⊥ & y = ?
x & ⊥ = ?
```

Simpler language design

& Operator

```
True  & True  = True
True  & False = False
False & True  = False
False & False = False
```

```
⊥ & y = ?
x & ⊥ = ?
```

```
⊥ & y = ⊥
x & ⊥ = False if x = False
x & ⊥ = ⊥ otherwise
```

Simpler language design

Reduction

Reduction

```
a = true
if a then b else c      ==>    b
```

Simpler language design

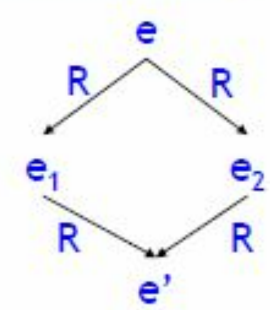
Reduction

```
a = true
if a then b else c        ==>    b
```

http://cseweb.ucsd.edu/classes/wi08/cse230/lectures/lec12.pdf

The Diamond Property

- Relation R has diamond property if:
  whenever e R e_1 and e R e_2 ,
  there exists e' such that e_1 R e' and e_2 R e'



Simpler language design

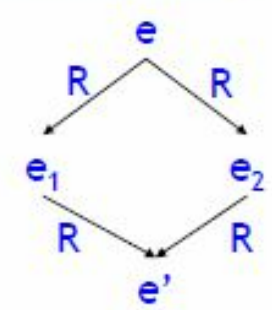
Reduction

```
a = true
if a then b else c      ==>    b
```

http://cseweb.ucsd.edu/classes/wi08/cse230/lectures/lec12.pdf

The Diamond Property

- Relation R has diamond property if:
  whenever e R e_1 and e R e_2 ,
  there exists e' such that e_1 R e' and e_2 R e'




```
-- is there always a normal form?
-- is it unique?
(YES <=> Strongly "Church-Rosser")
```

Life without ⊥

```
-- So far, so good
☺
```

Life without ⊥

```
-- So far, so good
☺
```

```
-- Not Turing complete!
-- Non termination?
```

Turing complete

L ⇔ Interpreter for L?

```
-- the interpreter
eval code input = result
-- the interpreter breaker
```

L ⇔ Interpreter for L?

```
-- the interpreter
eval code input = result
-- the interpreter breaker
evil code = 1 + eval code code
```

L ⇔ Interpreter for L?

```
-- the interpreter
eval code input = result
-- the interpreter breaker
evil code = 1 + eval code code

-- by definition of eval + evil "number"
eval 666 666 = evil 666
```

L ⇔ Interpreter for L?

```
-- the interpreter
eval code input = result
-- the interpreter breaker
evil code = 1 + eval code code

-- by definition of eval + evil “number”
eval 666 666 = evil 666
-- by definition of evil
evil 666 = 1 + (eval 666 666)
-- 'evil 666' ⇔  0 = 1
evil 666 = 1 + evil 666
```

Not Turing Complete

```
-- The rules of termination
```

Complete case analysis

Complete case analysis

```
-- taking the first element of a list
head a :: List a a -> a
head Nil default                = default
head (Cons a rest) default = a
```

Complete case analysis

```
-- taking the first element of a list
head a :: List a a -> a
head Nil default              = default
head (Cons a rest) default = a
```

```
data NonEmptyList a = NCons a (List a)

-- taking the first element of a
   non-empty list
head a :: NonEmptyList a -> a
head (NCons a rest) = a
```

Complete case analysis

Complete case analysis

```
-- Arithmetic operators?
1 / 0
0 / 0
```

Non-covariant type recursion

Non-covariant type recursion

```
data Silly a = Very (Silly a -> a)

bad a :: Silly a -> a
bad (Very f) = f (Very f)

-- infinite recursion, again…
ouch :: a
ouch = bad (Very bad)
```

Structural recursion

Structural recursion

```
factorial :: Nat -> Nat
factorial Zero          = 0
factorial (Suc Zero) = 1

-- we recurse with a sub-component
   of (Suc n)
factorial (Suc n)     = (Suc n) *
                        (factorial n)
```

Structural recursion

Structural recursion

```
-- Ackermann function
ack :: Nat Nat -> Nat
ack 0 n = n + 1

-- m + 1 is a shortcut for (Suc m)
ack (m + 1) 0       = ack m 1
ack (m + 1) (n + 1) = ack m (ack (m + 1) n)
```

Structural recursion

```
-- Ackermann function
ack :: Nat Nat -> Nat
ack 0 n = n + 1

-- m + 1 is a shortcut for (Suc m)
ack (m + 1) 0       = ack m 1
ack (m + 1) (n + 1) = ack m (ack (m + 1) n)
```

```
-- every provably terminating function
-- with first-order logic => a lot
```

Structural recursion

Structural recursion

```
-- Naive power function
pow :: Nat -> Nat ->
pow x n = 1,                        if n == 0
        = x * (pow x (n - 1)), otherwise
```

Structural recursion

```
-- Naive power function
pow :: Nat -> Nat ->
pow x n = 1,                              if n == 0
        = x * (pow x (n - 1)), otherwise
```

```
-- Faster
pow :: Nat -> Nat -> Nat
pow x n = 1,                              if n == 0
        = x * pow (x * x) (n / 2), if odd n
        = pow (x * x) (n / 2),        otherwise
```

Structural recursion

Structural recursion

```
-- representation of a binary digit
data Bit  = On | Off
-- built-in
bits :: Nat -> List Bit

-- primitive recursive now
pow :: Nat -> Nat -> Nat
pow x n = pow1 x (bits n)

pow1 :: Nat -> List Bit -> Nat
pow1 x n = 1
pow1 x (Cons On r)  = x * (pow1 (x * x) r)
pow1 x (Cons Off r) = pow1 (x * x) r
```

Codata for “infinite” computations

-- How to program an OS?

A new keyword

A new keyword

```
-- in Haskell
data Stream a = Cons a (Stream a)
```

A new keyword

```
-- in Haskell
data Stream a = Cons a (Stream a)
```

```
-- in SFP
-- (Cocons a rest) is in normal form
codata Colist a = Conil | a <> Colist a
```

A new rule

A new rule

```
-- functions on codata must always use a
-- coconstructor for their result
function a :: Colist a -> Colist a
function a <> rest = 'xxx' <> (function 'yyy')
```

A new rule

```
-- functions on codata must always use a
-- coconstructor for their result
function a :: Colist a -> Colist a
function a <> rest = 'xxx' <> (function ’yyy’)
```

```
-- looks familiar I suppose?
ones :: Colist Nat
ones = 1 <> ones

fibonacci :: Colist Nat
fibonacci = f 0 1
            where f a b =
                      a <> (fibonacci b (a + b))
```

A new proof mode

A new proof mode

```
-- iterate a function:
-- x, f x, f (f x), f (f (f x)),...
iterate f x = x <> iterate f (f x)

-- map a function on a colist
comap f Conil =  Conil
comap f a <> rest = (f a) <> (comap f rest)
```

A new proof mode

```
-- iterate a function:
-- x, f x, f (f x), f (f (f x)),...
iterate f x = x <> iterate f (f x)

-- map a function on a colist
comap f Conil =  Conil
comap f a <> rest = (f a) <> (comap f rest)
```

```
-- can you prove that?
iterate f (f x) = comap f (iterate f x)
```

A new proof mode

```
iterate f (f x)
-- 1. by definition of iterate
= (f x) <> iterate f (f (f x))

-- 2. by hypothesis
= (f x) <> comap f (iterate f (f x))

-- 3. by definition of comap
= comap f (x <> iterate f (f x))

-- 4. by definition of iterate
= comap f (iterate f x)
```

A new proof mode

```
iterate f (f x)
-- 1. by definition of iterate
= (f x) <> iterate f (f (f x))

-- 2. by hypothesis
= (f x) <> comap f (iterate f (f x))

-- 3. by definition of comap
= comap f (x <> iterate f (f x))

-- 4. by definition of iterate
= comap f (iterate f x)
```

Bisimilarity!

A new proof mode

```
iterate f (f x)
-- 1. by definition of iterate
= (f x) <> iterate f (f (f x))
                                        Bisimilarity!
-- 2. by hypothesis
= (f x) <> comap f (iterate f (f x))

-- 3. by definition of comap
= comap f (x <> iterate f (f x))

-- 4. by definition of iterate
= comap f (iterate f x)
```

Limitations

Limitations

```
-- not primary corecursive, but ok
evens = 2 <> (comap (+2) evens)
```

Limitations

```
-- not primary corecursive, but ok
evens = 2 <> (comap (+2) evens)
```

```
-- infinite lists
codata Colist a = a <> Colist a

cotail a :: Colist a -> Colist a
cotail a <> rest = rest

-- don't do this at home
bad = 1 <> (cotail bad)
```

Limitations

```
-- not primary corecursive, but ok
evens = 2 <> (comap (+2) evens)
```

```
-- infinite lists
codata Colist a = a <> Colist a

cotail a :: Colist a -> Colist a
cotail a <> rest = rest

-- don't do this at home
bad = 1 <> (cotail bad)
```

Count coconstructors!

A co-era is opening

```
extract :: W a -> a
cobind  :: W a -> b -> W a -> W b
```

A simple example

```
-- a Colist of Nats
nats = 0 <> comap (+1) nats

-- take the first 2 elements of a Colist
firstTwo a :: Colist a -> (a, a)
firstTwo a <> b <> rest = (a, b)

-- cobind firstTwo to nats
cobind firstTwo nats =
(0, 1) <> (1, 2) <> (2, 3) <> ...
```

Intuitions

```
-- State
-- “return a result based on an observable
state”
-- thread mutable state
State (s -> (s, a))

-- Costate
-- “return a result based on the internal
state and an external event”
-- aka ‘an Object’, ‘Store’
```

Intuitions

```
-- State
-- "return a result based on an observable state"
-- thread mutable state
State (s -> (s, a))

-- Costate
-- "return a result based on the internal state and an external event"
-- aka 'an Object', 'Store'
Costate (e, e -> a)
```