# Logic in Action

## Implementing and Understanding Programs

Paul Downen

April 15, 2021

# Problems in High-Assurance Programming

Consequences are huge, so correctness is paramount

Need to prove that programs "do the right thing"

E.g., Security protocols, private information management

Efficiency is often still a top concern

The right thing at the wrong time is still wrong!

If the answer comes too late, it doesn't matter

E.g., Automotive control systems, medical devices, high-speed network communication (Duff, OPLSS '18)

$$\begin{aligned} \text{propositions} &\approx \text{types} \\ \text{proofs} &\approx \text{programs} \end{aligned}$$

| Logic | Language |
|---|---|
| Natural deduction | $\lambda$-calculus |
| Proposition | Type |
| Proof | Program |

| Logic | Language |
| --- | --- |
| Natural deduction | $\lambda$-calculus |
| Proposition | Type |
| Proof | Program |
| Second-order quantification | Generics and modules |
| Classical logic | Control flow effects (call/cc) |
| $\vdots$ | $\vdots$ |

# Putting Logic to Work

1. Start with ideas from logic; find connections to computation
2. Use it to reason about program behavior
3. Apply it to compile programs better

# The Truth about Truth

## A non-constructive proof

**Theorem**
*There exist two irrational numbers, $x$ and $y$, such that $x^y$ is rational.*

**Proof.**

$\sqrt{2}$ is irrational, so consider $\sqrt{2}^{\sqrt{2}}$.

# A non-constructive proof

**Theorem**
*There exist two irrational numbers, $x$ and $y$, such that $x^y$ is rational.*

**Proof.**

$\sqrt{2}$ is irrational, so consider $\sqrt{2}^{\sqrt{2}}$.

$\sqrt{2}^{\sqrt{2}}$ is rational or not.

# A non-constructive proof

**Theorem**
*There exist two irrational numbers, $x$ and $y$, such that $x^y$ is rational.*

**Proof.**

$\sqrt{2}$ is irrational, so consider $\sqrt{2}^{\sqrt{2}}$.

$\sqrt{2}^{\sqrt{2}}$ is rational or not.

If it's rational, then $x = y = \sqrt{2}$. Done!

# A non-constructive proof

**Theorem**
*There exist two irrational numbers, $x$ and $y$, such that $x^y$ is rational.*

**Proof.**

$\sqrt{2}$ is irrational, so consider $\sqrt{2}^{\sqrt{2}}$.

$\sqrt{2}^{\sqrt{2}}$ is rational or not.

If it's rational, then $x = y = \sqrt{2}$. Done!

Otherwise, $\sqrt{2}^{\sqrt{2}}$ is irrational.

$$(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}^2} = \sqrt{2}^2 = 2$$

## A non-constructive proof

**Theorem**
*There exist two irrational numbers, $x$ and $y$, such that $x^y$ is rational.*

**Proof.**

$\sqrt{2}$ is irrational, so consider $\sqrt{2}^{\sqrt{2}}$.

$\sqrt{2}^{\sqrt{2}}$ is rational or not.

If it's rational, then $x = y = \sqrt{2}$. Done!

Otherwise, $\sqrt{2}^{\sqrt{2}}$ is irrational.

$$(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}^2} = \sqrt{2}^2 = 2$$

So $x = \sqrt{2}^{\sqrt{2}}$ and $y = \sqrt{2}$. Done! $\square$

# A non-constructive proof

**Theorem**
*There exist two irrational numbers, $x$ and $y$, such that $x^y$ is rational.*

**Proof.**

$\sqrt{2}$ is irrational, so consider $\sqrt{2}^{\sqrt{2}}$.

$\sqrt{2}^{\sqrt{2}}$ is rational or not.

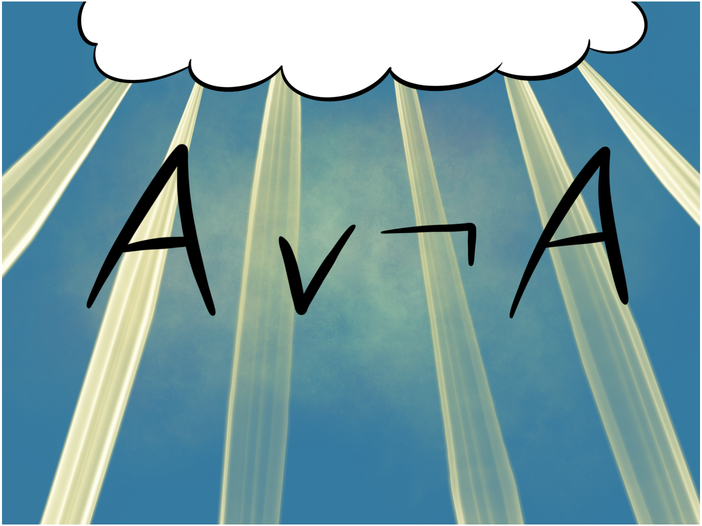
If it's rational, then $x = y = \sqrt{2}$. Done!

Otherwise, $\sqrt{2}^{\sqrt{2}}$ is irrational.

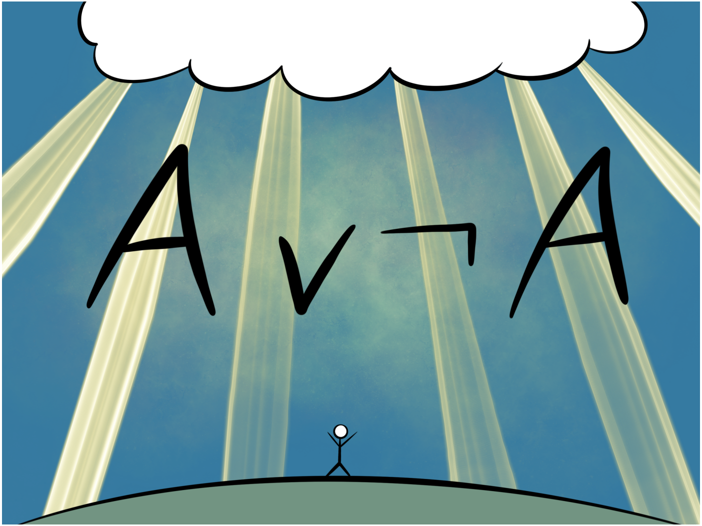
$$(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}^2} = \sqrt{2}^2 = 2$$

So $x = \sqrt{2}^{\sqrt{2}}$ and $y = \sqrt{2}$. Done! $\square$

## Truth is Divine



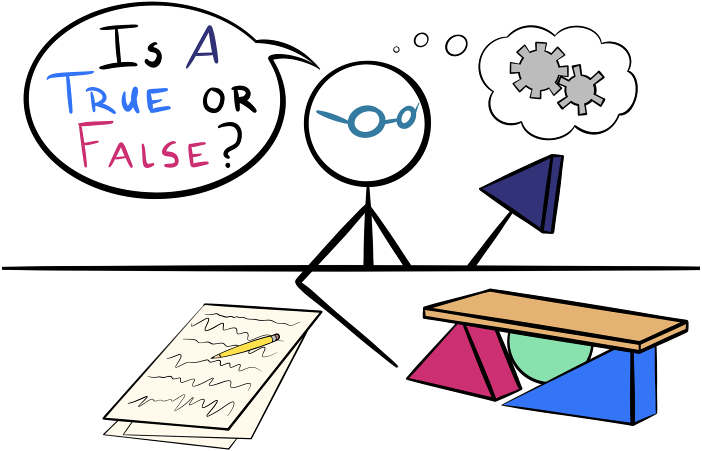
### Truth is Divine




…and sometimes out of reach to mortals

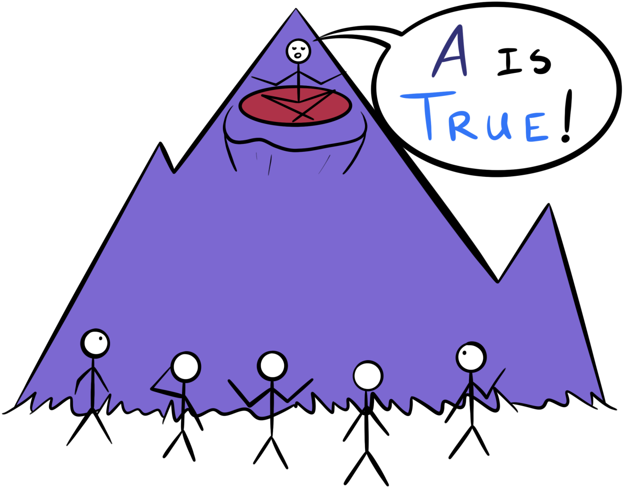
# Constructive Intuitionistic Logic

## Truth is the Work of Mortals



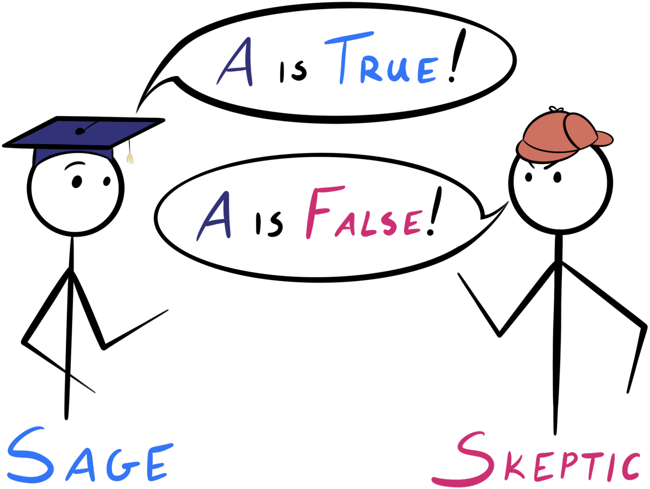
# The Monologue of the Sage

## Truth is Disseminated through Proclamations



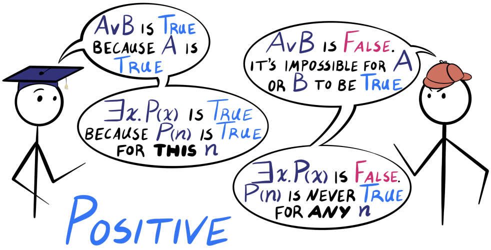
# The Dialogue of the Sage and the Skeptic

## Truth is Discovered through Debate



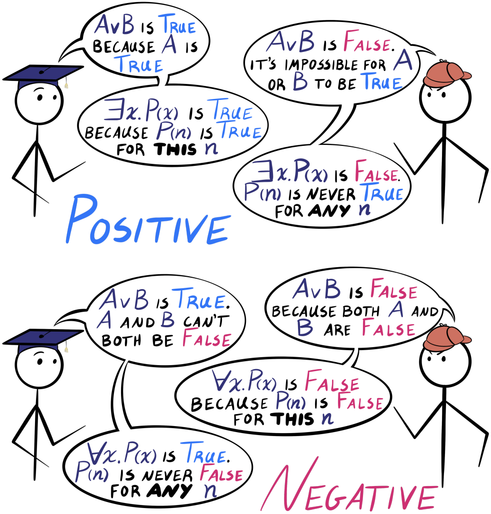
# Constructive Classical Logic

### Who Possesses The Burden of Proof?

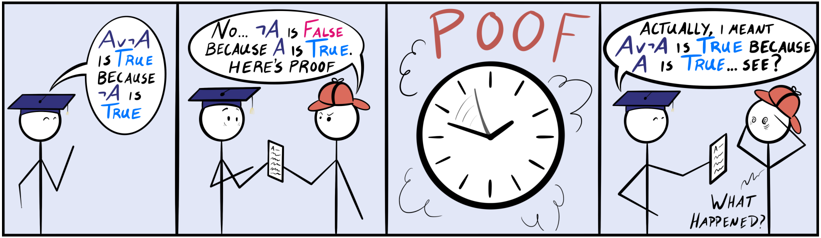


# Constructive Classical Logic

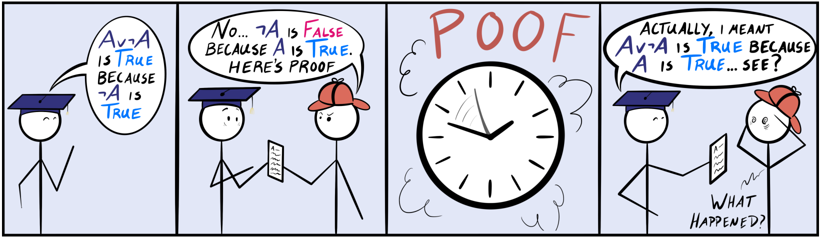
## Who Possesses The Burden of Proof?



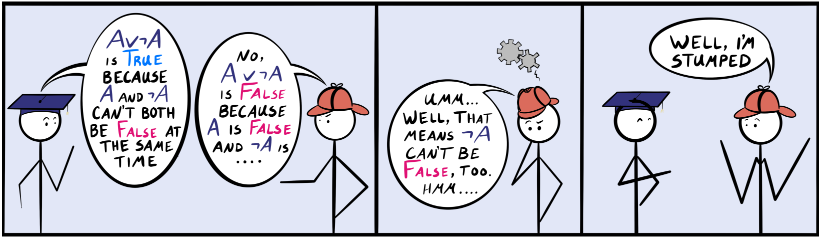
Excluded Middle with a Positive Mindset



Excluded Middle with a Positive Mindset




Excluded Middle with a Negative Mindset



# Duality in Practice

# Duality

"Co-things" are the opposite of "things"

De Morgan duals

$$\begin{aligned} \text{not true} &= \text{false} \\ \text{not false} &= \text{true} \end{aligned}$$

$$\begin{aligned} \text{not}(A \text{ and } B) &= (\text{not } A) \;\text{ or }\; (\text{not } B) \\ \text{not}(A \;\text{ or }\; B) &= (\text{not } A) \text{ and } (\text{not } B) \end{aligned}$$

$$A_1, A_2, \ldots A_n \vdash B_1, B_2, \ldots, B_m$$

means

$$A_1 \textit{ and } A_2 \textit{ and } \ldots \textit{ and } A_n$$

$$\Longrightarrow$$

$$B_1 \textit{ or } B_2 \textit{ or } \ldots \textit{ or } B_m$$

$$A_1, A_2, \ldots A_n \vdash B_1, B_2, \ldots, B_m$$

means

$$A_1 \textit{ and } A_2 \textit{ and } \ldots \textit{ and } A_n$$

$$\Longrightarrow$$

$$B_1 \textit{ or } B_2 \textit{ or } \ldots \textit{ or } B_m$$

| | |
|---|---|
| $\bullet \vdash A$ | means $A$ is true |
| $A \vdash \bullet$ | means $A$ is false |
| $\bullet \vdash \bullet$ | means contradiction |

| | |
|---|---|
| $\bullet \vdash A$ | means $A$ is true |
| $A \vdash \bullet$ | means $A$ is false |
| $\bullet \vdash \bullet$ | means contradiction |

| | |
|---|---|
| $\bullet \vdash P : A$ | is a producer of $A$ values |
| $C : A \vdash \bullet$ | is a consumer of $A$ values |
| $\langle P \| C \rangle : (\bullet \vdash \bullet)$ | is a runnable command |

Think: producer = sage, consumer = skeptic, command = dialogue

| | |
|---|---|
| $\bullet \vdash A$ | means $A$ is true |
| $A \vdash \bullet$ | means $A$ is false |
| $\bullet \vdash \bullet$ | means contradiction |

| | |
|---|---|
| $\bullet \vdash P : A$ | is a producer of $A$ values |
| $C : A \vdash \bullet$ | is a consumer of $A$ values |
| $\langle P \| C \rangle : (\bullet \vdash \bullet)$ | is a runnable command |

$$\langle P \| C \rangle : (x_1 : A_1 \dots x_n : A_n \vdash \alpha_1 : B_1 \dots \alpha_m : B_m)$$

is an open command

with free inputs $x_i$ and outputs $\alpha_j$

Think: producer = sage, consumer = skeptic, command = dialogue

$$\xRightarrow{\text{Answers}}$$

$$\langle P \| C \rangle$$

$$\xLeftarrow[\text{Questions}]{}$$

| | |
|---:|---|
| Producer | Consumer |
| Answers | Questions |
| Program | Context |

# Dualities of Computation

$$\xRightarrow{\text{Construction}}$$

$$\langle P \,\|\, C \rangle$$

$$\xLeftarrow[\text{Destruction}]{}$$

| Producer | Consumer |
|---:|:---|
| Answers | Questions |
| Program | Context |
| Construction | Destruction |

# Dualities of Computation

$$\xRightarrow{\text{Data Flow}}$$

$$\langle P \| C \rangle$$

$$\xLeftarrow[\text{Control Flow}]{}$$

| Producer | Consumer |
| --- | --- |
| Answers | Questions |
| Program | Context |
| Construction | Destruction |
| Data Flow | Control Flow |

# Dualities of Computation

$$\langle P \| C \rangle$$

$\xrightarrow{\text{Modules}}$

$\xleftarrow[\text{Generics}]{}$

| | |
|---|---|
| Producer | Consumer |
| Answers | Questions |
| Program | Context |
| Construction | Destruction |
| Data Flow | Control Flow |
| Generics | Modules |

# Classical Logic and Control[1]

Classical logic $\cong \lambda\mu = \lambda$-calculus + labels + jumps

Corresponds to Scheme's call/cc control operator

- $A \vee \neg A$ as application of call/cc
- "time travel" caused by invoking the continuation

Producer $\neq$ command:

- Producers return a value
- Commands don't return, they jump

Delimited control is much more expressive

- Can represent any (monadic) side effect

Delimited control is $\lambda\mu$ where expression = command

[1] Downen, Ariola, ICFP '14

# Data vs Codata[2]

$$\begin{aligned}&\textbf{data}\ a \oplus b\ \textbf{where}\\ &\qquad \mathrm{Left} : a \vdash a \oplus b\\ &\qquad \mathrm{Right} : b \vdash a \oplus b\end{aligned}$$

$$\begin{aligned}&\textbf{codata}\ a \,\&\, b\ \textbf{where}\\ &\qquad \mathrm{First} : a \,\&\, b \vdash a\\ &\qquad \mathrm{Second} : a \,\&\, b \vdash b\end{aligned}$$

[2] Downen & Ariola, ESOP '14

# Data vs Codata[2]

$$\textbf{data}\ a \oplus b\ \textbf{where}$$
$$\mathrm{Left} : a \vdash a \oplus b$$
$$\mathrm{Right} : b \vdash a \oplus b$$

$$\textbf{codata}\ a \mathbin{\&} b\ \textbf{where}$$
$$\mathrm{First} : a \mathbin{\&} b \vdash a$$
$$\mathrm{Second} : a \mathbin{\&} b \vdash b$$

$$\textbf{data}\ a \otimes b\ \textbf{where}$$
$$\mathrm{Pair} : a, b \vdash a \otimes b$$

$$\textbf{codata}\ a \parr b\ \textbf{where}$$
$$\mathrm{Split} : a \parr b \vdash a, b$$

[2] Downen & Ariola, ESOP ’14

# Data vs Codata[2]

$$\textbf{data}\ a \oplus b\ \textbf{where}$$
$$\mathsf{Left} : a \vdash a \oplus b$$
$$\mathsf{Right} : b \vdash a \oplus b$$

$$\textbf{codata}\ a \,\&\, b\ \textbf{where}$$
$$\mathsf{First} : a \,\&\, b \vdash a$$
$$\mathsf{Second} : a \,\&\, b \vdash b$$

$$\textbf{data}\ a \otimes b\ \textbf{where}$$
$$\mathsf{Pair} : a, b \vdash a \otimes b$$

$$\textbf{codata}\ a \parr b\ \textbf{where}$$
$$\mathsf{Split} : a \parr b \vdash a, b$$

$$\textbf{data}\ a \ominus b\ \textbf{where}$$
$$\mathsf{Yield} : a \vdash a \ominus b, b$$

$$\textbf{codata}\ a \to b\ \textbf{where}$$
$$\mathsf{Call} : a, a \to b \vdash b$$

[2] Downen & Ariola, ESOP '14

# Induction vs Coinduction[3]

Induction is a bottom-up, divide-and-conquer approach:

$$
\begin{array}{rrl}
\multicolumn{3}{l}{\mathbf{data}\ \mathsf{List}\ a\ \mathbf{where}} \\
\mathsf{Nil} : & \bullet \vdash & \mathsf{List}\ a \\
\mathsf{Cons} : & a, \mathsf{List}\ a \vdash & \mathsf{List}\ a
\end{array}
\qquad\qquad
\begin{array}{rrl}
\multicolumn{3}{l}{\mathbf{data}\ \mathsf{Nat}\ \mathbf{where}} \\
\mathsf{Zero} : & \bullet \vdash & \mathsf{Nat} \\
\mathsf{Succ} : & \mathsf{Nat} \vdash & \mathsf{Nat}
\end{array}
$$

$$
\begin{aligned}
\mathit{length}(\mathsf{Nil}) &= \mathsf{Zero} \\
\mathit{length}(\mathsf{Cons}(x, xs)) &= \mathsf{Succ}(\mathit{length}(xs))
\end{aligned}
$$

[3] Downen, Johnson-Freyd, Ariola, ICFP ’15

# Induction vs Coinduction[3]

Induction is a bottom-up, divide-and-conquer approach:

$$\begin{aligned}&\textbf{data}\ \mathsf{List}\ a\ \textbf{where}\\ &\quad \mathsf{Nil} : \quad \bullet \vdash \mathsf{List}\ a\\ &\quad \mathsf{Cons} : \quad a, \mathsf{List}\ a \vdash \mathsf{List}\ a\end{aligned} \qquad \begin{aligned}&\textbf{data}\ \mathsf{Nat}\ \textbf{where}\\ &\quad \mathsf{Zero} : \quad \bullet \vdash \mathsf{Nat}\\ &\quad \mathsf{Succ} : \quad \mathsf{Nat} \vdash \mathsf{Nat}\end{aligned}$$

$$\begin{aligned}\mathit{length}(\mathsf{Nil}) &= \mathsf{Zero}\\ \mathit{length}(\mathsf{Cons}(x, xs)) &= \mathsf{Succ}(\mathit{length}(xs))\end{aligned}$$

Coinduction is a top-down, demand-driven approach

$$\mathit{count}(0) = 0, 1, 2, \ldots \qquad \mathit{count}(x) = x, \mathit{count}(x+1)$$

[3] Downen, Johnson-Freyd, Ariola, ICFP '15

# Induction vs Coinduction[3]

Induction is a bottom-up, divide-and-conquer approach:

$$\begin{array}{rrl} \multicolumn{3}{l}{\textbf{data}\ \mathsf{List}\ a\ \textbf{where}} \\ \mathsf{Nil} : & \bullet \vdash & \mathsf{List}\ a \\ \mathsf{Cons} : & a, \mathsf{List}\ a \vdash & \mathsf{List}\ a \end{array} \qquad \begin{array}{rrl} \multicolumn{3}{l}{\textbf{data}\ \mathsf{Nat}\ \textbf{where}} \\ \mathsf{Zero} : & \bullet \vdash & \mathsf{Nat} \\ \mathsf{Succ} : & \mathsf{Nat} \vdash & \mathsf{Nat} \end{array}$$

$$\begin{aligned} \mathit{length}(\mathsf{Nil}) &= \mathsf{Zero} \\ \mathit{length}(\mathsf{Cons}(x, \mathit{xs})) &= \mathsf{Succ}(\mathit{length}(\mathit{xs})) \end{aligned}$$

Coinduction is a top-down, demand-driven approach

$$\mathit{count}(0) = 0, 1, 2, \ldots \qquad \mathit{count}(x) = x, \mathit{count}(x + 1)$$

$$\begin{array}{rrl} \multicolumn{3}{l}{\textbf{codata}\ \mathsf{Stream}\ a\ \textbf{where}} \\ \mathsf{Head} : & \mathsf{Stream}\ a \vdash & a \\ \mathsf{Tail} : & \mathsf{Stream}\ a \vdash & \mathsf{Stream}\ a \end{array}$$

$$\begin{aligned} \mathit{count}(x).\mathsf{Head} &= x \\ \mathit{count}(x).\mathsf{Tail} &= \mathit{count}(\mathsf{Succ}(x)) \end{aligned}$$

[3] Downen, Johnson-Freyd, Ariola, ICFP '15

# Functional vs Object-oriented[4]

```
record Stream A : Set where
  coinductive
  field head : A
        tail : Stream A
```

```
count : Nat → Stream Nat
head (count x) = x
 tail (count x) = count (x + 1)
```

[4] Downen & Ariola, *Classical (Co)Recursion: Programming*, 2021

# Functional vs Object-oriented[4]

```
record Stream A : Set where
  coinductive
  field head : A
        tail : Stream A
```

```
count : Nat → Stream Nat
head (count x) = x
 tail (count x) = count (x + 1)
```

```
public interface Stream⟨A⟩ {
    public A           head ();
    public Stream⟨A⟩  tail ();
}
public class Count implements Stream⟨Integer⟩ {
    private final Integer first ;
    public Count(Integer x)         { this. first = x; }
    public Integer          head()  { return this. first ; }
    public Stream⟨Integer⟩  tail () { return new Count(this. first +1); }
}
```

[4] Downen & Ariola, *Classical (Co)Recursion: Programming*, 2021

# Codata in Programming[5]

Codata integrates features of functional & OO languages

- First-class functions are codata
- Objects are codata

Codata connects methods of functional & OO programming

- Church Encodings are the Visitor Pattern

Codata captures several functional & OO design techniques

- Demand-driven programming
- Procedural abstraction
- Pre- and Post-Conditions

Codata improves $\lambda$-calculus theory (JDA WoC'16; JDA JFP'17)

[5] Downen, Sullivan, Ariola, Peyton Jones, ESOP '19

# Coinduction in Programming[6]

Induction represents terminating, batch-processing algorithm

Coinduction naturally represents interactive, infinite processes

- "Online" streaming algorithms & network telemetry
- Interactive programs, user interfaces, & web servers
- Operating systems & real-time systems

Instead of termination, productivity is important

- Service is always available, indefinitely
- Process ends only when client is done

Induction & coinduction are both structural recursion (ICFP'15)

- Induction follows structure of values (producers)
- Coinduction follows structure of contexts (consumers)

Coinductive hypothesis follows control flow (PPDP'20)

- Dual to induction following information flow

[6] Downen, Johnson-Freyd, Ariola, ICFP '15, Downen & Ariola, PPDP '20

# Orthogonal models of safety[7]

Domain-specific notion of safety: set of commands $\perp\!\!\!\perp$

Safe interaction is orthogonality

Individuals $P \perp\!\!\!\perp C \iff \langle P \| C \rangle \in \perp\!\!\!\perp$

Groups: $A^{+} \perp\!\!\!\perp A^{-} \iff \forall P \in A^{+}, C \in A^{-}.\ P \perp\!\!\!\perp C$

Adjoint duality: $A^{\perp\!\!\!\perp}$ is biggest $B$ s.t. $A \perp\!\!\!\perp B$ or $B \perp\!\!\!\perp A$

Types are fixed points: $A = (A^{+}, A^{-}) = (A^{-\perp\!\!\!\perp}, A^{+\perp\!\!\!\perp}) = A^{\perp\!\!\!\perp}$

$\perp\!\!\!\perp$ = type safety, termination, consistency, equivalence, ...

Handles many features of advanced & practical languages:

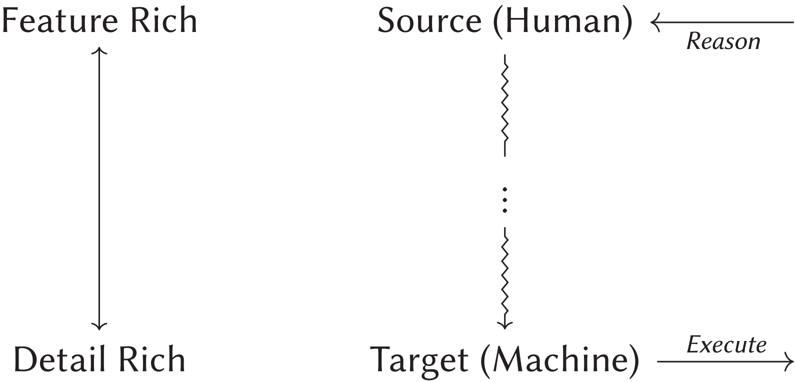
Linearity, effects, (co)recursion (DA, CSL'18), subtyping (DJA, WRLA'18), dependent types (DJA, ICFP'15), intersection & union types (DAG, FI'19)

Non-determinism and alternative evaluation orders via asymmetric orthogonality and the (co)value restriction

[7] Downen, Johnson-Freyd, Ariola, JLAMP '19; Downen, Johnson-Freyd, Ariola, WRLA '18

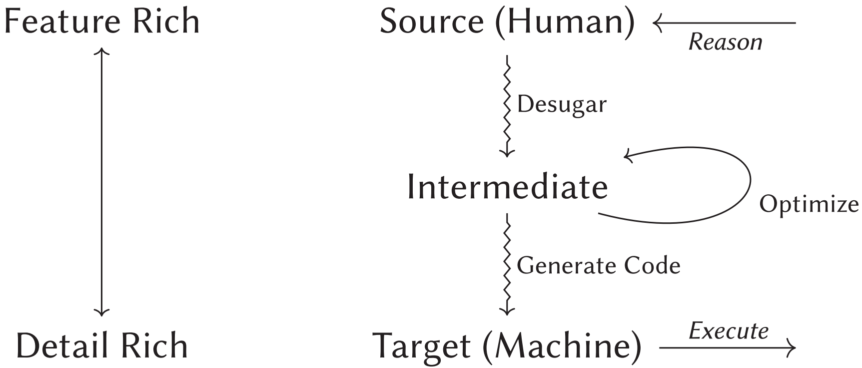
# Logic of Compilation

# The Life-cycle of a Program

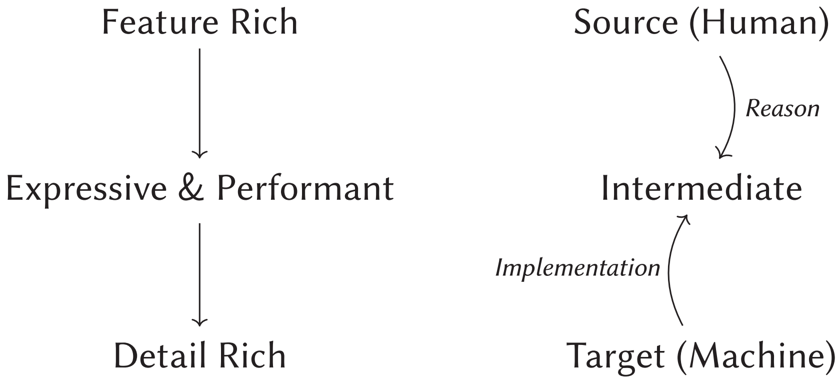



But this is a big jump; what goes in the middle?

# Intermediate Languages



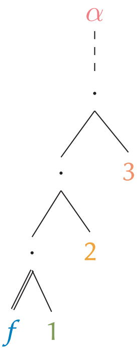
# The Two-Way Street of Influence



$$
\begin{tikzcd}
\text{Feature Rich} \arrow[d] \\
\text{Expressive \& Performant} \arrow[d] \\
\text{Detail Rich}
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\text{Source (Human)} \arrow[d, bend left, "\textit{Reason}"] \\
\text{Intermediate} \arrow[u, dashed, bend left, "\textit{Implementation}"] \arrow[d, dashed, bend left, "\textit{Reason}"] \\
\text{Target (Machine)} \arrow[u, bend left, "\textit{Implementation}"]
\end{tikzcd}
$$

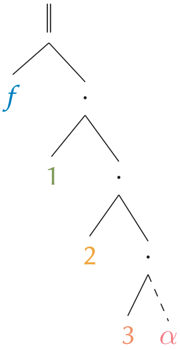
$\lambda$-calculus

$f\ 1\ 2\ 3 \ldots$




Sequent calculus

$\langle f \| 1 \cdot 2 \cdot 3 \cdot \alpha \rangle$



# Sequent Calculus as an Intermediate Language[8]

Bring the main action of a program to center stage

Similar to continuation-passing style (CPS) and static single assignment (SSA), but …

- Function calls are concrete, better for optimization
- Appropriate for both functional and imperative code

Gives an explicit representation of control flow

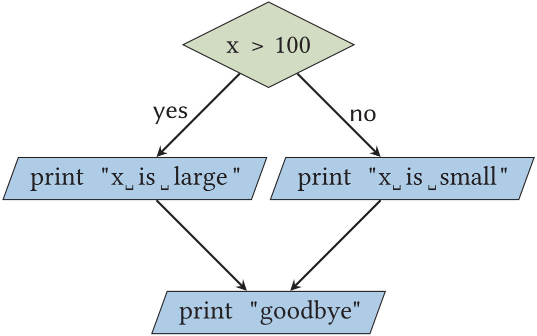
Shows how to implement codata

Helps to formalize and optimize calling conventions

[8] Downen, Maurer, Ariola, Peyton Jones, ICFP ’15; Downen, Ariola, JFP ’16

# Join Points in Control Flow

```
if x > 100 :
  print "x is large"
else :
  print "x is small"
print "goodbye"
```



# Purely Functional Join points[9]

Some optimizations follow control flow, not data flow

If careless, potential exponential blowup of code size

Join points are found in SSA and CPS, in different forms

Classical logic can represent join points in direct style

Classical-Intuitionistic hybrid gives join points while maintaining purity

[9] Maurer, Downen, Ariola, Peyton Jones, PLDI '17

## The Duality of Evaluation

$f(1+1)$: is $1+1$ done before or after call?

$$\xRightarrow{\text{Data Flow: CBN}}$$

$$\langle P \| C \rangle$$

$$\xLeftarrow[\text{Control Flow: CBV}]{}$$

Call-by-value favors producer $P$; follows control flow first

Call-by-name favors consumer $C$; follows data flow first

# Polarization Hypothesis

$$\xRightarrow{\text{Data Flow: Answers}}$$

$$\langle P \| C \rangle$$

$$\xLeftarrow[\text{Control Flow: Questions}]{}$$

**Positive: CBV Data Types**

| Answer | Question |
|---|---|
| Primary | Secondary |
| Action | Reaction |
| Concrete | Abstract |
| Finite | Infinite |

e.g., lists, trees, structures, …

**Negative: CBN Codata Types**

| Answer | Question |
|---|---|
| Secondary | Primary |
| Reaction | Action |
| Abstract | Concrete |
| Infinite | Finite |

e.g., functions, streams, processes, …

Think: Positive vs Negative burden of proof

# Polarity in Intermediate Languages

Dual (adjoint) language: "universal" IL for CBV and CBN

User-defined types encoded into finite set of primitives

- Purely functional (Downen & Ariola, CSL '18)
- Perfectly dual (Downen & Ariola, LMCS '20)

Encodings have same properties as source program

Must be robust in the face of computational effects

Going beyond polarity, for call-by-need, etc., requires only four extra "polarity shifts"

# Efficient Calling Conventions

Systems languages give fine-grained calling conventions:

- Fixed number of parameters
- Boxed (call-by-reference) versus unboxed (call-by-value)
- Many shapes (integer vs floating point vs arrays)
- All checks done statically at compile time

Functional languages make efficient calls difficult:

- Currying: $a \to (b \to c)$ instead of $(a, b) \to c$
- Polymorphism: $\forall a.a \to a$; is $a = \mathrm{Int}$ or $a = \mathrm{Int} \to \mathrm{Int}$?
- Pervasive Boxing: due to polymorphism or laziness

# Kinds are Calling Conventions[10]

Polarity points out types of efficient machine primitives

Hindsight: unboxed data must be positive (PJ&L, FPLCA'91)

Primitive function types must be negative (DSAP, Haskell'19)

Polarized types are so well-behaved they fuse together

- Unboxed tuples combine into a single structure
- Currying recomposes into single multi-arity function

Implementation details stored statically in types & kinds

- How many bits? Where are they stored?
- How can you use this object?
- When do you run this code?

*Kinds: the type system of the machine*

[10] Downen, Ariola, Peyton Jones, Eisenberg, ICFP '20

# Conclusion

# Summary

1. Translate an idea from logic to computation
2. Use it to understand program behavior
3. Apply it to implement programs more efficiently

# Lessons Learned

Curry-Howard is the gift that keeps giving

Good for theory of programming

- Proving properties
- Verifying correctness
- Designing programs

Good for practice of compilation

- Express low-level details in high-level representation
- Reason about performance
- Formalize and develop new optimizations

# Future Work

# A Dual Programming Language

Through the lens of duality, the two main paradigms are:

- Object-oriented: richness of codata types, paucity of data
- Functional: richness of data types, paucity of codata

Codata already captures many important OO principles

- Interfaces, encapsulation, dynamic dispatch, subtyping

Concurrency is modeled through communicating agents

- Session types specify concurrent protocols
- Linearity controls limited resources

Duality expresses communication between a server and client

Goal: Dual programming language fusing high- & low-level, functional & OO, sequential & concurrent programming

# The Duality of Information Security

Confidentiality (who knows?) & integrity (says who?) are dual

public $\sqsubseteq$ private yet trusted $\sqsubseteq$ untrusted

*"That duality is what makes security hard"* – Myers OPLSS '17

Both are dependent on data flow and control flow

$$\textbf{private}\ \text{bool}\ \ \text{secret}\ ;$$
$$\textbf{if}\ \ \text{secret}\ \ \{\ \textbf{return true};\ \}\ \ \textbf{else}\ \ \{\ \textbf{return false};\ \}$$

Are sensitivity & privacy dual? (co)effects, adjoint languages (Near et al., OOPSLA'19)

Can differential privacy be decomposed into orthogonality?

$M \perp\!\!\!\perp_{\epsilon,\delta} M'$ iff $\forall S \subseteq \mathbb{R}$, $\Pr[M \in S] \leq e^{\epsilon} \Pr[M' \in S] + \delta$

$x\ \ \mathrm{DB}_1\ \ y$ iff databases $x, y$ differ by 1 row

$\epsilon, \delta$-differentially private algorithm: ${\mathrm{DB}_1}^{\perp\!\!\!\perp_{\epsilon,\delta}}$

Hypothesis: Orthogonality gives a robust model for the dualities of information security

# A Logical Foundation of Compiler Correctness

Old: Compiling & running whole programs give right answer

Problems with whole-program correctness:

- Cannot link with system libraries
- No foreign-function interface
- Poor modularity and separate compilation

Compositional compiler correctness: Compiling part of a program and linking with a valid context gives the right answer

- Context $C \in A$ in target; program $P \in A^{\perp\!\!\!\perp}$ in source

Other properties (e.g., privacy and security) could be modeled as compositional correctness criteria preserved by compiler

Hypothesis: sequent calculus gives a logical framework for compositional compiler correctness & security

# Thank You

# Structural (Co)Induction

# Unifying (Co)Induction as Structural Recursion[11]

A call stack $\boxed{x \cdot \alpha}$ contains an:

argument $x$

return pointer $\alpha$

*length* is well-founded because its argument shrinks:

$$\begin{aligned}
\langle \mathit{length} \| \mathrm{Nil} \cdot \alpha \rangle &= \langle \mathrm{Zero} \| \alpha \rangle \\
\langle \mathit{length} \| \boxed{\mathrm{Cons}\ x\ xs} \cdot \alpha \rangle &= \langle \mathit{length} \| \boxed{xs} \cdot \mathrm{Succ} \circ \alpha \rangle
\end{aligned}$$

*count* is well-founded because its return pointer shrinks:

$$\begin{aligned}
\langle \mathit{count} \| x \cdot \mathrm{Head}\ \alpha \rangle &= \langle x \| \alpha \rangle \\
\langle \mathit{count} \| x \cdot \boxed{\mathrm{Tail}\ \alpha} \rangle &= \langle \mathit{count} \| \mathrm{Succ}\ x \cdot \boxed{\alpha} \rangle
\end{aligned}$$

[11] Downen, Johnson-Freyd, Ariola, ICFP '15

# Inductive Reasoning

$$\frac{\bullet \vdash P(\mathrm{True}) \qquad \bullet \vdash P(\mathrm{False})}{x : \mathrm{Bool} \vdash P(x)}$$

# Inductive Reasoning

$$\frac{\bullet \vdash P(\mathrm{True}) \qquad \bullet \vdash P(\mathrm{False})}{x : \mathrm{Bool} \vdash P(x)}$$

$$\frac{\bullet \vdash P(0) \qquad \bullet \vdash P(1) \qquad \bullet \vdash P(2) \qquad \ldots}{x : \mathrm{Nat} \vdash P(x)}$$

$$\frac{\bullet \vdash P(\mathrm{True}) \qquad \bullet \vdash P(\mathrm{False})}{x : \mathrm{Bool} \vdash P(x)}$$

$$\frac{\bullet \vdash P(0) \qquad \bullet \vdash P(1) \qquad \bullet \vdash P(2) \qquad \ldots}{x : \mathrm{Nat} \vdash P(x)}$$

$$\frac{\bullet \vdash P(0) \qquad y : \mathrm{Nat}, P(y) \vdash P(y+1)}{x : \mathrm{Nat} \vdash P(x)}$$

# Coinductive Reasoning[13]

$$\frac{x : \mathrm{Stream}\,A,\ P(x) \vdash P(x)}{x : \mathrm{Stream}\,A \vdash P(x)}\ \textit{warning!}$$

[12] Read $\alpha \div A$ as $\alpha : -A$, i.e., an assumption of not $A$, a continuation expecting $A$.

[13] Downen & Ariola, PPDP '20

# Coinductive Reasoning[13]

$$\dfrac{x : \mathrm{Stream}\,A, P(x) \vdash P(x)}{x : \mathrm{Stream}\,A \vdash P(x)}\ \textit{warning!}$$

$$\dfrac{x : \mathrm{Stream}\,A \vdash P\begin{pmatrix} x.\mathrm{Head}, x.\mathrm{Tail}.\mathrm{Head}, \\ x.\mathrm{Tail}.\mathrm{Tail}.\mathrm{Head}, \ldots \end{pmatrix}}{x : \mathrm{Stream}\,A \vdash P(x)}$$

[12] Read $\alpha \div A$ as $\alpha : -A$, i.e., an assumption of not $A$, a continuation expecting $A$.

[13] Downen & Ariola, PPDP '20

# Coinductive Reasoning[13]

$$\frac{x : \mathrm{Stream}\, A, P(x) \vdash P(x)}{x : \mathrm{Stream}\, A \vdash P(x)}\ \mathit{warning}!$$

$$\frac{x : \mathrm{Stream}\, A \vdash P\begin{pmatrix} x.\mathrm{Head}, x.\mathrm{Tail}.\mathrm{Head}, \\ x.\mathrm{Tail}.\mathrm{Tail}.\mathrm{Head}, \ldots \end{pmatrix}}{x : \mathrm{Stream}\, A \vdash P(x)}$$

$$\frac{\alpha \div A \vdash P(\mathrm{Head}\,\alpha) \quad \alpha \div A \vdash P(\mathrm{Tail}[\mathrm{Head}\,\alpha]) \quad \ldots}{\gamma \div \mathrm{Stream}\, A \vdash P(\gamma)}$$ [12]

[12] Read $\alpha \div A$ as $\alpha : -A$, i.e., an assumption of not $A$, a continuation expecting $A$.

[13] Downen & Ariola, PPDP '20

# Coinductive Reasoning[13]

$$\frac{x : \mathrm{Stream}\,A, P(x) \vdash P(x)}{x : \mathrm{Stream}\,A \vdash P(x)}\ \textit{warning!}$$

$$\frac{x : \mathrm{Stream}\,A \vdash P\begin{pmatrix} x.\mathrm{Head}, x.\mathrm{Tail}.\mathrm{Head}, \\ x.\mathrm{Tail}.\mathrm{Tail}.\mathrm{Head}, \ldots \end{pmatrix}}{x : \mathrm{Stream}\,A \vdash P(x)}$$

$$\frac{\alpha \div A \vdash P(\mathrm{Head}\,\alpha) \quad \alpha \div A \vdash P(\mathrm{Tail}[\mathrm{Head}\,\alpha]) \quad \ldots}{\gamma \div \mathrm{Stream}\,A \vdash P(\gamma)}\ ^{12}$$

$$\frac{\alpha \div A \vdash P(\mathrm{Head}\,\alpha) \quad \beta \div \mathrm{Stream}\,A, P(\beta) \vdash P(\mathrm{Tail}\,\beta)}{\gamma \div \mathrm{Stream}\,A \vdash P(\gamma)}$$

[12] Read $\alpha \div A$ as $\alpha : -A$, i.e., an assumption of not $A$, a continuation expecting $A$.

[13] Downen & Ariola, PPDP '20

# Control Flow

# Intuitionistic vs Classical Logic

Intuitionistic logic $\subset$ Classical logic

Intuitionistic logic rejects the following classical laws:

Excluded Middle: $A \vee \neg A$ (either $A$ or not $A$ is true)

Double Negation: $\neg\neg A \implies A$ (if not not $A$ is true, so is $A$)

Pierce's Law: $((A \implies B) \implies A) \implies A$

# Manipulating the Flow of Control

Control operators let the programmer manipulate control flow

These bind continuations that are the "rest of the computation"

Scheme's call/cc: $((A \to B) \to A) \to A$

Felleisen's $\mathcal{C}$: $\neg\neg A \to A$ (where $\neg A$ is a continuation)

Ambiguous choice: $A + \neg A$ (either a value or continuation)

# A Natural Extension of Classical Logic[14]

Parigot's classical $\lambda\mu$ = $\lambda$-calculus + labels + jumps

Expression $\neq$ command:

Expressions return a value

Commands don't return, they jump

Corresponds to call/cc

Delimited control is much more expressive

Can represent any (monadic) side effect

Delimited control is $\lambda\mu$ where expression = command

[14] Downen, Ariola, ICFP '14

# Delimited Control as (Resumable) Exceptions

```
def square_root(x):
    if x <= 0:
        raise ValueError("square root must be positive")
    ...

try:
    x = input("Please enter a number: ")
    print(square_root(int(x)))
except ValueError:
    print("That's not a valid number")
```

# Delimited Control as Coroutines

```
def depth_first_search(tree):
    if type(tree) is list:
        for child in tree:
            yield from depth_first_search(child)
    else:
        yield tree

def print_dfs(tree):
    for elem in depth_first_search(tree):
        print(elem)
```

```
print_dfs([[1], 2, [[3, 4], 5], [[6]]]) => 1, 2, 3, 4, 5, 6
```

# Delimited Control as Dynamic Labels[15]

Practical programs should be modular

Interference between side effects should be avoided

- E.g., exception handling
- Was the exception in parsing input, or processing value?

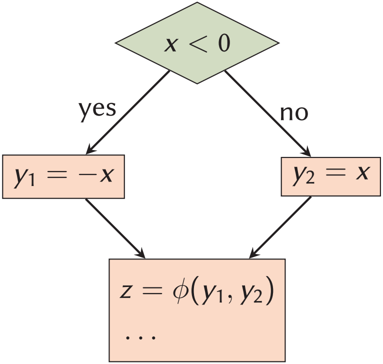
Solved by multiple control delimiters:

- A delimiter is a dynamically-bound label
- Different labels denote separate scopes

[15] Downen, Ariola, ESOP '12; Downen, Ariola JFP '14

# Join Points

# Join Points versus $\phi$-nodes




**label** $j(z) = \dots$
**in if** $x < 0$
  **then jump** $j(-x)$
  **else jump** $j(x)$

# Sequent Calculus

Natural Deduction

$$\frac{A \quad B}{A \wedge B}$$

Sequent calculus

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}$$

| Natural Deduction | Sequent calculus |
|---|---|
| $\dfrac{A \wedge B}{A}$ | $\dfrac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}$ |
| $\dfrac{A \wedge B}{B}$ | $\dfrac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}$ |

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \qquad \frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \vee B, \Delta}$$

# Curry-Howard

# All Natural Numbers are Even or Odd

What is even?

$$n = 2k$$

What is odd?

$$n = 2k + 1$$

Proof by induction…

$$\begin{aligned} 0 &= 2(0)\text{: even!} \\ 1 &= 2(0) + 1\text{: odd!} \\ n+1 &\text{ by inductive hypothesis, } n \text{ is:} \end{aligned}$$

$$\begin{aligned} 2k &\text{ then } n + 1 = 2k + 1\text{: odd!} \\ 2k+1 &\text{ then } n + 1 = 2k + 1 + 1 = 2(k+1)\text{: even!} \end{aligned}$$

# (Unsigned) integer division by 2

```
data Half  =  Even Natural      -- exact  division
           |  Odd Natural       -- remainder of 1

half  ::  Natural  -> Half
half  0        =  Even 0
half  1        =  Odd 0
half  (n+1)    =  case  half  n  of
                     Even  k  -> Odd k
                     Odd   k  -> Even (k+1)
```