Logic in Action

Implementing and Understanding Programs

Paul Downen
April 15, 2021
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)
propositions $\approx$ types
proofs $\approx$ programs
## Correspondence of Logic and Languages

<table>
<thead>
<tr>
<th>Logic</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural deduction</td>
<td>λ-calculus</td>
</tr>
<tr>
<td>Proposition</td>
<td>Type</td>
</tr>
<tr>
<td>Proof</td>
<td>Program</td>
</tr>
<tr>
<td>Logic</td>
<td>Language</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Natural deduction</td>
<td>λ-calculus</td>
</tr>
<tr>
<td>Proposition</td>
<td>Type</td>
</tr>
<tr>
<td>Proof</td>
<td>Program</td>
</tr>
<tr>
<td>Second-order quantification</td>
<td>Generics and modules</td>
</tr>
<tr>
<td>Classical logic</td>
<td>Control flow effects (call/cc)</td>
</tr>
</tbody>
</table>

...
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.

\[ \sqrt{2} \text{ is irrational, so consider } \sqrt{2}^{\sqrt{2}}. \]
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.
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.

$$(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}^2} = \sqrt{2}^2 = 2$$
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!
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!
Classic Classical Logic

Truth is Divine
...and sometimes out of reach to mortals
Is A True Or False?

Constructive Intuitionistic Logic

Truth is the Work of Mortals
The Monologue of the Sage

Truth is Disseminated through Proclamations

A is True!
THE DIALOGUE OF THE SAGE AND THE SKEPTIC

Truth is Discovered through Debate

SAGE

A is True!

A is False!

SKEPTIC
Constructive Classical Logic

Who Possesses the Burden of Proof?

AvB is True because A is True
AvB is False. It’s impossible for A or B to be True
Ex. P(x) is True because P(n) is True for this n
Ex. P(x) is False. P(n) is never True for any n

Positive
**Constructive Classical Logic**

**Who Possesses The Burden of Proof?**

- If $A \lor B$ is true, **because $A$ is true**.
- If $A \lor B$ is false, **it's impossible for $A$ or $B$ to be true**.
- If $\exists x. P(x)$ is true, **because $P(n)$ is true for this $n$**.
- If $\exists x. P(x)$ is false, **$P(n)$ is never true for any $n$**.

**Positive**

- $A \land B$ is true, **$A$ and $B$ can't both be false**.
- $A \land B$ is false, **because both $A$ and $B$ are false**.
- $\forall x. P(x)$ is true, **$P(n)$ is never false for any $n$**.
- $\forall x. P(x)$ is false, **because $P(n)$ is false for this $n$**.

**Negative**
Excluded Middle with a Positive Mindset

\[ \neg \neg A \text{ is True because } A \text{ is True. Here's proof.} \]

\[ A \lor \neg A \text{ is True because } A \text{ is True.} \]

\[ \text{POOF} \]

\[ \text{Actually, I meant } A \lor \neg A \text{ is True because } A \text{ is True... see?} \]

\[ \text{What happened?} \]
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

\[
\text{not true } = \text{ false}\\
\text{not false } = \text{ true}
\]

\[
\text{not}(A \text{ and } B) = (\text{not } A) \text{ or } (\text{not } B)\\
\text{not}(A \text{ or } B) = (\text{not } A) \text{ and } (\text{not } B)
\]
The sequent

\[ A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m \]

means

\[ A_1 \text{ and } A_2 \text{ and } \ldots \text{ and } A_n \quad \iff \quad B_1 \text{ or } B_2 \text{ or } \ldots \text{ or } B_m \]
The sequent

\[ A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m \]

means

\[ A_1 \text{ and } A_2 \text{ and } \ldots \text{ and } A_n \implies B_1 \text{ or } B_2 \text{ or } \ldots \text{ or } B_m \]

- \( \bullet \vdash A \) means \( A \) is true
- \( A \vdash \bullet \) means \( A \) is false
- \( \bullet \vdash \bullet \) means contradiction
**Computational Sequent Calculus**

- \( \bullet \vdash A \) 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
- \( \langle P \parallel C \rangle : (\bullet \vdash \bullet) \) is a **runnable command**

Think: **producer** = sage, **consumer** = skeptic, **command** = dialogue
Computational Sequent Calculus

\[ \bullet \vdash A \quad \text{means } A \text{ is true} \]
\[ A \vdash \bullet \quad \text{means } A \text{ is false} \]
\[ \bullet \vdash \bullet \quad \text{means contradiction} \]

\[ \blacklozenge \vdash P : A \quad \text{is a producer of } A \text{ values} \]
\[ C : A \vdash \bullet \quad \text{is a consumer of } A \text{ values} \]
\[ \langle P \| C \rangle : (\bullet \vdash \bullet) \quad \text{is a runnable command} \]

\[ \langle P \| C \rangle : (x_1 : A_1 \ldots x_n : A_n \vdash \alpha_1 : B_1 \ldots \alpha_m : B_m) \]

is an open command

with free inputs \( x_i \) and outputs \( \alpha_j \)

Think: producer = sage, consumer = skeptic, command = dialogue
DUALITIES OF COMPUTATION

\[ \langle P \parallel C \rangle \]

Answers

Questions

Producer | Consumer
---|---
Answers | Questions
Program | Context
Dualities of Computation

\[ \langle P \parallel C \rangle \]

Construction

Destruction

Producer | Consumer
---|---
Answers | Questions
Program | Context
Construction | Destruction
DUALITIES OF COMPUTATION

Data Flow

\[ \langle P \parallel C \rangle \]

Control Flow

<table>
<thead>
<tr>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Answers</td>
<td>Questions</td>
</tr>
<tr>
<td>Program</td>
<td>Context</td>
</tr>
<tr>
<td>Construction</td>
<td>Destruction</td>
</tr>
<tr>
<td>Data Flow</td>
<td>Control Flow</td>
</tr>
</tbody>
</table>
## Dualities of Computation

<table>
<thead>
<tr>
<th>Modules</th>
<th>Generics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle P \parallel C \rangle$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Answers</td>
<td>Questions</td>
</tr>
<tr>
<td>Program</td>
<td>Context</td>
</tr>
<tr>
<td>Construction</td>
<td>Destruction</td>
</tr>
<tr>
<td>Data Flow</td>
<td>Control Flow</td>
</tr>
<tr>
<td>Generics</td>
<td>Modules</td>
</tr>
</tbody>
</table>
Classical logic $\cong \lambda \mu = \lambda$-calculus + labels + jumps

Corresponds to Scheme’s call/cc control operator

$A \lor \neg A$ 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

---

1Downen, Ariola, ICFP ’14
Data vs Codata\textsuperscript{2}

\textbf{data} \ a \oplus \ b \ where

\begin{align*}
\text{Left} & : a \vdash a \oplus b \\
\text{Right} & : b \vdash a \oplus b
\end{align*}

\textbf{codata} \ a & b \ where

\begin{align*}
\text{First} & : a & b \vdash a \\
\text{Second} & : a & b \vdash b
\end{align*}

\textsuperscript{2}Downen \ & Ariola, ESOP ’14
**Data vs Codata**

\[
data a \oplus b \text{ where} \\
\quad \text{Left : } a \vdash a \oplus b \\
\quad \text{Right : } b \vdash a \oplus b \\
\]

\[
data a \otimes b \text{ where} \\
\quad \text{Pair : } a, b \vdash a \otimes b \\
\]

\[
codata a \& b \text{ where} \\
\quad \text{First : } a \& b \vdash a \\
\quad \text{Second : } a \& b \vdash b \\
\]

\[
codata a \triangleright b \text{ where} \\
\quad \text{Split : } a \triangleright b \vdash a, b \\
\]

\(^2\)Downen & Ariola, ESOP ’14
Data vs Codata

\textbf{data }a \oplus b \textbf{ where}

Left : \(a \vdash a \oplus b\)

Right : \(b \vdash a \oplus b\)

\textbf{data }a \otimes b \textbf{ where}

Pair : \(a, b \vdash a \otimes b\)

\textbf{data }a \ominus b \textbf{ where}

Yield : \(a \vdash a \ominus b, b\)

\textbf{codata }a \& b \textbf{ where}

First : \(a \& b \vdash a\)

Second : \(a \& b \vdash b\)

\textbf{codata }a \&\& b \textbf{ where}

Split : \(a \&\& b \vdash a, b\)

\textbf{codata }a \rightarrow b \textbf{ where}

Call : \(a, a \rightarrow b \vdash b\)

\(^2\)Downen & Ariola, ESOP ’14
Induction vs Coinduction

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

\[
\begin{align*}
\textbf{data} & \ \text{List} \ a \ \textbf{where} \\
\text{Nil} & : \ \bullet \vdash \text{List} \ a \\
\text{Cons} & : \ a, \text{List} \ a \vdash \text{List} \ a
\end{align*}
\]

\[
\begin{align*}
\text{length}(\text{Nil}) & = \text{Zero} \\
\text{length}(\text{Cons}(x, xs)) & = \text{Succ}(\text{length}(xs))
\end{align*}
\]

\[
\begin{align*}
\textbf{data} & \ \text{Nat} \ \textbf{where} \\
\text{Zero} & : \ \bullet \vdash \text{Nat} \\
\text{Succ} & : \ \text{Nat} \vdash \text{Nat}
\end{align*}
\]

\[3\text{Downen}, \text{Johnson-Freyd}, \text{Ariola, ICFP '15}\]
Induction vs Coinduction

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

```
data List a where
    Nil : • ⊢ List a
    Cons : a, List a ⊢ List a
```

```
data Nat where
    Zero : • ⊢ Nat
    Succ : Nat ⊢ Nat
```

```
length(Nil) = Zero
length(Cons(x, xs)) = Succ(length(xs))
```

Coinduction is a top-down, demand-driven approach

```
count(0) = 0, 1, 2, ...  
count(x) = x, count(x + 1)
```
Induction vs Coinduction

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

\[
\text{data List } a \text{ where }
\begin{align*}
\text{Nil} & : \quad \bullet \vdash \text{List } a \\
\text{Cons} & : \quad a, \text{List } a \vdash \text{List } a
\end{align*}
\]

\[
\text{length}(\text{Nil}) = \text{Zero} \\
\text{length}(\text{Cons}(x, xs)) = \text{Succ}(\text{length}(xs))
\]

Coinduction is a top-down, demand-driven approach

\[
\text{count}(0) = 0, 1, 2, \ldots \quad \text{count}(x) = x, \text{count}(x + 1)
\]

\[
\text{codata Stream } a \text{ where }
\begin{align*}
\text{Head} & : \quad \text{Stream } a \vdash a \\
\text{Tail} & : \quad \text{Stream } a \vdash \text{Stream } a
\end{align*}
\]

\[
\text{count}(x).\text{Head} = x \\
\text{count}(x).\text{Tail} = \text{count}(\text{Succ}(x))
\]

\[\text{Downen, Johnson-Freyd, Ariola, ICFP '15}\]
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
record Stream A : Set where
coinductive
field head : A
tail : Stream A

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); } 
}

\[
\text{count : Nat} \rightarrow \text{Stream Nat}
\]
\[
\text{head (count x) = x} \\
\text{tail (count x) = count (x + 1)}
\]

---

\text{4Downen & Ariola, Classical (Co)Recursion: Programming, 2021}
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)*

---

*Downen, Sullivan, Ariola, Peyton Jones, ESOP ’19*
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

Domain-specific notion of safety: set of commands \( \perp \)
Safe interaction is orthogonality

Individuals \( P \perp C \iff \langle P\|C \rangle \in \perp \)
Groups: \( A^+ \perp A^- \iff \forall P \in A^+, C \in A^-. P \perp C \)

Adjoint duality: \( A^\perp \) is biggest \( B \) s.t. \( A \perp B \) or \( B \perp A \)
Types are fixed points: \( A = (A^+, A^-) = (A^-\perp, A^+\perp) = A^\perp \)
\( \perp = \) 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

Feature Rich

Detail Rich

Source (Human) ← Reason

Target (Machine) → Execute

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

Feature Rich → Detail Rich

Source (Human) → Intermediate
  Desugar → Intermediate
  Generate Code → Intermediate
  Optimize → Intermediate

Intermediate → Target (Machine)
  Execute → Target (Machine)

Reason → Source (Human)
The Two-Way Street of Influence

Feature Rich

Expressive & Performant

Detail Rich

Source (Human)

Reason

Intermediate

Implementation

Target (Machine)
The Two-Way Street of Influence

Feature Rich

Expressive & Performant

Detail Rich

Source (Human)

Implementation

Reason

Intermediate

Implementation

Reason

Target (Machine)
Re-associating programs

\[ \lambda\text{-calculus} \]

\[ f \ 1 \ 2 \ 3 \ \ldots \]

\[ \langle f \parallel 1 \cdot 2 \cdot 3 \cdot \alpha \rangle \]

\[ \text{Sequent calculus} \]

\[ f \]

\[ 1 \]

\[ 2 \]

\[ 3 \]
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
if $x > 100$ :
    print "x is large"
else :
    print "x is small"
print "goodbye"
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
$f(1 + 1)$: is $1 + 1$ done before or after call?

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

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

Data Flow: Answers

$\langle P \parallel C \rangle$

Control Flow: Questions

Positive: CBV Data Types

<table>
<thead>
<tr>
<th>Answer</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Secondary</td>
</tr>
<tr>
<td>Action</td>
<td>Reaction</td>
</tr>
<tr>
<td>Concrete</td>
<td>Abstract</td>
</tr>
<tr>
<td>Finite</td>
<td>Infinite</td>
</tr>
</tbody>
</table>

e.g., lists, trees, structures,

Negative: CBN Codata Types

<table>
<thead>
<tr>
<th>Answer</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secondary</td>
<td>Primary</td>
</tr>
<tr>
<td>Reaction</td>
<td>Action</td>
</tr>
<tr>
<td>Abstract</td>
<td>Concrete</td>
</tr>
<tr>
<td>Infinite</td>
<td>Finite</td>
</tr>
</tbody>
</table>

e.g., functions, streams, processes,

Think: Positive vs Negative burden of proof
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: $a \rightarrow (b \rightarrow c)$ instead of $(a, b) \rightarrow c$
- Polymorphism: $\forall a. a \rightarrow a$; is $a = \text{Int}$ or $a = \text{Int} \rightarrow \text{Int}$?
- Pervasive Boxing: due to polymorphism or laziness
Kinds are Calling Conventions

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

---

\(^{10}\text{Downen, Ariola, Peyton Jones, Eisenberg, ICFP ’20} \)
Conclusion
1. Translate an idea from logic to computation

2. Use it to understand program behavior

3. Apply it to implement programs more efficiently
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
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

```cpp
private bool secret;
if secret { return true; } else { return false; }
```

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

Can differential privacy be decomposed into orthogonality?

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

$x$ DB$_1$ $y$ iff databases $x$, $y$ differ by 1 row

$\epsilon, \delta$-differentially private algorithm: DB$_1 \perp_{\epsilon,\delta}$

**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 $P \in A_{\parallel}$ 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
A call stack $x \cdot \alpha$ contains an:

- argument $x$
- return pointer $\alpha$

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

$$\langle length\|\text{Nil} \cdot \alpha \rangle = \langle \text{Zero}\|\alpha \rangle$$
$$\langle length\|\text{Cons} x \text{ xs} \cdot \alpha \rangle = \langle length\|\text{xs} \cdot \text{Succ} \circ \alpha \rangle$$

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

$$\langle count\|x \cdot \text{Head} \alpha \rangle = \langle x\|\alpha \rangle$$
$$\langle count\|x \cdot \text{Tail} \alpha \rangle = \langle count\|\text{Succ} x \cdot \alpha \rangle$$

---

11 Downen, Johnson-Freyd, Ariola, ICFP ’15
\[ \begin{align*} & \vdash P(\text{True}) \quad \vdash P(\text{False}) \\
& \quad \quad \quad \quad \quad \quad \quad x : \text{Bool} \vdash P(x) \end{align*} \]
INDUCTIVE REASONING

\[ \cdot \vdash P(\text{True}) \quad \cdot \vdash P(\text{False}) \]

\[ x : \text{Bool} \vdash P(x) \]

\[ \cdot \vdash P(0) \quad \cdot \vdash P(1) \quad \cdot \vdash P(2) \quad \ldots \]

\[ x : \text{Nat} \vdash P(x) \]
\[
\begin{align*}
\bullet \vdash P(\text{True}) & \quad \bullet \vdash P(\text{False}) \\
\hline
x : \text{Bool} \vdash P(x)
\end{align*}
\]

\[
\begin{align*}
\bullet \vdash P(0) & \quad \bullet \vdash P(1) & \quad \bullet \vdash P(2) & \quad \ldots \\
\hline
x : \text{Nat} \vdash P(x)
\end{align*}
\]

\[
\begin{align*}
\bullet \vdash P(0) & \quad y : \text{Nat}, \quad P(y) \vdash P(y + 1) \\
\hline
x : \text{Nat} \vdash P(x)
\end{align*}
\]
Coinductive Reasoning\textsuperscript{13}

\[
x : \text{Stream } A, \; P(x) \vdash P(x) \\
\]\[
x : \text{Stream } A \vdash P(x) \text{ \textit{warning!}}
\]

\footnote{Read $\alpha \div A$ as $\alpha : \neg A$, i.e., an assumption of not $A$, a continuation expecting $A$.}

\footnote{Downen \& Ariola, PPDP ‘20}
COINDUCTIVE REASONING\textsuperscript{13}

\[
\frac{x : \text{Stream } A, P(x) \vdash P(x)}{x : \text{Stream } A \vdash P(x)} \quad \text{warning!}
\]

\[
x : \text{Stream } A \vdash P\left(\left(x.\text{Head, } x.\text{Tail.}\text{Head,}\right)\right.
\]
\[
\left.\left(x.\text{Tail.}\text{Tail.}\text{Head,} \ldots \right)\right)
\]

\[
\frac{}{x : \text{Stream } A \vdash P(x)}
\]

\textsuperscript{12}Read $\alpha \vdash A$ as $\alpha : \neg A$, i.e., an assumption of not $A$, a continuation expecting $A$.

\textsuperscript{13}Downen & Ariola, PPDP '20
**Coinductive Reasoning**

\[
x : \text{Stream } A, P(x) \vdash P(x) \\
\text{warning!}
\]

\[
x : \text{Stream } A \vdash P\left(\left(\begin{array}{c}
x.\text{Head}, x.\text{Tail}\cdot\text{Head}, \\
x.\text{Tail}\cdot\text{Tail}\cdot\text{Head}, \ldots
\end{array}\right)\right)
\]

\[
x : \text{Stream } A \vdash P(x)
\]

\[
\alpha \div A \vdash P(\text{Head } \alpha) \quad \alpha \div A \vdash P(\text{Tail}[\text{Head } \alpha]) \quad \ldots \quad 12
\]

\[
\gamma \div \text{Stream } A \vdash P(\gamma)
\]

---

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

\[13\] Downen & Ariola, PPDP ’20
Coinductive Reasoning

\[
\frac{x : \text{Stream } A, P(x) \vdash P(x)}{x : \text{Stream } A \vdash P(x)} \quad \text{warning!}
\]

\[
\frac{x : \text{Stream } A \vdash P \left( \left( x.\text{Head}, x.\text{Tail}.\text{Head}, \right) \quad \left( x.\text{Tail}.\text{Tail}.\text{Head}, \ldots \right) \right)}{x : \text{Stream } A \vdash P(x)}
\]

\[
\frac{\alpha \div A \vdash P(\text{Head } \alpha) \quad \alpha \div A \vdash P(\text{Tail}[\text{Head } \alpha]) \quad \ldots}{\gamma \div \text{Stream } A \vdash P(\gamma)}
\]

\[
\frac{\alpha \div A \vdash P(\text{Head } \alpha) \quad \beta \div \text{Stream } A, P(\beta) \vdash P(\text{Tail } \beta)}{\gamma \div \text{Stream } A \vdash P(\gamma)}
\]

\[\text{Read } \alpha \div A \text{ as } \alpha : \neg A, \text{ i.e., an assumption of not } A, \text{ a continuation expecting } A.\]

\[\text{Downen & Ariola, PPDP '20}\]
Control Flow
Intuitionistic logic ⊂ Classical logic

Intuitionistic logic rejects the following classical laws:

Excluded Middle: $A \lor \neg A$ (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: $((A \implies B) \implies A) \implies A$
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 \(\neg\neg A \to A\) (where \(\neg A\) is a continuation)

Ambiguous choice: \(A + \neg A\) (either a value or continuation)
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
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")
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
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 versus $\phi$-nodes

$x < 0$

Yes

$y_1 = -x$

$z = \phi(y_1, y_2)$

...$

No

$y_2 = x$

Label $j(z) = \ldots$

In if $x < 0$

then jump $j(-x)$

everse jump $j(x)$
SEQUENT CALCULUS
Natural Deduction

\[
A \quad B \\
\hline
A \land B
\]

Sequent calculus

\[
\Gamma \vdash A, \Delta \\
\Gamma \vdash B, \Delta \\
\hline
\Gamma \vdash A \land B, \Delta
\]
**Re-orienting proofs**

**Natural Deduction**

\[
\begin{align*}
\frac{A \land B}{\Rightarrow A} \\
\frac{A \land B}{\Rightarrow B}
\end{align*}
\]

**Sequent calculus**

\[
\begin{align*}
\frac{\Gamma, A \vdash \Delta}{\Rightarrow \Gamma, A \land B \vdash \Delta} \\
\frac{\Gamma, B \vdash \Delta}{\Rightarrow \Gamma, A \land B \vdash \Delta}
\end{align*}
\]
A Syntax for Duality

\[
\begin{align*}
\Gamma & \vdash A, \Delta & \Gamma & \vdash B, \Delta \\
\hline
\Gamma & \vdash A \land B, \Delta
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash A, \Delta & \Gamma & \vdash B, \Delta \\
\hline
\Gamma & \vdash A \lor B, \Delta
\end{align*}
\]
Curry-Howard
All Natural Numbers are Even or Odd

What is even?

\[ n = 2k \]

What is odd?

\[ n = 2k + 1 \]

Proof by induction…

\[ 0 = 2(0): \text{even!} \]
\[ 1 = 2(0) + 1: \text{odd!} \]

\[ n + 1 \text{ by inductive hypothesis, } n \text{ is:} \]
\[ 2k \text{ then } n + 1 = 2k + 1: \text{odd!} \]
\[ 2k + 1 \text{ then } n + 1 = 2k + 1 + 1 = 2(k + 1): \text{even!} \]
**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)
```