The Strange Case of Dr. Admissibility and Mr. Derive

Dan Licata

Joint work with Noam Zeilberger and Robert Harper
Goal

A programming language that helps people:

- Define programming languages and logics
- Study their properties
- Use logics to reason about programs
Need

A programming language that makes it easy to:

- Represent syntax as data:
  - Programs of a programming language
  - Propositions of a logic

- Compute with these representations:
  - Write a compiler
  - Write a theorem prover
Derivability
In math

Polynomials over the reals:

\[ f(x) = x^2 + 2x + 1 \]
In math

Polynomials over the reals:

\[ f(x) = x^2 + 2x + 1 \]
In math

Polynomials over the reals:

\[ f(x) = x^2 + 2x + 1 \]

**Substitution:** plug in for the variable

\[ f(3) = 3^2 + 2*3 + 1 \]

\[ f(y+5) = (y+5)^2 + 2(y+5) + 1 \]
In programming

Functional abstraction:

\[
\text{fact}(x) = \begin{cases} 
1 & \text{if } (x = 0) \\
1 & \text{else} \\
x \cdot (\text{fact}(x-1)) & \text{if } (x > 0)
\end{cases}
\]
In programming

Functional abstraction:

\[
\text{fact}(x) = \begin{cases} 
1 & \text{if } (x = 0) \\
 x \times \text{fact}(x-1) & \text{else}
\end{cases}
\]

Compute by substitution:

\[
\text{fact}(5) = \begin{cases} 
1 & \text{if } (5 = 0) \\
5 \times \text{fact}(5-1) & \text{else}
\end{cases}
\]
In logic

(A implies B)
In logic

If \((A \implies B)\) and \(A\) then \(B\)

\[
\begin{array}{c}
A \\
\downarrow \\
B
\end{array}
\]

\((A \implies B)\)
In logic

If (A implies B) and A then B

(A implies B)
In logic

If (A implies B) and A then B
Derivability

Captures notion of **terms-with-holes**, which can be filled by substitution

- Represent syntax with variable binding
- Represent hypothetical judgements (typing for a programming language)
Admissibility
In math

Function from reals to reals specified by:

- set of ordered pairs
- every number appears exactly once on the LHS

\{
(0, 1),
(1, 4),
(\sqrt{2}, 3 + 2\sqrt{2}),
... 
\}
In programming

Specify a function by its behavior (e.g., memo table):

\[
\text{Factorial} = \{ (0, 1), \\
(1, 1), \\
(2, 2), \\
(3, 6), \\
(4, 24), \\
\ldots \}
\]
In logic

To prove $\forall x: \text{nat}. A(x)$,
prove $A(n)$ for each numeral $n$.

$A(0)$ and
$A(1)$ and
$A(2)$ and
$A(3)$ and
$A(4)$ ...
In logic

To prove $\forall x:\text{nat.}(\text{even}(x) \lor \text{odd}(x))$, 
prove $(\text{even}(n) \lor \text{odd}(n))$ for each numeral $n$.

\[
\text{even}(0) \lor \text{odd}(0) \text{ and } \\
\text{even}(1) \lor \text{odd}(1) \text{ and } \\
\text{even}(2) \lor \text{odd}(2) \text{ and } \\
\text{even}(3) \lor \text{odd}(3) \ldots
\]
Admissibility

Captures notion of an *arbitrary transformation*: only i/o-behavior matters

- Represent proofs with infinitely many cases
- Compute with logical systems (compiler transformation, translation between logics)
- Prove theorems *about* logics
Derivability

\[ f(x) = x^2 + 2x + 1 \]

Admissibility

\[ \{ (0, 1),
(1, 4),
(\sqrt{2}, 3 + 2\sqrt{2}),
\ldots \} \]
Proof assistants

How well do existing proof assistants support derivability and admissibility?
Contributions

Definitions using

- **Derivability**: easy in Twelf, hard in Coq
- **Admissibility**: easy in Coq, hard in Twelf
Contributions

Definitions using

- Derivability: easy in Twelf, hard in Coq
- Admissibility: easy in Coq, hard in Twelf

This work:

Supports definitions using both derivability and admissibility
Outline

- Representations using derivability
- Representations using admissibility
- Mixed representations
Outline

- Representations using derivability
- Representations using admissibility
- Mixed representations
Derivability

How do we represent variable binding?
Derivability

How do we represent variable binding?

What is variable binding?
Syntax with binding

\[ f(x) = x^2 + 2x + 1 \]

\[ \forall x : \text{nat. even}(x) \lor \text{odd}(x) \]
Structural properties

Properties of variables:

- Substitution
- Renaming
- Weakening
Structural properties

Properties of variables:

- Substitution (can plug in)
- Renaming
- Weakening
Structural properties

Properties of variables:

- Substitution (can plug in)
- Renaming
- Weakening
1. Factor the following: \( f(x) = x^2 + 2x + 1 \)

Answer:

\( f(y) = (y + 1)^2 \)
1. Factor the following: \( f(x) = x^2 + 2x + 1 \)

Answer:

\[ f(y) = (y + 1)^2 \]  

The answer is \( f(x) = (x + 1)^2 \)
1. Factor the following: \( f(x) = x^2 + 2x + 1 \)

Answer:

\[ f(y) = (y + 1)^2 \]

The answer is

\[ f(x) = (x + 1)^2 \]

(Dan finds a new math teacher...)
Renaming

Variables are **pronouns**: pointer structure matters, names don’t

\[ f(x) = x^2 + 2x + 1 \]

\[ f(y) = y^2 + 2y + 1 \]
Renaming

Variables are **pronouns**: pointer structure matters, names don’t

\[ f(x,y) = x^2 + 2y + 1 \]

\[ f(y,x) = y^2 + 2x + 1 \]
Renaming

Variables are **pronouns**: pointer structure matters, names don’t

\[ f(x, y) = x^2 + 2y + 1 \]

\[ f(x, y) = y^2 + 2x + 1 \]
Structural properties

Properties of variables:

- Substitution (can plug in)
- Renaming (names don’t matter)
- Weakening
Weakening

“A polynomial in **one variable** is called a **univariate polynomial**, a polynomial in **more than one variable** is called a **multivariate polynomial** … when working with **univariate polynomials** one does not exclude **constant polynomials**… although strictly speaking **constant polynomials do not contain any variables** at all.”

-- “Polynomial” in Wikipedia
Weakening

OK not to use a variable:

\[ f(x) = 4 \]

\[ f(x,y) = x^2 + 2x + 1 \]
Structural properties

Properties of variables:

- Substitution (can plug in)
- Renaming (names don’t matter)
- Weakening (unused variables ok)
Representing derivability

Q: How to implement data with binding?

Answer 1: variables = strings

Polynomial: \( f(x) = x^2 + 2 \times x + 1 \)

Representation: ("x", "x"^2 + 2 \times "x" + 1)

name of bound variable

reference to binding
Representing derivability

Q: How to implement data with binding?

Answer 1: variables = strings

(“x”, “x”^2 + 2 * “x” + 1)

Problems:

* Non-unique representations
* Do functions respect renaming?
* Have to implement substitution for each language
Representing derivability

Q: How to implement data with binding?

Other implement:
- de Bruijn indices/levels
- nominal logic [Pitts and Gabbay]
- cofinite quantification [Ayedemir et al.]
Representing derivability

Q: How to implement data with binding?

Other implement:

- de Bruijn indices/levels
- nominal logic [Pitts and Gabbay]
- cofinite quantification [Ayedemir et al.]

*These are *implementation*; we want the *interface*!*
Representing derivability

Key idea: represent binding with functions [HHP’93]

- Language provides a type of substitution functions with substitution, weakening, ...
- Implementation hidden from programmer
Representing derivability

\[ f(x) = x + 3 \]

represented by

\[ \lambda x. \text{plus } x \times 3 \]

Substitution functions written \( \lambda u. V \), where

\( V \) is a **value** (e.g. **constructors** and variables)
Representing derivability

Constructors generate syntax:

- **const** : rational \( \Rightarrow \) exp
  
  Example: 3 represented by \((\text{const } 3)\)

- **plus** : exp \( \Rightarrow \) exp \( \Rightarrow \) exp
  
  Example: 3 + 3 represented by
  
  \[
  \text{plus } (\text{const } 3) (\text{const } 3)
  \]
Representing derivability

Polynomials represented by substitution functions of type \((\text{exp} \Rightarrow \text{exp})\):

\[ f(x) = x + 3 \]

represented by

\[ \lambda x.\text{plus } x \ (\text{const} \ 3) \]

variables represented by variables!
Representing derivability

Polynomials represented by substitution functions of type \((\text{exp} \Rightarrow \text{exp})\):

- \(f(x) = x + 3\) represented by \(\lambda x. \text{plus} \ x \ (\text{const \ 3})\)
- \(f(5)\) represented by \((\lambda x. \text{plus} \ x \ (\text{const \ 3}))\) applied to \((\text{const \ 5})\)

\text{substitution} = \text{application}
Representing derivability

Polynomials represented by substitution functions of type (exp ⇒ exp):

\[ f(x) = x + 3 \]

represented by

\[ \lambda x. \text{plus } x \ (\text{const } 3) \]

substitution = application

\[ f(5) \]

represented by

\[ (\lambda x. \text{plus } x \ (\text{const } 3)) \]

applied to (\text{const } 5)

= \text{plus } (\text{const } 5) \ (\text{const } 3) \]
Adequacy

- Important that $\lambda u.V$ allows only substitution functions

- Cannot use admissibility to represent syntax: there are many more set-theoretic functions from expressions to expressions than polynomials

\[
\{(\text{const } n, 2 \times 2 \times 2 \times \ldots \times 2), \ldots\}
\]

$n$ times in total
Derivability

A theory of syntax with variable binding

1. Represent data with binding
   
   a) Define constructors *(plus)*
   
   b) Use derivability to represent binding *(⇒)*

2. Next: write recursive proofs/programs *(admissibility)*
Outline

- Representations using derivability
- Representations using admissibility
- Mixed representations
Admissibility

Admissibility \(\rightarrow\) provided by pattern-matching functional programs

zero : nat
succ : nat \(\Rightarrow\) nat

double : nat \(\rightarrow\) nat
double zero = zero
double (succ n) = succ (succ (double n))
Admissibility

Can pattern-match on substitution functions as well:

\( \text{eval} : (\text{exp} \Rightarrow \text{exp}) \rightarrow \text{rational} \rightarrow \text{rational} \)

\( \text{eval} (\lambda x. V) r = \text{evalexp} \ ( (\lambda x. V) \text{ applied to } (\text{const} \ r) ) \)

\( \text{evalexp} : \text{exp} \rightarrow \text{rational} \)

\( \text{evalexp} (\text{const} \ n) = n \)

\( \text{evalexp} (\text{plus} \ n1 \ n2) = \)

\( \text{add} (\text{evalexp} \ n1) (\text{evalexp} \ n2) \)

apply a substitution function
to perform substitution
Admissibility

Twelf:

Can use admissibility to reason about logical systems, but not to define logical systems

Coq and our framework:

Can use admissibility in definitions
Admissibility

shortestPath(a₁,a₂,n) iff path(a₁,a₂,n) and (\(\forall m.\) path(a₁,a₂,m) \(\rightarrow\) m \(\geq\) n)
Admissibility

\[\text{shortestPath}(a_1, a_2, n) \iff \text{path}(a_1, a_2, n) \text{ and } (\forall m. \text{path}(a_1, a_2, m) \rightarrow m \geq n)\]

can be written as

\[\text{path}(a_1, a_2, n) \quad (\forall m. \text{path}(a_1, a_2, m) \rightarrow m \geq n)\]

\[\text{shortestPath}(a_1, a_2, n)\]
Admissibility

\[
\text{path}(a,c,5) \quad (\forall m. \text{path}(a,c,m) \rightarrow m \geq 5)
\]

\[
\text{shortestPath}(a,c,5)
\]
Admissibility in rules

\[
\begin{align*}
\text{case } [ac]: & \quad 5 \geq 5 \\
\text{path}(a,c,5) & \quad \text{case } [abc]: \quad 6 \geq 5 \\
\text{shortestPath}(a,c,5) &
\end{align*}
\]
Admissibility in rules

(undirected, repeats ok)

Case [ac]: \(5 \geq 5\)
Case [abc]: \(6 \geq 5\)
Case [acbc]: \(11 \geq 5\)

\text{path}(a,c,5) \quad \ldots \text{infinitely many more} \ldots

\text{shortestPath}(a,c,5)
Contrast with derivability

- **Admissibility** \((\forall m. \text{path}(a,c,m) \rightarrow m \geq 5)\) allows proof by cases on the paths in this particular graph

- **Derivability** \((\forall m. \text{path}(a,c,m) \Rightarrow m \geq 5)\) requires proof for a new path of new length \(m\): **not true!**
Outline

- Representations using derivability
- Representations using admissibility
- Mixed representations
Arithmetic expressions

Arithmetic expressions with let-binding:

\[
\text{let } x \text{ be } (\text{const 4}) \text{ in } (\text{plus } x \ x)
\]

e ::= \text{const } n

| let x be e1 in e2
| \text{plus } e1 e2
| \text{times } e1 e2
| \text{sub } e1 e2
| \text{mod } e1 e2
| \text{div } e1 e2
Arithmetic expressions

Arithmetic expressions with let-binding:

\[
\text{let } x \text{ be (const 4) in (plus } x \ x)\\
\]

\( e ::= \text{ const } n \)
\[
| \text{ let } x \text{ be } e_1 \text{ in } e_2 \\
| \text{ plus } e_1 \ e_2 \\
| \text{ times } e_1 \ e_2 \\
| \text{ sub } e_1 \ e_2 \\
| \text{ mod } e_1 \ e_2 \\
| \text{ div } e_1 \ e_2 \\
\]

Suppose we want to treat binops uniformly…
Arithmetic expressions

Arithmetic expressions with let-binding

e ::= const n
  | let x be e1 in e2
  | binop e1 φ e2

where φ : (nat → nat → nat) is the code for the binop.
Arithmetic expressions

\[
\text{const} : \text{nat} \Rightarrow \text{exp} \\
\text{let} : \text{exp} \Rightarrow (\text{exp} \Rightarrow \text{exp}) \Rightarrow \text{exp} \\
\text{binop} : \text{exp} \Rightarrow (\text{nat} \rightarrow \text{nat} \rightarrow \text{nat}) \Rightarrow \text{exp} \Rightarrow \text{exp}
\]

let \(x\) be \((\text{const} 4)\) in \((x + x)\)

represented by

let \((\text{const} 4)\) \((\lambda x. \text{binop} x \text{add} x)\)

where \text{add}:(\text{nat} \rightarrow \text{nat} \rightarrow \text{nat})\ is the code for addition
Arithmetic expressions

const : nat ⇒ exp

let : exp ⇒ (exp ⇒ exp) ⇒ exp

binop : exp ⇒ (nat ⇒ nat ⇒ nat) ⇒ exp ⇒ exp

let x be (const 4) in (x + x)

let (const 4) (λx. binop x add x)

where add:(nat ⇒ nat ⇒ nat) is the code for addition
Arithmetic expressions

\[
\begin{align*}
const & : \text{nat} \Rightarrow \text{exp} \\
\text{let} & : \text{exp} \Rightarrow (\text{exp} \Rightarrow \text{exp}) \Rightarrow \text{exp} \\
\text{binop} & : \text{exp} \Rightarrow (\text{nat} \rightarrow \text{nat} \rightarrow \text{nat}) \Rightarrow \text{exp} \Rightarrow \text{exp}
\end{align*}
\]

let \( x \) be (const 4) in \((x + x)\)

let (const 4) (\( \lambda x. \text{binop} \) x add x) 

where add:(nat \rightarrow nat \rightarrow nat) is the code for addition
Arithmetic expressions

const : nat ⇒ exp
let : exp ⇒ (exp ⇒ exp) ⇒ exp
binop : exp ⇒ (nat → nat → nat) ⇒ exp ⇒ exp

let x be (const 4) in (x + x)

let (const 4) (λx.binop x add x)

where add:(nat → nat → nat) is the code for addition
Arithmetic expressions

const : nat ⇒ exp

let : exp ⇒ (exp ⇒ exp) ⇒ exp

binop : exp ⇒ (nat → nat → nat) ⇒ exp ⇒ exp

let x be (const 4) in (x + x)

let (const 4) (Λx. binop x add x)

where add:(nat → nat → nat) is the code for addition
Structural properties
Structural properties

Weakening for mixed representations:
Structural properties

Weakening for mixed representations:

1. Go from (nat \Rightarrow nat) to (nat \Rightarrow nat \Rightarrow nat)
Structural properties

Weakening for mixed representations:

1. Go from \((\text{nat} \Rightarrow \text{nat})\) to \((\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat})\)

\[ \lambda x. (\lambda y. V) \quad [x \text{ doesn’t occur in } V] \]
Structural properties

Weakening for mixed representations:

1. Go from (nat \rightarrow nat) to (nat \rightarrow nat \rightarrow nat)
   \[ \lambda x. (\lambda y. V) \]  [x doesn’t occur in V]

2. Go from (nat \rightarrow nat) to (nat \rightarrow nat \rightarrow nat)
Structural properties

Weakening for mixed representations:

1. Go from \((\text{nat} \rightarrow \text{nat})\) to \((\text{nat} \rightarrow \text{nat} \rightarrow \text{nat})\)
   \[\lambda x. (\lambda y. V)\] \([x \text{ doesn’t occur in } V]\]

2. Go from \((\text{nat} \rightarrow \text{nat})\) to \((\text{nat} \rightarrow \text{nat} \rightarrow \text{nat})\)
   \[\text{drop } x y = f y\] \([x \text{ doesn’t occur in } f]\)
Structural properties
Structural properties

Weakening for mixed representations:
Structural properties

Weakening for mixed representations:

3. Go from $\text{nat} \Rightarrow \text{nat}$ to $\text{nat} \rightarrow \text{nat} \Rightarrow \text{nat}$
Structural properties

Weakening for mixed representations:

3. Go from \((\text{nat} \Rightarrow \text{nat})\) to \((\text{nat} \rightarrow \text{nat} \Rightarrow \text{nat})\)

\[ h \ x = \lambda y. V \quad \text{[x doesn’t occur in V]} \]
Structural properties

Weakening for mixed representations:

3. Go from (nat ⇒ nat) to (nat → nat ⇒ nat)

\[ h \, x = \lambda y. V \quad [x \text{ doesn’t occur in } V] \]

4. Go from (nat → nat) to (nat ⇒ (nat → nat))
Structural properties

A function of type \((\text{nat} \rightarrow \text{nat})\):

\[
\begin{align*}
\text{double}\ zero &= \text{zero} \\
\text{double}\ (\text{succ}\ n) &= \text{succ}\ (\text{succ}\ (\text{double}\ n))
\end{align*}
\]
Structural properties

A (nat \Rightarrow (nat \rightarrow \text{nat})): 

\[ \lambda x. \text{double } zero = zero \]
\[ \text{double } (\text{succ } n) = \text{succ } (\text{succ } (\text{double } n)) \]
\[ \text{double } x = ??? \]

Problem: \text{double} doesn’t have a case for \text{x}, so mixed representations not necessarily structural!
Our solution:

- $\lambda x. V$ used by pattern-matching; application (=substitution) is not built-in
- Nothing forces $\Rightarrow$ to be structural
- Weakening/substitution implemented generically for a wide class of rule systems, using some technical conditions
Structural properties

\[
\begin{align*}
\text{const} & : \text{nat} \Rightarrow \text{exp} \\
\text{let} & : \text{exp} \Rightarrow (\text{exp} \Rightarrow \text{exp}) \Rightarrow \text{exp} \\
\text{binop} & : \text{exp} \Rightarrow (\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}) \Rightarrow \text{exp} \Rightarrow \text{exp}
\end{align*}
\]

- Can’t weaken \text{exp} with \text{nat}: could need new case for \text{→} in a \text{binop}
- Can weaken \text{exp} with \text{exp}: doesn’t appear to left of \text{→}
Conclusion
Summary

- **Derivability (variable binding)** provided by substitution functions (⇒)

- **Admissibility (arbitrary reasoning)** provided by pattern-matching functional programs (→)

- Our framework makes it easy to use both
Future work

- This talk: syntax of logical systems
  Need **dependent types** to represent judgements

- More examples mixing \( \Rightarrow \) \( \rightarrow \)

- Thesis work: a language in which you can define logics and use them to reason about programs
Strange Case of Dr. Admissibility and Mr. Derive