Dependently Typed Programming with Domain-Specific Logics

Daniel R. Licata

Thesis Committee:

Robert Harper, Chair

Karl Crary

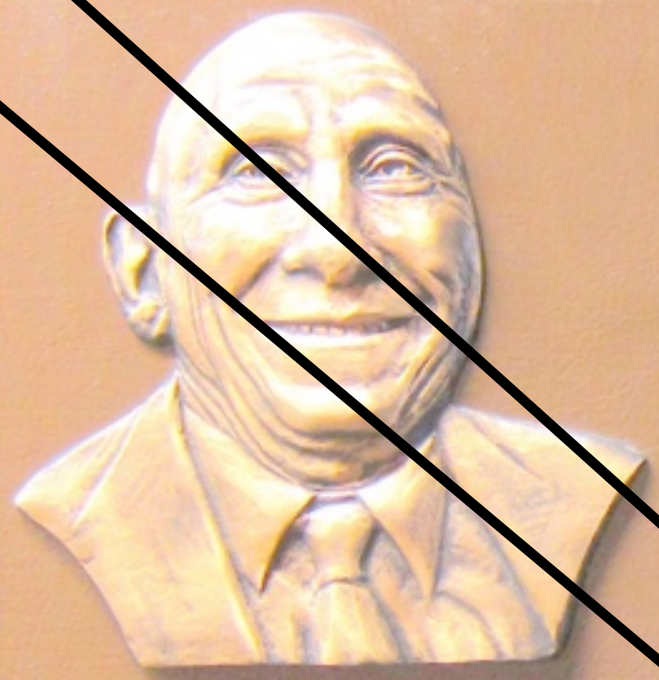
Frank Pfenning

Greg Morrisett, Harvard University

Supported by NSF CCF-0702381|CNS-0716469 and the Pradeep Sindhu Fellowship


Prolog
INSERT CRAZY OLD MAN
PROPHECY

- IT IS EASIER TO WRITE AN INCORRECT PROGRAM THAN UNDERSTAND A CORRECT ONE.



ALAN J. PERLIS
(1922-1990)

A FOUNDER OF THE
COMPUTER SCIENCE DEPARTMENT
CARNEGIE MELLON UNIVERSITY

FIRST DEPARTMENT HEAD
1965-1971

THE PERLIS EPIGRAMS

- ONE MAN'S CONSTANT IS ANOTHER MAN'S VARIABLE.
- IF A LISTENER NODS HIS HEAD WHEN YOU'RE EXPLAINING YOUR PROGRAM, WAKE HIM UP.
- DON'T HAVE GOOD IDEAS IF YOU AREN'T WILLING TO BE RESPONSIBLE FOR THEM.
- IN SOFTWARE SYSTEMS IT IS OFTEN THE EARLY BIRD THAT MAKES THE WORM.
- EVERY PROGRAM HAS (AT LEAST) TWO PURPOSES: THE ONE FOR WHICH IT WAS WRITTEN AND ANOTHER FOR WHICH IT WASN'T.

- IT IS EASIER TO WRITE AN INCORRECT PROGRAM THAN UNDERSTAND A CORRECT ONE.

- IF YOU HAVE A PROCEDURE WITH 10 PARAMETERS, YOU PROBABLY MISSED SOME.
- IN A 5 YEAR PERIOD WE GET ONE SUPERB PROGRAMMING LANGUAGE - ONLY WE CAN'T CONTROL WHEN THE 5 YEAR PERIOD WILL BEGIN

- IT IS EASIER TO WRITE AN INCORRECT PROGRAM THAN UNDERSTAND A CORRECT ONE.

- IT IS EASIER TO WRITE AN INCORRECT PROGRAM THAN UNDERSTAND A CORRECT ONE.

Goal:

Make it
harder to write incorrect programs
and
easier to understand correct ones

Method:

Make
type system & specification logic design
part of the programming process

Domain-specific logics

Method:

Make
type system & specification logic design
part of the programming process

Domain-specific logics

Method:

Make
type system & specification logic design
part of the programming process
using
dependent types

Examples

- Ynot: verifying imperative programs with separation logic [Morrisett et al.]
- PCML5, Aura, Fine: verifying security properties with authorization logic
  [Chapter 3; Morgenstern & Licata, ICFP'10]
- Reed&Pierce's type system for Differential Privacy
  [Chapter 4]

Examples

- Ynot: verifying imperative programs with separation logic [Morrisett et al.]
- **PCML5, Aura, Fine: verifying security properties with authorization logic [Chapter 3; Morgenstern & Licata, ICFP'10]**
- Reed&Pierce's type system for Differential Privacy [Chapter 4]

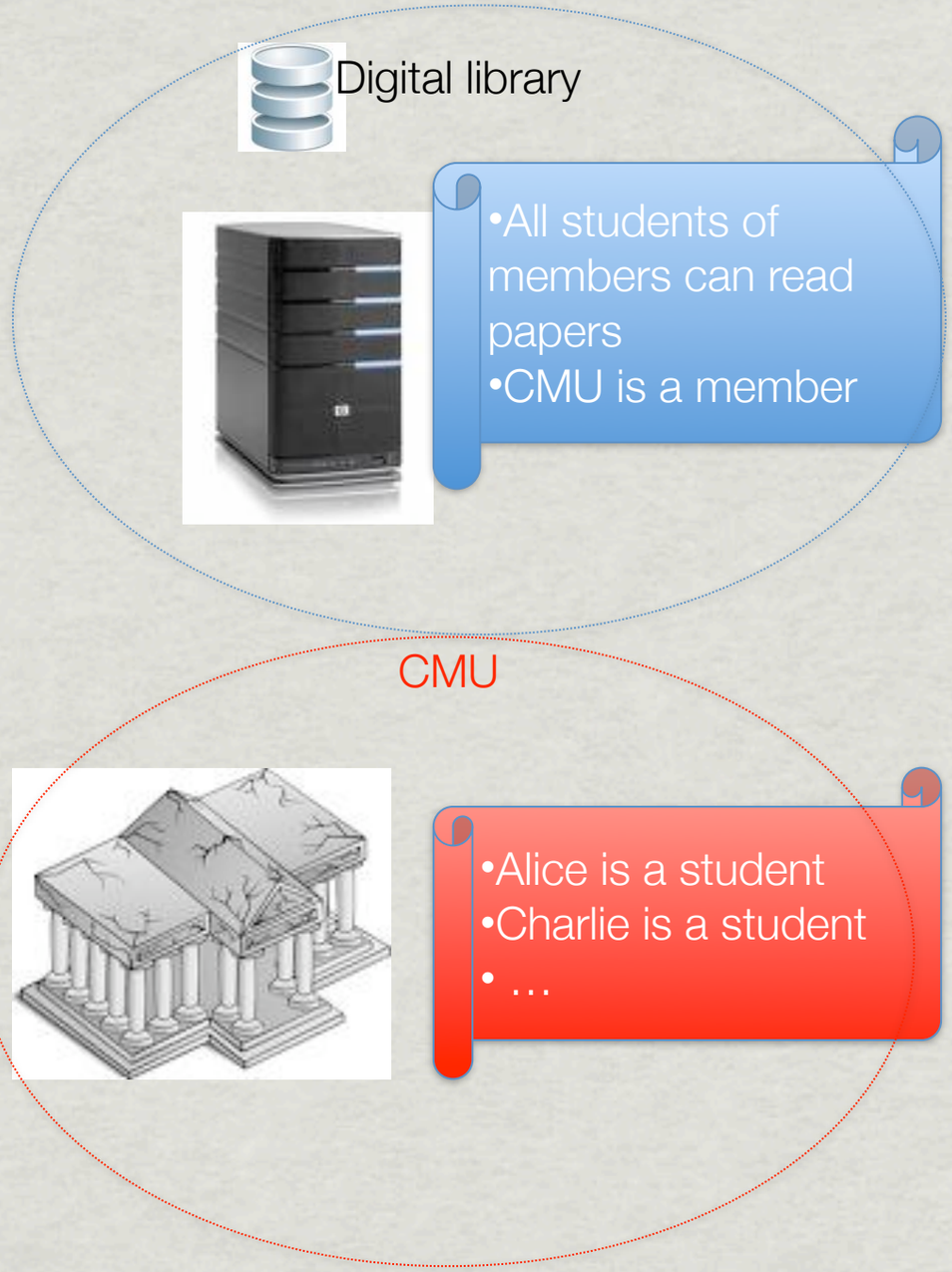
Security-Typed Programming

ACM **says** ∀ s:principal,
∀ i:principal,
∀ p:paper,
(member(i) ∧ i **says** student(s))
⊃ MayRead(s, p)

...

CMU **says** student(Alice)

...



(slide by Kumar Avijit)

Dependent Types! [Agda]

read : prin→ file
→ contents

Dependent Types! [Agda]

read : prin→ file → proof
→ contents

Dependent Types! [Agda]

read : prin → file → proof
→ contents

read : (k : prin) (f : file) (p : proof(mayread(k,f))
→ contents

Dependent Types! [Agda]

```
read : prin→ file → proof
       → contents
```

```
read : (k : prin) (f : file) (p : proof(mayread(k,f))
       → contents
```

- typing system ensures p is a well-formed proof
- and that proofs of appropriate theorems are used

Embed in Agda

- Indexed inductive definition to represent proofs
- Theorem prover to discharge proof obligations, run at compile-time and run-time
- Indexed monad to manage stateful+dynamic policies

Representing Logic

Sequent as indexed inductive definition:

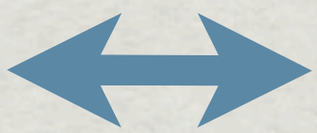
$\Gamma \vdash A$  data _⊢_ : Ctx → Propo → Type

Representing Logic

Sequent as indexed inductive definition:

$\Gamma \vdash A$  data _⊢_ : Ctx → Propo → Type

Classifying *only* well-formed derivations:

$\mathcal{D}$
$\Gamma \vdash A$  $\mathcal{D} : \Gamma \vdash A$

Representing Logic

Sequent as indexed inductive definition:

$\Gamma \vdash A$ → `data _⊢_ : Ctx → Propo → Type`

Classifying *only* well-formed derivations:

$\mathcal{D}$ / $\Gamma \vdash A$ ↔ $\mathcal{D} : \Gamma \vdash A$

Inference rules as datatype constructors:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}$$

```
⊃R : ∀ {Γ A B}
   → (A :: Γ) ⊢ B
   → Γ ⊢ (A ⊃ B)
```

Representing Logic

Sequent as indexed inductive definition:

$\Gamma \vdash A$ → `data _⊢_ : Ctx → Propo → Type`

Classifying *only* well-formed derivations:

$$\mathcal{D} \quad \Gamma \vdash A \qquad \leftrightarrow \qquad \mathcal{D} : \Gamma \vdash A$$

Inference rules as datatype constructors:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}$$

→

```
⊃R : ∀ {Γ A B}
   → (A :: Γ) ⊢ B
   → Γ ⊢ (A ⊃ B)
```

dependent de Bruijn indices

Theorem Prover

Implemented a *certified* theorem prover:

$$\text{prove} : (\Gamma : \text{Ctx})\ (A : \text{Propo}) \to \text{Maybe}\ (\Gamma \vdash A)$$

Theorem Prover

Implemented a *certified* theorem prover:

prove : (Γ : Ctx) (A : Propo) → Maybe (Γ ⊢ A)

Important that Propos are *inductive*!

data Propo where
  says : Principal → Propo → Propo
  ...

Examples

- Ynot: verifying imperative programs with separation logic [Morrisett et al.]
- PCML5, Aura, Fine: Security-Typed Programming [Chapter 3; Morgenstern & Licata, ICFP'10]
- **Reed&Pierce's type system for Differential Privacy [Chapter 4]**

Differential Privacy

- Ask questions about a database

- Any answer almost exactly as likely if any one person is omitted from the database

Reed&Pierce

- Type system based on affine logic tracks the sensitivity of a function

- Ensure differential privacy by adding noise proportional to sensitivity

Reed&Pierce

- Type system based on affine logic tracks the sensitivity of a function

$$x_1{:}A_1[s_1],\ x_2{:}A_2[s_2],\ \ldots\ x_n{:}A_n[s_n] \vdash C$$

- Ensure differential privacy by adding noise proportional to sensitivity

Reed&Pierce

- Type system based on affine logic tracks the sensitivity of a function

$$x_1{:}A_1[s_1],\ x_2{:}A_2[s_2],\ \ldots\ x_n{:}A_n[s_n] \vdash C$$

can use a variable if $s \geq 1.0$

- Ensure differential privacy by adding noise proportional to sensitivity

Semantics

Each type A denotes a metric space:

- Set of values |A|, equipped with notion of distance

- $A \vdash B$ means

  $f : |A| \to |B|$

  such that

  if $dist_A(x,y) \leq r$ then $dist_B(f\ x, f\ y) \leq r$

Primitives

Affine logic rules are sound
but lots of primitives are justified semantically:

cmpswp : real -o real -o real ⊗ real
rsplit : real -o real ⊗ real

Primitives

Affine logic rules are sound
but lots of primitives are justified semantically:

cmpswp : real -o real -o real ⊗ real
rsplit : real -o real ⊗ real

need to be baked into the language

Extensible Diff. Priv. [Chap 4]

- Implement the *semantics* using dependent types ($\Pi x,y,r.\ dist_A(x,y) \le r \rightarrow dist_B(f\ x, f\ y) \le r$)
- Primitives implemented and proved sound in the semantics
- Build affine type system on top

Part 1:

It is possible to define, study, automate, and use domain-specific logics within a dependently typed programming language

But how can we make it easier?

Outline

1. New examples of programming with domain-specific logics [Chapters 3 and 4]

2. An investigation into mixing derivability and admissibility [Chapter 5, 6, 7, 8]

3. [REDACTED]

Outline

1. New examples of programming with domain-specific logics [Chapters 3 and 4]

2. **An investigation into mixing derivability and admissibility [Chapter 5, 6, 7, 8]**

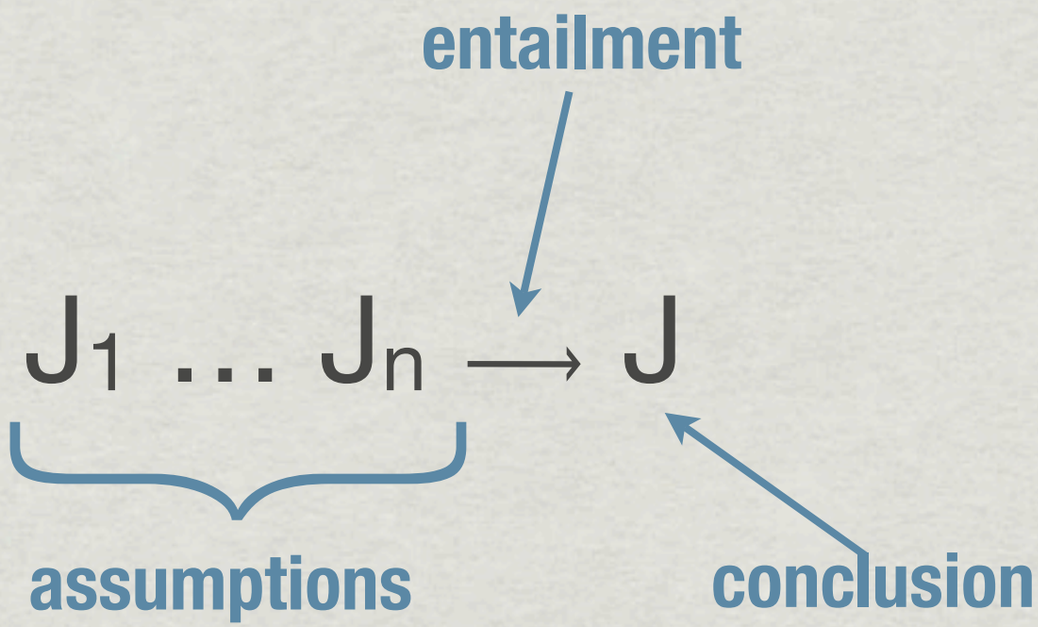
3. [REDACTED]

A Tale of Two Consequence Relations

A Tale of Two Consequence Relations

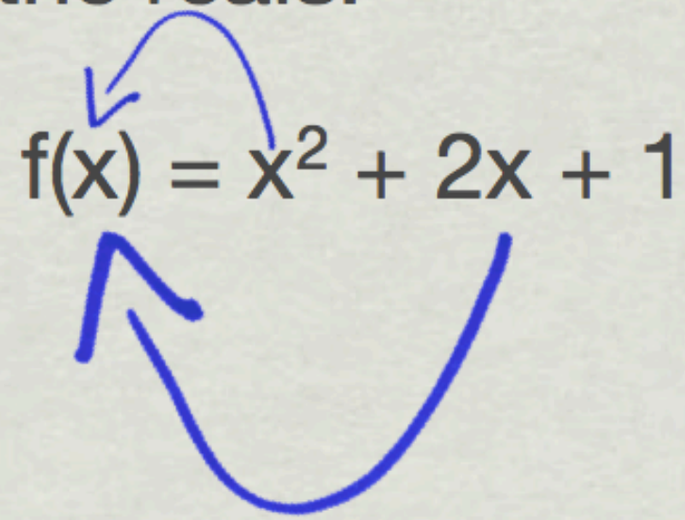
$$J_1 \ldots J_n \longrightarrow J$$

A Tale of Two Consequence Relations



Derivability (⊢)

Polynomials over the reals:



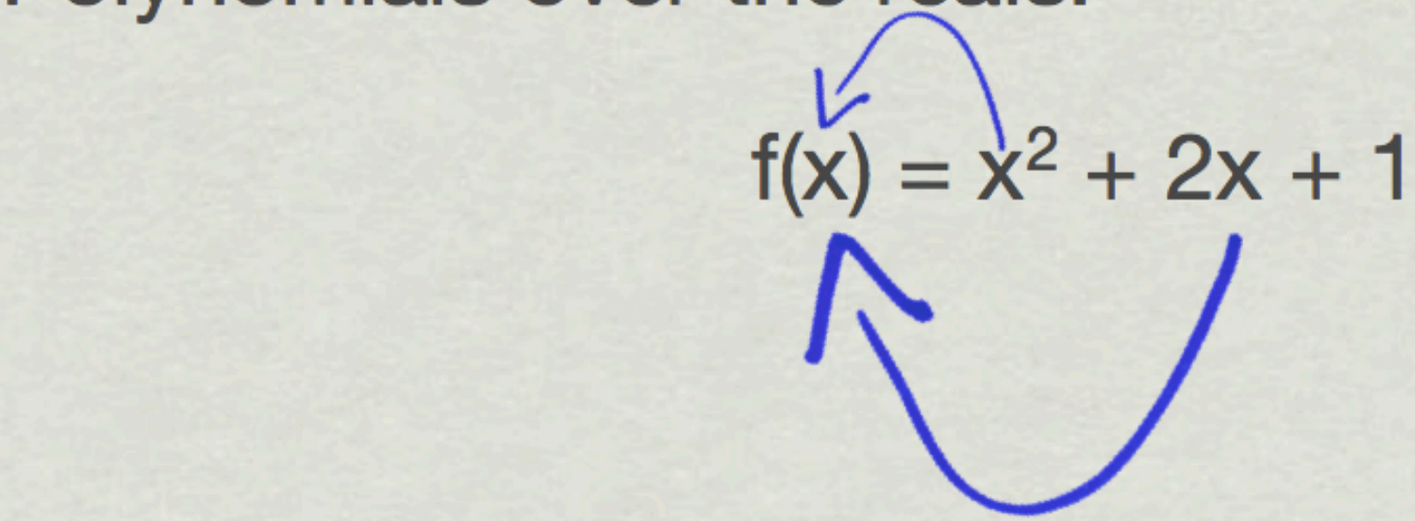
Substitution: plug in for the variable

- $f(3) = 3^2 + 2*3 + 1$
- $f(y+5) = (y+5)^2 + 2(y+5) + 1$

Derivability (⊢)

real ⊢ real

Polynomials over the reals:

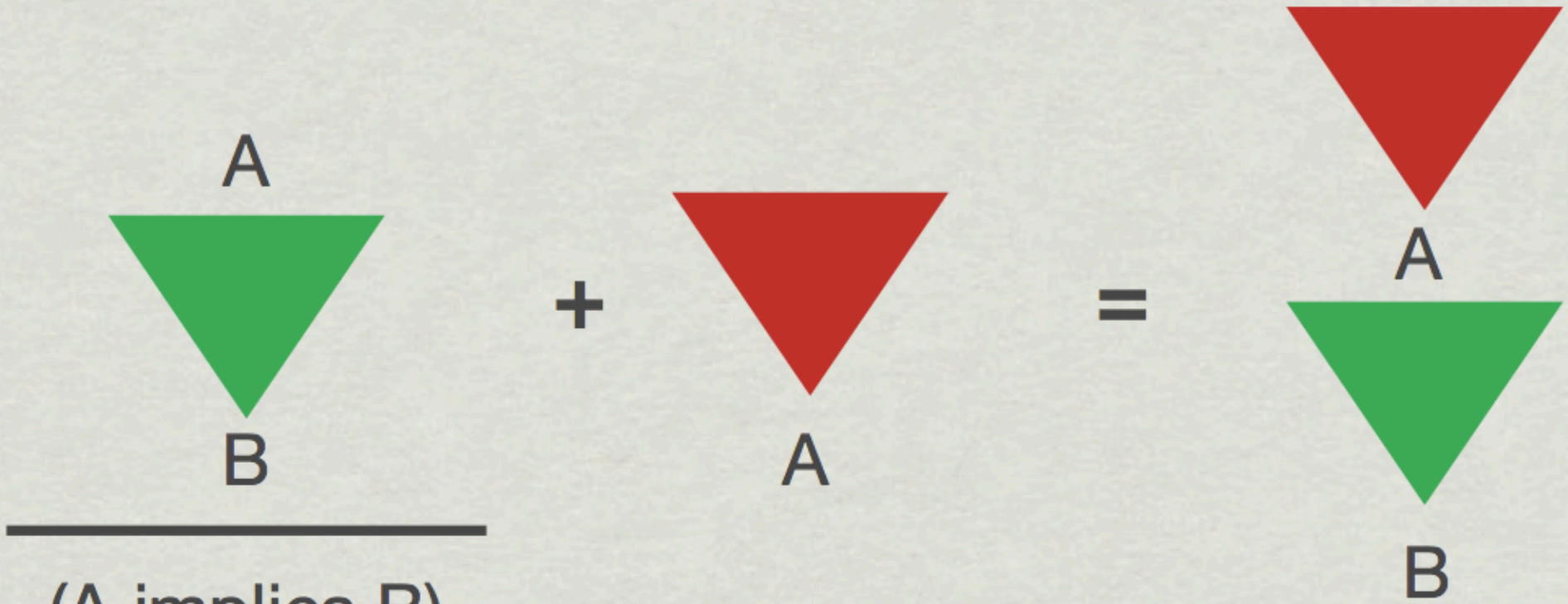


Substitution: plug in for the variable

- $f(3) = 3^2 + 2*3 + 1$
- $f(y+5) = (y+5)^2 + 2(y+5) + 1$

Derivability (⊢)

If (A implies B) and A then B



(A implies B)

Derivability (⊢)

If (A implies B) and A then B

Derivability (⊢)

$$\supset R : \forall \{\Gamma\ A\ B\} \to (A :: \Gamma) \vdash B \to \Gamma \vdash (A \supset B)$$

$$x_1{:}A_1[s_1],\ \ldots\ x_n{:}A_n[s_n] \vdash C$$

Derivability $J_1 \vdash J_2$:

syntactic variables given meaning by subst.

Admissibility (⊨)

Function from reals to reals specified by:

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

$$\{ (0 , 1), \\ (1 , 4), \\ (\sqrt{2}, 3 + 2\sqrt{2}), \\ \dots \}$$

Admissibility (⊨)

prove : (Γ : Ctx) (A : Propo)
→ Maybe (Γ ⊢ A)

f : |A| → |B| such that
$dist_A(x,y) \leq r \rightarrow dist_B(f\ x, f\ y) \leq r$

Admissibility J1 ⊨ J2:

inductive proofs and functional programs

Structural Properties

$$\frac{}{\Gamma, u : J, \Gamma' \vdash J}\ u \qquad \frac{\Gamma, \Gamma' \vdash J_1 \quad \Gamma, u : J_1, \Gamma' \vdash J_2}{\Gamma, \Gamma' \vdash J_2}\ \text{subst}$$

$$\frac{\Gamma, \Gamma' \vdash J'}{\Gamma, u : J, \Gamma' \vdash J'}\ \text{weakening} \qquad \frac{\Gamma, u_2 : J_2, u_1 : J_1, \Gamma' \vdash J'}{\Gamma, u_1 : J_1, u_2 : J_2, \Gamma' \vdash J'}\ \text{exchange}$$

$$\frac{\Gamma, u_1 : J, u_2 : J, \Gamma' \vdash J'}{\Gamma, u_1 : J, \Gamma' \vdash J'}\ \text{contraction}$$

In Existing Frameworks

MLTT: admissibility as functions
have to code up derivability yourself

LF: derivability as functions
admissibility in separate layer (Twelf, Delphin)

In Existing Frameworks

MLTT: admissibility as functions
have to code up derivability yourself

LF: derivability as functions
admissibility in separate layer (Twelf, Delphin)

inherently unequal!

Admissibility premises

Negated premises:

$$\frac{l_1 = l_2 \vDash \mathsf{false} \quad \mathsf{lookup}(M, l_1) = v}{\mathsf{lookup}(M[l_2 \mapsto \_], l_1) = v}$$

ω-rule:

$$\frac{t : \mathsf{nat} \quad n : \mathsf{nat} \vDash P(n)}{P(t)\ \ \mathsf{true}}$$

Admissibility premises

Negated premises:

$$\frac{l_1 = l_2 \vDash \mathsf{false} \quad \mathsf{lookup}(M, l_1) = v}{\mathsf{lookup}(M[l_2 \mapsto \_], l_1) = v}$$

ω-rule:

$$\frac{t : \mathsf{nat} \quad n : \mathsf{nat} \vDash P(n)}{P(t)\ \ \mathsf{true}}$$

concise representations of pattern matching [Zeilberger]

Problem

can no longer weaken with equality deriv. assumptions

$$\frac{l_1 = l_2 \vDash \mathsf{false} \quad \mathsf{lookup}(M, l_1) = v}{\mathsf{lookup}(M[l_2 \mapsto \_], l_1) = v}$$

J1 $\vdash$ (J2 $\vDash$ J3)

doesn't necessarily follow from

(J2 $\vDash$ J3)

Part II

It is possible to implement, within a dependently typed programming language, a simply typed logical framework that allows derivability and admissibility to be mixed in novel and interesting ways.

**[Licata and Harper, ICFP'09;
Licata, Zeilberger, Harper; LICS'08]**

Embedded Logical Framework

- Define a datatype representing framework types, including derivability ($\Psi \vdash A$) and admissibility functions ($A \models B$)
- Define framework programs by interpretation into Agda
- Automatically equip framework types with the structural properties using generic programming
- Do fun examples using mixing (NBE)

Structural Properties

- Weakening: A $\models$ (D $\vdash$ A) if […graph algorithm…]
- Substitution: (D $\Rightarrow$ A) $\supset$ (D $\supset$ A) if …
- Exchange: ($D_1 \Rightarrow D_2 \Rightarrow A$) $\supset$ ($D_2 \Rightarrow D_1 \Rightarrow A$) if ..
- Contraction: (D $\Rightarrow$ D $\Rightarrow$ A) $\supset$ (D $\Rightarrow$ A) if …
- Strengthening: (D $\Rightarrow$ A) $\supset$ A if …

Questions

- When do structural properties exist?
- Dependent types?
  - subst. into derivation yields subst. into judgement

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau, \Gamma' \vdash J}{\Gamma, \Gamma'[e/x] \vdash J[e/x]} \text{ subst}$$

  - requires *composition*

$$A[t/x][s/y] = A[s/y][t[s/y]/x]$$

Outline

1. New examples of programming with domain-specific logics [Chapters 3 and 4]

2. An investigation into mixing derivability and admissibility [Chapter 5, 6, 7]

3. [REDACTED]

Directed Type Theory

Directed Type Theory

[logo by RJS]

Directed Types

Each type has notion of transformation on elements:

$$M_1 \lesssim_A M_2$$

Every type family x:A ⊢ B type respects trans.:

$$\frac{\Gamma, x{:}A \ \vdash B \ \mathsf{type} \quad \Gamma \vdash \alpha : M_1 \lesssim_A M_2 \quad \Gamma \vdash M : B[M_1/x]}{\Gamma \vdash \mathsf{map}_{x:A.B}\ \alpha\ M : B[M_2/x]}$$

Directed Types

Each type has notion of transformation on elements:

$$M_1 \lesssim_A M_2$$

judgement, not type

Every type family x:A ⊢ B type respects trans.:

$$\frac{\Gamma, x{:}A \vdash B \text{ type} \quad \Gamma \vdash \alpha : M_1 \lesssim_A M_2 \quad \Gamma \vdash M : B[M_1/x]}{\Gamma \vdash \mathsf{map}_{x:A.B}\ \alpha\ M : B[M_2/x]}$$

map for Pairs

Action of map given by each type constructor:

$$map_{x:A.B \times C}\,(\alpha : M_1 \lesssim_A M_2)\,(e\,,\,e') =$$
$$(map_{x:A.B}\,\alpha\,e\,,\,map_{x:A.C}\,\alpha\,e')$$

map for Pairs

Action of map given by each type constructor:

$B[M_1] \times C[M_1]$

$$map_{x:A.B \times C}\ (\alpha : M_1 \lesssim_A M_2)\ (e\ ,\ e') =$$
$$(map_{x:A.B}\ \alpha\ e\ ,\ map_{x:A.C}\ \alpha\ e')$$

map for Pairs

Action of map given by each type constructor:

$B[M_1] \times C[M_1]$

$$map_{x:A.B \times C}\ (\alpha : M_1 \lesssim_A M_2)\ (e\ ,\ e') =$$
$$(map_{x:A.B}\ \alpha\ e\ ,\ map_{x:A.C}\ \alpha\ e')$$

Goal: $B[M_2] \times C[M_2]$

map for Pairs

Action of map given by each type constructor:

$B[M_1] \times C[M_1]$

$$map_{x:A.B \times C}\ (\alpha : M_1 \lesssim_A M_2)\ (e\ ,\ e') =$$

$$(map_{x:A.B}\ \alpha\ e\ ,\ map_{x:A.C}\ \alpha\ e')$$

$B[M_2]$ $C[M_2]$

Goal: $B[M_2] \times C[M_2]$

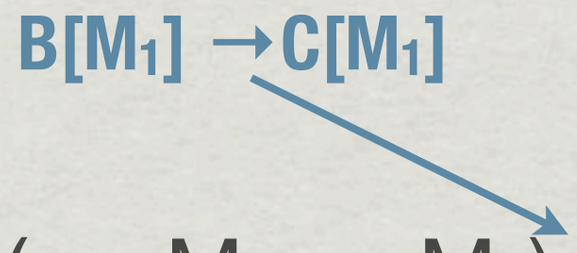
map for Functions

Action of map given by each type constructor:

$$\mathrm{map}_{x:A.B \to C}\,(\alpha : M_1 \lesssim_A M_2)\ f =$$
$$\quad \lambda\, x{:}B[M_2].\ \mathrm{map}_{x:A.C}\,\alpha\ (f\ (\mathrm{map}_{x:A.B}\,\alpha\ x))$$

map for Functions

Action of map given by each type constructor:



$$\mathrm{map}_{x:A.B \to C}\ (\alpha : M_1 \lesssim_A M_2)\ f =$$
$$\lambda\ x{:}B[M_2].\ \mathrm{map}_{x:A.C}\ \alpha\ (f\ (\mathrm{map}_{x:A.B}\ \alpha\ x))$$

map for Functions

Action of map given by each type constructor:

$B[M_1] \to C[M_1]$

$map_{x:A.B \to C}\ (\alpha : M_1 \lesssim_A M_2)\ f =$

$\lambda\ x{:}B[M_2].\ map_{x:A.C}\ \alpha\ (f\ (map_{x:A.B}\ \alpha\ x))$

Goal: $B[M_2] \to C[M_2]$

map for Functions

Action of map given by each type constructor:

B[M_1] → C[M_1]

$$\mathrm{map}_{x:A.B \to C}\ (\alpha : M_1 \lesssim_A M_2)\ f =$$
$$\lambda\ x{:}B[M_2].\ \mathrm{map}_{x:A.C}\ \alpha\ (f\ (\mathrm{map}_{x:A.B}\ \alpha\ x))$$

Goal: B[M_2] → C[M_2]

Contravariant:
B[M_2] → B[M_1]

map for Functions

Action of map given by each type constructor:

$B[M_1] \to C[M_1]$

$$\mathrm{map}_{x:A.B \to C}\ (\alpha : M_1 \lesssim_A M_2)\ f =$$
$$\lambda\ x{:}B[M_2].\ \mathrm{map}_{x:A.C}\ \alpha\ (f\ (\mathrm{map}_{x:A.B}\ \alpha\ x))$$

Goal: $B[M_2] \to C[M_2]$

Covariant:
$C[M_1] \to C[M_2]$

Contravariant:
$B[M_2] \to B[M_1]$

Variances

$$\frac{\Gamma \text{ ctx}}{\Gamma^{\text{op}} \text{ ctx}}$$

Contravariant **Covariant**

$$\frac{\Gamma^{\text{op}} \vdash A \text{ type} \quad \Gamma \vdash B \text{ type}}{\Gamma \vdash A \to B \text{ type}}$$

Functorial Syntax [FPT'99,AR'99,H'99]

Type Formula[Ψ : Ctx] representing formulas of DSL

Type Ctx:

elements: representations of DSL contexts Ψ
transformations Ψ ≲ Ψ':

DSL substitutions Ψ' ⊢ σ : Ψ

Functorial Syntax

Datatype definition in DTT:

$$\begin{array}{lcl} \text{formula} & : & \text{ctx} \to \text{set} \\ \text{formula}\ \psi & \cong & \text{v of } (\text{formula} \in \psi) \mid \text{says of principal } \psi \times \text{formula } \psi \end{array}$$

*action of **formula** on transformations =*
the structural properties!

$\text{map}_{x.\text{Formula}[x]}\ (\sigma : \Psi \lesssim \Psi') : \text{Formula}[\Psi] \to \text{Formula}\ [\Psi']$

Generalizations

I show that this extends to

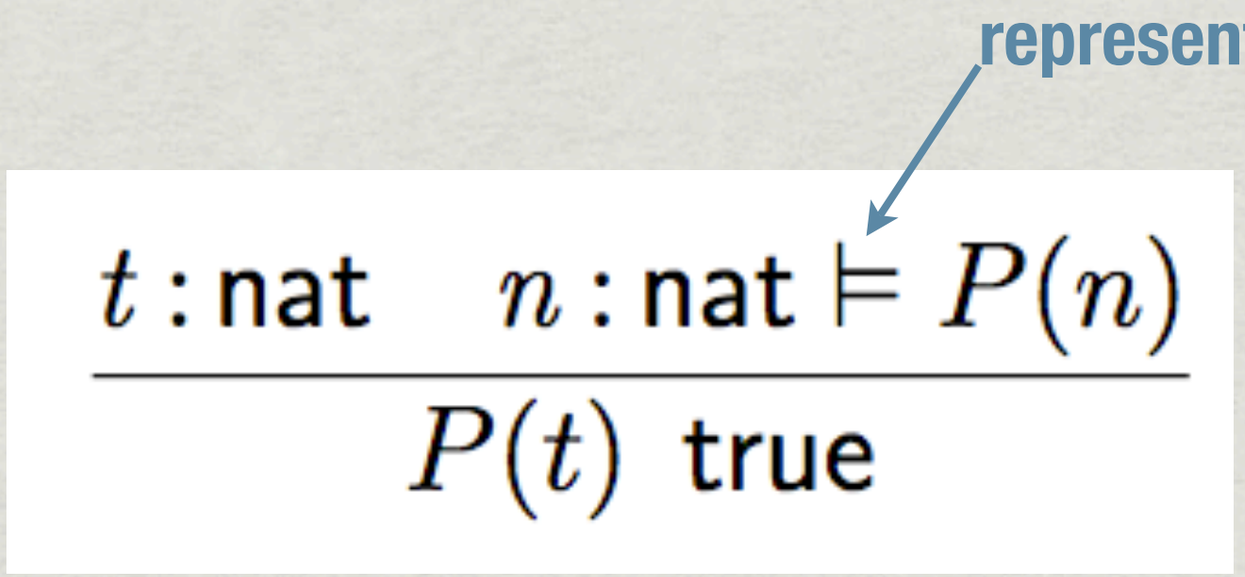
- admissibility premises
- dependent types

$$\frac{t : \mathsf{nat} \quad n : \mathsf{nat} \vDash P(n)}{P(t)\ \ \mathsf{true}}$$

Generalizations

I show that this extends to

- admissibility premises
- dependent types



Answers

- When do structural properties exist?
- Dependent types?
  - subst. into derivation yields subst. into judgement

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau, \Gamma' \vdash J}{\Gamma, \Gamma'[e/x] \vdash J[e/x]}$$

- requires *composition*

$$A[t/x][s/y] = A[s/y][t[s/y]/x]$$

Answers

- When do structural properties exist? **track variances**
- Dependent types?
  - subst. into derivation yields subst. into judgement

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau, \Gamma' \vdash J}{\Gamma, \Gamma'[e/x] \vdash J[e/x]}$$

- requires *composition*

$$A[t/x][s/y] = A[s/y][t[s/y]/x]$$

Answers

- When do structural properties exist? **track variances**
- Dependent types?
  - subst. into derivation yields subst. into judgement

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau, \Gamma' \vdash J}{\Gamma, \Gamma'[e/x] \vdash J[e/x]}$$

action on transform. at ∑

- requires *composition*

$$A[t/x][s/y] = A[s/y][t[s/y]/x]$$

Answers

- When do structural properties exist? **track variances**
- Dependent types?
  - subst. into derivation yields subst. into judgement

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau, \Gamma' \vdash J}{\Gamma, \Gamma'[e/x] \vdash J[e/x]}$$

action on transform. at ∑

- requires *composition*

$$A[t/x][s/y] = A[s/y][t[s/y]/x]$$

compos. law for map

Answers

Natural deductions $\Psi \vdash F$ where F can depend on Ψ

represented by

nd : ($\Sigma(\Psi$: Ctx). Formula[Ψ]) → type

Answers

Natural deductions $\Psi \vdash F$ where F can depend on Ψ

represented by

nd : ($\Sigma(\Psi$: Ctx). Formula[Ψ]) → type

Transformation (Ψ , F) ≲ (Ψ’ , F’) is exactly

- substitution Ψ’ ⊢ σ : Ψ
- such that F’ = map σ F

Answers

Natural deductions $\Psi \vdash F$ where F can depend on Ψ

represented by

$nd : (\Sigma(\Psi : Ctx).\ Formula[\Psi]) \to type$

Transformation $(\Psi , F) \lesssim (\Psi' , F')$ is exactly

- substitution $\Psi' \vdash \sigma : \Psi$
- such that $F' = map\ \sigma\ F$

so $map_{p.nd\ p}\ \sigma : nd\ \Psi\ F \to nd\ \Psi'\ F[\sigma]$

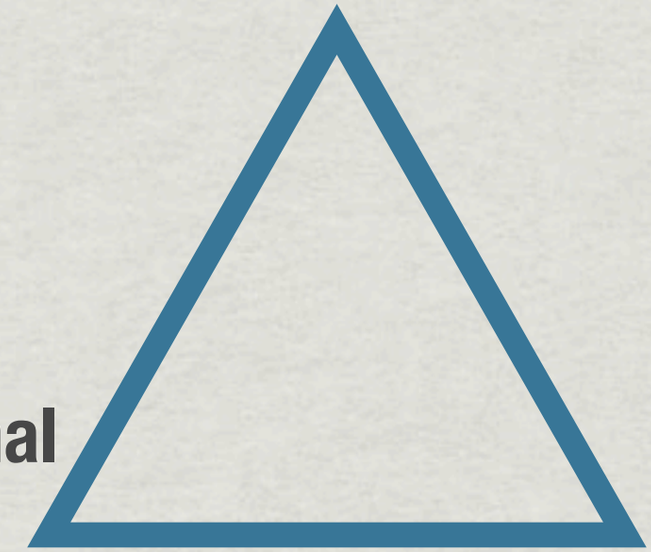
Part III

A language with directed types provides a useful framework for describing the structural properties of a dependently typed logical framework

Higher-Dimensional Directed Type Theory

Higher-Dimensional Symmetric Type Theory

types in intensional type theory



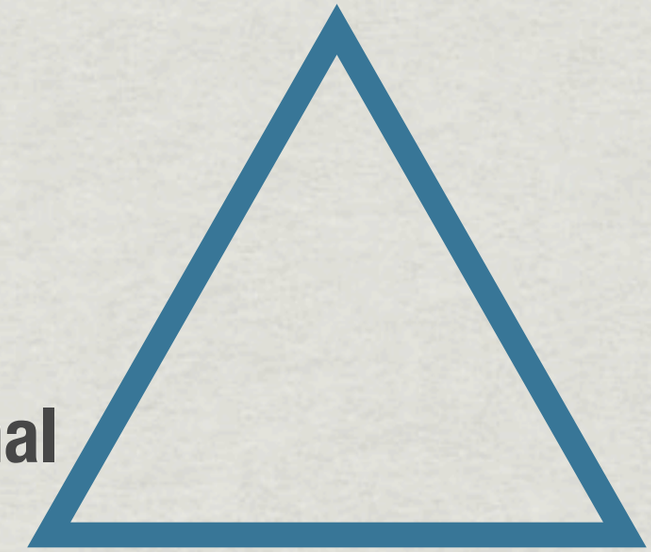
higher-dimensional groupoids in category theory

higher homotopy types in homotopy theory

justifies working up to (higher) isomoprhism

Higher-Dimensional Directed Type Theory

types in
directed type theory



higher-dimensional
categories
in category theory

higher
homotopy types
in directed homotopy theory

justifies working up to transformation

Semantics of DTT

- Context Γ denotes a category
- Type Γ ⊢ A type denotes a functor Γ → *Cat*
- Term Γ ⊢ M : A denotes
  a “dependently typed functor” Γ → A
- Transformation M ≲ N denotes
  a natural transformation

this is the 2-dimensional case in a hierarchy!

Contributions

- New examples of programming with domain-specific logics [Chapters 3 and 4]
- An investigation into mixing derivability and admissibility [Chapter 5, 6, 7, 8]
- A new notion of Directed Type Theory, corresponding to higher-dimensional category theory and homotopy theory [Chapters 7,8]

Part I

It is possible to define, study, automate, and use domain-specific logics within a dependently typed programming language

Part II

It is possible to implement, within a dependently typed programming language, a simply typed logical framework that allows derivability and admissibility to be mixed in novel and interesting ways.

Part III

A language with directed types provides a useful framework for describing the structural properties of a dependently typed logical framework

Future Work

- DTT, theory: inductive types, directed hom-types, opposite types, covariant Π
- DTT, practice: implementation, decidable definitional equality
- More examples of domain-specific logics, and bigger programs verified using them

Thanks to

Thanks to

- My advisor, Robert Harper

Thanks to

- My advisor, Robert Harper
- My committee, Karl Crary, Frank Pfenning, and Greg Morrisett

Thanks to

- My advisor, Robert Harper
- My committee, Karl Crary, Frank Pfenning, and Greg Morrisett
- My coauthors, Noam Zeilberger and Jamie Morgenstern

Thanks to

- My advisor, Robert Harper
- My committee, Karl Crary, Frank Pfenning, and Greg Morrisett
- My coauthors, Noam Zeilberger and Jamie Morgenstern
- Friends in the PoP group and philosophy dept.

Thanks to

- My advisor, Robert Harper
- My committee, Karl Crary, Frank Pfenning, and Greg Morrisett
- My coauthors, Noam Zeilberger and Jamie Morgenstern
- Friends in the PoP group and philosophy dept.
- My parents

Contributions

- New examples of programming with domain-specific logics [Chapters 3 and 4]
- An investigation into mixing derivability and admissibility [Chapters 5, 6, 7, 8]
- A new notion of Directed Type Theory, corresponding to higher-dimensional category theory and homotopy theory [Chapters 7 and 8]