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
Security-Typed Programming within DTP

Dan Licata and Jamie Morgenstern
It is easier to write an incorrect program than understand a correct one.

- 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.
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
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

Domain-specific logics
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 $\forall \ s:principal,$
$\forall \ i:principal,$
$\forall \ p:paper,$
$(member(i) \land i \ says \ student(s))$
$\implies MayRead(s, p)$

...

CMU says student(Alice)

...

Digital library

- All students of members can read papers
- CMU is a member

CMU

- Alice is a student
- Charlie is a student
- ...

(slide by Kumar Avijit)
Dependent Types! [Agda]

\[\text{read} : \text{prin} \rightarrow \text{file} \rightarrow \text{contents}\]
Dependent Types! [Agda]

read : prin → file → proof → contents
Dependent Types! [Agda]

\[
\text{read : prin} \rightarrow \text{file} \rightarrow \text{proof} \rightarrow \text{contents}
\]

\[
\text{read : } (k : \text{prin}) (f : \text{file}) (p : \text{proof}(\text{mayread}(k,f))) \rightarrow \text{contents}
\]
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:

\[ \Gamma \vdash A \quad \text{data } _\vdash _ : \text{Ctx} \to \text{Propo} \to \text{Type} \]
Representing Logic

Sequent as indexed inductive definition:

\[ \Gamma \vdash A \quad \text{data } \vdash \_ : \text{Ctx} \to \text{Propo} \to \text{Type} \]

Classifying *only* well-formed derivations:

\[ \mathcal{D} : \Gamma \vdash A \]

\[ \Gamma \vdash A \quad \text{data } \vdash \_ : \text{Ctx} \to \text{Propo} \to \text{Type} \]

\[ \mathcal{D} : \Gamma \vdash A \]
Representing Logic

Sequent as indexed inductive definition:

\[ \Gamma \vdash A \quad \text{data} \quad \vdash_\_ : \text{Ctx} \to \text{Propo} \to \text{Type} \]

Classifying only well-formed derivations:

\[ \emptyset : \Gamma \vdash A \]

Inference rules as datatype constructors:

\[ \Gamma, A \vdash B \quad \Rightarrow \quad \emptyset R : \forall \{\Gamma A B\} \]
\[ \to (A :: \Gamma) \vdash B \]
\[ \to \Gamma \vdash (A \supset B) \]
Representing Logic

Sequent as indexed inductive definition:

\[ \Gamma \vdash A \quad \text{data} \_ \vdash \_ : \text{Ctx} \rightarrow \text{Propo} \rightarrow \text{Type} \]

Classifying *only* well-formed derivations:

\[ \forall \Gamma \vdash A \quad \text{D} : \Gamma \vdash A \]

Inference rules as datatype constructors:

\[
\frac{\Gamma, A \vdash B}{\Gamma \vdash A \bowtie B} \quad \text{R} : \forall \{\Gamma A B\}
\rightarrow (A :: \Gamma) \vdash B
\rightarrow \Gamma \vdash (A \bowtie B)
\]

dependent de Bruijn indices
Theorem Prover

Implemented a *certified* theorem prover:

\[
\text{prove} : (\Gamma : \text{Ctx}) (A : \text{Propo}) \rightarrow \text{Maybe } (\Gamma \vdash A)
\]
Theorem Prover

Implemented a certified theorem prover:

\[
\text{prove} : (\Gamma : \text{Ctx}) \ (A : \text{Propo}) \rightarrow \text{Maybe} (\Gamma \vdash A)
\]

Important that Propos are *inductive*!

data Propo where

\[\text{says} : \text{Principal} \rightarrow \text{Propo} \rightarrow \text{Propo}\]

...
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
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:

\[ \text{can use a variable if } s \geq 1.0 \]
Each type $A$ denotes a metric space:

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

- $A \vdash B$ means $f : |A| \rightarrow |B|$ such that if $\text{dist}_A(x,y) \leq r$ then $\text{dist}_B(f(x), f(y)) \leq r$
Primitives

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

\[ \text{cmpswp} : \text{real} -o \text{real} -o \text{real} \otimes \text{real} \]
\[ \text{rsplit} : \text{real} -o \text{real} \otimes \text{real} \]
Primitives

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

\[
\text{cmpswp} : \text{real} \iff \text{real} \iff \text{real} \otimes \text{real}
\]

\[
\text{rsplit} : \text{real} \iff \text{real} \otimes \text{real}
\]

Need to be baked into the language
Extensible Diff. Priv.  
[Chap 4]

- Implement the semantics using dependent types  
  \( (\Pi x,y,r. \text{dist}_A(x,y) \leq r \rightarrow \text{dist}_B(f x, f y) \leq r) \)

- Primitives implemented and proved sound  
  in the semantics

- Build affine type system on top
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

\[ J_1 \ldots J_n \rightarrow J \]
A Tale of Two Consequence Relations

$J_1 \ldots J_n \rightarrow J$

- entailment
- assumptions
- conclusion
Derivability \( \vdash \)

Polynomials over the reals:

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

**Substitution:** plug in for the variable

- \[ f(3) = 3^2 + 2*3 + 1 \]
- \[ f(y+5) = (y+5)^2 + 2(y+5) + 1 \]
Derivability (⊢)

Polynomials over the reals:

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

Substitution: plug in for the variable

\[ f(3) = 3^2 + 2 \cdot 3 + 1 \]
\[ f(y+5) = (y+5)^2 + 2(y+5) + 1 \]
Derivability (⊢)

If (A implies B) and A then B

\[
\begin{align*}
(A \text{ implies } B) & \quad + \quad \begin{array}{c} 
\text{A} \\
\text{B}
\end{array} \quad = \quad \begin{array}{c} 
\text{A} \\
\text{B}
\end{array}
\end{align*}
\]
Derivability ($\vdash$)

If (A implies B) and A then B

$A \text{ true} \vdash B \text{ true}$

(A implies B)
Derivability ($\vdash$)

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

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

**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}),
 ... }
Admissibility ($\models$)

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

f : $|A| \rightarrow |B|$ such that
$\text{dist}_A(x,y) \leq r \rightarrow \text{dist}_B(f\ x, f\ y) \leq r$

Admissibility $J_1 \models J_2$: inductive proofs and functional programs
Structural Properties

\[ \frac{\Gamma, u : J, \Gamma' \vdash J}{\Gamma, \Gamma' \vdash J} u \]

\[ \frac{\Gamma, \Gamma' \vdash J_1 \quad \Gamma, u : J_1, \Gamma' \vdash J_2}{\Gamma, \Gamma' \vdash J_2} \text{ subst} \]

\[ \frac{\Gamma, \Gamma' \vdash J'}{\Gamma, u : J, \Gamma' \vdash J'} \text{ weakening} \]

\[ \frac{\Gamma, u_2 : J_2, u_1 : J_1, \Gamma' \vdash J'}{\Gamma, u_1 : J_1, u_2 : J_2, \Gamma' \vdash J'} \text{ exchange} \]

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

\[
\frac{l_1 = l_2 \models \text{false} \quad \text{lookup}(M, l_1) = \nu}{\text{lookup}(M[l_2 \mapsto \_], l_1) = \nu}
\]

\(\omega\)-rule:

\[
\frac{t: \text{nat} \quad n: \text{nat} \models P(n)}{P(t) \; \text{true}}
\]
Admissibility premises

Negated premises:

\[
\frac{l_1 = l_2 \models \text{false} \quad \text{lookup}(M, l_1) = v}{\text{lookup}(M[l_2 \mapsto \_], l_1) = v}
\]

\[
\frac{t : \text{nat} \quad n : \text{nat} \models P(n)}{P(t) \quad \text{true}}
\]

\(\omega\)-rule:

Concise representations of pattern matching [Zeilberger]
Problem

can no longer weaken with equality deriv. assumptions

\[
\begin{align*}
  l_1 = l_2 & \vdash \text{false} \\
  \text{lookup}(M, l_1) & = v \\
  \text{lookup}(M[l_2 \mapsto _], l_1) & = v
\end{align*}
\]

\[J1 \not\vdash (J2 \vdash J3)\]
\[\text{doesn’t necessarily follow from} \]
\[ (J2 \vdash J3) \]
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

★ requires composition
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

(DTT
(Take 2)
Directed Type Theory
Directed Types

Each type has notion of transformation on elements:

\[ M_1 \preceq_A M_2 \]

Every type family \( x:A \vdash B \) type respects trans.:

\[
\Gamma, x:A \vdash B \text{ type} \quad \Gamma \vdash \alpha : M_1 \preceq_A M_2 \quad \Gamma \vdash M : B[M_1/x] \\
\overline{\Gamma \vdash \text{map}_{x:A \cdot B} \alpha M : B[M_2/x]}
\]
Directed Types

Each type has notion of transformation on elements:

\[ M_1 \simeq_A M_2 \]

judgement, not type

Every type family \( x:A \vdash B \) type respects trans.:

\[
\Gamma, x:A \vdash B \text{ type} \quad \Gamma \vdash \alpha : M_1 \preceq_A M_2 \quad \Gamma \vdash M : B[M_1/x] \\
\Gamma \vdash \text{map}_{x:A} \cdot B \alpha M : B[M_2/x]
\]
map for Pairs

Action of map given by each type constructor:

\[
\text{map}_{x:A.B \times C} (\alpha : M_1 \cong_A M_2) (e, e') = \\
(\text{map}_{x:A.B} \alpha e, \text{map}_{x:A.C} \alpha e')
\]
map for Pairs

Action of map given by each type constructor:

\[
\text{map}_x : A.B \times C (\alpha : M_1 \equiv_A M_2) (e, e') = (\text{map}_x : A.B \alpha e, \text{map}_x : A.C \alpha e')
\]
map for Pairs

Action of map given by each type constructor:

$$\text{map}_{x:A.B \times C} (\alpha : M_1 \cong_A M_2) (e, e') = (\text{map}_{x:A.B} \alpha \ e, \text{map}_{x:A.C} \alpha \ e')$$

Goal: $B[M_2] \times C[M_2]$
map for Pairs

Action of map given by each type constructor:

\[
\text{map}_{x:A.B \times C} (\alpha : M_1 \approx_A M_2) (e, e') = (\text{map}_{x:A.B} \alpha e, \text{map}_{x:A.C} \alpha e')
\]

Goal: \(B[M_2] \times C[M_2]\)
map for Functions

Action of map given by each type constructor:

$$\text{map}_{x:A.B \to C} (\alpha : M_1 \cong_A M_2) \ f = \lambda \ x:B[M_2]. \ \text{map}_{x:A.C} \alpha \ (f \ (\text{map}_{x:A.B} \alpha \ x))$$
map for Functions

Action of map given by each type constructor:

\[
\text{map}_{x:A.B \to C} (\alpha : M_1 \equiv_A M_2) \ f = \\
\lambda x:B[M_2]. \ \text{map}_{x:A.C} \ \alpha \ (f \ (\text{map}_{x:A.B} \ \alpha \ x))
\]
map for Functions

Action of map given by each type constructor:

\[ \mathbb{B}[M_1] \rightarrow \mathbb{C}[M_1] \]

\[ \text{map}_{x:A.B \rightarrow C} (\alpha : M_1 \cong_{A} M_2) f = \]

\[ \lambda x:B[M_2]. \text{map}_{x:A.C} \alpha (f (\text{map}_{x:A.B} \alpha x)) \]

Goal: \( B[M_2] \rightarrow C[M_2] \)
map for Functions

Action of map given by each type constructor:

\[ \text{map}_{x:A.B \rightarrow C} (\alpha : M_1 \cong_A M_2) f = \lambda x:B[M_2]. \text{map}_{x:A.C} \alpha (f (\text{map}_{x:A.B} \alpha x)) \]

Goal: \( B[M_2] \rightarrow C[M_2] \)

Contravariant:
\( B[M_2] \rightarrow B[M_1] \)
map for Functions

Action of map given by each type constructor:

\[
\begin{align*}
B[M_1] & \rightarrow C[M_1] \\
\text{map}_{x:A.B} \rightarrow C (\alpha : M_1 \cong_A M_2) f = \\
\lambda x:B[M_2]. \text{map}_{x:A.C} \alpha (f (\text{map}_{x:A.B} \alpha x)) \\
\text{Goal: } B[M_2] & \rightarrow C[M_2] \\
\text{Covariant:} & \\
C[M_1] & \rightarrow C[M_2] \\
\text{Contravariant:} & \\
B[M_2] & \rightarrow B[M_1]
\end{align*}
\]
Variances

Contravariant

\[ \Gamma^{\text{op}} \vdash A \text{ type} \]
\[ \Gamma \vdash B \text{ type} \]
\[ \Gamma \vdash A \rightarrow B \text{ type} \]

Covariant

\[ \Gamma \text{ ctx} \]
\[ \Gamma^{\text{op}} \text{ ctx} \]
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:

\[
\begin{align*}
\text{formula} & : \text{ctx} \to \text{set} \\
\text{formula } \psi & \equiv \nu \text{ of } (\text{formula } \in \psi) \mid \text{says of principal } \psi \times \text{formula } \psi
\end{align*}
\]

action of \textbf{formula} on transformations = \textbf{the structural properties}!

\[
\text{map}_x.\text{Formula}[x] \ (\sigma : \Psi \equiv \Psi') : \text{Formula}[\Psi] \to \text{Formula}[\Psi']
\]
Generalizations

I show that this extends to

- admissibility premises
- dependent types

\[
\frac{t : \text{nat} \quad n : \text{nat} \models P(n)}{P(t) \text{ true}}
\]
Generalizations

I show that this extends to

- admissibility premises
- dependent types

represented by $\rightarrow$ or $\prod$

\[
\begin{array}{c}
t : \text{nat} \\
n : \text{nat} \vdash P(n) \\
\hline
P(t) \text{ true}
\end{array}
\]
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?
★ 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?
- Dependent types?
  - subst. into derivation yields subst. into judgement
  - requires composition

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

- track variances
- action on transform. at \( \sum \)

\[
A[t/x][s/y] = A[s/y][t[s/y]/x]
\]
Answers

« When do structural properties exist? 
« Dependent types? 

« subst. into derivation yields subst. into judgement

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

« requires composition

\[ A[t/x][s/y] = A[s/y][t[s/y]/x] \]
Answers

Natural deductions $\Psi \vdash F$ where $F$ can depend on $\Psi$ represented by

$\text{nd} : (\sum (\Psi : \text{Ctx}). \text{Formula}[\Psi]) \rightarrow \text{type}$
Answers

Natural deductions $\Psi \vdash F$ where $F$ can depend on $\Psi$ represented by

$\text{nd} : (\Sigma(\Psi : \text{Ctx}). \text{Formula}[\Psi]) \rightarrow \text{type}$

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

- substitution $\Psi' \vdash \sigma : \Psi$
- such that $F' = \text{map } \sigma F$
Natural deductions $\Psi \vdash F$ where $F$ can depend on $\Psi$ represented by

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

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

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

so $\text{map}_{\text{nd}_p} \sigma : \text{nd} \Psi F \to \text{nd} \Psi' F[\sigma]$
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

(DDT (Take 2))
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) isomorphism
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 $\Gamma$ denotes a category
- Type $\Gamma \vdash A$ type denotes a functor $\Gamma \rightarrow \text{Cat}$
- Term $\Gamma \vdash M : A$ denotes a “dependently typed functor” $\Gamma \rightarrow A$
- Transformation $M \simeq N$ 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*
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 $\Pi$

- 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]