

Type Systems

Lecture 12: Introduction to the Theory of Dependent Types

Neel Krishnaswami
University of Cambridge

Setting the stage

- In the last lecture, we introduced *dependent types*
- These are types which permit *program terms* to occur inside types
- This enables proving the correctness of programs through type checking

Syntax of Dependent Types

Terms $A, e ::= x$
| $\langle \rangle$ | 1
| $e e'$ | $\lambda x : A. e$ | $\Pi x : A. B$
| $\text{refl } e$ | $\text{subst}[x : A. B](e, e')$ | $(e = e' : A)$

Contexts $\Gamma ::= \cdot$ | $\Gamma, x : A$

- Types and expression grammars are merged
- Use judgements to decide whether something is a type or a term!

Judgements of Dependent Type Theory

Judgement	Description
$\Gamma \vdash A \text{ type}$	A is a type
$\Gamma \vdash e : A$	e has type A
$\Gamma \vdash A \equiv B \text{ type}$	A and B are identical types
$\Gamma \vdash e \equiv e' : A$	e and e' are equal terms of type A
$\Gamma \text{ ok}$	Γ is a well-formed context

The Unit Type

Type Formation

$$\frac{}{\Gamma \vdash 1 \text{ type}}$$

Introduction

$$\frac{}{\Gamma \vdash \langle \rangle : 1}$$

(No Elimination)

Function Types

Type Formation

$$\frac{\Gamma \vdash A \text{ type} \quad \Gamma, x : A \vdash B \text{ type}}{\Gamma \vdash \Pi x : A. B \text{ type}}$$

Introduction

$$\frac{\Gamma \vdash A \text{ type} \quad \Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A. e : \Pi x : A. B}$$

Elimination

$$\frac{\Gamma \vdash e : \Pi x : A. B \quad \Gamma \vdash e' : A}{\Gamma \vdash e e' : [e'/x]B}$$

Equality Types

Type Formation

$$\frac{\Gamma \vdash A \text{ type} \quad \Gamma \vdash e : A \quad \Gamma \vdash e' : A}{\Gamma \vdash (e = e' : A) \text{ type}}$$

Introduction

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \text{refl } e : (e = e : A)}$$

Elimination

$$\frac{\Gamma, x : A \vdash B \text{ type} \quad \Gamma \vdash A \text{ type} \quad \Gamma \vdash e : (e_1 = e_2 : A) \quad \Gamma \vdash e' : [e_1/x]B}{\Gamma \vdash \text{subst}[x : A. B](e, e') : [e_2/x]B}$$

(Equality elimination not the most general form!)

Variables and Equality

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \text{VAR}$$

$$\frac{\Gamma \vdash e : A \quad \Gamma \vdash A \equiv B \text{ type}}{\Gamma \vdash e : B}$$

What Is Judgmental Equality For?

$$\frac{\Gamma \vdash e : A \quad \Gamma \vdash A \equiv B \text{ type}}{\Gamma \vdash e : B}$$

- *THE* typing rule that makes dependent types expressive
- *THE* typing rule that makes dependent types difficult
- It enables computation inside of types

Example of Judgemental Equality

```
1 data Vec (A : Set) : Nat → Set where
2   [] : Vec A z
3   _,_ : {n : Nat} → A → Vec A n → Vec A (s n)
4
5   _+_ : Nat → Nat → Nat
6   z + m = m
7   s n + m = s (n + m)
8
9   append : {A : Set} → {n m : Nat} →
10          Vec A n → Vec A m → Vec A (n + m)
11   append [] ys = ys
12   append (x , xs) ys = (x , append xs ys)
```

Example

Suppose we have:

```
1 xs : Vec A (s (s z))
2 ys : Vec A (s (s z))
3
4 zs : Vec A (s (s (s (s z))))
5 zs = append xs ys
```

- Why is this well-typed?
- The signature tells us
`append xs ys : Vec A ((s (s z)) + (s (s z)))`
- This is well-typed because `(s (s z)) + (s (s z))`
evaluates to `(s (s (s (s z))))`

Judgmental Type Equality

$$\frac{}{\Gamma \vdash 1 \equiv 1 \text{ type}} \qquad \frac{\Gamma \vdash A \equiv X \text{ type} \quad \Gamma, x : A \vdash B \equiv Y \text{ type}}{\Gamma \vdash \Pi x : A. B \equiv \Pi x : X. Y \text{ type}}$$

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A \quad \Gamma \vdash e'_1 : A' \quad \Gamma \vdash e'_2 : A' \quad \Gamma \vdash A \equiv A' \text{ type} \quad \Gamma \vdash e_1 \equiv e'_1 : A \quad \Gamma \vdash e_2 \equiv e'_2 : A}{\Gamma \vdash (e_1 = e_2 : A) \equiv (e'_1 = e'_2 : A') \text{ type}}$$

Judgmental Term Equality: Equivalence Relation

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e \equiv e : A}$$

$$\frac{\Gamma \vdash e \equiv e' : A}{\Gamma \vdash e' \equiv e : A}$$

$$\frac{\Gamma \vdash e \equiv e' : A \quad \Gamma \vdash e' \equiv e'' : A}{\Gamma \vdash e \equiv e'' : A}$$

Judgmental Term Equality: Congruence Rules

$$\frac{}{\Gamma \vdash \langle \rangle \equiv \langle \rangle : 1} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x \equiv x : A}$$

$$\frac{\Gamma \vdash e_1 \equiv e'_1 : \Pi x : A. B \quad \Gamma \vdash e_2 \equiv e'_2 : A}{\Gamma \vdash e_1 e_2 \equiv e'_1 e'_2 : [e_1/x]B}$$

$$\frac{\Gamma \vdash A \equiv A' \text{ type} \quad \Gamma, x : A \vdash e \equiv e' : B}{\Gamma \vdash \lambda x : A. e \equiv \lambda x : A'. e' : \Pi x : A. B} \qquad \frac{\Gamma \vdash e \equiv e' : A}{\Gamma \vdash \text{refl } e \equiv \text{refl } e' : (e = e : A)}$$

$$\frac{\Gamma \vdash A \equiv A' \text{ type} \quad \Gamma, x : A \vdash B \equiv B' \text{ type} \quad \Gamma \vdash e_1 \equiv e'_1 : (e = e' : A) \quad \Gamma \vdash e_2 \equiv e'_2 : [e/x]B}{\Gamma \vdash \text{subst}[x : A. B](e_1, e_2) \equiv \text{subst}[x : A'. B'](e'_1, e'_2) : [e'/x]B}$$

Judgemental Equality: Conversion rules

$$\frac{\Gamma \vdash \lambda x : A. e : \Pi x : A. B \quad \Gamma \vdash e' : A \quad \Gamma \vdash [e'/x]e : [e'/x]B}{\Gamma \vdash (\lambda x : A. e) e' \equiv [e'/x]e : [e'/x]B}$$

$$\frac{\Gamma \vdash \text{subst}[x : A. B](\text{refl } e', e) : [e'/x]B \quad \Gamma \vdash e : [e'/x]B}{\Gamma \vdash \text{subst}[x : A. B](\text{refl } e', e) \equiv e : [e'/x]B}$$

$$\frac{\Gamma \vdash e \equiv e' : A \quad \Gamma \vdash A \equiv B \text{ type}}{\Gamma \vdash e \equiv e' : B}$$

Context Well-formedness

$$\frac{}{\cdot \text{ ok}} \qquad \frac{\Gamma \text{ ok} \quad \Gamma \vdash A \text{ type}}{\Gamma, x : A \text{ ok}}$$

Lemma: If $\Gamma \vdash C$ type, then

1. If $\Gamma, \Gamma' \vdash A$ type then $\Gamma, z : C, \Gamma' \vdash A$ type
2. If $\Gamma, \Gamma' \vdash e : A$ then $\Gamma, z : C, \Gamma' \vdash e : A$
3. If $\Gamma, \Gamma' \vdash A \equiv B$ type then $\Gamma, z : C, \Gamma' \vdash A \equiv B$ type
4. If $\Gamma, \Gamma' \vdash e \equiv e' : A$ then $\Gamma, z : C, \Gamma' \vdash e \equiv e' : A$
5. If Γ, Γ' ok then $\Gamma, z : C, \Gamma'$ ok

Proof: By mutual induction on derivations in 1-4, and a subsequent induction on derivations in 5

Metatheory: Substitution

If $\Gamma \vdash e' : C$, then

1. If $\Gamma, z : C, \Gamma' \vdash A$ type then $\Gamma, [e'/z]\Gamma' \vdash [e'/z]A$ type
2. If $\Gamma, z : C, \Gamma' \vdash e : A$ then $\Gamma, [e'/z]\Gamma' \vdash [e'/z]e : [e'/z]A$
3. If $\Gamma, z : C, \Gamma' \vdash A \equiv B$ type then
 $\Gamma, [e'/z]\Gamma' \vdash [e'/z]A \equiv [e'/z]B$ type
4. If $\Gamma, z : C, \Gamma' \vdash e_1 \equiv e_2 : A$ then
 $\Gamma, [e'/z]\Gamma' \vdash [e'/z]e_1 \equiv [e'/z]e_2 : [e'/z]A$
5. If $\Gamma, z : C, \Gamma'$ ok then $\Gamma, [e'/z]\Gamma'$ ok

Proof: By mutual induction on derivations in 1-4, and a subsequent induction on derivations in 5

Lemma: If $\Gamma \vdash C \equiv C'$ type then

1. If $\Gamma, z : C, \Gamma' \vdash A$ type then $\Gamma, z : C', \Gamma' \vdash A$ type
2. If $\Gamma, z : C, \Gamma' \vdash e : A$ then $\Gamma, z : C', \Gamma' \vdash e : A$
3. If $\Gamma, z : C, \Gamma' \vdash A \equiv B$ type then $\Gamma, z : C', \Gamma' \vdash A \equiv B$ type
4. If $\Gamma, z : C, \Gamma' \vdash e_1 \equiv e_2 : A$ then $\Gamma, z : C', \Gamma' \vdash e_1 \equiv e_2 : A$
5. If $\Gamma, z : C, \Gamma'$ ok then $\Gamma, z : C', \Gamma'$ ok

Proof: By mutual induction on derivations in 1-4, and a subsequent induction on derivations in 5

Lemma: If Γ ok then:

1. If $\Gamma \vdash e : A$ then $\Gamma \vdash A$ type.
2. If $\Gamma \vdash A \equiv B$ type then $\Gamma \vdash A$ type and $\Gamma \vdash B$ type.

Proof: By mutual induction on the derivations.

Reflections on Regularity

Calculus	Difficulty of Regularity Proof
STLC	Trivial
System F	Easy
Dependent Type Theory	A Lot of Work!

- Dependent types make all judgements mutually recursive
- Dependent types introduce new judgements (eg, judgemental equality)
- This makes establishing basic properties a lot of work

Advice on Language Design

- In your career, you will probably design at least a few languages
- Even a configuration file with notion of variable is a programming language
- Much of the pain in programming is dealing with the “accidental languages” that grew up around bigger languages (eg, shell scripts, build systems, package manager configurations, etc)

A Failure Mode

```
Lectures=1 2 3 4 5 6 7 8 9 10 11 12
```

```
LectureNames=$(patsubst %, lec-%.pdf, ${Lectures})
```

```
HandoutNames=$(patsubst %, lec-%-handout.pdf, ${Lectures})
```

```
lec-%-handout.pdf: lec-%.tex lec-%.pdf defs.tex
```

```
^^Icat handout-header.tex $< > $(patsubst %.pdf, %.tex, $@)
```

```
^^Ixelatex -shell-escape $(patsubst %.pdf, %.tex, $@)
```

```
^^Ixelatex -shell-escape $(patsubst %.pdf, %.tex, $@)
```

- Observe the specialized variable bindings %, \$< etc
- Even ordinary variables \${foo} are recursive
- Makes it hard to read, and hard to remember!

Takeaway Principles

The highest value ideas in this course are the most basic:

1. Figure out the abstract syntax tree up front
2. Design with contexts to figure out what variable scoping looks like
3. Sketch a substitution lemma to figure out if your notion of variable is right
4. Sketch a type safety argument