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

$$
\begin{array}{llcl}
\text{Terms} & A, e & ::= & x \\
 & & \mid & \langle\rangle \mid 1 \\
 & & \mid & e\,e' \mid \lambda x : A.\, e \mid \Pi x : A.\, B \\
 & & \mid & \mathrm{refl}\, e \mid \mathrm{subst}[x : A.\, B](e, e') \mid (e = e' : A) \\
 & & & \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

- 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
$\Gamma \vdash A$ type	A is a type
$\Gamma \vdash e : A$	e has type A
$\Gamma \vdash A \equiv B$ type	A and B are identical types
$\Gamma \vdash e \equiv e' : A$	e and e' are equal terms of type A
Γ 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

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

Introduction

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

Elimination

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

Equality Types

Type Formation

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

Introduction

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

Elimination

$$\frac{\begin{array}{c}\Gamma \vdash A \text{ type} \\ \Gamma, x : A \vdash B \text{ type} \qquad \Gamma \vdash e : (e_1 = e_2 : A) \qquad \Gamma \vdash e' : [e_1/x]B\end{array}}{\Gamma \vdash \mathsf{subst}[x : A.\, B](e, e') : [e_2/x]B}$$

(Equality elimination not the most general form!)

Variables and Equality

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var}$$

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

What Is Judgmental Equality For?

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

- *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

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A z
  _,_ : {n : Nat} → A → Vec A n → Vec A (s n)

_+_ : Nat → Nat → Nat
z   + m = m
s n + m = s (n + m)

append : {A : Set} → {n m : Nat} →
         Vec A n → Vec A m → Vec A (n + m)
append [] ys = ys
append (x , xs) ys = (x , append xs ys)
```

Example

Suppose we have:

```
xs : Vec A (s (s z))
ys : Vec A (s (s z))

zs : Vec A (s (s (s (s z))))
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

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

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : A \qquad \Gamma \vdash e_1' : A' \qquad \Gamma \vdash e_2' : A' \\ \Gamma \vdash A \equiv A' \text{ type} \qquad \Gamma \vdash e_1 \equiv e_1' : A \qquad \Gamma \vdash e_2 \equiv e_2' : A\end{array}}{\Gamma \vdash (e_1 = e_2 : A) \equiv (e_1' = e_2' : A') \text{ type}}$$

Judgmental Term Equality: Equivalence Relation

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

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

Judgmental Term Equality: Congruence Rules

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

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \Pi x : A.\, B \qquad \Gamma \vdash e_2 \equiv e_2' : A}{\Gamma \vdash e_1\, e_2 \equiv e_1'\, e_2' : [e_1/x]B}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash A \equiv A' \text{ type} \qquad \Gamma, x : A \vdash B \equiv B' \text{ type} \\ \Gamma \vdash e_1 \equiv e_1' : (e = e' : A) \qquad \Gamma \vdash e_2 \equiv e_2' : [e/x]B\end{array}}{\Gamma \vdash \mathrm{subst}[x : A.\, B](e_1, e_2) \equiv \mathrm{subst}[x : A'.\, B'](e_1', e_2') : [e'/x]B}$$

Judgemental Equality: Conversion rules

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

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

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

Context Well-formedness

$$\frac{}{\cdot \;\mathsf{ok}} \qquad\qquad \frac{\Gamma \;\mathsf{ok} \qquad \Gamma \vdash A \;\mathsf{type}}{\Gamma, x : A \;\mathsf{ok}}$$

Metatheory: Weakening

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

Metatheory: Context Equality

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

Metatheory: Regularity

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

```make
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
	cat handout-header.tex $< > $(patsubst %.pdf, %.tex, $@)
	xelatex -shell-escape $(patsubst %.pdf, %.tex, $@)
	xelatex -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