

Type Systems

Lecture 1

Neel Krishnaswami
University of Cambridge

Type Systems for Programming Languages

- Type systems lead a double life
- They are an essential part of modern programming languages
- They are a fundamental concept from logic and proof theory
- As a result, they form the most important channel for connecting theoretical computer science to practical programming language design.

What are type systems used for?

- Error detection via *type checking*
- Support for structuring large (or even medium) sized programs
- Documentation
- Efficiency
- Safety

A Language of Booleans and Integers

Terms $e ::= \text{true} \mid \text{false} \mid n \mid e \leq e \mid e + e \mid e \wedge e \mid \neg e$

Some terms make sense:

- $3 + 4$
- $3 + 4 \leq 5$
- $(3 + 4 \leq 7) \wedge (7 \leq 3 + 4)$

Some terms don't:

- $4 \wedge \text{true}$
- $3 \leq \text{true}$
- $\text{true} + 7$

Types for Booleans and Integers

Types $\tau ::= \text{bool} \mid \mathbb{N}$

Terms $e ::= \text{true} \mid \text{false} \mid n \mid e \leq e \mid e + e \mid e \wedge e$

- How to connect term (like $3 + 4$) with a type (like \mathbb{N})?
- Via a *typing judgement* $e : \tau$
- A two-place relation saying that “the term e has the type τ ”
- So $_ : _$ is an infix relation symbol
- How do we define this?

Typing Rules

$$\begin{array}{c} \frac{}{n : \mathbb{N}} \text{ NUM} \qquad \frac{}{\text{true} : \text{bool}} \text{ TRUE} \qquad \frac{}{\text{false} : \text{bool}} \text{ FALSE} \qquad \frac{e : \mathbb{N} \quad e' : \mathbb{N}}{e + e' : \mathbb{N}} \text{ PLUS} \\[2ex] \frac{e : \text{bool} \quad e' : \text{bool}}{e \wedge e' : \text{bool}} \text{ AND} \qquad \frac{e : \mathbb{N} \quad e' : \mathbb{N}}{e \leq e' : \text{bool}} \text{ LEQ} \end{array}$$

- Above the line: premises
- Below the line: conclusion

An Example Derivation Tree

$$\frac{\frac{\frac{}{3 : \mathbb{N}} \text{ NUM} \quad \frac{\frac{}{4 : \mathbb{N}} \text{ NUM}}{3 + 4 : \mathbb{N}} \text{ PLUS} \quad \frac{\frac{}{5 : \mathbb{N}} \text{ NUM}}{3 + 4 \leq 5 : \text{bool}} \text{ LEQ}}{3 + 4 \leq 5 : \text{bool}}$$

Adding Variables

Types $\tau ::= \text{bool} \mid \mathbb{N}$

Terms $e ::= \dots \mid x \mid \text{let } x = e \text{ in } e'$

- Example: $\text{let } x = 5 \text{ in } (x + x) \leq 10$
- But what type should x have: $x : ?$
- To handle this, the typing judgement must know what the variables are.
- So we change the typing judgement to be $\Gamma \vdash e : \tau$, where Γ associates a list of variables to their types.

Contexts $\Gamma ::= \cdot \mid \Gamma, x : \tau$

$$\frac{}{\Gamma \vdash n : \mathbb{N}} \text{NUM}$$

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \text{TRUE}$$

$$\frac{}{\Gamma \vdash \text{false} : \text{bool}} \text{FALSE}$$

$$\frac{\Gamma \vdash e : \mathbb{N} \quad \Gamma \vdash e' : \mathbb{N}}{\Gamma \vdash e + e' : \mathbb{N}} \text{PLUS}$$

$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e' : \text{bool}}{\Gamma \vdash e \wedge e' : \text{bool}} \text{AND}$$

$$\frac{\Gamma \vdash e : \mathbb{N} \quad \Gamma \vdash e' : \mathbb{N}}{\Gamma \vdash e \leq e' : \text{bool}} \text{LEQ}$$

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

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \text{let } x = e \text{ in } e' : \tau'} \text{LET}$$

Does this make sense?

- We have: a type system, associating elements from one grammar (the terms) with elements from another grammar (the types)
- We *claim* that this rules out “bad” terms
- But does it really?
- To prove, we must show *type safety*

Prelude: Substitution

We have introduced variables into our language, so we should introduce a notion of substitution as well

$$\begin{aligned} [e/x]\text{true} &= \text{true} \\ [e/x]\text{false} &= \text{false} \\ [e/x]n &= n \\ [e/x](e_1 + e_2) &= [e/x]e_1 + [e/x]e_2 \\ [e/x](e_1 \leq e_2) &= [e/x]e_1 \leq [e/x]e_2 \\ [e/x](e_1 \wedge e_2) &= [e/x]e_1 \wedge [e/x]e_2 \\ [e/x]z &= \begin{cases} e & \text{when } z = x \\ z & \text{when } z \neq x \end{cases} \\ [e/x](\text{let } z = e_1 \text{ in } e_2) &= \text{let } z = [e/x]e_1 \text{ in } [e/x]e_2 \quad (*) \end{aligned}$$

(*) α -rename to ensure z does not occur in e !

Structural Properties and Substitution

1. (Weakening) If $\Gamma, \Gamma' \vdash e : \tau$ then $\Gamma, x : \tau'', \Gamma' \vdash e : \tau$.
If a term typechecks in a context, then it will still typecheck in a bigger context.
2. (Exchange) If $\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e : \tau$ then $\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash e : \tau$.
If a term typechecks in a context, then it will still typecheck after reordering the variables in the context.
3. (Substitution) If $\Gamma \vdash e : \tau$ and $\Gamma, x : \tau \vdash e' : \tau'$ then $\Gamma \vdash [e/x]e' : \tau'$.
Substituting a type-correct term for a variable will preserve type correctness.

A Proof of Weakening

- Proof goes by *structural induction*
- Suppose we have a derivation tree of $\Gamma, \Gamma' \vdash e : \tau$
- By case-analysing the root of the derivation tree, we construct a derivation tree of $\Gamma, x : \tau'', \Gamma' \vdash e : \tau$, assuming inductively that the theorem works on subtrees.

$$\frac{}{\Gamma, \Gamma' \vdash n : \mathbb{N}} \text{NUM}$$

By assumption

$$\frac{}{\Gamma, x : \tau'', \Gamma' \vdash n : \mathbb{N}} \text{NUM}$$

By rule NUM

- Similarly for TRUE and FALSE rules

$$\frac{\Gamma, \Gamma' \vdash e_1 : \mathbb{N} \quad \Gamma, \Gamma' \vdash e_2 : \mathbb{N}}{\Gamma, \Gamma' \vdash e_1 + e_2 : \mathbb{N}} \text{ PLUS}$$

By assumption

$\Gamma, \Gamma' \vdash e_1 : \mathbb{N}$

Subderivation 1

$\Gamma, \Gamma' \vdash e_2 : \mathbb{N}$

Subderivation 2

$\Gamma, x : \tau'', \Gamma' \vdash e_1 : \mathbb{N}$

Induction on subderivation 1

$\Gamma, x : \tau'', \Gamma' \vdash e_2 : \mathbb{N}$

Induction on subderivation 2

$\Gamma, x : \tau'', \Gamma' \vdash e_1 + e_2 : \mathbb{N}$

By rule PLUS

- Similarly for LEQ and AND rules

Proving Weakening, 3/4

$$\frac{\Gamma, \Gamma' \vdash e_1 : \tau_1 \quad \Gamma, \Gamma', z : \tau_1 \vdash e_2 : \tau_2}{\Gamma, \Gamma' \vdash \text{let } z = e_1 \text{ in } e_2 : \tau_2} \text{LET} \quad \text{By assumption}$$

$\Gamma, \Gamma' \vdash e_1 : \tau_1$

Subderivation 1

$\Gamma, \Gamma', z : \tau_1 \vdash e_2 : \tau_2$

Subderivation 2

$\Gamma, x : \tau'', \Gamma' \vdash e_1 : \tau_1$

Induction on subderivation 1

Extended context

$\Gamma, x : \tau'', \quad \overbrace{\Gamma', z : \tau_1} \quad \vdash e_2 : \tau_2$ Induction on subderivation 2

$\Gamma, x : \tau'', \Gamma' \vdash \text{let } z = e_1 \text{ in } e_2 : \tau_2$ By rule LET

$$\frac{z : \tau \in \Gamma, \Gamma'}{\Gamma, \Gamma' \vdash z : \tau} \text{VAR} \quad \text{By assumption}$$

$z : \tau \in \Gamma, \Gamma'$ By assumption

$z : \tau \in \Gamma, x : \tau'', \Gamma'$ An element of a list is also in a bigger list

$\Gamma, x : \tau'', \Gamma' \vdash z : \tau$ By rule VAR

$$\frac{}{\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash n : \mathbb{N}} \text{NUM} \quad \text{By assumption}$$
$$\frac{}{\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash n : \mathbb{N}} \text{NUM} \quad \text{By rule NUM}$$

- Similarly for TRUE and FALSE rules

Proving Exchange, 2/4

$$\frac{\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 : \mathbb{N} \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_2 : \mathbb{N}}{\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 + e_2 : \mathbb{N}} \text{ PLUS}$$

By assumption

$$\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 : \mathbb{N}$$

Subderivation 1

$$\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_2 : \mathbb{N}$$

Subderivation 2

$$\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash e_1 : \mathbb{N}$$

Induction on subderivation 1

$$\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash e_2 : \mathbb{N}$$

Induction on subderivation 2

$$\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash e_1 + e_2 : \mathbb{N}$$

By rule PLUS

- Similarly for LEQ and AND rules

Proving Exchange, 3/4

$$\frac{\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 : \tau' \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma', z : \tau' \vdash e_2 : \tau''}{\Gamma, \Gamma' \vdash \text{let } z = e_1 \text{ in } e_2 : \tau''} \text{LET} \quad \text{By assumption}$$

$$\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 : \tau'$$

Subderivation 1

$$\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma', z : \tau' \vdash e_2 : \tau''$$

Subderivation 2

$$\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash e_1 : \tau'$$

Induction on s.d. 1

Extended context

$$\Gamma, x_2 : \tau_2, x_1 : \tau_1, \overbrace{\Gamma', z : \tau_1} \vdash e_2 : \tau''$$

Induction on s.d. 2

$$\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash \text{let } z = e_1 \text{ in } e_2 : \tau''$$

By rule LET

$$\frac{z : \tau \in \Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma'}{\Gamma, \Gamma' \vdash z : \tau} \text{VAR} \quad \text{By assumption}$$

$z : \tau \in \Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma'$ By assumption

$z : \tau \in \Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma'$ An element of a list is
also in a permutation of the list

$\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash z : \tau$ By rule VAR

A Proof of Substitution

- Proof also goes by *structural induction*
- Suppose we have derivation trees $\Gamma \vdash e : \tau$ and $\Gamma, x : \tau \vdash e' : \tau'$.
- By case-analysing the root of the derivation tree of $\Gamma, x : \tau \vdash e' : \tau'$, we construct a derivation tree of $\Gamma \vdash [e/x]e' : \tau'$, assuming inductively that substitution works on subtrees.

$\frac{}{\Gamma, x : \tau \vdash n : \mathbb{N}}$	NUM	
$\Gamma \vdash e : \tau$		By assumption
$\Gamma \vdash n : \mathbb{N}$		By rule NUM
$\Gamma \vdash [e/x]n : \mathbb{N}$		Def. of substitution

- Similarly for TRUE and FALSE rules

Proving Substitution, 2/4

$$\frac{\Gamma, x : \tau \vdash e_1 : \mathbb{N} \quad \Gamma, x : \tau \vdash e_2 : \mathbb{N}}{\Gamma, x : \tau \vdash e_1 + e_2 : \mathbb{N}}$$

By assumption: (1)

$$\Gamma \vdash e : \tau$$

By assumption: (2)

$$\Gamma, x : \tau \vdash e_1 : \mathbb{N}$$

Subderivation of (1): (3)

$$\Gamma, x : \tau \vdash e_2 : \mathbb{N}$$

Subderivation of (1): (4)

$$\Gamma \vdash [e/x]e_1 : \mathbb{N}$$

Induction on (2), (3): (5)

$$\Gamma \vdash [e/x]e_2 : \mathbb{N}$$

Induction on (2), (4): (6)

$$\Gamma \vdash [e/x]e_1 + [e/x]e_2 : \mathbb{N}$$

By rule PLUS on (5), (6)

$$\Gamma \vdash [e/x](e_1 + e_2) : \mathbb{N}$$

Def. of substitution

- Similarly for LEQ and AND rules

Proving Substitution, 3/4

$$\frac{\Gamma, x : \tau \vdash e_1 : \tau' \quad \Gamma, x : \tau, z : \tau' \vdash e_2 : \tau_2}{\Gamma, x : \tau \vdash \text{let } z = e_1 \text{ in } e_2 : \tau_2} \text{LET} \quad \text{By assumption: (1)}$$

$\Gamma \vdash e : \tau$	By assumption: (2)
$\Gamma, x : \tau \vdash e_1 : \tau'$	Subderivation of (1): (3)
$\Gamma, x : \tau, z : \tau' \vdash e_2 : \tau_2$	Subderivation of (1): (4)
$\Gamma \vdash [e/x]e_1 : \tau'$	Induction on (2) and (3): (4)
$\Gamma, z : \tau' \vdash e : \tau$	Weakening on (2): (5)
$\Gamma, z : \tau', x : \tau \vdash e_2 : \tau_2$	Exchange on (4): (6)
$\Gamma, z : \tau' \vdash [e/x]e_2 : \tau_2$	Induction on (5) and (6): (7)
$\Gamma \vdash \text{let } z = [e/x]e_1 \text{ in } [e/x]e_2 : \tau_2$	By rule LET on (6), (7)
$\Gamma \vdash [e/x](\text{let } z = e_1 \text{ in } e_2) : \tau_2$	By def. of substitution

$$\frac{z : \tau' \in \Gamma, x : \tau}{\Gamma, x : \tau \vdash z : \tau'} \text{VAR} \quad \text{By assumption}$$

$$\Gamma \vdash e : \tau \quad \text{By assumption}$$

Case $x = z$:

$$\Gamma \vdash [e/x]x : \tau \quad \text{By def. of substitution}$$

Proving Substitution, 4b/4

$$\frac{z : \tau' \in \Gamma, x : \tau}{\Gamma, x : \tau \vdash z : \tau'} \text{VAR} \quad \text{By assumption}$$

$$\Gamma \vdash e : \tau \quad \text{By assumption}$$

Case $x \neq z$:

$$z : \tau' \in \Gamma \quad \text{since } x \neq z \text{ and } z : \tau' \in \Gamma, x : \tau$$

$$\Gamma, z : \tau' \vdash z : \tau' \quad \text{By rule VAR}$$

$$\Gamma, z : \tau' \vdash [e/x]z : \tau' \quad \text{By def. of substitution}$$

Operational Semantics

- We have a language and type system
- We have a proof of substitution
- How do we say what *value* a program computes?
- With an *operational semantics*
- Define a grammar of *values*
- Define a two-place relation on terms $e \rightsquigarrow e'$
- Pronounced as “ e steps to e' ”

An operational semantics

Values $v ::= n \mid \text{true} \mid \text{false}$

$$\frac{e_1 \rightsquigarrow e'_1}{e_1 \wedge e_2 \rightsquigarrow e'_1 \wedge e_2} \text{ ANDCONG}$$

$$\frac{}{\text{true} \wedge e \rightsquigarrow e} \text{ ANDTRUE}$$

$$\frac{}{\text{false} \wedge e \rightsquigarrow \text{false}} \text{ ANDFALSE}$$

(similar rules for \leq and $+$)

$$\frac{e_1 \rightsquigarrow e'_1}{\text{let } z = e_1 \text{ in } e_2 \rightsquigarrow \text{let } z = e'_1 \text{ in } e_2} \text{ LETCONG}$$

$$\frac{}{\text{let } z = v \text{ in } e_2 \rightsquigarrow [v/z]e_2} \text{ LETSTEP}$$

Reduction Sequences

- A *reduction sequence* is a sequence of transitions $e_0 \rightsquigarrow e_1, e_1 \rightsquigarrow e_2, \dots, e_{n-1} \rightsquigarrow e_n$.
- A term e is *stuck* if it is not a value, and there is no e' such that $e \rightsquigarrow e'$

Successful sequence	Stuck sequence
$(3 + 4) \leq (2 + 3)$	$(3 + 4) \wedge (2 + 3)$
$\rightsquigarrow 7 \leq (2 + 3)$	$\rightsquigarrow 7 \wedge (2 + 3)$
$\rightsquigarrow 7 \leq 5$	$\rightsquigarrow ???$
$\rightsquigarrow \text{false}$	

Stuck terms are erroneous programs with no defined behaviour.

Type Safety

A program is *safe* if it never gets stuck.

1. (Progress) If $\cdot \vdash e : \tau$ then either e is a value, or there exists e' such that $e \rightsquigarrow e'$.
 2. (Preservation) If $\cdot \vdash e : \tau$ and $e \rightsquigarrow e'$ then $\cdot \vdash e' : \tau$.
- Progress means that well-typed programs are not stuck: they can always take a step of progress (or are done).
 - Preservation means that if a well-typed program takes a step, it will stay well-typed.
 - So a well-typed term won't reduce to a stuck term: the final term will be well-typed (due to preservation), and well-typed terms are never stuck (due to progress).

(Progress) If $\cdot \vdash e : \tau$ then either e is a value, or there exists e' such that $e \rightsquigarrow e'$.

- To show this, we do structural induction on the derivation of $\cdot \vdash e : \tau$.
- For each typing rule, we show that either e is a value, or can step.

$\frac{}{\cdot \vdash n : \mathbb{N}} \text{ NUM}$ By assumption

n is a value Def. of value grammar

Similarly for boolean literals...

Progress: Let-bindings

$$\frac{\cdot \vdash e_1 : \tau \quad x : \tau \vdash e_2 : \tau'}{\cdot \vdash \text{let } x = e_1 \text{ in } e_2 : \tau'} \text{ LET}$$

By assumption: (1)

$$\cdot \vdash e_1 : \tau$$

Subderivation of (1): (2)

$$x : \tau \vdash e_2 : \tau'$$

Subderivation of (1): (3)

$$e_1 \rightsquigarrow e'_1 \text{ or } e_1 \text{ value}$$

Induction on (2)

$$\text{Case } e_1 \rightsquigarrow e'_1 :$$

$$\text{let } x = e_1 \text{ in } e_2 \rightsquigarrow \text{let } x = e'_1 \text{ in } e_2$$

By rule LETCONG

$$\text{Case } e_1 \text{ value :}$$

$$\text{let } x = e_1 \text{ in } e_2 \rightsquigarrow [e_1/x]e_2$$

By rule LETSTEP

(Preservation) If $\cdot \vdash e : \tau$ and $e \rightsquigarrow e'$ then $\cdot \vdash e' : \tau$.

1. We will use structural induction again, but on which derivation?
2. Two choices: (1) $\cdot \vdash e : \tau$ and (2) $e \rightsquigarrow e'$
3. The right choice is induction on $e \rightsquigarrow e'$
4. We will still need to deconstruct $\cdot \vdash e : \tau$ alongside it!

Type Preservation: Let Bindings 1

$$\frac{e_1 \rightsquigarrow e'_1}{\text{let } x = e_1 \text{ in } e_2 \rightsquigarrow \text{let } x = e'_1 \text{ in } e_2}$$

By assumption: (1)

$$\frac{\cdot \vdash e_1 : \tau \quad x : \tau \vdash e_2 : \tau'}{\cdot \vdash \text{let } x = e_1 \text{ in } e_2 : \tau'}$$

By assumption: (2)

$$e_1 \rightsquigarrow e'_1$$

$$\cdot \vdash e_1 : \tau$$

$$x : \tau \vdash e_2 : \tau'$$

$$\cdot \vdash e'_1 : \tau$$

$$\cdot \vdash \text{let } x = e'_1 \text{ in } e_2 : \tau'$$

Subderivation of (1): (3)

Subderivation of (2): (4)

Subderivation of (2): (5)

Induction on (3), (4): (6)

Rule LET on (6), (4)

Type Preservation: Let Bindings 2

$$\frac{}{\text{let } x = v_1 \text{ in } e_2 \rightsquigarrow [v_1/x]e_2}$$

By assumption: (1)

$$\frac{\cdot \vdash v_1 : \tau \quad x : \tau \vdash e_2 : \tau'}{\cdot \vdash \text{let } x = v_1 \text{ in } e_2 : \tau'}$$

By assumption: (2)

$$\cdot \vdash v_1 : \tau$$

Subderivation of (2): (3)

$$x : \tau \vdash e_2 : \tau'$$

Subderivation of (2): (4)

$$\cdot \vdash [v_1/x]e_2 : \tau'$$

Substitution on (3), (4)

Given a language of program terms and a language of types:

- A type system ascribes types to terms
- An operational semantics describes how terms evaluate
- A type safety proof connects the type system and the operational semantics
- Proofs are intricate, but not difficult

1. Give cases of the operational semantics for \leq and $+$.
2. Extend the progress proof to cover $e \wedge e'$.
3. Extend the preservation proof to cover $e \wedge e'$.

(This should mostly be review of IB *Semantics of Programming Languages*.)

Type Systems

Lecture 2: The Curry-Howard Correspondence

Neel Krishnaswami
University of Cambridge

Type Systems for Programming Languages

- Type systems lead a double life
- They are a fundamental concept from logic and proof theory
- They are an essential part of modern programming languages

Natural Deduction

- In the early part of the 20th century, mathematics grew very abstract
- As a result, simple numerical and geometric intuitions no longer seemed to be sufficient to justify mathematical proofs (eg, Cantor's proofs about infinite sets)
- Big idea of Frege, Russell, Hilbert: what if we treated theorems and proofs as ordinary mathematical objects?
- Dramatic successes and failures, but the formal systems they introduced were unnatural – proofs didn't look like human proofs
- In 1933 (at age 23!) Gerhard Gentzen invented natural deduction
- “Natural” because the proof style is natural (with a little squinting)

Gerhard Gentzen



- Inventor of structural proof theory
- All of modern type theory rests on his ideas
- Indifferent to politics
- PhD advisor Paul Bernays
- Nazi party member (joined in 1933, age 24)
- Died in his mid 30s in Prague after the war

Natural Deduction: Propositional Logic

What are propositions?

- \top is a proposition
- $P \wedge Q$ is a proposition, if P and Q are propositions
- \perp is a proposition
- $P \vee Q$ is a proposition, if P and Q are propositions
- $P \supset Q$ is a proposition, if P and Q are propositions

These are the formulas of propositional logic (i.e., no quantifiers of the form “for all x , $P(x)$ ” or “there exists x , $P(x)$ ”).

- Some claims follow (e.g. $P \wedge Q \supset Q \wedge P$).
- Some claims don't. (e.g., $\top \supset \perp$)
- We judge which propositions hold, and which don't with judgements
- In particular, “ P true” means we judge P to be true.
- How do we justify judgements? With inference rules!

Truth and Conjunction

$$\frac{}{\top \text{ true}} \top I$$

$$\frac{P \text{ true} \quad Q \text{ true}}{P \wedge Q \text{ true}} \wedge I$$

$$\frac{P \wedge Q \text{ true}}{P \text{ true}} \wedge E_1$$

$$\frac{P \wedge Q \text{ true}}{Q \text{ true}} \wedge E_2$$

Implication

- To prove $P \supset Q$ in math, we assume P and prove Q
- Therefore, our notion of judgement needs to keep track of assumptions as well!
- So we introduce $\Psi \vdash P \text{ true}$, where Ψ is a list of assumptions
- Read: “Under assumptions Ψ , we judge P true”

$$\frac{P \in \Psi}{\Psi \vdash P \text{ true}} \text{ HYP}$$

$$\frac{\Psi, P \vdash Q \text{ true}}{\Psi \vdash P \supset Q \text{ true}} \supset I$$

$$\frac{\Psi \vdash P \supset Q \text{ true} \quad \Psi \vdash P \text{ true}}{\Psi \vdash Q \text{ true}} \supset E$$

Disjunction and Falsehood

$$\frac{\Psi \vdash P \text{ true}}{\Psi \vdash P \vee Q \text{ true}} \vee I_1$$

$$\frac{\Psi \vdash Q \text{ true}}{\Psi \vdash P \vee Q \text{ true}} \vee I_2$$

$$\frac{\Psi \vdash P \vee Q \text{ true} \quad \Psi, P \vdash R \text{ true} \quad \Psi, Q \vdash R \text{ true}}{\Psi \vdash R \text{ true}} \vee E$$

(no intro for \perp)

$$\frac{\Psi \vdash \perp \text{ true}}{\Psi \vdash R \text{ true}} \perp E$$

Example

$$\frac{\frac{(P \vee Q) \supset R, P \vdash (P \vee Q) \supset R \text{ true}}{(P \vee Q) \supset R, P \vdash R \text{ true}} \quad \frac{\frac{(P \vee Q) \supset R, P \vdash P \text{ true}}{(P \vee Q) \supset R, P \vdash P \vee Q \text{ true}}}{(P \vee Q) \supset R, P \vdash P \supset R \text{ true}} \quad \dots}{(P \vee Q) \supset R \vdash (P \supset R) \wedge (Q \supset R) \text{ true}} \quad \dots$$
$$\vdash ((P \vee Q) \supset R) \supset ((P \supset R) \wedge (Q \supset R)) \text{ true}$$

The Typed Lambda Calculus

Types	$X ::= 1 \mid X \times Y \mid 0 \mid X + Y \mid X \rightarrow Y$
Terms	$e ::= x \mid \langle \rangle \mid \langle e, e \rangle \mid \text{fst } e \mid \text{snd } e$ $\mid \text{abort} \mid L e \mid R e \mid \text{case}(e, Lx \rightarrow e', Ry \rightarrow e'')$ $\mid \lambda x : X. e \mid e e'$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : X$

A typing judgement is of the form $\Gamma \vdash e : X$.

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

$$\frac{\Gamma \vdash e : X \quad \Gamma \vdash e' : Y}{\Gamma \vdash \langle e, e' \rangle : X \times Y} \text{xI}$$

$$\frac{\Gamma \vdash e : X \times Y}{\Gamma \vdash \text{fst } e : X} \text{xE}_1$$

$$\frac{\Gamma \vdash e : X \times Y}{\Gamma \vdash \text{snd } e : Y} \text{xE}_2$$

Functions and Variables

$$\frac{x : X \in \Gamma}{\Gamma \vdash x : X} \text{HYP}$$

$$\frac{\Gamma, x : X \vdash e : Y}{\Gamma \vdash \lambda x : X. e : X \rightarrow Y} \rightarrow I$$

$$\frac{\Gamma \vdash e : X \rightarrow Y \quad \Gamma \vdash e' : X}{\Gamma \vdash e e' : Y} \rightarrow E$$

Sums and the Empty Type

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash Le : X + Y} +I_1$$

$$\frac{\Gamma \vdash e : Y}{\Gamma \vdash Re : X + Y} +I_2$$

$$\frac{\Gamma \vdash e : X + Y \quad \Gamma, x : X \vdash e' : Z \quad \Gamma, y : Y \vdash e'' : Z}{\Gamma \vdash \text{case}(e, Lx \rightarrow e', Ry \rightarrow e'') : Z} +E$$

(no intro for 0)

$$\frac{\Gamma \vdash e : 0}{\Gamma \vdash \text{abort } e : Z} 0E$$

Example

$$\begin{aligned} \lambda f : (X + Y) \rightarrow Z. \langle \lambda x : X. f(Lx), \lambda y : Y. f(Ry) \rangle \\ : \\ ((X + Y) \rightarrow Z) \rightarrow (X \rightarrow Z) \times (Y \rightarrow Z) \end{aligned}$$

You may notice a similarity here...!

The Curry-Howard Correspondence, Part 1

Logic	Programming
Formulas	Types
Proofs	Programs
Truth	Unit
Falsehood	Empty type
Conjunction	Pairing/Records
Disjunction	Tagged Union
Implication	Functions

Something missing: language semantics?

Operational Semantics of the Typed Lambda Calculus

Values $v ::= \langle \rangle \mid \langle v, v' \rangle \mid \lambda x : A. e \mid L v \mid R v$

The transition relation is $e \rightsquigarrow e'$, pronounced “ e steps to e' ”.

Operational Semantics: Units and Pairs

(no rules for unit)

$$\frac{e_1 \rightsquigarrow e'_1}{\langle e_1, e_2 \rangle \rightsquigarrow \langle e'_1, e_2 \rangle}$$

$$\frac{e_2 \rightsquigarrow e'_2}{\langle v_1, e_2 \rangle \rightsquigarrow \langle v_1, e'_2 \rangle}$$

$$\frac{}{\text{fst } \langle v_1, v_2 \rangle \rightsquigarrow v_1}$$

$$\frac{}{\text{snd } \langle v_1, v_2 \rangle \rightsquigarrow v_2}$$

$$\frac{e \rightsquigarrow e'}{\text{fst } e \rightsquigarrow \text{fst } e'}$$

$$\frac{e \rightsquigarrow e'}{\text{snd } e \rightsquigarrow \text{snd } e'}$$

Operational Semantics: Void and Sums

$$\frac{e \rightsquigarrow e'}{\text{abort } e \rightsquigarrow \text{abort } e'}$$

$$\frac{e \rightsquigarrow e'}{L e \rightsquigarrow L e'}$$

$$\frac{e \rightsquigarrow e'}{R e \rightsquigarrow R e'}$$

$$\frac{e \rightsquigarrow e'}{\text{case}(e, Lx \rightarrow e_1, Ry \rightarrow e_2) \rightsquigarrow \text{case}(e', Lx \rightarrow e_1, Ry \rightarrow e_2)}$$

$$\frac{}{\text{case}(L v, Lx \rightarrow e_1, Ry \rightarrow e_2) \rightsquigarrow [v/x]e_1}$$

$$\frac{}{\text{case}(R v, Lx \rightarrow e_1, Ry \rightarrow e_2) \rightsquigarrow [v/y]e_2}$$

$$\frac{e_1 \rightsquigarrow e'_1}{e_1 e_2 \rightsquigarrow e'_1 e_2}$$

$$\frac{e_2 \rightsquigarrow e'_2}{v_1 e_2 \rightsquigarrow v_1 e'_2}$$

$$\frac{}{(\lambda x : X. e) v \rightsquigarrow [v/x]e}$$

Five Easy Lemmas

1. (Weakening) If $\Gamma, \Gamma' \vdash e : X$ then $\Gamma, z : Z, \Gamma' \vdash e : X$.
2. (Exchange) If $\Gamma, y : Y, z : Z, \Gamma' \vdash e : X$ then $\Gamma, z : Z, y : Y, \Gamma' \vdash e : X$.
3. (Substitution) If $\Gamma \vdash e : X$ and $\Gamma, x : X \vdash e' : Y$ then $\Gamma \vdash [e/x]e' : Y$.
4. (Progress) If $\cdot \vdash e : X$ then e is a value, or $e \rightsquigarrow e'$.
5. (Preservation) If $\cdot \vdash e : X$ and $e \rightsquigarrow e'$, then $\cdot \vdash e' : X$.

Proof technique similar to previous lecture. But what does it mean, logically?

Two Kinds of Reduction Step

Congruence Rules	Reduction Rules
$\frac{e_1 \rightsquigarrow e'_1}{\langle e_1, e_2 \rangle \rightsquigarrow \langle e'_1, e_2 \rangle}$	$\frac{}{\text{fst } \langle v_1, v_2 \rangle \rightsquigarrow v_1}$
$\frac{e_2 \rightsquigarrow e'_2}{v_1 e_2 \rightsquigarrow v_1 e'_2}$	$\frac{}{(\lambda x : X. e) v \rightsquigarrow [v/x]e}$

- Congruence rules recursively act on a subterm
 - Controls evaluation order
- Reduction rules actually transform a term
 - Actually evaluates!

A Closer Look at Reduction

Let's look at the function reduction case:

$$(\lambda x : X. e) v \rightsquigarrow [v/x]e$$

$$\frac{\frac{\boxed{x : X \vdash e : Y}}{\cdot \vdash \lambda x : X. e : X \rightarrow Y} \rightarrow I \quad \boxed{\cdot \vdash v : X}}{\cdot \vdash (\lambda x : X. e) v : Y} \rightarrow E$$

- Reducible term = intro immediately followed by an elim
- Evaluation = removal of this detour

All Reductions Remove Detours

$$\overline{\text{fst } \langle v_1, v_2 \rangle \rightsquigarrow v_1}$$

$$\overline{\text{snd } \langle v_1, v_2 \rangle \rightsquigarrow v_2}$$

$$\overline{\text{case}(\text{L } v, \text{L } x \rightarrow e_1, \text{R } y \rightarrow e_2) \rightsquigarrow [v/x]e_1}$$

$$\overline{\text{case}(\text{R } v, \text{L } x \rightarrow e_1, \text{R } y \rightarrow e_2) \rightsquigarrow [v/y]e_2}$$

$$\overline{(\lambda x : X. e) v \rightsquigarrow [v/x]e}$$

Every reduction is of an introduction followed by an eliminator!

Values as Normal Forms

Values $v ::= \langle \rangle \mid \langle v, v' \rangle \mid \lambda x : A. e \mid L v \mid R v$

- Note that values are introduction forms
- Note that values are not reducible expressions
- So programs evaluate towards a normal form
- Choice of which normal form to look at it determined by evaluation order

The Curry-Howard Correspondence, Continued

Logic	Programming
Formulas	Types
Proofs	Programs
Truth	Unit
Falsehood	Empty type
Conjunction	Pairing/Records
Disjunction	Tagged Union
Implication	Functions
Normal form	Value
Proof normalization	Evaluation
Normalization strategy	Evaluation order

The Curry-Howard Correspondence is Not an Isomorphism

The logical derivation:

$$\frac{\frac{}{P, P \vdash P \text{ true}} \quad \frac{}{P, P \vdash P \text{ true}}}{P, P \vdash P \wedge P \text{ true}}$$

has 4 type-theoretic versions:

$$\begin{array}{cc} \frac{\vdots}{x : X, y : X \vdash \langle x, x \rangle : X \times X} & \frac{\vdots}{x : X, y : X \vdash \langle y, y \rangle : X \times X} \\[2em] \frac{\vdots}{x : X, y : X \vdash \langle x, y \rangle : X \times X} & \frac{\vdots}{x : X, y : X \vdash \langle y, x \rangle : X \times X} \end{array}$$

For the $1, \rightarrow$ fragment of the typed lambda calculus, prove type safety.

1. Prove weakening.
2. Prove exchange.
3. Prove substitution.
4. Prove progress.
5. Prove type preservation.

Type Systems

Lecture 3: Consistency and Termination

Neel Krishnaswami
University of Cambridge

From Type Safety to Stronger Properties

- In the last lecture, we saw how evaluation corresponded to proof normalization
- This was an act of knowledge transfer from computation to logic
- Are there any transfers we can make in the other direction?

- An important property of any logic is consistency: there are no proofs of \perp !
- Otherwise, the \perp E rule will let us prove anything.
- What does this look like in a programming language?

Types and Values

Types $X ::= 1 \mid X \times Y \mid 0 \mid X + Y \mid X \rightarrow Y$
Values $v ::= \langle \rangle \mid \langle v, v' \rangle \mid \lambda x : A. e \mid L v \mid R v$

- There are no values of type 0
- I.e., no normal forms of type 0
- But what about non-normal forms?

What Type Safety Does, and Doesn't Show

- We have proved type safety:
 - Progress: If $\cdot \vdash e : X$ then e is a value or $e \rightsquigarrow e'$.
 - Type preservation If $\cdot \vdash e : X$ and $e \rightsquigarrow e'$ then $\cdot \vdash e' : X$.
- If there were a closed term of type 0, then progress means it must always step (since there are no values of type 0)
- But the term it would step to also has type 0 (by preservation)
- So any closed term of type 0 must loop – it must step forever.

A Naive Proof that Does Not Work

Theorem: If $\cdot \vdash e : X$ then there is a value v such that $e \rightsquigarrow^* v$.

“Proof”: By structural induction on $\cdot \vdash e : X$

- | | | |
|------|-----------------------------------------------------------------------------------------------------------------------------------|--------------------------|
| | $\frac{\overbrace{\Gamma \vdash e : X \rightarrow Y}^{(2)} \quad \overbrace{\Gamma \vdash e' : X}^{(3)}}{\Gamma \vdash e e' : Y}$ | |
| (1) | $\Gamma \vdash e e' : Y$ | Assumption |
| (4) | $e \rightsquigarrow^* v$ | Induction on (2) |
| (5) | $e' \rightsquigarrow^* v'$ | Induction on (3) |
| (6) | $\cdot \vdash v : X \rightarrow Y$ | Preservation on (2), (4) |
| (7) | $\cdot \vdash v' : X$ | Preservation on (3), (5) |
| (8) | $\cdot \vdash v \equiv \lambda x : X. e'' : X \rightarrow Y$ | Canonical forms on (6) |
| (9) | $x : X \vdash e'' : Y$ | Subderivation |
| (10) | $\cdot \vdash [v'/x]e'' : Y$ | Substitution |
| | $\underbrace{\quad}_{\text{Can't do induction on this!}}$ | |

A Minimal Typed Lambda Calculus

Types $X ::= 1 \mid X \rightarrow Y \mid 0$

Terms $e ::= x \mid \langle \rangle \mid \lambda x : X. e \mid e e' \mid \text{abort } e$

Values $v ::= \langle \rangle \mid \lambda x : X. e$

$$\frac{X : X \in \Gamma}{\Gamma \vdash x : X} \text{HYP}$$

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

$$\frac{\Gamma, X \vdash e : Y}{\Gamma \vdash \lambda x : X. e : X \rightarrow Y} \rightarrow I$$

$$\frac{\Gamma \vdash e : X \rightarrow Y \quad \Gamma \vdash e' : X}{\Gamma \vdash e e' : Y} \rightarrow E$$

$$\frac{\Gamma \vdash e : 0}{\Gamma \vdash \text{abort } e : Z} 0E$$

$$\frac{e \rightsquigarrow e'}{\text{abort } e \rightsquigarrow \text{abort } e'}$$

$$\frac{e_1 \rightsquigarrow e'_1}{e_1 e_2 \rightsquigarrow e'_1 e_2}$$

$$\frac{e_2 \rightsquigarrow e'_2}{v_1 e_2 \rightsquigarrow v_1 e'_2}$$

$$\frac{}{(\lambda x : X. e) v \rightsquigarrow [v/x]e}$$

Theorem (Determinacy): If $e \rightsquigarrow e'$ and $e \rightsquigarrow e''$ then $e' = e''$

Proof: By structural induction on $e \rightsquigarrow e'$

Why Can't We Prove Termination

- We can't prove termination by structural induction
- Problem is that knowing a term evaluates to a function doesn't tell us that applying the function terminates
- We need to assume something stronger

A Logical Relation

1. We say that e halts if and only if there is a v such that $e \rightsquigarrow^* v$.
2. Now, we will define a type-indexed family of set of terms:
 - $\text{Halt}_0 = \emptyset$ (i.e, for all e , $e \notin \text{Halt}_0$)
 - $e \in \text{Halt}_1$ holds just when e halts.
 - $e \in \text{Halt}_{X \rightarrow Y}$ holds just when
 1. e halts
 2. For all e' , if $e' \in \text{Halt}_X$ then $(e\ e') \in \text{Halt}_Y$.
3. Hereditary definition:
 - Halt_1 halts
 - $\text{Halt}_{1 \rightarrow 1}$ preserves the property of halting
 - $\text{Halt}_{(1 \rightarrow 1) \rightarrow (1 \rightarrow 1)}$ preserves the property of preserving the property of halting...

The Goal

Imagine we can prove:

Conjecture: If $\cdot \vdash e : X$, then $e \in \text{Halt}_X$.

Then we know that every closed program terminates! But to prove this, we need to first establish a lemma or two.

Lemma: If $e \rightsquigarrow e'$ then $e' \in \text{Halt}_X$ iff $e \in \text{Halt}_X$.

Proof: By induction on X :

- (1) $e \rightsquigarrow e'$ Assumption
- (2) $e' \in \text{Halt}_1$ Assumption
- Case $X = 1, \Rightarrow$:
 - (3) $e' \rightsquigarrow^* v$ Definition of Halt_1
 - (4) $e \rightsquigarrow^* v$ Def. of transitive closure, (1) and (3)
 - (5) $e \in \text{Halt}_1$ Definition of Halt_1

- Case $X = 1$, \Leftarrow :
 - (1) $e \rightsquigarrow e'$ Assumption
 - (2) $e \in \text{Halt}_1$ Assumption
 - (3) $e \rightsquigarrow^* v$ Definition of Halt_1
 - (4) e is not a value: Since $e \rightsquigarrow e'$
 - (5) $e \rightsquigarrow e''$ and $e'' \rightsquigarrow^* v$ Definition of $e \rightsquigarrow^* v$
 - (6) $e'' = e'$ By determinacy on (1), (5)
 - (7) $e' \rightsquigarrow^* v$ By equality (6) on (5)
 - (8) $e' \in \text{Halt}_1$ Definition of Halt_1

Closure Lemma, 3/5

- Case $X = Y \rightarrow Z, \Rightarrow$:

- | | | |
|--------------------------------|-------------------------------------------------------|----------------------------------------------------|
| (1) | $e \rightsquigarrow e'$ | Assumption |
| (2) | $e' \in \text{Halt}_{Y \rightarrow Z}$ | Assumption |
| (3) | $e' \rightsquigarrow^* v$ | Def. of $\text{Halt}_{Y \rightarrow Z}$ |
| (4) | $\forall t \in \text{Halt}_Y, e' t \in \text{Halt}_Z$ | " |
| (5) | $e \rightsquigarrow^* v$ | Transitive closure, (1) and (3) |
| Assume $t \in \text{Halt}_Y$: | | |
| (6) | $e t \rightsquigarrow e' t$ | By congruence rule on (1) |
| (7) | $e' t \in \text{Halt}_Z$ | By (4) |
| | $e t \in \text{Halt}_Z$ | By induction on (6), (7) |
| (8) | $\forall t \in \text{Halt}_Y, e t \in \text{Halt}_Z$ | |
| (9) | $e \in \text{Halt}_{Y \rightarrow Z}$ | Def of $\text{Halt}_{Y \rightarrow Z}$ on (5), (8) |

Closure Lemma, 4/5

- Case $X = Y \rightarrow Z$, \Leftarrow :

- | | | |
|--------------------------------|------------------------------------------------------------------------------|----------------------------------------------------|
| (1) | $e \rightsquigarrow e'$ | Assumption |
| (2) | $e \in \text{Halt}_{Y \rightarrow Z}$ | Assumption |
| (3) | $e \rightsquigarrow^* v$ | Def. of $\text{Halt}_{Y \rightarrow Z}$ |
| (4) | $\forall t \in \text{Halt}_Y, e \ t \in \text{Halt}_Z$
e is not a value | "
Since (1) |
| (5) | $e \rightsquigarrow e''$ and $e'' \rightsquigarrow^* v$ | Definition of $e \rightsquigarrow^* v$ |
| (6) | $e'' = e'$ | By determinacy on (1), (5) |
| Assume $t \in \text{Halt}_Y$: | | |
| (7) | $e \ t \rightsquigarrow e' \ t$ | By congruence rule on (1) |
| (8) | $e \ t \in \text{Halt}_Z$
$e' \ t \in \text{Halt}_Z$ | By (4)
By induction on (6), (7) |
| (9) | $\forall t \in \text{Halt}_Y, e' \ t \in \text{Halt}_Z$ | |
| (10) | $e' \in \text{Halt}_{Y \rightarrow Z}$ | Def of $\text{Halt}_{Y \rightarrow Z}$ on (5), (8) |

Closure Lemma, 5/5

- Case $X = 0, \Rightarrow$:
 - (1) $e \rightsquigarrow e'$ Assumption
 - (2) $e' \in \text{Halt}_0$ Assumption
 - (3) $e' \in \emptyset$ Definition of Halt_0
 - (4) Contradiction!
- Case $X = 0, \Leftarrow$:
 - (1) $e \rightsquigarrow e'$ Assumption
 - (2) $e \in \text{Halt}_0$ Assumption
 - (3) $e \in \emptyset$ Definition of Halt_0
 - (4) Contradiction!

The Fundamental Lemma

Lemma:

If we have that:

- $x_1 : X_1, \dots, x_n : X_n \vdash e : Z$, and
- for $i \in \{1 \dots n\}$, $v_i \in \text{Halt}_{X_i}$

then $[v_1/x_1, \dots, v_n/x_n]e \in \text{Halt}_Z$

Proof:

By structural induction on $x_1 : X_1, \dots, x_n : X_n \vdash e : Z$!

The Fundamental Lemma, 1/5

- Case Hyp:

- | | | |
|-----|-----------------------------------------------------------------------------------------------------------|----------------------|
| | $\frac{x_j : X_j \in \overrightarrow{x_i : X_i}}{\overrightarrow{x_i : X_i} \vdash x_j : X_j} \text{HYP}$ | |
| (1) | $\overrightarrow{x_i : X_i} \vdash x_j : X_j$ | Assumption |
| (2) | $\overrightarrow{[v_i/x_i]}x_j = v_j$ | Def. of substitution |
| (3) | $v_j \in \text{Halt}_{X_j}$ | Assumption |
| (4) | $\overrightarrow{[v_i/x_i]}x_j \in \text{Halt}_{X_j}$ | Equality (2) on (3) |

- Case 1l:

- (1) $\overrightarrow{x_i : X_i \vdash \langle \rangle : 1}$ ^{1l} Assumption
- (2) $\overrightarrow{[v_i/x_i]} \langle \rangle = \langle \rangle$ Def. of substitution
- (3) $\langle \rangle \rightsquigarrow^* \langle \rangle$ Def. of transitive closure
- (4) $\langle \rangle \in \text{Halt}_1$ Def. of Halt_1
- (5) $\overrightarrow{[v_i/x_i]} \langle \rangle \in \text{Halt}_1$ Equality (2) on (4)

The Fundamental Lemma, 3a/5

- Case \rightarrow I:

$$(1) \frac{\overrightarrow{x_i : X_i}, y : Y \vdash e : Z}{\overrightarrow{x_i : X_i} \vdash \lambda y : Y. e : Y \rightarrow Z} \rightarrow I$$

Assumption

$$(2) \overrightarrow{x_i : X_i}, y : Y \vdash e : Z$$

Subderivation of (1)

$$(3) [\overrightarrow{v_i/x_i}](\lambda y : Y. e) = \lambda y : Y. [\overrightarrow{v_i/x_i}]e$$

Def of substitution

$$(4) \lambda y : Y. [\overrightarrow{v_i/x_i}]e \rightsquigarrow^* \lambda y : Y. [\overrightarrow{v_i/x_i}]e$$

Def of closure (first goal)

The Fundamental Lemma, 3b/5

Case \rightarrow l:

WTS $\lambda y : Y. [\overrightarrow{v_i/x_i}]e \in \text{Halt}_{Y \rightarrow Z}$

(5) Assume $t \in \text{Halt}_Y$:

(6) $t \rightsquigarrow^* v_y$

Def of Halt_Y

(7) $v_y \in \text{Halt}_Y$

Closure on (6)

(8) $(\lambda y : Y. [\overrightarrow{v_i/x_i}]e) t \rightsquigarrow^* (\lambda y : Y. [\overrightarrow{v_i/x_i}]e) v_y$

Congruence on (6)

(9) $(\lambda y : Y. [\overrightarrow{v_i/x_i}]e) v_y \rightsquigarrow [\overrightarrow{v_i/x_i}, v_y/y]e$

Reduction rule

(10) $[\overrightarrow{v_i/x_i}, v_y/y]e \in \text{Halt}_Z$

Induction

(11) $(\lambda y : Y. [\overrightarrow{v_i/x_i}]e) t \in \text{Halt}_Z$

Closure

(12) $\forall t \in \text{Halt}_Y, (\lambda y : Y. [\overrightarrow{v_i/x_i}]e) t \in \text{Halt}_Z$

(Second goal)

Case \rightarrow l:

- | | | |
|------|------------------------------------------------------------------------------------------------------------|-----------------------------------------|
| (4) | $\lambda y : Y. [\overrightarrow{v_i/x_i}]e \rightsquigarrow^* \lambda y : Y. [\overrightarrow{v_i/x_i}]e$ | First goal |
| (12) | $\forall t \in \text{Halt}_Y, (\lambda y : Y. [\overrightarrow{v_i/x_i}]e) t \in \text{Halt}_Z$ | Second goal |
| (13) | $(\lambda y : Y. [\overrightarrow{v_i/x_i}]e) \in \text{Halt}_{Y \rightarrow Z}$ | Def. of $\text{Halt}_{Y \rightarrow Z}$ |

The Fundamental Lemma, 4/5

- Case $\rightarrow E$:

$$\begin{array}{ll}
 (1) & \frac{\overrightarrow{x_i : X_i} \vdash e : Y \rightarrow Z \quad \overrightarrow{x_i : X_i} \vdash e' : Y}{\overrightarrow{x_i : X_i} \vdash e e' : Z} \rightarrow E \quad \text{Assumption} \\
 (2) & \overrightarrow{x_i : X_i} \vdash e : Y \rightarrow Z \quad \text{Subderivation} \\
 (3) & \overrightarrow{x_i : X_i} \vdash e' : Y \quad \text{Subderivation} \\
 (4) & [\overrightarrow{v_i/x_i}]e \in \text{Halt}_{Y \rightarrow Z} \quad \text{Induction} \\
 (5) & \forall t \in \text{Halt}_Y, [\overrightarrow{v_i/x_i}]e t \in \text{Halt}_Z \quad \text{Def of } \text{Halt}_{Y \rightarrow Z} \\
 (6) & [\overrightarrow{v_i/x_i}]e' \in \text{Halt}_Y \quad \text{Induction} \\
 (7) & ([\overrightarrow{v_i/x_i}]e) ([\overrightarrow{v_i/x_i}]e') \in \text{Halt}_Z \quad \text{Instantiate (5) w/ (6)} \\
 (8) & [\overrightarrow{v_i/x_i}](e e') \in \text{Halt}_Z \quad \text{Def. of substitution}
 \end{array}$$

The Fundamental Lemma, 5/5

- Case 0E:

- | | | |
|-----|-------------------------------------------------------------------------|------------------------|
| | $\overrightarrow{x_i : X_i \vdash e : 0}$ | |
| (1) | $\overrightarrow{x_i : X_i \vdash \text{abort } e : Z} \quad \text{0E}$ | Assumption |
| (2) | $\overrightarrow{x_i : X_i \vdash e : 0}$ | Subderivation |
| (3) | $\overrightarrow{[v_i/x_i]e \in \text{Halt}_0}$ | Induction |
| (4) | $\overrightarrow{[v_i/x_i]e \in \emptyset}$ | Def of Halt_0 |
| (5) | Contradiction! | |

Theorem: There are no terms $\cdot \vdash e : 0$.

Proof:

- (1) $\cdot \vdash e : 0$ Assumption
- (2) $e \in \text{Halt}_0$ Fundamental lemma
- (3) $e \in \emptyset$ Definition of Halt_0
- (4) Contradiction!

Conclusions

- Consistency and termination are very closely linked
- We have proved that the simply-typed lambda calculus is a total programming language
- Since every closed program reduces to a value, and there are no values of empty type, there are no programs of empty type
- We seem to have circumvented the Halting Theorem?
- No: we do not accept all terminating programs!

1. Extend the logical relation to support products
2. (Harder) Extend the logical relation to support sum types

Type Systems

Lecture 4: Datatypes and Polymorphism

Neel Krishnaswami
University of Cambridge

Data Types in the Simply Typed Lambda Calculus

- One of the essential features of programming languages is *data*
- So far, we have sums and product types
- This is enough to represent basic datatypes

Booleans

Builtin	Encoding
bool	$1 + 1$
true	$L \langle \rangle$
false	$R \langle \rangle$
if e then e' else e''	$\text{case}(e, L_ \rightarrow e', R_ \rightarrow e'')$

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}}$$
$$\frac{}{\Gamma \vdash \text{false} : \text{bool}}$$
$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e' : X \quad \Gamma \vdash e'' : X}{\Gamma \vdash \text{if } e \text{ then } e' \text{ else } e'' : X}$$

Builtin	Encoding
char	bool ⁷ (for ASCII!)
'A'	(true, false, false, false, false, false, true)
'B'	(true, false, false, false, false, true, false)
...	...

- This is not a wieldy encoding!
- But it works, more or less
- Example: define equality on characters

The STLC gives us:

- Representations of data
- The ability to do conditional branches on data
- The ability to do functional abstraction on operations
- **MISSING: the ability to loop**

Unbounded Recursion = Inconsistency

$$\frac{\Gamma, f : X \rightarrow Y, x : X \vdash e : Y}{\Gamma \vdash \text{fun}_{X \rightarrow Y} f x. e : X \rightarrow Y} \text{Fix}$$

$$\frac{e' \rightsquigarrow e''}{(\text{fun}_{X \rightarrow Y} f x. e) e' \rightsquigarrow (\text{fun}_{X \rightarrow Y} f x. e) e''} \qquad \frac{}{(\text{fun}_{X \rightarrow Y} f x. e) v \rightsquigarrow [\text{fun}_{X \rightarrow Y} f x. e / f, v / x] e}$$

- Modulo type inference, this is basically the typing rule Ocaml uses
- It permits defining recursive functions very naturally

The Typing of a Perfectly Fine Factorial Function

$$\begin{array}{c}
 \vdots \\
 \hline
 \Delta \vdash fact : \text{int} \rightarrow \text{int} \quad \Delta \vdash n - 1 : \text{int} \\
 \hline
 \dots \quad \Delta \vdash fact(n - 1) : \text{int} \\
 \hline
 \dots \quad \Delta \vdash n \times fact(n - 1) : \text{int} \\
 \hline
 \overbrace{\Gamma, fact : \text{int} \rightarrow \text{int}, n : \text{int}}^{\Delta} \vdash \text{if } n = 0 \text{ then } 1 \text{ else } n \times fact(n - 1) : \text{int} \\
 \hline
 \Gamma \vdash \text{fun}_{\text{int} \rightarrow \text{int}} fact\ n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times fact(n - 1) : \text{int} \rightarrow \text{int}
 \end{array}$$

A Bad Use of Recursion

$$\frac{\frac{}{f: 1 \rightarrow 0, x: 1 \vdash f: 1 \rightarrow 0} \quad \frac{}{f: 1 \rightarrow 0, x: 1 \vdash x: 1}}{f: 1 \rightarrow 0, x: 1 \vdash fx: 0}}{\cdot \vdash \text{fun}_{1 \rightarrow 0} fx. fx: 1 \rightarrow 0}$$

$$\begin{aligned} (\text{fun}_{1 \rightarrow 0} fx. fx) \langle \rangle &\leadsto [\text{fun}_{1 \rightarrow 0} fx. fx / f, \langle \rangle / x](fx) \\ &\equiv (\text{fun}_{1 \rightarrow 0} fx. fx) \langle \rangle \\ &\leadsto [\text{fun}_{1 \rightarrow 0} fx. fx / f, \langle \rangle / x](fx) \\ &\equiv (\text{fun}_{1 \rightarrow 0} fx. fx) \langle \rangle \\ &\dots \end{aligned}$$

Numbers, More Safely

$$\frac{}{\Gamma \vdash z : \mathbb{N}} \text{NI}_z$$

$$\frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash s(e) : \mathbb{N}} \text{NI}_s$$

$$\frac{\Gamma \vdash e_0 : \mathbb{N} \quad \Gamma \vdash e_1 : X \quad \Gamma, x : X \vdash e_2 : X}{\Gamma \vdash \text{iter}(e_0, z \rightarrow e_1, s(x) \rightarrow e_2) : X} \text{NE}$$

$$\frac{e_0 \rightsquigarrow e'_0}{\text{iter}(e_0, z \rightarrow e_1, s(x) \rightarrow e_2) \rightsquigarrow \text{iter}(e'_0, z \rightarrow e_1, s(x) \rightarrow e_2)}$$

$$\frac{}{\text{iter}(z, z \rightarrow e_1, s(x) \rightarrow e_2) \rightsquigarrow e_1}$$

$$\frac{}{\text{iter}(s(v), z \rightarrow e_1, s(x) \rightarrow e_2) \rightsquigarrow [\text{iter}(v, z \rightarrow e_1, s(x) \rightarrow e_2)/x]e_2}$$

Expressiveness of Gödel's T

- Iteration looks like a bounded for-loop
- It is surprisingly expressive:

$$\begin{aligned}n + m &\triangleq \text{iter}(n, z \rightarrow m, s(x) \rightarrow s(x)) \\n \times m &\triangleq \text{iter}(n, z \rightarrow z, s(x) \rightarrow m + x) \\ \text{pow}(n, m) &\triangleq \text{iter}(m, z \rightarrow s(z), s(x) \rightarrow n \times x)\end{aligned}$$

- These definitions are *primitive recursive*
- Our language is more expressive!

The Ackermann-Péter Function

$$A(0, n) = n + 1$$

$$A(m + 1, 0) = A(m, 1)$$

$$A(m + 1, n + 1) = A(m, A(m + 1, n))$$

- One of the simplest fast-growing functions
- It's not “primitive recursive” (we won't prove this)
- However, it *does* terminate
 - Either m decreases (and n can change arbitrarily), or
 - m stays the same and n decreases
 - Lexicographic argument

The Ackermann-Péter Function in Gödel's T

repeat : $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$

repeat $\triangleq \lambda f. \lambda n. \text{iter}(n, z \rightarrow \text{id}, s(r) \rightarrow f \circ r)$

ack : $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$

ack $\triangleq \lambda m. \text{iter}(m, z \rightarrow (\lambda n. s(n)), s(r) \rightarrow \lambda n. \text{repeat } r \ n \ (r1))$

- Proposition: $A(n, m) \triangleq \text{ack } n \ m$
- Note the critical use of iteration at “higher type”
- Despite totality, the calculus is extremely powerful
- Functional programmers call things like iter *recursion schemes*

$$\begin{array}{c} \frac{}{\Gamma \vdash [] : \text{list } X} \text{LISTNIL} \qquad \frac{\Gamma \vdash e : X \quad \Gamma \vdash e' : \text{list } X}{\Gamma \vdash e :: e' : \text{list } X} \text{LISTCONS} \\[2ex] \frac{\Gamma \vdash e_0 : \text{list } X \quad \Gamma \vdash e_1 : Z \quad \Gamma, x : X, r : Z \vdash e_2 : Z}{\Gamma \vdash \text{fold}(e_0, [] \rightarrow e_1, x :: r \rightarrow e_2) : Z} \text{LISTFOLD} \end{array}$$

$$\frac{e_0 \rightsquigarrow e'_0}{e_0 :: e_1 \rightsquigarrow e'_0 :: e_1}$$

$$\frac{e_1 \rightsquigarrow e'_1}{v_0 :: e_1 \rightsquigarrow v_0 :: e'_1}$$

$$\frac{e_0 \rightsquigarrow e'_0}{\text{fold}(e_0, [] \rightarrow e_1, x :: r \rightarrow e_2) \rightsquigarrow \text{fold}(e'_0, [] \rightarrow e_1, x :: r \rightarrow e_2)}$$

$$\frac{}{\text{fold}([], [] \rightarrow e_1, x :: r \rightarrow e_2) \rightsquigarrow e_1}$$

$$\frac{R \triangleq \text{fold}(v', [] \rightarrow e_1, x :: r \rightarrow e_2)}{\text{fold}(v :: v', [] \rightarrow e_1, x :: r \rightarrow e_2) \rightsquigarrow [v/x, R/r]e_2}$$

Some Functions on Lists

length : list $X \rightarrow \mathbb{N}$

length $\triangleq \lambda xs. \text{fold}(xs, [] \rightarrow z, x :: r \rightarrow s(r))$

append : list $X \rightarrow \text{list } X \rightarrow \text{list } X$

append $\triangleq \lambda x. \lambda ys. \text{fold}(xs, [] \rightarrow ys, x :: r \rightarrow x :: r)$

map : $(X \rightarrow Y) \rightarrow \text{list } X \rightarrow \text{list } Y$

map $\triangleq \lambda f. \lambda xs. \text{fold}(xs, [] \rightarrow [], x :: r \rightarrow (fx) :: r)$

A Logical Perversity

- The Curry-Howard Correspondence tells us to think of *types as propositions*
- But what logical propositions do \mathbb{N} or $\text{list } X$, correspond to?
- The following biconditionals hold:
 - $1 \iff \mathbb{N}$
 - $1 \iff \text{list } X$
 - $\mathbb{N} \iff \text{list } X$
- So \mathbb{N} is “equivalent to” truth?

A Practical Perversity

$$\begin{aligned}\text{map} & : (X \rightarrow Y) \rightarrow \text{list } X \rightarrow \text{list } Y \\ \text{map} & \triangleq \lambda f. \lambda xs. \text{fold}(xs, [], x :: r \rightarrow (fx) :: r)\end{aligned}$$

- This definition is *schematic* – it tells us how to define map for each pair of types X and Y
- However, when writing programs in the STLC+lists, we must re-define map for each function type we want to apply it at
- This is annoying, since the definition will be *identical* save for the types

The Polymorphic Lambda Calculus

Types $A ::= \alpha \mid A \rightarrow B \mid \forall \alpha. A$

Terms $e ::= x \mid \lambda x : A. e \mid e e \mid \Lambda \alpha. e \mid e A$

- We want to support *type polymorphism*
 - $\text{append} : \forall \alpha. \text{list } \alpha \rightarrow \text{list } \alpha \rightarrow \text{list } \alpha$
 - $\text{map} : \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow \text{list } \alpha \rightarrow \text{list } \beta$
- To do this, we introduce *type variables* and *type polymorphism*
- Invented (twice!) in the early 1970s
 - By the French logician Jean-Yves Girard (1972)
 - By the American computer scientist John C. Reynolds (1974)

Well-formedness of Types

Type Contexts $\Theta ::= \cdot \mid \Theta, \alpha$

$$\frac{\alpha \in \Theta}{\Theta \vdash \alpha \text{ type}}$$

$$\frac{\Theta \vdash A \text{ type} \quad \Theta \vdash B \text{ type}}{\Theta \vdash A \rightarrow B \text{ type}}$$

$$\frac{\Theta, \alpha \vdash A \text{ type}}{\Theta \vdash \forall \alpha. A \text{ type}}$$

- Judgement $\Theta \vdash A \text{ type}$ checks if a type is well-formed
- Because types can have free variables, we need to check if a type is well-scoped

Well-formedness of Term Contexts

Term Variable Contexts $\Gamma ::= \cdot \mid \Gamma, x : A$

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

- Judgement $\Theta \vdash \Gamma \text{ ctx}$ checks if a *term context* is well-formed
- We need this because contexts associate variables with types, and types now have a well-formedness condition

Typing for System F

$$\frac{x : A \in \Gamma}{\Theta; \Gamma \vdash x : A}$$

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

$$\frac{\Theta; \Gamma \vdash e : A \rightarrow B \quad \Theta; \Gamma \vdash e' : A}{\Theta; \Gamma \vdash e e' : B}$$

$$\frac{\Theta, \alpha; \Gamma \vdash e : B}{\Theta; \Gamma \vdash \Lambda \alpha. e : \forall \alpha. B}$$

$$\frac{\Theta; \Gamma \vdash e : \forall \alpha. B \quad \Theta \vdash A \text{ type}}{\Theta; \Gamma \vdash e A : \boxed{[A/\alpha]B}}$$

- Note the presence of substitution in the typing rules!

- Ultimately, we want to prove type safety for System F
- However, the introduction of type variables means that a fair amount of additional administrative overhead is introduced
- This may look intimidating on first glance, BUT really it's all just about keeping track of the free variables in types
- As a result, none of these lemmas are hard – just a little tedious

Structural Properties and Substitution for Types

1. (Type Weakening) If $\Theta, \Theta' \vdash A$ type then $\Theta, \beta, \Theta' \vdash A$ type.
2. (Type Exchange) If $\Theta, \beta, \gamma, \Theta' \vdash A$ type then $\Theta, \gamma, \beta, \Theta' \vdash A$ type
3. (Type Substitution) If $\Theta \vdash A$ type and $\Theta, \alpha \vdash B$ type then $\Theta \vdash [A/\alpha]B$ type
 - These follow the pattern in lecture 1, except with fewer cases
 - Needed to handle the type application rule

Structural Properties and Substitutions for Contexts

1. (Context Weakening) If $\Theta, \Theta' \vdash \Gamma \text{ ctx}$ then $\Theta, \alpha, \Theta' \vdash \Gamma \text{ ctx}$
2. (Context Exchange) If $\Theta, \beta, \gamma, \Theta' \vdash \Gamma \text{ ctx}$ then $\Theta, \gamma, \beta, \Theta' \vdash \Gamma \text{ ctx}$
3. (Context Substitution) If $\Theta \vdash A \text{ type}$ and $\Theta, \alpha \vdash \Gamma \text{ type}$ then $\Theta \vdash [A/\alpha]\Gamma \text{ type}$
 - This just lifts the type-level structural properties to contexts

Regularity: If $\Theta \vdash \Gamma$ ctx and $\Theta; \Gamma \vdash e : A$ then $\Theta \vdash A$ type

Proof: By induction on the derivation of $\Theta; \Gamma \vdash e : A$

- This just says if typechecking succeeds, then it found a well-formed type

Structural Properties and Substitution of Types into Terms

- (Type Weakening of Terms) If $\Theta, \Theta' \vdash \Gamma \text{ ctx}$ and $\Theta, \Theta'; \Gamma \vdash e : A$ then $\Theta, \alpha, \Theta'; \Gamma \vdash e : A$.
- (Type Exchange of Terms) If $\Theta, \alpha, \beta, \Theta' \vdash \Gamma \text{ ctx}$ and $\Theta, \alpha, \beta, \Theta'; \Gamma \vdash e : A$ then $\Theta, \beta, \alpha, \Theta'; \Gamma \vdash e : A$.
- (Type Substitution of Terms) If $\Theta, \alpha \vdash \Gamma \text{ ctx}$ and $\Theta \vdash A \text{ type}$ and $\Theta, \alpha; \Gamma \vdash e : B$ then $\Theta; [A/\alpha]\Gamma \vdash [A/\alpha]e : [A/\alpha]B$.

Structural Properties and Substitution for Term Variables

- (Weakening of Terms) If $\Theta \vdash \Gamma, \Gamma' \text{ ctx}$ and $\Theta \vdash B \text{ type}$ and $\Theta; \Gamma, \Gamma' \vdash e : A$ then $\Theta; \Gamma, y : B, \Gamma' \vdash e : A$
- (Exchange of Terms) If $\Theta \vdash \Gamma, y : B, z : C, \Gamma' \text{ ctx}$ and $\Theta; \Gamma, y : B, z : C, \Gamma' \vdash e : A$, then $\Theta; \Gamma, z : C, y : B, \Gamma' \vdash e : A$
- (Substitution of Terms) If $\Theta \vdash \Gamma, x : A \text{ ctx}$ and $\Theta; \Gamma \vdash e : A$ and $\Theta; \Gamma, x : A \vdash e' : B$ then $\Theta; \Gamma \vdash [e/x]e' : B$.
- There are two sets of substitution theorems, since there are two contexts
- We also need to assume well-formedness conditions
- But the proofs are all otherwise similar

- We have seen how data works in the pure lambda calculus
- We have started to make it more useful with polymorphism
- But where did the data go in System F? (Next lecture!)

Type Systems

Lecture 5: System F and Church Encodings

Neel Krishnaswami
University of Cambridge

System F, The Girard-Reynolds Polymorphic Lambda Calculus

Types $A ::= \alpha \mid A \rightarrow B \mid \forall \alpha. A$

Terms $e ::= x \mid \lambda x : A. e \mid e e \mid \Lambda \alpha. e \mid e A$

Type Contexts $\Theta ::= \cdot \mid \Theta, \alpha$

Term Contexts $\Gamma ::= \cdot \mid \Gamma, x : A$

Judgement	Notation
Well-formedness of types	$\Theta \vdash A \text{ type}$
Well-formedness of term contexts	$\Theta \vdash \Gamma \text{ ctx}$
Term typing	$\Theta; \Gamma \vdash e : A$

Well-formedness of Types

$$\frac{\alpha \in \Theta}{\Theta \vdash \alpha \text{ type}}$$

$$\frac{\Theta \vdash A \text{ type} \quad \Theta \vdash B \text{ type}}{\Theta \vdash A \rightarrow B \text{ type}}$$

$$\frac{\Theta, \alpha \vdash A \text{ type}}{\Theta \vdash \forall \alpha. A \text{ type}}$$

- Judgement $\Theta \vdash A \text{ type}$ checks if a type is well-formed
- Because types can have free variables, we need to check if a type is well-scoped

Well-formedness of Term Contexts

Term Variable Contexts $\Gamma ::= \cdot \mid \Gamma, x : A$

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

- Judgement $\Theta \vdash \Gamma$ type checks if a *term context* is well-formed
- We need this because contexts associate variables with types, and types now have a well-formedness condition

Typing for System F

$$\frac{x : A \in \Gamma}{\Theta; \Gamma \vdash x : A}$$

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

$$\frac{\Theta; \Gamma \vdash e : A \rightarrow B \quad \Theta; \Gamma \vdash e' : A}{\Theta; \Gamma \vdash e e' : B}$$

$$\frac{\Theta, \alpha; \Gamma \vdash e : B}{\Theta; \Gamma \vdash \Lambda \alpha. e : \forall \alpha. B}$$

$$\frac{\Theta; \Gamma \vdash e : \forall \alpha. B \quad \Theta \vdash A \text{ type}}{\Theta; \Gamma \vdash e A : \boxed{[A/\alpha]B}}$$

- Note the presence of substitution in the typing rules!

Values $v ::= \lambda x : A. e \mid \Lambda \alpha. e$

$$\frac{e_0 \rightsquigarrow e'_0}{e_0 e_1 \rightsquigarrow e'_0 e_1} \text{ CONGFUN}$$

$$\frac{e_1 \rightsquigarrow e'_1}{v_0 e_1 \rightsquigarrow v_0 e'_1} \text{ CONGFUNARG}$$

$$\frac{}{(\lambda x : A. e) v \rightsquigarrow [v/x]e} \text{ FUNEVAL}$$

$$\frac{e \rightsquigarrow e'}{e A \rightsquigarrow e' A} \text{ CONGFORALL}$$

$$\frac{}{(\Lambda \alpha. e) A \rightsquigarrow [A/\alpha]e} \text{ FORALLEVAL}$$

The Bookkeeping

- Ultimately, we want to prove type safety for System F
- However, the introduction of type variables means that a fair amount of additional administrative overhead is introduced
- This may look intimidating on first glance, BUT really it's all just about keeping track of the free variables in types
- As a result, none of these lemmas are hard – just a little tedious

Structural Properties and Substitution for Types

1. (Type Weakening) If $\Theta, \Theta' \vdash A$ type then $\Theta, \beta, \Theta' \vdash A$ type.
2. (Type Exchange) If $\Theta, \beta, \gamma, \Theta' \vdash A$ type then $\Theta, \gamma, \beta, \Theta' \vdash A$ type
3. (Type Substitution) If $\Theta \vdash A$ type and $\Theta, \alpha \vdash B$ type then $\Theta \vdash [A/\alpha]B$ type
 - These follow the pattern in lecture 1, except with fewer cases
 - Needed to handle the type application rule

Structural Properties and Substitutions for Contexts

1. (Context Weakening) If $\Theta, \Theta' \vdash \Gamma \text{ ctx}$ then $\Theta, \alpha, \Theta' \vdash \Gamma \text{ ctx}$
2. (Context Exchange) If $\Theta, \beta, \gamma, \Theta' \vdash \Gamma \text{ ctx}$ then $\Theta, \gamma, \beta, \Theta' \vdash \Gamma \text{ ctx}$
3. (Context Substitution) If $\Theta \vdash A \text{ type}$ and $\Theta, \alpha \vdash \Gamma \text{ type}$ then $\Theta \vdash [A/\alpha]\Gamma \text{ type}$
 - This just lifts the type-level structural properties to contexts
 - Proof via induction on derivations of $\Theta \vdash \Gamma \text{ ctx}$

Regularity: If $\Theta \vdash \Gamma$ ctx and $\Theta; \Gamma \vdash e : A$ then $\Theta \vdash A$ type

Proof: By induction on the derivation of $\Theta; \Gamma \vdash e : A$

- This just says if typechecking succeeds, then it found a well-formed type

Structural Properties and Substitution of Types into Terms

- (Type Weakening of Terms) If $\Theta, \Theta' \vdash \Gamma \text{ ctx}$ and $\Theta, \Theta'; \Gamma \vdash e : A$ then $\Theta, \alpha, \Theta'; \Gamma \vdash e : A$.
- (Type Exchange of Terms) If $\Theta, \alpha, \beta, \Theta' \vdash \Gamma \text{ ctx}$ and $\Theta, \alpha, \beta, \Theta'; \Gamma \vdash e : A$ then $\Theta, \beta, \alpha, \Theta'; \Gamma \vdash e : A$.
- (Type Substitution of Terms) If $\Theta, \alpha \vdash \Gamma \text{ ctx}$ and $\Theta \vdash A \text{ type}$ and $\Theta, \alpha; \Gamma \vdash e : B$ then $\Theta; [A/\alpha]\Gamma \vdash [A/\alpha]e : [A/\alpha]B$.

Structural Properties and Substitution for Term Variables

- (Weakening for Terms) If $\Theta \vdash \Gamma, \Gamma' \text{ ctx}$ and $\Theta \vdash B \text{ type}$ and $\Theta; \Gamma, \Gamma' \vdash e : A$ then $\Theta; \Gamma, y : B, \Gamma' \vdash e : A$
- (Exchange for Terms) If $\Theta \vdash \Gamma, y : B, z : C, \Gamma' \text{ ctx}$ and $\Theta; \Gamma, y : B, z : C, \Gamma' \vdash e : A$, then $\Theta; \Gamma, z : C, y : B, \Gamma' \vdash e : A$
- (Substitution of Terms) If $\Theta \vdash \Gamma, x : A \text{ ctx}$ and $\Theta; \Gamma \vdash e : A$ and $\Theta; \Gamma, x : A \vdash e' : B$ then $\Theta; \Gamma \vdash [e/x]e' : B$.

- There are two sets of substitution theorems, since there are two contexts
- We also need to assume well-formedness conditions
- But proofs are all otherwise similar to the simply-typed case

Progress: If $\cdot; \cdot \vdash e : A$ then either e is a value or $e \rightsquigarrow e'$.

Type preservation: If $\cdot; \cdot \vdash e : A$ and $e \rightsquigarrow e'$ then $\cdot; \cdot \vdash e' : A$.

Progress: Big Lambdas

Proof by induction on derivations:

$$\frac{\overbrace{.; \cdot \vdash e : \forall \alpha. B}^{(2)} \quad \overbrace{\cdot \vdash A \text{ type}}^{(3)}}{\cdot; \cdot \vdash e A : [A/\alpha]B}$$

(1) $\cdot; \cdot \vdash e A : [A/\alpha]B$

Assumption

(4) $e \rightsquigarrow e'$ or e is a value

Induction on (2)

Case on (4)

(5) Case $e \rightsquigarrow e'$:

(6) $e A \rightsquigarrow e' A$

by CONGFORALL on (5)

(7) Case e is a value:

(8) $e = \Lambda \alpha. e'$

By canonical forms on (2)

(9) $(\Lambda \alpha. e') A \rightsquigarrow [A/\alpha]e$

By FORALLEVAL

Preservation: Big Lambdas

By induction on the derivation of $e \rightsquigarrow e'$:

$$(1) \quad \frac{}{(\Lambda\alpha. e) A \rightsquigarrow [A/\alpha]e} \text{FORALLEVAL} \quad \text{Assumption}$$

$$(2) \quad \frac{\overbrace{\alpha; \cdot \vdash e : B}^{(3)} \quad \overbrace{\cdot \vdash A \text{ type}}^{(4)}}{\cdot; \cdot \vdash \Lambda\alpha. e : \forall\alpha. B} \quad \text{Assumption}$$

$$(5) \quad \cdot; \cdot \vdash [A/\alpha]e : [A/\alpha]B \quad \text{Type subst. on (3), (4)}$$

Church Encodings: Representing Data with Functions

- System has the types $\forall\alpha. A$ and $A \rightarrow B$
- No booleans, sums, numbers, tuples or anything else
- Seemingly, there is no data in this calculus
- Surprisingly, it is unnecessary!
- Discovered in 1941 by Alonzo Church
- The idea:
 1. Data is used to make choices
 2. Based on the choice, you perform different results
 3. So we can encode data as functions which take different possible results, and return the right one

Church Encodings: Booleans

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \quad \frac{}{\Gamma \vdash \text{false} : \text{bool}} \quad \frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e' : X \quad \Gamma \vdash e'' : X}{\Gamma \vdash \text{if } e \text{ then } e' \text{ else } e'' : X}$$

- Boolean type has two values, true and false
- Conditional switches between two X 's based on e 's value

Type		Encoding
bool	\triangleq	$\forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha$
True	\triangleq	$\Lambda \alpha. \lambda x : \alpha. \lambda y : \alpha. x$
False	\triangleq	$\Lambda \alpha. \lambda x : \alpha. \lambda y : \alpha. y$
if e then e' else $e'' : X$	\triangleq	$e X e' e''$

Evaluating Church conditionals

$$\begin{aligned}\text{if true then } e' \text{ else } e'' : A &= \text{true } A \ e' \ e'' \\ &= (\Lambda \alpha. \lambda x : \alpha. \lambda y : \alpha. x) A \ e' \ e'' \\ &= (\lambda x : A. \lambda y : A. x) \ e' \ e'' \\ &= (\lambda y : A. e') \ e'' \\ &= e'\end{aligned}$$

$$\begin{aligned}\text{if false then } e' \text{ else } e'' : A &= \text{false } A \ e' \ e'' \\ &= (\Lambda \alpha. \lambda x : \alpha. \lambda y : \alpha. y) A \ e' \ e'' \\ &= (\lambda x : A. \lambda y : A. y) \ e' \ e'' \\ &= (\lambda y : A. y) \ e'' \\ &= e''\end{aligned}$$

Type		Encoding
$X \times Y$	\triangleq	$\forall \alpha. (X \rightarrow Y \rightarrow \alpha) \rightarrow \alpha$
$\langle e, e' \rangle$	\triangleq	$\Lambda \alpha. \lambda k : X \rightarrow Y \rightarrow \alpha. k e e'$
$\text{fst } e$	\triangleq	$e X (\lambda x : X. \lambda y : Y. x)$
$\text{snd } e$	\triangleq	$e Y (\lambda x : X. \lambda y : Y. y)$

Evaluating Church Pairs

$$\begin{aligned}\text{fst } \langle e, e' \rangle &= \langle e, e' \rangle X (\lambda x : X. \lambda y : Y. x) \\ &= (\Lambda \alpha. \lambda k : X \rightarrow Y \rightarrow \alpha. k e e') X (\lambda x : X. \lambda y : Y. x) \\ &= (\lambda k : X \rightarrow Y \rightarrow X. k e e') (\lambda x : X. \lambda y : Y. x) \\ &= (\lambda x : X. \lambda y : Y. x) e e' \\ &= (\lambda y : Y. e) e' \\ &= e\end{aligned}$$

$$\begin{aligned}\text{snd } \langle e, e' \rangle &= \langle e, e' \rangle Y (\lambda x : X. \lambda y : Y. y) \\ &= (\Lambda \alpha. \lambda k : X \rightarrow Y \rightarrow \alpha. k e e') Y (\lambda x : X. \lambda y : Y. y) \\ &= (\lambda k : X \rightarrow Y \rightarrow Y. k e e') (\lambda x : X. \lambda y : Y. y) \\ &= (\lambda x : X. \lambda y : Y. y) e e' \\ &= (\lambda y : Y. y) e' \\ &= e'\end{aligned}$$

Type	Encoding
$X + Y$	$\forall \alpha. (X \rightarrow \alpha) \rightarrow (Y \rightarrow \alpha) \rightarrow \alpha$
$L e$	$\Lambda \alpha. \lambda f : X \rightarrow \alpha. \lambda g : Y \rightarrow \alpha. f e$
$R e$	$\Lambda \alpha. \lambda f : X \rightarrow \alpha. \lambda g : Y \rightarrow \alpha. g e$
$\text{case}(e, Lx \rightarrow e_1, Ry \rightarrow e_2) : Z$	$e Z (\lambda x : X. e_1) (\lambda y : Y. e_2)$

$$\begin{aligned} & \text{case}(\text{L } e, \text{L } x \rightarrow e_1, \text{R } y \rightarrow e_2) : Z \\ &= (\text{L } e) Z (\lambda x : X. e_1) (\lambda y : Y. e_2) \\ &= (\Lambda \alpha. \lambda f : X \rightarrow \alpha. \lambda g : Y \rightarrow \alpha. f e) \\ &\quad Z (\lambda x : X. e_1) (\lambda y : Y. e_2) \\ &= (\lambda f : X. \lambda g : Y. f e) \\ &\quad (\lambda x : X. e_1) (\lambda y : Y. e_2) \\ &= (\lambda g : Y. (\lambda x : X. e_1) e) \\ &\quad (\lambda y : Y. e_2) \\ &= (\lambda x : X. e_1) e \\ &= [e/x]e_1 \end{aligned}$$

Church Encodings: Natural Numbers

Type	Encoding
\mathbb{N}	$\forall \alpha. \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha$
z	$\Lambda \alpha. \lambda Z : \alpha. \lambda S : \alpha \rightarrow \alpha. Z$
$s(e)$	$\Lambda \alpha. \lambda Z : \alpha. \lambda S : \alpha \rightarrow \alpha. S (e \alpha Z S)$
$\text{iter}(e, z \rightarrow e_z, s(x) \rightarrow e_s) : X$	$e X e_z (\lambda x : X. e_s)$

$$\begin{aligned} & \text{iter}(z, z \rightarrow e_z, s(x) \rightarrow e_s) \\ &= z X e_z (\lambda x : X. e_s) \\ &= (\Lambda \alpha. \lambda z : \alpha. \lambda s : \alpha \rightarrow \alpha. z) X e_z (\lambda x : X. e_s) \\ &= (\lambda z : X. \lambda s : X \rightarrow X. z) e_z (\lambda x : X. e_s) \\ &= (\lambda s : X \rightarrow X. e_z) (\lambda x : X. e_s) \\ &= e_z \end{aligned}$$

$$\begin{aligned} & \text{iter}(s(e), z \rightarrow e_z, s(x) \rightarrow e_s) \\ &= (s(e)) X e_z (\lambda x : X. e_s) \\ &= (\Lambda \alpha. \lambda z : \alpha. \lambda s : \alpha \rightarrow \alpha. s (e \alpha z s)) X e_z (\lambda x : X. e_s) \\ &= (\lambda z : X. \lambda s : X \rightarrow X. s (e X z s)) e_z (\lambda x : X. e_s) \\ &= (\lambda s : X \rightarrow X. s (e X e_z s)) (\lambda x : X. e_s) \\ &= (\lambda x : X. e_s) (e X e_z (\lambda x : X. e_s))) \\ &= (\lambda x : X. e_s) \text{iter}(e, z \rightarrow e_z, s(x) \rightarrow e_s) \\ &= [\text{iter}(e, z \rightarrow e_z, s(x) \rightarrow e_s) / x] e_s \end{aligned}$$

Type	Encoding
$\text{list } X$	$\forall \alpha. \alpha \rightarrow (X \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha$
$[]$	$\Lambda \alpha. \lambda n : \alpha. \lambda c : X \rightarrow \alpha \rightarrow \alpha. n$
$e :: e'$	$\Lambda \alpha. \lambda n : \alpha. \lambda c : X \rightarrow \alpha \rightarrow \alpha. c \ e \ (e' \ \alpha \ n \ c)$

$\text{fold}(e, [] \rightarrow e_n, x :: r \rightarrow e_c) : Z = e \ Z \ e_n \ (\lambda x : X. \lambda r : Z. e_c)$

- System F is very simple, and very expressive
- Formal basis of polymorphism in ML, Java, Haskell, etc.
- Surprise: from polymorphism and functions, data is definable

1. Prove the regularity lemma.
2. Define a Church encoding for the unit type.
3. Define a Church encoding for the empty type.
4. Define a Church encoding for binary trees, corresponding to the ML datatype
`type tree = Leaf | Node of tree * X * tree.`

Type Systems

Lecture 6: Existentials, Data Abstraction, and Termination for System F

Neel Krishnaswami
University of Cambridge

Polymorphism and Data Abstraction

- So far, we have used polymorphism to model datatypes and genericity
- Reynolds's original motivation was to model *data abstraction*

An ML Module Signature

```
module type BOOL = sig
  type t
  val yes : t
  val no : t
  val choose
    : t -> 'a -> 'a -> 'a
end
```

- We introduce an abstract type `t`
- There are two values, `yes` and `no` of type `t`
- There is an operation `choose`, which takes a `t` and two values, and switches between them.

An Implementation

```
module M1 : BOOL = struct
  type t = unit option
  let yes = Some ()
  let no = None
  let choose v ifyes ifno =
    match v with
    | Some () -> ifyes
    | None -> ifno
end
```

- Implementation uses option type over unit
- There are two values, one for true and one for false
- **choose** implemented via pattern matching

Another Implementation

```
module M2 : BOOL = struct
  type t = int
  let yes = 1
  let no = 0
  let choose b ifyes ifno =
    if b = 1 then
      ifyes
    else
      ifno
end
```

- Implement booleans with integers
- Use 1 for true, 0 for false
- Why is this okay? (Many more integers than booleans, after all)

Yet Another Implementation

```
module M3 : BOOL = struct
  type t =
    {f : 'a. 'a -> 'a -> 'a}
  let yes =
    {f = fun a b -> a}
  let no =
    {f = fun a b -> b}
  let choose b ifyes ifno =
    b.f ifyes ifno
end
```

- Implement booleans with Church encoding (plus some Ocaml hacks)
- Is this really the same type as in the previous lecture?

A Common Pattern

- We have a signature — **BOOL** — with an abstract type in it
- We choose a concrete implementation of that abstract type
- We implement the other operations (**yes**, **no**, **choose**) of the interface in terms of that concrete representation
- Client code cannot identify the representation type because it sees an abstract type variable **t** rather than the representation

Abstract Data Types in System F

Types $A ::= \dots \mid \exists \alpha. A$

Terms $e ::= \dots \mid \text{pack}_{\alpha.B}(A, e) \mid \text{let pack}(\alpha, x) = e \text{ in } e'$

Values $v ::= \text{pack}_{\alpha.B}(A, v)$

$$\frac{\Theta, \alpha \vdash B \text{ type} \quad \Theta \vdash A \text{ type} \quad \Theta; \Gamma \vdash e : [A/\alpha]B}{\Theta; \Gamma \vdash \text{pack}_{\alpha.B}(A, e) : \exists \alpha. B} \exists I$$

$$\frac{\Theta; \Gamma \vdash e : \exists \alpha. A \quad \Theta, \alpha; \Gamma, x : A \vdash e' : C \quad \Theta \vdash C \text{ type}}{\Theta; \Gamma \vdash \text{let pack}(\alpha, x) = e \text{ in } e' : C} \exists E$$

Operational Semantics for Abstract Types

$$\frac{e \rightsquigarrow e'}{\text{pack}_{\alpha.B}(A, e) \rightsquigarrow \text{pack}_{\alpha.B}(A, e')}$$

$$\frac{e \rightsquigarrow e'}{\text{let pack}(\alpha, x) = e \text{ in } t \rightsquigarrow \text{let pack}(\alpha, x) = e' \text{ in } t}$$

$$\frac{}{\text{let pack}(\alpha, x) = \text{pack}_{\alpha.B}(A, v) \text{ in } e \rightsquigarrow [A/\alpha, v/x]e}$$

Data Abstraction in System F

- We have a signature with an abstract type in it
- We choose a concrete implementation of that abstract type
- We implement the operations of the interface in terms of the concrete representation
- Client code sees an abstract type variable α rather than the representation

$$\frac{\Theta, \alpha \vdash B \text{ type} \quad \Theta \vdash A \text{ type} \quad \Theta; \Gamma \vdash e : [A/\alpha]B}{\Theta; \Gamma \vdash \text{pack}_{\alpha.B}(A, e) : \exists \alpha. B} \exists I$$
$$\frac{\Theta; \Gamma \vdash e : \exists \alpha. A \quad \Theta, \alpha; \Gamma, x : A \vdash e' : C \quad \Theta \vdash C \text{ type}}{\Theta; \Gamma \vdash \text{let pack}(\alpha, x) = e \text{ in } e' : C} \exists E$$

Abstract Types Have Existential Type

- No accident we write $\exists \alpha. B$ for abstract types!
- This is exactly the same thing as existential quantification in second-order logic
- Discovered by Mitchell and Plotkin in 1988 – *Abstract Types Have Existential Type*
- But Reynolds was thinking about data abstraction in 1976...?

A Church Encoding for Existential Types

$$\frac{\Theta, \alpha \vdash B \text{ type} \quad \Theta \vdash A \text{ type} \quad \Theta; \Gamma \vdash e : [A/\alpha]B}{\Theta; \Gamma \vdash \text{pack}_{\alpha.B}(A, e) : \exists \alpha. B} \exists I$$

$$\frac{\Theta; \Gamma \vdash e : \exists \alpha. B \quad \Theta, \alpha; \Gamma, x : B \vdash e' : C \quad \Theta \vdash C \text{ type}}{\Theta; \Gamma \vdash \text{let pack}(\alpha, x) = e \text{ in } e' : C} \exists E$$

Original	Encoding
$\exists \alpha. B$	$\forall \beta. (\forall \alpha. B \rightarrow \beta) \rightarrow \beta$
$\text{pack}_{\alpha.B}(A, e)$	$\Lambda \beta. \lambda k : \forall \alpha. B \rightarrow \beta. k A e$
$\text{let pack}(\alpha, x) = e \text{ in } e' : C$	$e C (\Lambda \alpha. \lambda x : B. e')$

$$\begin{aligned} \text{let pack}(\alpha, x) &= \text{pack}_{\alpha.B}(A, e) \text{ in } e' : C \\ &= \text{pack}_{\alpha.B}(A, e) C \ (\Lambda\alpha. \lambda x : B. e') \\ &= (\Lambda\beta. \lambda k : \forall\alpha. B \rightarrow \beta. k A e) C \ (\Lambda\alpha. \lambda x : B. e') \\ &= (\lambda k : \forall\alpha. B \rightarrow C. k A e) (\Lambda\alpha. \lambda x : B. e') \\ &= (\Lambda\alpha. \lambda x : B. e') A e \\ &= (\lambda x : [A/\alpha]B. [A/\alpha]e') e \\ &= [e/x][A/\alpha]e' \end{aligned}$$

System F, The Girard-Reynolds Polymorphic Lambda Calculus

Types $A ::= \alpha \mid A \rightarrow B \mid \forall \alpha. A$

Terms $e ::= x \mid \lambda x : A. e \mid e e \mid \Lambda \alpha. e \mid e A$

Values $v ::= \lambda x : A. e \mid \Lambda \alpha. e$

$$\frac{e_0 \rightsquigarrow e'_0}{e_0 e_1 \rightsquigarrow e'_0 e_1} \text{ CONGFUN}$$

$$\frac{e_1 \rightsquigarrow e'_1}{v_0 e_1 \rightsquigarrow v_0 e'_1} \text{ CONGFUNARG}$$

$$\frac{}{(\lambda x : A. e) v \rightsquigarrow [v/x]e} \text{ FUNEVAL}$$

$$\frac{e \rightsquigarrow e'}{e A \rightsquigarrow e' A} \text{ CONGFORALL}$$

$$\frac{}{(\Lambda \alpha. e) A \rightsquigarrow [A/\alpha]e} \text{ FORALLEVAL}$$

So far:

1. We have seen System F and its basic properties
2. Sketched a proof of type safety
3. Saw that a variety of datatypes were encodable in it
4. We saw that even data abstraction was representable in it
5. We asserted, but did not prove, termination

- We proved termination for the STLC by defining a *logical relation*
 - This was a family of relations
 - Relations defined by recursion on the structure of the type
 - Enforced a “hereditary termination” property
- Can we define a logical relation for System F?
 - How do we handle free type variables? (i.e., what’s the interpretation of α ?)
 - How do we handle quantifiers? (i.e., what’s the interpretation of $\forall\alpha. A$?)

Semantic Types

A *semantic type* is a set of closed terms X such that:

- (Halting) If $e \in X$, then e halts (i.e. $e \rightsquigarrow^* v$ for some v).
- (Closure) If $e \rightsquigarrow e'$, then $e' \in X$ iff $e \in X$.

Idea:

- Build generic properties of the logical relation into the definition of a type.
- Use this to interpret variables!

Semantic Type Interpretations

$$\frac{\alpha \in \Theta}{\Theta \vdash \alpha \text{ type}}$$

$$\frac{\Theta \vdash A \text{ type} \quad \Theta \vdash B \text{ type}}{\Theta \vdash A \rightarrow B \text{ type}}$$

$$\frac{\Theta, \alpha \vdash A \text{ type}}{\Theta \vdash \forall \alpha. A \text{ type}}$$

- We can interpret *type well-formedness derivations*
- Given a type variable context Θ , we will define a variable interpretation θ as a map from $\text{dom}(\Theta)$ to semantic types.
- Given a variable interpretation θ , we write $(\theta, X/\alpha)$ to mean extending θ with an interpretation X for a variable α .

Interpretation of Types

$\llbracket - \rrbracket \in \text{WellFormedType} \rightarrow \text{VarInterpretation} \rightarrow \text{SemanticType}$

$$\llbracket \Theta \vdash \alpha \text{ type} \rrbracket \theta = \theta(\alpha)$$

$$\llbracket \Theta \vdash A \rightarrow B \text{ type} \rrbracket \theta = \left\{ e \left| \begin{array}{l} e \text{ halts} \wedge \\ \forall e' \in \llbracket \Theta \vdash A \text{ type} \rrbracket \theta. \\ (e e') \in \llbracket \Theta \vdash B \text{ type} \rrbracket \theta \end{array} \right. \right\}$$

$$\llbracket \Theta \vdash \forall \alpha. B \text{ type} \rrbracket \theta = \left\{ e \left| \begin{array}{l} e \text{ halts} \wedge \\ \forall A \in \text{type}, X \in \text{SemType}. \\ (e A) \in \llbracket \Theta, \alpha \vdash B \text{ type} \rrbracket (\theta, X/\alpha) \end{array} \right. \right\}$$

Note the *lack* of a link between A and X in the $\forall \alpha. B$ case

Properties of the Interpretation

- **Closure:** If θ is an interpretation for Θ , then $\llbracket \Theta \vdash A \text{ type} \rrbracket \theta$ is a semantic type.
- **Exchange:** $\llbracket \Theta, \alpha, \beta, \Theta' \vdash A \text{ type} \rrbracket = \llbracket \Theta, \beta, \alpha, \Theta' \vdash A \text{ type} \rrbracket$
- **Weakening:** If $\Theta \vdash A \text{ type}$, then $\llbracket \Theta, \alpha \vdash A \text{ type} \rrbracket (\theta, X/\alpha) = \llbracket \Theta \vdash A \text{ type} \rrbracket \theta$.
- **Substitution:** If $\Theta \vdash A \text{ type}$ and $\Theta, \alpha \vdash B \text{ type}$ then
 $\llbracket \Theta \vdash [A/\alpha]B \text{ type} \rrbracket \theta = \llbracket \Theta, \alpha \vdash B \text{ type} \rrbracket (\theta, \llbracket \Theta \vdash A \text{ type} \rrbracket \theta/\alpha)$

Each property is proved by induction on a type well-formedness derivation.

Closure: (one half of the) \forall Case

Closure: If θ interprets Θ , then $\llbracket \Theta \vdash \forall \alpha. A \text{ type} \rrbracket \theta$ is a type.

Suffices to show: if $e \rightsquigarrow e'$, then $e \in \llbracket \Theta \vdash \forall \alpha. A \text{ type} \rrbracket \theta$ iff $e' \in \llbracket \Theta \vdash \forall \alpha. A \text{ type} \rrbracket \theta$.

- | | | |
|---|----------------------------------------------------------------------------------------------------------|------------------|
| 0 | $e \rightsquigarrow e'$ | Assumption |
| 1 | $e' \in \llbracket \Theta \vdash \forall \alpha. A \text{ type} \rrbracket \theta$ | Assumption |
| 2 | $\forall (C, X). e' C \in \llbracket \Theta, \alpha \vdash A \text{ type} \rrbracket (\theta, X/\alpha)$ | Def. |
| 3 | Fix arbitrary (C, X) | |
| 4 | $e' C \in \llbracket \Theta, \alpha \vdash A \text{ type} \rrbracket (\theta, X/\alpha)$ | By 2 |
| 5 | $e C \rightsquigarrow e' C$ | CONGFORALL on 0 |
| 6 | $e C \in \llbracket \Theta, \alpha \vdash A \text{ type} \rrbracket (\theta, X/\alpha)$ | Induction on 4,5 |
| 7 | $\forall (C, X). e C \in \llbracket \Theta, \alpha \vdash A \text{ type} \rrbracket (\theta, X/\alpha)$ | |
| 8 | $e \in \llbracket \Theta \vdash \forall \alpha. A \text{ type} \rrbracket \theta$ | From 7 |

Substitution: (one half of) the \forall case

$$\llbracket \Theta, \alpha \vdash \forall \beta. B \text{ type} \rrbracket (\theta, \llbracket \Theta \vdash A \text{ type} \rrbracket \theta) = \llbracket \Theta \vdash [A/\alpha](\forall \beta. B) \text{ type} \rrbracket \theta$$

1. We assume $e \in \llbracket \Theta, \alpha \vdash \forall \beta. B \text{ type} \rrbracket (\theta, \llbracket \Theta \vdash A \text{ type} \rrbracket \theta)$
2. We want to show: $e \in \llbracket \Theta \vdash [A/\alpha](\forall \beta. B) \text{ type} \rrbracket \theta$.
3. Expanding the definition of 1:
 $\forall (C, X). e C \in \llbracket \Theta, \alpha, \beta \vdash B \text{ type} \rrbracket (\theta, \llbracket \Theta \vdash A \text{ type} \rrbracket \theta, X/\beta)$.
4. For 2, it suffices to show: $\forall (C, X). e C \in \llbracket \Theta, \beta \vdash [A/\alpha](B) \text{ type} \rrbracket (\theta, X/\beta)$.
 - Fix (C, X)
 - So $e C \in \llbracket \Theta, \alpha, \beta \vdash B \text{ type} \rrbracket (\theta, \llbracket \Theta \vdash A \text{ type} \rrbracket \theta, X/\beta)$
 - Exchange: $e C \in \llbracket \Theta, \beta, \alpha \vdash B \text{ type} \rrbracket (\theta, X/\beta, \llbracket \Theta \vdash A \text{ type} \rrbracket \theta)$
 - Weaken: $e C \in \llbracket \Theta, \beta, \alpha \vdash B \text{ type} \rrbracket (\theta, X/\beta, \llbracket \Theta, \beta \vdash A \text{ type} \rrbracket (\theta, X/\beta))$
 - Induction: $e C \in \llbracket \Theta, \beta \vdash [A/\alpha]B \text{ type} \rrbracket (\theta, X/\beta)$

The Fundamental Lemma

If we have that

- $\overbrace{\alpha_1, \dots, \alpha_k}^{\Theta}; \overbrace{x_1 : A_1, \dots, x_n : A_n}^{\Gamma} \vdash e : B$
- $\Theta \vdash \Gamma \text{ ctx}$
- θ interprets Θ
- For each $x_i : A_i \in \Gamma$, we have $e_i \in \llbracket \Theta \vdash A_i \text{ type} \rrbracket \theta$

Then it follows that:

- $[C_1/\alpha_1, \dots, C_k/\alpha_k][e_1/x_1, \dots, e_n/x_n]e \in \llbracket \Theta \vdash B \text{ type} \rrbracket \theta$

1. Prove the other direction of the closure property for the $\Theta \vdash \forall\alpha. A$ type case.
2. Prove the other direction of the substitution property for the $\Theta \vdash \forall\alpha. A$ type case.
3. Prove the fundamental lemma for the forall-introduction case $\Theta; \Gamma \vdash \Lambda\alpha. e : \forall\alpha. A$.

Type Systems

Lecture 7: Programming with Effects

Neel Krishnaswami
University of Cambridge

Wrapping up Polymorphism

System F is Explicit

We saw that in System F has explicit type abstraction and application:

$$\frac{\Theta, \alpha; \Gamma \vdash e : B}{\Theta; \Gamma \vdash \lambda\alpha. e : \forall\alpha. B} \qquad \frac{\Theta; \Gamma \vdash e : \forall\alpha. B \quad \Theta \vdash A \text{ type}}{\Theta; \Gamma \vdash eA : [A/\alpha]B}$$

This is fine in theory, but what do programs look like in practice?

System F is Very, Very Explicit

Suppose we have a map functional and an isEven function:

$$\begin{aligned} \text{map} & : \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow \text{list } \alpha \rightarrow \text{list } \beta \\ \text{isEven} & : \mathbb{N} \rightarrow \text{bool} \end{aligned}$$

A function taking a list of numbers and applying isEven to it:

$$\text{map } \mathbb{N} \text{ bool } \text{isEven} : \text{list } \mathbb{N} \rightarrow \text{list bool}$$

If you have a list of lists of natural numbers:

$$\begin{aligned} & \text{map } (\text{list } \mathbb{N}) (\text{list bool}) (\text{map } \mathbb{N} \text{ bool } \text{isEven}) \\ & : \text{list } (\text{list } \mathbb{N}) \rightarrow \text{list } (\text{list bool}) \end{aligned}$$

The type arguments overwhelm everything else!

Type Inference

- Luckily, ML and Haskell have **type inference**
- Explicit type applications are omitted – we write *map isEven* instead of *map \mathbb{N} bool isEven*
- Constraint propagation via the *unification algorithm* figures out what the applications should have been

Example:

<i>map isEven</i>	Term that needs type inference
<i>map ?a ?b isEven</i>	Introduce placeholders ?a and ?b
<i>map ?a ?b</i>	$: (?a \rightarrow ?b) \rightarrow \text{list } ?a \rightarrow \text{list } ?b$
<i>isEven : $\mathbb{N} \rightarrow \text{bool}$</i>	So $?a \rightarrow ?b$ must equal $\mathbb{N} \rightarrow \text{bool}$
<i>?a = \mathbb{N}, ?b = bool</i>	Only choice that makes $?a \rightarrow ?b = \mathbb{N} \rightarrow \text{bool}$

Effects

The Story so Far...

- We introduced the simply-typed lambda calculus
- ...and its double life as constructive propositional logic
- We extended it to the polymorphic lambda calculus
- ...and *its* double life as second-order logic

This is a story of **pure, total** functional programming

- Sometimes, we write programs that takes an input and computes an answer:
 - Physics simulations
 - Compiling programs
 - Ray-tracing software
- Other times, we write programs to *do things*:
 - communicate with the world via I/O and networking
 - update and modify physical state (eg, file systems)
 - build interactive systems like GUIs
 - control physical systems (eg, robots)
 - generate random numbers
- PL jargon: pure vs effectful code

Two Paradigms of Effects

- From the POV of type theory, two main classes of effects:
 1. State:
 - Mutable data structures (hash tables, arrays)
 - References/pointers
 2. Control:
 - Exceptions
 - Coroutines/generators
 - Nondeterminism
- Other effects (eg, I/O and concurrency/multithreading) can be modelled in terms of state and control effects
- In this lecture, we will focus on state and how to model it


```
# let r = ref 5;;  
val r : int ref = {contents = 5}
```

```
# !r;;  
- : int = 5
```

```
# r := !r + 15;;  
- : unit = ()
```

```
# !r;;  
- : int = 20
```

- We can *create fresh reference* with `ref e`
- We can *read a reference* with `!e`
- We can *update a reference* with `e := e'`

A Type System for State

Types	$X ::= 1 \mid \mathbb{N} \mid X \rightarrow Y \mid \text{ref} X$
Terms	$e ::= \langle \rangle \mid n \mid \lambda x : X. e \mid e e'$ $\mid \text{new } e \mid !e \mid e := e' \mid l$
Values	$v ::= \langle \rangle \mid n \mid \lambda x : X. e \mid l$
Stores	$\sigma ::= \cdot \mid \sigma, l : v$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : X$
Store Typings	$\Sigma ::= \cdot \mid \Sigma, l : X$

$$\frac{\langle \sigma; e_0 \rangle \rightsquigarrow \langle \sigma'; e'_0 \rangle}{\langle \sigma; e_0 e_1 \rangle \rightsquigarrow \langle \sigma'; e'_0 e_1 \rangle}$$

$$\frac{\langle \sigma; e_1 \rangle \rightsquigarrow \langle \sigma'; e'_1 \rangle}{\langle \sigma; v_0 e_1 \rangle \rightsquigarrow \langle \sigma'; v_0 e'_1 \rangle}$$

$$\frac{}{\langle \sigma; (\lambda x : X. e) v \rangle \rightsquigarrow \langle \sigma; [v/x]e \rangle}$$

- Similar to the basic STLC operational rules
- Threads a store σ through each transition

$$\frac{\langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle}{\langle \sigma; \text{new } e \rangle \rightsquigarrow \langle \sigma'; \text{new } e' \rangle}$$

$$\frac{l \notin \text{dom}(\sigma)}{\langle \sigma; \text{new } v \rangle \rightsquigarrow \langle (\sigma, l : v); l \rangle}$$

$$\frac{\langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle}{\langle \sigma; !e \rangle \rightsquigarrow \langle \sigma'; !e' \rangle}$$

$$\frac{l : v \in \sigma}{\langle \sigma; !l \rangle \rightsquigarrow \langle \sigma; v \rangle}$$

$$\frac{\langle \sigma; e_0 \rangle \rightsquigarrow \langle \sigma'; e'_0 \rangle}{\langle \sigma; e_0 := e_1 \rangle \rightsquigarrow \langle \sigma'; e'_0 := e_1 \rangle}$$

$$\frac{\langle \sigma; e_1 \rangle \rightsquigarrow \langle \sigma'; e'_1 \rangle}{\langle \sigma; v_0 := e_1 \rangle \rightsquigarrow \langle \sigma'; v_0 := e'_1 \rangle}$$

$$\frac{}{\langle (\sigma, l : v, \sigma'); l := v' \rangle \rightsquigarrow \langle (\sigma, l : v', \sigma'); \langle \rangle \rangle}$$

Typing for Terms

$$\boxed{\Sigma; \Gamma \vdash e : X}$$

$$\frac{x : X \in \Gamma}{\Sigma; \Gamma \vdash x : X} \text{HYP}$$

$$\frac{}{\Sigma; \Gamma \vdash \langle \rangle : 1} 1I$$

$$\frac{}{\Sigma; \Gamma \vdash n : \mathbb{N}} NI$$

$$\frac{\Sigma; \Gamma, x : X \vdash e : Y}{\Sigma; \Gamma \vdash \lambda x : X. e : X \rightarrow Y} \rightarrow I$$

$$\frac{\Sigma; \Gamma \vdash e : X \rightarrow Y \quad \Sigma; \Gamma \vdash e' : X}{\Sigma; \Gamma \vdash e e' : Y} \rightarrow E$$

- Similar to STLC rules + thread Σ through all judgements

Typing for Imperative Terms

$$\boxed{\Sigma; \Gamma \vdash e : X}$$

$$\frac{\Sigma; \Gamma \vdash e : X}{\Sigma; \Gamma \vdash \text{new } e : \text{ref } X} \text{REFI}$$

$$\frac{\Sigma; \Gamma \vdash e : \text{ref } X}{\Sigma; \Gamma \vdash !e : X} \text{REFGET}$$

$$\frac{\Sigma; \Gamma \vdash e : \text{ref } X \quad \Sigma; \Gamma \vdash e' : X}{\Sigma; \Gamma \vdash e := e' : 1} \text{REFSET}$$

$$\frac{l : X \in \Sigma}{\Sigma; \Gamma \vdash l : \text{ref } X} \text{REFBAR}$$

- Usual rules for references
- But why do we have the bare reference rule?

Proving Type Safety

- Original progress and preservations talked about well-typed terms e and evaluation steps $e \rightsquigarrow e'$
- New operational semantics $\langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle$ mentions stores, too.
- To prove type safety, we will need a notion of **store typing**

Store and Configuration Typing

$$\begin{array}{c} \boxed{\Sigma \vdash \sigma' : \Sigma'} \qquad \boxed{\langle \sigma; e \rangle : \langle \Sigma; X \rangle} \\[1em] \frac{}{\Sigma \vdash \cdot : \cdot} \text{STORENIL} \qquad \frac{\Sigma \vdash \sigma' : \Sigma' \quad \Sigma; \cdot \vdash v : X}{\Sigma \vdash (\sigma', l : v) : (\Sigma', l : X)} \text{STORECONS} \\[1em] \frac{\Sigma \vdash \sigma : \Sigma \quad \Sigma; \cdot \vdash e : X}{\langle \sigma; e \rangle : \langle \Sigma; X \rangle} \text{CONFIGOK} \end{array}$$

- Check that all the closed values in the store σ' are well-typed
- Types come from Σ' , checked in store Σ
- Configurations are well-typed if the store and term are well-typed

A Broken Theorem

Progress:

If $\langle \sigma; e \rangle : \langle \Sigma; X \rangle$ then e is a value or $\langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle$.

Preservation:

If $\langle \sigma; e \rangle : \langle \Sigma; X \rangle$ and $\langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle$ then $\langle \sigma'; e' \rangle : \langle \Sigma; X \rangle$.

- One of these theorems is false!

The Counterexample to Preservation

Note that

1. $\langle \cdot; \text{new } \langle \rangle \rangle : \langle \cdot; \text{ref } 1 \rangle$
2. $\langle \cdot; \text{new } \langle \rangle \rangle \rightsquigarrow \langle (l : \langle \rangle); l \rangle$ for some l

However, it is not the case that

$$\langle (l : \langle \rangle); l \rangle : \langle \cdot; \text{ref } 1 \rangle$$

The heap has **grown!**

Store Monotonicity

Definition (Store extension):

Define $\Sigma \leq \Sigma'$ to mean there is a Σ'' such that $\Sigma' = \Sigma, \Sigma''$.

Lemma (Store Monotonicity):

If $\Sigma \leq \Sigma'$ then:

1. If $\Sigma; \Gamma \vdash e : X$ then $\Sigma'; \Gamma \vdash e : X$.
2. If $\Sigma \vdash \sigma_0 : \Sigma_0$ then $\Sigma' \vdash \sigma_0 : \Sigma_0$.

The proof is by structural induction on the appropriate definition.

This property means allocating new references never breaks the typability of a term.

Substitution and Structural Properties

- (Weakening)

If $\Sigma; \Gamma, \Gamma' \vdash e : X$ then $\Sigma; \Gamma, z : Z, \Gamma' \vdash e : X$.

- (Exchange)

If $\Sigma; \Gamma, y : Y, z : Z, \Gamma' \vdash e : X$ then $\Sigma; \Gamma, z : Z, y : Y, \Gamma' \vdash e : X$.

- (Substitution)

If $\Sigma; \Gamma \vdash e : X$ and $\Sigma; \Gamma, x : X \vdash e' : Z$ then $\Sigma; \Gamma \vdash [e/x]e' : Z$.

Theorem (Progress):

If $\langle \sigma; e \rangle : \langle \Sigma; X \rangle$ then e is a value or $\langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle$.

Theorem (Preservation):

If $\langle \sigma; e \rangle : \langle \Sigma; X \rangle$ and $\langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle$ then there exists $\Sigma' \geq \Sigma$ such that $\langle \sigma'; e' \rangle : \langle \Sigma'; X \rangle$.

Proof:

- For progress, induction on derivation of $\Sigma; \cdot \vdash e : X$
- For preservation, induction on derivation of $\langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle$

A Curious Higher-order Function

- Suppose we have an unknown function in the STLC:

$$f: ((1 \rightarrow 1) \rightarrow 1) \rightarrow \mathbb{N}$$

- Q: What can this function do?
- A: It is a constant function, returning some n
- Q: Why?
- A: No matter what $f(g)$ does with its argument g , it can only get $\langle \rangle$ out of it. So the argument can never influence the value of type \mathbb{N} that f produces.

The Power of the State

```
count      : ((1 → 1) → 1) → ℕ  
count f = let r : ref ℕ = new 0 in  
           let inc : 1 → 1 = λz : 1. r := !r + 1 in  
           f(inc); !r
```

- This function initializes a counter r
- It creates a function inc which silently increments r
- It passes inc to its argument f
- Then it returns the value of the counter r
- That is, it returns the **number of times** inc was called!

Backpatching with Landin's Knot

```
1  let knot : ((int -> int) -> int -> int) -> int -> int =  
2    fun f ->  
3      let r      = ref (fun n -> 0) in  
4      let recur   = fun n -> !r n in  
5      let ()      = r := fun n -> f recur n in  
6      recur
```

1. Create a reference holding a function
2. Define a function that forwards its argument to the ref
3. Set the reference to a function that calls f on the forwarder and the argument n
4. Now f will call itself recursively!

Not a Theorem: (Termination) Every well-typed program $\cdot; \cdot \vdash e : X$ terminates.

- Landin's knot lets us *define recursive functions* by backpatching
- As a result, we can write nonterminating programs!

Consistency vs Computation

- Do we have to choose between state/effects and logical consistency?
- Is there a way to get the best of both?
- Alternately, is there a Curry-Howard interpretation for effects?
- Next lecture:
 - A modal logic suggested by Curry in 1952
 - Now known to functional programmers as *monads*
 - Also known as *effect systems*

Questions

1. Using Landin's knot, implement the fibonacci function.
2. The type safety proof for state would fail if we added a C-like `free()` operation to the reference API.
 - 2.1 Give a plausible-looking typing rule and operational semantics for `free`.
 - 2.2 Find an example of a program that would break.

Type Systems

Lecture 8: Using Monads to Control Effects

Neel Krishnaswami
University of Cambridge

Last Lecture

```
1  let knot : ((int -> int) -> int -> int) -> int -> int =  
2    fun f ->  
3      let r      = ref (fun n -> 0) in  
4      let recur  = fun n -> !r n in  
5      let ()     = r := fun n -> f recur n in  
6      recur
```

1. Create a reference holding a function
2. Define a function that forwards its argument to the ref
3. Set the reference to a function that calls f on the forwarder and the argument n
4. Now f will call itself recursively!

Not a Theorem: (Termination) Every well-typed program $\cdot; \cdot \vdash e : X$ terminates.

- Landin's knot lets us *define recursive functions* by backpatching
- As a result, we can write nonterminating programs

What is the Problem?

1. We began with the typed lambda calculus
2. We added state as a set of primitive operations
3. We lost termination
4. Problem: unforeseen **interaction** between different parts of the language
 - Recursive definitions = state + functions
5. Question: is this a real problem?

What is the Solution?

- Restrict the use of state:
 1. Limit what references can store (eg, only to booleans and integers)
 2. Restrict how references can be referred to (eg, in core safe Rust)
 3. We don't have time to pursue these in this course
- Mark the use of state:
 - Distinguish between *pure* and *impure* code
 - Impure computations can depend on pure ones
 - Pure computations **cannot** depend upon impure ones
 - A form of **taint tracking**

Monads for State

Types	$X ::= 1 \mid \mathbb{N} \mid X \rightarrow Y \mid \text{ref } X \mid \top X$
Pure Terms	$e ::= \langle \rangle \mid n \mid \lambda x : X. e \mid e e' \mid l \mid \{t\}$
Impure Terms	$t ::= \text{new } e \mid !e \mid e := e' \mid \text{let } x = e; t \mid \text{return } e$
Values	$v ::= \langle \rangle \mid n \mid \lambda x : X. e \mid l \mid \{t\}$
Stores	$\sigma ::= \cdot \mid \sigma, l : v$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : X$
Store Typings	$\Sigma ::= \cdot \mid \Sigma, l : X$

Typing for Pure Terms

$$\boxed{\Sigma; \Gamma \vdash e : X}$$

$$\frac{x : X \in \Gamma}{\Sigma; \Gamma \vdash x : X} \text{HYP}$$

$$\frac{}{\Sigma; \Gamma \vdash \langle \rangle : 1} \text{1I}$$

$$\frac{}{\Sigma; \Gamma \vdash n : \mathbb{N}} \text{NI}$$

$$\frac{\Sigma; \Gamma, x : X \vdash e : Y}{\Sigma; \Gamma \vdash \lambda x : X. e : X \rightarrow Y} \rightarrow \text{I}$$

$$\frac{\Sigma; \Gamma \vdash e : X \rightarrow Y \quad \Sigma; \Gamma \vdash e' : X}{\Sigma; \Gamma \vdash e e' : Y} \rightarrow \text{E}$$

$$\frac{l : X \in \Sigma}{\Sigma; \Gamma \vdash l : \text{ref } X} \text{REFBAR}$$

$$\frac{\Sigma; \Gamma \vdash t \div X}{\Sigma; \Gamma \vdash \{t\} : \text{TX}} \text{TI}$$

- Similar to STLC rules + thread Σ through all judgements
- New judgement $\Sigma; \Gamma \vdash t \div X$ for imperative computations

Typing for Effectful Terms

$$\boxed{\Sigma; \Gamma \vdash t \div X}$$

$$\frac{\Sigma; \Gamma \vdash e : X}{\Sigma; \Gamma \vdash \text{new } e \div \text{ref } X} \text{REFI}$$

$$\frac{\Sigma; \Gamma \vdash e : \text{ref } X}{\Sigma; \Gamma \vdash !e \div X} \text{REFGET}$$

$$\frac{\Sigma; \Gamma \vdash e : \text{ref } X \quad \Sigma; \Gamma \vdash e' : X}{\Sigma; \Gamma \vdash e := e' \div 1} \text{REFSET}$$

$$\frac{\Sigma; \Gamma \vdash e : X}{\Sigma; \Gamma \vdash \text{return } e \div X} \text{TRET}$$

$$\frac{\Sigma; \Gamma \vdash e : TX \quad \Sigma; \Gamma, x : X \vdash t \div Z}{\Sigma; \Gamma \vdash \text{let } x = e; t \div Z} \text{TLET}$$

- We now mark potentially effectful terms in the judgement
- Note that `return e` isn't effectful – conservative approximation!

A Two-Level Operational Semantics: Pure Part

$$\frac{e_0 \rightsquigarrow e'_0}{e_0 e_1 \rightsquigarrow e'_0 e_1}$$

$$\frac{e_1 \rightsquigarrow e'_1}{v_0 e_1 \rightsquigarrow v_0 e'_1}$$

$$\frac{}{(\lambda x : X. e) v \rightsquigarrow [v/x]e}$$

- Similar to the basic STLC operational rules
- We no longer thread a store σ through each transition!

A Two-Level Operational Semantics: Impure Part, 1/2

$$\frac{e \rightsquigarrow e'}{\langle \sigma; \text{new } e \rangle \rightsquigarrow \langle \sigma; \text{new } e' \rangle}$$

$$\frac{l \notin \text{dom}(\sigma)}{\langle \sigma; \text{new } v \rangle \rightsquigarrow \langle (\sigma, l : v); \text{return } l \rangle}$$

$$\frac{e \rightsquigarrow e'}{\langle \sigma; !e \rangle \rightsquigarrow \langle \sigma; !e' \rangle}$$

$$\frac{l : v \in \sigma}{\langle \sigma; !l \rangle \rightsquigarrow \langle \sigma; \text{return } v \rangle}$$

$$\frac{e_0 \rightsquigarrow e'_0}{\langle \sigma; e_0 := e_1 \rangle \rightsquigarrow \langle \sigma; e'_0 := e_1 \rangle}$$

$$\frac{e_1 \rightsquigarrow e'_1}{\langle \sigma; v_0 := e_1 \rangle \rightsquigarrow \langle \sigma; v_0 := e'_1 \rangle}$$

$$\frac{}{\langle (\sigma, l : v, \sigma'); l := v' \rangle \rightsquigarrow \langle (\sigma, l : v', \sigma'); \text{return } \langle \rangle \rangle}$$

A Two-Level Operational Semantics: Impure Part, 2/2

$$\frac{e \rightsquigarrow e'}{\langle \sigma; \text{return } e \rangle \rightsquigarrow \langle \sigma; \text{return } e' \rangle}$$

$$\frac{e \rightsquigarrow e'}{\langle \sigma; \text{let } x = e; t \rangle \rightsquigarrow \langle \sigma; \text{let } x = e'; t \rangle}$$

$$\frac{}{\langle \sigma; \text{let } x = \{\text{return } v\}; t_1 \rangle \rightsquigarrow \langle \sigma; [v/x]t_1 \rangle}$$

$$\frac{\langle \sigma; t_0 \rangle \rightsquigarrow \langle \sigma'; t'_0 \rangle}{\langle \sigma; \text{let } x = \{t_0\}; t_1 \rangle \rightsquigarrow \langle \sigma'; \text{let } x = \{t'_0\}; t_1 \rangle}$$

Store and Configuration Typing

$$\begin{array}{c} \boxed{\Sigma \vdash \sigma' : \Sigma'} \qquad \boxed{\langle \sigma; e \rangle : \langle \Sigma; X \rangle} \\[1em] \frac{}{\Sigma \vdash \cdot : \cdot} \text{STORENIL} \qquad \frac{\Sigma \vdash \sigma' : \Sigma' \quad \Sigma; \cdot \vdash v : X}{\Sigma \vdash (\sigma', l : v) : (\Sigma', l : X)} \text{STORECONS} \\[1em] \frac{\Sigma \vdash \sigma : \Sigma \quad \Sigma; \cdot \vdash t \div X}{\langle \sigma; t \rangle : \langle \Sigma; X \rangle} \text{CONFIGOK} \end{array}$$

- Check that all the closed values in the store σ' are well-typed
- Types come from Σ' , checked in store Σ
- Configurations are well-typed if the store and term are well-typed

- **Pure Term Weakening:**

If $\Sigma; \Gamma, \Gamma' \vdash e : X$ then $\Sigma; \Gamma, z : Z, \Gamma' \vdash e : X$.

- **Pure Term Exchange:**

If $\Sigma; \Gamma, y : Y, z : Z, \Gamma' \vdash e : X$ then $\Sigma; \Gamma, z : Z, y : Y, \Gamma' \vdash e : X$.

- **Pure Term Substitution:**

If $\Sigma; \Gamma \vdash e : X$ and $\Sigma; \Gamma, x : X \vdash e' : Z$ then $\Sigma; \Gamma \vdash [e/x]e' : Z$.

- **Effectful Term Weakening:**

If $\Sigma; \Gamma, \Gamma' \vdash t \div X$ then $\Sigma; \Gamma, z : Z, \Gamma' \vdash t \div X$.

- **Effectful Term Exchange:**

If $\Sigma; \Gamma, y : Y, z : Z, \Gamma' \vdash t \div X$ then $\Sigma; \Gamma, z : Z, y : Y, \Gamma' \vdash t \div X$.

- **Effectful Term Substitution:**

If $\Sigma; \Gamma \vdash e : X$ and $\Sigma; \Gamma, x : X \vdash t \div Z$ then $\Sigma; \Gamma \vdash [e/x]t \div Z$.

1. Prove Pure Term Weakening and Impure Term Weakening mutually inductively
2. Prove Pure Term Exchange and Impure Term Exchange mutually inductively
3. Prove Pure Term Substitution and Impure Term Substitution mutually inductively

Two mutually-recursive judgements \implies Two mutually-inductive proofs

Store Monotonicity

Definition (Store extension):

Define $\Sigma \leq \Sigma'$ to mean there is a Σ'' such that $\Sigma' = \Sigma, \Sigma''$.

Lemma (Store Monotonicity):

If $\Sigma \leq \Sigma'$ then:

1. If $\Sigma; \Gamma \vdash e : X$ then $\Sigma'; \Gamma \vdash e : X$.
2. If $\Sigma; \Gamma \vdash t \div X$ then $\Sigma'; \Gamma \vdash t \div X$.
3. If $\Sigma \vdash \sigma_0 : \Sigma_0$ then $\Sigma' \vdash \sigma_0 : \Sigma_0$.

The proof is by structural induction on the appropriate definition. (Prove 1. and 2. mutually-inductively!)

This property means allocating new references never breaks the typability of a term.

Theorem (Pure Progress):

If $\Sigma; \cdot \vdash e : X$ then $e = v$ or $e \rightsquigarrow e'$.

Theorem (Pure Preservation):

If $\Sigma; \cdot \vdash e : X$ and $e \rightsquigarrow e'$ then $\Sigma; \cdot \vdash e' : X$.

Proof:

- For progress, induction on derivation of $\Sigma; \cdot \vdash e : X$
- For preservation, induction on derivation of $e \rightsquigarrow e'$

Type Safety for the Monadic Language

Theorem (Progress):

If $\langle \sigma; t \rangle : \langle \Sigma; X \rangle$ then $t = \text{return } v$ or $\langle \sigma; t \rangle \rightsquigarrow \langle \sigma'; t' \rangle$.

Theorem (Preservation):

If $\langle \sigma; t \rangle : \langle \Sigma; X \rangle$ and $\langle \sigma; t \rangle \rightsquigarrow \langle \sigma'; t' \rangle$ then there exists $\Sigma' \geq \Sigma$ such that $\langle \sigma'; t' \rangle : \langle \Sigma'; X \rangle$.

Proof:

- For progress, induction on derivation of $\Sigma; \cdot \vdash t \div X$
- For preservation, induction on derivation of $\langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle$

What Have we Accomplished?

- In the monadic language, pure and effectful code is strictly separated
- As a result, *pure programs terminate*
- However, *we can still write imperative programs*

Monads for I/O

Types	$X ::= 1 \mid \mathbb{N} \mid X \rightarrow Y \mid T_{IO} X$
Pure Terms	$e ::= \langle \rangle \mid n \mid \lambda x : X. e \mid e e' \mid \{t\}$
Impure Terms	$t ::= \text{print } e \mid \text{let } x = e; t \mid \text{return } e$
Values	$v ::= \langle \rangle \mid n \mid \lambda x : X. e \mid \{t\}$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : X$

Monads for I/O: Typing Pure Terms

$$\boxed{\Gamma \vdash e : X}$$

$$\frac{x : X \in \Gamma}{\Gamma \vdash x : X} \text{HYP}$$

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

$$\frac{}{\Gamma \vdash n : \mathbb{N}} \text{NI}$$

$$\frac{\Gamma, x : X \vdash e : Y}{\Gamma \vdash \lambda x : X. e : X \rightarrow Y} \rightarrow \text{I}$$

$$\frac{\Gamma \vdash e : X \rightarrow Y \quad \Gamma \vdash e' : X}{\Gamma \vdash e e' : Y} \rightarrow \text{E}$$

$$\frac{\Gamma \vdash t \div X}{\Gamma \vdash \{t\} : \text{TX}} \text{TI}$$

- Similar to STLC rules (no store typing!)
- New judgement $\Gamma \vdash t \div X$ for imperative computations

$$\boxed{\Gamma \vdash t \div X}$$

$$\frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \text{print } e \div 1} \text{TPRINT}$$

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash \text{return } e \div X} \text{TRET}$$

$$\frac{\Gamma \vdash e : TX \quad \Gamma, x : X \vdash t \div Z}{\Gamma \vdash \text{let } x = e; t \div Z} \text{TLET}$$

- TRET and TLET are identical rules
- Difference is in the operations – $\text{print } e$ vs get/set/new

Operational Semantics for I/O: Pure Part

$$\frac{e_0 \rightsquigarrow e'_0}{e_0 e_1 \rightsquigarrow e'_0 e_1}$$

$$\frac{e_1 \rightsquigarrow e'_1}{v_0 e_1 \rightsquigarrow v_0 e'_1}$$

$$\frac{}{(\lambda x : X. e) v \rightsquigarrow [v/x]e}$$

- *Identical* to the pure rules for state!

Operational Semantics for I/O: Impure Part

$$\frac{e \rightsquigarrow e'}{\langle \omega; \text{print } e \rangle \rightsquigarrow \langle \omega; \text{print } e' \rangle}$$

$$\frac{}{\langle \omega; \text{print } n \rangle \rightsquigarrow \langle (n :: \omega); \text{return } \langle \rangle \rangle}$$

$$\frac{e \rightsquigarrow e'}{\langle \omega; \text{return } e \rangle \rightsquigarrow \langle \omega; \text{return } e' \rangle}$$

$$\frac{e \rightsquigarrow e'}{\langle \omega; \text{let } x = e; t \rangle \rightsquigarrow \langle \omega; \text{let } x = e'; t \rangle}$$

$$\frac{}{\langle \omega; \text{let } x = \{\text{return } v\}; t_1 \rangle \rightsquigarrow \langle \omega; [v/x]t_1 \rangle}$$

$$\frac{\langle \omega; t_0 \rangle \rightsquigarrow \langle \omega'; t'_0 \rangle}{\langle \omega; \text{let } x = \{t_0\}; t_1 \rangle \rightsquigarrow \langle \omega'; \text{let } x = \{t'_0\}; t_1 \rangle}$$

- State is now a list of output tokens
- All rules otherwise identical except for operations

Limitations of Monadic Style: Encapsulating Effects

```
1  let fact : int -> int = fun n ->
2    let r = ref 1 in
3    let rec loop n =
4      match n with
5      | 0 -> !r
6      | n -> let () = r := !r * n in
7              loop (n-1)
8    in
9    loop n
```

- This function use *local state*
- No caller can tell if it uses state or not
- Should it have a pure type, or a monadic type?

Limitations of Monadic Style: Encapsulating Effects

```
1  let rec find' : ('a -> bool) -> 'a list -> 'a =  
2      fun p ys ->  
3          match ys with  
4          | [] -> raise Not_found  
5          | y :: ys -> if p y then y else find' p ys  
6  
7  let find : ('a -> bool) -> 'a list -> 'a option =  
8      fun p xs ->  
9          try Some (find' p xs)  
10         with Not_found -> None
```

- `find'` has an effect – it can raise an exception
- But `find` calls `find'`, *and* catches the exception
- Should `find` have an exception monad in its type?

Limitations of Monadic Style: Combining Effects

Suppose you have two programs:

```
1      p1 : (int -> ans) state
2      p2 : int io
```

- we write **a state** for a state monad computation
- we write **b io** for a I/O monad computation
- How do we write a program that does **p2**, and passes its argument to **p1**?

Checked Exceptions in Java

- Java *checked exceptions* implement a simple form of effect typing
- Method declarations state which exceptions a method can raise
- Programmer must catch and handle any exceptions they haven't declared they can raise
- Not much used in modern code – type system too inflexible

Effects in Koka

```
fun square1( x : int ) : total int    { x*x }
fun square2( x : int ) : console int { println( "a not so secret side-effect" ); x*x }
fun square3( x : int ) : div int      { x * square3( x ) }
fun square4( x : int ) : exn int      { throw( "oops" ); x*x }
```

checked

- Koka is a new language from Microsoft Research
- Uses effect tracking to track totality, partiality, exceptions, I/O, state and even user-defined effects
- Good playground to understand how monadic effects could look like in a practical language
- See: <https://github.com/koka-lang/koka>

For the monadic I/O language:

1. State the weakening, exchange, and substitution lemmas
2. Define machine configurations and configuration typing
3. State the type safety property

Type Systems

Lecture 9: Classical Logic

Neel Krishnaswami
University of Cambridge

Where We Are

We have seen the Curry Howard correspondence:

- **Intuitionistic** propositional logic \longleftrightarrow Simply-typed lambda calculus
- Second-order **intuitionistic** logic \longleftrightarrow Polymorphic lambda calculus

We have seen effectful programs:

- State
- I/O
- Monads

But what about:

- Control operators (eg, exceptions, **goto**, etc)
- **Classical** logic

A Review of Intuitionistic Propositional Logic

$$\frac{P \in \Psi}{\Psi \vdash P \text{ true}} \text{ HYP}$$

$$\frac{}{\Psi \vdash \top \text{ true}} \top\text{I}$$

$$\frac{\Psi \vdash P \text{ true} \quad \Psi \vdash Q \text{ true}}{\Psi \vdash P \wedge Q \text{ true}} \wedge\text{I}$$

$$\frac{\Psi \vdash P_1 \wedge P_2 \text{ true}}{\Psi \vdash P_i \text{ true}} \wedge\text{E}_i$$

$$\frac{\Psi, P \vdash Q \text{ true}}{\Psi \vdash P \supset Q \text{ true}} \supset\text{I}$$

$$\frac{\Psi \vdash P \supset Q \text{ true} \quad \Psi \vdash P \text{ true}}{\Psi \vdash Q \text{ true}} \supset\text{E}$$

Disjunction and Falsehood

$$\frac{\Psi \vdash P \text{ true}}{\Psi \vdash P \vee Q \text{ true}} \vee I_1$$

$$\frac{\Psi \vdash Q \text{ true}}{\Psi \vdash P \vee Q \text{ true}} \vee I_2$$

$$\frac{\Psi \vdash P \vee Q \text{ true} \quad \Psi, P \vdash R \text{ true} \quad \Psi, Q \vdash R \text{ true}}{\Psi \vdash R \text{ true}} \vee E$$

(no intro for \perp)

$$\frac{\Psi \vdash \perp \text{ true}}{\Psi \vdash R \text{ true}} \perp E$$

Intuitionistic Propositional Logic

- Key judgement: $\Psi \vdash R$ true
 - “If everything in Ψ is true, then R is true”
- Negation $\neg P$ is a derived notion
 - Definition: $\neg P = P \rightarrow \perp$
 - “Not P ” means “ P implies false”
 - To *refute* P means to give a *proof* that P implies false

What if we treat refutations as a first-class notion?

A Calculus of Truth and Falsehood

Propositions $A ::= \top \mid A \wedge B \mid \perp \mid A \vee B \mid \neg A$

True contexts $\Gamma ::= \cdot \mid \Gamma, A$

False contexts $\Delta ::= \cdot \mid \Delta, A$

Proofs $\Gamma; \Delta \vdash A$ true If Γ is true and Δ is false, A is true

Refutations $\Gamma; \Delta \vdash A$ false If Γ is true and Δ is false, A is false

Contradictions $\Gamma; \Delta \vdash \text{contr}$ Γ and Δ contradict one another

- $\neg A$ is primitive (no implication $A \rightarrow B$)
- Eventually, we'll encode it as $\neg A \vee B$

(No rule for \perp)

$$\frac{}{\Gamma; \Delta \vdash \top \text{ true}} \top P$$

$$\frac{\Gamma; \Delta \vdash A \text{ true} \quad \Gamma; \Delta \vdash B \text{ true}}{\Gamma; \Delta \vdash A \wedge B \text{ true}} \wedge P$$

$$\frac{\Gamma; \Delta \vdash A \text{ true}}{\Gamma; \Delta \vdash A \vee B \text{ true}} \vee P_1$$

$$\frac{\Gamma; \Delta \vdash B \text{ true}}{\Gamma; \Delta \vdash A \vee B \text{ true}} \vee P_2$$

$$\frac{A \in \Gamma}{\Gamma; \Delta \vdash A \text{ true}} \text{HYP}$$

$$\frac{\Gamma; \Delta \vdash A \text{ false}}{\Gamma; \Delta \vdash \neg A \text{ true}} \neg P$$

$$\text{(No rule for } \top \text{)} \qquad \frac{}{\Gamma; \Delta \vdash \perp \text{ false}} \perp R$$

$$\frac{\Gamma; \Delta \vdash A \text{ false} \quad \Gamma; \Delta \vdash B \text{ false}}{\Gamma; \Delta \vdash A \vee B \text{ false}} \vee R$$

$$\frac{\Gamma; \Delta \vdash A \text{ false}}{\Gamma; \Delta \vdash A \wedge B \text{ false}} \wedge R_1$$

$$\frac{\Gamma; \Delta \vdash B \text{ false}}{\Gamma; \Delta \vdash A \wedge B \text{ false}} \wedge R_2$$

$$\frac{A \in \Delta}{\Gamma; \Delta \vdash A \text{ false}} \text{HYP}$$

$$\frac{\Gamma; \Delta \vdash A \text{ true}}{\Gamma; \Delta \vdash \neg A \text{ false}} \neg R$$

75% of the Way to Classical Logic

Connective	To Prove	To Refute
\top	Do nothing	Impossible!
$A \wedge B$	Prove <i>A</i> and prove <i>B</i>	Refute <i>A</i> or refute <i>B</i>
\perp	Impossible!	Do nothing
$A \vee B$	Prove <i>A</i> or prove <i>B</i>	Refute <i>A</i> and refute <i>B</i>
$\neg A$	Refute <i>A</i>	Prove <i>A</i>

Something We Can Prove: A entails $\neg\neg A$

$$\frac{\frac{\frac{}{A; \cdot \vdash A \text{ true}}{\text{HYP}}}{A; \cdot \vdash \neg A \text{ false}} \neg R}{A; \cdot \vdash \neg\neg A \text{ true}} \neg P$$

Something We Cannot Prove: $\neg\neg A$ entails A

$$\frac{???}{\neg\neg A; \cdot \vdash A \text{ true}}$$

- There is no rule that applies in this case
- Proofs and refutations are mutually recursive
- But we have no way to use assumptions!

Something Else We Cannot Prove: $A \wedge B$ entails A

$$\frac{???}{A \wedge B; \cdot \vdash A \text{ true}}$$

- This is intuitionistically valid: $\lambda x : A \times B. \text{fst } x$
- But it's not derivable here
- Again, we can't use hypotheses nontrivially

A Bold Assumption

- Proofs and refutations are perfectly symmetrical
- This suggests the following idea:
 1. To refute A means to give direct evidence it is false
 2. This is also how we prove $\neg A$
 3. If we show a contradiction from assuming A is false, we have proved it
 4. If we can show a contradiction from assuming A is true, we have refuted it

$$\frac{\Gamma; \Delta, A \vdash \text{contr}}{\Gamma; \Delta \vdash A \text{ true}}$$

$$\frac{\Gamma, A; \Delta \vdash \text{contr}}{\Gamma; \Delta \vdash A \text{ false}}$$

$$\frac{\Gamma; \Delta \vdash A \text{ true} \quad \Gamma; \Delta \vdash A \text{ false}}{\Gamma; \Delta \vdash \text{contr}} \text{ CONTR}$$

- A contradiction arises when A has a proof *and* a refutation

Double Negation Elimination

	$\neg\neg A; A \vdash A$ false
	$\neg\neg A; A \vdash \neg A$ true
$\neg\neg A; A \vdash \neg\neg A$ true	$\neg\neg A; A \vdash \neg\neg A$ false
$\neg\neg A; A \vdash \text{contr}$	
$\neg\neg A; \cdot \vdash A$ true	

Projections: $A \wedge B$ entails A

$$\frac{\frac{}{A \wedge B; A \vdash A \wedge B \text{ true}} \quad \frac{\frac{}{A \wedge B; A \vdash A \text{ false}}}{A \wedge B; A \vdash A \wedge B \text{ false}}}{A \wedge B; A \vdash \text{contr}} \quad \frac{}{A \wedge B; \cdot \vdash A \text{ true}}$$

Projections: $A \vee B$ false entails A false

$$\frac{\frac{}{A; A \vee B \vdash A \vee B \text{ false}} \quad \frac{\frac{}{A; A \vee B \vdash A \text{ true}}}{A; A \vee B \vdash A \vee B \text{ true}}}{A; A \vee B \vdash \text{contr}} \quad \frac{}{.; A \vee B \vdash A \text{ false}}$$

The Excluded Middle

$$\begin{array}{c} \vdots \\ \hline \cdot; A \vee \neg A \vdash A \text{ false} \\ \hline \cdot; A \vee \neg A \vdash \neg A \text{ true} \\ \hline \cdot; A \vee \neg A \vdash A \vee \neg A \text{ true} \qquad \cdot; A \vee \neg A \vdash A \vee \neg A \text{ false} \\ \hline \cdot; A \vee \neg A \vdash \text{contr} \\ \hline \cdot; \cdot \vdash A \vee \neg A \text{ true} \end{array}$$

Proof (and Refutation) Terms

Propositions	$A ::= \top \mid A \wedge B \mid \perp \mid A \vee B \mid \neg A$
True contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$
False contexts	$\Delta ::= \cdot \mid \Delta, u : A$
Values	$e ::= \langle \rangle \mid \langle e, e' \rangle \mid L e \mid R e \mid \text{not}(k)$ $\mid \mu u : A. c$
Continuations	$k ::= [] \mid [k, k'] \mid \text{fst } k \mid \text{snd } k \mid \text{not}(e)$ $\mid \mu x : A. c$
Contradictions	$c ::= \langle e \mid_A k \rangle$

$$\text{(No rule for } \perp \text{)} \qquad \frac{}{\Gamma; \Delta \vdash \langle \rangle : \top \text{ true}} \top P$$

$$\frac{\Gamma; \Delta \vdash e : A \text{ true} \quad \Gamma; \Delta \vdash e' : B \text{ true}}{\Gamma; \Delta \vdash \langle e, e' \rangle : A \wedge B \text{ true}} \wedge P$$

$$\frac{\Gamma; \Delta \vdash e : A \text{ true}}{\Gamma; \Delta \vdash L e : A \vee B \text{ true}} \vee P_1$$

$$\frac{\Gamma; \Delta \vdash e : B \text{ true}}{\Gamma; \Delta \vdash R e : A \vee B \text{ true}} \vee P_2$$

$$\frac{x : A \in \Gamma}{\Gamma; \Delta \vdash x : A \text{ true}} \text{HYP}$$

$$\frac{\Gamma; \Delta \vdash k : A \text{ false}}{\Gamma; \Delta \vdash \text{not}(k) : \neg A \text{ true}} \neg P$$

Continuations — Refutation Terms

$$\text{(No rule for } \top \text{)} \qquad \frac{}{\Gamma; \Delta \vdash [] : \perp \text{ false}} \perp R$$

$$\frac{\Gamma; \Delta \vdash k : A \text{ false} \quad \Gamma; \Delta \vdash k' : B \text{ false}}{\Gamma; \Delta \vdash [k, k'] : A \vee B \text{ false}} \vee R$$

$$\frac{\Gamma; \Delta \vdash k : A \text{ false}}{\Gamma; \Delta \vdash \text{fst } k : A \wedge B \text{ false}} \wedge R_1$$

$$\frac{\Gamma; \Delta \vdash k : B \text{ false}}{\Gamma; \Delta \vdash \text{snd } k : A \wedge B \text{ false}} \wedge R_2$$

$$\frac{x : A \in \Delta}{\Gamma; \Delta \vdash x : A \text{ false}} \text{HYP}$$

$$\frac{\Gamma; \Delta \vdash e : A \text{ true}}{\Gamma; \Delta \vdash \text{not}(e) : \neg A \text{ false}} \neg R$$

Contradictions

$$\frac{\Gamma; \Delta \vdash e : A \text{ true} \quad \Gamma; \Delta \vdash k : A \text{ false}}{\Gamma; \Delta \vdash \langle e \mid_A k \rangle \text{ contr}} \text{ CONTR}$$

$$\frac{\Gamma; \Delta, u : A \vdash c \text{ contr}}{\Gamma; \Delta \vdash \mu u : A. c : A \text{ true}}$$

$$\frac{\Gamma, x : A; \Delta \vdash c \text{ contr}}{\Gamma; \Delta \vdash \mu x : A. c : A \text{ false}}$$

$$\langle \langle e_1, e_2 \rangle \mid_{A \wedge B} \text{fst } k \rangle \quad \mapsto \quad \langle e_1 \mid_A k \rangle$$

$$\langle \langle e_1, e_2 \rangle \mid_{A \wedge B} \text{snd } k \rangle \quad \mapsto \quad \langle e_2 \mid_B k \rangle$$

$$\langle L e \mid_{A \vee B} [k_1, k_2] \rangle \quad \mapsto \quad \langle e \mid_A k_1 \rangle$$

$$\langle R e \mid_{A \vee B} [k_1, k_2] \rangle \quad \mapsto \quad \langle e \mid_B k_2 \rangle$$

$$\langle \text{not}(k) \mid_{\neg A} \text{not}(e) \rangle \quad \mapsto \quad \langle e \mid_A k \rangle$$

$$\langle \mu u : A. c \mid_A k \rangle \quad \mapsto \quad [k/u]c$$

$$\langle e \mid_A \mu X : A. c \rangle \quad \mapsto \quad [e/X]c$$

$$\langle \mu u : A. c \mid_A \mu x : A. c' \rangle \mapsto ?$$

- Two rules apply!
- Different choices of priority correspond to *evaluation order*
- Similar situation in the simply-typed lambda calculus
- The STLC is *confluent*, so evaluation order doesn't matter
- But in the classical case, evaluation order matters a lot!

Metatheory: Substitution

- If $\Gamma; \Delta \vdash e : A$ true then
 1. If $\Gamma, x : A; \Delta \vdash e' : C$ true then $\Gamma; \Delta \vdash [e/x]e' : C$ true.
 2. If $\Gamma, x : A; \Delta \vdash k : C$ false then $\Gamma; \Delta \vdash [e/x]k : C$ false.
 3. If $\Gamma, x : A; \Delta \vdash c$ contr then $\Gamma; \Delta \vdash [e/x]c$ contr.
- If $\Gamma; \Delta \vdash k : A$ false then
 1. If $\Gamma; \Delta, u : A \vdash e' : C$ true then $\Gamma; \Delta \vdash [k/u]e' : C$ true.
 2. If $\Gamma; \Delta, x : A \vdash k' : C$ false then $\Gamma; \Delta \vdash [k/u]k' : C$ false.
 3. If $\Gamma; \Delta, u : A \vdash c$ contr then $\Gamma; \Delta \vdash [k/u]c$ contr.
- We *also* need to prove weakening and exchange!
- Because there are 2 kinds of assumptions, and 3 kinds of judgement, there are $2 \times 3 = 6$ lemmas!

What Is This For?

- We have introduced a proof theory for classical logic
- Expected tautologies and metatheory holds...
- ...but it looks totally different from STLC?
- Computationally, this is a calculus for *stack machines*
- Related to *continuation passing style* (next lecture!)

Questions

1. Show that $\neg A \vee B, A; \cdot \vdash B$ true is derivable
2. Show that $\neg(\neg A \wedge \neg B); \cdot \vdash A \vee B$ true is derivable
3. Prove substitution for values (you may assume exchange and weakening hold).

Type Systems

Lecture 10: Classical Logic and Continuation-Passing Style

Neel Krishnaswami
University of Cambridge

Proof (and Refutation) Terms

Propositions	$A ::= \top \mid A \wedge B \mid \perp \mid A \vee B \mid \neg A$
True contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$
False contexts	$\Delta ::= \cdot \mid \Delta, u : A$
Values	$e ::= \langle \rangle \mid \langle e, e' \rangle \mid L e \mid R e \mid \text{not}(k)$ $\mid \mu u : A. c$
Continuations	$k ::= [] \mid [k, k'] \mid \text{fst } k \mid \text{snd } k \mid \text{not}(e)$ $\mid \mu x : A. c$
Contradictions	$c ::= \langle e \mid_A k \rangle$

$$\text{(No rule for } \perp \text{)} \qquad \frac{}{\Gamma; \Delta \vdash \langle \rangle : \top \text{ true}} \top P$$

$$\frac{\Gamma; \Delta \vdash e : A \text{ true} \quad \Gamma; \Delta \vdash e' : B \text{ true}}{\Gamma; \Delta \vdash \langle e, e' \rangle : A \wedge B \text{ true}} \wedge P$$

$$\frac{\Gamma; \Delta \vdash e : A \text{ true}}{\Gamma; \Delta \vdash L e : A \vee B \text{ true}} \vee P_1$$

$$\frac{\Gamma; \Delta \vdash e : B \text{ true}}{\Gamma; \Delta \vdash R e : A \vee B \text{ true}} \vee P_2$$

$$\frac{x : A \in \Gamma}{\Gamma; \Delta \vdash x : A \text{ true}} \text{HYP}$$

$$\frac{\Gamma; \Delta \vdash k : A \text{ false}}{\Gamma; \Delta \vdash \text{not}(k) : \neg A \text{ true}} \neg P$$

Continuations — Refutation Terms

$$\text{(No rule for } \top \text{)} \qquad \frac{}{\Gamma; \Delta \vdash [] : \perp \text{ false}} \perp R$$

$$\frac{\Gamma; \Delta \vdash k : A \text{ false} \quad \Gamma; \Delta \vdash k' : B \text{ false}}{\Gamma; \Delta \vdash [k, k'] : A \vee B \text{ false}} \vee R$$

$$\frac{\Gamma; \Delta \vdash k : A \text{ false}}{\Gamma; \Delta \vdash \text{fst } k : A \wedge B \text{ false}} \wedge R_1$$

$$\frac{\Gamma; \Delta \vdash k : B \text{ false}}{\Gamma; \Delta \vdash \text{snd } k : A \wedge B \text{ false}} \wedge R_2$$

$$\frac{x : A \in \Delta}{\Gamma; \Delta \vdash x : A \text{ false}} \text{HYP}$$

$$\frac{\Gamma; \Delta \vdash e : A \text{ true}}{\Gamma; \Delta \vdash \text{not}(e) : \neg A \text{ false}} \neg R$$

Contradictions

$$\frac{\Gamma; \Delta \vdash e : A \text{ true} \quad \Gamma; \Delta \vdash k : A \text{ false}}{\Gamma; \Delta \vdash \langle e \mid_A k \rangle \text{ contr}} \text{ CONTR}$$

$$\frac{\Gamma; \Delta, u : A \vdash c \text{ contr}}{\Gamma; \Delta \vdash \mu u : A. c : A \text{ true}}$$

$$\frac{\Gamma, x : A; \Delta \vdash c \text{ contr}}{\Gamma; \Delta \vdash \mu x : A. c : A \text{ false}}$$

$$\langle \langle e_1, e_2 \rangle \mid_{A \wedge B} \text{fst } k \rangle \mapsto \langle e_1 \mid_A k \rangle$$

$$\langle \langle e_1, e_2 \rangle \mid_{A \wedge B} \text{snd } k \rangle \mapsto \langle e_2 \mid_B k \rangle$$

$$\langle L e \mid_{A \vee B} [k_1, k_2] \rangle \mapsto \langle e \mid_A k_1 \rangle$$

$$\langle R e \mid_{A \vee B} [k_1, k_2] \rangle \mapsto \langle e \mid_B k_2 \rangle$$

$$\langle \text{not}(k) \mid_{\neg A} \text{not}(e) \rangle \mapsto \langle e \mid_A k \rangle$$

$$\langle \mu u : A. c \mid_A k \rangle \mapsto [k/u]c$$

$$\langle e \mid_A \mu x : A. c \rangle \mapsto [e/x]c$$

Preservation If $\cdot; \cdot \vdash c$ contr and $c \rightsquigarrow c'$ then $\cdot; \cdot \vdash c'$ contr.

Proof By *case analysis* on evaluation derivations!

(We don't even need induction!)

Type Preservation

$$\langle \langle e_1, e_2 \rangle \mid_{A \wedge B} \text{fst } k \rangle \rightsquigarrow \langle e_1 \mid_A k \rangle$$

Assumption

$$\frac{\overbrace{\Gamma; \Delta \vdash \langle e_1, e_2 \rangle : A \wedge B \text{ true}}^{(1)} \quad \overbrace{\Gamma; \Delta \vdash \text{fst } k : A \wedge B \text{ false}}^{(2)}}{\Gamma; \Delta \vdash \langle \langle e_1, e_2 \rangle \mid_{A \wedge B} \text{fst } k \rangle \text{ contr}}$$

Assumption

$$\frac{\overbrace{\Gamma; \Delta \vdash e_1 : A \text{ true}}^{(3)} \quad \Gamma; \Delta \vdash e_2 : B \text{ true}}{\Gamma; \Delta \vdash \langle e_1, e_2 \rangle : A \wedge B \text{ true}} \wedge P$$

Analysis of (1)

$$\frac{\overbrace{\Gamma; \Delta \vdash k : A \text{ false}}^{(4)}}{\Gamma; \Delta \vdash \text{fst } k : A \wedge B \text{ false}} \wedge R_1$$

Analysis of (2)

$$\therefore \cdot \vdash \langle e_1 \mid_A k \rangle \text{ contr}$$

By rule on (3), (4)

Embedding Classical Logic into Intuitionistic Logic

- Intuitionistic logic has a clean computational reading
- Classical logic *almost* has a clean computational reading
- Q: Is there any way to equip classical logic with computational meaning?
- A: Embed classical logic *into* intuitionistic logic

But Isn't Classical Logic an Extension of Constructive Logic?

- Normally, we think of classical logic as “constructive logic plus double-negation elimination”
- Surprisingly, classical logic is **also** a subset of intuitionistic logic!
- How can this work?

Double Negation Elimination

- The definition of negation is $\neg A \triangleq A \rightarrow \perp$
- We cannot prove there are any functions $((A \rightarrow 0) \rightarrow 0) \rightarrow A$

Proof sketch

- We can model each type as a set.
- Model 0 as the empty set.
- Model $A \rightarrow B$ as the set of functions from A to B .
- A mathematical function from A to B is a functional relation.
- If A has any elements at all, then $A \rightarrow 0$ has *no* elements.
- Then $(A \rightarrow 0) \rightarrow 0$ has no elements if A has any elements (and one otherwise).

Triple-Negation Elimination

In general, $\neg\neg X \rightarrow X$ is not derivable constructively. However, the following *is* derivable:

Lemma For all X , there is a function $\text{tne} : (\neg\neg\neg X) \rightarrow \neg X$

$$\frac{\begin{array}{c} \frac{\dots \vdash q : X \rightarrow p \quad \dots \vdash x : X}{k : \neg\neg\neg X, x : X, q : \neg X \vdash qx : p} \\ \dots \vdash k : \neg\neg\neg X \quad k : \neg\neg\neg X, x : X \vdash \lambda q. qx : \neg X \\ \hline k : \neg\neg\neg X, x : X \vdash k(\lambda q. qx) : p \\ \hline k : \neg\neg\neg X \vdash \lambda x. k(\lambda q. qa) : \neg X \\ \hline \cdot \vdash \underbrace{\lambda k. \lambda a. k(\lambda q. qa)}_{\text{tne}} : (\neg\neg\neg X) \rightarrow \neg X \end{array}}$$

- As a mathematical function space, $A \rightarrow 0$ has at most one element.
- From a programming perspective, this is terrible!
- Define “quasi-negation” $\sim X \triangleq X \rightarrow p$, where p is a fixed *arbitrary* type.

Triple-Negation Elimination for Quasi-Negation

Triple-negation elimination still holds for quasi-negations:

Lemma For all X , there is a function $\text{tne} : (\sim\sim\sim X) \rightarrow \sim X$

$$\frac{\begin{array}{c} \dots \vdash q : X \rightarrow p \quad \dots \vdash x : X \\ \hline k : \sim\sim\sim X, x : X, q : \sim X \vdash qx : p \\ \hline \dots \vdash k : \sim\sim\sim X \quad k : \sim\sim\sim X, x : X \vdash \lambda q. qx : \sim\sim X \\ \hline k : \sim\sim\sim X, x : X \vdash k(\lambda q. qx) : p \\ \hline k : \sim\sim\sim X \vdash \lambda x. k(\lambda q. qa) : \sim X \\ \hline \cdot \vdash \underbrace{\lambda k. \lambda a. k(\lambda q. qa)}_{\text{tne}} : (\sim\sim\sim X) \rightarrow \sim X \end{array}}$$

The Recipe for Embedding Classical Logic into Intuitionistic Logic

1. We define a translation function A°
2. It takes a classical type A and produces an intuitionistic type X .
3. A and X should be equivalent classically, but not necessarily constructively.
4. Then, we define a translation function from classical terms into intuitionistic terms.

Many Different Embeddings

- Many different translations of classical logic were discovered many times
 - Gerhard Gentzen and Kurt Gödel
 - Andrey Kolmogorov
 - Valery Glivenko
 - Sigekatu Kuroda
- The key property is to show that $\sim\sim A^\circ \rightarrow A^\circ$ holds.

The Kolmogorov Translation

Now, we can define another translation on types as follows:

$$\begin{aligned}\neg A^\bullet &= \sim\sim\sim A^\bullet \\ A \supset B^\bullet &= \sim\sim(A^\bullet \rightarrow B^\bullet) \\ \top^\bullet &= \sim\sim 1 \\ (A \wedge B)^\bullet &= \sim\sim(A^\bullet \times B^\bullet) \\ \perp^\bullet &= \sim\sim \perp \\ (A \vee B)^\bullet &= \sim\sim(A^\bullet + B^\bullet)\end{aligned}$$

- Uniformly stick a double-negation in front of each connective.
- Deriving $\sim\sim A^\bullet \rightarrow A^\bullet$ is particularly easy:
 - The **tne** term will always work!

The Embedding Theorem

Theorem Classical terms embed into intuitionistic terms:

1. If $\Gamma; \Delta \vdash e : A$ true then $\Gamma^\bullet, \sim\Delta^\bullet \vdash e^\bullet : A^\bullet$.
2. If $\Gamma; \Delta \vdash k : A$ false then $\Gamma^\bullet, \sim\Delta^\bullet \vdash k^\bullet : \sim A^\bullet$.
3. If $\Gamma; \Delta \vdash c$ contr then $\Gamma^\bullet, \sim\Delta \vdash c^\bullet : p$.

Proof By induction on derivations – for a suitable choice of translation function.

Implementing Classical Logic Axiomatically

- The proof theory of classical logic is elegant
- It is also very awkward to use:
 - Binding only arises from proof by contradiction
 - Difficult to write nested computations
 - Continuations/stacks are always explicit
- Functional languages make the stack implicit
- Can we make the continuations implicit?

The Typed Lambda Calculus with Continuations

Types	$X ::= 1 \mid X \times Y \mid 0 \mid X + Y \mid X \rightarrow Y \mid \neg X$
Terms	$e ::= x \mid \langle \rangle \mid \langle e, e \rangle \mid \text{fst } e \mid \text{snd } e$ $\mid \text{abort} \mid L e \mid R e \mid \text{case}(e, Lx \rightarrow e', Ry \rightarrow e'')$ $\mid \lambda x : X. e \mid e e'$ $\mid \text{throw}(e, e') \mid \text{letcont } x. e$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : X$

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

$$\frac{\Gamma \vdash e : X \quad \Gamma \vdash e' : Y}{\Gamma \vdash \langle e, e' \rangle : X \times Y} \text{xI}$$

$$\frac{\Gamma \vdash e : X \times Y}{\Gamma \vdash \text{fst } e : X} \text{xE}_1$$

$$\frac{\Gamma \vdash e : X \times Y}{\Gamma \vdash \text{snd } e : Y} \text{xE}_1$$

Functions and Variables

$$\frac{X : X \in \Gamma}{\Gamma \vdash x : X} \text{HYP}$$

$$\frac{\Gamma, x : X \vdash e : Y}{\Gamma \vdash \lambda x : X. e : X \rightarrow Y} \rightarrow I$$

$$\frac{\Gamma \vdash e : X \rightarrow Y \quad \Gamma \vdash e' : X}{\Gamma \vdash e e' : Y} \rightarrow E$$

Sums and the Empty Type

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash Le : X + Y} +I_1$$

$$\frac{\Gamma \vdash e : Y}{\Gamma \vdash Re : X + Y} +I_2$$

$$\frac{\Gamma \vdash e : X + Y \quad \Gamma, x : X \vdash e' : Z \quad \Gamma, y : Y \vdash e'' : Z}{\Gamma \vdash \text{case}(e, Lx \rightarrow e', Ry \rightarrow e'') : Z} +E$$

(no intro for 0)

$$\frac{\Gamma \vdash e : 0}{\Gamma \vdash \text{abort } e : Z} 0E$$

$$\frac{\Gamma, u : \neg X \vdash e : X}{\Gamma \vdash \text{letcont } u : \neg X. e : X} \text{CONT}$$

$$\frac{\Gamma \vdash e : \neg X \quad \Gamma \vdash e' : X}{\Gamma \vdash \text{throw}_Y(e, e') : Y} \text{THROW}$$

Examples

Double-negation elimination:

$$\text{dne}_X : \neg\neg X \rightarrow X$$

$$\text{dne}_X \triangleq \lambda k : \neg\neg X. \text{letcont } u : \neg X. \text{throw}(k, u)$$

The Excluded Middle:

$$t : X \vee \neg X$$

$$t \triangleq \text{letcont } u : \neg(X \vee \neg X).$$

$$\text{throw}(u, R(\text{letcont } q : \neg\neg X.$$

$$\text{throw}(u, L(\text{dne}_X q)))$$

Continuation-Passing Style (CPS) Translation

Type translation (almost the Kolmogorov translation):

$$\begin{aligned}\neg X^\circ &= \sim X^\circ \\ X \rightarrow Y^\circ &= (X^\bullet \rightarrow Y^\bullet) \\ 1^\circ &= 1 \\ (X \times Y)^\circ &= (X^\bullet \times Y^\bullet) \\ 0^\circ &= 0 \\ (X + Y)^\circ &= (X^\bullet + Y^\bullet) \\ X^\bullet &= \sim\sim X^\circ\end{aligned}$$

Translating contexts:

$$\begin{aligned}(\cdot)^\bullet &= \cdot \\ (\Gamma, x : A)^\bullet &= \Gamma^\bullet, x : A^\bullet\end{aligned}$$

The CPS Translation Theorem

Theorem If $\Gamma \vdash e : X$ then $\Gamma^\bullet \vdash e^\bullet : X^\bullet$.

Proof: By induction on derivations – we “just” need to define e^\bullet .

The CPS Translation

$$\begin{aligned}x^\bullet &= \lambda k : \sim X^\circ. x\ k \\ \langle \rangle^\bullet &= \lambda k : \sim 1. k\ \langle \rangle \\ \langle e_1, e_2 \rangle^\bullet &= \lambda k : \sim (X \times Y)^\circ. e_1^\bullet (\lambda x : X^\circ. e_2^\bullet (\lambda y : Y^\circ. k\ (x, y))) \\ (\text{fst } e)^\bullet &= \lambda k : \sim X^\circ. e^\bullet (\lambda p : (X \times Y)^\circ. k\ (\text{fst } p)) \\ (\text{snd } e)^\bullet &= \lambda k : \sim Y^\circ. e^\bullet (\lambda p : (X \times Y)^\circ. k\ (\text{snd } p)) \\ (L\ e)^\bullet &= \lambda k : \sim (X + Y)^\circ. e^\bullet (\lambda x : X^\circ. k\ (L\ x)) \\ (R\ e)^\bullet &= \lambda k : \sim (X + Y)^\circ. e^\bullet (\lambda y : Y^\circ. k\ (R\ y)) \\ \text{case}(e, L\ x \rightarrow e_1, R\ y \rightarrow e_2)^\bullet &= \lambda k : \sim Z^\circ. e^\bullet (\lambda v : (X + Y)^\circ. \text{case}(v, \\ &\quad L\ (x : X^\bullet) \rightarrow e_1^\bullet k \\ &\quad R\ (y : Y^\bullet) \rightarrow e_2^\bullet k)) \\ (\lambda x : X. e)^\bullet &= \lambda k : \sim (A \rightarrow B)^\circ. k\ (\lambda x : X^\bullet. e^\bullet) \\ (e_1\ e_2)^\bullet &= \lambda k : \sim Y^\circ. e_1^\bullet (\lambda f : (X \rightarrow Y)^\circ. f\ e_2^\bullet k))\end{aligned}$$

The CPS Translation for Continuations

$$(\text{letcont } u : \neg X. e)^{\bullet} = \lambda k : \sim X^{\circ}. [\text{dni}(k)/u](e^{\bullet})$$

$$\text{throw}(e_1, e_2)^{\bullet} = \lambda k : \sim Y^{\circ}. e_1^{\bullet} e_2^{\bullet}$$

$$\text{dni} : X \rightarrow \sim\sim X = \lambda x : X. \lambda k : \sim X. k x$$

- The rest of the CPS translation is bookkeeping to enable these two clauses to work!

Case $\Gamma \vdash \text{throw}(e_1, e_2) : Y$

$\Gamma \vdash \text{throw}(e_1, e_2) : X$	Assumption
$\Gamma \vdash e_1 : \neg X$	Subderivation
$\Gamma \vdash e_2 : X$	Subderivation
$\Gamma^\bullet \vdash e_1^\bullet : \neg X^\bullet$	Induction
$\Gamma^\bullet \vdash e_2^\bullet : X^\bullet$	Induction
$\neg X^\bullet = \sim X^\bullet$	Definition
$Y^\bullet = \sim \sim Y^\circ$	Definition
$\Gamma^\bullet \vdash e_1^\bullet e_2^\bullet : p$	By app rule
$\Gamma^\bullet, k : \sim Y \vdash e_1^\bullet e_2^\bullet : p$	By weakening
$\Gamma^\bullet \vdash \lambda k : \sim Y^\circ. e_1^\bullet e_2^\bullet : \sim \sim Y^\circ$	By lambda
$\Gamma^\bullet \vdash \lambda k : \sim Y^\circ. e_1^\bullet e_2^\bullet : Y^\bullet$	By above

Questions

1. Give the embedding (ie, the e° and k° translations) of classical into intuitionistic logic for the Gödel-Gentzen translation (see next slide).
2. Using the intuitionistic calculus extended with continuations, give a typed term proving *Peirce's law*:

$$((X \rightarrow Y) \rightarrow X) \rightarrow X$$

The Gödel-Gentzen Translation

Now, we can define a translation on types as follows:

$$\neg A^\circ = \sim A^\circ$$

$$\top^\circ = 1$$

$$(A \wedge B)^\circ = A^\circ \times B^\circ$$

$$\perp^\circ = p$$

$$(A \vee B)^\circ = \sim(\sim A^\circ \times \sim B^\circ)$$

- This uses a different de Morgan duality for disjunction

Type Systems

Lecture 11: Applications of Continuations, and Dependent Types

Neel Krishnaswami
University of Cambridge

Applications of Continuations

We have seen that:

- Classical logic has a beautiful inference system
- Embeds into constructive logic via double-negation translations
- This yields an operational interpretation
- What can we program with continuations?

The Typed Lambda Calculus with Continuations

Types	$X ::= 1 \mid X \times Y \mid 0 \mid X + Y \mid X \rightarrow Y \mid \neg X$
Terms	$e ::= x \mid \langle \rangle \mid \langle e, e \rangle \mid \text{fst } e \mid \text{snd } e$ $\mid \text{abort} \mid L e \mid R e \mid \text{case}(e, Lx \rightarrow e', Ry \rightarrow e'')$ $\mid \lambda x : X. e \mid e e'$ $\mid \text{throw}(e, e') \mid \text{letcont } x. e$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : X$

Continuation Typing

$$\frac{\Gamma, u : \neg X \vdash e : X}{\Gamma \vdash \text{letcont } u : \neg X. e : X} \text{CONT}$$

$$\frac{\Gamma \vdash e : \neg X \quad \Gamma \vdash e' : X}{\Gamma \vdash \text{throw}_Y(e, e') : Y} \text{THROW}$$

Continuation API in Standard ML

```
1 signature CONT = sig
2   type 'a cont
3   val callcc : ('a cont -> 'a) -> 'a
4   val throw : 'a cont -> 'a -> 'b
5 end
```

SML	Type Theory
'a cont	$\neg A$
throw k v	throw(k, v)
callcc (fn x => e)	letcont $x : \neg X. e$

An Inefficient Program

```
1  val mul : int list -> int
2
3  fun mul [] = 1
4  | mul (n :: ns) = n * mul ns
```

- This function multiplies a list of integers
- If 0 occurs in the list, the whole result is 0

A Less Inefficient Program

```
1  val mul' : int list -> int
2
3  fun mul' [] = 1
4    | mul' (0 :: ns) = 0
5    | mul' (n :: ns) = n * mul ns
```

- This function multiplies a list of integers
- If 0 occurs in the list, it immediately returns 0
 - `mul' [0,1,2,3,4,5,6,7,8,9]` will immediately return
 - `mul' [1,2,3,4,5,6,7,8,9,0]` will multiply by 0, 9 times

Even Less Inefficiency, via Escape Continuations

```
1  val loop = fn : int cont -> int list -> int
2  fun loop return [] = 1
3    | loop return (0 :: ns) = throw return 0
4    | loop return (n :: ns) = n * loop return ns
5
6  val mul_fast : int list -> int
7  fun mul_fast ns = callcc (fn ret => loop ret ns)
```

- `loop` multiplies its arguments, unless it hits 0
- In that case, it throws 0 to its continuation
- `mul_fast` captures its continuation, and passes it to `loop`
- So if `loop` finds 0, it does no multiplications!

McCarthy's amb Primitive

- In 1961, John McCarthy (inventor of Lisp) proposed a language construct **amb**
- This was an operator for *angelic nondeterminism*

```
1      let val x = amb [1,2,3]
2          val y = amb [4,5,6]
3      in
4          assert (x * y = 10);
5          (x, y)
6      end
7      (* Returns (2,5) *)
```

- Does search to find a succesful assignment of values
- Can be implemented via backtracking – *using continuations*

The AMB signature

```
1  signature AMB = sig
2      (* Internal implementation *)
3      val stack : int option cont list ref
4      val fail : unit -> 'a
5
6      (* External API *)
7      exception AmbFail
8      val assert : bool -> unit
9      val amb : int list -> int
10 end
```

Implementation, Part 1

```
1  exception AmbFail
2  val stack
3      : int option cont list ref
4      = ref []
5
6  fun fail () =
7      case !stack of
8          []          => raise AmbFail
9      | (k :: ks) => (stack := ks;
10                     throw k NONE)
11
12  fun assert b =
13      if b then () else fail()
```

- `AmbFail` is the failure exception for unsatisfiable computations
- `stack` is a stack of backtrack points
- `fail` grabs the topmost backtrack point, and resumes execution there
- `assert` backtracks if its condition is false

Implementation, Part 2

```
1 fun amb [] = fail ()
2 | amb (x :: xs) =
3   let fun next y k =
4     (stack := k :: !stack;
5      SOME y)
6   in
7     case callcc (next x) of
8       SOME v => v
9     | NONE => amb xs
10  end
```

- `amb []` backtracks immediately!
- `next y k` pushes `k` onto the backtrack stack, and returns `SOME y`
- Save the backtrack point, then see
 - if we immediately return, or
 - if we are resuming from a backtrack point and must try the other values

Examples

```
1  fun test2() =  
2      let val x = amb [1,2,3,4,5,6]  
3          val y = amb [1,2,3,4,5,6]  
4          val z = amb [1,2,3,4,5,6]  
5      in  
6          assert(x + y + z >= 13);  
7          assert(x > 1);  
8          assert(y > 1);  
9          assert(z > 1);  
10         (x, y, z)  
11     end  
12  
13 (* Returns (2, 5, 6) *)
```

Conclusions

- **amb** required the *combination* of state and continuations
- Theorem of Andrzej Filinski that this is **universal**
- Any “definable monadic effect” can be expressed as a combination of state and first-class control:
 - Exceptions
 - Green threads
 - Coroutines/generators
 - Random number generation
 - Nondeterminism

Dependent Types

The Curry Howard Correspondence

Logic	Language
Intuitionistic Propositional Logic	STLC
Classical Propositional Logic	STLC + 1 st class continuations
Pure Second-Order Logic	System F

- Each logical system has a corresponding computational system
- One thing is missing, however
- Mathematics uses quantification over *individual elements*
- Eg, $\forall x, y, z, n \in \mathbb{N}. \text{ if } n > 2 \text{ then } x^n + y^n \neq z^n$

$$\frac{}{\Gamma \vdash z : \mathbb{N}} \text{NI}_z \qquad \frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash s(e) : \mathbb{N}} \text{NI}_s$$
$$\frac{\Gamma \vdash e_0 : \mathbb{N} \quad \Gamma \vdash e_1 : X \quad \Gamma, x : X \vdash e_2 : X}{\Gamma \vdash \text{iter}(e_0, z \rightarrow e_1, s(x) \rightarrow e_2) : X} \text{NE}$$

- \mathbb{N} is the type of natural numbers
- Logically, it is equivalent to the unit type:
 - $(\lambda x : 1. z) : 1 \rightarrow \mathbb{N}$
 - $(\lambda x : \mathbb{N}. \langle \rangle) : \mathbb{N} \rightarrow 1$
- Language of types has no way of distinguishing z from $s(z)$.

Dependent Types

- Language of types has no way of distinguishing z from $s(z)$.
- So let's fix that: let types refer to values
- Type grammar and term grammar mutually recursive
- Huge gain in expressive power

An Introduction to Agda

- Much of earlier course leaned on prior knowledge of ML for motivation
- Before we get to the theory of dependent types, let's look at an implementation
- Agda: a dependently-typed functional programming language
- <http://wiki.portal.chalmers.se/agda/pmwiki.php>

Agda: Basic Datatypes

```
1 data Bool : Set where
```

```
2   true  : Bool
```

```
3   false : Bool
```

```
4  
5 not : Bool → Bool
```

```
6 not true = false
```

```
7 not false = true
```

- Datatype declarations give constructors and their types
- Functions given type signature, and clausal definition

Agda: Inductive Datatypes

```
1  data Nat : Set where
2    z : Nat
3    s : Nat → Nat
4
5  _+_ : Nat → Nat → Nat
6    z + m = m
7    s n + m = s (n + m)
8
9  _*_ : Nat → Nat → Nat
10   z × m = z
11   s n × m = m + (n × m)
```

- Datatype constructors can be recursive
- Functions can be recursive, but checked for termination

Agda: Polymorphic Datatypes

```
1 data List (A : Set) : Set where
2   [] : List A
3   _,_ : A → List A → List A
4
5 app : (A : Set) → List A → List A → List A
6 app A [] ys = ys
7 app A (x , xs) ys = x , app A xs ys
8
9 app' : {A : Set} → List A → List A → List A
10 app' [] ys = ys
11 app' (x , xs) ys = (x , app' xs ys)
```

- Datatypes can be polymorphic
- `app` has F-style explicit polymorphism
- `app'` has implicit, inferred polymorphism

Agda: Indexed Datatypes

```
1 data Vec (A : Set) : Nat → Set where
```

```
2   [] : Vec A z
```

```
3   _,_ : {n : Nat} → A → Vec A n → Vec A (s n)
```

- This is a *length-indexed list*
- Cons takes a head and a list of length n , and produces a list of length $n + 1$
- The empty list has a length of 0

Agda: Indexed Datatypes

```
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 head : {A : Set} → {n : Nat} → Vec A (s n) → A
6 head (x , xs) = x
```

- `head` takes a list of length > 0 , and returns an element
- No `[]` pattern present
- Not needed for coverage checking!
- Note that `{n:Nat}` is *also* an implicit (inferred) argument

Agda: Indexed Datatypes

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

- Note the appearance of $n + m$ in the type
- This type guarantees that appending two vectors yields a vector whose length is the sum of the two

Agda: Indexed Datatypes

```
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 -- Won't typecheck!
6 app : {A : Set} → {n m : Nat} →
7       Vec A n → Vec A m → Vec A (n + m)
8 app [] ys = ys
9 app (x , xs) ys = app xs ys
```

- We forgot to cons x here
- This program **won't type check!**
- Static typechecking ensures a runtime guarantee

The Identity Type

```
data _≡_ {A : Set} (a : A) : A → Set where  
  refl : a ≡ a
```

- $a \equiv b$ is the type of proofs that a and b are equal
- The constructor **refl** says that a term a is equal to itself
- Equalities arising from evaluation are automatic
- Other equalities have to be **proved**

An Automatic Theorem

```
data _≡_ {A : Set} (a : A) : A → Set where  
  refl : a ≡ a
```

```
_+_ : Nat → Nat → Nat
```

```
z + m = m
```

```
s n + m = s (n + m)
```

```
z-+-left-unit : (n : Nat) → (z + n) ≡ n
```

```
z-+-left-unit n = refl
```

- $z + n$ evaluates to n
- So Agda considers these two terms to be identical

A Manual Theorem

```
data _≡_ {A : Set} (a : A) : A → Set where  
  refl : a ≡ a
```

```
cong : {A B : Set} → {a a' : A} →  
      (f : A → B) → (a ≡ a') → (f a ≡ f a')  
cong f refl = refl
```

```
z-+-right-unit : (n : Nat) → (n + z) ≡ n  
z-+-right-unit z = refl  
z-+-right-unit (s n) = cong s (z-+-right-unit n)
```

- We prove the right unit law inductively
 - Note that *inductive proofs are recursive functions*
 - To do this, we need to show that equality is a congruence

The Equality Toolkit

```
data _≡_ {A : Set} (a : A) : A → Set where  
  refl : a ≡ a
```

```
sym : {A : Set} → {a b : A} →  
      a ≡ b → b ≡ a  
sym refl = refl
```

```
trans : {A : Set} → {a b c : A} →  
        a ≡ b → b ≡ c → a ≡ c  
trans refl refl = refl
```

```
cong : {A B : Set} → {a a' : A} →  
       (f : A → B) → (a ≡ a') → (f a ≡ f a')  
cong f refl = refl
```

- An *equivalence relation* is a reflexive, symmetric transitive relation
- Equality is congruent with everything

Commutativity of Addition

```
z-+-right : (n : Nat) → (n + z) ≡ n
```

```
z-+-right z = refl
```

```
z-+-right (s n) =  
  cong s (z-+-right n)
```

```
s-+-right : (n m : Nat) →  
  (s (n + m)) ≡ (n + (s m))
```

```
s-+-right z m = refl
```

```
s-+-right (s n) m = cong s (s-+-right n m)
```

```
+--comm : (i j : Nat) → (i + j) ≡ (j + i)
```

```
+--comm z j = z-+-right j
```

```
+--comm (s i) j = trans p2 p3
```

```
  where p1 : (i + j) ≡ (j + i)
```

```
        p1 = +--comm i j
```

```
        p2 : (s (i + j)) ≡ (s (j + i))
```

```
        p2 = cong s p1
```

```
        p3 : (s (j + i)) ≡ (j + (s i))
```

```
        p3 = s-+-right j i
```

- First we prove that adding zero on the right does nothing
- Then we prove that successor commutes with addition
- Then we use these two facts to inductively prove commutativity of addition

Conclusion

- Dependent types permit referring to program terms in types
- This enables writing types which state very precise properties of programs
 - Eg, equality is expressible as a type
- Writing a program becomes the same as proving it correct
- This is hard, like learning to program again!
- But also extremely fun...

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
| $e e'$ | $\lambda x : A. e$ | $\prod x : A. B$
| $\text{refl } e$ | $\text{subst}[x : A. B](e, e')$ | $(e = e' : A)$

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

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

Introduction

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

(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

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

(Equality elimination not the most general form!)

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

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

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

$$\frac{}{\Gamma \vdash 1 \equiv 1 \text{ type}} \qquad \frac{\Gamma \vdash A \equiv X \text{ type} \quad \Gamma, x : A \vdash B \equiv Y \text{ type}}{\Gamma \vdash \Pi x : A. B \equiv \Pi x : X. Y \text{ type}}$$
$$\frac{\Gamma \vdash e'_2 : A' \quad \Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A \quad \Gamma \vdash e'_1 : A' \quad \Gamma \vdash A \equiv A' \text{ type} \quad \Gamma \vdash e_1 \equiv e'_1 : A \quad \Gamma \vdash e_2 \equiv e'_2 : A}{\Gamma \vdash (e_1 = e_2 : A) \equiv (e'_1 = e'_2 : A') \text{ type}}$$

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

$$\begin{array}{c} \frac{}{\Gamma \vdash \langle \rangle \equiv \langle \rangle : 1} \quad \frac{x : A \in \Gamma}{\Gamma \vdash x \equiv x : A} \quad \frac{\Gamma \vdash e_1 \equiv e'_1 : \Pi x : A. B \quad \Gamma \vdash e_2 \equiv e'_2 : A}{\Gamma \vdash e_1 e_2 \equiv e'_1 e'_2 : [e_1/x]B} \\[10pt] \frac{\Gamma \vdash A \equiv A' \text{ type} \quad \Gamma, x : A \vdash e \equiv e' : B}{\Gamma \vdash \lambda x : A. e \equiv \lambda x : A'. e' : \Pi x : A. B} \quad \frac{\Gamma \vdash e \equiv e' : A}{\Gamma \vdash \text{refl } e \equiv \text{refl } e' : (e = e : A)} \\[10pt] \frac{\Gamma \vdash A \equiv A' \text{ type} \quad \Gamma, x : A \vdash B \equiv B' \text{ type} \quad \Gamma \vdash e_1 \equiv e'_1 : (e = e' : A) \quad \Gamma \vdash e_2 \equiv e'_2 : [e/x]B}{\Gamma \vdash \text{subst}[x : A. B](e_1, e_2) \equiv \text{subst}[x : A'. B'](e'_1, e'_2) : [e'/x]B} \end{array}$$

Judgemental Equality: Conversion rules

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

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

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

$$\frac{}{\cdot \text{ ok}}$$

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

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

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
    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