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

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

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

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \text{bool} \mid \mathbb{N} \\
\text{Terms} & e & ::= & \text{true} \mid \text{false} \mid n \mid e \leq e \mid e + e \mid e \wedge e
\end{array}
$$

- 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

$$\frac{}{n : \mathbb{N}}\ \textsc{Num} \qquad \frac{}{\mathsf{true} : \mathsf{bool}}\ \textsc{True} \qquad \frac{}{\mathsf{false} : \mathsf{bool}}\ \textsc{False}$$

$$\frac{e : \mathbb{N} \qquad e' : \mathbb{N}}{e + e' : \mathbb{N}}\ \textsc{Plus} \qquad \frac{e : \mathsf{bool} \qquad e' : \mathsf{bool}}{e \wedge e' : \mathsf{bool}}\ \textsc{And}$$

$$\frac{e : \mathbb{N} \qquad e' : \mathbb{N}}{e \leq e' : \mathsf{bool}}\ \textsc{Leq}$$

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

An Example Derivation Tree

$$\dfrac{\dfrac{\dfrac{}{3 : \mathbb{N}}\ \textsc{Num} \qquad \dfrac{}{4 : \mathbb{N}}\ \textsc{Num}}{3 + 4 : \mathbb{N}}\ \textsc{Plus} \qquad \dfrac{}{5 : \mathbb{N}}\ \textsc{Num}}{3 + 4 \leq 5 : \mathrm{bool}}\ \textsc{Leq}$$

Adding Variables

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \text{bool} \mid \mathbb{N} \\
\text{Terms} & e & ::= & \ldots \mid x \mid \text{let } x = e \text{ in } e'
\end{array}
$$

- 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

$$\text{Contexts} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : \tau$$

$$\frac{}{\Gamma \vdash n : \mathbb{N}}\ \textsc{Num} \qquad \frac{}{\Gamma \vdash \mathrm{true} : \mathrm{bool}}\ \textsc{True} \qquad \frac{}{\Gamma \vdash \mathrm{false} : \mathrm{bool}}\ \textsc{False}$$

$$\frac{\Gamma \vdash e : \mathbb{N} \qquad \Gamma \vdash e' : \mathbb{N}}{\Gamma \vdash e + e' : \mathbb{N}}\ \textsc{Plus}$$

$$\frac{\Gamma \vdash e : \mathrm{bool} \qquad \Gamma \vdash e' : \mathrm{bool}}{\Gamma \vdash e \wedge e' : \mathrm{bool}}\ \textsc{And} \qquad \frac{\Gamma \vdash e : \mathbb{N} \qquad \Gamma \vdash e' : \mathbb{N}}{\Gamma \vdash e \leq e' : \mathrm{bool}}\ \textsc{Leq}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \mathrm{let}\ x = e\ \mathrm{in}\ e' : \tau'}\ \textsc{Let}$$

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

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

$(*)$ 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 $\Gamma \vdash [e/x]e' : \tau'$.
   Subsituting 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.

Proving Weakening, 1/4

$$\frac{}{\Gamma, \Gamma' \vdash n : \mathbb{N}}\ \textsc{Num} \qquad \text{By assumption}$$

$$\frac{}{\Gamma, x : \tau'', \Gamma' \vdash n : \mathbb{N}}\ \textsc{Num} \qquad \text{By rule } \textsc{Num}$$

- Similarly for True and False rules

Proving Weakening, 2/4

$$\dfrac{\Gamma, \Gamma' \vdash e_1 : \mathbb{N} \qquad \Gamma, \Gamma' \vdash e_2 : \mathbb{N}}{\Gamma, \Gamma' \vdash e_1 + e_2 : \mathbb{N}}\ \textsc{Plus}$$ 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

$$\dfrac{\Gamma, \Gamma' \vdash e_1 : \tau_1 \qquad \Gamma, \Gamma', z : \tau_1 \vdash e_2 : \tau_2}{\Gamma, \Gamma' \vdash \text{let } z = e_1 \text{ in } e_2 : \tau_2} \textsc{Let} \qquad \text{By assumption}$$

$\Gamma, \Gamma' \vdash e_1 : \tau_1$	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
$\Gamma, x : \tau'', \overbrace{\Gamma', z : \tau_1}^{\text{Extended context}} \vdash e_2 : \tau_2$	Induction on subderivation 2
$\Gamma, x : \tau'', \Gamma' \vdash \text{let } z = e_1 \text{ in } e_2 : \tau_2$	By rule Let

Proving Weakening, 4/4

$$\frac{z : \tau \in \Gamma, \Gamma'}{\Gamma, \Gamma' \vdash z : \tau}\ \textsc{Var} \qquad \text{By assumption}$$

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

Proving Exchange, 1/4

$$\frac{}{\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash n : \mathbb{N}}\ \textsc{Num} \qquad \text{By assumption}$$

$$\frac{}{\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash n : \mathbb{N}}\ \textsc{Num} \qquad \text{By rule } \textsc{Num}$$

- Similarly for True and False rules

Proving Exchange, 2/4

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

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

$$\dfrac{\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 : \tau' \qquad \Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma', z : \tau' \vdash e_2 : \tau_2}{\Gamma, \Gamma' \vdash \text{let } z = e_1 \text{ in } e_2 : \tau_2} \textsc{Let} \qquad \text{By assumption}$$

$\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 : \tau'$ Subderivation 1

$\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma', z : \tau' \vdash e_2 : \tau_2$ Subderivation 2

$\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash e_1 : \tau_1$ Induction on s.d. 1

$\Gamma, x_2 : \tau_2, x_1 : \tau_1, \overbrace{\Gamma', z : \tau_1}^{\text{Extended context}} \vdash e_2 : \mathbb{N}$ Induction on s.d. 2

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

Proving Exchange, 4/4

$$\frac{z : \tau \in \Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma'}{\Gamma, \Gamma' \vdash z : \tau}\ \textsc{Var}$$

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.

Substitution 1/4

$$\dfrac{}{\Gamma, x : \tau \vdash n : \mathbb{N}}\ \textsc{Num} \qquad \text{By assumption}$$

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

$$\Gamma \vdash n : \mathbb{N} \qquad \text{By rule } \textsc{Num}$$

$$\Gamma \vdash [e/x]n : \mathbb{N} \qquad \text{Def. of substitution}$$

- Similarly for True and False rules

Proving Substitution, 2/4

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

$$\dfrac{\Gamma, x : \tau \vdash e_1 : \tau' \qquad \Gamma, x : \tau, z : \tau' \vdash e_2 : \tau_2}{\Gamma, x : \tau \vdash \text{let } z = e_1 \text{ in } e_2 : \tau_2}\ \textsc{Let}$$ 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

Proving Substitution, 4a/4

$$\frac{z : \tau' \in \Gamma, x : \tau}{\Gamma, x : \tau \vdash z : \tau'}\ \textsc{Var}$$ By assumption

$\Gamma \vdash e : \tau$ By assumption

Case $x = z$:

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

Proving Substitution, 4b/4

$$\frac{z : \tau' \in \Gamma, x : \tau}{\Gamma, x : \tau \vdash z : \tau'}\ \textsc{Var}$$ By assumption

$\Gamma \vdash e : \tau$ By assumption

Case $x \neq z$:

$z : \tau' \in \Gamma$	since $x \neq z$ and $z : \tau' \in \Gamma, x : \tau$
$\Gamma, z : \tau' \vdash z : \tau'$	By rule Var
$\Gamma, z : \tau' \vdash [e/x]z : \tau'$	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 $e \leadsto e'$
- Pronounced as "e steps to e'"

An operational semantics

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

$$\frac{e_1 \rightsquigarrow e_1'}{e_1 \wedge e_2 \rightsquigarrow e_1' \wedge e_2}\ \textsc{AndCong} \qquad \frac{}{\text{true} \wedge e \rightsquigarrow e}\ \textsc{AndTrue}$$

$$\frac{}{\text{false} \wedge e \rightsquigarrow \text{false}}\ \textsc{AndFalse}$$

(similar rules for $\leq$ and $+$)

$$\frac{e_1 \rightsquigarrow e_1'}{\text{let } z = e_1 \text{ in } e_2 \rightsquigarrow \text{let } z = e_1' \text{ in } e_2}\ \textsc{LetCong}$$

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

Reduction Sequences

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

Successful sequence	Stuck sequence
$(3+4) \leq (2+3)$	$(3+4) \wedge (2+3)$
$\leadsto \; 7 \leq (2+3)$	$\leadsto \; 7 \wedge (2+3)$
$\leadsto \; 7 \leq 5$	$\leadsto$???
$\leadsto$ 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 $e \leadsto e'$.
2. (Preservation) If $\cdot \vdash e : \tau$ and $e \leadsto e'$ 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).

Proving Progress

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

- 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.

Progress: Values

$$\frac{}{\cdot \vdash n : \mathbb{N}}\ \textsc{Num}$$ By assumption

n is a value — Def. of value grammar

Similarly for boolean literals...

Progress: Let-bindings

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

By assumption: (1)

$\cdot \vdash e_1 : \tau$	Subderivation of (1): (2)
$x : \tau \vdash e_2 : \tau'$	Subderivation of (1): (3)
$e_1 \rightsquigarrow e_1'$ or e_1 value	Induction on (2)
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
Case e_1 value :	
$\text{let } x = e_1 \text{ in } e_2 \rightsquigarrow [e_1/x]e_2$	By rule LetStep

Type Preservation

(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

$$\frac{e_1 \leadsto e_1'}{\text{let } x = e_1 \text{ in } e_2 \leadsto \text{let } x = e_1' \text{ in } e_2}$$ By assumption: (1)

$$\frac{\cdot \vdash e_1 : \tau \qquad x : \tau \vdash e_2 : \tau'}{\cdot \vdash \text{let } x = e_1 \text{ in } e_2 : \tau'}$$ By assumption: (2)

$e_1 \leadsto e_1'$	Subderivation of (1): (3)
$\cdot \vdash e_1 : \tau$	Subderivation of (2): (4)
$x : \tau \vdash e_2 : \tau'$	Subderivation of (2): (5)
$\cdot \vdash e_1' : \tau$	Induction on (3), (4): (6)
$\cdot \vdash \text{let } x = e_1' \text{ in } e_2 : \tau'$	Rule Let on (6), (4)

Type Preservation: Let Bindings 2

$$\frac{}{\text{let } x = v_1 \text{ in } e_2 \rightsquigarrow [v_1/x]e_2} \qquad \text{By assumption: (1)}$$

$$\frac{\cdot \vdash v_1 : \tau \qquad x : \tau \vdash e_2 : \tau'}{\cdot \vdash \text{let } x = v_1 \text{ in } e_2 : \tau'} \qquad \text{By assumption: (2)}$$

$\cdot \vdash v_1 : \tau$ 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)

Conclusion

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

Exercises

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 <u>theorems and proofs</u> 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 <u>natural deduction</u>
- "Natural" because the proof style is natural (with a little squinting)

Natural Deduction: Propositional Logic

What are propositions?

- $\top$ is a proposition
- $P \wedge Q$ is a proposition, if P and Q are propositions
- $\bot$ 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, $P(x)$").

Judgements

- Some claims follow (e.g. $P \wedge Q \supset Q \wedge P$).
- Some claims don't. (e.g., $\top \supset \bot$)
- 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

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

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

$$\frac{P \wedge Q \text{ true}}{P \text{ true}} \wedge\text{E}_1 \qquad \frac{P \wedge Q \text{ true}}{Q \text{ true}} \wedge\text{E}_2$$

Implication

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

$$\frac{P \in \Psi}{\Psi \vdash P \text{ true}}\ \textsc{Hyp} \qquad\qquad \frac{\Psi, P \vdash Q \text{ true}}{\Psi \vdash P \supset Q \text{ true}}\ \supset\mathrm{I}$$

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

Disjunction and Falsehood

$$\frac{\Psi \vdash P \text{ true}}{\Psi \vdash P \vee Q \text{ true}} \vee\mathrm{I}_1 \qquad \frac{\Psi \vdash Q \text{ true}}{\Psi \vdash P \vee Q \text{ true}} \vee\mathrm{I}_2$$

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

(no intro for $\perp$)

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

Example

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{(P \vee Q) \supset R, P \vdash (P \vee Q) \supset R \text{ true}}
      \quad
      \dfrac{\dfrac{}{(P \vee Q) \supset R, P \vdash P \text{ true}}}{(P \vee Q) \supset R, P \vdash P \vee Q \text{ true}}
    }{(P \vee Q) \supset R, P \vdash R \text{ true}}
  }{(P \vee Q) \supset R \vdash P \supset R \text{ true}}
  \quad \ldots
}{
  \dfrac{(P \vee Q) \supset R \vdash (P \supset R) \wedge (Q \supset R) \text{ true}}{\cdot \vdash ((P \vee Q) \supset R) \supset ((P \supset R) \wedge (Q \supset R)) \text{ true}}
}
$$

$$
\begin{array}{llll}
\text{Types} & X & ::= & 1 \mid X \times Y \mid 0 \mid X + Y \mid X \to Y \\
\text{Terms} & e & ::= & x \mid \langle\rangle \mid \langle e, e\rangle \mid \mathsf{fst}\, e \mid \mathsf{snd}\, e \\
& & \mid & \mathsf{abort} \mid \mathsf{L}\, e \mid \mathsf{R}\, e \mid \mathsf{case}(e, \mathsf{L}\, x \to e', \mathsf{R}\, y \to e'') \\
& & \mid & \lambda x : X.\, e \mid e\, e' \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : X
\end{array}
$$

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

Units and Pairs

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

$$\frac{\Gamma \vdash e : X \qquad \Gamma \vdash e' : Y}{\Gamma \vdash \left\langle e, e' \right\rangle : X \times Y}\ \times\mathsf{I}$$

$$\frac{\Gamma \vdash e : X \times Y}{\Gamma \vdash \mathsf{fst}\, e : X}\ \times\mathsf{E}_1 \qquad\qquad \frac{\Gamma \vdash e : X \times Y}{\Gamma \vdash \mathsf{snd}\, e : Y}\ \times\mathsf{E}_2$$

Functions and Variables

$$\frac{x : X \in \Gamma}{\Gamma \vdash x : X}\ \textsc{Hyp} \qquad\qquad \frac{\Gamma, x : X \vdash e : Y}{\Gamma \vdash \lambda x : X.\, e : X \to Y}\ {\to}\mathrm{I}$$

$$\frac{\Gamma \vdash e : X \to Y \qquad \Gamma \vdash e' : X}{\Gamma \vdash e\, e' : Y}\ {\to}\mathrm{E}$$

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash \mathsf{L}\, e : X + Y}\ +\mathsf{I}_1 \qquad\qquad \frac{\Gamma \vdash e : Y}{\Gamma \vdash \mathsf{R}\, e : X + Y}\ +\mathsf{I}_2$$

$$\frac{\Gamma \vdash e : X + Y \qquad \Gamma, x : X \vdash e' : Z \qquad \Gamma, y : Y \vdash e'' : Z}{\Gamma \vdash \mathsf{case}(e, \mathsf{L}\, x \to e', \mathsf{R}\, y \to e'') : Z}\ +\mathsf{E}$$

(no intro for 0)

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

Example

$$\lambda f : (X + Y) \to Z.\, \langle \lambda x : X.\, f\,(\mathsf{L}\, x), \lambda y : Y.\, f\,(\mathsf{R}\, y) \rangle$$
$$:$$
$$((X + Y) \to Z) \to (X \to Z) \times (Y \to Z)$$

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

$$\text{Values} \quad v \quad ::= \quad \langle\rangle \;\mid\; \langle v, v' \rangle \;\mid\; \lambda x : A.\, e \;\mid\; \mathsf{L}\, v \;\mid\; \mathsf{R}\, v$$

The transition relation is $e \leadsto e'$, pronounced "e steps to e'".

Operational Semantics: Units and Pairs

(no rules for unit)

$$\frac{e_1 \rightsquigarrow e_1'}{\langle e_1, e_2 \rangle \rightsquigarrow \langle e_1', e_2 \rangle} \qquad \frac{e_2 \rightsquigarrow e_2'}{\langle v_1, e_2 \rangle \rightsquigarrow \langle v_1, e_2' \rangle}$$

$$\frac{}{\mathsf{fst}\, \langle v_1, v_2 \rangle \rightsquigarrow v_1} \qquad \frac{}{\mathsf{snd}\, \langle v_1, v_2 \rangle \rightsquigarrow v_2}$$

$$\frac{e \rightsquigarrow e'}{\mathsf{fst}\, e \rightsquigarrow \mathsf{fst}\, e'} \qquad \frac{e \rightsquigarrow e'}{\mathsf{snd}\, e \rightsquigarrow \mathsf{snd}\, e'}$$

Operational Semantics: Void and Sums

$$\frac{e \leadsto e'}{\mathsf{abort}\, e \leadsto \mathsf{abort}\, e'}$$

$$\frac{e \leadsto e'}{\mathsf{L}\, e \leadsto \mathsf{L}\, e'} \qquad \frac{e \leadsto e'}{\mathsf{R}\, e \leadsto \mathsf{R}\, e'}$$

$$\frac{e \leadsto e'}{\mathsf{case}(e, \mathsf{L}\, x \to e_1, \mathsf{R}\, y \to e_2) \leadsto \mathsf{case}(e', \mathsf{L}\, x \to e_1, \mathsf{R}\, y \to e_2)}$$

$$\frac{}{\mathsf{case}(\mathsf{L}\, v, \mathsf{L}\, x \to e_1, \mathsf{R}\, y \to e_2) \leadsto [v/x]e_1}$$

$$\frac{}{\mathsf{case}(\mathsf{R}\, v, \mathsf{L}\, x \to e_1, \mathsf{R}\, y \to e_2) \leadsto [v/y]e_2}$$

Operational Semantics: Functions

$$\frac{e_1 \leadsto e_1'}{e_1\, e_2 \leadsto e_1'\, e_2} \qquad \frac{e_2 \leadsto e_2'}{v_1\, e_2 \leadsto v_1\, e_2'}$$

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

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 $e \leadsto e'$.
5. (Preservation) If $\cdot \vdash e : X$ and $e \leadsto e'$, 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
$\dfrac{e_1 \rightsquigarrow e_1'}{\langle e_1, e_2 \rangle \rightsquigarrow \langle e_1', e_2 \rangle}$	$\dfrac{}{\mathsf{fst}\,\langle v_1, v_2 \rangle \rightsquigarrow v_1}$
$\dfrac{e_2 \rightsquigarrow e_2'}{v_1\, e_2 \rightsquigarrow v_1\, e_2'}$	$\dfrac{}{(\lambda x : X.\, e)\, v \rightsquigarrow [v/x]e}$

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

Let's look at the function reduction case:

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

$$\dfrac{\dfrac{\boxed{x : X \vdash e : Y}}{\cdot \vdash \lambda x : X.\, e : X \to Y}\ \to\mathrm{I} \qquad \boxed{\cdot \vdash v : X}}{\cdot \vdash (\lambda x : X.\, e)\, v : Y}\ \to\mathrm{E}$$

- Reducible term = intro <u>immediately</u> followed by an elim
- Evaluation = removal of this detour

All Reductions Remove Detours

$$\frac{}{\mathsf{fst}\,\langle v_1, v_2\rangle \rightsquigarrow v_1} \qquad \frac{}{\mathsf{snd}\,\langle v_1, v_2\rangle \rightsquigarrow v_2}$$

$$\frac{}{\mathsf{case}(\mathsf{L}\,v, \mathsf{L}\,x \to e_1, \mathsf{R}\,y \to e_2) \rightsquigarrow [v/x]e_1}$$

$$\frac{}{\mathsf{case}(\mathsf{R}\,v, \mathsf{L}\,x \to e_1, \mathsf{R}\,y \to e_2) \rightsquigarrow [v/y]e_2}$$

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

Every reduction is of an introduction followed by an eliminator!

Values as Normal Forms

$$\text{Values} \quad v \quad ::= \quad \langle\rangle \;\mid\; \langle v, v' \rangle \;\mid\; \lambda x : A.\, e \;\mid\; \mathsf{L}\, v \;\mid\; \mathsf{R}\, v$$

- 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:

$$\dfrac{\dfrac{}{P, P \vdash P \text{ true}} \qquad \dfrac{}{P, P \vdash P \text{ true}}}{P, P \vdash P \wedge P \text{ true}}$$

has <u>4</u> type-theoretic versions:

$$\dfrac{\vdots}{x : X, y : X \vdash \langle x, x \rangle : X \times X} \qquad \dfrac{\vdots}{x : X, y : X \vdash \langle y, y \rangle : X \times X}$$

$$\dfrac{\vdots}{x : X, y : X \vdash \langle x, y \rangle : X \times X} \qquad \dfrac{\vdots}{x : X, y : X \vdash \langle y, x \rangle : X \times X}$$

Exercises

For the $1, \to$ 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?

Logical Consistency

- 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

$$
\begin{array}{llll}
\text{Types} & X & ::= & 1 \mid X \times Y \mid 0 \mid X + Y \mid X \to Y \\
\text{Values} & v & ::= & \langle\rangle \mid \langle v, v' \rangle \mid \lambda x : A.\, e \mid \mathsf{L}\, v \mid \mathsf{R}\, v
\end{array}
$$

- 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 <u>type safety</u>:
  - Progress: If $\cdot \vdash e : X$ then e is a value or $e \leadsto e'$.
  - Type preservation If $\cdot \vdash e : X$ and $e \leadsto e'$ 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 <u>loop</u> – 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 $e \leadsto^* v$.

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

$$
\begin{array}{lll}
(1) & \dfrac{\overbrace{\Gamma \vdash e : X \to Y}^{(2)} \qquad \overbrace{\Gamma \vdash e' : X}^{(3)}}{\Gamma \vdash e\, e' : Y} & \text{Assumption} \\
(4) & e \leadsto^* v & \text{Induction on (2)} \\
(5) & e' \leadsto^* v' & \text{Induction on (3)} \\
(6) & \cdot \vdash v : X \to Y & \text{Preservation on (2), (4)} \\
(7) & \cdot \vdash v' : X & \text{Preservation on (3), (5)} \\
(8) & \cdot \vdash v \equiv \lambda x : X.\, e'' : X \to Y & \text{Canonical forms on (6)} \\
(9) & x : X \vdash e'' : Y & \text{Subderivation} \\
(10) & \underbrace{\cdot \vdash [v'/x]e'' : Y}_{\text{Can't do induction on this!}} & \text{Substitution}
\end{array}
$$

A Minimal Typed Lambda Calculus

$$
\begin{array}{llcl}
\text{Types} & X & ::= & 1 \mid X \to Y \mid 0 \\
\text{Terms} & e & ::= & x \mid \langle\rangle \mid \lambda x : X.\, e \mid e\, e' \mid \mathrm{abort}\, e \\
\text{Values} & v & ::= & \langle\rangle \mid \lambda x : X.\, e
\end{array}
$$

$$
\frac{X : X \in \Gamma}{\Gamma \vdash x : X}\ \textsc{Hyp} \qquad \frac{}{\Gamma \vdash \langle\rangle : 1}\ 1\mathrm{I}
$$

$$
\frac{\Gamma, X \vdash e : Y}{\Gamma \vdash \lambda x : X.\, e : X \to Y}\ {\to}\mathrm{I} \qquad \frac{\Gamma \vdash e : X \to Y \qquad \Gamma \vdash e' : X}{\Gamma \vdash e\, e' : Y}\ {\to}\mathrm{E}
$$

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

Reductions

$$\frac{e \rightsquigarrow e'}{\mathsf{abort}\, e \rightsquigarrow \mathsf{abort}\, e'}$$

$$\frac{e_1 \rightsquigarrow e_1'}{e_1\, e_2 \rightsquigarrow e_1'\, e_2} \qquad \frac{e_2 \rightsquigarrow e_2'}{v_1\, e_2 \rightsquigarrow v_1\, e_2'}$$

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

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 <u>e halts</u> if and only if there is a v such that $e \leadsto^* v$.

2. Now, we will define a type-indexed family of set of terms:

   - $\mathrm{Halt}_0 = \emptyset$ (i.e, for all e, $e \notin \mathrm{Halt}_0$)
   - $e \in \mathrm{Halt}_1$ holds just when e halts.
   - $e \in \mathrm{Halt}_{X \to Y}$ holds just when
     1. e halts
     2. For all e', if $e' \in \mathrm{Halt}_X$ then $(e\ e') \in \mathrm{Halt}_Y$.

3. Hereditary definition:

   - Halt_1 halts
   - $\mathrm{Halt}_{1 \to 1}$ preserves the property of halting
   - $\mathrm{Halt}_{(1 \to 1) \to (1 \to 1)}$ preserves the property of preserving the property of halting...

Closure Lemma, 1/5

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

Proof: By induction on X:

- Case $X = 1$, $\Rightarrow$:

(1)	$e \leadsto e'$	Assumption
(2)	$e' \in \text{Halt}_1$	Assumption
(3)	$e' \leadsto^* v$	Definition of Halt_1
(4)	$e \leadsto^* v$	Def. of transitive closure, (1) and (3)
(5)	$e \in \text{Halt}_1$	Definition of Halt_1

Closure Lemma, 2/5

- Case $X = 1$, $\Leftarrow$:

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

Closure Lemma, 3/5

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

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

Closure Lemma, 4/5

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

(1)	$e \rightsquigarrow e'$	Assumption
(2)	$e \in \text{Halt}_{Y \to Z}$	Assumption
(3)	$e \rightsquigarrow^* v$	Def. of $\text{Halt}_{Y \to Z}$
(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)	$\quad e\ t \rightsquigarrow e'\ t$	By congruence rule on (1)
(8)	$\quad e\ t \in \text{Halt}_Z$	By (4)
	$\quad e'\ t \in \text{Halt}_Z$	By induction on (6), (7)
(9)	$\forall t \in \text{Halt}_Y, e'\ t \in \text{Halt}_Z$	
(10)	$e' \in \text{Halt}_{Y \to Z}$	Def of $\text{Halt}_{Y \to Z}$ on (5), (8)

Closure Lemma, 5/5

- Case $X = 0$, $\Rightarrow$:

(1)	$e \leadsto e'$	Assumption
(2)	$e' \in \text{Halt}_0$	Assumption
(3)	$e' \in \emptyset$	Definition of Halt_0
(4)	Contradiction!	

- Case $X = 0$, $\Leftarrow$:

(1)	$e \leadsto e'$	Assumption
(2)	$e \in \text{Halt}_0$	Assumption
(3)	$e \in \emptyset$	Definition of Halt_0
(4)	Contradiction!	

Lemma:

If we have that:

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

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

Proof:

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

- Case Hyp:

(1)	$\dfrac{x_j : X_j \in \overrightarrow{x_i : X_i}}{\overrightarrow{x_i : X_i} \vdash x_j : X_j}\ \textsc{Hyp}$	Assumption
(2)	$[\overrightarrow{v_i/x_i}]x_j = v_j$	Def. of substitution
(3)	$v_j \in \mathrm{Halt}_{X_j}$	Assumption
(4)	$[\overrightarrow{v_i/x_i}]x_j \in \mathrm{Halt}_{X_j}$	Equality (2) on (3)

- Case 1I:

(1)	$\dfrac{}{\overrightarrow{x_i : X_i} \vdash \langle\rangle : 1}\ 1\mathrm{I}$	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)	$\langle\rangle \in \mathrm{Halt}_1$	Def. of Halt_1
(5)	$[\overrightarrow{v_i/x_i}]\,\langle\rangle \in \mathrm{Halt}_1$	Equality (2) on (4)

- Case $\to$I:

(1)	$\dfrac{\overrightarrow{x_i : X_i}, y : Y \vdash e : Z}{\overrightarrow{x_i : X_i} \vdash \lambda y : Y.\, e : Y \to Z} \to\mathrm{I}$	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)	$\lambda y : Y.\, [\overrightarrow{v_i/x_i}]e \leadsto^* \lambda y : Y.\, [\overrightarrow{v_i/x_i}]e$	Def of closure

Case $\rightarrow$I:

(5)	Assume $t \in \mathrm{Halt}_Y$:	
(6)	$\quad t \rightsquigarrow^* v_y$	Def of Halt_Y
(7)	$\quad v_y \in \mathrm{Halt}_Y$	Closure on (6)
(8)	$\quad (\lambda y : Y.\, [\overrightarrow{v_i/x_i}]e)\ v_y \rightsquigarrow [\overrightarrow{v_i/x_i}, v_y/y]e$	Rule
(9)	$\quad [\overrightarrow{v_i/x_i}, v_y/y]e \in \mathrm{Halt}_Z$	Induction
(10)	$\quad (\lambda y : Y.\, [\overrightarrow{v_i/x_i}]e)\ t \rightsquigarrow (\lambda y : Y.\, [\overrightarrow{v_i/x_i}]e)\ v_y$	Congruence
(11)	$\quad (\lambda y : Y.\, [\overrightarrow{v_i/x_i}]e)\ t \in \mathrm{Halt}_Z$	Closure
(12)	$\forall t \in \mathrm{Halt}_Y, (\lambda y : Y.\, [\overrightarrow{v_i/x_i}]e)\ t \in \mathrm{Halt}_Z$	

Case $\to$I:

(4)	$\lambda y : Y.\, [\overrightarrow{v_i/x_i}]e \leadsto^* \lambda y : Y.\, [\overrightarrow{v_i/x_i}]e$	Def of closure
(12)	$\forall t \in \mathrm{Halt}_Y, (\lambda y : Y.\, [\overrightarrow{v_i/x_i}]e)\ t \in \mathrm{Halt}_Z$	
(13)	$(\lambda y : Y.\, [\overrightarrow{v_i/x_i}]e) \in \mathrm{Halt}_{Y \to Z}$	Def. of $\mathrm{Halt}_{Y\to Z}$

The Fundamental Lemma, 4/5

- Case $\to$E:

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

- Case 0E:

$$
\begin{array}{lll}
(1) & \dfrac{\overrightarrow{x_i : X_i} \vdash e : 0}{\overrightarrow{x_i : X_i} \vdash \mathrm{abort}\, e : Z}\ 0\mathrm{E} & \text{Assumption} \\[1em]
(2) & \overrightarrow{x_i : X_i} \vdash e : 0 & \text{Subderivation} \\
(3) & [\overrightarrow{v_i / x_i}]e \in \mathrm{Halt}_0 & \text{Induction} \\
(4) & [\overrightarrow{v_i / x_i}]e \in \emptyset & \text{Def of } \mathrm{Halt}_0 \\
(5) & \text{Contradiction!} &
\end{array}
$$

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

Proof:

(1)	$\cdot \vdash e : 0$	Assumption
(2)	$e \in \mathrm{Halt}_0$	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 <u>total</u> 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 <u>all</u> terminating programs!

Exercises

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	$\mathsf{L}\ \langle\rangle$
false	$\mathsf{R}\ \langle\rangle$
if e then e' else e''	$\mathrm{case}(e, \mathsf{L}\ \_ \to e', \mathsf{R}\ \_ \to e'')$

$$\frac{}{\Gamma \vdash \mathrm{true} : \mathrm{bool}} \qquad \frac{}{\Gamma \vdash \mathrm{false} : \mathrm{bool}}$$

$$\frac{\Gamma \vdash e : \mathrm{bool} \qquad \Gamma \vdash e' : X \qquad \Gamma \vdash e'' : X}{\Gamma \vdash \mathrm{if}\ e\ \mathrm{then}\ e'\ \mathrm{else}\ e'' : X}$$

Characters

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

Limitations

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

$$\frac{\Gamma, f : X \to Y, x : X \vdash e : Y}{\Gamma \vdash \mathsf{fun}_{X \to Y}\, f\, x.\, e : X \to Y}\ \textsc{Fix}$$

$$\frac{e' \leadsto e''}{(\mathsf{fun}_{X \to Y}\, f\, x.\, e)\, e' \leadsto (\mathsf{fun}_{X \to Y}\, f\, x.\, e)\, e''}$$

$$\frac{}{(\mathsf{fun}_{X \to Y}\, f\, x.\, e)\, v \leadsto [\mathsf{fun}_{X \to Y}\, f\, x.\, e/f, v/x]e}$$

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

$$
\frac{\dfrac{\dots \qquad \dfrac{\dots \qquad \dfrac{\Delta \vdash \mathit{fact} : \mathtt{int} \to \mathtt{int} \qquad \dfrac{\vdots}{\Delta \vdash n - 1 : \mathtt{int}}}{\Delta \vdash \mathit{fact}(n-1) : \mathtt{int}}}{\Delta \vdash n \times \mathit{fact}(n-1) : \mathtt{int}}}{\overbrace{\Gamma, \mathit{fact} : \mathtt{int} \to \mathtt{int}, n : \mathtt{int}}^{\Delta} \vdash \text{if } n = 0 \text{ then } 1 \text{ else } n \times \mathit{fact}(n-1) : \mathtt{int}}}{\Gamma \vdash \text{fun}_{\mathtt{int} \to \mathtt{int}}\, \mathit{fact}\, n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times \mathit{fact}(n-1) : \mathtt{int} \to \mathtt{int}}
$$

A Bad Use of Recursion

$$\dfrac{\dfrac{}{f : 1 \to 0, x : 1 \vdash f : 1 \to 0} \qquad \dfrac{}{f : 1 \to 0, x : 1 \vdash x : 1}}{\dfrac{f : 1 \to 0, x : 1 \vdash f\,x : 0}{\cdot \vdash \mathrm{fun}_{1\to 0}\, f\,x.\, f\,x : 1 \to 0}}$$

$$\begin{array}{rcl} (\mathrm{fun}_{1\to 0}\, f\,x.\, f\,x)\ \langle\rangle & \leadsto & [\mathrm{fun}_{1\to 0}\, f\,x.\, f\,x/f, \langle\rangle\,/x](f\,x) \\ & \equiv & (\mathrm{fun}_{1\to 0}\, f\,x.\, f\,x)\ \langle\rangle \\ & \leadsto & [\mathrm{fun}_{1\to 0}\, f\,x.\, f\,x/f, \langle\rangle\,/x](f\,x) \\ & \equiv & (\mathrm{fun}_{1\to 0}\, f\,x.\, f\,x)\ \langle\rangle \\ & \ldots & \end{array}$$

Numbers, More Safely

$$\frac{}{\Gamma \vdash \mathsf{z} : \mathbb{N}}\ \mathbb{N}\mathsf{I}_z \qquad \frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \mathsf{s}(e) : \mathbb{N}}\ \mathbb{N}\mathsf{I}_s$$

$$\frac{\Gamma \vdash e_0 : \mathbb{N} \qquad \Gamma \vdash e_1 : X \qquad \Gamma, x : X \vdash e_2 : X}{\Gamma \vdash \mathsf{iter}(e_0, \mathsf{z} \to e_1, \mathsf{s}(x) \to e_2) : X}\ \mathbb{N}\mathsf{E}$$

$$\frac{e_0 \leadsto e_0'}{\mathsf{iter}(e_0, \mathsf{z} \to e_1, \mathsf{s}(x) \to e_2) \leadsto \mathsf{iter}(e_0', \mathsf{z} \to e_1, \mathsf{s}(x) \to e_2)}$$

$$\frac{}{\mathsf{iter}(\mathsf{z}, \mathsf{z} \to e_1, \mathsf{s}(x) \to e_2) \leadsto e_1}$$

$$\frac{}{\mathsf{iter}(\mathsf{s}(v), \mathsf{z} \to e_1, \mathsf{s}(x) \to e_2) \leadsto [\mathsf{iter}(v, \mathsf{z} \to e_1, \mathsf{s}(x) \to e_2)/x]e_2}$$

Expressiveness of Gödel's T

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

$$
\begin{array}{lcl}
n + m & \triangleq & \mathrm{iter}(n, \mathrm{z} \to m, \mathrm{s}(x) \to \mathrm{s}(x)) \\
n \times m & \triangleq & \mathrm{iter}(n, \mathrm{z} \to \mathrm{z}, \mathrm{s}(x) \to m + x) \\
\mathtt{pow}(n, m) & \triangleq & \mathrm{iter}(m, \mathrm{z} \to \mathrm{s}(\mathrm{z}), \mathrm{s}(x) \to n \times x)
\end{array}
$$

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

The Ackermann-Péter Function

$$
\begin{aligned}
A(0, n) &= n + 1 \\
A(m+1, 0) &= A(m, 1) \\
A(m+1, n+1) &= A(m, A(m+1, n))
\end{aligned}
$$

- 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

$$\begin{aligned}
\text{repeat} &\;:\; (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N} \\
\text{repeat} &\;\triangleq\; \lambda f.\, \lambda n.\, \mathsf{iter}(n, \mathsf{z} \to f, \mathsf{s}(x) \to f \circ x)
\end{aligned}$$

$$\begin{aligned}
\text{ack} &\;:\; \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
\text{ack} &\;\triangleq\; \lambda m.\, \lambda n.\, \mathsf{iter}(m, \mathsf{z} \to (\lambda x.\, \mathsf{s}(x)), \mathsf{s}(r) \to \text{repeat}\, r)\, n
\end{aligned}$$

- 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*

Data Structures: Lists

$$\frac{}{\Gamma \vdash [] : \mathsf{list}\, X}\ \textsc{ListNil} \qquad \frac{\Gamma \vdash e : X \qquad \Gamma \vdash e' : \mathsf{list}\, X}{\Gamma \vdash e :: e' : \mathsf{list}\, X}\ \textsc{ListCons}$$

$$\frac{\Gamma \vdash e_0 : \mathsf{list}\, X \qquad \Gamma \vdash e_1 : Z \qquad \Gamma, x : X, r : Z \vdash e_2 : Z}{\Gamma \vdash \mathsf{fold}(e_0, [] \to e_1, x :: r \to e_2) : Z}\ \textsc{ListFold}$$

Data Structures: Lists

$$\frac{e_0 \leadsto e_0'}{e_0 :: e_1 \leadsto e_0' :: e_1} \qquad \frac{e_1 \leadsto e_1'}{v_0 :: e_1 \leadsto v_0 :: e_1'}$$

$$\frac{e_0 \leadsto e_0'}{\mathrm{fold}(e_0, [] \to e_1, x :: r \to e_2) \leadsto \mathrm{fold}(e_0', [] \to e_1, x :: r \to e_2)}$$

$$\frac{}{\mathrm{fold}([], [] \to e_1, x :: r \to e_2) \leadsto e_1}$$

$$\frac{R \triangleq \mathrm{fold}(v', [] \to e_1, x :: r \to e_2)}{\mathrm{fold}(v :: v', [] \to e_1, x :: r \to e_2) \leadsto [v/x, R/r]e_2}$$

Some Functions on Lists

$$
\begin{array}{lcl}
\mathrm{length} & : & \mathrm{list}\,X \to \mathbb{N} \\
\mathrm{length} & \triangleq & \lambda xs.\,\mathrm{fold}(xs, [] \to \mathrm{z}, x :: r \to \mathrm{s}(r)) \\
\\
\mathrm{append} & : & \mathrm{list}\,X \to \mathrm{list}\,X \to \mathrm{list}\,X \\
\mathrm{append} & \triangleq & \lambda x.\,\lambda ys.\,\mathrm{fold}(xs, [] \to ys, x :: r \to x :: r) \\
\\
\mathrm{map} & : & (X \to Y) \to \mathrm{list}\,X \to \mathrm{list}\,Y \\
\mathrm{map} & \triangleq & \lambda f.\,\lambda xs.\,\mathrm{fold}(xs, [] \to [], x :: r \to (f\,x) :: r)
\end{array}
$$

A Logical Perversity

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

A Practical Perversity

$$\begin{array}{lcl} \mathsf{map} & : & (X \to Y) \to \mathsf{list}\, X \to \mathsf{list}\, Y \\ \mathsf{map} & \triangleq & \lambda f.\, \lambda xs.\, \mathsf{fold}(xs, [] \to [], x :: r \to (f\, x) :: r) \end{array}$$

- 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

$$\begin{array}{llll} \text{Types} & A & ::= & \alpha \mid A \to B \mid \forall\alpha.\, A \\ \text{Terms} & e & ::= & x \mid \lambda x : A.\, e \mid e\, e \mid \Lambda\alpha.\, e \mid e\, A \end{array}$$

- We want to support *type polymorphism*
  - append : $\forall\alpha.\, \text{list}\,\alpha \to \text{list}\,\alpha \to \text{list}\,\alpha$
  - map : $\forall\alpha.\forall\beta.\, (\alpha \to \beta) \to \text{list}\,\alpha \to \text{list}\,\beta$
- 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

$$\text{Type Contexts} \quad \Theta \quad ::= \quad \cdot \mid \Theta, \alpha$$

$$\frac{\alpha \in \Theta}{\Theta \vdash \alpha \text{ type}} \qquad \frac{\Theta \vdash A \text{ type} \qquad \Theta \vdash B \text{ type}}{\Theta \vdash A \to B \text{ type}}$$

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

- Judgement $\Theta \vdash A$ 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

$$\text{Term Variable Contexts} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : A$$

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

- Judgement $\Theta \vdash \Gamma$ 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

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

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

$$\frac{\Theta; \Gamma \vdash e : A \to B \qquad \Theta; \Gamma \vdash e' : A}{\Theta; \Gamma \vdash e\, e' : B}$$

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

- Note the presence of substitution in the typing rules!

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 $\Theta, \Theta' \vdash \Gamma$ ctx then $\Theta, \alpha, \Theta' \vdash \Gamma$ ctx
2. (Context Exchange) If $\Theta, \beta, \gamma, \Theta' \vdash \Gamma$ ctx then $\Theta, \gamma, \beta, \Theta' \vdash \Gamma$ ctx
3. (Context Substitution) If $\Theta \vdash A$ type and $\Theta, \alpha \vdash \Gamma$ type then $\Theta \vdash [A/\alpha]\Gamma$ type

- This just lifts the type-level structural properties to contexts

Regularity of Typing

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 $\Theta, \Theta' \vdash \Gamma$ ctx and $\Theta, \Theta'; \Gamma \vdash e : A$ then $\Theta, \alpha, \Theta'; \Gamma \vdash e : A$.
- (Type Exchange of Terms) If $\Theta, \alpha, \beta, \Theta' \vdash \Gamma$ ctx and $\Theta, \alpha, \beta, \Theta'; \Gamma \vdash e : A$ then $\Theta, \beta, \alpha, \Theta'; \Gamma \vdash e : A$.
- (Type Substitution of Terms) If $\Theta, \alpha \vdash \Gamma$ ctx and $\Theta \vdash A$ 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 $\Theta \vdash \Gamma, \Gamma'$ ctx and $\Theta \vdash B$ type and $\Theta; \Gamma, \Gamma' \vdash e : A$ then $\Theta; \Gamma, y : B, \Gamma' \vdash e : A$
- (Exchange of Terms) If $\Theta \vdash \Gamma, y : B, z : C, \Gamma'$ 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 $\Theta \vdash \Gamma, x : A$ 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

Conclusion

- 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	$::=$	$\alpha \mid A \to B \mid \forall\alpha.\, A$
Terms	e	$::=$	$x \mid \lambda x : A.\, e \mid e\, e \mid \Lambda\alpha.\, e \mid e\, A$
Type Contexts	Θ	$::=$	$\cdot \mid \Theta, \alpha$
Term Contexts	Γ	$::=$	$\cdot \mid \Gamma, x : A$

Judgement	Notation
Well-formedness of types	$\Theta \vdash A$ type
Well-formedness of term contexts	$\Theta \vdash \Gamma$ ctx
Term typing	$\Theta \vdash \Gamma : eA$

Well-formedness of Types

$$\frac{\alpha \in \Theta}{\Theta \vdash \alpha \text{ type}} \qquad \frac{\Theta \vdash A \text{ type} \qquad \Theta \vdash B \text{ type}}{\Theta \vdash A \to B \text{ type}}$$

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

- Judgement $\Theta \vdash A$ 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

$$\text{Term Variable Contexts} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : A$$

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

- 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

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

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

$$\frac{\Theta; \Gamma \vdash e : A \to B \qquad \Theta; \Gamma \vdash e' : A}{\Theta; \Gamma \vdash e\, e' : B}$$

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

- Note the presence of substitution in the typing rules!

Operational Semantics

$$\text{Values} \quad v \quad ::= \quad \lambda x : A.\, e \;\mid\; \Lambda\alpha.\, e$$

$$\frac{e_0 \rightsquigarrow e_0'}{e_0\; e_1 \rightsquigarrow e_0'\; e_1}\ \textsc{CongFun} \qquad \frac{e_1 \rightsquigarrow e_1'}{v_0\; e_1 \rightsquigarrow v_0\; e_1'}\ \textsc{CongFunArg}$$

$$\frac{}{(\lambda x : A.\, e)\, v \rightsquigarrow [v/x]e}\ \textsc{FunEval}$$

$$\frac{e \rightsquigarrow e'}{e\, A \rightsquigarrow e'\, A}\ \textsc{CongForall} \qquad \frac{}{(\Lambda\alpha.\, e)\, A \rightsquigarrow [A/\alpha]e}\ \textsc{ForallEval}$$

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 $\Theta, \Theta' \vdash \Gamma$ ctx then $\Theta, \alpha, \Theta' \vdash \Gamma$ ctx
2. (Context Exchange) If $\Theta, \beta, \gamma, \Theta' \vdash \Gamma$ ctx then $\Theta, \gamma, \beta, \Theta' \vdash \Gamma$ ctx
3. (Context Substitution) If $\Theta \vdash A$ type and $\Theta, \alpha \vdash \Gamma$ type then $\Theta \vdash [A/\alpha]\Gamma$ type

- This just lifts the type-level structural properties to contexts
- Proof via induction on derivations of $\Theta \vdash \Gamma$ ctx

Regularity of Typing

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

Summary

- 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

Type Safety

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:

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

(1)	$\cdot;\cdot \vdash e\,A : [A/\alpha]B$	Assumption
(4)	$e \leadsto e'$ or e is a value	Induction on (2)
	Case on (4)	
(5)	Case $e \leadsto e'$:	
(6)	$e\,A \leadsto e'\,A$	by CongForall on (5)
(7)	Case e is a value:	
(8)	$e = \Lambda\alpha.\, e'$	By canonical forms on (2)
(9)	$(\Lambda\alpha.\, e')\,A \leadsto [A/\alpha]e$	By ForallEval

Preservation: Big Lambdas

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

(1) $$\dfrac{}{(\Lambda\alpha.\, e)\, A \rightsquigarrow [A/\alpha]e}\ \textsc{ForallEval}$$ Assumption

(2) $$\dfrac{\dfrac{\overbrace{\alpha; \cdot \vdash e : B}^{(3)}}{\cdot; \cdot \vdash \Lambda\alpha.\, e : \forall\alpha.\, B} \qquad \overbrace{\cdot \vdash A \text{ type}}^{(4)}}{\cdot; \cdot \vdash (\Lambda\alpha.\, e)\, A : [A/\alpha]B}$$ Assumption

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

Church Encodings: Representing Data with Functions

- System has the types $\forall\alpha. A$ and $A \to B$
- 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

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \qquad \frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

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

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

Type		Encoding
bool	$\triangleq$	$\forall\alpha.\, \alpha \to \alpha \to \alpha$
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

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

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

Church Encodings: Pairs

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

Evaluating Church Pairs

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

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

Church Encodings: Sums

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

Evaluating Church Sums

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

Church Encodings: Natural Numbers

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

Evaluating Church Naturals

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

Evaluating Church Naturals

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

Church Encodings: Lists

Type	Encoding
$\mathrm{list}\,X$	$\forall\alpha.\,\alpha \to (X \to \alpha \to \alpha) \to \alpha$
$[]$	$\Lambda\alpha.\,\lambda n : \alpha.\,\lambda c : X \to \alpha \to \alpha.\,n$
$e :: e'$	$\Lambda\alpha.\,\lambda n : \alpha.\,\lambda c : X \to \alpha \to \alpha.\,c\;e\;(e'\;\alpha\;n\;c)$

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

Conclusions

- 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

Exercises

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

$$
\begin{array}{llll}
\text{Types} & A & ::= & \ldots \mid \exists\alpha.\, A \\
\text{Terms} & e & ::= & \ldots \mid \mathsf{pack}_{\alpha.B}(A, e) \mid \mathsf{let}\ \mathsf{pack}(\alpha, x) = e\ \mathsf{in}\ e' \\
\text{Values} & v & ::= & \mathsf{pack}_{\alpha.B}(A, v)
\end{array}
$$

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

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

Operational Semantics for Abstract Types

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

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

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

Data Abstraction in System F

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

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

- 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

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

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

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

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

Reduction of the Encoding

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

System F, The Girard-Reynolds Polymorphic Lambda Calculus

$$
\begin{array}{llll}
\text{Types} & A & ::= & \alpha \mid A \to B \mid \forall\alpha.\, A \\
\text{Terms} & e & ::= & x \mid \lambda x : A.\, e \mid e\, e \mid \Lambda\alpha.\, e \mid e\, A \\
\text{Values} & v & ::= & \lambda x : A.\, e \mid \Lambda\alpha.\, e
\end{array}
$$

$$
\frac{e_0 \leadsto e_0'}{e_0\, e_1 \leadsto e_0'\, e_1}\ \textsc{CongFun}
\qquad
\frac{e_1 \leadsto e_1'}{v_0\, e_1 \leadsto v_0\, e_1'}\ \textsc{CongFunArg}
$$

$$
\frac{}{(\lambda x : A.\, e)\, v \leadsto [v/x]e}\ \textsc{FunEval}
$$

$$
\frac{e \leadsto e'}{e\, A \leadsto e'\, A}\ \textsc{CongForall}
\qquad
\frac{}{(\Lambda\alpha.\, e)\, A \leadsto [A/\alpha]e}\ \textsc{ForallEval}
$$

Summary

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

Termination for System F

- 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. $e \leadsto^* v$ for some v).
- (Closure) If $e \leadsto e'$, 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

$$\frac{\alpha \in \Theta}{\Theta \vdash \alpha \text{ type}} \qquad \frac{\Theta \vdash A \text{ type} \qquad \Theta \vdash B \text{ type}}{\Theta \vdash A \to B \text{ type}} \qquad \frac{\Theta, \alpha \vdash A \text{ type}}{\Theta \vdash \forall \alpha.\, A \text{ type}}$$

- We can interpret *type well-formedness derivations*
- Given a type variable context Θ, we define will define a variable interpretation θ as a map from $\mathrm{dom}(\Theta)$ 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

$$[\![-]\!] \in \text{WellFormedType} \rightarrow \text{VarInterpretation} \rightarrow \text{SemanticType}$$

$$\begin{array}{lcl}
[\![\Theta \vdash \alpha \text{ type}]\!]\ \theta & = & \theta(\alpha) \\
[\![\Theta \vdash A \rightarrow B \text{ type}]\!]\ \theta & = & \left\{ e \;\middle|\; \begin{array}{l} e \text{ halts} \wedge \\ \forall e' \in [\![\Theta \vdash A \text{ type}]\!]\ \theta. \\ \qquad (e\, e') \in [\![\Theta \vdash B \text{ type}]\!]\ \theta \end{array} \right\} \\
[\![\Theta \vdash \forall \alpha.\, B \text{ type}]\!]\ \theta & = & \left\{ e \;\middle|\; \begin{array}{l} e \text{ halts} \wedge \\ \forall A \in \text{type}, X \in \mathrm{SemType}. \\ \qquad (e\, A) \in [\![\Theta, \alpha \vdash B \text{ type}]\!]\ (\theta, X/\alpha) \end{array} \right\}
\end{array}$$

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 $[\![\Theta \vdash A \text{ type}]\!]\ \theta$ is a semantic type.
- **Exchange:** $[\![\Theta, \alpha, \beta, \Theta' \vdash A \text{ type}]\!] = [\![\Theta, \beta, \alpha, \Theta' \vdash A \text{ type}]\!]$
- **Weakening:** If $\Theta \vdash A$ type, then $[\![\Theta, \alpha \vdash A \text{ type}]\!]\ (\theta, X/\alpha) = [\![\Theta \vdash A \text{ type}]\!]\ \theta$.
- **Substitution:** If $\Theta \vdash A$ type and $\Theta, \alpha \vdash B$ type then $[\![\Theta \vdash [A/\alpha]B \text{ type}]\!]\ \theta = [\![\Theta, \alpha \vdash B \text{ type}]\!]\ (\theta, [\![\Theta \vdash A \text{ type}]\!]\ \theta)$

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 $[\![\Theta \vdash \forall \alpha.\, A \text{ type}]\!]\ \theta$ is a type.

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

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

Substitution: (one half of) the $\forall$ case

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

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

The Fundamental Lemma

If we have that

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

Then it follows that:

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

Questions

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:

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

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:

$$\begin{array}{lcl} \mathit{map} & : & \forall\alpha.\forall\beta.\,(\alpha \to \beta) \to \mathsf{list}\,\alpha \to \mathsf{list}\,\beta \\ \mathit{isEven} & : & \mathbb{N} \to \mathsf{bool} \end{array}$$

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

$$\mathit{map}\,\mathbb{N}\,\mathsf{bool}\,\mathit{isEven} \quad : \quad \mathsf{list}\,\mathbb{N} \to \mathsf{list}\,\mathsf{bool}$$

If you have a list of lists of natural numbers:

$$\begin{array}{l} \mathit{map}\,(\mathsf{list}\,\mathbb{N})\,(\mathsf{list}\,\mathsf{bool})\,(\mathit{map}\,\mathbb{N}\,\mathsf{bool}\,\mathit{isEven}) \\ \quad : \mathsf{list}\,(\mathsf{list}\,\mathbb{N}) \to \mathsf{list}\,(\mathsf{list}\,\mathsf{bool}) \end{array}$$

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 $map\,\mathbb{N}\,\mathrm{bool}\,isEven$
- Constraint propagation via the *unification algorithm* figures out what the applications should have been

Example:

$map \quad isEven$	Term that needs type inference
$map\ ?a\ ?b\ isEven$	Introduce placeholders $?a$ and $?b$
$map\ ?a\ ?b$	$: (?a \to ?b) \to \mathrm{list}\,?a \to \mathrm{list}\,?b$
$isEven : \mathbb{N} \to \mathrm{bool}$	So $?a \to ?b$ must equal $\mathbb{N} \to \mathrm{bool}$
$?a = \mathbb{N}, ?b = \mathrm{bool}$	Only choice that makes $?a \to ?b = \mathbb{N} \to \mathrm{bool}$

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

Effects

- 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

State

```
# 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

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

Operational Semantics

$$\frac{\langle \sigma; e_0 \rangle \rightsquigarrow \langle \sigma'; e_0' \rangle}{\langle \sigma; e_0\, e_1 \rangle \rightsquigarrow \langle \sigma'; e_0'\, e_1 \rangle} \qquad \frac{\langle \sigma; e_1 \rangle \rightsquigarrow \langle \sigma'; e_1' \rangle}{\langle \sigma; v_0\, e_1 \rangle \rightsquigarrow \langle \sigma'; v_0\, e_1' \rangle}$$

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

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

Operational Semantics

$$\frac{\langle \sigma; e \rangle \leadsto \langle \sigma'; e' \rangle}{\langle \sigma; \mathsf{new}\, e \rangle \leadsto \langle \sigma'; \mathsf{new}\, e' \rangle} \qquad \frac{l \notin \mathrm{dom}(\sigma)}{\langle \sigma; \mathsf{new}\, v \rangle \leadsto \langle (\sigma, l : v); l \rangle}$$

$$\frac{\langle \sigma; e \rangle \leadsto \langle \sigma'; e' \rangle}{\langle \sigma; !e \rangle \leadsto \langle \sigma'; !e' \rangle} \qquad \frac{l : v \in \sigma}{\langle \sigma; !l \rangle \leadsto \langle \sigma; v \rangle}$$

$$\frac{\langle \sigma; e_0 \rangle \leadsto \langle \sigma'; e_0' \rangle}{\langle \sigma; e_0 := e_1 \rangle \leadsto \langle \sigma'; e_0' := e_1 \rangle} \qquad \frac{\langle \sigma; e_1 \rangle \leadsto \langle \sigma'; e_1' \rangle}{\langle \sigma; v_0 := e_1 \rangle \leadsto \langle \sigma'; v_0 := e_1' \rangle}$$

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

Typing for Terms

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

$$\frac{x : X \in \Gamma}{\Sigma; \Gamma \vdash x : X}\ \textsc{Hyp} \qquad \frac{}{\Sigma; \Gamma \vdash \langle\rangle : 1}\ 1\mathrm{I} \qquad \frac{}{\Sigma; \Gamma \vdash n : \mathbb{N}}\ \mathbb{N}\mathrm{I}$$

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

$$\frac{\Sigma; \Gamma \vdash e : X \to Y \qquad \Sigma; \Gamma \vdash e' : X}{\Sigma; \Gamma \vdash e\, e' : Y}\ {\to}\mathrm{E}$$

- Similar to STLC rules + thread Σ through all judgements

Typing for Imperative Terms

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

$$\frac{\Sigma; \Gamma \vdash e : X}{\Sigma; \Gamma \vdash \mathsf{new}\, e : \mathsf{ref}\, X}\ \textsc{RefI} \qquad \frac{\Sigma; \Gamma \vdash e : \mathsf{ref}\, X}{\Sigma; \Gamma \vdash !e : X}\ \textsc{RefGet}$$

$$\frac{\Sigma; \Gamma \vdash e : \mathsf{ref}\, X \qquad \Sigma; \Gamma \vdash e' : X}{\Sigma; \Gamma \vdash e := e' : 1}\ \textsc{RefSet}$$

$$\frac{l : X \in \Sigma}{\Sigma; \Gamma \vdash l : \mathsf{ref}\, X}\ \textsc{RefBar}$$

- 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 $e \leadsto e'$
- New operational semantics $\langle \sigma; e \rangle \leadsto \langle \sigma'; e' \rangle$ mentions stores, too.
- To prove type safety, we will need a notion of store typing

Store and Configuration Typing

$$\boxed{\Sigma \vdash \sigma' : \Sigma'} \qquad \boxed{\langle \sigma; e \rangle : \langle \Sigma; X \rangle}$$

$$\frac{}{\Sigma \vdash \cdot : \cdot}\ \textsc{StoreNil} \qquad \frac{\Sigma \vdash \sigma' : \Sigma' \qquad \Sigma; \cdot \vdash v : X}{\Sigma \vdash (\sigma', l : v) : (\Sigma', l : X)}\ \textsc{StoreCons}$$

$$\frac{\Sigma \vdash \sigma : \Sigma \qquad \Sigma; \cdot \vdash e : X}{\langle \sigma; e \rangle : \langle \Sigma; X \rangle}\ \textsc{ConfigOK}$$

- 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

$$\begin{array}{ll} 1. & \langle \cdot; \mathsf{new}\,\langle\rangle\rangle : \langle \cdot; \mathsf{ref}\,1\rangle \\ 2. & \langle \cdot; \mathsf{new}\,\langle\rangle\rangle \rightsquigarrow \langle (l : \langle\rangle); l\rangle \quad \text{for some } l \end{array}$$

However, it is not the case that

$$\langle l : \langle\rangle; l\rangle : \langle \cdot; \mathsf{ref}\,1\rangle$$

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 $\Sigma; \Gamma \vdash [e/x]e' : Z$.

Type Safety, Repaired

Theorem (Progress):

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

Theorem (Preservation):

If $\langle \sigma; e \rangle : \langle \Sigma; X \rangle$ and $\langle \sigma; e \rangle \leadsto \langle \sigma'; e' \rangle$ 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 $\langle \sigma; e \rangle \leadsto \langle \sigma'; e' \rangle$

A Curious Higher-order Function

- Suppose we have an unknown function in the STLC:

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

- 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 gets $\langle\rangle$ out of it. So the argument can never influence the value of type $\mathbb{N}$ that f produces.

The Power of the State

$$\begin{array}{lcl} count & : & ((1 \to 1) \to 1) \to \mathbb{N} \\ count\ \ f & = & \mathrm{let}\ r : \mathrm{ref}\,\mathbb{N} = \mathrm{new}\,0\ \mathrm{in} \\ & & \mathrm{let}\ inc : 1 \to 1 = \lambda z : 1.\, r := !r + 1\ \mathrm{in} \\ & & f(inc) \end{array}$$

- 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

```
let knot : ((int -> int) -> int -> int) -> int -> int =
  fun f ->
    let r       = ref (fun n -> 0) in
    let recur   = fun n -> !r n in
    let ()      = r := fun n -> f recur n in
    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!

Another False Theorem

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
- So every type is inhabited, and consistency fails

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

```ocaml
let knot : ((int -> int) -> int -> int) -> int -> int =
  fun f ->
    let r       = ref (fun n -> 0) in
    let recur   = fun n -> !r n in
    let ()      = r := fun n -> f recur n in
    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!

Another False Theorem

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
- So every type is inhabited, and consistency fails

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 consistency
4. Problem: unforseen 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

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

Typing for Pure Terms

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

$$\frac{x : X \in \Gamma}{\Sigma; \Gamma \vdash x : X}\ \textsc{Hyp} \qquad \frac{}{\Sigma; \Gamma \vdash \langle\rangle : 1}\ 1\mathrm{I} \qquad \frac{}{\Sigma; \Gamma \vdash n : \mathbb{N}}\ \mathbb{N}\mathrm{I}$$

$$\frac{\Sigma; \Gamma, x : X \vdash e : Y}{\Sigma; \Gamma \vdash \lambda x : X.\, e : X \to Y}\ {\to}\mathrm{I} \qquad \frac{\Sigma; \Gamma \vdash e : X \to Y \qquad \Sigma; \Gamma \vdash e' : X}{\Sigma; \Gamma \vdash e\, e' : Y}\ {\to}\mathrm{E}$$

$$\frac{l : X \in \Sigma}{\Sigma; \Gamma \vdash l : \mathrm{ref}\, X}\ \textsc{RefBar} \qquad \frac{\color{orange}{\Sigma; \Gamma \vdash t \div X}}{\Sigma; \Gamma \vdash \{t\} : \mathrm{T}\, X}\ \mathrm{TI}$$

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

Typing for Effectful Terms

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

$$\frac{\Sigma;\Gamma \vdash e : X}{\Sigma;\Gamma \vdash \mathsf{new}\, e \div \mathsf{ref}\, X}\ \textsc{RefI} \qquad \frac{\Sigma;\Gamma \vdash e : \mathsf{ref}\, X}{\Sigma;\Gamma \vdash !e \div X}\ \textsc{RefGet}$$

$$\frac{\Sigma;\Gamma \vdash e : \mathsf{ref}\, X \qquad \Sigma;\Gamma \vdash e' : X}{\Sigma;\Gamma \vdash e := e' \div 1}\ \textsc{RefSet}$$

$$\frac{\Sigma;\Gamma \vdash e : X}{\Sigma;\Gamma \vdash \mathsf{return}\, e \div X}\ \textsc{tRet} \qquad \frac{\Sigma;\Gamma \vdash e : \mathsf{T}X \qquad \Sigma;\Gamma, x : X \vdash t \div Z}{\Sigma;\Gamma \vdash \mathsf{let}\, x = e;\ t \div Z}\ \textsc{tLet}$$

- We now mark potentially effectful terms in the judgement
- Note that $\mathsf{return}\, e$ isn't effectful – conservative approximation!

A Two-Level Operational Semantics: Pure Part

$$\frac{e_0 \rightsquigarrow e_0'}{e_0\, e_1 \rightsquigarrow e_0'\, e_1} \qquad \frac{e_1 \rightsquigarrow e_1'}{v_0\, e_1 \rightsquigarrow v_0\, e_1'} \qquad \frac{}{(\lambda x : X.\, e)\, v \rightsquigarrow [v/x]e}$$

- 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

$$\frac{e \rightsquigarrow e'}{\langle \sigma; \mathsf{new}\, e \rangle \rightsquigarrow \langle \sigma; \mathsf{new}\, e' \rangle} \qquad \frac{l \notin \mathrm{dom}(\sigma)}{\langle \sigma; \mathsf{new}\, v \rangle \rightsquigarrow \langle (\sigma, l : v); \mathsf{return}\, l \rangle}$$

$$\frac{e \rightsquigarrow e'}{\langle \sigma; !e \rangle \rightsquigarrow \langle \sigma; !e' \rangle} \qquad \frac{l : v \in \sigma}{\langle \sigma; !l \rangle \rightsquigarrow \langle \sigma; \mathsf{return}\, v \rangle}$$

$$\frac{e_0 \rightsquigarrow e_0'}{\langle \sigma; e_0 := e_1 \rangle \rightsquigarrow \langle \sigma; e_0' := e_1 \rangle} \qquad \frac{e_1 \rightsquigarrow e_1'}{\langle \sigma; v_0 := e_1 \rangle \rightsquigarrow \langle \sigma; v_0 := e_1' \rangle}$$

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

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

$$\frac{e \rightsquigarrow e'}{\langle \sigma; \mathrm{return}\, e \rangle \rightsquigarrow \left\langle \sigma; \mathrm{return}\, e' \right\rangle}$$

$$\frac{e \rightsquigarrow e'}{\langle \sigma; \mathrm{let}\, x = e;\ t \rangle \rightsquigarrow \left\langle \sigma; \mathrm{let}\, x = e';\ t \right\rangle}$$

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

$$\frac{\langle \sigma; t_0 \rangle \rightsquigarrow \left\langle \sigma'; t_0' \right\rangle}{\langle \sigma; \mathrm{let}\, x = \{t_0\};\ t_1 \rangle \rightsquigarrow \left\langle \sigma'; \mathrm{let}\, x = \left\{t_0'\right\};\ t_1 \right\rangle}$$

Store and Configuration Typing

$$\boxed{\Sigma \vdash \sigma' : \Sigma'} \qquad \boxed{\langle \sigma; e \rangle : \langle \Sigma; X \rangle}$$

$$\frac{}{\Sigma \vdash \cdot : \cdot}\ \textsc{StoreNil} \qquad \frac{\Sigma \vdash \sigma' : \Sigma' \qquad \Sigma; \cdot \vdash v : X}{\Sigma \vdash (\sigma', l : v) : (\Sigma', l : X)}\ \textsc{StoreCons}$$

$$\frac{\Sigma \vdash \sigma : \Sigma \qquad \Sigma; \cdot \vdash t \div X}{\langle \sigma; t \rangle : \langle \Sigma; X \rangle}\ \textsc{ConfigOK}$$

- 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

Substitution and Structural Properties, 1/2

- **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 $\Sigma; \Gamma \vdash [e/x]e' : Z$.

Substitution and Structural Properties, 2/2

- **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 $\Sigma; \Gamma \vdash [e/x]t \div Z$.

Proof Order

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 $\Longrightarrow$ 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 (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

$$
\begin{array}{llll}
\text{Types} & X & ::= & 1 \mid \mathbb{N} \mid X \to Y \mid \mathsf{T}_{\mathsf{IO}}\, X \\
\text{Pure Terms} & e & ::= & \langle\rangle \mid n \mid \lambda x : X.\, e \mid e\, e' \mid \{t\} \\
\text{Impure Terms} & t & ::= & \mathsf{print}\, e \mid \mathsf{let}\; x = e;\; t \mid \mathsf{return}\, e \\
\\
\text{Values} & v & ::= & \langle\rangle \mid n \mid \lambda x : X.\, e \mid \{t\} \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : X
\end{array}
$$

Monads for I/O: Typing Pure Terms

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

$$\frac{x : X \in \Gamma}{\Gamma \vdash x : X}\ \textsc{Hyp} \qquad \frac{}{\Gamma \vdash \langle\rangle : 1}\ 1\mathrm{I} \qquad \frac{}{\Gamma \vdash n : \mathbb{N}}\ \mathbb{N}\mathrm{I}$$

$$\frac{\Gamma, x : X \vdash e : Y}{\Gamma \vdash \lambda x : X.\, e : X \to Y}\ {\to}\mathrm{I} \qquad \frac{\Gamma \vdash e : X \to Y \qquad \Gamma \vdash e' : X}{\Gamma \vdash e\, e' : Y}\ {\to}\mathrm{E}$$

$$\frac{\Gamma \vdash t \div X}{\Gamma \vdash \{t\} : \mathrm{T}X}\ \mathrm{TI}$$

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

Typing for Effectful Terms

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

$$\frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \mathsf{print}\, e \div 1}\ \textsc{TPrint}$$

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash \mathsf{return}\, e \div X}\ \textsc{TRet} \qquad \frac{\Gamma \vdash e : \mathsf{T}X \qquad \Gamma, x : X \vdash t \div Z}{\Gamma \vdash \mathsf{let}\, x = e;\ t \div Z}\ \textsc{TLet}$$

- TRet and TLet are identical rules
- Difference is in the operations – print e vs get/set/new

Operational Semantics for I/O: Pure Part

$$\frac{e_0 \leadsto e_0'}{e_0\, e_1 \leadsto e_0'\, e_1} \qquad \frac{e_1 \leadsto e_1'}{v_0\, e_1 \leadsto v_0\, e_1'} \qquad \frac{}{(\lambda x : X.\, e)\, v \leadsto [v/x]e}$$

- *Identical* to the pure rules for state!

Operational Semantics for I/O: Impure Part

$$\frac{e \leadsto e'}{\langle \omega; \mathrm{print}\, e \rangle \leadsto \langle \omega; \mathrm{print}\, e' \rangle} \qquad \frac{}{\langle \omega; \mathrm{print}\, n \rangle \leadsto \langle (n :: \omega); \mathrm{return}\, \langle\rangle \rangle}$$

$$\frac{e \leadsto e'}{\langle \omega; \mathrm{return}\, e \rangle \leadsto \langle \omega; \mathrm{return}\, e' \rangle} \qquad \frac{e \leadsto e'}{\langle \omega; \mathrm{let}\, x = e;\ t \rangle \leadsto \langle \omega; \mathrm{let}\, x = e';\ t \rangle}$$

$$\frac{}{\langle \omega; \mathrm{let}\, x = \{\mathrm{return}\, v\};\ t_1 \rangle \leadsto \langle \omega; [v/x]t_1 \rangle}$$

$$\frac{\langle \omega; t_0 \rangle \leadsto \langle \omega'; t_0' \rangle}{\langle \omega; \mathrm{let}\, x = \{t_0\};\ t_1 \rangle \leadsto \langle \omega'; \mathrm{let}\, x = \{t_0'\};\ t_1 \rangle}$$

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

Limitations of Monadic Style: Encapsulating Effects

```
let fact : int -> int = fun n ->
  let r = ref 1 in
  let rec loop n =
    match n with
    | 0 ->  !r
    | n -> let () = r := !r * n in
           loop (n-1)
  in
  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

```
let rec find' : ('a -> bool) -> 'a list -> 'a =
  fun p ys ->
    match ys with
    | [] -> raise Not_found
    | y :: ys -> if p y then y else find' p ys

let find : ('a -> bool) -> 'a list -> 'a option =
  fun p xs ->
    try Some (find' p xs)
    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:

```
p1 : (int -> ans) state
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

- 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`

Questions

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

$$\frac{P \in \Psi}{\Psi \vdash P \text{ true}}\ \textsc{Hyp} \qquad \frac{}{\Psi \vdash \top \text{ true}}\ \top\mathrm{I}$$

$$\frac{\Psi \vdash P \text{ true} \qquad \Psi \vdash Q \text{ true}}{\Psi \vdash P \wedge Q \text{ true}}\ \wedge\mathrm{I} \qquad \frac{\Psi \vdash P_1 \wedge P_2 \text{ true}}{\Psi \vdash P_i \text{ true}}\ \wedge\mathrm{E}_i$$

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

Disjunction and Falsehood

$$\frac{\Psi \vdash P \text{ true}}{\Psi \vdash P \vee Q \text{ true}} \vee\mathrm{I}_1 \qquad \frac{\Psi \vdash Q \text{ true}}{\Psi \vdash P \vee Q \text{ true}} \vee\mathrm{I}_2$$

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

(no intro for $\perp$)

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

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: $\neg P = P \to \bot$
  - "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

$$
\begin{array}{llcl}
\text{Propositions} & A & ::= & \top \mid A \wedge B \mid \bot \mid A \vee B \mid \neg A \\
\text{True contexts} & \Gamma & ::= & \cdot \mid \Gamma, A \\
\text{False contexts} & \Delta & ::= & \cdot \mid \Delta, A
\end{array}
$$

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	$\Gamma; \Delta \vdash$ contr	Γ and Δ contradict one another

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

Proofs

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

(No rule for $\bot$)

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

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

$$\frac{\Gamma; \Delta \vdash A \;\; \text{true}}{\Gamma; \Delta \vdash A \vee B \;\; \text{true}}\ \vee\mathrm{P}_1 \qquad \frac{\Gamma; \Delta \vdash B \;\; \text{true}}{\Gamma; \Delta \vdash A \vee B \;\; \text{true}}\ \vee\mathrm{P}_2$$

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

Refutations

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

(No rule for $\top$)

$$\frac{}{\Gamma; \Delta \vdash \bot \text{ false}}\ \bot R$$

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

$$\frac{\Gamma; \Delta \vdash A \text{ false}}{\Gamma; \Delta \vdash A \wedge B \text{ false}}\ \wedge R_1 \qquad \frac{\Gamma; \Delta \vdash B \text{ false}}{\Gamma; \Delta \vdash A \wedge B \text{ false}}\ \wedge R_2$$

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

75% of the Way to Classical Logic

Connective	To Prove	To Refute
$\top$	Do nothing	Impossible!
$A \wedge B$	Prove A *and* prove B	Refute A *or* refute B
$\bot$	Impossible!	Do nothing
$A \vee B$	Prove A *or* prove B	Refute A *and* refute B
$\neg A$	Refute A	Prove A

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

$$\dfrac{\dfrac{\dfrac{}{A; \cdot \vdash A \;\; \text{true}}\;\textsc{Hyp}}{A; \cdot \vdash \neg A \;\text{false}}\;\neg\text{R}}{A; \cdot \vdash \neg\neg A \;\; \text{true}}\;\neg\text{P}$$

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: $\lambda x : A \times B.\, \mathsf{fst}\, x$
- 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

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

Contradictions

$$\frac{\Gamma; \Delta \vdash A \ \text{ true} \qquad \Gamma; \Delta \vdash A \text{ false}}{\Gamma; \Delta \vdash \text{contr}} \ \textsc{Contr}$$

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

Double Negation Elimination

$$
\dfrac{
  \dfrac{}{\neg\neg A; A \vdash \neg\neg A \;\; \mathsf{true}}
  \qquad
  \dfrac{\dfrac{\dfrac{}{\neg\neg A; A \vdash A \;\mathsf{false}}}{\neg\neg A; A \vdash \neg A \;\; \mathsf{true}}}{\neg\neg A; A \vdash \neg\neg A \;\mathsf{false}}
}{
  \dfrac{\neg\neg A; A \vdash \mathsf{contr}}{\neg\neg A; \cdot \vdash A \;\; \mathsf{true}}
}
$$

Projections: $A \wedge B$ entails A

$$
\dfrac{\dfrac{}{A \wedge B; A \vdash A \wedge B \;\; \mathsf{true}} \qquad \dfrac{\dfrac{}{A \wedge B; A \vdash A \; \mathsf{false}}}{A \wedge B; A \vdash A \wedge B \; \mathsf{false}}}{\dfrac{A \wedge B; A \vdash \mathsf{contr}}{A \wedge B; \cdot \vdash A \;\; \mathsf{true}}}
$$

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

$$
\dfrac{\dfrac{}{A; A \vee B \vdash A \vee B \text{ false}} \qquad \dfrac{\dfrac{}{A; A \vee B \vdash A \text{ true}}}{A; A \vee B \vdash A \vee B \text{ true}}}{\dfrac{A; A \vee B \vdash \text{contr}}{\cdot; A \vee B \vdash A \text{ false}}}
$$

The Excluded Middle

$$
\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\cdot; A \vee \neg A \vdash A \ \mathsf{false}}}{\cdot; A \vee \neg A \vdash \neg A \ \mathsf{true}}}{\cdot; A \vee \neg A \vdash A \vee \neg A \ \mathsf{true}} \qquad \dfrac{}{\cdot; A \vee \neg A \vdash A \vee \neg A \ \mathsf{false}}}{\dfrac{\cdot; A \vee \neg A \vdash \mathsf{contr}}{\cdot; \cdot \vdash A \vee \neg A \ \mathsf{true}}}
$$

Proof (and Refutation) Terms

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

Expressions — Proof Terms

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

(No rule for $\bot$)

$$\frac{}{\Gamma; \Delta \vdash \langle\rangle : \top \text{ true}}\ \top\mathrm{P}$$

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

$$\frac{\Gamma; \Delta \vdash e : A \text{ true}}{\Gamma; \Delta \vdash \mathrm{L}\, e : A \vee B \text{ true}}\ \vee\mathrm{P}_1 \qquad \frac{\Gamma; \Delta \vdash e : B \text{ true}}{\Gamma; \Delta \vdash \mathrm{R}\, e : A \vee B \text{ true}}\ \vee\mathrm{P}_2$$

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

Continuations — Refutation Terms

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

(No rule for $\top$)

$$\frac{}{\Gamma; \Delta \vdash [] : \bot \text{ false}} \bot\text{R}$$

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

$$\frac{\Gamma; \Delta \vdash k : A \text{ false}}{\Gamma; \Delta \vdash \mathsf{fst}\, k : A \wedge B \text{ false}} \wedge\text{R}_1 \qquad \frac{\Gamma; \Delta \vdash k : B \text{ false}}{\Gamma; \Delta \vdash \mathsf{snd}\, k : A \wedge B \text{ false}} \wedge\text{R}_2$$

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

Contradictions

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

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

Operational Semantics

$$
\begin{array}{lcl}
\langle \langle e_1, e_2 \rangle \;|_{A \wedge B}\; \mathsf{fst}\, k \rangle & \mapsto & \langle e_1 \;|_{A}\; k \rangle \\
\langle \langle e_1, e_2 \rangle \;|_{A \wedge B}\; \mathsf{snd}\, k \rangle & \mapsto & \langle e_2 \;|_{B}\; k \rangle \\
\langle \mathsf{L}\, e \;|_{A \vee B}\; [k_1, k_2] \rangle & \mapsto & \langle e \;|_{A}\; k_1 \rangle \\
\langle \mathsf{R}\, e \;|_{A \vee B}\; [k_1, k_2] \rangle & \mapsto & \langle e \;|_{B}\; k_2 \rangle \\
\langle \mathsf{not}(k) \;|_{\neg A}\; \mathsf{not}(e) \rangle & \mapsto & \langle e \;|_{A}\; k \rangle \\
\hline
\langle \mu u : A.\, c \;|_{A}\; k \rangle & \mapsto & [k/u]c \\
\langle e \;|_{A}\; \mu x : A.\, c \rangle & \mapsto & [e/x]c
\end{array}
$$

A Bit of Non-Determinism

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

- 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

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

Expressions — Proof Terms

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

(No rule for $\perp$)

$$\frac{}{\Gamma; \Delta \vdash \langle\rangle : \top \text{ true}}\ \top\mathrm{P}$$

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

$$\frac{\Gamma; \Delta \vdash e : A \text{ true}}{\Gamma; \Delta \vdash \mathrm{L}\, e : A \vee B \text{ true}}\ \vee\mathrm{P}_1 \qquad \frac{\Gamma; \Delta \vdash e : B \text{ true}}{\Gamma; \Delta \vdash \mathrm{R}\, e : A \vee B \text{ true}}\ \vee\mathrm{P}_2$$

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

Continuations — Refutation Terms

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

(No rule for $\top$)

$$\frac{}{\Gamma; \Delta \vdash [] : \bot \text{ false}} \bot\mathrm{R}$$

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

$$\frac{\Gamma; \Delta \vdash k : A \text{ false}}{\Gamma; \Delta \vdash \mathsf{fst}\, k : A \wedge B \text{ false}} \wedge\mathrm{R}_1 \qquad \frac{\Gamma; \Delta \vdash k : B \text{ false}}{\Gamma; \Delta \vdash \mathsf{snd}\, k : A \wedge B \text{ false}} \wedge\mathrm{R}_2$$

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

Contradictions

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

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

Operational Semantics

$$
\begin{array}{lcl}
\langle \langle e_1, e_2 \rangle \mid_{A \wedge B} \mathsf{fst}\, k \rangle & \mapsto & \langle e_1 \mid_A k \rangle \\
\langle \langle e_1, e_2 \rangle \mid_{A \wedge B} \mathsf{snd}\, k \rangle & \mapsto & \langle e_2 \mid_B k \rangle \\
\langle \mathsf{L}\, e \mid_{A \vee B} [k_1, k_2] \rangle & \mapsto & \langle e \mid_A k_1 \rangle \\
\langle \mathsf{R}\, e \mid_{A \vee B} [k_1, k_2] \rangle & \mapsto & \langle e \mid_B k_2 \rangle \\
\langle \mathsf{not}(k) \mid_{\neg A} \mathsf{not}(e) \rangle & \mapsto & \langle e \mid_A k \rangle \\
\hline
\langle \mu u : A.\, c \mid_A k \rangle & \mapsto & [k/u]c \\
\langle e \mid_A \mu x : A.\, c \rangle & \mapsto & [e/x]c
\end{array}
$$

Type Safety?

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

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

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

$$\dfrac{\overbrace{\Gamma; \Delta \vdash e_1 : A \text{ true}}^{(3)} \qquad \Gamma; \Delta \vdash e_2 : B \text{ true}}{\Gamma; \Delta \vdash \langle e_1, e_2\rangle : A \wedge B \text{ true}} \wedge\mathrm{P}$$ Analysis of (1)

$$\dfrac{\overbrace{\Gamma; \Delta \vdash k : A \text{ false}}^{(4)}}{\Gamma; \Delta \vdash \mathsf{fst}\, k : A \wedge B \text{ false}} \wedge\mathrm{R}_1$$ Analysis of (2)

$\cdot; \cdot \vdash \langle e_1 \mid_A k\rangle$ contr By rule on (3), (4)

Progress?

Progress? If $\cdot; \cdot \vdash c$ contr then $c \leadsto c'$ (or c final).

Proof:

1. A closed term c is a contradiction
2. Hopefully, there aren't any contradictions!
3. So this theorem is vacuous (assuming classical logic is consistent)

Making Progress Less Vacuous

$$\begin{array}{llcl} \text{Propositions} & A & ::= & \ldots \mid \text{ans} \\ \text{Values} & e & ::= & \ldots \mid \text{halt} \\ \text{Continuations} & k & ::= & \ldots \mid \text{done} \end{array}$$

$$\frac{}{\Gamma; \Delta \vdash \text{halt} : \text{ans true}} \qquad \frac{}{\Gamma; \Delta \vdash \text{done} : \text{ans false}}$$

Progress

Progress If $\cdot;\cdot \vdash c$ contr then $c \rightsquigarrow c'$ or $c = \langle \mathsf{halt} \mid_{\mathsf{ans}} \mathsf{done} \rangle$.

Proof By induction on typing derivations

The Price of Progress

$$\dfrac{\dfrac{\dfrac{\Gamma; \Delta, A \vdash \text{ans true} \qquad \Gamma; \Delta, A \vdash \text{ans false}}{\Gamma; \Delta, A \vdash \text{contr}}}{\Gamma; \Delta \vdash A \ \text{true}} \qquad \dfrac{\dfrac{\dfrac{\Gamma, A; \Delta \vdash \text{ans true} \qquad \Gamma, A; \Delta \vdash \text{ans false}}{\Gamma, A; \Delta \vdash \text{contr}}}{\Gamma; \Delta \vdash A \ \text{false}}}{\Gamma; \Delta \vdash \neg A \ \text{true}}}{\Gamma; \Delta \vdash A \wedge \neg A \ \text{true}}$$

- As a term:

$$\langle \mu u : A.\, \langle \text{halt} \mid \text{done} \rangle, \text{not}(\mu x : A.\, \langle \text{halt} \mid \text{done} \rangle) \rangle$$

- Adding a halt configuration makes classical logic inconsistent – $A \wedge \neg A$ is derivable

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

The Double Negation Translation

- Fix an intuitionistic proposition p
- Define "quasi-negation" $\sim X$ as $X \to p$
- Now, we can define a translation on types as follows:

$$
\begin{array}{lcl}
(\neg A)^\circ & = & \sim A^\circ \\
\top^\circ & = & 1 \\
(A \wedge B)^\circ & = & A^\circ \times B^\circ \\
\bot^\circ & = & p \\
(A \vee B)^\circ & = & \sim\sim(A^\circ + B^\circ)
\end{array}
$$

Triple-Negation Elimination

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

Lemma For all X, there is a function tne : $(\sim\sim\sim X) \to \sim X$

$$
\dfrac{
  \dfrac{
    \dots \qquad
    \dfrac{
      \dfrac{
        \dots \vdash q : X \to p \qquad \dots \vdash x : X
      }{
        k : \sim\sim\sim X, x : X, q : \sim X \vdash q\,x : p
      }
    }{
      k : \sim\sim\sim X, x : X \vdash \lambda q.\, q\,x : \sim\sim X
    }
  }{
    \dfrac{
      k : \sim\sim\sim X, x : X \vdash k\,(\lambda q.\, q\,x) : p
    }{
      k : \sim\sim\sim X \vdash \lambda x.\, k\,(\lambda q.\, q\,a) : \sim X
    }
  }
}{
  \cdot \vdash \underbrace{\lambda k.\, \lambda a.\, k\,(\lambda q.\, q\,a)}_{\text{tne}} : (\sim\sim\sim X) \to \sim X
}
$$

Intuitionistic Double Negation Elimination

Lemma For all A, there is a term dne_A such that

$$\cdot \vdash \mathrm{dne}_A : \sim\sim A^\circ \to A^\circ$$

Proof By induction on A.

$$\begin{array}{lcl}
\mathrm{dne}_\top & = & \lambda q.\, \langle\rangle \\
\mathrm{dne}_{A\wedge B} & = & \lambda q. \left\langle \begin{array}{l} \mathrm{dne}_A\,(\lambda k.\, q\,(\lambda p.\, k\,(\mathrm{fst}\, p))), \\ \mathrm{dne}_B\,(\lambda k.\, q\,(\lambda p.\, k\,(\mathrm{snd}\, p))) \end{array} \right\rangle \\
\mathrm{dne}_\bot & = & \lambda q.\, q\,(\lambda x.\, x) \\
\mathrm{dne}_{A\vee B} & = & \lambda q : \sim\sim \underbrace{\sim\sim(A^\circ \vee B^\circ)}_{(A\vee B)^\circ}.\, \mathrm{tne}\, q \\
\mathrm{dne}_{\neg A} & = & \lambda q : \sim\sim \underbrace{(\sim A^\circ)}_{(\neg A)^\circ}.\, \mathrm{tne}\, q
\end{array}$$

Double Negation Elimination for $\bot$

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{q : (p \to p) \to p \vdash q : (p \to p) \to p} \qquad \dfrac{\dfrac{}{q : (p \to p) \to p, x : p \vdash x : p}}{q : (p \to p) \to p \vdash \lambda x : p.\, x : p}}{q : (p \to p) \to p \vdash q\,(\lambda x : p.\, x) : p}}{\cdot \vdash \lambda q : (p \to p) \to p.\, q\,(\lambda x : p.\, x) : ((p \to p) \to p) \to p}}{\dfrac{\cdot \vdash \lambda q : \sim\sim p.\, q\,(\lambda x : p.\, x) : \sim\sim p \to p}{\cdot \vdash \lambda q : \sim\sim\bot^\circ.\, q\,(\lambda x : p.\, x) : \sim\sim\bot^\circ \to \bot^\circ}}
$$

Translating Derivations

Theorem Classical terms embed into intutionistic terms:

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

Proof By induction on derivations – but first, we have to define the translation!

Translating Contexts

Translating Value Contexts:

$$\begin{array}{lcl} (\cdot)^\circ & = & \cdot \\ (\Gamma, x : A)^\circ & = & \Gamma^\circ, x : A^\circ \end{array}$$

Translating Continuation Contexts:

$$\begin{array}{lcl} {\sim}(\cdot) & = & \cdot \\ {\sim}(\Gamma, x : A) & = & {\sim}\Gamma, x : {\sim}A^\circ \end{array}$$

Translating Contradictions

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

Define:

$$\langle e \mid_A k \rangle^\circ \quad = \quad k^\circ\, e^\circ$$

Translating (Most) Expressions

$$\begin{aligned}
x^\circ &= x \\
\langle\rangle^\circ &= \langle\rangle \\
\langle e_1, e_2\rangle^\circ &= \langle {e_1}^\circ, {e_2}^\circ\rangle \\
(\mathsf{L}\, e)^\circ &= \lambda k : {\sim}(A^\circ + B^\circ).\, k\,(\mathsf{L}\, e^\circ) \\
(\mathsf{R}\, e)^\circ &= \lambda k : {\sim}(A^\circ + B^\circ).\, k\,(\mathsf{R}\, e^\circ) \\
(\mathrm{not}(k))^\circ &= k^\circ
\end{aligned}$$

Translating (Most) Continuations

$$
\begin{array}{lcl}
x^\circ & = & x \\
[]^\circ & = & \lambda x : p.\, x \\
[k_1, k_2]^\circ & = & \lambda k : \sim\sim(A^\circ + B^\circ). \\
& & \quad k\ (\lambda i : A^\circ + B^\circ. \\
& & \qquad\quad \mathsf{case}(i, \mathsf{L}\, x \to k_1^\circ\, x, \mathsf{R}\, y \to k_2^\circ\, y)) \\
(\mathsf{fst}\, k)^\circ & = & \lambda p : (A^\circ \times B^\circ).\, k^\circ\, (\mathsf{fst}\, p) \\
(\mathsf{snd}\, k)^\circ & = & \lambda p : (A^\circ \times B^\circ).\, k^\circ\, (\mathsf{snd}\, p) \\
(\mathsf{not}(e))^\circ & = & \lambda k : \underbrace{\sim A^\circ}_{(\neg A)^\circ}.\, k\, e^\circ
\end{array}
$$

Translating Refutation by Contradiction

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

1. We assume $(\Gamma, x : A)^\circ, {\sim}\Delta \vdash c^\circ : p$
2. So $\Gamma^\circ, x : A^\circ, {\sim}\Delta \vdash c^\circ : p$
3. So $\Gamma^\circ, {\sim}\Delta \vdash \lambda x : A^\circ.\, c^\circ : A^\circ \to p$
4. So $\Gamma^\circ, {\sim}\Delta \vdash \lambda x : A^\circ.\, c^\circ : {\sim}A^\circ$

So we define

$$(\mu x : A.\, c)^\circ = \lambda x : A^\circ.\, c^\circ$$

Translating Proof by Contradiction

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

1	$\Gamma^\circ, \sim(\Delta, u : A) \vdash c^\circ : p$	Assumption
2	$\Gamma^\circ, \sim\Delta, u : \sim A^\circ \vdash c^\circ : p$	Def. of $\sim$ on contexts
3	$\Gamma^\circ, \sim\Delta \vdash \lambda u : \sim A^\circ.\, c^\circ : \sim A^\circ \to p$	$\to$I
4	$\Gamma^\circ, \sim\Delta \vdash \lambda u : \sim A^\circ.\, c^\circ : \sim\sim A^\circ$	Def. of $\sim$ on types
5	$\Gamma^\circ, \sim\Delta \vdash \mathrm{dne}_A(\lambda u : u : \sim A^\circ.\, c^\circ) : A^\circ$	$\to$E

So we define

$$(\mu u : A.\, c)^\circ = \mathrm{dne}_A(\lambda u : \sim A^\circ.\, c^\circ)$$

Consequences

- We now have a proof that every classical proof has a corresponding intuitionistic proof
- So classical logic is a *subsystem* of intuitionistic logic
- Because intuitionistic logic is consistent, so is classical logic
- Classical logic can inherit operational semantics from intuitionistic logic!

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 $\sim\sim A^\circ \to A^\circ$ holds.

The Gödel-Gentzen Translation

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

$$
\begin{array}{lcl}
\neg A^\circ & = & {\sim}A^\circ \\
\top^\circ & = & 1 \\
(A \wedge B)^\circ & = & A^\circ \times B^\circ \\
\bot^\circ & = & p \\
(A \vee B)^\circ & = & {\sim}({\sim}A^\circ \times {\sim}B^\circ)
\end{array}
$$

- This uses a different de Morgan duality for disjunction

The Kolmogorov Translation

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

$$\begin{array}{lcl} \neg A^\bullet & = & \sim\sim\sim A^\bullet \\ A \supset B^\bullet & = & \sim\sim(A^\bullet \to B^\bullet) \\ \top^\bullet & = & \sim\sim 1 \\ (A \wedge B)^\bullet & = & \sim\sim(A^\bullet \times B^\bullet) \\ \bot^\bullet & = & \sim\sim\bot \\ (A \vee B)^\bullet & = & \sim\sim(A^\bullet + B^\bullet) \end{array}$$

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

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

$$
\begin{array}{llll}
\text{Types} & X & ::= & 1 \mid X \times Y \mid 0 \mid X + Y \mid X \to Y \mid \neg X \\
\text{Terms} & e & ::= & x \mid \langle\rangle \mid \langle e, e\rangle \mid \mathsf{fst}\, e \mid \mathsf{snd}\, e \\
 & & \mid & \mathsf{abort} \mid \mathsf{L}\, e \mid \mathsf{R}\, e \mid \mathsf{case}(e, \mathsf{L}\, x \to e', \mathsf{R}\, y \to e'') \\
 & & \mid & \lambda x : X.\, e \mid e\, e' \\
 & & \mid & \mathsf{throw}(e, e') \mid \mathsf{letcont}\, x.\, e \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : X
\end{array}
$$

Units and Pairs

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

$$\frac{\Gamma \vdash e : X \qquad \Gamma \vdash e' : Y}{\Gamma \vdash \left\langle e, e' \right\rangle : X \times Y}\ \times\mathrm{I}$$

$$\frac{\Gamma \vdash e : X \times Y}{\Gamma \vdash \mathsf{fst}\, e : X}\ \times\mathrm{E}_1 \qquad\qquad \frac{\Gamma \vdash e : X \times Y}{\Gamma \vdash \mathsf{snd}\, e : Y}\ \times\mathrm{E}_1$$

Functions and Variables

$$\frac{X : X \in \Gamma}{\Gamma \vdash x : X}\ \textsc{Hyp} \qquad\qquad \frac{\Gamma, x : X \vdash e : Y}{\Gamma \vdash \lambda x : X.\, e : X \to Y}\ {\to}\mathrm{I}$$

$$\frac{\Gamma \vdash e : X \to Y \qquad \Gamma \vdash e' : X}{\Gamma \vdash e\, e' : Y}\ {\to}\mathrm{E}$$

Sums and the Empty Type

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash \mathsf{L}\, e : X + Y} +\mathsf{I}_1 \qquad \frac{\Gamma \vdash e : Y}{\Gamma \vdash \mathsf{R}\, e : X + Y} +\mathsf{I}_2$$

$$\frac{\Gamma \vdash e : X + Y \qquad \Gamma, x : X \vdash e' : Z \qquad \Gamma, y : Y \vdash e'' : Z}{\Gamma \vdash \mathsf{case}(e, \mathsf{L}\, x \to e', \mathsf{R}\, y \to e'') : Z} +\mathsf{E}$$

(no intro for 0)

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

Continuation Typing

$$\frac{\Gamma, u : \neg X \vdash e : X}{\Gamma \vdash \mathrm{letcont}\, u : \neg X.\ e : X}\ \textsc{Cont} \qquad \frac{\Gamma \vdash e : \neg X \qquad \Gamma \vdash e' : X}{\Gamma \vdash \mathrm{throw}_Y(e, e') : Y}\ \textsc{Throw}$$

Examples

Double-negation elimination:

$$\begin{array}{rcl} \mathsf{dne}_X & : & \neg\neg X \to X \\ \mathsf{dne}_X & \triangleq & \lambda k : \neg\neg X.\ \mathsf{letcont}\, u : \neg X.\ \mathsf{throw}(k, u) \end{array}$$

The Excluded Middle:

$$\begin{array}{l} t : X \vee \neg X \\ t \triangleq \mathsf{letcont}\, u : \neg(X \vee \neg X). \\ \qquad \mathsf{throw}(u, \mathsf{R}\,(\mathsf{letcont}\, q : \neg\neg X. \\ \qquad\qquad\qquad \mathsf{throw}(u, \mathsf{L}\,(\mathsf{dne}_X\, q))) \end{array}$$

Continuation-Passing Style (CPS) Translation

Type translation:

$$
\begin{array}{lcl}
\neg X^\bullet & = & \sim\sim\sim X^\bullet \\
X \to Y^\bullet & = & \sim\sim(X^\bullet \to Y^\bullet) \\
1^\bullet & = & \sim\sim 1 \\
(X \times Y)^\bullet & = & \sim\sim(X^\bullet \times Y^\bullet) \\
0^\bullet & = & \sim\sim 0 \\
(X + Y)^\bullet & = & \sim\sim(X^\bullet + Y^\bullet)
\end{array}
$$

Translating contexts:

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

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

$$
\begin{array}{lcl}
x^{\bullet} & = & \lambda k.\, x\, k \\
\langle\rangle^{\bullet} & = & \lambda k.\, k\ \langle\rangle \\
\langle e_1, e_2\rangle^{\bullet} & = & \lambda k.\, e_1^{\bullet}\, (\lambda x.\, e_2{}^{\bullet}\, (\lambda y.\, k\, (x, y))) \\
(\mathsf{fst}\, e)^{\bullet} & = & \lambda k.\, e^{\bullet}\, (\lambda p.\, k\, (\mathsf{fst}\, p)) \\
(\mathsf{snd}\, e)^{\bullet} & = & \lambda k.\, e^{\bullet}\, (\lambda p.\, k\, (\mathsf{snd}\, p)) \\
(\mathsf{L}\, e)^{\bullet} & = & \lambda k.\, e^{\bullet}\, (\lambda x.\, k\, (\mathsf{L}\, x)) \\
(\mathsf{R}\, e)^{\bullet} & = & \lambda k.\, e^{\bullet}\, (\lambda y.\, k\, (\mathsf{R}\, y)) \\
\mathsf{case}(e, \mathsf{L}\, x \to e_1, \mathsf{R}\, y \to e_2)^{\bullet} & = & \lambda k.\, e^{\bullet}\, (\lambda v.\, \mathsf{case}(v, \\
 & & \qquad\qquad\quad \mathsf{L}\, x \to e_1^{\bullet}\, k \\
 & & \qquad\qquad\quad \mathsf{R}\, y \to e_2^{\bullet}\, k) \\
(\lambda x : X.\, e)^{\bullet} & = & \lambda k.\, k\, (\lambda x : X^{\bullet}.\, e^{\bullet}) \\
(e_1\, e_2)^{\bullet} & = & \lambda k.\, e_1^{\bullet}\, (\lambda f.\, e_2^{\bullet}\, (\lambda x.\ k\, (f\, x)))
\end{array}
$$

The CPS Translation for Continuations

$$
\begin{aligned}
(\mathrm{letcont}\, u : \neg X.\ e)^{\bullet} &= \lambda k.\, [(\lambda q.\, q\, k)/u](e^{\bullet}) \\
\mathrm{throw}(e_1, e_2)^{\bullet} &= \lambda k.\, \mathrm{tne}(e_1^{\bullet})\, e_2^{\bullet}
\end{aligned}
$$

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

1. Give the embedding (ie, the e° and k° translations) of classical into intuitionistic logic for the Gödel-Gentzen translation. You just need to give the embeddings for sums, since that is the only case different from lecture.
2. Using the intuitionistic calculus extended with continuations, give a typed term proving *Peirce's law*:

$$((X \to Y) \to X) \to X$$

Type Systems

Lecture 11: Applications of Continuations, and Dependent Types

Neel Krishnaswami
University of Cambridge

Applications of Continuations

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

$$
\begin{array}{llcl}
\text{Types} & X & ::= & 1 \mid X \times Y \mid 0 \mid X + Y \mid X \to Y \mid \neg X \\
\text{Terms} & e & ::= & x \mid \langle\rangle \mid \langle e, e \rangle \mid \mathsf{fst}\, e \mid \mathsf{snd}\, e \\
 & & \mid & \mathsf{abort} \mid \mathsf{L}\, e \mid \mathsf{R}\, e \mid \mathsf{case}(e, \mathsf{L}\, x \to e', \mathsf{R}\, y \to e'') \\
 & & \mid & \lambda x : X.\, e \mid e\, e' \\
 & & \mid & \mathsf{throw}(e, e') \mid \mathsf{letcont}\, x.\, e \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : X
\end{array}
$$

Continuation Typing

$$\frac{\Gamma, u : \neg X \vdash e : X}{\Gamma \vdash \mathrm{letcont}\, u : \neg X.\ e : X}\ \textsc{Cont} \qquad \frac{\Gamma \vdash e : \neg X \qquad \Gamma \vdash e' : X}{\Gamma \vdash \mathrm{throw}_Y(e, e') : Y}\ \textsc{Throw}$$

Continuation API in Standard ML

```
signature CONT = sig
  type 'a cont
  val callcc : ('a cont -> 'a) -> 'a
  val throw : 'a cont -> 'a -> 'b
end
```

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

An Inefficient Program

```
val mul : int list -> int

fun mul []          = 1
  | 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

```
val mul' : int list -> int

fun mul' []           = 1
  | mul' (0 :: ns) = 0
  | 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

```
val loop = fn : int cont -> int list -> int
fun loop return []          = 1
  | loop return (0 :: ns) = throw return 0
  | loop return (n :: ns) = n * loop return ns

val mul_fast : int list -> int
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*

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

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

The AMB signature

```
signature AMB = sig
  (* Internal implementation *)
  val stack : int option cont list ref
  val fail : unit -> 'a

  (* External API *)
  exception AmbFail
  val assert : bool -> unit
  val amb : int list -> int
end
```

Implementation, Part 1

```
exception AmbFail
val stack
  : int option cont list ref
  = ref []

fun fail () =
  case !stack of
    []         => raise AmbFail
  | (k :: ks) => (stack := ks;
                  throw k NONE)

fun assert b =
  if b then () else fail()^^I^^I^^I
```

- `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

```
fun amb []          = fail ()
  | amb (x :: xs) =
    let fun next y k =
          (stack := k :: !stack;
           SOME y)
    in
        case callcc (next x) of
            SOME v => v
          | NONE => amb xs
    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

```
fun test2() =
    let val x = amb [1,2,3,4,5,6]
        val y = amb [1,2,3,4,5,6]
        val z = amb [1,2,3,4,5,6]
    in
        assert(x + y + z >= 13);
        assert(x > 1);
        assert(y > 1);
        assert(z > 1);
        (x, y, z)
    end

(* 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, $\forall x, y, z, n \in \mathbb{N}$. if $n > 2$ then $x^n + y^n \neq z^n$

A Logical Curiosity

$$\frac{}{\Gamma \vdash \mathsf{z} : \mathbb{N}}\ \mathbb{N}\mathsf{I}_z \qquad\qquad \frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \mathsf{s}(e) : \mathbb{N}}\ \mathbb{N}\mathsf{I}_s$$

$$\frac{\Gamma \vdash e_0 : \mathbb{N} \qquad \Gamma \vdash e_1 : X \qquad \Gamma, x : X \vdash e_2 : X}{\Gamma \vdash \mathsf{iter}(e_0, \mathsf{z} \to e_1, \mathsf{s}(x) \to e_2) : X}\ \mathbb{N}\mathsf{E}$$

- $\mathbb{N}$ is the type of natural numbers
- Logically, it is equivalent to the unit type:
  - $(\lambda x : 1.\, \mathsf{z}) : 1 \to \mathbb{N}$
  - $(\lambda x : \mathbb{N}.\, \langle\rangle) : \mathbb{N} \to 1$
- Language of types has no way of distinguishing z from $\mathsf{s}(z)$.

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

```
data Bool : Set where
  true : Bool
  false : Bool

not : Bool → Bool
not true = false
not false = true
```

- Datatype declarations give constructors and their types
- Functions given type signature, and clausal definition

Agda: Inductive Datatypes

```
data Nat : Set where
  z : Nat
  s : Nat → Nat

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

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

- Datatype constructors can be recursive
- Functions can be recursive, but checked for termination

Agda: Polymorphic Datatypes

```
data List (A : Set) : Set where
  [] : List A
  _,_ : A → List A → List A

app : (A : Set) → List A → List A → List A
app A [] ys = ys
app A (x , xs) ys = x , app A xs ys

app' : {A : Set} → List A → List A → List A
app' [] ys = ys
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

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

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

head : {A : Set} → {n : Nat} → Vec A (s n) → A
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

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

app : {A : Set} → {n m : Nat} →
      Vec A n → Vec A m → Vec A (n + m)
app [] ys = ys
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

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

-- Won't typecheck!
app : {A : Set} → {n m : Nat} →
      Vec A n → Vec A m → Vec A (n + m)
app [] ys = ys
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 ≡ 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

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

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

Judgements of Dependent Type Theory

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

The Unit Type

Type Formation

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

Introduction

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

(No Elimination)

Function Types

Type Formation

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

Introduction

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

Elimination

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

Equality Types

Type Formation

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

Introduction

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

Elimination

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

(Equality elimination not the most general form!)

Variables and Equality

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

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

What Is Judgmental Equality For?

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

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

Example of Judgemental Equality

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

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

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

Example

Suppose we have:

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

zs : Vec A (s (s (s (s z))))
zs = append xs ys
```

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

Judgmental Type Equality

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

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

Judgmental Term Equality: Equivalence Relation

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

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

Judgmental Term Equality: Congruence Rules

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

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

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

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

Judgemental Equality: Conversion rules

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

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

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

Context Well-formedness

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

Metatheory: Weakening

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

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

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

Metatheory: Substitution

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

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

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

Metatheory: Context Equality

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

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

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

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